华为认证 GaussDB 系列教程

HCIP-GaussDB-OLAP

GaussDB OLAP数据库高级工程师

实验指导手册

版本:1.5

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： http://e.huawei.com

华为认证体系介绍

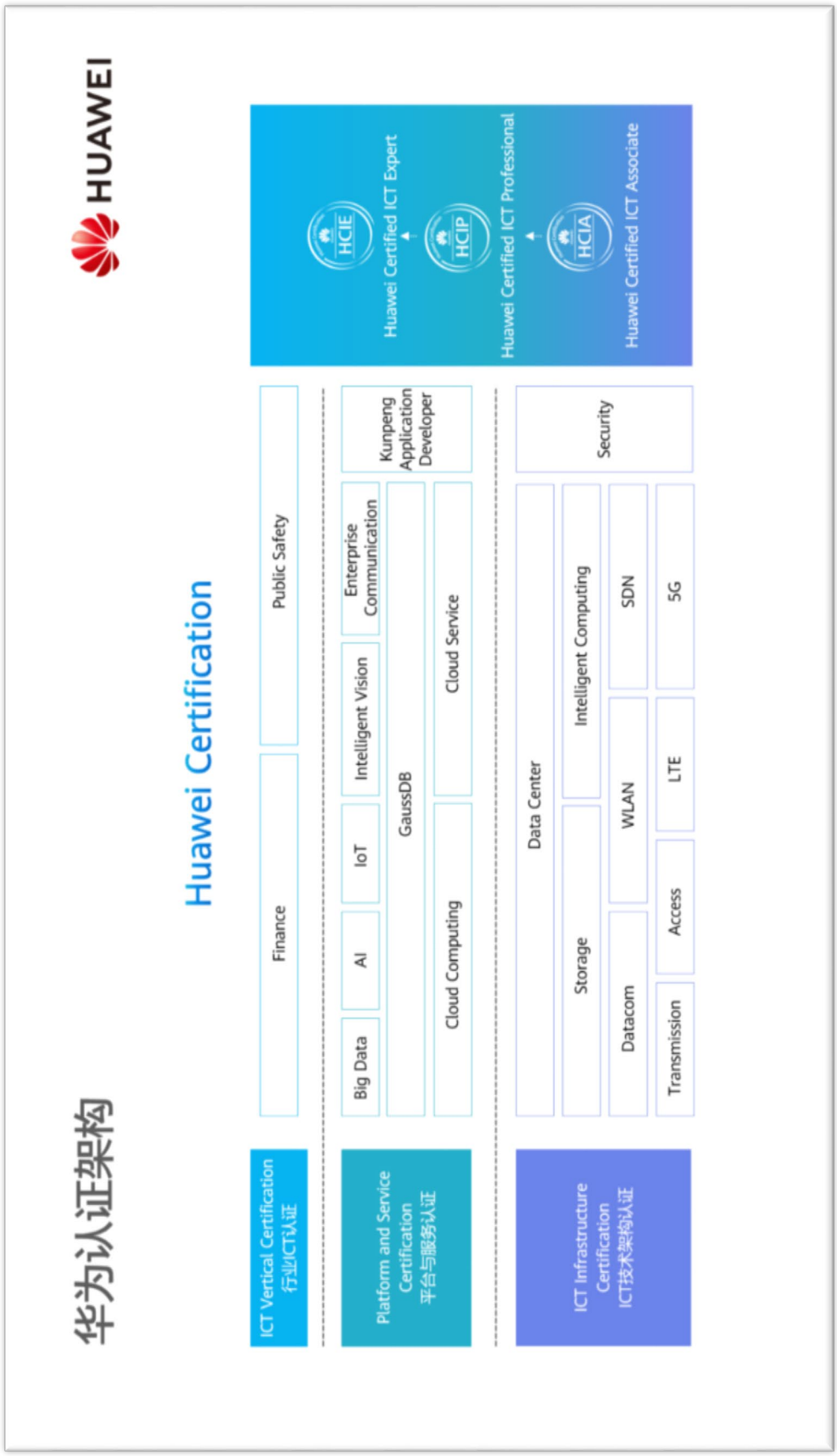
基于“平台+生态”战略，围绕“云-管-端”协同的新ICT技术架构，华为公司打造了覆盖ICT领域的认证体系，包含ICT技术架构认证、平台与服务认证和行业ICT认证三类认证。

根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

华为认证覆盖ICT全领域，符合ICT融合的技术趋势，致力于提供领先的人才培养体系和认证标准，培养数字化时代的新型ICT人才，构建良性的ICT人才生态。

HCIP-GaussDB-OLAP V1.5定位于GaussDB OLAP数据库的高级工程师级别认证，适用于DWS数据仓库二次开发工程师以及数据仓库管理员的技能提升。该认证提供了丰富的随堂实验和行业案例，旨在提升学员的实践能力，促进数据库人才的培养。

华为认证协助您打开行业之窗，开启改变之门，屹立在数据库世界的潮头浪尖！


HUAWEI
华为认证架构
Huawei Certification
Huawei Certified ICT Expert
Huawei Certified ICT Professional
Huawei Certified ICT Associate
HCIE
HCIP
HCIA
ICT Vertical Certification
行业ICT认证
Finance
Public Safety
Platform and Service Certification
平台与服务认证
Big Data
AI
IoT
Intelligent Vision
Enterprise Communication
Kunpeng Application Developer
GaussDB
Cloud Computing
Cloud Service
ICT Infrastructure Certification
ICT技术架构认证
Data Center
Security
Storage
Intelligent Computing
Datacom
WLAN
SDN
Transmission
Access
LTE
5G

前 言

简介

本书为 HCIP-GaussDB-OLAP V1.5 认证培训教程，适用于准备参加 HCIP-GaussDB-OLAP V1.5 考试的学员或希望了解数据仓库的基础知识、SQL 高阶语法、数据库设计与管理、数据库安全管理、数据库集群管理等相关 GaussDB OLAP 数据库技术的读者。本实验以华为公有云 GaussDB(DWS)服务为平台展开。

内容描述

本实验指导书书共包含 7 个随堂实验和 1 个综合实验。

- 实验一为 SQL 高级语法的实验，主要包括通过 DS 客户端方式导入数据，不同数据类型实验，常用函数和操作联系等。
- 实验二为 Explain 工具介绍，主要通过模拟各种类型语句，以不同方式查看其执行计划，并对执行计划进行解析。
- 实验三为数据库对象的设计与管理，主要包括数据库、表、索引、视图、序列和 SCHEMA 等不同对象的创建和管理。
- 实验四为数据分区实验，主要练习创建和管理分区表，及对各个分区的各种常规操作练习。
- 实验五介绍了数据库事务的管理，主要从锁的应用、事务一致性、多 session 测试读写隔离和子事务四个方面，对事务进行讲解和练习。
- 实验六为数据库安全，主要讲述了三权分立、RBAC 关系模型和对象权限的操作等内容。
- 实验七为集群管理，通过控制台界面对数据库集群进行管理，包括查询集群状态，重启集群，设置集群安全配置及设置集群容量等内容。
- 综合实验为实际场景下，把一些常规的操作全部贯穿起来，提供更全面的练习。

目录

1 SQL 高级语法

1.1 实验介绍

1.1.1 关于本实验

本实验主要了解 GaussDB(DWS)语法，包含基本的数据类型、函数及操作符。

1.1.2 实验目的

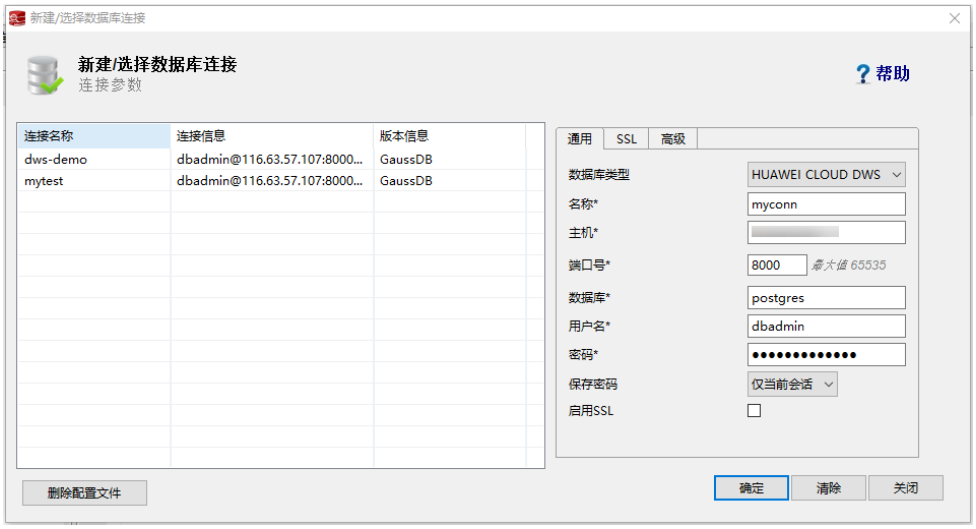
- 理解各种数据类型的基本用法。
- 理解各种函数及操作符的基本用法。

1.2 实验步骤

1.2.1 环境准备

连接 GaussDB(DWS)：可以使用 Data Studio 进行连接，也可以使用 gsql 进行连接。除数据导入练习外，本小节其他内容主要使用 Data Studio 完成。

步骤 1 使用 Data Studio 连接。



数据库类型：选择“HUAWEI CLOUD DWS”

名称：连接名称，例如 myconn

主机：GaussDB(DWS)集群的公网 IP

端口号：8000（根据实际集群修改）

数据库：postgres

用户名：dbadmin

密码：在创建集群时设置的密码

SSL 选项：去掉勾选“SSL 选项”，然后单击“确定”，在弹出对话框中单击“继续”。

连接成功。

步骤 2 连接后创建新的数据库。

数据库命名建议为“个人姓名简称_demo”，例如 hql_demo。

```
create database hql_demo;
```

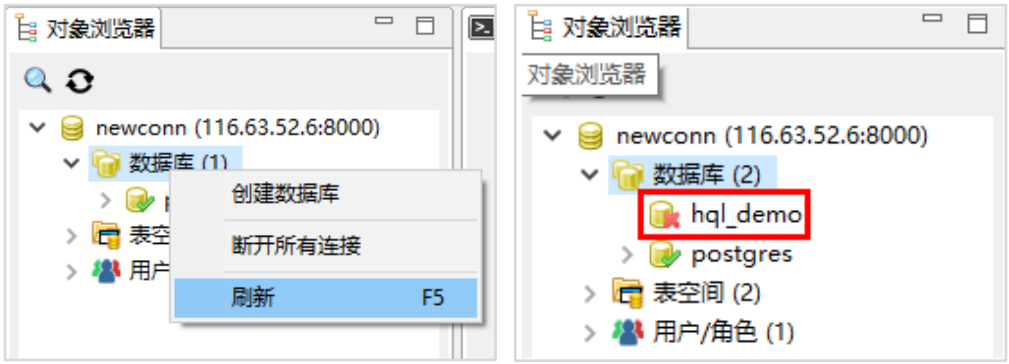
执行结果：

[INFO] 操作影响的记录行数 ：0

[INFO] 执行时间 ：926 ms

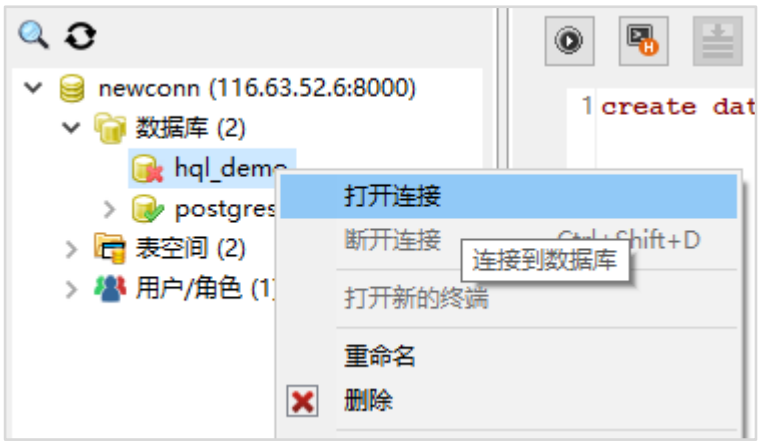
[INFO] 执行成功...

鼠标右键数据库，刷新数据库。



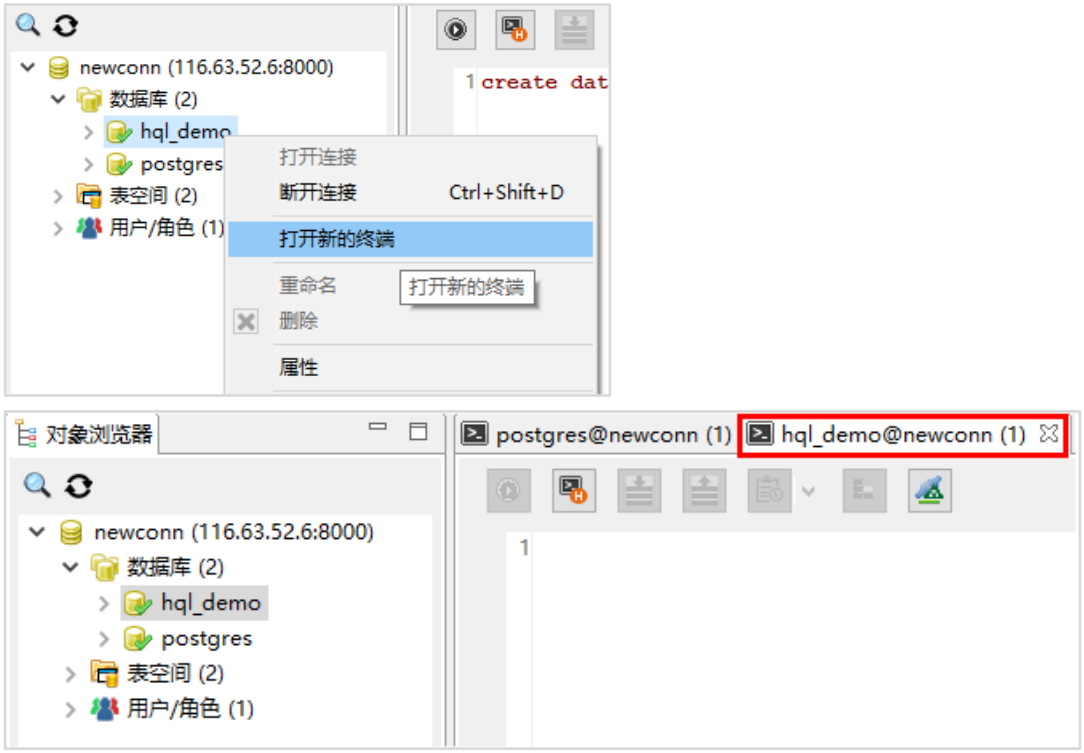
步骤 3 建立连接。

双击数据库 hql_demo 或者右键“打开连接”，建立连接。



步骤 4 打开新的终端。

鼠标右键 hql_demo 数据库，打开新的终端。



此小节所有的操作均在新创建的数据库下操作。

1.2.2 数据导入练习

准备好数据文件：date_dim.csv

另外再次说明，无论使用哪个客户端进行实验，请先创建个人数据库，例如：

CREATE DATABASE hql_demo;

本小节后续实验均在该数据库中完成。

1.2.2.1 使用 Data Studio 完成数据导入

步骤 1 运行 Data Studio

在 Data Studio 上“右键”，选择“以管理员身份运行”。在连接界面填入 GaussDB(DWS) 相关参数。

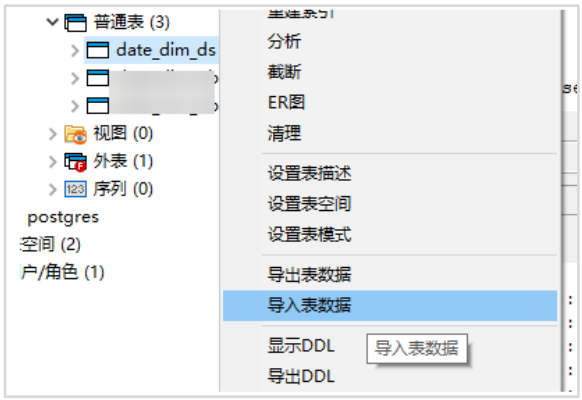
步骤 2 创建表 date_dim_ds

```
DROP TABLE IF EXISTS date_dim_ds;
CREATE TABLE date_dim_ds
(
    d_date_sk                 integer                  not null,
    d_date_id                 char(16)                 not null,
    d_date                    date                              ,
```

```
    d_month_seq               integer                       ,
    d_week_seq                integer                       ,
    d_quarter_seq             integer                       ,
    d_year                    integer                       ,
    d_dow                     integer                       ,
    d_moy                     integer                       ,
    d_dom                     integer                       ,
    d_qoy                     integer                       ,
    d_fy_year                 integer                       ,
    d_fy_quarter_seq          integer                       ,
    d_fy_week_seq             integer                       ,
    d_day_name                char(9)                       ,
    d_quarter_name            char(6)                       ,
    d_holiday                 char(1)                       ,
    d_weekend                 char(1)                       ,
    d_following_holiday       char(1)                       ,
    d_first_dom               integer                       ,
    d_last_dom                integer                       ,
    d_same_day_ly             integer                       ,
    d_same_day_lq             integer                       ,
    d_current_day             char(1)                       ,
    d_current_week            char(1)                       ,
    d_current_month           char(1)                       ,
    d_current_quarter         char(1)                       ,
    d_current_year            char(1)
);
```

步骤 3　导入数据

在新建的表 date_dim_ds 上右键，选择“导入表数据”。

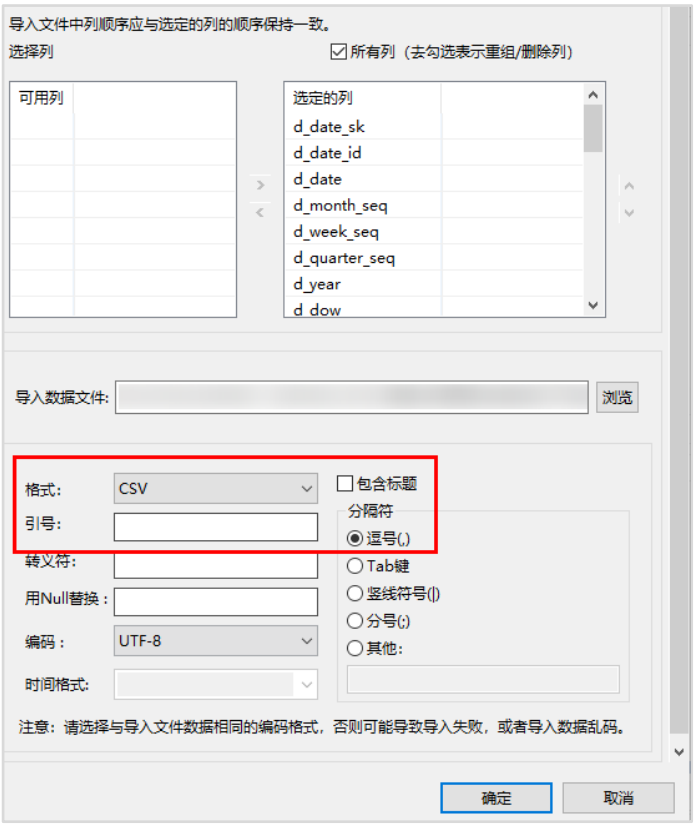


在导入界面点击“浏览”，找到 date_dim.csv 文件。

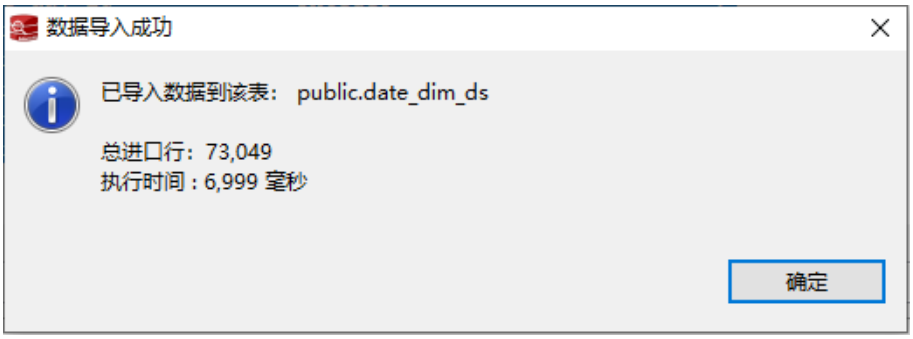
格式：CSV

包含标题：去掉对勾

分隔符：逗号



可以看到，成功导入 73049 条数据，用时为 6999 毫秒。



1.2.2.2 使用 gsql 元命令完成导入

步骤 1 登陆 ECS 并创建数据目录

以 root 用户登陆安装了 gsql 的 ECS，创建存放源数据的目录。

```
mkdir -p /opt/gds_input_data
```

将数据文件 date_dim.csv 通过 WinSCP 上传至该数据目录下。

在数据文件目录下确认数据文件已存在。

```
cd /opt/gds_input_data
ls
```

```
[root@ecs-tools-hql input_data]# ls
date_dim.csv
```

步骤 2 连接数据库

进入 gsql 安装目录，启动客户端并连接个人数据库，例如：

```
cd /opt
source gsql_env.sh
gsql -d hql_demo -h 114.116.200.18 -p 8000 -U dbadmin -W Dws@123456 -r
```

说明：请替换为个人的数据库名，集群公网 IP，管理员用户密码

步骤 3 创建表 date_dim_gsql

```
DROP TABLE IF EXISTS date_dim_gsql;
CREATE TABLE date_dim_gsql
(
    d_date_sk                 integer                  not null,
    d_date_id                 char(16)                 not null,
    d_date                    date                             ,
    d_month_seq               integer                          ,
    d_week_seq                integer                          ,
    d_quarter_seq             integer                          ,
    d_year                    integer                          ,
    d_dow                     integer                          ,
    d_moy                     integer                          ,
    d_dom                     integer                          ,
    d_qoy                     integer                          ,
    d_fy_year                 integer                          ,
    d_fy_quarter_seq          integer                          ,
    d_fy_week_seq             integer                          ,
    d_day_name                char(9)                          ,
    d_quarter_name            char(6)                          ,
    d_holiday                 char(1)                          ,
    d_weekend                 char(1)                          ,
    d_following_holiday       char(1)                          ,
    d_first_dom               integer                          ,
    d_last_dom                integer                          ,
    d_same_day_ly             integer                          ,
    d_same_day_lq             integer                          ,
    d_current_day             char(1)                          ,
    d_current_week            char(1)                          ,
    d_current_month           char(1)                          ,
    d_current_quarter         char(1)                          ,
    d_current_year            char(1)
);
```

返回结果：

```
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'd_date_sk' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
```

步骤 4 导入数据

输入：

```
\timing on
```

返回结果：

```
Timing is on.
```

输入：

```
\copy date_dim_gsql FROM '/opt/gds_input_data/date_dim.csv' WITH( FORMAT 'csv', DELIMITER ',', ignore_extra_data 'true', ENCODING 'utf8');
```

返回结果：

```
Time: 12073.206 ms
```

步骤 5 查看导入数据量

输入：

```
SELECT COUNT(*) FROM date_dim_gsql;
```

返回结果：

```
WARNING:  Statistics in some tables or columns(public.date_dim_gsql.d_date_sk) are not collected.
HINT:  Do analyze for them in order to generate optimized plan.
 count
-------
 73049
(1 row)

Time: 12.956 ms
```

退出数据库

```
\q
```

由此可以看到，使用\copy 来完成数据导入的效率相对来说是比较低下的。

1.2.2.3 使用 GDS 完成导入

此处默认已经配置好 GDS 工具，若没有，请参考前面实验内容。

此小节假定 GDS 专有用户为 gds_user，数据目录为/opt/gds_input_data，GDS 与 gsql 部署在同一台 ECS 上。

利用 WinSCP 将 date_dim.csv 数据文件传至/opt/gds_input_data 下。

步骤 1 切换到 gds_user 用户

```
su - gds_user
```

步骤 2 启动 GDS 服务

```
/opt/gds/gds -d /opt/gds_input_data/ -p 192.168.0.125:5000 -H 0.0.0.0/0 -l /opt/gds/gds_log.txt -D -t 2
```

步骤 3 创建目标表 date_dim_gds

切换为 root 用户：

```
su - root
```

连接数据库

```
cd /opt
source gsql_env.sh
gsql -d hql_demo -h 114.116.200.18 -p 8000 -U dbadmin -W Dws@123456 –r
```

创建目标表

```
DROP TABLE IF EXISTS date_dim_gds;
CREATE TABLE date_dim_gds
(
    d_date_sk                 integer                 not null,
    d_date_id                 char(16)                not null,
    d_date                    date                    ,
    d_month_seq               integer                 ,
    d_week_seq                integer                 ,
    d_quarter_seq             integer                 ,
    d_year                    integer                 ,
    d_dow                     integer                 ,
    d_moy                     integer                 ,
    d_dom                     integer                 ,
    d_qoy                     integer                 ,
    d_fy_year                 integer                 ,
    d_fy_quarter_seq          integer                 ,
    d_fy_week_seq             integer                 ,
    d_day_name                char(9)                 ,
    d_quarter_name            char(6)                 ,
    d_holiday                 char(1)                 ,
    d_weekend                 char(1)                 ,
    d_following_holiday       char(1)                 ,
    d_first_dom               integer                 ,
    d_last_dom                integer                 ,
    d_same_day_ly             integer                 ,
    d_same_day_lq             integer                 ,
    d_current_day             char(1)                 ,
    d_current_week            char(1)                 ,
    d_current_month           char(1)                 ,
    d_current_quarter         char(1)                 ,
    d_current_year            char(1)
);
```

返回结果：

```
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'd_date_sk' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
Time: 26.875 ms
```

步骤 4 创建外表

```
DROP FOREIGN TABLE IF EXISTS foreign_date_dim_gds;
```

```
CREATE FOREIGN TABLE foreign_date_dim_gds
(
    d_date_sk                integer                 not null,
    d_date_id                char(16)                not null,
    d_date                   date                    ,
    d_month_seq              integer                 ,
    d_week_seq               integer                 ,
    d_quarter_seq            integer                 ,
    d_year                   integer                 ,
    d_dow                    integer                 ,
    d_moy                    integer                 ,
    d_dom                    integer                 ,
    d_qoy                    integer                 ,
    d_fy_year                integer                 ,
    d_fy_quarter_seq         integer                 ,
    d_fy_week_seq            integer                 ,
    d_day_name               char(9)                 ,
    d_quarter_name           char(6)                 ,
    d_holiday                char(1)                 ,
    d_weekend                char(1)                 ,
    d_following_holiday      char(1)                 ,
    d_first_dom              integer                 ,
    d_last_dom               integer                 ,
    d_same_day_ly            integer                 ,
    d_same_day_lq            integer                 ,
    d_current_day            char(1)                 ,
    d_current_week           char(1)                 ,
    d_current_month          char(1)                 ,
    d_current_quarter        char(1)                 ,
    d_current_year           char(1)
)
SERVER gsmpp_server
OPTIONS(location 'gsfs://192.168.0.125:5000/date_dim.csv',
FORMAT 'CSV' ,
DELIMITER ',',
ENCODING 'utf8',
MODE 'Normal',
ignore_extra_data 'true'
)
WITH gds_date_dim_err;
```

步骤 5 通过 GDS 外表将数据导入目标表中

```
INSERT INTO date_dim_gds SELECT * FROM foreign_date_dim_gds;
```

返回结果：

```
INSERT 0 73049
Time: 230.354 ms
```

由此可以看到，使用 GDS 完成 73049 条数据导入花费 230.354ms。GDS 的导入效率较高。

步骤 6 处理错误表

SELECT * FROM gds_date_dim_err;

返回结果：

```
nodeid | begintime | filename | rownum | rawrecord | detail
--------+-----------+----------+--------+-----------+--------
(0 rows)
```

说明没有错误信息，导入完毕。

步骤 7 停止 GDS

查询 GDS 进程号。

ps -ef|grep gds

返回结果：

```
root       19309       1   0 18:37 ?           00:00:00 /opt/gds/gds -d /opt/gds_input_data/ -p
192.168.0.125:5000 -H 0.0.0.0/0 -l /opt/gds/gds_log.txt -D -t 2
root       19368 19311   0 18:52 pts/0       00:00:00 grep --color=auto gds
```

其中 GDS 进程号为 19309。

kill -9 19309

1.2.2.4 使用 OBS 完成数据导入

步骤 1 创建访问密钥（AK 和 SK）

通过访问以下地址登录 DWS 管理控制台：https://console.huaweicloud.com/dws。

点击右上角用户名，在下拉菜单中单击“我的凭证”。

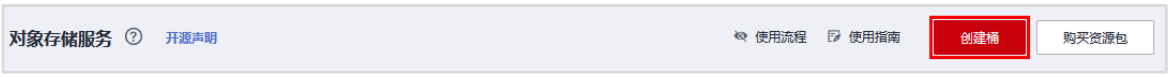
在左侧导航树选择“访问密钥”，单击“新增访问密钥”，输入验证码，“确定”保存 credentials.csv 文件。

打开下载下来的“credentials.csv”文件即可获取到访问密钥（AK 和 SK）。

步骤 2 上传数据到 OBS

1. 访问以下地址登录 OBS 管理控制台：https://storage.huaweicloud.com。或者在华为云界面单击“服务列表”，选择“对象存储服务”，打开 OBS 管理控制台页面。

点击“创建桶”。



区域：和 GaussDB(DWS)同一个区域，此处选择“华北-北京四”

数据冗余存储策略：多 AZ 存储（默认）

桶名称：hqlmybucket（该命名只是举例，请自行设置）。需全局唯一，不能与已有的任何桶名称重复。

存储类别：标准存储

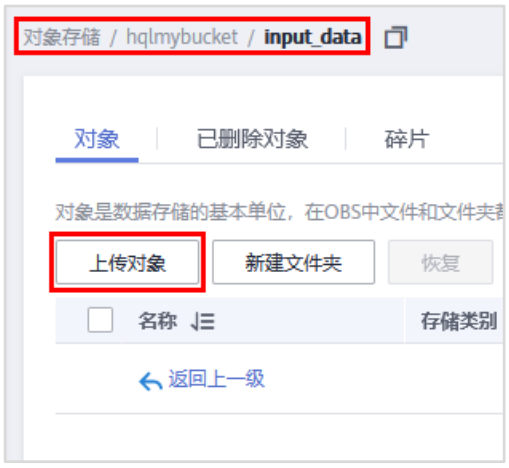
桶策略：公共读写

其他默认设置，点击“立即购买”即可创建桶。

在控制台可以看到个人创建的桶。

在已创建的 OBS 桶“hqlmybucket”中新建一个文件夹“input_data”。

将数据文件 date_dim.dat 上传到 OBS 桶“hqlmybucket”的“input_data”目录中。



获取数据源文件的 OBS 路径。

数据源文件在上传到 OBS 桶之后，会生成全局唯一的访问路径。数据源文件的 OBS 路径用于创建外表时 location 参数设置。location 参数中 OBS 文件的路径由“obs://”、桶名和文件路径组成，即为：obs://<bucket_name>/<file_path>

在本例中，location 参数中数据文件的 OBS 路径为：obs://hqlmybucket/input_data/date_dim.dat

步骤 3 创建 OBS 外表

在 Data Studio 或者 gsql 中创建外表。以下步骤 3-步骤 5 使用 Data Studio 完成。

注意，创建外表时，需要在 ACCESS_KEY 和 SECRET_ACCESS_KEY 处需要进行相应替换。

```
DROP FOREIGN TABLE IF EXISTS foreign_date_dim_obs;
CREATE FOREIGN TABLE foreign_date_dim_obs
(
    d_date_sk               integer               not null,
    d_date_id               char(16)              not null,
    d_date                  date                  ,
    d_month_seq             integer               ,
    d_week_seq              integer               ,
    d_quarter_seq           integer               ,
    d_year                  integer               ,
    d_dow                   integer               ,
    d_moy                   integer               ,
    d_dom                   integer               ,
    d_qoy                   integer               ,
    d_fy_year               integer               ,
    d_fy_quarter_seq        integer               ,
    d_fy_week_seq           integer               ,
    d_day_name              char(9)               ,
    d_quarter_name          char(6)               ,
    d_holiday               char(1)               ,
    d_weekend               char(1)               ,
    d_following_holiday     char(1)               ,
    d_first_dom             integer               ,
    d_last_dom              integer               ,
    d_same_day_ly           integer               ,
    d_same_day_lq           integer               ,
    d_current_day           char(1)               ,
    d_current_week          char(1)               ,
```

```
    d_current_month          char(1)                 ,
    d_current_quarter        char(1)                 ,
    d_current_year           char(1)
)
SERVER gsmpp_server
OPTIONS(
location 'obs://hqlmybucket/input_data/date_dim.csv',
FORMAT 'CSV' ,
DELIMITER ',',
encoding 'utf8',
header 'false',
ACCESS_KEY 'access_key_value_to_be_replaced',
SECRET_ACCESS_KEY 'secret_access_key_value_to_be_replaced',
fill_missing_fields 'true',
ignore_extra_data 'true'
)
READ ONLY
LOG INTO date_dim_obs_err
PER NODE REJECT LIMIT 'unlimited';
```

说明：创建外表时，需要在 ACCESS_KEY 和 SECRET_ACCESS_KEY 中替换为自己创建的 AK 和 SK。

步骤 4 执行导入操作

创建目标表 date_dim_obs。

```
DROP TABLE IF EXISTS date_dim_obs;
create table date_dim_obs
(
    d_date_sk                integer                 not null,
    d_date_id                char(16)                not null,
    d_date                   date                    ,
    d_month_seq              integer                 ,
    d_week_seq               integer                 ,
    d_quarter_seq            integer                 ,
    d_year                   integer                 ,
    d_dow                    integer                 ,
    d_moy                    integer                 ,
    d_dom                    integer                 ,
    d_qoy                    integer                 ,
    d_fy_year                integer                 ,
    d_fy_quarter_seq         integer                 ,
    d_fy_week_seq            integer                 ,
    d_day_name               char(9)                 ,
    d_quarter_name           char(6)                 ,
    d_holiday                char(1)                 ,
    d_weekend                char(1)                 ,
    d_following_holiday      char(1)                 ,
    d_first_dom              integer                 ,
```

```
    d_last_dom              integer                 ,
    d_same_day_ly           integer                 ,
    d_same_day_lq           integer                 ,
    d_current_day           char(1)                 ,
    d_current_week          char(1)                 ,
    d_current_month         char(1)                 ,
    d_current_quarter       char(1)                 ,
    d_current_year          char(1)
);
```

执行导入。

```
INSERT INTO date_dim_obs SELECT * FROM foreign_date_dim_obs;
```

返回结果：

[INFO] 操作影响的记录行数 ：73,049

[INFO] 执行时间 ：932 ms

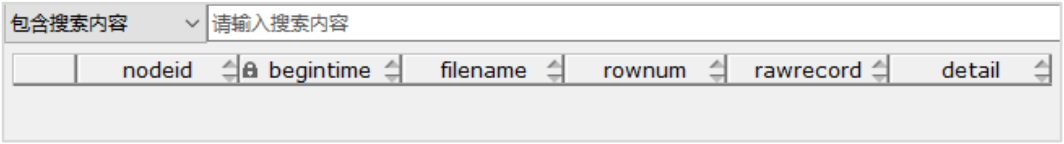
[INFO] 执行成功...

由此可以看到，使用 OBS 完成 73049 条数据导入花费 932ms。GDS 的导入效率较高。

步骤 5 处理错误表

```
SELECT * FROM date_dim_obs_err;
```

返回结果：



说明没有错误信息，导入完毕。

1.2.3 数据类型

完成以下操作：

TINYINT

创建具有 TINYINT 类型数据的表。

```
CREATE TABLE int_type_t1
(
IT_COL1 TINYINT
) ;
```

返回结果：

[INFO] 操作影响的记录行数 ：0

[INFO] 执行时间 ：157 ms

[INFO] 执行成功...

[NOTICE] The 'DISTRIBUTE BY' clause is not specified. Using 'it_col1' as the distribution column by default.

插入数据。

```
INSERT INTO int_type_t1 VALUES(10);
```

返回结果：

[INFO] 操作影响的记录行数 ：1

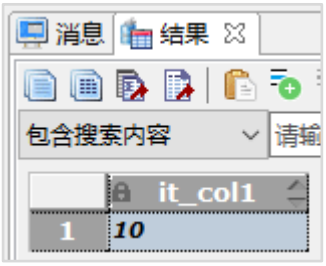
[INFO] 执行时间 ：75 ms

[INFO] 执行成功...

查看数据。

```
SELECT * FROM int_type_t1;
```

返回结果：



DECIMAL

创建表

```
CREATE TABLE decimal_type_t1
(
DT_COL1 DECIMAL(10,4)
);
```

返回结果：

[INFO] 操作影响的记录行数 ：0

[INFO] 执行时间 ：40 ms

[INFO] 执行成功...

[NOTICE] The 'DISTRIBUTE BY' clause is not specified. Using 'dt_col1' as the distribution column by default.

插入数据

```
INSERT INTO decimal_type_t1 VALUES(123456.122331);
```

返回结果：

[INFO] 操作影响的记录行数 ：1

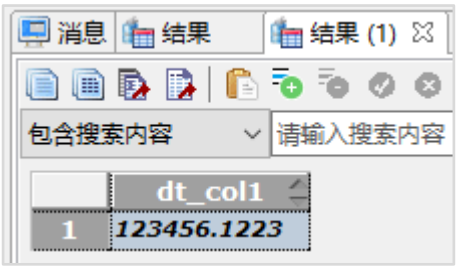
[INFO] 执行时间 ：30 ms

[INFO] 执行成功...

查询表中的数据

```
SELECT * FROM decimal_type_t1;
```

返回结果：



SMALLSERIAL

创建表

CREATE TABLE smallserial_type_tab(a SMALLSERIAL);

返回结果：

[INFO] 操作影响的记录行数 ：0

[INFO] 执行时间 ：209 ms

[INFO] 执行成功...

[NOTICE] CREATE TABLE will create implicit sequence "smallserial_type_tab_a_seq" for serial column "smallserial_type_tab.a"

[NOTICE] The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.

插入数据（完成 2 次插入）

INSERT INTO smallserial_type_tab VALUES(default);
INSERT INTO smallserial_type_tab VALUES(default);

返回结果：

[INFO] 操作影响的记录行数 ：1

[INFO] 执行时间 ：141 ms

[INFO] 执行成功...

[INFO] 操作影响的记录行数 ：1

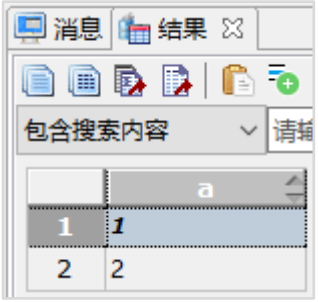
[INFO] 执行时间 ：172 ms

[INFO] 执行成功...

查看数据

SELECT * FROM smallserial_type_tab;

返回结果：



FLOAT

创建表

```
CREATE TABLE float_type_t2
(
FT_COL1 INTEGER,
FT_COL2 FLOAT4,
FT_COL3 FLOAT8,
FT_COL4 FLOAT(3),
FT_COL5 BINARY_DOUBLE,
FT_COL6 DECIMAL(10,4),
FT_COL7 INTEGER(6,3)
) DISTRIBUTE BY HASH ( ft_col1);
```

返回结果：

[INFO] 操作影响的记录行数 ：0

[INFO] 执行时间 ：34 ms

[INFO] 执行成功...

插入数据

```
INSERT INTO float_type_t2 VALUES(10,10.365456,123456.1234,10.3214, 321.321,
123.123654,123.123654);
```

返回结果：

[INFO] 操作影响的记录行数 ：1

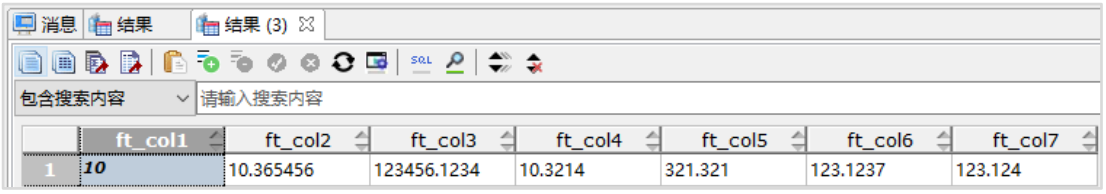
[INFO] 执行时间 ：32 ms

[INFO] 执行成功...

查看数据

```
SELECT * FROM float_type_t2 ;
```

返回结果：



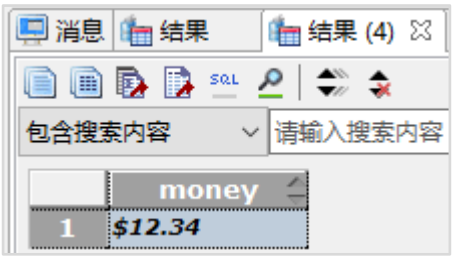
	ft_col1	ft_col2	ft_col3	ft_col4	ft_col5	ft_col6	ft_col7
1	10	10.365456	123456.1234	10.3214	321.321	123.1237	123.124

money

从 real 和 double precision 类型转换到 money 类型，可以先转化为 numeric 类型，再转化为 money 类型。（这种用法是不推荐使用的。浮点数不应该用来处理货币类型，因为小数点的位数可能会导致错误。）

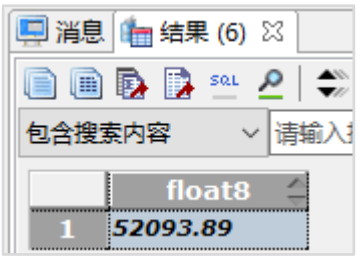
```
SELECT '12.34'::float8::numeric::money;
```

返回结果：



money 类型的值可以转换为 numeric 类型而不丢失精度。转换为其他类型可能丢失精度，并且必须通过以下两步来完成。

```
SELECT '52093.89'::money::numeric::float8;
```



BOOLEAN

创建表

```
CREATE TABLE bool_type_t1
(
BT_COL1 BOOLEAN,
BT_COL2 TEXT
) DISTRIBUTE BY HASH(BT_COL2);
```

返回结果：

[INFO] 操作影响的记录行数 ：0

[INFO] 执行时间 ：47 ms

[INFO] 执行成功...

插入数据

```
INSERT INTO bool_type_t1 VALUES (TRUE, 'sic est');
INSERT INTO bool_type_t1 VALUES (FALSE, 'non est');
```

返回结果：

[INFO] 操作影响的记录行数 ：1

[INFO] 执行时间 ：27 ms

[INFO] 执行成功...

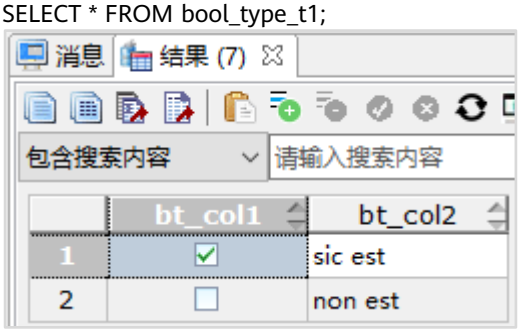
[INFO] 操作影响的记录行数 ：1

[INFO] 执行时间 ：19 ms

[INFO] 执行成功...

查看数据

SELECT * FROM bool_type_t1;



字符类型

创建表

```
CREATE TABLE char_type_t1
(
CT_COL1 CHARACTER(4)
) DISTRIBUTE BY HASH (CT_COL1);
```

返回结果：

[INFO] 操作影响的记录行数 ：0

[INFO] 执行时间 ：33 ms

[INFO] 执行成功...

插入数据

INSERT INTO char_type_t1 VALUES ('ok');

返回结果：

[INFO] 操作影响的记录行数 ：1

[INFO] 执行时间 ：19 ms

[INFO] 执行成功...

查询表中的数据

SELECT ct_col1, char_length(ct_col1) FROM char_type_t1;

返回结果：

二进制类型

创建表

```
CREATE TABLE blob_type_t1
(
BT_COL1 INTEGER,
BT_COL2 BLOB,
BT_COL3 RAW,
BT_COL4 BYTEA
) DISTRIBUTE BY REPLICATION;
```

返回结果：

[INFO] 操作影响的记录行数 ：0

[INFO] 执行时间 ：44 ms

[INFO] 执行成功...

插入数据

```
INSERT INTO blob_type_t1 VALUES(10,empty_blob(), HEXTORAW('DEADBEEF'), E'\\xDEADBEEF');
```

返回结果：

[INFO] 操作影响的记录行数 ：1

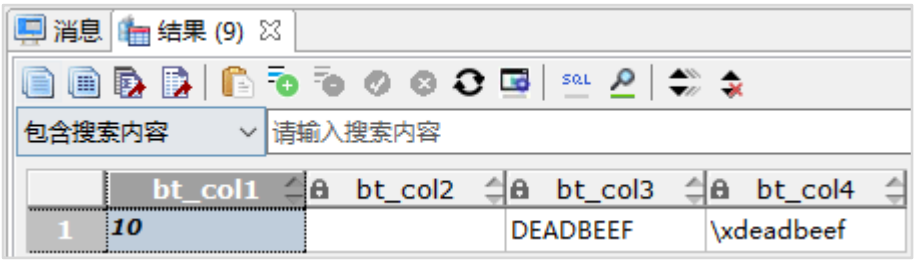
[INFO] 执行时间 ：36 ms

[INFO] 执行成功...

查询表中的数据

```
SELECT * FROM blob_type_t1;
```

返回结果：



日期/时间类型

创建表

```
CREATE TABLE date_type_tab(coll date);
```

返回结果：

[INFO] 操作影响的记录行数 ：0

[INFO] 执行时间 ：41 ms

[INFO] 执行成功...

[NOTICE] The 'DISTRIBUTE BY' clause is not specified. Using 'coll' as the distribution column by default.

插入数据

```
INSERT INTO date_type_tab VALUES (date '12-10-2010');
```

返回结果：

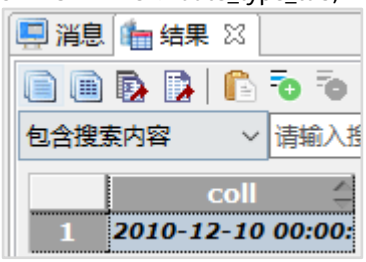
[INFO] 操作影响的记录行数 ：1

[INFO] 执行时间 ：32 ms

[INFO] 执行成功...

查看数据

```
SELECT * FROM date_type_tab;
```



位串类型

创建表

```
CREATE TABLE bit_type_t1
(
BT_COL1 INTEGER,
BT_COL2 BIT(3),
BT_COL3 BIT VARYING(5)
) DISTRIBUTE BY REPLICATION;
```

返回结果：

[INFO] 操作影响的记录行数 ：0

[INFO] 执行时间 ：39 ms

[INFO] 执行成功...

插入数据

```
INSERT INTO bit_type_t1 VALUES(1, B'101', B'00');
```

返回结果：

[INFO] 操作影响的记录行数 ：1

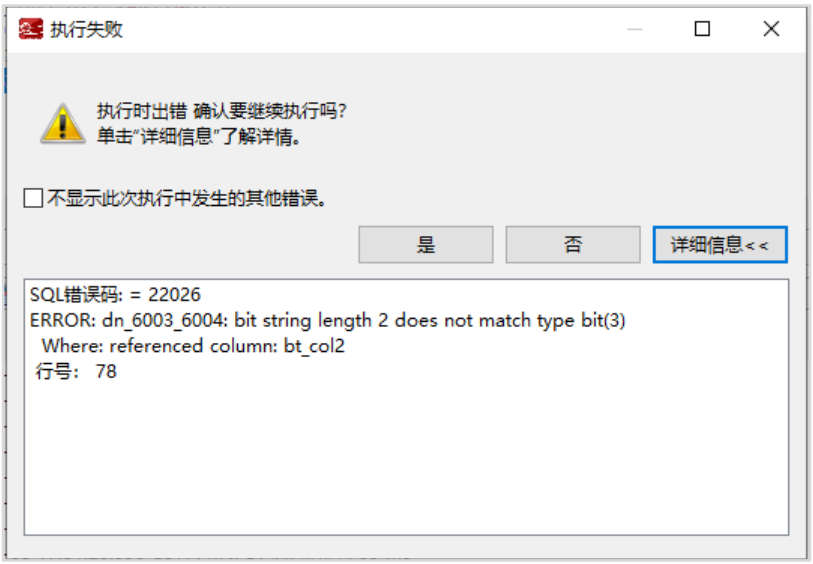
[INFO] 执行时间 ：33 ms

[INFO] 执行成功...

插入数据的长度不符合类型的标准会报错

```
INSERT INTO bit_type_t1 VALUES(2, B'10', B'101');
```

返回结果：



[ERROR] 执行失败

错误代码：[0]SQL 错误码: = 22026

ERROR: dn_6003_6004: bit string length 2 does not match type bit(3)
Where: referenced column: bt_col2
Line Number: 78

将不符合类型长度的数据进行转换。

```
INSERT INTO bit_type_t1 VALUES(2, B'10'::bit(3), B'101');
```

返回结果：

[INFO] 操作影响的记录行数 ：1

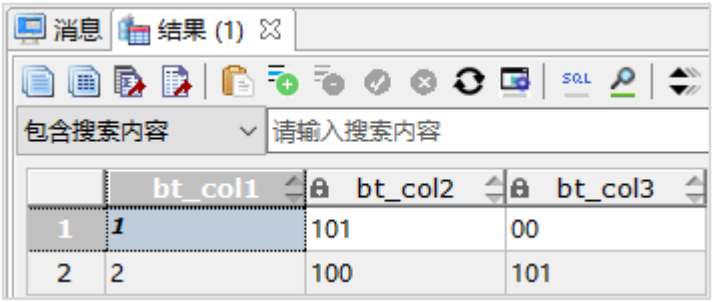
[INFO] 执行时间 ：85 ms

[INFO] 执行成功...

查看数据

```
SELECT * FROM bit_type_t1;
```

返回结果：

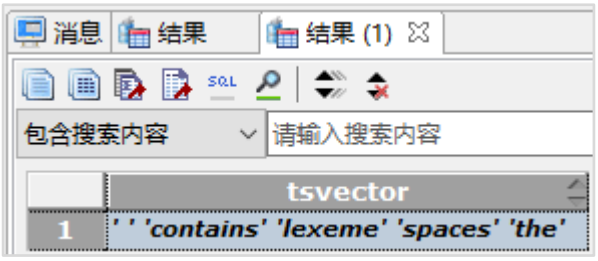


文本搜索类型

tsvector 的值是唯一分词的分类列表，把一句话的词格式化为不同的词条，在进行分词处理的时候 tsvector 会自动去掉分词中重复的词条，按照一定的顺序录入。如：

SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
返回结果：

从上面的例子可以看出，通过 tsvector 把一个字符串按照空格进行分词，分词的顺序是按照长短和字母排序的。但是如果词条中需要包含空格或标点符号，可以用引号标记：
SELECT $$the lexeme ' ' contains spaces$$::tsvector;
返回结果：



tsquery 类型表示一个检索条件，存储用于检索的词汇，并且使用布尔操作符&（ AND），|（ OR）和!（ NOT）来组合他们，括号用来强调操作符的分组。 to_tsquery 函数及 plainto_tsquery 函数会将单词转换为 tsquery 类型前进行规范化处理。
SELECT 'fat & rat'::tsquery;
返回结果：

SELECT 'fat & (rat | cat)'::tsquery;
返回结果：

SELECT 'fat & rat & ! cat'::tsquery;
返回结果：

1.2.4 函数与操作符

1.2.4.1 字符处理函数和操作符

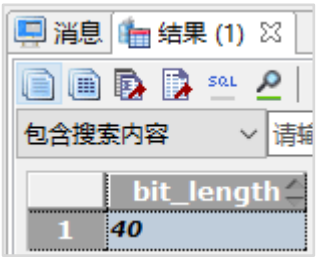
函数：bit_length(string)

描述：字符串的位数。

返回值类型：int

SELECT bit_length('world');

返回结果：



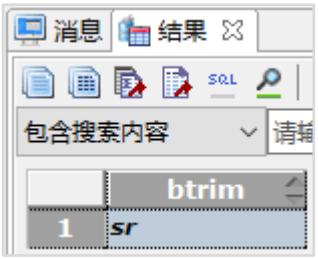
函数：btrim(string text [, characters text])

描述：从 string 开头和结尾删除只包含 characters 中字符（缺省是空白）的最长字符串。

返回值类型：text

SELECT btrim('sring' , 'ing');

返回结果：



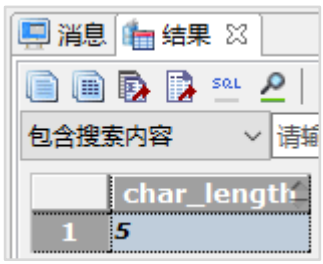
函数：char_length(string)或 character_length(string)

描述：字符串中的字符个数。

返回值类型：int

```
SELECT char_length('hello');
```

返回结果：



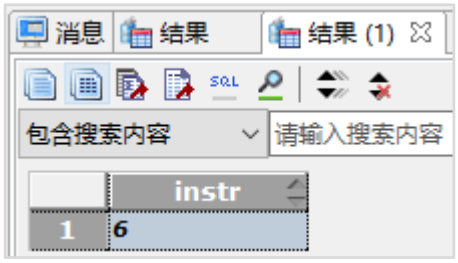
函数：instr(text,text,int,int)

描述：字符串匹配函数的位置，第一个 int 表示开始匹配起始位置，第二个 int 表示匹配的次数。

返回值类型：int

```
SELECT instr( 'abcdabcdabcd', 'bcd', 2, 2 );
```

返回结果：



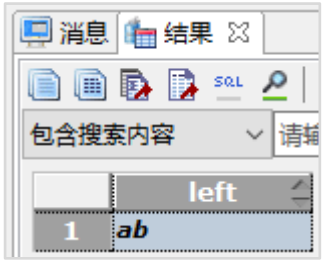
函数：left(str text, n int)

描述：返回字符串的前 n 个字符。当 n 是负数时，返回除最后|n|个字符以外的所有字符。

返回值类型：text

```
SELECT left('abcde', 2);
```

返回结果：



函数：length(string bytea, encoding name)

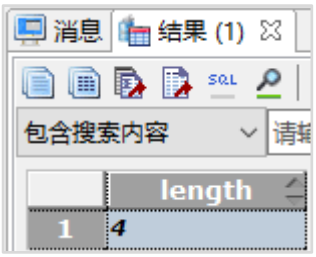
描述：指定 encoding 编码格式的 string 的字符数。在这个编码格式中，string 必须是有效的。

返回值类型：int

示例：

```
SELECT length('jose', 'UTF8');
```

返回结果：



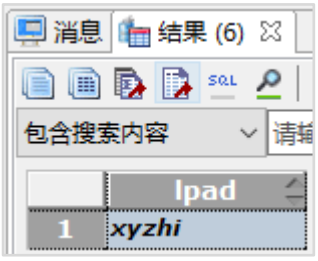
函数：lpad(string text, length int [, fill text])

描述：通过填充字符 fill（缺省时为空白），把 string 填充为 length 长度。如果 string 已经比 length 长则将其尾部截断。

返回值类型：text

```
SELECT lpad('hi', 5, 'xyza');
```

返回结果：



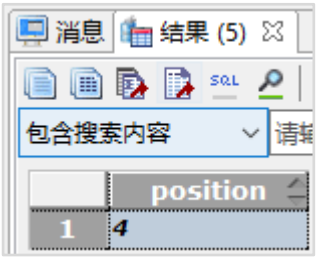
函数：position(substring in string)

描述：指定子字符串的位置。

返回值类型：int

```
SELECT position('ing' in 'string');
```

返回结果：



函数：replace(string text, from text, to text)

描述：把字符串 string 里出现地所有子字符串 from 的内容替换成子字符串 to 的内容。

返回值类型：text

SELECT replace('abcdefabcdef', 'cd', 'XXX');

返回结果：

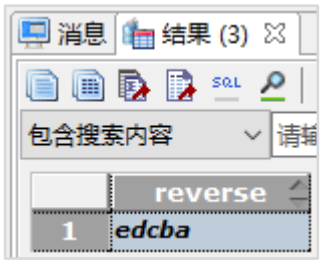
函数：reverse(str)

描述：返回颠倒的字符串。

返回值类型：text

SELECT reverse('abcde');

返回结果：



操作符：string || string

描述：连接字符串。

返回值类型：text

SELECT 'MPP'||'DB' AS RESULT;

返回结果：

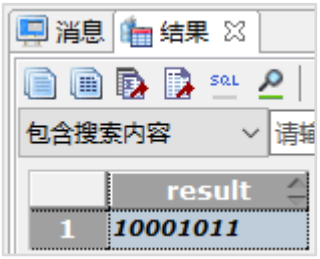
1.2.4.2 位串函数和操作符

操作符：||

描述：位串之间进行连接。

SELECT B'10001' || B'011' AS RESULT;

返回结果：

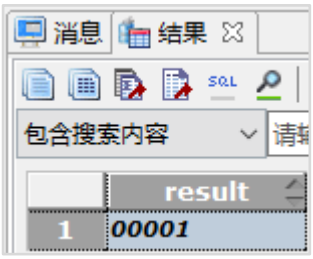


操作符：&

描述：位串之间进行“与”操作。

```
SELECT CAST(B'10001' & B'01101' AS VARCHAR) AS RESULT;
```

返回结果：

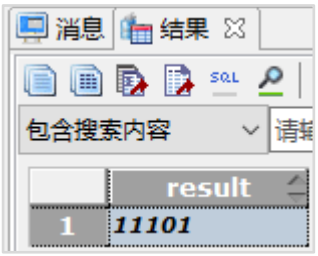


操作符：|

描述：位串之间进行“或”操作。

```
SELECT CAST(B'10001' | B'01101' AS VARCHAR) AS RESULT;
```

返回结果：

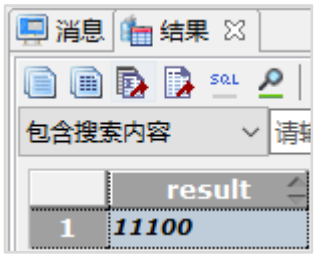


操作符：#

描述：位串之间如果不一致进行“或”操作。如果两个位串中对应位置都为 1 或者 0，则该位置返回为 0。

```
SELECT CAST(B'10001' # B'01101' AS VARCHAR) AS RESULT;
```

返回结果：

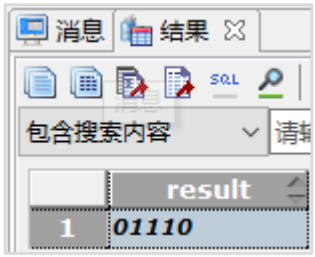


操作符：~

描述：位串之间进行“非”操作。

SELECT CAST(~B'10001' AS VARCHAR) AS RESULT;

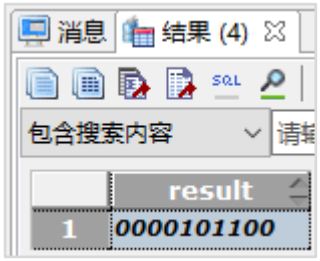
返回结果：



整数和 bit 之间来回转换

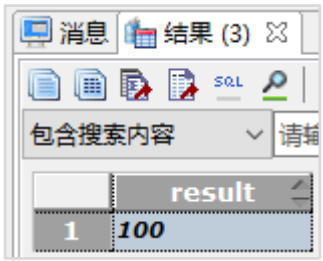
SELECT 44::bit(10) AS RESULT;

返回结果：



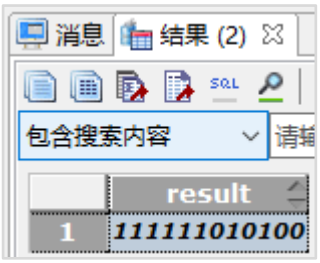
SELECT 44::bit(3) AS RESULT;

返回结果：



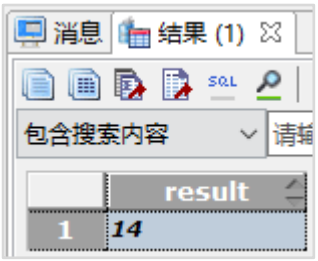
SELECT cast(-44 as bit(12)) AS RESULT;

返回结果：



SELECT '1110'::bit(4)::integer AS RESULT;

返回结果：



1.2.4.3 数字操作函数和操作符

操作符：+

描述：加

```
SELECT 2+3 AS RESULT;
 result
--------
        5
```

操作符：|/

描述：平方根

```
SELECT |/ 25.0 AS RESULT;
 result
--------
        5
```

函数：abs(x)

描述：绝对值。

返回值类型：和输入相同。

```
SELECT abs(-17.4);
 abs
------
 17.4
```

函数：ceil(x)

描述：不小于参数的最小的整数。

返回值类型：整数。

```
SELECT ceil(-42.8);
 ceil
------
  -42
```

函数：degrees(dp)

描述：把弧度转为角度。

返回值类型：double precision

```
SELECT degrees(0.5);
       degrees
------------------
 28.6478897565412
```

函数：floor(x)

描述：不大于参数的最大整数。

返回值类型：与输入相同。

```
SELECT floor(-42.8);
 floor
-------
   -43
```

函数：mod(x,y)

描述：x/y 的余数（模），如果 x 是 0，则返回 y。

返回值类型：与参数类型相同。

```
SELECT mod(9,4);
 mod
-----
   1

SELECT mod(9,0);
 mod
-----
   9
```

函数：round(v numeric, s int)

描述：保留小数点后 s 位，s 后一位进行四舍五入。

返回值类型：numeric

```
SELECT round(42.4382, 2);
 round
-------
 42.44
```

1.2.4.4 时间和日期处理函数和操作符

函数：age(timestamp, timestamp)

描述：将两个参数相减，并以年、月、日作为返回值。若相减值为负，则函数返回亦为负。

返回值类型：interval

```
SELECT age(timestamp '2001-04-10', timestamp '1957-06-13');
                age
-------------------------
 43 years 9 mons 27 days
```

函数：current_date

描述：当前日期。

返回值类型：date

```
SELECT current_date;
     date
------------
 2020-02-27
```

函数：date_trunc(text, timestamp)

描述：截取到参数 text 指定的精度。

返回值类型：timestamp

```
SELECT date_trunc('hour', timestamp   '2001-02-16 20:38:40');
       date_trunc
----------------------
 2001-02-16 20:00:00
```

函数：extract(field from timestamp)

描述：获取小时的值。

返回值类型：double precision

```
SELECT extract(hour from timestamp '2001-02-16 20:38:40');
 date_part
-----------
        20
```

函数：now()

描述：当前日期及时间。

返回值类型：timestamp with time zone

```
SELECT now();
                  now
-------------------------------
 2020-02-27 10:15:42.549426+08
```

函数：add_months(d,n)

描述：用于计算时间点 d 再加上 n 个月的时间。

返回值类型：timestamp

```
SELECT add_months(to_date('2017-5-29', 'yyyy-mm-dd'), 11) FROM dual;
      add_months
---------------------
 2018-04-29 00:00:00
```

1.2.4.5 类型转换函数

函数：cast(x as y)

描述：类型转换函数，将 x 转换成 y 指定的类型。

```
SELECT cast('22-oct-1997' as timestamp);
       timestamp
---------------------
 1997-10-22 00:00:00
```

函数：to_date(text)

描述：将文本类型的值转换为指定格式的时间戳。

返回值类型：timestamp

```
SELECT to_date('2015-08-14');
         to_date
---------------------
 2015-08-14 00:00:00
```

函数：to_char(int, text)

描述：将整数类型的值转换为指定格式的字符串。

返回值类型：text

```
SELECT to_char(125, '999');
 to_char
---------
   125
```

函数：numtoday(numeric)

描述：将数字类型的值转换为指定格式的时间戳。

返回值类型：timestamp

```
SELECT numtoday(2);
 numtoday
----------
 2 days
```

1.2.4.6 几何函数和操作符

操作符：+

描述：平移。

```
SELECT box '((0,0),(1,1))' + point '(2.0,0)' AS RESULT;
    result
-------------
 (3,1),(2,0)
```

操作符：@-@

描述：图形的长度或者周长。

```
SELECT @-@ path '((0,0),(1,0))' AS RESULT;
 result
--------
    2.0
```

函数：area(object)

描述：计算图形的面积。

返回类型：double precision

```
SELECT area(box '((0,0),(1,1))') AS RESULT;
 result
--------
    1.0
```

函数：height(box)

描述：矩形的竖直高度。

返回类型：double precision

```
SELECT height(box '((0,0),(1,1))') AS RESULT;
 result
--------
    1.0
```

函数：length(object)

描述：计算图形的长度。

返回类型：double precision

```
SELECT length(path '((-1,0),(1,0))') AS RESULT;
 result
--------
    4.0
```

函数：box(circle)

描述：将圆转换成矩形

返回类型：box

```
SELECT box(circle '((0,0),2.0)') AS RESULT;
                                          result
-------------------------------------------------------------------------
 (1.41421356237309,1.41421356237309),(-1.41421356237309,-1.41421356237309)
```

函数：circle(polygon)

描述：将多边形转换成圆

返回类型：circle

```
SELECT circle(polygon '((0,0),(1,1),(2,0))') AS RESULT;
                        result
-----------------------------------------
 <(1,0.333333333333333),0.924950591148529>
```

1.2.4.7 网络地址函数和操作符

操作符：<

描述：小于

```
SELECT inet '192.168.1.5' < inet '192.168.1.6' AS RESULT;
 result
--------
 t
```

函数：broadcast(inet)

描述：网络广播地址。

返回类型：inet

```
SELECT broadcast('192.168.1.5/24') AS RESULT;
        result
------------------
 192.168.1.255/24
```

函数：family(inet)

描述：抽取地址族，4 为 IPv4，6 为 IPv6。

返回类型：int

```
SELECT family('::1') AS RESULT;
 result
--------
      6
```

函数：host(inet)

描述：将主机地址类型抽出为文本。

返回类型：text

```
SELECT host('192.168.1.5/24') AS RESULT;
    result
-------------
 192.168.1.5
```

1.2.4.8 UUID 函数

函数：uuid_generate_v1()

描述：生成一个 UUID 类型的序列号。

返回类型：UUID

```
SELECT uuid_generate_v1();
              uuid_generate_v1
---------------------------------------
 c71ceaca-a175-11e9-a920-797ff7000001
```

函数：sys_guid()

描述：生成一个和 Oracle 的 sys_guid 方法相同的序列号。

返回类型：text

```
SELECT sys_guid();
                sys_guid
---------------------------------
 4EBD3C74A17A11E9A1BF797FF7000001
```

1.2.4.9 JSON 函数

函数：array_to_json(anyarray [, pretty_bool])

描述：返回 JSON 类型的数组。一个多维数组成为一个 JSON 数组的数组。如果 pretty_bool 为 true，将在一维元素之间添加换行符。

返回类型：json

```
SELECT array_to_json('{{1,5},{99,100}}'::int[]);
```

```
array_to_json
------------------
[[1,5],[99,100]]
```

函数：row_to_json(record [, pretty_bool])

描述：返回 JSON 类型的行。如果 pretty_bool 为 true，将在第一级元素之间添加换行符。

返回类型：json

```
SELECT row_to_json(row(1,'foo'));
      row_to_json
----------------------
 {"f1":1,"f2":"foo"}
```

1.2.4.10 SEQUENCE 函数

创建序列

```
CREATE SEQUENCE seqDemo START 2 INCREMENT 3 MAXVALUE 110;
```

返回结果：

[INFO] 操作影响的记录行数 ：0

[INFO] 执行时间 ：425 ms

[INFO] 执行成功...

函数：nextval(regclass)

描述：递增序列并返回新值。

返回类型：bigint

nextval 函数有两种调用方式（其中第二种调用方式兼容 Oracle 的语法，目前不支持 Sequence 命名中有特殊字符"."的情况），如下：

示例 1：

```
select nextval('seqDemo');
 nextval
---------
         2
```

示例 2（在完成示例 1 的基础上）：

```
select seqDemo.nextval;
 nextval
---------
         5
```

说明：为了避免从同一个序列获取值的并发事务被阻塞， nextval 操作不会回滚；也就是说，一旦一个值已经被抓取， 那么就认为它已经被用过了，并且不会再被返回。即使该操作处于

事务中，当事务之后中断，或者如果调用查询结束不使用该值，也是如此。这种情况将在指定值的顺序中留下未使用的"空洞"。因此，GaussDB 200 序列对象不能用于获得"无间隙"序列。

如果 nextval 被下推到 DN 上时，各个 DN 会自动连接 GTM，请求 next values 值，例如（insert into t1 select xxx，t1 某一列需要调用 nextval 函数），由于 GTM 上有最大连接数为 8192 的限制，而这类下推语句会导致消耗过多的 GTM 连接数，因此对于这类语句的并发数目限制为 7000（其它语句需要占用部分连接）/集群 DN 数目。

1.2.4.11 聚集函数

创建表

```
CREATE TABLE test
(
COL1 INTEGER,
COL2 FLOAT4,
COL3 FLOAT8,
COL4 FLOAT(3),
COL6 DECIMAL(10,4),
COL7 INTEGER(6,3)
) DISTRIBUTE BY HASH (col1);
```

插入数据

```
INSERT INTO test VALUES(10,10.365456,123456.1234,10.3214, 123.123654,123.123654);
INSERT INTO test VALUES(11,11.365456,123756.1234,14.3214, 173.145654,133.125654);
INSERT INTO test VALUES(12,12.365456,203463.5634,13.3124, 123.167654,323.323657);
INSERT INTO test VALUES(13,13.365456,123346.1234,12.3315, 269.123784,123.122690);
INSERT INTO test VALUES(14,14.365456,122356.1234,11.6714, 347.124856,563.127664);
INSERT INTO test VALUES(15,15.365456,112456.1234,11.6664, 234.123754,233.123764);
INSERT INTO test VALUES(16,16.365456,123436.1234,12.2224, 237.147654,163.157354);
INSERT INTO test VALUES(17,17.365456,223456.1234,10.6784, 437.124784,383.125655);
INSERT INTO test VALUES(18,18.365456,123472.1234,9.3414, 343.123784,143.135674);
INSERT INTO test VALUES(19,19.365456,132456.1234,8.1214, 213.123754,120.124554);
```

函数：sum(expression)

描述：所有输入行的 expression 总和。

返回类型：通常情况下输入数据类型和输出数据类型是相同的，但以下情况会发生类型转换：

对于 SMALLINT 或 INT 输入，输出类型为 BIGINT。

对于 BIGINT 输入，输出类型为 NUMBER 。

对于浮点数输入，输出类型为 DOUBLE PRECISION。

示例：

```
SELECT SUM(col1) FROM test;
  sum
----------
  145
```

函数：max(expression)

描述：所有输入行中 expression 的最大值；

参数类型：任意数组、数值、字符串、日期/时间类型。

返回类型：与参数数据类型相同

```
SELECT MAX(col2) FROM test;
  max
----------
  19.3655
```

函数：avg(expression)

描述：所有输入值的均值（算术平均）。

返回类型：

对于任何整数类型输入，结果都是 NUMBER 类型。

对于任何浮点输入，结果都是 DOUBLE PRECISION 类型。

否则和输入数据类型相同。

示例：

```
SELECT AVG(col3) FROM test;
            avg
----------------------
 141165.4674
```

函数：count(expression)

描述：返回表中满足 expression 不为 NULL 的行数。

返回类型：BIGINT

示例：

```
SELECT COUNT(*) FROM test;
  count
----------
 10
```

函数：corr(Y, X)

描述：相关系数

返回类型：double precision

示例：

```
SELECT CORR(col4, col6) FROM test;
          corr
-------------------
 -.30002176975077
```

1.2.4.12 窗口函数

列存表目前只支持 rank(expression)和 row_number(expression)两个函数。

窗口函数与 OVER 语句一起使用。OVER 语句用于对数据进行分组，并对组内元素进行排序。窗口函数用于给组内的值生成序号。

创建表

```
create table date_dim
(
    d_date_sk                 integer                 not null,
    d_date_id                 char(16)                not null,
    d_date                    date                          ,
    d_month_seq               integer                       ,
    d_week_seq                integer                       ,
    d_quarter_seq             integer                       ,
    d_year                    integer                       ,
    d_dow                     integer                       ,
    d_moy                     integer                       ,
    d_dom                     integer                       ,
    d_qoy                     integer                       ,
    d_fy_year                 integer                       ,
    d_fy_quarter_seq          integer                       ,
    d_fy_week_seq             integer                       ,
    d_day_name                char(9)                       ,
    d_quarter_name            char(6)                       ,
    d_holiday                 char(1)                       ,
    d_weekend                 char(1)                       ,
    d_following_holiday       char(1)                       ,
    d_first_dom               integer                       ,
    d_last_dom                integer                       ,
    d_same_day_ly             integer                       ,
    d_same_day_lq             integer                       ,
    d_current_day             char(1)                       ,
    d_current_week            char(1)                       ,
    d_current_month           char(1)                       ,
    d_current_quarter         char(1)                       ,
    d_current_year            char(1)
    );
```

数据准备

使用 Data Studio 完成数据导入，具体可以参考数据导入练习部分。

函数：RANK()

描述：RANK 函数为各组内值生成跳跃排序序号，其中，相同的值具有相同序号。

返回值类型：BIGINT

示例：

SELECT d_moy, d_fy_week_seq, rank() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;

显示 42 行，因篇幅问题不再展示。

函数：ROW_NUMBER()

描述：ROW_NUMBER 函数为各组内值生成连续排序序号，其中，相同的值其序号不相同。

返回值类型：BIGINT

示例：

SELECT d_moy, d_fy_week_seq, Row_number() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;

显示 42 行，因篇幅问题不再展示。

函数：DENSE_RANK()

描述：DENSE_RANK 函数为各组内值生成连续排序序号，其中，相同的值具有相同序号。

返回值类型：BIGINT

示例：

SELECT d_moy, d_fy_week_seq, dense_rank() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;

显示 42 行，因篇幅问题不再展示。

函数：PERCENT_RANK()

描述：PERCENT_RANK 函数为各组内对应值生成相对序号，即根据公式（rank - 1）/ (total rows - 1)计算所得的值。其中 rank 为该值依据 RANK 函数所生成的对应序号，totalrows 为该分组内的总元素个数。

返回值类型：DOUBLE PRECISION

示例：

SELECT d_moy, d_fy_week_seq, percent_rank() OVER(PARTITION BY d_moy ORDER BY d_fy_week_seq) FROM date_dim WHERE d_moy < 4 AND d_fy_week_seq < 7 ORDER BY 1,2;

显示 42 行，因篇幅问题不再展示。

1.2.4.13 安全函数

函数：gs_encrypt_aes128(encryptstr,keystr)

描述：以 keystr 为密钥对 encryptstr 字符串进行加密，返回加密后的字符串。keystr 的长度范围为 1~16 字节。支持的加密数据类型：目前数据库支持的数值类型，字符类型，二进制类型中的 RAW，日期/时间类型中的 DATE、TIMESTAMP、SMALLDATETIME。

返回值类型：text

返回值长度：至少为 92 字节，不超过 4*[(Len+68)/3] 字节，其中 Len 为加密前数据长度（单位为字节）。

示例：

```
SELECT gs_encrypt_aes128('MPPDB','1234');

                                                gs_encrypt_aes128
--------------------------------------------------------------------------------------------
gwditQLQG8NhFw4OuoKhhQJoXojhFlYkjeG0aYdSCtLCnIUgkNwvYI04KbuhmcGZp8jWizBdR1vU9CspjuzI0lbz12A=
```

函数：gs_decrypt_aes128(decryptstr,keystr)

描述：以 keystr 为密钥对 decrypt 字符串进行解密，返回解密后的字符串。解密使用的 keystr 必须保证与加密时使用的 keystr 一致才能正常解密。keystr 不得为空。

说明：

此参数需要结合 gs_encrypt_aes128 加密函数共同使用。

返回值类型：text

示例：

```
SELECT
gs_decrypt_aes128('gwditQLQG8NhFw4OuoKhhQJoXojhFlYkjeG0aYdSCtLCnIUgkNwvYI04KbuhmcGZp8jWizBdR1vU9CspjuzI0lbz12A=','1234');
 gs_decrypt_aes128
-------------------
 MPPDB
```

1.2.4.14 返回集合的函数

generate_series(start, stop)

描述：生成一个数值序列，从 start 到 stop，步长为 1。

参数类型：int、bigint、numeric

返回值类型：setof int、setof bigint、setof numeric（与参数类型相同）

generate_series(start, stop, step)

描述：生成一个数值序列，从 start 到 stop，步长为 step。

参数类型：int、bigint、numeric

返回值类型：setof int、setof bigint、setof numeric（与参数类型相同）

generate_series(start, stop, step interval)

描述：生成一个数值序列，从 start 到 stop，步长为 step。

参数类型：timestamp 或 timestamp with time zone

返回值类型：setof timestamp 或 setof timestamp with time zone（与参数类型相同）

如果 step 是正数且 start 大于 stop，则返回零行。相反，如果 step 是负数且 start 小于 stop，则也返回零行。如果输入是 NULL，同样产生零行。如果 step 为零则是一个错误。

```
SELECT * FROM generate_series(2,4);
 generate_series
```

```
-----------------
                2
                3
                4

SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
                5
                3
                1

SELECT * FROM generate_series(4,3);
 generate_series
-----------------
```

1.2.4.15 条件表达式函数

函数：nullif(expr1, expr2)

描述：

当且仅当 expr1 和 expr2 相等时，NULLIF 才返回 NULL，否则它返回 expr1。

nullif(expr1, expr2) 逻辑上等价于 CASE WHEN expr1 = expr2 THEN NULL ELSE expr1 END。

示例：

```
SELECT nullif('hello','world');
 nullif
--------
 hello
```

备注：

如果两个参数的数据类型不同，则：

若两种数据类型之间存在隐式转换，则以其中优先级较高的数据类型为基准将另一个参数隐式转换成该类型，转换成功则进行计算，转换失败则返回错误。如：

```
SELECT nullif('1234'::VARCHAR,123::INT4);
 nullif
--------
    1234

SELECT nullif('1234'::VARCHAR,'2012-12-24'::DATE);
ERROR:  invalid input syntax for type timestamp: "1234"
```

若两种数据类型之间不存在隐式转换，则返回错误 。如：

```
SELECT nullif(TRUE::BOOLEAN,'2012-12-24'::DATE);
ERROR:  operator does not exist: boolean = timestamp without time zone
```

```
LINE 1: SELECT nullif(TRUE::BOOLEAN,'2012-12-24'::DATE) FROM DUAL;
^
HINT:  No operator matches the given name and argument type(s). You might need to add explicit type casts.
```

函数：nvl(expr1 , expr2)

描述：

如果 expr1 为 NULL 则返回 expr2。

如果 expr1 非 NULL，则返回 expr1。

```
SELECT nvl('hello','world');
  nvl
-------
 hello
```

备注：参数 expr1 和 expr2 可以为任意类型，当 NVL 的两个参数不属于同类型时，看第二个参数是否可以向第一个参数进行隐式转换，如果可以则返回第一个参数类型。如果第二个参数不能向第一个参数进行隐式转换而第一个参数可以向第二个参数进行隐式转换，则返回第二个参数的类型。如果两个参数之间不存在隐式类型转换并且也不属于同一类型则报错。

1.2.4.16 系统信息函数

函数：current_database()

描述：当前数据库的名字。

返回值类型：name

```
SELECT current_database();
 current_database
------------------
 hql_demo
```

函数：current_schemas(boolean)

描述：搜索路径中的模式名字。

返回值类型：name[]

```
SELECT current_schemas(true);
    current_schemas
----------------------
 {pg_catalog,public}
```

函数：pg_backend_pid()

描述：当前会话连接的服务进程的进程 ID。

返回值类型：int

```
SELECT pg_backend_pid();
 pg_backend_pid
```

```
-----------------
 140229352617744
```

函数：session_user

描述：会话用户名。

返回值类型：name

```
SELECT session_user;
 session_user
--------------
 dbadmin
```

备注：session_user 通常是连接当前数据库的初始用户，不过系统管理员可以用 SET SESSION AUTHORIZATION 修改这个设置。

1.2.5 清理实验环境

```
# 通过 DROP 语句删除所有表和序列
drop table bit_type_t1;
drop table blob_type_t1;
drop table bool_type_t1;
drop table char_type_t1;
drop table daily_uniques;
drop table date_dim;
drop table date_type_tab;
drop table decimal_type_t1;
drop table facts;
drop table float_type_t2;
drop table int_type_t1;
drop sequence seqdemo;
drop table smallserial_type_tab;
drop table test;
```

1.2.6 本节小结

本小节实验主要完成 GaussDB(DWS)中基本的数据类型、函数与操作符的相关操作，并对通过不同方式进行数据导入的效率进行简要对比。

2 Explain 工具介绍

2.1 实验介绍

2.1.1 关于 explain

本节内容通过对 explain 的介绍，使学员可以读懂 SQL 执行计划，发现执行中的性能瓶颈点。

2.1.2 实验目的

- 理解 explain 基本语法。
- 读懂 SQL 执行计划。
- 可以根据执行信息找出 SQL 的瓶颈点。
- 读懂不同显示格式下的内容。

2.2 实验步骤

2.2.1 环境准备

连接 DWS：可以使用 Data Studio 进行连接，也可以使用 gsql 进行连接。本小节主要使用 gsql 完成。

使用 gsql 连接。

```
gsql -d postgres -h 121.36.103.171 -U dbadmin -W Dws@1234 -p 8000 -r
```

其中，121.36.103.171 为 DWS 公网 IP，请进行替换。连接后请创建新的数据库，数据库命名建议为“个人姓名简称_demo”，例如 hql_demo。

```
CREATE DATABASE hql_demo;
```

切换到新数据库：

```
\c hql_demo（此处替换为个人创建的数据库名）
```

此小节所有的操作均在新创建的数据库下操作。

2.2.2 数据准备

步骤 1 准备数据文件

将数据文件 web_page.dat, inventory.dat, customer.dat 存放到 ECS 某个目录下，如：/opt/gsql_input_data

进入数据文件目录：

cd /opt/gsql_input_data

查看目录下的数据文件：

ll

返回结果：

```
total 232228
-rw-r--r-- 1 root root   13109372 Jul 14 19:34 customer.dat
-rw-r--r-- 1 root root 224675139 Jul 14 19:37 inventory.dat
-rw-r--r-- 1 root root         5716 Jul 14 19:37 web_page.dat
```

步骤 2 创建表

连接数据库：

cd /opt

source gsql_env.sh

gsql -d hql_demo -h 114.116.200.18 -p 8000 -U dbadmin -W Dws@123456 -r

说明：请替换为个人的数据库名，集群公网 IP，管理员用户密码

依次创建如下三张表：

表定义 1：

```
DROP TABLE IF EXISTS inventory;
create table inventory
(
    inv_date_sk                integer                 not null,
    inv_item_sk                integer                 not null,
    inv_warehouse_sk            integer                 not null,
    inv_quantity_on_hand        integer
)
distribute by hash (inv_item_sk)
partition by range(inv_date_sk)
(
        partition p1 values less than(2451179),
        partition p2 values less than(2451544),
        partition p3 values less than(2451910),
        partition p4 values less than(2452275),
        partition p5 values less than(2452640),
        partition p6 values less than(2453005),
        partition p7 values less than(maxvalue)
)
;
```

说明：

inventory 表是一个分区表，其中的 distribute by hash (inv_item_sk)

指定的是该表的分布列，该表是以 inv_item_sk 字段来进行哈希分布的，即数据是以该字段分布到不同的数据节点上去的。partition by range(inv_date_sk)指定了分区键，range(inv_date_sk)说明的是根据字段 inv_date_sk 的取值范围来决定数据插入到哪一个分区表中。

partition p1 values less than(2451179)：即限制 inv_date_sk 小于 2451179 的数据都插入到分区表 p1 中，p1 就是当前这个分区的分区表名称。

partition p2 values less than(2451544)：及要求 2451179 <=inv_date_sk<2451544 的数据都插入到分区表 p2 中，其他的分区类似。

partition p7 values less than(maxvalue)： 这个限定了 inv_date_sk 的最大值，即为 integer 类型的最大值。

表定义 2：

```
DROP TABLE IF EXISTS customer;
create table customer
(
    c_customer_sk             integer               not null,
    c_customer_id             char(16)              not null,
    c_current_cdemo_sk        integer                       ,
    c_current_hdemo_sk        integer                       ,
    c_current_addr_sk         integer                       ,
    c_first_shipto_date_sk    integer                       ,
    c_first_sales_date_sk     integer                       ,
    c_salutation              char(10)                      ,
    c_first_name              char(20)                      ,
    c_last_name               char(30)                      ,
    c_preferred_cust_flag     char(1)                       ,
    c_birth_day               integer                       ,
    c_birth_month             integer                       ,
    c_birth_year              integer                       ,
    c_birth_country           varchar(20)                   ,
    c_login                   char(13)                      ,
    c_email_address           char(50)                      ,
    c_last_review_date        char(10)
)
distribute by hash (c_customer_sk);
```

表定义 3：

```
DROP TABLE IF EXISTS web_page;
create table web_page
(
    wp_web_page_sk            integer               not null,
    wp_web_page_id            char(16)              not null,
    wp_rec_start_date         date                          ,
    wp_rec_end_date           date                          ,
    wp_creation_date_sk       integer                       ,
    wp_access_date_sk         integer                       ,
```

```
    wp_autogen_flag          char(1)                  ,
    wp_customer_sk           integer                  ,
    wp_url                   varchar(100)             ,
    wp_type                  char(50)                 ,
    wp_char_count            integer                  ,
    wp_link_count            integer                  ,
    wp_image_count           integer                  ,
    wp_max_ad_count          integer
)
distribute by replication;
```

说明：

distribute by replication 说明该表是个复制表，即每个数据节点上都会存储全量的数据。

使用\d 查看当前模式下的所有数据库对象：

```
\d
                        List of relations
 Schema |     Name      | Type  | Owner |                 Storage
--------+---------------+-------+-------+---------------------------------
 public | customer      | table | dbadmin     | {orientation=row,compression=no}
 public | inventory     | table | dbadmin     | {orientation=row,compression=no}
 public | web_page      | table | dbadmin     | {orientation=row,compression=no}
(3 rows)
```

步骤 3 导入数据

导入数据：

```
\copy customer from '/opt/gsql_input_data/customer.dat' delimiter '|';
```

导入 100000 条。

```
\copy inventory from '/opt/gsql_input_data/inventory.dat' delimiter '|';
```

导入 11745000 条（时间可能在几分钟）。

```
\copy web_page from '/opt/gsql_input_data/web_page.dat' delimiter '|';
```

导入 60 条。

说明：

- 数据文件中的数据以 '|'作为分隔符，所以在导入数据时使用 delimiter 来指定数据的分隔符。

可以使用 count(*)查看导入的数据量：

```
select count(*) from customer;
 count
--------
 100000
(1 row)

select count(*) from inventory;
   count
```

```
----------
 11745000
(1 row)

select count(*) from web_page;
 count
-------
    60
(1 row)
```

说明：其中的/opt/gsql_input_data/路径是数据文件存放的路径，执行的时候，换成自己存放数据的路径即可。

2.2.3 Explain 介绍

SQL 执行计划是一个节点数，显示执行一条 SQL 语句时的详细步骤。每一个步骤是一个数据库运算符，也叫作一个执行算子。使用 explain 命令可以查看优化器为每个查询生成的具体执行计划。

Explain 显示 SQL 语句的执行计划，支持多种选项，对选项顺序无要求，语法如下：

```
EXPLAIN [ (  option  [, ...] )  ] statement;
EXPLAIN  { [ ANALYZE ] [ VERBOSE ]  | PERFORMANCE } statement;
```

其中的 option 可以是以下选项：

ANALYZE | VERBOSE | COSTS| CPU| DETAIL | NODES |NUM_NODES |BUFFERS |TIMING |FORMAT

其中 FORMAT 的取值为 { TEXT | XML | JSON | YAML }，其余均为 bool 值，取值 on/off。

参数介绍：

介绍之前先建一个简单的表对象，用来辅助说明以下参数的含义。

创建表：

```
create table explain_option(a int, b int);
```

返回结果：

```
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'a' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
```

插入数据：

```
insert into explain_option values(1,2);
```

返回结果：

```
INSERT 0 1
```

为了更好的说明参数的含义，显示格式使用 normal 显示格式展示：

```
set explain_perf_mode=normal;
```

analyze：此参数用来控制 explain 显示执行计划时，对应的查询语句是否被执行，默认为 off，analyze 会打印出实际的执行时间、输出行数以及循环次数信息。整个手册中出现的时间信息单位都是毫秒。

```
explain analyze select count(*) from explain_option;
```

返回结果：

```
                                                QUERY PLAN
------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=14.24..14.31 rows=1 width=8) (actual time=2.948..2.948 rows=1 loops=1)
   ->  Streaming (type: GATHER)  (cost=14.24..14.31 rows=3 width=8) (actual time=2.628..2.903 rows=3 loops=1)
         Node/s: All datanodes
         ->  Aggregate  (cost=14.18..14.19 rows=3 width=8) (actual time=[0.004,0.004]..[0.012,0.012], rows=3)
               ->  Seq Scan on explain_option  (cost=0.00..14.14 rows=30 width=0) (actual time=[0.001,0.001]..[0.008,0.009], rows=1)
 Total runtime: 3.083 ms
(6 rows)
```

说明：每个算子后边跟的执行信息，第一个大括号里的是估算信息，第二个大括号里的是执行信息。也可以使用 explain （analyze on） select count(*) from explain_option;

verbose：此参数用来控制输出计划中每一层是否显示输出列，默认值为 off。

```
explain verbose select count(*) from explain_option;
```

返回结果：

```
WARNING:  Statistics in some tables or columns(public.explain_option.a) are not collected.
HINT:  Do analyze for them in order to generate optimized plan.
                                        QUERY PLAN
-------------------------------------------------------------------------------------
 Aggregate  (cost=14.24..14.31 rows=1 width=8)
   Output: pg_catalog.count(*)
   ->  Streaming (type: GATHER)  (cost=14.24..14.31 rows=3 width=8)
         Output: (count(*))
         Node/s: All datanodes
         ->  Aggregate  (cost=14.18..14.19 rows=3 width=8)
               Output: count(*)
               ->  Seq Scan on public.explain_option  (cost=0.00..14.14 rows=30 width=0)
                     Output: a, b
                     Distribute Key: a
(10 rows)
```

说明：计划中的 Output 信息就是展示的输出列信息。也可以使用 explain (verbose on) select count(*) from explain_option;

costs：用来控制是否显示计划中的估算信息，包括预估的数据宽度、输出行数以及代价信息，默认值为 on。

默认的 explain 显示的执行计划如下，带有 cost 信息。

```
explain select count(*) from explain_option;
```

返回结果：

```
QUERY PLAN
------------------------------------------------------------------------------
 Aggregate  (cost=14.24..14.31 rows=1 width=8)
   ->  Streaming (type: GATHER)  (cost=14.24..14.31 rows=3 width=8)
         Node/s: All datanodes
```

```
          ->  Aggregate  (cost=14.18..14.19 rows=3 width=8)
                ->  Seq Scan on explain_option  (cost=0.00..14.14 rows=30 width=0)
(5 rows)
```

关闭 cost 信息之后的计划如下：

explain (costs off) select count(*) from explain_option;

返回结果：

```
                  QUERY PLAN
----------------------------------------------
 Aggregate
   ->  Streaming (type: GATHER)
         Node/s: All datanodes
         ->  Aggregate
               ->  Seq Scan on explain_option
(5 rows)
```

cpu：用来控制是否输出执行的 cpu 信息，默认值为 off，此参数设置为 on 时要求 analyze 也为 on，因为只有实际执行才会统计 cpu 信息。

explain (analyze on, cpu on) select count(*) from explain_option;

返回结果：

```
                                                                  QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
------------
 Aggregate  (cost=14.24..14.31 rows=1 width=8) (actual time=2.699..2.699 rows=1 loops=1)
   (CPU: ex c/r=1066, ex row=3, ex cyc=3198, inc cyc=269899)
   ->  Streaming (type: GATHER)  (cost=14.24..14.31 rows=3 width=8) (actual time=2.380..2.667
rows=3 loops=1)
         Node/s: All datanodes
         (CPU: ex c/r=88900, ex row=3, ex cyc=266701, inc cyc=266701)
         ->  Aggregate  (cost=14.18..14.19 rows=3 width=8) (actual time=[0.004,0.004]..[0.012,0.012],
rows=3)
               CPU: max_ex c/r=459, max_ex cyc=459, max_inc cyc=1226
                    min_ex c/r=290, min_ex cyc=290, min_inc cyc=436
               ->  Seq Scan on explain_option  (cost=0.00..14.14 rows=30 width=0) (actual
time=[0.002,0.002]..[0.007,0.008], rows=1)
                     CPU: max_ex c/r=767, max_ex cyc=767, max_inc cyc=767
                          min_ex c/r=0, min_ex cyc=146, min_inc cyc=146
 Total runtime: 2.831 ms
(12 rows)
```

计划中的 CPU 字段相关的就是显示的 cpu 信息。必须指定 analyze，否则

explain (cpu on) select count(*) from explain_option;会返回

ERROR: EXPLAIN option CPU requires ANALYZE

nodes：用来控制是否输出执行的节点信息，默认值为 on。

explain select count(*) from explain_option;

返回结果：

```
                             QUERY PLAN
--------------------------------------------------------------------------------
 Aggregate  (cost=14.24..14.31 rows=1 width=8)
```

```
   ->  Streaming (type: GATHER)  (cost=14.24..14.31 rows=3 width=8)
         Node/s: All datanodes
         ->  Aggregate  (cost=14.18..14.19 rows=3 width=8)
               ->  Seq Scan on explain_option  (cost=0.00..14.14 rows=30 width=0)
(5 rows)
```

默认的 explain 都会显示 Node/s 这样的信息，用来说明计划在哪些节点上执行。关闭 nodes 之后，就不会显示了。

```
explain (nodes off) select count(*) from explain_option;
```

返回结果：

```
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate  (cost=14.24..14.31 rows=1 width=8)
   ->  Streaming (type: GATHER)  (cost=14.24..14.31 rows=3 width=8)
         ->  Aggregate  (cost=14.18..14.19 rows=3 width=8)
               ->  Seq Scan on explain_option  (cost=0.00..14.14 rows=30 width=0)
(4 rows)
```

num_nodes：用来控制是否输出执行的节点个数，默认值为 off。

```
explain (num_nodes on) select count(*) from explain_option;
```

返回结果：

```
                                       QUERY PLAN
-----------------------------------------------------------------------------------------------------
 Aggregate  (cost=14.24..14.31 rows=1 width=8)
   ->  Streaming (type: GATHER) (primary node count=0, node count=3)  (cost=14.24..14.31 rows=3
width=8)
         Node/s: All datanodes
         ->  Aggregate  (cost=14.18..14.19 rows=3 width=8)
               ->  Seq Scan on explain_option  (cost=0.00..14.14 rows=30 width=0)
(5 rows)
```

计划中的(primary node count=0, node count=3)这一部分就是由 num_nodes 选项控制的，用来显示参数运算的 datanode 的个数。

Buffers：用来控制是否输出执行时的 buffer 信息，默认值为 off，此参数和 cpu 参数类似，要求必须指定 analyze。

```
explain (buffers on, analyze on) select count(*) from explain_option;
```

返回结果：

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
------------
 Aggregate  (cost=14.24..14.31 rows=1 width=8) (actual time=2.655..2.655 rows=1 loops=1)
   ->  Streaming (type: GATHER)  (cost=14.24..14.31 rows=3 width=8) (actual time=2.355..2.627
rows=3 loops=1)
         Node/s: All datanodes
         ->  Aggregate  (cost=14.18..14.19 rows=3 width=8) (actual time=[0.005,0.005]..[0.012,0.013],
rows=3)
               Buffers: shared max hit=1
               ->  Seq Scan on explain_option  (cost=0.00..14.14 rows=30 width=0) (actual
time=[0.002,0.002]..[0.006,0.007], rows=1)
```

```
                                Buffers: shared max hit=1
 Total runtime: 2.776 ms
(8 rows)
```

计划中的 Buffers 相关的信息就是由 buffers 选项控制的，也是要求和 analyze 一起使用，选项的顺序可以随意指定。

Format：用来控制指定计划最终的输出格式，默认值为 text 格式。

默认的就是 text 格式，以上显示都是 text 格式。现在展示下 xml 格式的显示，如下：

explain (format xml) select count(*) from explain_option;
返回结果：

```
                                QUERY PLAN
---------------------------------------------------------------------
 <explain xmlns="http://www.postgresql.org/2009/explain">            +
   <Query>                                                           +
     <Plan>                                                          +
       <Node-Type>Aggregate</Node-Type>                              +
       <Strategy>Plain</Strategy>                                    +
       <Startup-Cost>14.24</Startup-Cost>                            +
       <Total-Cost>14.31</Total-Cost>                                +
       <Plan-Rows>1</Plan-Rows>                                      +
       <Plan-Width>8</Plan-Width>                                    +
       <Plans>                                                       +
         <Plan>                                                      +
           <Node-Type>Streaming (type: GATHER)</Node-Type>           +
           <Parent-Relationship>Outer</Parent-Relationship>          +
           <Startup-Cost>14.24</Startup-Cost>                        +
           <Total-Cost>14.31</Total-Cost>                            +
           <Plan-Rows>3</Plan-Rows>                                  +
           <Plan-Width>8</Plan-Width>                                +
           <Nodes>All datanodes</Nodes>                              +
           <Plans>                                                   +
             <Plan>                                                  +
               <Node-Type>Aggregate</Node-Type>                      +
               <Strategy>Plain</Strategy>                            +
               <Parent-Relationship>Outer</Parent-Relationship>      +
               <Startup-Cost>14.18</Startup-Cost>                    +
               <Total-Cost>14.19</Total-Cost>                        +
               <Plan-Rows>3</Plan-Rows>                              +
               <Plan-Width>8</Plan-Width>                            +
               <Plans>                                               +
                 <Plan>                                              +
                   <Node-Type>Seq Scan</Node-Type>                   +
                   <Parent-Relationship>Outer</Parent-Relationship>+
                   <Relation-Name>explain_option</Relation-Name>     +
                   <Alias>explain_option</Alias>                     +
                   <Startup-Cost>0.00</Startup-Cost>                 +
                   <Total-Cost>14.14</Total-Cost>                    +
```

```
                    <Plan-Rows>30</Plan-Rows>                    +
                    <Plan-Width>0</Plan-Width>                   +
                  </Plan>                                        +
                </Plans>                                         +
              </Plan>                                            +
            </Plans>                                             +
          </Plan>                                                +
        </Plans>                                                 +
      </Plan>                                                    +
    </Query>                                                     +
 </explain>
(1 row)
```

说明：默认使用 text 格式，如果有工具要求的话，则选用其他的显示格式。

detail：用来控制是否输出各个数据节点的执行信息，如果只指定 analyze 的话，输出的是各个数据节点的执行信息的汇总数据，一般都是执行信息的最大值、最小值，默认为 off。这个也要求 analyze 打开的时候生效。

```
explain (analyze on, detail on, cpu on) select count(*) from explain_option;
```

返回结果：

```
                                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=14.24..14.31 rows=1 width=8) (actual time=2.693..2.694 rows=1 loops=1)
   (CPU: ex c/r=832, ex row=3, ex cyc=2498, inc cyc=269313)
   ->  Streaming (type: GATHER)  (cost=14.24..14.31 rows=3 width=8) (actual time=2.371..2.667
rows=3 loops=1)
         Node/s: All datanodes
         (CPU: ex c/r=88938, ex row=3, ex cyc=266815, inc cyc=266815)
         ->  Aggregate  (cost=14.18..14.19 rows=3 width=8)
               dn_6001_6002 (actual time=0.005..0.006 rows=1 loops=1)
               dn_6003_6004 (actual time=0.011..0.011 rows=1 loops=1)
               dn_6005_6006 (actual time=0.005..0.005 rows=1 loops=1)
               dn_6001_6002 (CPU: ex c/r=0, ex row=0, ex cyc=336, inc cyc=531)
               dn_6003_6004 (CPU: ex c/r=416, ex row=1, ex cyc=416, inc cyc=1032)
               dn_6005_6006 (CPU: ex c/r=0, ex row=0, ex cyc=330, inc cyc=476)
               ->  Seq Scan on explain_option  (cost=0.00..14.14 rows=30 width=0)
                     dn_6001_6002 (actual time=0.002..0.002 rows=0 loops=1)
                     dn_6003_6004 (actual time=0.005..0.006 rows=1 loops=1)
                     dn_6005_6006 (actual time=0.002..0.002 rows=0 loops=1)
                     dn_6001_6002 (CPU: ex c/r=0, ex row=0, ex cyc=195, inc cyc=195)
                     dn_6003_6004 (CPU: ex c/r=616, ex row=1, ex cyc=616, inc cyc=616)
                     dn_6005_6006 (CPU: ex c/r=0, ex row=0, ex cyc=146, inc cyc=146)
 Total runtime: 2.827 ms
(20 rows)
```

说明：可以看到，detail on 的时候，将各个 datanode 上的执行时间信息、cpu 信息都展示出来了，而之前的用例没有指定 detail，显示的都是汇总信息。

Performance：这个是所有参数的集合，即使用 explain performance 的时候，所有的参数都是打开状态，format 是 text 格式。

explain performance select count(*) from explain_option;

返回结果：

```
WARNING:  Statistics in some tables or columns(public.explain_option.a) are not collected.
HINT:  Do analyze for them in order to generate optimized plan.
                                                                          QUERY PLAN

--------------------------------------------------------------------------------------------------------------------------------
----------
 Aggregate  (cost=14.24..14.31 rows=1 width=8) (actual time=2.645..2.645 rows=1 loops=1)
   Output: pg_catalog.count(*)
   (CPU: ex c/r=855, ex row=3, ex cyc=2565, inc cyc=264428)
   ->  Streaming (type: GATHER) (primary node count=0, node count=3)  (cost=14.24..14.31 rows=3 width=8) (actual time=2.295..2.619 rows=3
 loops=1)
         Output: (count(*))
         Node/s: All datanodes
         (CPU: ex c/r=87287, ex row=3, ex cyc=261863, inc cyc=261863)
         ->  Aggregate  (cost=14.18..14.19 rows=3 width=8)
               dn_6001_6002 (actual time=0.005..0.005 rows=1 loops=1)
               dn_6003_6004 (actual time=0.010..0.010 rows=1 loops=1)
               dn_6005_6006 (actual time=0.005..0.005 rows=1 loops=1)
              Output: count(*)
               dn_6001_6002 (Buffers: 0)
               dn_6003_6004 (Buffers: shared hit=1)
               dn_6005_6006 (Buffers: 0)
               dn_6001_6002 (CPU: ex c/r=0, ex row=0, ex cyc=336, inc cyc=536)
               dn_6003_6004 (CPU: ex c/r=468, ex row=1, ex cyc=468, inc cyc=1062)
               dn_6005_6006 (CPU: ex c/r=0, ex row=0, ex cyc=356, inc cyc=499)
              ->  Seq Scan on public.explain_option  (cost=0.00..14.14 rows=30 width=0)
                     dn_6001_6002 (actual time=0.002..0.002 rows=0 loops=1)
                     dn_6003_6004 (actual time=0.005..0.006 rows=1 loops=1)
                     dn_6005_6006 (actual time=0.002..0.002 rows=0 loops=1)
                    Output: a, b
                    Distribute Key: a
                     dn_6001_6002 (Buffers: 0)
                     dn_6003_6004 (Buffers: shared hit=1)
                     dn_6005_6006 (Buffers: 0)
                     dn_6001_6002 (CPU: ex c/r=0, ex row=0, ex cyc=200, inc cyc=200)
                     dn_6003_6004 (CPU: ex c/r=594, ex row=1, ex cyc=594, inc cyc=594)
                     dn_6005_6006 (CPU: ex c/r=0, ex row=0, ex cyc=143, inc cyc=143)
 Total runtime: 2.777 ms
(31 rows)
```

说明：Explain + query 并不会实际执行，只会将计划打印出来，要实际执行必须指定 analyze。explain 的各个参数都可以组合使用，除了 performance 之外。

例如：

Explain analyze statement;

Expalin verbose statement;

Explain analyze verbose statement;

Explain (analyze on, verbose on, costs off, buffers on, cpu off) statement;

Explain performance statement;

2.2.4 执行计划显示格式

GaussDB(DWS)提供了两种显示格式，使用参数 explain_perf_mode 来控制。在 Data Studio 中，该参数暂不发挥作用。

Normal 显示格式如下：

set explain_perf_mode=normal;
explain select count(*) from inventory;

结果如下：

```
hql_demo=> explain select count(*) from inventory;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------
 Aggregate  (cost=14.24..14.31 rows=1 width=8)
   ->  Streaming (type: GATHER)  (cost=14.24..14.31 rows=3 width=8)
         Node/s: All datanodes
         ->  Aggregate  (cost=14.18..14.19 rows=3 width=8)
               ->  Partition Iterator  (cost=0.00..14.14 rows=30 width=0)
                     Iterations: 7
                     ->  Partitioned Seq Scan on inventory  (cost=0.00..14.14 rows=30 width=0)
                           Selected Partitions:  1..7
(8 rows)
```

Pretty：改进后的显示格式，层次清晰，计划包含了 plan node id，性能分析会更加简单直接。当前数据库默认的显示格式。

例如：

set explain_perf_mode=pretty;
explain select count(*) from inventory;

结果如下：

```
hql_demo=> explain select count(*) from inventory;
 id |                    operation                    | E-rows | E-width | E-costs
----+-------------------------------------------------+--------+---------+---------
  1 | ->  Aggregate                                   |      1 |       8 | 14.31
  2 |     ->  Streaming (type: GATHER)                |      3 |       8 | 14.31
  3 |        ->  Aggregate                            |      3 |       8 | 14.19
  4 |           ->  Partition Iterator                |     30 |       0 | 14.14
  5 |              ->  Partitioned Seq Scan on inventory |     30 |       0 | 14.14
(5 rows)

 Predicate Information (identified by plan id)
-----------------------------------------------
   4 --Partition Iterator
         Iterations: 7
   5 --Partitioned Seq Scan on inventory
         Selected Partitions:  1..7
(4 rows)
```

可以使用 show explain_perf_mode;来查看当前数据库使用的是哪种显示格式。

show explain_perf_mode;
返回结果：

```
explain_perf_mode
-------------------
 pretty
(1 row)
```

可以使用 set explain_perf_mode=normal/pretty 来设置你想要的输出格式。

2.2.5 Explain 解析

2.2.5.1 解析执行计划

查询 Query1：select inv_warehouse_sk from customer, inventory where inv_date_sk=c_current_cdemo_sk and c_current_addr_sk <inv_item_sk limit 10;

以下内容提到查询 Query1 都代表的是此查询语句。

此处先看 normal 格式的计划。

```
set explain_perf_mode=normal;
explain select inv_warehouse_sk from customer, inventory where inv_date_sk=c_current_cdemo_sk and c_current_addr_sk <inv_item_sk limit 10;
```

执行计划如下：

```
                                                  QUERY PLAN
------------------------------------------------------------------------------------------------------------------
 Limit  (cost=15.85..31.70 rows=9 width=4)
   ->  Streaming (type: GATHER)  (cost=15.85..31.70 rows=9 width=4)
         Node/s: All datanodes
         ->  Limit  (cost=15.66..31.33 rows=9 width=4)
               ->  Hash Join  (cost=15.66..31.33 rows=10 width=4)
                     Hash Cond: (customer.c_current_cdemo_sk = inventory.inv_date_sk)
                     Join Filter: (customer.c_current_addr_sk < inventory.inv_item_sk)
                     ->  Streaming(type: REDISTRIBUTE)  (cost=0.00..15.49 rows=30 width=8)
                           Spawn on: All datanodes
                           ->  Seq Scan on customer  (cost=0.00..14.14 rows=30 width=8)
                     ->  Hash  (cost=15.49..15.49 rows=29 width=12)
                           ->  Streaming(type: REDISTRIBUTE)  (cost=0.00..15.49 rows=30 width=12)
                                 Spawn on: All datanodes
                                 ->  Partition Iterator  (cost=0.00..14.14 rows=30 width=12)
                                       Iterations: 7
                                       ->  Partitioned Seq Scan on inventory  (cost=0.00..14.14 rows=30 width=12)
                                             Selected Partitions:  1..7
(17 rows)
```

使用 explain 查看 Query1 的执行计划，该查询并未实际执行，只是将优化器生成的查询计划通过 explain 命令展示出来。所以展示的内容都是优化器生成的信息，并不是实际执行的信息，我们把这个信息叫静态信息。

处于最底层的 Seq Scan 节点是表扫描节点，它执行的时候会扫描表并返回原始数据。表的扫描方式也有多种，比如顺序扫描、索引扫描、分区扫描等。

处于扫描节点之上的节点叫作数据处理节点，比如表连接、聚集、排序。例如 HashAggregate 节点就是实现聚集功能，Hash Join 实现两个表的连接功能。

每一个节点后的(cost=15.66..31.33 rows=10 width=4)信息，是优化器在生成计划的过程中预估的执行开销。

cost=15.66..31.33 中的 15.66 代表的是启动代价，31.33 代表的是总的代价。

rows=10 代表的是预估当前节点会返回 3 条数据，

width=4 代表返回数据的宽度（根据输出字段的数据类型进行估算）。这个信息受 COSTS 控制，即当选项 costs=off 时，不显示此信息，即执行以下命令时：

```
explain (costs off) select inv_warehouse_sk from customer, inventory where
inv_date_sk=c_current_cdemo_sk and c_current_addr_sk <inv_item_sk limit 10;
```

结果如下：

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Limit
   ->  Streaming (type: GATHER)
         Node/s: All datanodes
         ->  Limit
               ->  Hash Join
                     Hash Cond: (customer.c_current_cdemo_sk = inventory.inv_date_sk)
                     Join Filter: (customer.c_current_addr_sk < inventory.inv_item_sk)
                     ->  Streaming(type: REDISTRIBUTE)
                           Spawn on: All datanodes
                           ->  Seq Scan on customer
                     ->  Hash
                           ->  Streaming(type: REDISTRIBUTE)
                                 Spawn on: All datanodes
                                 ->  Partition Iterator
                                       Iterations: 7
                                       ->  Partitioned Seq Scan on inventory
                                             Selected Partitions:  1..7
(17 rows)
```

使用 explain verbose 可以查看 Query1 每一个节点输出的字段的名称。

explain (verbose, costs off) select inv_warehouse_sk from customer, inventory where inv_date_sk=c_current_cdemo_sk and c_current_addr_sk <inv_item_sk limit 10;

结果如下：

```
                                                 QUERY PLAN
------------------------------------------------------------------------------------------------------------
Limit
  Output: inventory.inv_warehouse_sk
  ->  Streaming (type: GATHER)
        Output: inventory.inv_warehouse_sk
        Node/s: All datanodes
        ->  Limit
              Output: inventory.inv_warehouse_sk
              ->  Hash Join
                    Output: inventory.inv_warehouse_sk
                    Hash Cond: (customer.c_current_cdemo_sk = inventory.inv_date_sk)
                    Join Filter: (customer.c_current_addr_sk < inventory.inv_item_sk)
                    ->  Streaming(type: REDISTRIBUTE)
                          Output: customer.c_current_cdemo_sk, customer.c_current_addr_sk
                          Distribute Key: customer.c_current_cdemo_sk
                          Spawn on: All datanodes
                          Consumer Nodes: All datanodes
                          ->  Seq Scan on public.customer
                                Output: customer.c_current_cdemo_sk, customer.c_current_addr_sk
                                Distribute Key: customer.c_customer_sk
                    ->  Hash
                          Output: inventory.inv_warehouse_sk, inventory.inv_date_sk, inventory.inv_item_sk
                          ->  Streaming(type: REDISTRIBUTE)
                                Output: inventory.inv_warehouse_sk, inventory.inv_date_sk, inventory.inv_item_sk
                                Distribute Key: inventory.inv_date_sk
                                Spawn on: All datanodes
                                Consumer Nodes: All datanodes
                                ->  Partition Iterator
                                      Output: inventory.inv_warehouse_sk, inventory.inv_date_sk, inventory.inv_item_sk
                                      Iterations: 7
                                      ->  Partitioned Seq Scan on public.inventory
                                            Output: inventory.inv_warehouse_sk, inventory.inv_date_sk, inventory.inv_item_sk
                                            Distribute Key: inventory.inv_item_sk
                                            Selected Partitions:  1..7
(33 rows)
```

其中每一个节点下边的 output 信息，就是当前节点要向上层返回的字段的个数以及名称。

Node/s 信息受选项 nodes 控制，指定为 off 的时候不显示，默认为 on。

explain (nodes off) select inv_warehouse_sk from customer, inventory where inv_date_sk=c_current_cdemo_sk and c_current_addr_sk <inv_item_sk limit 10;

```
                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
Limit  (cost=15.85..31.70 rows=9 width=4)
  ->  Streaming (type: GATHER)  (cost=15.85..31.70 rows=9 width=4)
        ->  Limit  (cost=15.66..31.33 rows=9 width=4)
              ->  Hash Join  (cost=15.66..31.33 rows=10 width=4)
                    Hash Cond: (customer.c_current_cdemo_sk = inventory.inv_date_sk)
                    Join Filter: (customer.c_current_addr_sk < inventory.inv_item_sk)
                    ->  Streaming(type: REDISTRIBUTE)  (cost=0.00..15.49 rows=30 width=8)
                          ->  Seq Scan on customer  (cost=0.00..14.14 rows=30 width=8)
                    ->  Hash  (cost=15.49..15.49 rows=29 width=12)
                          ->  Streaming(type: REDISTRIBUTE)  (cost=0.00..15.49 rows=30 width=12)
                                ->  Partition Iterator  (cost=0.00..14.14 rows=30 width=12)
                                      Iterations: 7
                                      ->  Partitioned Seq Scan on inventory  (cost=0.00..14.14 rows=30 width=12)
                                            Selected Partitions:  1..7
(14 rows)
```

2.2.5.2 执行节点介绍

执行计划中的每一个节点，我们也称作算子(operator)。根据功能，可以分为以下几类：

1. 表访问方式

Seq Scan：全表顺序扫描，即执行时要将表中所有的数据都扫出来，然后再根据过滤条件进行过滤，返回给上层算子的数据就是过滤过的数据。

Index Scan：索引扫描，如果选择条件涉及的属性上建立了索引，则生成的查询计划中涉及表的扫描时会使用索引扫面。

Partitioned Seq Scan： 针对分区表的顺序扫描，每次只扫描一个分区。

Partition Iterator：此算子是用来做分区切换的，下层算子就是 Partitioned Seq Scan。

列存扫描：基表是列存表，就会使用列扫描。

2. 表连接方式

在介绍连接方式之前，先介绍下连接中所说的内外表。

```
 4 |           ->  Hash Join (5,8)
 5 |              ->  Streaming(type: BROADCAST)
 6 |                 ->  Partition Iterator
 7 |                    ->  Partitioned Seq Scan on public.inventory
 8 |              ->  Hash
 9 |                 ->  Seq Scan on public.customer
```

每一种连接方式都是两个表之间的连接，我们把需要连接的两个表称为内外表，其中第一个是外表，第二个是内表。即这个例子中的 5 号算子就是 hashjoin 算子的外表（也叫左子树），8 号算子是内表（也叫右子树）。

```
 4 |           ->  Nested Loop (5,7)
 5 |              ->  Partition Iterator
 6 |                 ->  Partitioned Seq Scan on public.inventory
 7 |              ->  Materialize
 8 |                 ->  Streaming(type: BROADCAST)
 9 |                    ->  Seq Scan on public.customer
```

大家可以看下这个计划的左右子树都是什么？（其实已经很明确了，5 和 7 分别代表了左右子树，pretty 模式的显示可以让我们很快找到它的左右子树哦。）

Nested Loop：嵌套循环，适用于被连接的数据量较小的查询。嵌套循环，即外表驱动内表，外表返回的每一行都要在内表中检索和它匹配的行。因此整个查询返回的结果集不能太大，要把返回子集较小的作为外表，而且在内表的连接字段上建议要有索引。

Hash Join：哈希连接，适用于数据量大的表的连接方式。优化器使用两个表中较小的表，利用连接键（就是 Query1 中的 t1.salary=t2.salary）在内存中建立 hash 表（即将内表建 hash 表），然后扫描较大的表并探测散列，找到与散列匹配的行。

Merge Join：归并连接，实现了对排序关系的归并连接算法，归并连接的输入都是已经排序好的。

3. 聚集操作

Hashagg: 对下层未排序的数据使用散列的形式进行分组，并进行计算。

Sortagg: 下层节点提供排序好的数据，然后进行分组计算。

Windowagg：用于处理窗口函数，即处理与当前元组相关的一组元组上的执行函数。

4. 集合操作：

Setop：用于处理集合操作。

5. 物化操作：

Materialize：将下层算子返回数据都缓存起来，主要用于 nestloop，当再次访问的时候直接从缓存中取数据，不用再找下层算子要数据了。

Hash：这个算子是给 hashjoin 算子使用的，hashjoin 要将右子树的数据全部存到 hash 表里，此算子就是将数据写入到 hash 表的处理算子。

6. 数据处理

Sort：用于对下层节点的输出结果进行排序。

Limit：限定结果输出的行数。

7. stream

Streaming (type: GATHER)：用来做数据的汇总，将各个 datanode 的数据收集到 coordinator 上。

Streaming(type: BROADCAST)：数据广播，即把数据给每个 datanode 上都发一份。

Streaming(type: REDSITRIBUTE)：数据分布，将数据根据分布列信息，计算哈希值，分布到不同的 datanode 上。

2.2.5.3 执行信息介绍

实际执行的信息，因为 pretty 显示模式相对于 normal 模式，会有更多的信息用来定位性能问题，所以我们 set explain_perf_mode=pretty。

使用 explain performance 执行 query1。

```
set explain_perf_mode=pretty;
explain performance   select inv_warehouse_sk from customer, inventory where
inv_date_sk=c_current_cdemo_sk and c_current_addr_sk <inv_item_sk limit 10;
```

说明：执行的结果集太大了，可以重定向到文件里，使用

```
\o explain.out
explain performance   select inv_warehouse_sk from customer, inventory where
inv_date_sk=c_current_cdemo_sk and c_current_addr_sk <inv_item_sk limit 10;
\o
```

即在当前的目录下会生成一个 explain.out 的文件，用来存储执行结果。

以下截取部分数据来进行说明：

```
 id |                              operation                               |       A-time        |  A-rows  | E-rows | E-distinct |  Peak Me
mory   | A-width | E-width | E-costs
----+----------------------------------------------------------------------+---------------------+----------+--------+------------+---------
-------+---------+---------+---------
  1 | ->  Limit                                                            | 4660.699            |        0 |      9 |            | 1KB
     |         |       4 | 31.70
  2 |    ->  Streaming (type: GATHER)                                      | 4660.693            |        0 |      9 |            | 76KB
     |         |       4 | 31.70
  3 |       ->  Limit                                                      | [4620.563,4639.705] |        0 |      9 |            | [1KB, 1K
B]   |         |       4 | 31.33
  4 |          ->  Hash Join (5,7)                                         | [4620.561,4639.704] |        0 |     10 | 10, 10     | [5KB, 5K
B]   |         |       4 | 31.33
  5 |            ->  Streaming(type: REDISTRIBUTE)                         | [7.484,13.462]      |   100000 |     30 |            | [67KB, 6
7KB] |         |       8 | 15.49
  6 |               ->  Seq Scan on public.customer                        | [11.891,12.697]     |   100000 |     30 |            | [12KB, 1
2KB] |         |       8 | 14.14
  7 |            ->  Hash                                                  | [4563.702,4570.599] | 11745000 |     29 |            | [157MB,
200MB] | [28,28] |      12 | 15.49
  8 |               ->  Streaming(type: REDISTRIBUTE)                      | [1965.775,2528.185] | 11745000 |     30 |            | [67KB, 6
7KB] |         |      12 | 15.49
  9 |                  ->  Partition Iterator                              | [1616.627,1689.279] | 11745000 |     30 |            | [9KB, 9K
B]   |         |      12 | 14.14
 10 |                     ->  Partitioned Seq Scan on public.inventory     | [994.989,1054.835]  | 11745000 |     30 |            | [3KB, 3K
B]   |         |      12 | 14.14
(10 rows)

              Predicate Information (identified by plan id)
-------------------------------------------------------------------------
   4 --Hash Join (5,7)
         Hash Cond: (customer.c_current_cdemo_sk = inventory.inv_date_sk)
         Join Filter: (customer.c_current_addr_sk < inventory.inv_item_sk)
   9 --Partition Iterator
         Iterations: 7
  10 --Partitioned Seq Scan on public.inventory
         Selected Partitions:  1..7
(7 rows)
```

```
                                        Memory Information (identified by plan id)
-----------------------------------------------------------------------------------------------------------------------------------------------
 Coordinator Query Peak Memory:
         Query Peak Memory: 1MB
 DataNode Query Peak Memory
         dn_6001_6002 Query Peak Memory: 162MB
         dn_6003_6004 Query Peak Memory: 187MB
         dn_6005_6006 Query Peak Memory: 205MB
   1 --Limit
         Peak Memory: 1KB, Estimate Memory: 2048MB
   2 --Streaming (type: GATHER)
         Peak Memory: 76KB, Estimate Memory: 2048MB
   3 --Limit
         dn_6001_6002 Peak Memory: 1KB, Estimate Memory: 2048MB
         dn_6003_6004 Peak Memory: 1KB, Estimate Memory: 2048MB
         dn_6005_6006 Peak Memory: 1KB, Estimate Memory: 2048MB
   4 --Hash Join (5,7)
         dn_6001_6002 Peak Memory: 5KB, Estimate Memory: 2048MB
         dn_6003_6004 Peak Memory: 5KB, Estimate Memory: 2048MB
         dn_6005_6006 Peak Memory: 5KB, Estimate Memory: 2048MB
   5 --Streaming(type: REDISTRIBUTE)
         dn_6001_6002 Peak Memory: 67KB, Estimate Memory: 2048MB
         dn_6003_6004 Peak Memory: 67KB, Estimate Memory: 2048MB
         dn_6005_6006 Peak Memory: 67KB, Estimate Memory: 2048MB
         dn_6001_6002 Stream Network: 722kB, Network Poll Time: 5.369; Data Deserialize Time: 2.050
         dn_6003_6004 Stream Network: 662kB, Network Poll Time: 0.089; Data Deserialize Time: 1.901
         dn_6005_6006 Stream Network: 667kB, Network Poll Time: 3.629; Data Deserialize Time: 1.916
   6 --Seq Scan on public.customer
         dn_6001_6002 Peak Memory: 12KB, Estimate Memory: 2048MB
         dn_6003_6004 Peak Memory: 12KB, Estimate Memory: 2048MB
         dn_6005_6006 Peak Memory: 12KB, Estimate Memory: 2048MB
         dn_6001_6002 Stream Send time: 4566.888, Wait Quota time: 4565.751, OS Kernel Send time: 1.072; Data Serialize time: 9.209
         dn_6003_6004 Stream Send time: 4567.877, Wait Quota time: 4566.412, OS Kernel Send time: 1.375; Data Serialize time: 9.395
         dn_6005_6006 Stream Send time: 4566.538, Wait Quota time: 4565.043, OS Kernel Send time: 1.434; Data Serialize time: 8.871
```

```
   7 --Hash
         dn_6001_6002 Peak Memory: 161209KB, Width: 28
         dn_6003_6004 Peak Memory: 186627KB, Width: 28
         dn_6005_6006 Peak Memory: 205691KB, Width: 28
         dn_6001_6002  Buckets: 32768  Batches: 1  Memory Usage: 146954kB
         dn_6003_6004  Buckets: 32768  Batches: 1  Memory Usage: 170157kB
         dn_6005_6006  Buckets: 32768  Batches: 1  Memory Usage: 187559kB
   8 --Streaming(type: REDISTRIBUTE)
         dn_6001_6002 Peak Memory: 67KB, Estimate Memory: 2048MB
         dn_6003_6004 Peak Memory: 67KB, Estimate Memory: 2048MB
         dn_6005_6006 Peak Memory: 67KB, Estimate Memory: 2048MB
         dn_6001_6002 Stream Network: 88115kB, Network Poll Time: 1723.753; Data Deserialize Time: 244.478
         dn_6003_6004 Stream Network: 102028kB, Network Poll Time: 1221.570; Data Deserialize Time: 294.598
         dn_6005_6006 Stream Network: 112462kB, Network Poll Time: 922.751; Data Deserialize Time: 312.369
   9 --Partition Iterator
         dn_6001_6002 Peak Memory: 9KB, Estimate Memory: 2048MB
         dn_6003_6004 Peak Memory: 9KB, Estimate Memory: 2048MB
         dn_6005_6006 Peak Memory: 9KB, Estimate Memory: 2048MB
         dn_6001_6002 Stream Send time: 461.155, Wait Quota time: 346.693, OS Kernel Send time: 109.529; Data Serialize time: 1359.714
         dn_6003_6004 Stream Send time: 315.390, Wait Quota time: 208.671, OS Kernel Send time: 102.429; Data Serialize time: 1321.520
         dn_6005_6006 Stream Send time: 504.304, Wait Quota time: 408.638, OS Kernel Send time: 91.195; Data Serialize time: 1298.929
  10 --Partitioned Seq Scan on public.inventory
         dn_6001_6002 Peak Memory: 3KB, Estimate Memory: 2048MB
         dn_6003_6004 Peak Memory: 3KB, Estimate Memory: 2048MB
         dn_6005_6006 Peak Memory: 3KB, Estimate Memory: 2048MB
(57 rows)
```

```
                    Targetlist Information (identified by plan id)
--------------------------------------------------------------------------------
   1 --Limit
         Output: inventory.inv_warehouse_sk
   2 --Streaming (type: GATHER)
         Output: inventory.inv_warehouse_sk
         Node/s: All datanodes
   3 --Limit
         Output: inventory.inv_warehouse_sk
   4 --Hash Join (5,7)
         Output: inventory.inv_warehouse_sk
   5 --Streaming(type: REDISTRIBUTE)
         Output: customer.c_current_cdemo_sk, customer.c_current_addr_sk
         Distribute Key: customer.c_current_cdemo_sk
         Spawn on: All datanodes
         Consumer Nodes: All datanodes
   6 --Seq Scan on public.customer
         Output: customer.c_current_cdemo_sk, customer.c_current_addr_sk
         Distribute Key: customer.c_customer_sk
   7 --Hash
         Output: inventory.inv_warehouse_sk, inventory.inv_date_sk, inventory.inv_item_sk
   8 --Streaming(type: REDISTRIBUTE)
         Output: inventory.inv_warehouse_sk, inventory.inv_date_sk, inventory.inv_item_sk
         Distribute Key: inventory.inv_date_sk
         Spawn on: All datanodes
         Consumer Nodes: All datanodes
   9 --Partition Iterator
         Output: inventory.inv_warehouse_sk, inventory.inv_date_sk, inventory.inv_item_sk
  10 --Partitioned Seq Scan on public.inventory
         Output: inventory.inv_warehouse_sk, inventory.inv_date_sk, inventory.inv_item_sk
         Distribute Key: inventory.inv_item_sk
(29 rows)
```

```
                    Datanode Information (identified by plan id)
--------------------------------------------------------------------------------
   1 --Limit
        (actual time=4660.699..4660.699 rows=0 loops=1)
        (CPU: ex c/r=0, ex row=0, ex cyc=406, inc cyc=466062499)
   2 --Streaming (type: GATHER)
        (actual time=4660.693..4660.693 rows=0 loops=1)
        (Buffers: 0)
        (CPU: ex c/r=0, ex row=0, ex cyc=466062093, inc cyc=466062093)
   3 --Limit
        dn_6001_6002 (actual time=4622.637..4622.637 rows=0 loops=1)
        dn_6003_6004 (actual time=4620.563..4620.563 rows=0 loops=1)
        dn_6005_6006 (actual time=4639.705..4639.705 rows=0 loops=1)
        dn_6001_6002 (CPU: ex c/r=0, ex row=0, ex cyc=166, inc cyc=462261528)
        dn_6003_6004 (CPU: ex c/r=0, ex row=0, ex cyc=143, inc cyc=462049116)
        dn_6005_6006 (CPU: ex c/r=0, ex row=0, ex cyc=157, inc cyc=463966351)
   4 --Hash Join (5,7)
        dn_6001_6002 (actual time=4622.636..4622.636 rows=0 loops=1)
        dn_6003_6004 (actual time=4620.561..4620.561 rows=0 loops=1)
        dn_6005_6006 (actual time=4639.704..4639.704 rows=0 loops=1)
        dn_6001_6002 (Buffers: 0)
        dn_6003_6004 (Buffers: 0)
        dn_6005_6006 (Buffers: 0)
        dn_6001_6002 (CPU: ex c/r=1, ex row=3455891, ex cyc=4719567, inc cyc=462261362)
        dn_6003_6004 (CPU: ex c/r=1, ex row=3991930, ex cyc=4407695, inc cyc=462048973)
        dn_6005_6006 (CPU: ex c/r=1, ex row=4397179, ex cyc=6605352, inc cyc=463966194)
   5 --Streaming(type: REDISTRIBUTE)
        dn_6001_6002 (actual time=0.083..13.462 rows=35891 loops=1)
        dn_6003_6004 (actual time=0.068..7.484 rows=31930 loops=1)
        dn_6005_6006 (actual time=0.074..11.115 rows=32179 loops=1)
        dn_6001_6002 (Buffers: 0)
        dn_6003_6004 (Buffers: 0)
        dn_6005_6006 (Buffers: 0)
        dn_6001_6002 (CPU: ex c/r=32, ex row=35891, ex cyc=1173790, inc cyc=1173790)
        dn_6003_6004 (CPU: ex c/r=18, ex row=31930, ex cyc=588473, inc cyc=588473)
```

```
  6 --Seq Scan on public.customer
        dn_6001_6002 (actual time=0.008..12.697 rows=33581 loops=1)
        dn_6003_6004 (actual time=0.010..12.578 rows=33357 loops=1)
        dn_6005_6006 (actual time=0.006..11.891 rows=33062 loops=1)
        dn_6001_6002 (Buffers: shared hit=966)
        dn_6003_6004 (Buffers: shared hit=959)
        dn_6005_6006 (Buffers: shared hit=951)
        dn_6001_6002 (CPU: ex c/r=32, ex row=33581, ex cyc=1081818, inc cyc=1081818)
        dn_6003_6004 (CPU: ex c/r=32, ex row=33357, ex cyc=1080064, inc cyc=1080064)
        dn_6005_6006 (CPU: ex c/r=30, ex row=33062, ex cyc=1012684, inc cyc=1012684)
  7 --Hash
        dn_6001_6002 (actual time=4563.702..4563.702 rows=3420000 loops=1)
        dn_6003_6004 (actual time=4570.599..4570.599 rows=3960000 loops=1)
        dn_6005_6006 (actual time=4564.130..4564.130 rows=4365000 loops=1)
        dn_6001_6002 (Buffers: 0)
        dn_6003_6004 (Buffers: 0)
        dn_6005_6006 (Buffers: 0)
        dn_6001_6002 (CPU: ex c/r=64, ex row=3420000, ex cyc=220457196, inc cyc=456368005)
        dn_6003_6004 (CPU: ex c/r=65, ex row=3960000, ex cyc=259023059, inc cyc=457052805)
        dn_6005_6006 (CPU: ex c/r=64, ex row=4365000, ex cyc=282535719, inc cyc=456408882)
  8 --Streaming(type: REDISTRIBUTE)
        dn_6001_6002 (actual time=3.305..2528.185 rows=3420000 loops=1)
        dn_6003_6004 (actual time=56.770..2187.582 rows=3960000 loops=1)
        dn_6005_6006 (actual time=87.945..1965.775 rows=4365000 loops=1)
        dn_6001_6002 (Buffers: 0)
        dn_6003_6004 (Buffers: 0)
        dn_6005_6006 (Buffers: 0)
        dn_6001_6002 (CPU: ex c/r=68, ex row=3420000, ex cyc=235910809, inc cyc=235910809)
        dn_6003_6004 (CPU: ex c/r=50, ex row=3960000, ex cyc=198029746, inc cyc=198029746)
        dn_6005_6006 (CPU: ex c/r=39, ex row=4365000, ex cyc=173873163, inc cyc=173873163)
  9 --Partition Iterator
        dn_6001_6002 (actual time=0.020..1689.279 rows=3967160 loops=1)
        dn_6003_6004 (actual time=0.020..1616.627 rows=3849385 loops=1)
        dn_6005_6006 (actual time=0.018..1627.817 rows=3928455 loops=1)
        dn_6001_6002 (CPU: ex c/r=16, ex row=3967160, ex cyc=64639467, inc cyc=148872993)
        dn_6003_6004 (CPU: ex c/r=16, ex row=3849385, ex cyc=62645516, inc cyc=141931375)
        dn_6005_6006 (CPU: ex c/r=16, ex row=3928455, ex cyc=63392416, inc cyc=143121934)
```

```
  10 --Partitioned Seq Scan on public.inventory
        dn_6001_6002 (actual time=0.058..1054.835 rows=3967160 loops=7)
        dn_6003_6004 (actual time=0.056..994.989 rows=3849385 loops=7)
        dn_6005_6006 (actual time=0.055..1003.106 rows=3928455 loops=7)
        dn_6001_6002 (Buffers: shared hit=21682)
        dn_6003_6004 (Buffers: shared hit=21039)
        dn_6005_6006 (Buffers: shared hit=21470)
        dn_6001_6002 (CPU: ex c/r=21, ex row=3967160, ex cyc=84233526, inc cyc=84233526)
        dn_6003_6004 (CPU: ex c/r=20, ex row=3849385, ex cyc=79285859, inc cyc=79285859)
        dn_6005_6006 (CPU: ex c/r=20, ex row=3928455, ex cyc=79729518, inc cyc=79729518)
(81 rows)

                    User Define Profiling
---------------------------------------------------------------
 Segment Id: 3  Track name: Datanode build connection
       dn_6001_6002 (time=0.465 total_calls=1 loops=1)
       dn_6003_6004 (time=0.461 total_calls=1 loops=1)
       dn_6005_6006 (time=0.394 total_calls=1 loops=1)
 Plan Node id: 2  Track name: coordinator get datanode connection
       cn_5001: (time=0.013 total_calls=1 loops=1)
(6 rows)

                        ====== Query Summary =====
-------------------------------------------------------------------------------
 Datanode executor start time [dn_6005_6006, dn_6003_6004]: [1.019 ms,1.112 ms]
 Datanode executor end time [dn_6003_6004, dn_6005_6006]: [0.123 ms,0.144 ms]
 Remote query poll time: 4658.621 ms, Deserialze time: 0.000 ms
 Coordinator executor start time: 0.101 ms
 Coordinator executor run time: 4660.718 ms
 Coordinator executor end time: 0.105 ms
 Total network : 304653kB
 Planner runtime: 5.631 ms
 Query Id: 146929937843407343
 Total runtime: 4660.975 ms
(10 rows)
```

图中显示的执行信息分为以下 7 部分：

第一部分：以表格的形式将计划显示出来，包含的字段介绍如下：

a) Id：算子编号，从执行计划的顶层开始，按照从上往下，先左后右的原则编号

b) Operation：实际的执行计划

c) A-time：当前算子的执行时间，一般 DN 上执行的算子的 A-time 是由[]括起来的两个值，分别表示此算子在所有 DN 上执行的最短时间和最长时间。CN 上的就是当前算子的执行时间。

d) A-rows：表示当前算子实际输出的数据行数。

e) E-rows：预估的数据行数（优化器产生的），这个输出受 costs 的控制，即 costs 为 off 的时候不显示。

f) E-distinct：预估的 distinct 值。

g) Peak Memory：当前算子执行中使用的峰值内存。

h) E-memory：预估当前算子需要使用的内存（只有 datanode 上执行的算子有），这个输出受 costs 的控制，即 costs 为 off 的时候不显示。

i) A-width：表示当前算子输出元组的实际宽度。

j) E-width：预估的宽度（优化器产生的），和 normal 格式下的 width 信息是一样的。这个输出受 costs 的控制，即 costs 为 off 的时候不显示。

k) E-costs：优化器估算的代价信息，这个输出受 costs 的控制，即 costs 为 off 的时候不显示。

第二部分：Predicate Information (identified by plan id)：静态信息，主要是连接算子的连接信息和过滤信息。

第三部分：Memory Information (identified by plan id)：内存信息，显示的是整个计划中内存的使用情况。每个算子都有的峰值内存（peak memory），控制内存（control memory），估算内存（estimate memory）。对于涉及到数据缓存的几个算子，还有执行时的实际宽度（width），内存使用自动扩展次数（auto spread num），是否提前下盘以及下盘文件数。

第四部分：Targetlist Information (identified by plan id)：每个算子的输出列。

第五部分：Datanode Information (identified by plan id)：各个算子实际的执行时间、CPU、buffer 信息。

第六部分：User Define Profiling：显示的是分布式执行时 CN 和 DN、DN 和 DN 之间的建连时间，以及存储层的一些执行信息，主要用于性能问题定位时使用。

第七部分：====== Query Summary =====：汇总信息，打印总的执行时间和网络流量，包括了各个 DN 上初始化和结束阶段的最大最小执行时间、CN 上的初始化、执行、结束阶段的时间，以及当前查询执行时系统可用内存、语句估算内存等信息。

针对以上的七部分内容，现在重点介绍下内存和实际执行时间部分的内容。

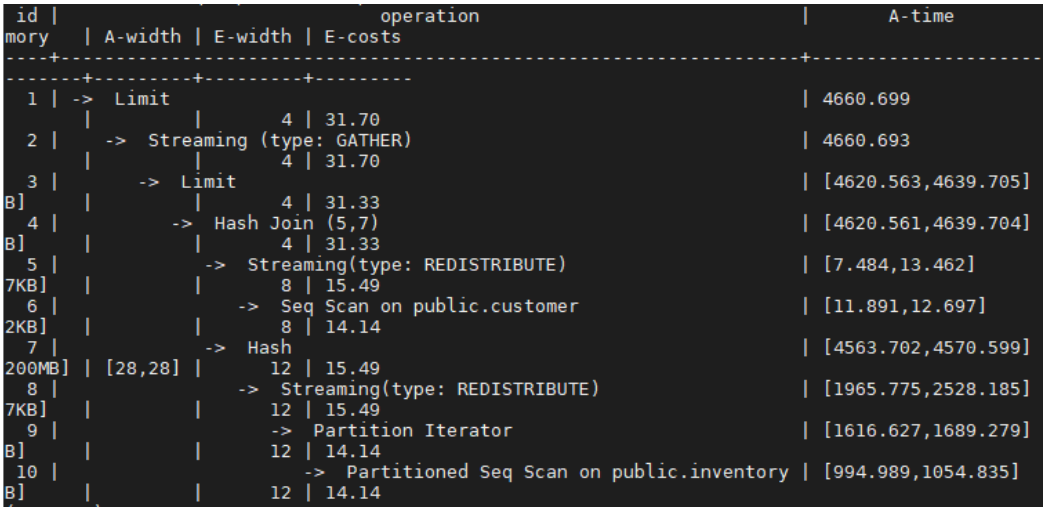
步骤 1　执行时间介绍

每个算子的时间执行信息都包含三部分。

```
dn_6001_6002 (actual time=0.008..12.697 rows=33581 loops=1)
dn_6003_6004 (actual time=0.010..12.578 rows=33357 loops=1)
```

前边的 dn_6001_6002 表示当前显示的是名字为 dn_6001_6002 的数据节点，括号里边显示的是实际的执行信息，其中 actual time=0.008..12.697 表示实际的执行时间，第一个数字 0.008 表示的是执行时从进入当前算子到输出第一条数据所花费的时间，12.697 表示的是输出所有数据的总的执行时间。

注意：整个计划中，除了叶子节点的执行时间是算子本身的执行时间，其余算子的执行时间都包含了其子节点的执行时间。



```
 id |                                operation                                |       A-time
mory   | A-width | E-width | E-costs
----+-------------------------------------------------------------------------+--------------------
-------+---------+---------+---------
  1 | ->  Limit                                                              | 4660.699
    |             |       4 | 31.70
  2 |   ->  Streaming (type: GATHER)                                         | 4660.693
    |             |       4 | 31.70
  3 |     ->  Limit                                                          | [4620.563,4639.705]
B]    |             |       4 | 31.33
  4 |       ->  Hash Join (5,7)                                              | [4620.561,4639.704]
B]    |             |       4 | 31.33
  5 |         ->  Streaming(type: REDISTRIBUTE)                              | [7.484,13.462]
7KB]  |             |       8 | 15.49
  6 |           ->  Seq Scan on public.customer                              | [11.891,12.697]
2KB]  |             |       8 | 14.14
  7 |         ->  Hash                                                       | [4563.702,4570.599]
200MB] | [28,28] |      12 | 15.49
  8 |           ->  Streaming(type: REDISTRIBUTE)                            | [1965.775,2528.185]
7KB]  |             |      12 | 15.49
  9 |             ->  Partition Iterator                                     | [1616.627,1689.279]
B]    |             |      12 | 14.14
 10 |               ->  Partitioned Seq Scan on public.inventory             | [994.989,1054.835]
B]    |             |      12 | 14.14
```

这个计划中的 6 号和 10 号是叶子节点，其余的都非叶子节点，1 号算子是顶层节点，它的执行时间就可以作为整个查询的执行时间。

怎么判断当前节点是否是叶子节点呢?

执行计划中的每个->代表一个算子，下层算子都会往右边缩进几个空格，如果没有下层算子了，就代表当前算子是叶子节点。像这个计划中的 6 号和 10 号算子，因为其下层没有向右偏移的算子信息，所以就是叶子节点了。

顶层节点其实就是整个执行计划的最上层节点，也就是 1 号节点，因为这个算子是整个执行计划的执行入口。

所以在分析性能问题时，中间算子的实际执行时间是要减去下层算子的执行时间的。

rows=33581 表示当前算子输出数据 33581 行；

loops=1 表示当前算子的执行只做了一次。对于分区表来说，这个 loops 为什么是 7 呢?

dn_6001_6002 (actual time=0.058..1054.835 rows=3967160 loops=7)
dn_6003_6004 (actual time=0.056..994.989 rows=3849385 loops=7)

我们说的一次操作，指的是从进入到当前算子直到最后一个元组返回，这整个叫一次操作。但是对于分区表来说，对每一个分区表的扫描就是一次完整的扫面操作，当切换到下一个分区的时候，又是一次新的查询操作。执行以下命令查看分区数。

\d+ inventory

Table "public.inventory"

Column	Type	Modifiers	Storage	Stats target	Description
inv_date_sk	integer	not null	plain		
inv_item_sk	integer	not null	plain		

```
 inv_warehouse_sk       | integer | not null   | plain    |                  |
 inv_quantity_on_hand | integer |                | plain    |                  |
Range partition by(inv_date_sk)
Number of partition: 7 (View pg_partition to check each partition range.)
Has OIDs: no
Distribute By: HASH(inv_item_sk)
Location Nodes: ALL DATANODES
Options: orientation=row, compression=no
```

Inventory 表有 7 个分区，所以就执行了 7 次表扫描操作，所以 loops=7。

步骤 2　CPU 信息介绍

```
dn_6001_6002 (CPU: ex c/r=32, ex row=33581, ex cyc=1081818, inc cyc=1081818)
dn_6003_6004 (CPU: ex c/r=32, ex row=33357, ex cyc=1080064, inc cyc=1080064)
```

每个算子执行的过程都有 cpu 信息，其中的 cyc 代表的是 cpu 周期数，ex cyc 表示的是当前算子的周期数，不包含其子节点。inc cyc 是包含了子节点的周期数，ex row 是当前算子输出的数据行数，ex c/r 则是 ex cyc/ex row 得到的平均值。

说明：

这里说的 cpu 周期数即 cpu 时钟周期数，是 CPU 工作的最小时间单位。时钟周期表示了运行的最高频率，时钟周期越小工作频率越高。1/时钟周期=工作频率，时钟周期的单位为纳秒。

步骤 3　Buffer 信息介绍

```
dn_6001_6002 (Buffers: shared hit=21682)
dn_6003_6004 (Buffers: shared hit=21039)
```

buffers 显示缓冲区信息，包括共享块和临时块读和写。

共享块包含表和索引，临时块在排序和物化中使用的磁盘块。上层节点显示出来的块数包含了其所有子节点使用的块数。

Buffers 涉及的参数有 2 个 shared 和 temp，即 shared hit/read/dirtied/written，temp read/write。

Hit blocks 代表的是从磁盘里面读到的数据块数。

Dirtied blocks 表示的是当前查询中被修改了的并且此前未被修改的数据块数。

Written blocks 表示当期线程将 shared buffer 里的被修改的数据写回到磁盘的块数。

步骤 4　执行内存

每个算子都有的内存信息：

dn_6001_6002 Peak Memory: 3KB, Estimate Memory: 2048MB

dn_6003_6004 Peak Memory: 3KB, Estimate Memory: 2048MB

其中：

Peak Memory: 3KB 表示当前算子实际执行时使用的峰值内存。

Estimate Memory: 2048MB 是预估的内存，这个是优化器给的预估值。

峰值内存是算子在执行的过程中使用的最大内存，而预估的内存则是优化器根据预估行数和宽度计算的一个预估值，只是用来做参考的。

对于涉及到数据缓存的算子，预估内存是用来控制当前算子的内存使用的，另外还有 Width: 12 信息，代表的是数据的实际宽度（因为对于字符串类型的数据，它的长度是不确定的，只能实际执行的时候计算）。

对于每个数据库节点：

```
Coordinator Query Peak Memory:
        Query Peak Memory: 1MB
 DataNode Query Peak Memory
        dn_6001_6002 Query Peak Memory: 162MB
        dn_6003_6004 Query Peak Memory: 187MB
        dn_6005_6006 Query Peak Memory: 205MB
```

这一部分显示的是每个数据节点上整个执行过程中实际使用的内存情况。这个内存包括了执行当前查询所使用的查询解析、优化、执行以及事务操作等所有内存的总和。

步骤 5 其他执行信息

对于涉及到数据缓存的算子，对内存的使用会比较大，所以会额外的显示这些算子使用内存的情况。

Sort 算子：

如：explain performance select inv_warehouse_sk from inventory order by 1;

会看到排序信息：

```
dn_6001_6002 Sort Method: quicksort  Memory: 284265kB
dn_6003_6004 Sort Method: quicksort  Memory: 278744kB
```

Sort Method 代表的是排序方法，有两种：快排和外排。快排即内存够用，所有的排序操作在内存中完成，外排则说明当前可用内存不够，需要下盘。下盘即将内存中的数据写到磁盘中，使用时再从磁盘上读到内存中。执行中会根据当前算子的 Estimate Memory 的大小和数据的宽度来计算内存中可以存放数据的行数（Estimate Memory/width），即阈值。当放入内存的数据超过这个阈值，就会发生下盘操作。

Memory: 则是说明排序使用的内存大小。

Hashjoin 算子：

如：explain performance select count(*) from web_page t1 join web_page t2 on t1.wp_creation_date_sk=t2.wp_creation_date_sk;

计划中的 hash 算子下会有以下信息：

```
dn_6005_6006  Buckets: 32768  Batches: 1  Memory Usage: 3kB
```

Buckets：代表 hash 表中实际使用的桶的个数。

Batches：代表 hashjoin 中实际分块的数量。如果 Batches=1，则说明所有的数据全在内存中，没有下盘操作；如果 Batches>1，则说明有数据下盘，Batches 代表的就是下盘所使用的临时文件的个数。

Memory Usage：就是 hashjoin 中内存的使用情况。

Hashagg 算子：

如果发生数据下盘，会有 File Num: 512 信息，即会显示临时文件的个数。

Stream 算子：

dn_6001_6002 Stream Network: 722kB, Network Poll Time: 5.369; Data Deserialize Time: 2.050
dn_6003_6004 Stream Network: 662kB, Network Poll Time: 0.089; Data Deserialize Time: 1.901

stream 算子会统计当前算子处理数据的字节数，其从子线程获取数据的时间(poll time)以及处理数据的时间(Deserialize Time)。

stream 算子的子节点会统计发送端的时间信息，如下：

dn_6001_6002 Stream Send time: 4566.888, Wait Quota time: 4565.751, OS Kernel Send time: 1.072; Data Serialize time: 9.209
dn_6003_6004 Stream Send time: 4567.877, Wait Quota time: 4566.412, OS Kernel Send time: 1.375; Data Serialize time: 9.395

发送时间(Send time)，排队时间(Wait Quota time)，os 发送时间以及数据处理时间。

2.2.6 执行方式

2.2.6.1 Fast_query_shipping

对于查询语句 select c_customer_sk from customer group by c_customer_sk;

执行 explain 查看执行计划如下：

explain verbose select c_customer_sk from customer group by c_customer_sk;

结果如下：

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Data Node Scan  (cost=0.00..0.00 rows=0 width=0)
   Output: customer.c_customer_sk
   Node/s: All datanodes
   Remote query: SELECT c_customer_sk FROM public.customer GROUP BY c_customer_sk
(4 rows)
```

生成了 Data Node Scan 这样的计划，下面的 Remote query 代表的是要下发到 datanode 上执行的查询语句。生成这样的计划，表示的是当前查询可以在各个 datanode 上独立执行，coordinator 只用做最终的数据收集就可以了。

这样的计划实际上并未看到实际的执行计划，需要看的话，就需要关闭 fast_query_shipping 的功能了。

set enable_fast_query_shipping=off;
explain verbose select c_customer_sk from customer group by c_customer_sk;

结果如下：

```
 id |              operation               | E-rows | E-distinct | E-width | E-costs
----+--------------------------------------+--------+------------+---------+---------
  1 | ->  Streaming (type: GATHER)         |     30 |            |       4 | 15.53
  2 |    ->  HashAggregate                 |     30 |            |       4 | 14.28
  3 |       ->  Seq Scan on public.customer |     30 |            |       4 | 14.14
(3 rows)

 Targetlist Information (identified by plan id)
------------------------------------------------
   1 --Streaming (type: GATHER)
         Output: c_customer_sk
         Node/s: All datanodes
   2 --HashAggregate
         Output: c_customer_sk
         Group By Key: customer.c_customer_sk
   3 --Seq Scan on public.customer
         Output: c_customer_sk
         Distribute Key: c_customer_sk
(9 rows)
```

2.2.6.2 Remote query

通常对于不下推的查询，即只在 datanode 上做简单的数据查询，连接、聚集操作都在 coordinator 上执行。

例如：

```
set enable_stream_operator=off;
explain verbose select inv_warehouse_sk from customer, inventory where
inv_date_sk=c_current_cdemo_sk and c_current_addr_sk <inv_item_sk limit 10;
```

结果如下：

```
                                                
```

explain verbose select distinct on(c_customer_id) c_customer_id from customer;

结果如下：

```
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Unique  (cost=0.74..0.89 rows=30 width=20)
   Output: customer.c_customer_id
   ->  Sort  (cost=0.74..0.81 rows=30 width=20)
         Output: customer.c_customer_id
         Sort Key: customer.c_customer_id
         ->  Data Node Scan on customer "_REMOTE_TABLE_QUERY_"  (cost=0.00..0.00 rows=30 width=20)
               Output: customer.c_customer_id
               Node/s: All datanodes
               Remote query: SELECT c_customer_id FROM ONLY public.customer WHERE true
(9 rows)
```

2.2.6.3 Stream

最常用的计划就是可下推的执行计划，也就是分布式执行框架使用的 stream。前提肯定就是 enable_stream_operator=on 了。

还是拿 Query1 来举例。Query1（ select inv_warehouse_sk from customer, inventory where inv_date_sk=c_current_cdemo_sk and c_current_addr_sk <inv_item_sk limit 10; ）的执行计划为：

```
->  Limit
    ->  Streaming (type: GATHER)
        ->  Limit
            ->  Hash Join (5,8)
                ->  Streaming(type: BROADCAST)
                    ->  Partition Iterator
                        ->  Partitioned Seq Scan on public.inventory
                ->  Hash
                    ->  Seq Scan on public.customer
->  Limit
    ->  Streaming (type: GATHER)
        ->  Limit
            ->  Hash Join (5,7)
                ->  Streaming(type: REDISTRIBUTE)
                    ->  Seq Scan on customer
                ->  Hash
                    ->  Streaming(type: REDISTRIBUTE)
                        ->  Partition Iterator
                            ->  Partitioned Seq Scan on inventory
```

计划中出现了 Streaming (type: GATHER) 就可以说明当前计划是支持 stream 的，而且该节点就是个分界线，处于该节点之下的节点都是在 datanode 上执行的，该节点及其之上的节点都是在 coordinator 上执行的。

Streaming(type: REDISTRIBUTE) 也是 stream 节点，不过这个是在 datanode 上执行的。类似的还有 Streaming(type: BROADCAST)。

Datanode 上的 stream 节点，也是数据收集的节点，不过它的子节点的执行和它不是在同一个线程里的。在每一个 datanode 上，stream 节点所在的线程叫作主线程，stream 的子节点所在的线程叫子线程。

Stream 类型介绍：

- GATHER：聚合节点，在 coordinator 上执行，目的就是将所有 datanode 上的数据收集到 coordinator 上进行处理。
- BROADCAST：数据广播，即将子线程返回的数据发送给每一个 datanode 上，每个 datanode 上最终收到的都是全量的数据。
- REDSITRIBUTE：数据分布，即将子线程返回的数据根据分布列计算哈希，重新分布到不同的 datanode 上，所有 datanode 的数据总和才是全量的数据。

2.2.6.4 清理实验环境

使用\d 查看所有的表，注意\d 为 gsql 的元命令，只有在 gsql 才能使用。

```
\d
```

使用 drop 语句删除表。

```
drop table customer;
drop table inventory;
drop table web_page;
```

2.2.7 本节小结

本小节实验主要对 explain 命令进行了介绍，目的在于了解执行计划的显示格式设置、执行计划解析以及不同的执行方式效果。

3 数据库对象设计与管理

3.1 实验介绍

3.1.1 关于本实验

通过使用 GaussDB(DWS)默认用户库 postgres 创建和管理各种数据库对象，强化数据库各种对象的概念，以及如何创建与管理这些数据库对象。

3.1.2 实验目的

- 掌握常见的数据库对象的创建、删除与管理。

3.2 实验任务

3.2.1 连接数据库

在华为公有云购买 ECS 服务器，搭建 gsql 客户端。本地通过弹性 IP 连接 ECS 服务器，通过 gsql 客户端链接 GaussDB(DWS)数据库。159.138.228.159 为弹性 IP 网址，dbadmin 为管理员用户：

```
source gsql_env.sh
gsql -d postgres -h 159.138.228.159 -U dbadmin -p 8000 -r
```

3.2.2 数据库创建与管理

步骤 1　创建数据库 tpch

```
postgres=> create database tpch;
```

实验结果：CREATE DATABASE。

创建成功

步骤 2　切换到数据库 tpch

```
postgres=> \c tpch
```

根据提示输入密码 Password for user dbadmin:

结果为：SSL connection (cipher: DHE-RSA-AES256-GCM-SHA384, bits: 256)

You are now connected to database "tpch" as user "dbadmin".

切换成功

3.2.3 表的创建与管理

步骤 1 创建复制表

复制表的每个节点都有完整的表数据。

```
tpch=> CREATE TABLE REGION
tpch->  (
tpch(>          R_REGIONKEY    INT NOT NULL
tpch(>        , R_NAME           CHAR(25) NOT NULL
tpch(>        , R_COMMENT        VARCHAR(152)
tpch(>  )
tpch->  with (orientation = column , COMPRESSION = MIDDLE)
tpch->  distribute by replication
tpch->  ;
```

结果：CREATE TABLE

创建成功

步骤 2 创建行存 HASH 分区表

表 PART 是 HASH 分区表，表示表中的数据会按照 P_PARTKEY 来做 HASH 分布，按照 P_SIZE 分到不同 DN 节点。

```
tpch=> CREATE TABLE PART
tpch->  (
tpch(>          P_PARTKEY        BIGINT NOT NULL
tpch(>        , P_NAME           VARCHAR(55) NOT NULL
tpch(>        , P_MFGR           CHAR(25) NOT NULL
tpch(>        , P_BRAND          CHAR(10) NOT NULL
tpch(>        , P_TYPE           VARCHAR(25) NOT NULL
tpch(>        , P_SIZE           BIGINT NOT NULL
tpch(>        , P_CONTAINER      CHAR(10) NOT NULL
tpch(>        , P_RETAILPRICE DECIMAL(15,2) NOT NULL
tpch(>        , P_COMMENT        VARCHAR(23) NOT NULL
tpch(>  )
tpch->  distribute by hash(P_PARTKEY)
tpch->  PARTITION BY RANGE(P_SIZE)
tpch->  (
tpch(>          PARTITION P_SIZE_1 VALUES LESS THAN(11),
tpch(>          PARTITION P_SIZE_2 VALUES LESS THAN(21),
tpch(>          PARTITION P_SIZE_3 VALUES LESS THAN(31),
tpch(>          PARTITION P_SIZE_4 VALUES LESS THAN(41),
tpch(>          PARTITION P_SIZE_5 VALUES LESS THAN(51)
tpch(>  );
```

结果：CREATE TABLE

创建成功

步骤 3　创建 unlogged 表

UNLOGGED 表不会记录预写日志，可以提高数据导入性能，但是在异常情况下可能会发生数据丢失。

```
tpch=> CREATE UNLOGGED TABLE REGION_UNLOGGED
tpch-> (
tpch(>        R_REGIONKEY    INT NOT NULL
tpch(>     , R_NAME          CHAR(25) NOT NULL
tpch(>     , R_COMMENT       VARCHAR(152)
tpch(> )
tpch-> distribute by replication;
```

结果：CREATE TABLE

创建成功

步骤 4　创建临时表

临时表只存在于当前的会话，会话结束后，会自动删除。

```
tpch=> CREATE TEMP TABLE NATION_TEMP
tpch-> (
tpch(>        N_NATIONKEY    INT NOT NULL
tpch(>     , N_NAME          CHAR(25) NOT NULL
tpch(>     , N_REGIONKEY     INT NOT NULL
tpch(>     , N_COMMENT       VARCHAR(152)
tpch(> )
tpch-> distribute by replication
tpch-> ;
```

结果：CREATE TABLE

创建成功

步骤 5　CREATE TABLE LIKE 语法创建表

创建新表，新表自动继承 LIKE 子句后面的表的所有字段，数据类型和非空约束。

```
tpch=> CREATE TABLE part_like (LIke part);
```

结果：NOTICE:　The 'DISTRIBUTE BY' clause is not specified. Using 'p_partkey' as the distribution column by default.

HINT:　Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.

CREATE TABLE

创建成功

步骤 6　CREATE TABLE AS 语法创建表

创建新表，表的字段与 SELECT 后面的查询输出字段保持一致，同时把 SELECT 的结果写入新表中。

```
tpch=> CREATE TABLE part_as AS SELECT * FROM part;
```

结果：NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'p_partkey' as the distribution column by default.

HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.

WARNING: Statistics in some tables or columns(public.part.p_partkey) are not collected.

HINT: Do analyze for them in order to generate optimized plan.

INSERT 0 0

创建成功

步骤 7　修改表的属性-增加列

在 PART 表中新增一列 p_col1，数据类型是 BIGINT。

```
tpch=> ALTER TABLE part ADD COLUMN p_col1 bigint;
```

结果：ALTER TABLE

修改成功

步骤 8　增加列上的默认值

在 PART 表的 p_col1 增加 DEFAULT 设置，如果表中插入数据的时候没有指定这一列的数据，那么会自动补 1。

```
tpch=> ALTER TABLE part ALTER COLUMN p_col1 SET DEFAULT 1;
```

结果：ALTER TABLE

修改成功

步骤 9　删除列上的默认值

删除 PART 表的 p_col1 的默认值。

```
tpch=> ALTER TABLE part ALTER COLUMN p_col1 drop DEFAULT ;
```

结果：ALTER TABLE

删除成功

步骤 10　修改表的数据类型

修改 PART 表的 p_col1 的数据类型为 INT 型。

```
tpch=> ALTER TABLE part MODIFY p_col1 INT;
```

结果：ALTER TABLE

修改成功

步骤 11　修改表的列的名

把 PART 表的 p_col1 列重命名为 p_col。

```
tpch=> ALTER TABLE part RENAME p_col1 to p_col;
```

结果：ALTER TABLE

修改成功

步骤 12　删除列

删除 PART 表的 p_col 这一列。

```
tpch=> ALTER TABLE part DROP COLUMN p_col;
```

结果：ALTER TABLE

删除成功

步骤 13　删除表

删除 REGION 表

```
tpch=> DROP TABLE region;
```

结果：DROP TABLE

删除成功

3.2.4 索引的创建与管理

步骤 1　创建索引

在 PART 表的 p_partkey 列创建索引。PART 为分区表，要加 LOCAL 关键字。

```
tpch=> CREATE INDEX p_partkey_index on part(p_partkey) local;
```

结果：CREATE INDEX

创建成功

步骤 2　修改索引属性

修改索引的名称，然后将索引失效。

```
tpch=> ALTER INDEX p_partkey_index RENAME TO p_partkey_index_1;
```

结果：ALTER INDEX

```
tpch=> ALTER INDEX p_partkey_index_1 UNUSABLE;
```

结果：ALTER INDEX

修改成功

步骤 3　删除索引

```
tpch=> DROP INDEX p_partkey_index_1;
```

结果：DROP INDEX

删除成功

3.2.5 视图的创建与管理

步骤 1　创建视图

在 part 表上创建视图 part_v，要求只能查看 p_partkey 字段小于 30 的数据。

```
tpch=> CREATE OR REPLACE VIEW part_v AS SELECT * FROM part where p_partkey < 30;
```

结果：CREATE VIEW

创建成功

步骤 2　修改视图属性

将视图 part_v 重命名为 part_v1。

```
tpch=> ALTER VIEW part_v RENAME TO part_v1;
```

结果：ALTER VIEW

修改成功

步骤 3　删除视图

```
tpch=> DROP VIEW part_v1;
```

结果：DROP VIEW

删除成功

3.2.6 序列的创建与管理

步骤 1　创建序列

```
tpch=> CREATE SEQUENCE seq1;
```

结果：CREATE SEQUENCE

创建成功

步骤 2　引用序列

当向 part 表中插入数据的时候，如果 p_col 列没有指定数据，那么会将 SEQUENCE 生成的序列的值插入表中。

```
tpch=> ALTER TABLE part ADD COLUMN p_col BIGINT;
```

结果：ALTER TABLE

```
tpch=> ALTER TABLE part ALTER COLUMN p_col SET DEFAULT nextval('seq1');
```

结果：ALTER TABLE

引用成功

步骤 3　隐式创建带有 SEQUENCE 的表

隐式创建的效果与显式绑定 SEQUENCE 的效果一样。

```
tpch=> CREATE TABLE part_seq (p_seq serial, p_partkey bigint, p_name varchar(55));
```

结果：NOTICE:　CREATE TABLE will create implicit sequence "part_seq_p_seq_seq" for serial column "part_seq.p_seq"

NOTICE:　The 'DISTRIBUTE BY' clause is not specified. Using 'p_seq' as the distribution column by default.

HINT:　Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.

CREATE TABLE

```
tpch=> DROP SEQUENCE seq1 CASCADE;
```

NOTICE:　drop cascades to default for table part column p_col

DROP SEQUENCE

创建成功

步骤 4　删除 SEQUENCE

```
tpch=> DROP SEQUENCE seq1 CASCADE;
```

结果：NOTICE:　drop cascades to default for table part column p_col

DROP SEQUENCE

删除成功

3.2.7 SCHEMA 的创建与管理

步骤 1　创建 SCHEMA

```
tpch=> CREATE SCHEMA tpch;
```

结果：CREATE SCHEMA

创建成功

步骤 2　创建对象并使用

在 TPCH 这个 SCHEMA 下新建表 REGION，然后插入数据并查看。

```
tpch=> CREATE TABLE tpch.region
tpch-> (
tpch(>        R_REGIONKEY    INT NOT NULL
tpch(>     , R_NAME          CHAR(25) NOT NULL
tpch(>     , R_COMMENT       VARCHAR(152)
tpch(> )
tpch-> with (orientation = column , COMPRESSION = MIDDLE)
tpch-> distribute by replication
tpch-> ;
```

结果：CREATE TABLE

```
tpch=> INSERT INTO tpch.region VALUES(5, 'TEST MSG', 'this is a test record');
```

结果：INSERT 0 1

```
tpch=> SELECT * FROM tpch.region;
```

结果：

```
r_regionkey |          r_name          |          r_comment
-------------+--------------------------+----------------------
           5 | TEST MSG                 | this is a test record
(1 row)
```

步骤 3　设置 SCHEMA 默认路径

设置默认搜索路径为 TPCH 之后，可以不带模式名直接访问上述新建的表。

```
tpch=> SET SEARCH_PATH TO tpch;
```

结果：SET

```
tpch=> SELECT * FROM region;
```

结果：

```
r_regionkey |         r_name          |          r_comment
-------------+-------------------------+-----------------------
           5 | TEST MSG                | this is a test record
(1 row)
```

步骤 4　删除 SCHEMA

CASCADE 关键字表示级联删除 SCHEMA 下面所有的对象。如果不指定 CASCADE 关键字，当 SCHEMA 下有其他对象时，会删除 SCHEMA 失败。

```
tpch=> DROP SCHEMA tpch CASCADE;
```

结果：NOTICE:　drop cascades to table region

DROP SCHEMA

删除成功

3.3 清理运行环境

删除本章环境中所创建的数据库对象，将环境恢复到实验开始前的状态。因为本章所创建的对象都在数据库 tpch 内，因此只需要将数据库 tpch 删除即可。切换到默认用户数据库 postgres，然后删除数据库 tpch。

```
postgres=> drop database tpch;
```

结果：DROP DATABASE

删除成功

4 数据分区

4.1 实验介绍

4.1.1 关于本实验

分区表是把逻辑上的一张表根据某种方案分成若干张物理块进行存储。这张逻辑上的表称之为分区表，物理块称之为分区。分区表是一张逻辑表，不存储数据，数据实际是存储在分区上的。

范围分区表：将数据基于范围映射到每一个分区，这个范围是由创建分区表时指定的分区键决定的。这种分区方式是最为常用的，并且分区键经常采用日期，例如将销售数据按照月份进行分区。

本实验以华为公有云 GaussDB(DWS)服务为基础平台，同时在 ECS 服务器服务器上搭建 gsql 客户端，本地通过弹性公网 IP 连接 ECS 服务器，通过 gsql 客户端连接 GaussDB(DWS)数据库。

4.1.2 实验目的

- 强化分区表的概念
- 熟练掌握分区表的操作

4.1.3 注意事项

1、添加分区的表空间名不能是 PG_GLOBAL 且不能与该分区表已有分区的名字相同。

2、添加分区的分区键值要和分区表的分区键的类型一致，且要大于分区表中最后一个范围分区的上边界。

3、如果目标分区表中已有分区数达到了最大值（32767），则不能继续添加分区。

4、当分区表只有一个分区时，不能删除该分区。

5、选择分区使用 PARTITION FOR()，括号里指定值个数应该与定义分区时使用的列个数相同，并且一一对应。

6、Value 分区表不支持相应的 Alter Partition 操作。

7、在创建分区表时，START END 与 LESS THAN 语法不可混合使用。

8、对于从句是 VALUE LESS THAN 的语法格式，范围分区策略的分区键最多支持 4 列。

9、对于从句是 START END 的语法格式，范围分区策略的分区键仅支持 1 列。

10、每个分区都需要指定一个上边界，且分区上边界的类型应当和分区键的类型一致。

11、对于从句是 START END 的语法格式：

① 每个分区包含起始值，不包含终点值，即形如：[起始值，终点值)，起始值是 MINVALUE 时则不包含；

② 相邻的两个 partition_start_end_item，第一个的 END 值必须等于第二个的 START 值；

③ 每个 partition_start_end_item 中 START 值（如果有的话，下同）必须小于其 END 值；

12、分区列表是按照分区上边界升序排列的，值较小的分区位于值较大的分区之前。

4.2 实验任务

4.2.1 创建分区表

步骤 1　LESS THAN 语法

用 LESS THAN 语法创建分区表 customer_address，共分八个区。

首先创建数据库 tpch，在该数据库下完成章实验内容。

```
postgres=> create database tpch;
CREATE DATABASE
postgres=> \c tpch
Password for user dbadmin:
SSL connection (cipher: DHE-RSA-AES256-GCM-SHA384, bits: 256)
You are now connected to database "tpch" as user "dbadmin".
tpch=>
```

建库成功。同时已切换到 tpch 数据库。

创建分区表：

```
tpch=> CREATE TABLE customer_address
tpch-> (
tpch(> ca_address_sk integer NOT NULL ,
tpch(> ca_address_id character(16) NOT NULL ,
tpch(> ca_street_number character(10) ,
tpch(> ca_street_name character varying(60) ,
tpch(> ca_street_type character(15) ,
tpch(> ca_suite_number character(10) ,
tpch(> ca_city character varying(60) ,
tpch(> ca_county character varying(30) ,
tpch(> ca_state character(2) ,
tpch(> ca_zip character(10) ,
tpch(> ca_country character varying(20) ,
tpch(> ca_gmt_offset numeric(5,2) ,
tpch(> ca_location_type character(20)
tpch(> )
tpch-> DISTRIBUTE BY HASH (ca_address_sk)
```

```
tpch-> PARTITION BY RANGE (ca_address_sk)
tpch-> (
tpch(> PARTITION P1 VALUES LESS THAN(5000),
tpch(> PARTITION P2 VALUES LESS THAN(10000),
tpch(> PARTITION P3 VALUES LESS THAN(15000),
tpch(> PARTITION P4 VALUES LESS THAN(20000),
tpch(> PARTITION P5 VALUES LESS THAN(25000),
tpch(> PARTITION P6 VALUES LESS THAN(30000),
tpch(> PARTITION P7 VALUES LESS THAN(40000),
tpch(> PARTITION P8 VALUES LESS THAN(MAXVALUE)
tpch(> )
tpch-> ENABLE ROW MOVEMENT;
```

结果：CREATE TABLE

创建成功

步骤 2　START END 语法

START END 语法创建表分区表 customer_address2，共分 5 个分区。

```
tpch=> CREATE TABLE customer_address2
tpch-> (
tpch(> ca_address_sk integer NOT NULL ,
tpch(> ca_address_id character(16) NOT NULL ,
tpch(> ca_street_number character(10) ,
tpch(> ca_street_name character varying(60) ,
tpch(> ca_street_type character(15) ,
tpch(> ca_suite_number character(10) ,
tpch(> ca_city character varying(60) ,
tpch(> ca_county character varying(30) ,
tpch(> ca_state character(2) ,
tpch(> ca_zip character(10) ,
tpch(> ca_country character varying(20) ,
tpch(> ca_gmt_offset numeric(5,2) ,
tpch(> ca_location_type character(20)
tpch(> )
tpch-> DISTRIBUTE BY HASH (ca_address_sk)
tpch-> PARTITION BY RANGE (ca_address_sk)
tpch-> (
tpch(> PARTITION P1 START(1) END(1000) EVERY(500),
tpch(> PARTITION P2 END(2000),
tpch(> PARTITION P3 START(2000) END(2500) ,
tpch(> PARTITION P4 START(2500),
tpch(> PARTITION P5 START(3000) END(5000)
tpch(> )
tpch-> ENABLE ROW MOVEMENT;
```

结果：CREATE TABLE

创建成功

4.2.2 管理分区表

步骤 1 行迁移属性

修改分区表 customer_address 属性，不允许行迁移。

```
tpch=> ALTER TABLE customer_address DISABLE ROW MOVEMENT;
```

结果：ALTER TABLE

修改成功

步骤 2 删除分区

删除表 customer_address 的分区 p8

```
tpch=> ALTER TABLE customer_address DROP PARTITION P8;
```

结果：ALTER TABLE

删除成功

步骤 3 增加分区

增加分区 P8，范围为 40000<= P8<MAXVALUE

```
tpch=> ALTER TABLE customer_address ADD PARTITION P8 VALUES LESS THAN (MAXVALUE);
```

结果：ALTER TABLE

修改成功

步骤 4 重命名分区

重命名 P8 分区为 P_9

```
tpch=> ALTER TABLE customer_address RENAME PARTITION P8 TO P_9;
```

结果：ALTER TABLE

重命名 40000 所在的分区为 P8

```
tpch=> ALTER TABLE customer_address RENAME PARTITION FOR (40000) TO P8;
```

结果：ALTER TABLE

修改成功

步骤 5 查询分区

查询分区 P6

```
tpch=> SELECT * FROM customer_address PARTITION (P6);
```

结果：ca_address_sk | ca_address_id | ca_street_number | ca_street_name | ca_street_type | ca_suite_number | ca_city | ca_county | ca_state | ca_zip | ca_country | ca_gmt_offset | ca_location_type

```
---------------+---------------+-----------------+----------------+----------------+----------------+---------+-----------+----------+--------+------------+---------------+-----------------
```

(0 rows)

查询 35888 所在的分区

```
tpch=> SELECT * FROM customer_address PARTITION FOR (35888);
```

结果：ca_address_sk | ca_address_id | ca_street_number | ca_street_name | ca_street_type | ca_suite_number | ca_city | ca_county | ca_state | ca_zip | ca_country | ca_gmt_offset | ca_location_type

```
---------------+---------------+-----------------+----------------+----------------+-----------------+---------+-----------+----------+--------+------------+---------------+------------------
```

(0 rows)

查询成功

4.3 清理运行环境

删除本章环境中所创建的数据库对象，将环境恢复到实验开始前的状态。因为本章所创建的对象都在数据库 tpch 内，因此只需要将数据库 tpch 删除即可。切换到默认用户数据库 postgres，然后删除数据库 tpch。

```
postgres=> drop database tpch;
```

结果：DROP DATABASE

删除成功

5 事务管理

5.1 实验介绍

5.1.1 关于本实验

本实验主要分 4 方面的内容：锁的使用、事务一致性、多 session 测试读写冲突和子事务。

以华为公有云 GaussDB(DWS)服务为基础平台，同时在 ECS 服务器服务器上搭建 gsql 客户端，本地通过弹性公网 IP 连接 ECS 服务器，通过 gsql 客户端连接 GaussDB(DWS)数据库。在该基础上进行实验。

5.1.2 实验目的

- 强化对锁基本原理的理解，熟练掌握锁的使用。
- 通过实验直观感受事务的一致性，强化对事务的理解。
- 通过实验直观感受数据库并发的控制，强化对数据库中事务隔离的理解。
- 通过实验强化对子事务的理解。

5.2 实验任务

5.2.1 锁的应用

5.2.1.1 实验原理

数据库实现并发控制的基本方法是使用锁来控制临界区互斥访问。后台进程对磁盘文件进行访问操作是，首先要获取锁。如果成功获得目标锁，则进入临界区执行磁盘读写访问，访问完成后退出临界区并释放锁；否则，进程睡眠直到被别的后台进程唤醒后重试。

数据库事务管理过程中的锁一般称为 RegularLock 锁，RegularLock 由 LWLock 实现，其特点是：有等待队列，有死锁检测，能自动释放锁。

RegularLock 支持的锁模式有八种，按排他（Exclusive）级别从低到高分别是：

- 访问共享锁（AccessShareLock）
- 行共享锁（RowShareLock）
- 行排他锁（RowExclusiveLock）
- 共享更新排他锁（ShareUpdateExclusiveLock）

- 共享锁（ShareLock）
- 共享行排他锁（ShareRowExclusiveLock）
- 排他锁（ExclusiveLock）
- 访问排他锁（AccessRowExclusiveLock）

每种锁模式都有与之相冲突的锁模式，由锁冲突表定义相关信息。

本节实验将会对表格中 8 级锁 AccessExclusiveLock 与 1 级锁 AccessShareLock 的锁冲突场景进行测试，了解这些锁的使用对数据库的日常维护具有非常重要的帮助。

5.2.1.2 实验步骤

步骤 1　连接数据库

```
source gsql_env.sh
gsql -d postgres -h 159.138.228.159 -U dbadmin -p 8000 -r
```

步骤 2　创建数据库 tpch

创建数据库 tpch，并切换到新建数据库。

```
postgres=> create database tpch;
postgres=> \c tpch
```

步骤 3　创建表

创建表 test1，插入数据，并查看所插入数据。

```
tpch=> create table test1(id integer, name varchar(32));
tpch=> insert into test1 values(1, 'Gauss');
tpch=> insert into test1 values(2, 'Mppdb');
tpch=> select * from test1;
```

结果：

```
id | name
----+-------
  1 | Gauss
  2 | Mppdb
(2 rows)
```

插入成功

步骤 4　新打开窗口 2

重新打开一个窗口会话 2，切换到新建的库，并执行 begin 语句开启事务块内运行 SQL。

```
tpch=> begin;
```

结果：BEGIN

步骤 5　窗口 2 执行查询语句

在会话 2 的事务块内执行查询表 test1，此处 select 操作获取 AccessShareLock 锁。

```
tpch=> select * from test1 where id=1;
```

结果：

```
id | name
----+-------
  1 | Gauss
(1 row)
```

步骤 6　窗口 1 执行修改语句

窗口 1（最初开启的窗口）执行更改表结构的 SQL 操作。在表 test1 中增加一列，此处 ALTER 申请 AccessExclusiveLock。

```
tpch=> alter table test1 add column city char(1);
```

结果：一直处于等待语句执行中，无法查看结果

步骤 7　新打开窗口 3

开启会话 3 并切换到创建的数据库 tpch，查询表 test1 中各个会话的状态，首先根据表名 test1 在系统表 pg_class 中查询 test1 对应的 oid。

```
tpch=> select oid, relname from pg_class where relname='test1';
```

结果：

```
  oid   | relname
-------+---------
 16605 | test1
(1 row)
```

步骤 8　窗口 3 查询表线程及锁

窗口 3 中，通过获取到的表 test1 对应的 oid，在 pg_locks 中获取当前访问该表的线程以及加锁情况。

```
tpch=> select locktype, database, relation, pid, mode from pg_locks where relation='16605';
```

结果：

```
locktype | database | relation |        pid        |        mode
----------+----------+----------+-----------------+---------------------
 relation |    16604 |    16605 | 140257578080000 | AccessShareLock
 relation |    16604 |    16605 | 140257602926336 | AccessExclusiveLock
(2 rows)
```

步骤 9　窗口 3 查询执行中的 SQL 语句

窗口 3 中，通过获取到的 pid 线程号，在 pg_stat_activity 中找到对应线程正在执行的 query。

```
tpch=> select datname, pid, query from pg_stat_activity where pid in(140257578080000, 140257602926336);
```

结果：

```
datname |        pid        |                    query
---------+-----------------+-------------------------------------------
 tpch    | 140257578080000 | select * from test1 where id=1;
 tpch    | 140257602926336 | alter table test1 add column city char(1);
```

(2 rows)

从上述查询系统表结果可知，窗口 2（pid=140257578080000）获取的是 AccessShareLock，窗口 1（140257602926336）获取的是 AccessExclusiveLock。

步骤 10　窗口 2 提交事务

此时再回到会话 2 中，执行 commit 将该事务执行提交。

```
tpch=> commit;
```

结果：COMMIT

提交成功

步骤 11　查看窗口 1

窗口 2 中事务提交完成后，与此同时会发现窗口 1 中的 ALTER TABLE 命令执行成功。

结果：ALTER TABLE

同时查看 test1 表结构变化情况：

```
tpch=> \d test1
```

结果：

```
                Table "public.test1"
 Column |          Type          | Modifiers
--------+------------------------+-----------
 id     | integer                |
 name   | character varying(32) |
 city   | character(1)           |
```

修改成功。已添加字段

步骤 12　窗口 3 查看锁

此时，回到窗口 3 查看锁的情况：

```
tpch=> select locktype, database, relation, pid, mode from pg_locks where relation='16605';
```

结果：

```
locktype | database | relation | pid | mode
----------+----------+----------+-----+------
(0 rows)
```

锁已释放。

通过该实验测试 8 级锁 AccessExclusiveLock 与 1 级锁 AccessShareLock 的所冲突情况，了解到‘ALTER TABLE’SQL 会与‘SELECT’命令相冲突。表格中的其他锁冲突的情况都可以通过该方法进行测试。

5.2.2 事务一致性

5.2.2.1 实验原理

事务（TRANSACTION）是数据库管理系统操作的执行单位，数据库事务通常包含了一个序列对数据库的读/写操作。当事务提交时，数据库系统需要确保事务中的所有操作都成功完成且结果被永久保留，如果事务中的有的操作没有完成，则事务中的所有操作都需要回滚，回到事务执行前的状态。

事务执行具有以下四个标准属性：

- 原子性（Atomicity）：在事务中的所有操作，只可能出现都成功或者都失败，没有其他任何中间状态。
- 一致性（Consistency）：事务开始之前以及结束后，数据库的完整性没有被破坏。
- 隔离性（Isolation）：多个事务并发访问时，事务之间是隔离的，不同事务之间不应互相影响。
- 持久性（Durability）：事务完成后，该事务对数据库所做的更改将会持久保存在数据库中。

参考实际生活中的应用场景，使大家对事务一致性问题有更进一步地理解。某人要在商店中使用电子货币购买 100 元商品，在这个过程中至少包含两个操作：

（1）付款者账户减少 100 元。

（2）商店账户增加 100 元。

5.2.2.2 实验步骤

步骤 1　建表并插入数据

创建交易订单表，插入数据，初始化付款者账户余额 500 元，商店账户余额 1000 元。

```
tpch=> create table orderlist(name varchar(32), amount integer);
tpch=> insert into orderlist values('customer', 500);
tpch=> insert into orderlist values('shop', 1000);
tpch=> SELECT * FROM ORDERLIST;
```

结果：

```
    name     | amount
----------+--------
 customer |      500
 shop        |     1000
(2 rows)
```

创建并插入数据成功

步骤 2　开启事务

初始化账户操作完成后，执行 begin 开启事务，模拟用户支付 100 元，商店账户收入 100 元的交易。

```
tpch=> begin;
```

```
tpch=> update orderlist set amount=400 where name='customer';
tpch=> update orderlist set amount=1100 where name='shop';
tpch=> select * from orderlist;
```

结果：

```
    name     | amount
----------+--------
 customer |      400
 shop        |     1100
(2 rows)
```

步骤 3　提交事务

模拟完成交易后，用户和商店账户的余额已经更新完毕，如果交易没有产生问题，执行 commit 将该事务提交，表示完成该笔交易。如果交易存在问题，则跳过步骤 3，执行步骤 4 中的命令。

```
tpch=> commit;
tpch=> select * from orderlist;
```

结果：

```
    name     | amount
----------+--------
 customer |      400
 shop        |     1100
(2 rows)
```

交易完成，事务提交。

步骤 4　回滚事务

如果在交易过程中出现问题，则跳过步骤 3，执行 rollback 命令，回滚该事物，取消该笔交易。交易终止，用户和商店的账户余额恢复到交易前的值。

```
tpch=> rollback;
tpch=> select * from orderlist;
```

结果：

```
    name     | amount
----------+--------
 customer |      500
 shop        |     1000
(2 rows)
```

总结：实验结果表明，支持事务的数据库系统必须确保对两个账户余额的更新操作都能够完成，或者一起取消，否则会出现账户余额异常的状况，将会造成非常严重的影响。

5.2.3 多 session 测试读写冲突

5.2.3.1 实验原理

数据库中存在多个会话试图同时访问同一数据的情况，并发控制的目标就是保证所有 session 高效地访问，同时维护数据的完整性。数据库利用多版本并发控制（MVCC）来维护数据的一致性。每个事务看到的都是一段时间之前的快照。如果对每个 session 会话进行事务隔离，就可以避免一个事务看到因其他并发事务的更新而导致不一致的数据。

SQL 执行过程中必须考虑在并发情况下三种问题：

- 脏读（Dirty Read）:一个事务读取了另一个未提交的并行事务写的数据。
- 不可重复读（Non-repeatable Reads）：一个事务重新读取前面读取过的数据，发现该数据已经被另一个已提交的事务修改过。
- 幻读（Phantom Read）：一个事务重新执行一个查询，返回一套符合查询条件的数据，发现这些数据因为其他最近提交的事务而发生改变。

本实验将对多 session 并发时，不同隔离级别下出现的数据可见性问题进行实验操作，更好地理解事务并发情况下的数据库处理机制。如果选择了读未提交级别，实际上是读已提交；若选择可重复读的级别，实际上用的是可串行化数据库内部读未提交，可串行化实际使用可重复读的隔离级别。故仅进行读已提交和可重复读两种隔离级别的测试。

5.2.3.2 实验步骤

#读已提交级别-脏读：

步骤 1　创建测试表 test

创建测试表 test。

```
tpch=> create table test(id int primary key, age int);
```

步骤 2　测试脏读

分别开启两个窗口，分别为 session1 窗口，session2 窗口，由于数据库系统默认隔离级别为读已提交，因此不用重新设置数据库的隔离级别。

（1）　在 session1，session2 中分别查看当前隔离级别，均为 read committed 级别；

（2）　在 session1 中先对表 test 进行查询，发现结果为空；

（3）　查看完毕后在 session2 中开启事务并插入数据；

（4）　session2 插入完成后，在 session1 中查询 test 表，发现结果仍为空；

（5）　session1 中查询完成后，在 session2 中执行 rollback 回滚 session2 中的事务；

（6）　session2 中事务回滚完成后，在 session1 中查询表 test，结果为空。

```
Session1:                                   Session2:
tpch=> show transaction_isolation;          tpch=> show transaction_isolation;
transaction_isolation                               transaction_isolation
----------------------                      ----------------------
read committed                                      read committed
(1 row)                                             (1 row)
```

```
tpch=> begin;
BEGIN
tpch=> select * from test;
id | age
----+-----
(0 rows)
```

```
tpch=> begin;
BEGIN
tpch=> insert into test values(1,1);
INSERT 0 1
```

```
tpch=> select * from test;
id | age
----+-----
(0 rows)
```

```
tpch=> rollback;
ROLLBACK
```

```
tpch=> select * from test;
id | age
----+-----
(0 row)
```

由此可见，当事务运行在隔离级别为读已提交时，一个 select 查询只能看到查询开始之前提交的数据，而永远无法看到未提交的数据或者是在查询执行时其他并行的事务提交所作的改变。因此在该隔离级别下，脏读可以避免，不可以读取未提交事务的数据。

#读已提交级别-不可重复读：

步骤 3　创建测试表 test2

```
tpch=> create table test2(id int primary key, age int);
```

步骤 4　不可重复读测试

别开启两个窗口，分别为 session1 窗口，session2 窗口，由于数据库系统默认隔离级别为读已提交，因此不用重新设置数据库的隔离级别。

（1）　在 session1 中先对表 test2 进行查询，结果为空；

（2）　session1 中查看完毕后在 session2 中开启事务并插入数据；

（3）　session2 插入完成后，在 session1 中查询 test2 表，发现结果为空；

（4）　session1 中查询完成后，在 session2 中执行 commit 提交 session2 中的事务；

（5）　session2 中事务提交完成后，在 session1 中查询表 test2，可以看到 session2 中插入的数据。

Session1:

```
tpch=> begin;
BEGIN
tpch=> select * from test2;
id | age
----+-----
```

Session2:

```
(0 rows)
                                                    tpch=> begin;
                                                    BEGIN
                                                    tpch=> insert into test2 values(1,1);
                                                    INSERT 0 1
tpch=> select * from test2;
id | age
----+-----
(0 rows)
                                                    tpch=> commit;
                                                    COMMIT
tpch=> select * from test2;
id | age
----+-----
  1 |   1
 (1 row)
```

由此可见，当事务运行在在隔离级别为读已提交时，一个 select 操作只可以查看到另一个事务提交后的数据。

#可重复读级别-脏读、不可重复读：

步骤 5　创建测试表 test4

```
tpch=> create table test4(id int primary key, age int);
```

步骤 6　测试脏读

分别开启两个窗口，分别为 session1 窗口，session2 窗口，由于数据库系统默认隔离级别为读已提交，因此需要重新设置数据库的隔离级别。

（1）　在 session1 和 session2 中设置隔离级别为 repeatable read，并查看设置后的隔离级别；

（2）　在 session1 中开启事务，查询表 test4，结果为空；

（3）　在 session1 中查询完成后，在 session2 中开启事务，并插入数据；

（4）　在 session2 中插入数据成功后，在 session1 中查询表 test4，结果为空；

（5）　在 session2 中提交该事务；

（6）　session2 事务提交成功后，在 session1 中继续查询表 test4，结果仍为空。

```
Session1:                                          Session2:
tpch=> set                                         tpch=> set
tpch-# default_transaction_isolation=              tpch-# default_transaction_isolation=
tpch-# 'REPEATABLE READ';                          tpch-# 'REPEATABLE READ';
tpch=> show transaction_isolation;                 tpch=> show transaction_isolation;
transaction_isolation                                    transaction_isolation
-----------------------                            -----------------------
repeatable read                                          repeatable read
(1 row)                                                  (1 row)
tpch=> begin;
BEGIN
```

```
tpch=> select * from test4;
id | age
----+-----
(0 rows)
```

```
tpch=> begin;
BEGIN
tpch=> insert into test4 values(1,1);
INSERT 0 1
```

```
tpch=> select * from test4;
id | age
----+-----
(0 rows)
```

```
tpch=> commit;
COMMIT
```

```
tpch=> select * from test4;
id | age
----+-----
(0 row)
```

由此可见，当事务运行在可重复读隔离级别下，事务内的查询在该事务中可以始终保持读取数据结果的一致性，并不会因为其他事务的提交而受到影响。

#可重复读级别-幻读：

步骤 7　创建测试表 test5

```
tpch=> create table test5(id int primary key, age int);
```

步骤 8　测试幻读

分别开启两个窗口，分别为 session1 窗口，session2 窗口，由于数据库系统默认隔离级别为读已提交，因此需要重新设置数据库的隔离级别。

（1）　在 session1 和 session2 中设置隔离级别为 repeatable read，并查看设置后的隔离级别；

（2）　在 session1 中开启事务，查询表 test5，结果为空；

（3）　在 session1 中查询完成后，在 session2 中开启事务，并插入数据；

（4）　在 session2 中插入数据成功后，在 session1 中查询表 test5，结果为空；

（5）　在 session2 中提交该事务；

（6）　session2 事务提交成功后，在 session1 中继续查询表 test5，结果仍为空；

（7）　在 session1 中插入与 session2 中相同的数据，提示 ERROR。

```
Session1:
tpch=> set
tpch-# default_transaction_isolation=
tpch-# 'REPEATABLE READ';
tpch=> show transaction_isolation;
transaction_isolation
-----------------------
repeatable read
```

```
Session2:
tpch=> set
tpch-# default_transaction_isolation=
tpch-# 'REPEATABLE READ';
tpch=> show transaction_isolation;
transaction_isolation
-----------------------
repeatable read
```

```
(1 row)                                                    (1 row)
tpch=> begin;
BEGIN
tpch=> select * from test5;
id | age
----+-----
(0 rows)
                                                           tpch=> begin;
                                                           BEGIN
                                                           tpch=> insert into test5 values(0,1);
                                                           INSERT 0 1
tpch=> select * from test5;
id | age
----+-----
(0 rows)
                                                           tpch=> commit;
                                                           COMMIT
tpch=> select * from test5;
id | age
----+-----
(0 row)
tpch=> insert into test5 values(0,1);
ERROR:  datanode1: duplicate key value violates unique constraint "test_pkey"
DETAIL:  Key (id)=(0) already exists.
```

由此可见，当事务运行在隔离级别为可重复读时，避免了脏读、不可重复读。但是当一个事务重新读取前面读取过的数据，会发现该数据已被另一个提交的事务修改，该情况就称为幻读。

步骤 9 清理环境

删除数据库 tpch：

```
postgres=> drop database tpch;
```

结果：DROP DATABASE

删除成功

5.2.4 子事务

5.2.4.1 实验原理

事务中可以嵌套一个或者多个事务，外层的事务称为父事务，里层嵌套的事务称为子事务。子事务具有和父事务一样的特性，可以整体做提交或者回滚。如果回滚父事务，不管是否单独提交过子事务，也将回滚所有子事务。

5.2.4.2 实验步骤

#BEGIN…END 嵌套形式：

步骤 1 创建测试库 tpch

登入 postgres 数据库，创建数据库 tpch，然后切换到该库。

```
source gsql_env.sh
gsql -d postgres -h 159.138.228.159 -U dbadmin -p 8000 -r
create database tpch;
\c tpch
```

步骤 2　创建测试表 test

```
tpch=> CREATE TABLE TEST(ID INT, AGE INT);
```

步骤 3　子事务验证

开启事务语句。由于子事务中 UPDATE TEST SET ID=3, AGE=3 WHERE ID=1;存在语法错误，当执行到该 sql 时报错，该子事务报错会发生回滚，因此 SQL 会转而执行后面的 INSERT INTO TEST VALUES(4,4) 。

```
tpch=> BEGIN
tpch$>  INSERT INTO TEST VALUES(1,1);
tpch$>  BEGIN
tpch$>    INSERT INTO TEST VALUES(2,2);
tpch$>    UPDATE TEST SET ID=3, AGE=3 WHERE ID=1;
tpch$>  EXCEPTION
tpch$>    WHEN OTHERS THEN
tpch$>    RAISE INFO 'ERROR OCCURS';
tpch$>  END;
tpch$>  INSERT INTO TEST VALUES(4,4);
tpch$>  END;
tpch$>  /
```

结果：

```
NFO:  ERROR OCCURS
ANONYMOUS BLOCK EXECUTE
```

步骤 4　查询 test 表

```
tpch=> SELECT * FROM TEST;
```

结果：

```
id | age
----+-----
  4 |   4
  1 |   1
(2 rows)
```

结论：由此可见，子事务执行失败回滚并不会影响父事务的正常提交。

步骤 5　环境清理

删除 test 测试表

```
tpch=> drop table test;
```

#BEGIN…SAVEPOINT…COMMIT/ROLLBACK 形式：

步骤 6　创建测试表 test

```
tpch=> CREATE TABLE TEST(ID INT, AGE INT);
```

步骤 7　开启事务

开启事务块并插入数据。

```
tpch=> begin;
BEGIN
tpch=> INSERT INTO TEST VALUES(1,1);
INSERT 0 1
```

步骤 8　设置保存点 S1

```
tpch=> SAVEPOINT S1;
SAVEPOINT
tpch=> INSERT INTO TEST VALUES(2,2);
INSERT 0 1
tpch=> INSERT INTO TEST VALUES(3,3);
INSERT 0 1
```

步骤 9　回滚到保存点 S1

回滚到保存点 S1，发现保存点 S1 之后插入的数据没有，表示回滚至 S1 成功。

```
tpch=> ROLLBACK TO SAVEPOINT S1;
ROLLBACK
tpch=> SELECT * FROM TEST;
```

结果：

```
d | age
----+-----
 1 |   1
(1 row)
```

步骤 10　设置保存点 S2

```
tpch=> SAVEPOINT S2;
SAVEPOINT
tpch=> INSERT INTO TEST VALUES(4,4);
INSERT 0 1
```

步骤 11　释放保存点 S2

```
tpch=> RELEASE SAVEPOINT S2;
RELEASE
tpch=> SELECT * FROM TEST;
```

结果：

```
id | age
----+-----
 1 |   1
```

```
 4 |  4
(2 rows)
```

步骤 12 设置保存点 S3

```
tpch=> SAVEPOINT S3;
SAVEPOINT
tpch=> INSERT INTO TEST VALUES(5,5);
INSERT 0 1
```

步骤 13 设置保存点 S4

```
tpch=> SAVEPOINT S4;
SAVEPOINT
tpch=> INSERT INTO TEST VALUES(6,6,6);
ERROR:  INSERT has more expressions than target columns
LINE 1: INSERT INTO TEST VALUES(6,6,6);
```

步骤 14 回滚到保存点 S4

```
tpch=> ROLLBACK TO SAVEPOINT S4;
ROLLBACK
tpch=> INSERT INTO TEST VALUES(6,6);
INSERT 0 1
```

步骤 15 释放保存点 S3

```
tpch=> RELEASE SAVEPOINT S3;
```

步骤 16 提交事务并查询

```
tpch=> COMMIT;
COMMIT
tpch=> SELECT * FROM TEST;
```

结果：

```
id | age
----+-----
  1 |   1
  4 |   4
  5 |   5
  6 |   6
(4 rows)
```

结论：通过设置保存点 savepoint 子事务的方式，我们能够回滚到任意一个已经存在的 savepoint 点，子事务和父事务具有一样的特性，可以整体提交或者回滚。

5.3 清理运行环境

删除本章环境中所创建的数据库对象，将环境恢复到实验开始前的状态。因为本章所创建的对象都在数据库 tpch 内，因此只需要将数据库 tpch 删除即可。切换到默认用户数据库 postgres，然后删除数据库 tpch。

```
postgres=> drop database tpch;
```

结果：DROP DATABASE

删除成功

6 数据库安全

6.1 实验介绍

6.1.1 关于本实验

演示数据库中对于用户以及角色的相关操作；在 GaussDB(DWS)数据库集群上执行对象权限相关操作；演示端到端构建基于角色的权限管理，并在数据库模型中加以应用。

本章实验以华为公有云 GaussDB(DWS)服务为基础平台，同时在 ECS 服务器服务器上搭建 gsql 客户端，本地通过弹性公网 IP 连接 ECS 服务器，通过 gsql 客户端连接 GaussDB(DWS)数据库。

6.1.2 实验目的

- 强化数据库用户与角色的概念。
- 掌握 GaussDB(DWS)中的用户与角色相关操作。
- 强化数据库中的对象权限的认识，同时掌握如何操作这些权限。
- 掌握数据库系统中如何构建实际的 RBAC 模型。

6.2 实验任务

6.2.1 用户、角色与权限操作

步骤 1 连接数据库

通过管理员用户 dbadmin 登入数据库 postgres

```
source gsql_env.sh
gsql -d postgres -h 159.138.228.159 -U dbadmin -p 8000 –r
```

步骤 2 创建用户 user_1

```
postgres=> create user user_1 password 'user@123';
```

结果：CREATE ROLE

创建成功

步骤 3 查看用户

分别从 pg_roles 视图和 pg_authid 系统表中查看创建的用户。

```
postgres=> select * from pg_roles where rolname = 'user_1';
postgres=> select * from pg_authid where rolname = 'user_1';
```

步骤 4　修改用户属性

修改用户 user_1 的密码

```
postgres=> alter user user_1 IDENTIFIED BY 'Gauss@123' replace 'user@123';
```

步骤 5　删除用户

```
postgres=> drop user user_1;
```

结果：DROP ROLE

删除成功

步骤 6　创建角色

创建普通角色 role_1 和带有系统权限的角色 role_2

```
postgres=> CREATE ROLE role_1 IDENTIFIED BY "role@123";
postgres=> CREATE ROLE role_2 WITH SYSADMIN IDENTIFIED BY "role@123";
```

步骤 7　查看角色

查看角色 role_1 信息，可以发现 rolcanlogin 为 f（false），说明角色 role_1 是无法登录连接数据库的。

```
postgres=> select * from pg_roles where rolname = 'role_1';
```

结果：

```
rolname | rolsuper | rolinherit | rolcreaterole | rolcreatedb | rolcatupdate | rolcanlogin | rolreplication | rolauditadmin |
rolsystemadmin | rolconnlimit |
  rolpassword | rolvalidbegin | rolvaliduntil |    rolrespool    | rolparentid | roltabspace | rolconfig |    oid    | roluseft |
rolkind | nodegroup
---------+----------+------------+---------------+-------------+--------------+-------------+----------------+---------------+-----------
-----+--------------+
-------------+---------------+---------------+--------------+-------------+-------------+-----------+-------+----------+---------+------
-----
  role_1   | f           | t            | f                | f               | f               | f               | f
| f               | f                |              -1 |
  ********      |                   |                  | default_pool |               0 |                 |               |
24681 | f           | n          |
(1 row)
```

步骤 8　修改角色属性

修改角色 role_1 的属性，增加登录属性

```
postgres=> ALTER ROLE role_1 LOGIN;
```

步骤 9　查看修改后的角色

查看角色 role_1 修改后的属性，可以发现 rolcanlogin 由 f（false）变为 t（true）。此时重新使用 role_1 进行登录验证。

```
postgres=> select * from pg_roles where rolname = 'role_1';
```

步骤 10　删除角色

```
postgres=> DROP ROLE role_1;
postgres=> DROP ROLE role_2;
```

6.2.2 对象权限操作

步骤 1　创建用户 user_1

创建用户 user_1，密码为 user@123，并用 user_1 登录数据库。

```
postgres=> create user user_1 password 'user@123';
gsql -d postgres -h 159.138.228.159 -U user_1 -p 8000  -r
```

步骤 2　创建表 table_1

```
postgres=> create table public.table_1 (id int, name text);
```

步骤 3　插入数据

```
postgres=> insert into public.table_1 values(1, 'string1'),(2, 'string2');
```

步骤 4　查看表的权限

```
postgres=> select * from INFORMATION_SCHEMA.role_table_grants where table_name = 'table_1';
```

INFORMATION_SCHEMA.role_table_grants 视图说明

列column	含义
grantor	授权者，上述步骤中table_1是user_1创建的，所以grantor为user_1自己。
grantee	被授权者，user_1创建table_1后，自己默认的权限被授权者也是user_1自己。
table_catalog	table_1属于postgres数据库
table_schema	由于create table时，没有指定table_1的模式，所以默认属于public模式。
table_name	表名table_1
privilege_type	权限类型，表对象每种权限对应role_table_grants中的一条行记录。
is_grantable	权限是否可被授予
with_hierarchy	是否允许在表继承层级上的特定操作，GaussDB A被包括在SELECT中。

步骤 5　创建用户 user_2

创建用户 user_2，并登录数据库

```
postgres=> create user user_2 password 'user@123';
gsql -d postgres -h 159.138.228.159 -U user_1 -p 8000 –r
```

步骤 6 访问表 table_1

```
postgres=> select * from table_1;
```

结果：ERROR: permission denied for relation table_1

访问被拒绝

步骤 7 给用户 user_2 授权

user_1 登录数据库，将 table_1 表的读权限赋给 user_2 用户。

```
gsql -d postgres -h 159.138.228.159 -U user_1 -p 8000 –r
postgres=> grant select on table public.table_1 to user_2;
```

步骤 8 user_2 访问 table_1

授权后再使用 user_2 登录数据库，并访问表 table_1。

```
gsql -d postgres -h 159.138.228.159 -U user_2 -p 8000 –r
postgres=> select * from table_1;
```

结果：

```
id |   name
----+---------
  1 | string1
  2 | string2
(2 rows)
```

步骤 9 删除用户

```
postgres=> drop user user_1 cascade;
postgres=> drop user user_2;
```

6.2.3 RBAC 关系模型

说明：一张 notices 表，Author 角色具有插入权限，Reader 角色具有读取权限，Author 角色下的所有用户都具备插入权限，同理 Reader 角色下所有用户都具有读取权限。对用户的权限分配转换成为了对角色的权限分配。

步骤 1 创建角色

管理员用户 dbadmin 创建 Author 与 Reader 角色

```
postgres=> CREATE ROLE author WITH CREATEDB CREATEROLE LOGIN PASSWORD 'author@123';
postgres=> CREATE ROLE reader WITH CREATEROLE LOGIN PASSWORD 'reader@123';
```

步骤 2 创建用户

author 角色登录数据库，并创建用户 author_1 与 author_2。

```
gsql -d postgres -h 159.138.228.159 -U author -p 8000 –r
postgres=> CREATE USER author_1 PASSWORD 'author@123';
```

```
postgres=> CREATE USER author_2 PASSWORD 'author@123';
```

步骤 3　创建表

author 角色创建 notices 表

```
postgres=> CREATE TABLE notices (id int , message text);
```

步骤 4　查看表权限

此时 author 角色具有 notices 表所有权限。

```
postgres=> select * from INFORMATION_SCHEMA.role_table_grants where table_name = 'notices';
```

步骤 5　赋权

将 author 角色的权限赋给 author_1 与 author_2，author 角色将表 notices 的 select 权限赋给 reader 角色。

```
postgres=> GRANT author TO author_1;
GRANT ROLE
postgres=> GRANT author TO author_2;
GRANT ROLE
postgres=> GRANT SELECT ON notices TO reader;
GRANT
```

步骤 6　再次查看表的权限

再次查看 notices 表的权限，发现 reader 具有了 notices 表的 SELECT 权限。

```
postgres=> select * from INFORMATION_SCHEMA.role_table_grants where table_name = 'notices';
```

步骤 7　再次创建用户

使用 reader 角色创建用户 reader_1 与 reader_2，操作步骤同 author。

```
gsql -d postgres -h 159.138.228.159 -U reader -p 8000 –r
postgres=> CREATE USER reader_1 PASSWORD 'author@123';
CREATE ROLE
postgres=> CREATE USER reader_2 PASSWORD 'author@123';
CREATE ROLE
```

步骤 8　赋权

将 eader 角色的权限赋给 reader_1 与 reader_2，操作步骤同 author。

```
postgres=> GRANT reader TO reader_1;
GRANT ROLE
postgres=> GRANT reader TO reader_2;
GRANT ROLE
```

步骤 9　author_1 发布消息

```
\q
gsql -d postgres -h 159.138.228.159 -U author_1 -p 8000 –r
postgres=> insert into notices values(1, 'message1'),(2, 'message2');
INSERT 0 2
```

步骤 10 reader_1 读取消息

```
\q
gsql -d postgres -h 159.138.228.159 -U reader_1 -p 8000  -r
postgres=> select * from notices;
```

结果：

```
id | message
----+----------
  1 | message1
  2 | message2
(2 rows)
```

6.3 清理运行环境

删除所创建的角色，用户和表。

```
postgres=> drop table notices;
DROP TABLE
postgres=> drop role author;
DROP ROLE
postgres=> drop role reader;
DROP ROLE
postgres=> drop user author_1;drop user author_2;
DROP ROLE
DROP ROLE
postgres=> drop user reader_1;drop user reader_2;
DROP ROLE
DROP ROLE
```

7 集群管理

7.1 实验介绍

7.1.1 关于本实验

本实验通过华为云控制台界面，对 GaussDB(DWS)集群进行状态查询、集群重启、集群基本信息及参数修改、集群安全设置、集群容量设置、集群快照设置等操作。

7.1.2 实验目的

- 熟悉华为云控制台界面。
- 掌握 GaussDB(DWS)集群状态的种类。
- 掌握 GaussDB(DWS)集群重启
- 掌握 GaussDB(DWS)集群基本信息及参数修改
- 掌握 GaussDB(DWS)集群安全设置
- 掌握 GaussDB(DWS)集群容量设置
- 掌握 GaussDB(DWS)集群快照设置

7.2 实验任务

7.2.1 集群状态

步骤 1 登陆华为云，点击“控制台”，进入控制台界面。

步骤 2 点击左侧的服务列表，选择“EI 企业智能”，选择“数据仓库服务”，进入 GaussDB(DWS)服务。

步骤 3 选择需要查看的集群，在集群状态栏信息中，显示集群的状态。集群状态分为“可用”、“只读”、“低性能”、“重分布中”、“重分布失败”、“节点故障”、“不可用”、“创建中”、“创建失败”、“创建中，恢复中”和“已冻结”。当集群状态为“可用”时，表明集群是正常的状态。

7.2.2 集群重启

步骤 1 登陆华为云，点击“控制台”，进入控制台界面。

步骤 2 点击左侧的服务列表，选择“EI 企业智能”，选择“数据仓库服务”，进入 GaussDB(DWS)服务。

步骤 3 选择集群列表中，需要重启的集群，点击右侧的“重启”，对集群进行重启。

步骤 4 点击重启后，弹出确认窗口，再次确认集群名称是否是需要重启的集群后，点击“确定”，集群进行重启。

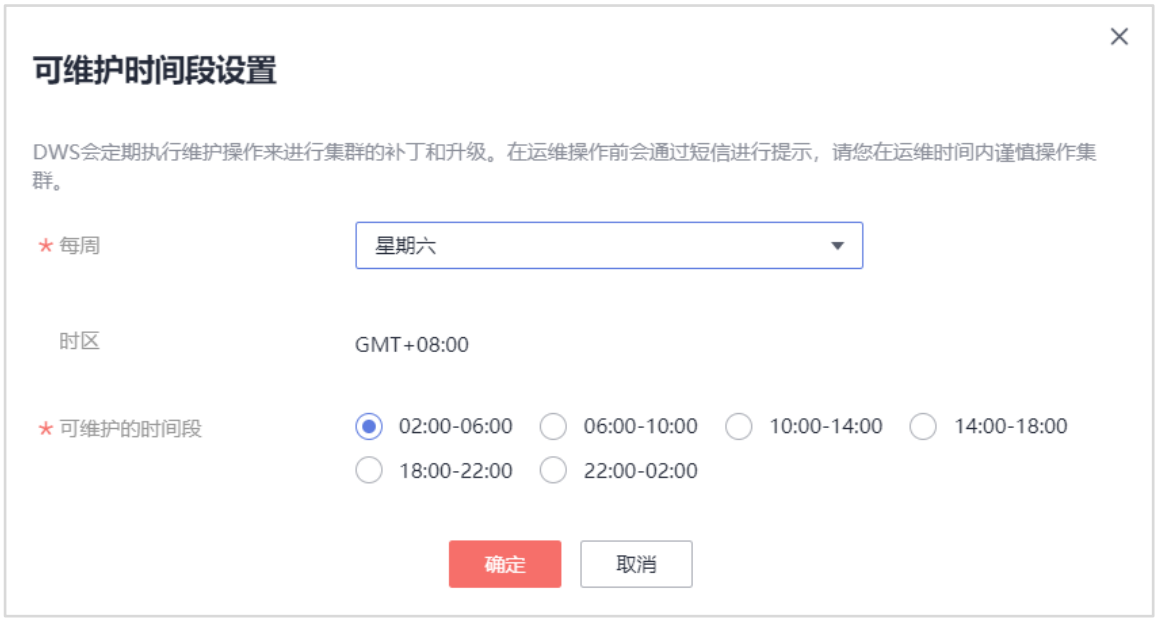
步骤 5 确定集群重启后，集群列表中，集群状态信息变为“不可用”，集群任务信息显示为“重启中”。

7.2.3 集群基本信息及参数修改

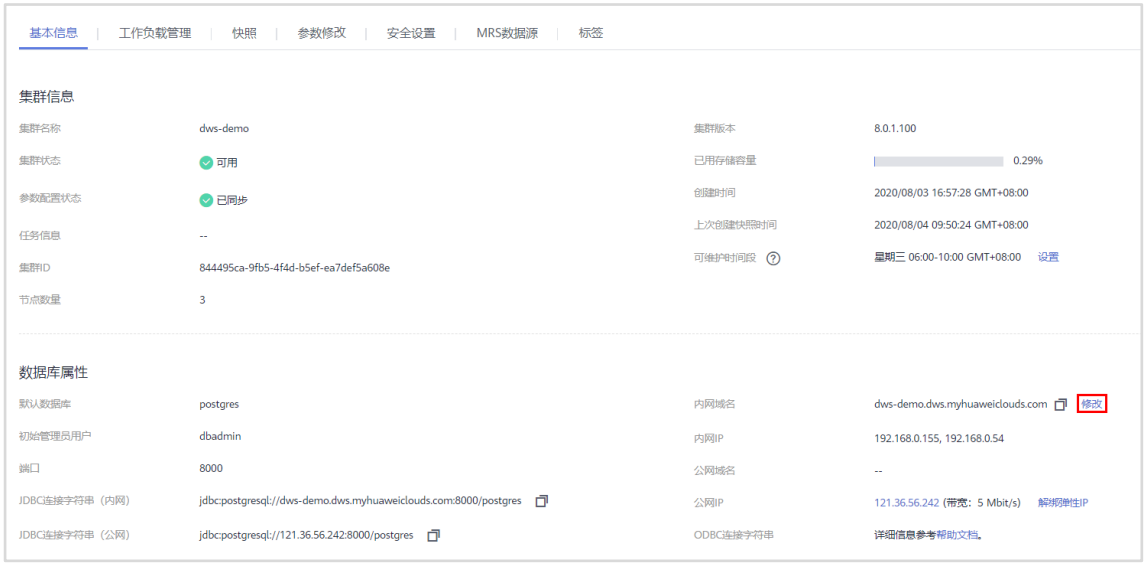
步骤 1 在集群列表，选择需要进行修改的集群，点击“集群名称”下的集群名。

步骤 2 在集群的“基本信息”的“集群信息”中，找到“可维护时间段”，点击右侧“设置”，对集群的维护时间进行修改。

步骤 3 将维护时间段修改成“星期六”的“02:00-06:00”区间后，点击“确定”，完成修改。



步骤 4 在“基本信息”的“数据库属性”中，找到“内网域名”，点击右侧“修改”，对集群的内网域名进行修改。

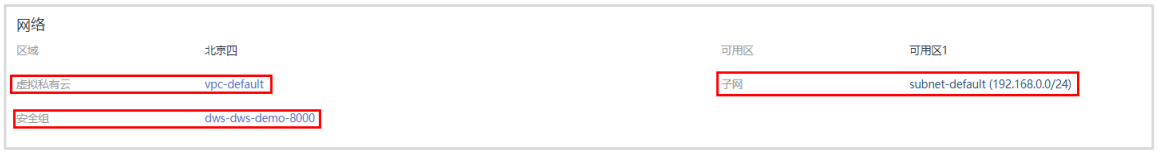


步骤 5 在弹出的修改框中，对内网域名进行修改，内网域名需以大小写字母开头，由字母，数字和中划线组成，长度为 4~63 个字符。修改完成后，点击“确定”，自动保存。（公网域名修改同内网域名）

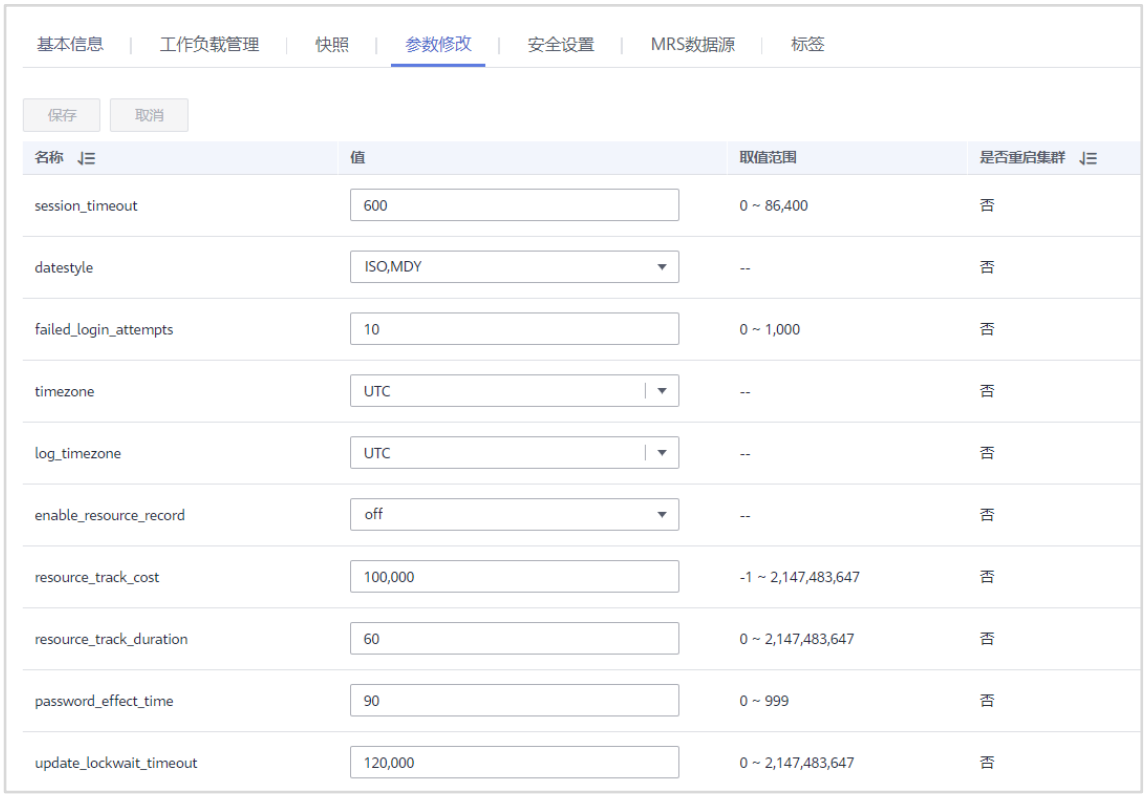
步骤 6 找到“公网 IP”，点击右侧“解绑弹性 IP”，在弹出确认框后，选择“是”，则将当前公网 IP 与集群进行解绑。

步骤 7 解绑后，点击“绑定弹性 IP”，可以选择其他弹性 IP 与 GaussDB(DWS)进行绑定，选定后点击“确认”予以绑定。

步骤 8 在“网络”中，可以对集群相关的虚拟私有云(VPC)、子网和安全组策略进行修改和调整。（此处不展开 VPC 和安全组策略的内容）



步骤 9 点击“参数修改”，进入参数修改页面，可以对集群参数进行修改，其中修改的值必须在“取值范围”内，修改完成后，点击保存，则对应参数值被修改并且生效（在此界面的参数修改保存后即生效，不需要重启集群。）。

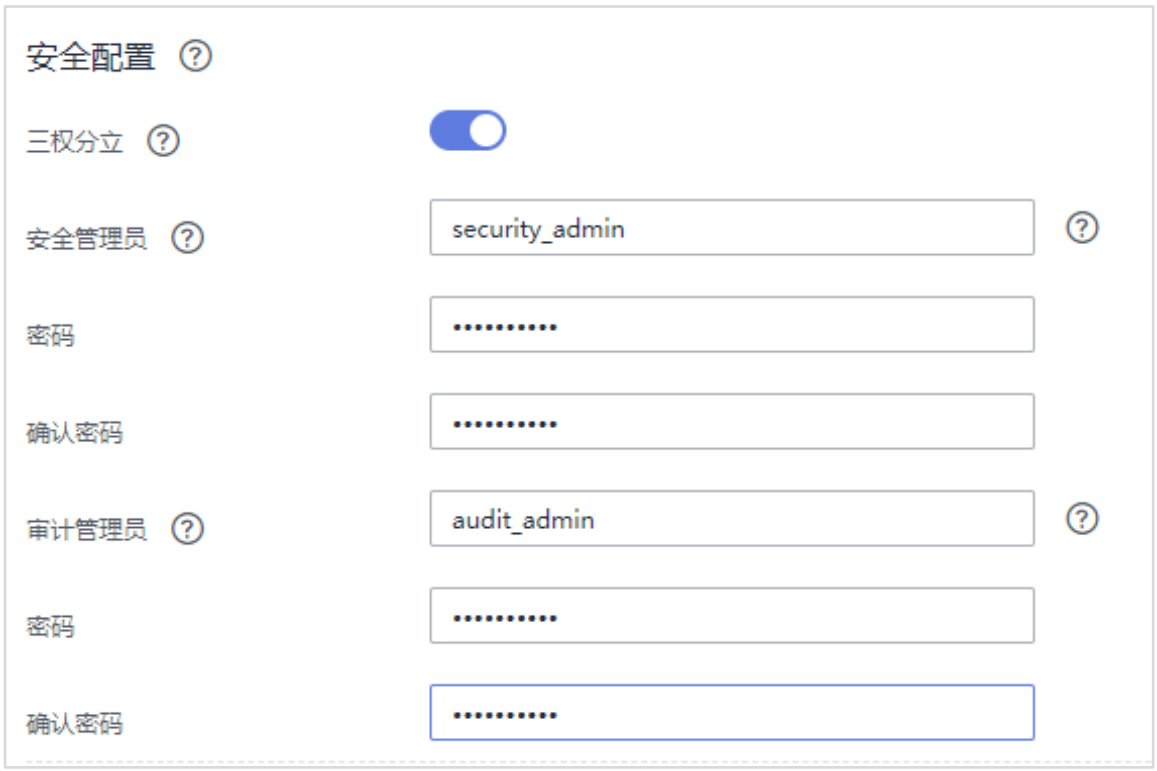


步骤 10 此处将 session_timeout 的值从默认的 600 秒改成 1800 秒，修改完点击“保存”后，弹出“修改预览”，预览完成后，点击确认“保存”。

7.2.4 集群安全

步骤 1 点击“安全设置”，进入安全设置界面。

步骤 2 打开“三权分立”的开关，开启三权分立，分别输入“安全管理员”和“审计管理员”的用户名和密码，点击页面下方的“应用”。

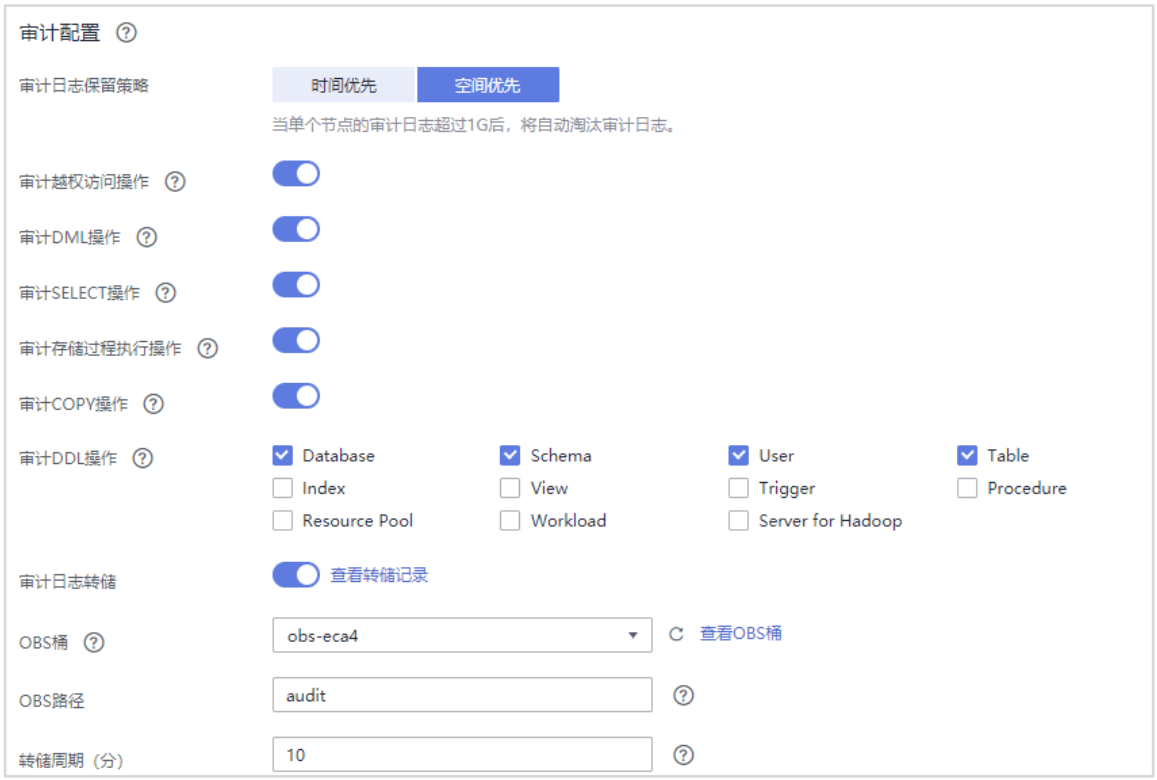


步骤 3 弹出确认窗口，根据集群使用情况进行确认。如果不选择立刻重启集群，需要在系统闲暇时，手动进行集群的重启；如果勾选了立刻重启集群，在确认“是”之后，则会重启集群，应用配置。

步骤 4 在集群重启过程中，集群状态信息会变为“不可用”，集群的任务信息会变为“配置中”。

步骤 5 等重启完成后，三权分立生效。

步骤 6 开启审计日志，选用默认的空间优先策略，将需要审计的内容进行打开和勾选。这里将所有操作打开，并且勾选 DDL 操作中的表。勾选完后，需要开启“审计日志转储”开关，选择云服务中的 OBS 桶，如果没有需要创建。填写对应存放 OBS 中的路径，以及转储周期使用默认的 10 分钟。完成之后，选择页面下方的“应用”，下发任务配置。



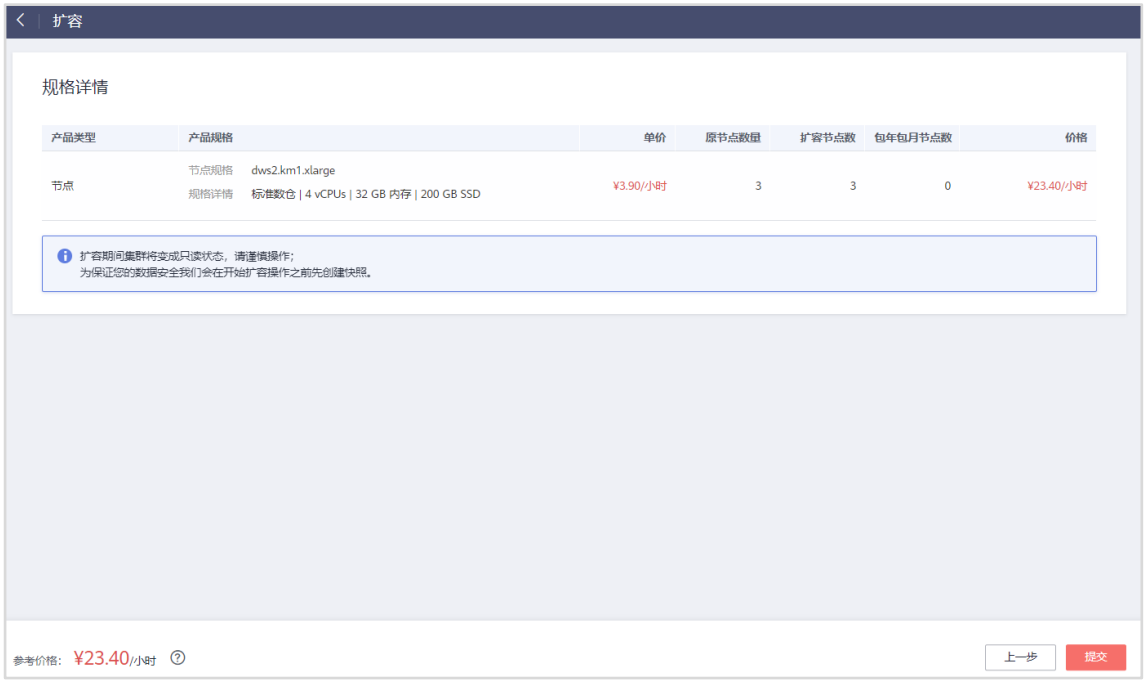
步骤 7 在集群配置过程中，集群状态仍为“可用”，集群的任务信息会变为“配置中”。

7.2.5 集群容量

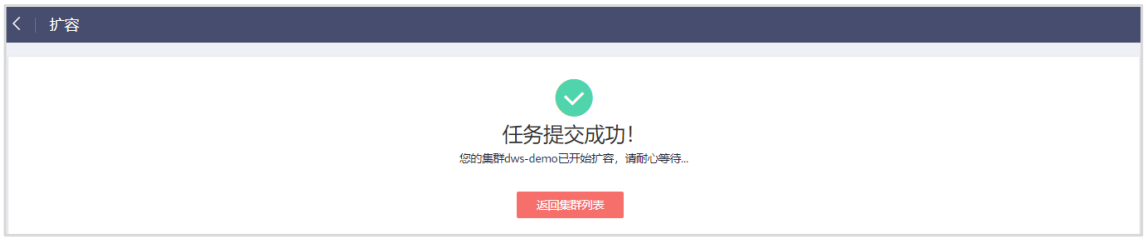
步骤 1 在集群列表中，在待扩容集群右侧选择“更多”。选中“扩容”，进入扩容界面。

步骤 2 依照扩容后的总容量，选择将集群扩容到多少个节点（目前默认最大扩容到 32 个节点，如果需要更多节点，需要申请更多配额）。本次实验，将集群扩容到 6 个节点。完成后，点击“下一步：规格确认”

步骤 3 在“规格确认”界面，再次确认扩容的节点数和单价。在界面上有提示： 扩容期间集群将变成只读状态和扩容操作之前先创建快照。确认完成后，点击“提交”，将扩容任务提交。



步骤 4 任务提交成功后，系统会进行集群扩容工作。



步骤 5 返回集群列表，可以看到任务信息变为“创建快照中”。

步骤 6 快照创建完成后，对节点开始进行扩容，任务信息变为“节点扩容”。

步骤 7 节点扩容完成后，集群的任务信息处变为空，并且集群的状态信息为“可用”。

7.2.6 集群快照

步骤 1 在集群列表中，选中需要创建快照的集群，点击“更多”，选择创建快照。

步骤 2 填写快照名称后，点击“确认”。注意：快照名称在 4 位到 64 位之间，必须以字母开头，不区分大小写，可以包含字母、数字、中划线或者下划线，不能包含其他的特殊字符。

步骤 3 任务下发后，在集群列表中，可以在创建快照的集群的任务信息处看到“创建快照中”。

步骤 4 点击左侧的“快照管理”，可以看到创建完成的快照。

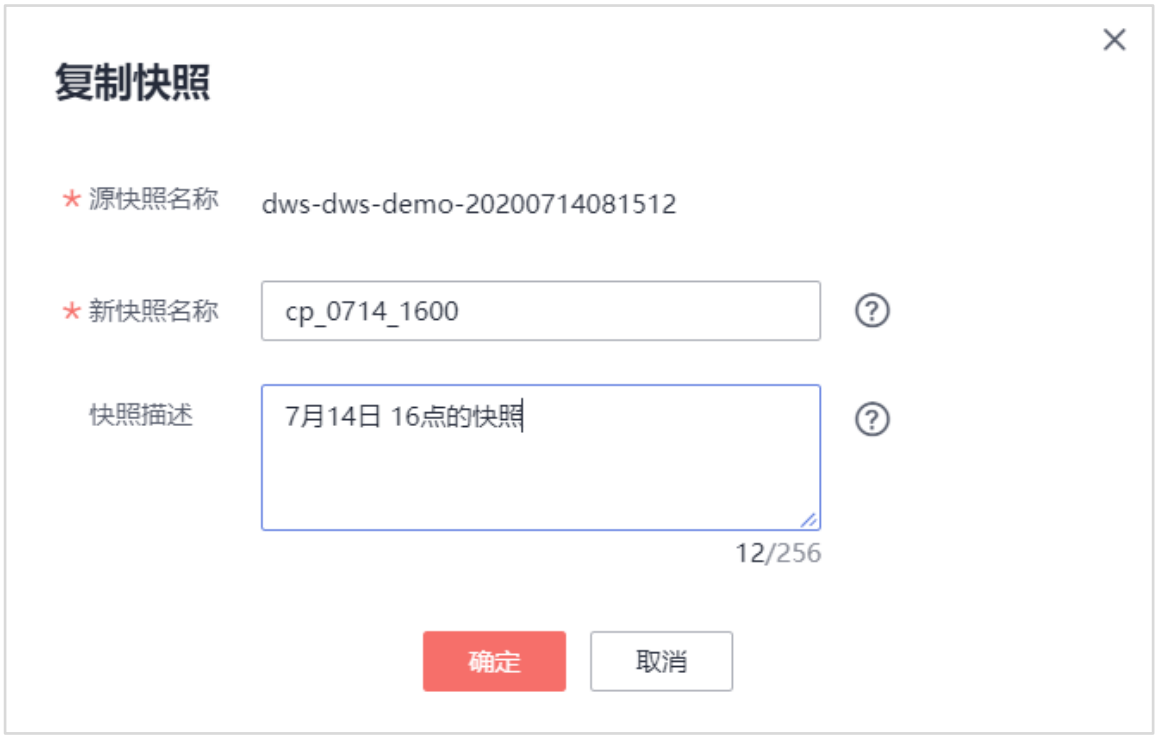
步骤 5 在“集群管理”中，点击需要进行管理的集群名称。

步骤 6 点击“快照”功能，在快照管理界面，可以对自动快照的策略进行修改。点击“修改快照策略”。

步骤 7 在快照策略中，可以对自动快照保留的时间、自动快照执行的周期进行修改。本实验，我们将自动快照保留的天数改为 7 天，将自动快照的时间改为每 4 小时执行一次。完成后，点击“确定”。

步骤 8 由于自动快照不能长期有效保存，可以将自动快照复制成为手动快照，用于长期保存。点击自动快照后的“复制”。

步骤 9 弹出需要复制快照界面，在界面中输入“新快照名称”（快照名称在 4 位到 64 位之间，必须以字母开头，不区分大小写，可以包含字母、数字、中划线或者下划线，不能包含其他的特殊字符），“快照描述”是用来方便进行管理，不强制要求填写。填写完成后，点击“确认”，完成快照复制。



步骤 10 自动快照系统会自动根据保留策略进行删除，手动快照需要手动进行删除，点击选中的手动快照后的“删除”，弹出确认界面。

步骤 11 确认需要进行手动删除的快照，确认无误后点击“是”，将手动删除快照。

步骤 12 通过自动快照或手动快照，对集群进行恢复。点击需要恢复的快照右侧的“恢复”，进入恢复界面。

步骤 13 选择跟原集群相同的可用区，在集群规格上，也默认选择跟原来集群相同的配置。

步骤 14 填写新的集群名称，设置新集群的数据库端口号。

步骤 15 根据业务要求，配置新集群的虚拟私有云、子网和安全组，如无改变，可以使用与原集群相同的配置。配置完成后，点击“立即恢复”。

步骤 16 在“规格确认”界面，对新集群的规格信息（如可用区、节点规格、集群名称等信息）进行确认，确认无误后，点击“提交”，进行集群的恢复。

步骤 17 返回集群列表界面，可以看到在新集群恢复时，集群的状态是“创建中，恢复中”。等待集群状态成为“可用”，则新的集群恢复完成。

7.3 本节小结

本节主要练习了 GaussDB(DWS)集群通过控制台界面的管理和运维，包括查看集群状态，重启集群，查看集群基本信息及参数配置，设置集群安全，设置集群容量和管理集群快照等。

8 综合实验

8.1 数据库对象设计

8.1.1 实验介绍

8.1.1.1 关于本实验

本实验通过操作创建和管理数据库、模式、表、索引和视图，逐步理解和掌握 GaussDB(DWS) SQL 语法。

8.1.1.2 实验目的

掌握基本对象的 SQL 语法。

8.1.2 实验任务

步骤 1　数据库相关操作

创建一个数据库，编码格式为 UTF-8。

```
postgres=> CREATE DATABASE tpcds100x template template0 encoding 'UTF8' lc_ctype 'en_US.UTF-8' lc_collate 'en_US.UTF-8';
CREATE DATABASE
```

创建兼容 Teradata 格式的数据库。

```
postgres=> CREATE DATABASE td_compatible_db DBCOMPATIBILITY 'TD';
CREATE DATABASE
```

创建兼容 ORACLE 格式的数据库。

```
postgres=> CREATE DATABASE ora_compatible_db DBCOMPATIBILITY 'ORA';
CREATE DATABASE
```

创建一个数据库，指定所属者。

```
postgres=> CREATE USER tom PASSWORD 'Bigdata123@';
CREATE ROLE
postgres=> CREATE DATABASE music2 OWNER Tom;
CREATE DATABASE
```

用模板 template0 创建数据库 edward，并指定所有者为 tom。

```
postgres=> CREATE DATABASE edward OWNER Tom TEMPLATE template0;
CREATE DATABASE
```

设置数据库的连接数为 10。

```
postgres=> ALTER DATABASE edward CONNECTION LIMIT= 10;
ALTER DATABASE
# 修改数据库的名称。
postgres=> ALTER DATABASE edward RENAME TO edward2;
ALTER DATABASE
# 修改数据的所属者。
postgres=> CREATE USER jim PASSWORD 'Bigdata123@';
CREATE ROLE
postgres=> ALTER DATABASE edward2 OWNER TO jim;
ALTER DATABASE
# 关闭在数据库 edward2 上缺省的索引扫描。
postgres=> ALTER DATABASE edward2 SET enable_indexscan TO off;
ALTER DATABASE
# 重置数据库 edward2 的 enable_indexscan 参数。
postgres=> ALTER DATABASE edward2 RESET enable_indexscan;
ALTER DATABASE
# 清理环境。
postgres=> DROP DATABASE tpcds100x;
DROP DATABASE
postgres=> DROP DATABASE td_compatible_db;
DROP DATABASE
postgres=> DROP DATABASE ora_compatible_db;
DROP DATABASE
postgres=> DROP DATABASE music2;
DROP DATABASE
postgres=> DROP DATABASE edward2;
DROP DATABASE
postgres=> DROP USER tom;
DROP ROLE
postgres=> DROP USER jim;
DROP ROLE
```

步骤 2　模式相关操作

```
# 创建模式 schema_tpcds。
postgres=> DROP SCHEMA IF EXISTS schema_tpcds;
NOTICE:  schema "schema_tpcds" does not exist, skipping
DROP SCHEMA
postgres=> CREATE SCHEMA schema_tpcds;
CREATE SCHEME
# 切换到模式 schema_tpcds 下，创建一张表。
```

```
postgres=> SET SEARCH_PATH TO schema_tpcds;
SET
postgres=> CREATE TABLE region (id int,name varchar);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'id' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
# 修改模式名字。
postgres=> ALTER SCHEMA schema_tpcds RENAME TO s_tpcds;
ALTER SCHEMA
# 修改模式属主。
postgres=> CREATE USER tom PASSWORD 'Bigdata123@';
CREATE ROLE
postgres=> ALTER SCHEMA s_tpcds OWNER TO tom;
ALTER SCHEMA
# 删除模式，带 RESTRICT 参数，未删除模式 s_tpcds 里面的对象时，删除失败。
postgres=> DROP SCHEMA s_tpcds RESTRICT;
ERROR:  cannot drop schema s_tpcds because other objects depend on it
DETAIL:  table s_tpcds.region depends on schema s_tpcds
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
# 删除模式，带 CASCADE 参数，自动删除包含在模式中的对象。
postgres=> DROP SCHEMA s_tpcds CASCADE;
NOTICE:  drop cascades to table s_tpcds.region
DROP SCHEMA
# 清理环境。
postgres=> DROP USER tom;
DROP ROLE
postgres=> SET SEARCH_PATH TO public;
SET
```

步骤 3　表相关操作

```
# 创建行存普通表，分布方式 replication。
postgres=> CREATE TABLE warehouse_row_1
postgres-# (
postgres(#     w_warehouse_sk          integer                 not null,
postgres(#     w_warehouse_id          char(16)                not null,
postgres(#     w_warehouse_name         varchar(20)                    ,
postgres(#     w_warehouse_sq_ft       integer                        ,
postgres(#     w_street_number         char(10)                       ,
postgres(#     w_street_name           varchar(60)                    ,
postgres(#     w_street_type           char(15)                       ,
postgres(#     w_suite_number          char(10)                       ,
```

```
postgres(#      w_city                  varchar(60)                 ,
postgres(#      w_county                varchar(30)                 ,
postgres(#      w_state                 char(2)                     ,
postgres(#      w_zip                   char(10)                    ,
postgres(#      w_country               varchar(20)                 ,
postgres(#      w_gmt_offset            decimal(5,2)
postgres(# )
postgres-# DISTRIBUTE BY REPLICATION;
CREATE TABLE
#  创建行存分区表，分布方式 hash。
postgres=> CREATE TABLE warehouse_row_2
postgres-# (
postgres(#      w_warehouse_sk          integer                 not null,
postgres(#      w_warehouse_id          char(16)                not null,
postgres(#      w_warehouse_name        varchar(20)                 ,
postgres(#      w_warehouse_sq_ft       integer                     ,
postgres(#      w_street_number         char(10)                    ,
postgres(#      w_street_name           varchar(60)                 ,
postgres(#      w_street_type           char(15)                    ,
postgres(#      w_suite_number          char(10)                    ,
postgres(#      w_city                  varchar(60)                 ,
postgres(#      w_county                varchar(30)                 ,
postgres(#      w_state                 char(2)                     ,
postgres(#      w_zip                   char(10)                    ,
postgres(#      w_country               varchar(20)                 ,
postgres(#      w_gmt_offset            decimal(5,2)
postgres(# )
postgres-# DISTRIBUTE BY HASH(w_warehouse_sk)
postgres-# PARTITION BY RANGE(w_warehouse_sq_ft)
postgres-# (
postgres(#      PARTITION p1 VALUES LESS THAN(1000),
postgres(#      PARTITION p2 VALUES LESS THAN(2000),
postgres(#      PARTITION p3 VALUES LESS THAN(3000),
postgres(#      PARTITION p4 VALUES LESS THAN(4000),
postgres(#      PARTITION p5 VALUES LESS THAN(5000),
postgres(#      PARTITION p6 VALUES LESS THAN(maxvalue)
postgres(# );
CREATE TABLE
#  创建列存表。
postgres=> CREATE TABLE warehouse_col
postgres-# (
postgres(#      w_warehouse_sk          integer                 not null,
postgres(#      w_warehouse_id          char(16)                not null,
postgres(#      w_warehouse_name        varchar(20)                 ,
postgres(#      w_warehouse_sq_ft       integer                     ,
postgres(#      w_street_number         char(10)                    ,
```

```
postgres(#      w_street_name              varchar(60)                ,
postgres(#      w_street_type             char(15)                   ,
postgres(#      w_suite_number            char(10)                   ,
postgres(#      w_city                    varchar(60)                ,
postgres(#      w_county                  varchar(30)                ,
postgres(#      w_state                   char(2)                    ,
postgres(#      w_zip                     char(10)                   ,
postgres(#      w_country                 varchar(20)                ,
postgres(#      w_gmt_offset              decimal(5,2)
postgres(# )
postgres-# WITH (ORIENTATION = COLUMN)
postgres-# DISTRIBUTE BY HASH(w_warehouse_sk);
CREATE TABLE
```

创建表，指定字段的缺省值和 NOT NULL 约束。

```
postgres=> CREATE TABLE warehouse
postgres-# (
postgres(#      W_WAREHOUSE_SK            INTEGER                NOT NULL,
postgres(#      W_WAREHOUSE_ID            CHAR(16)               NOT NULL,
postgres(#      W_WAREHOUSE_NAME          VARCHAR(20)                    ,
postgres(#      W_WAREHOUSE_SQ_FT         INTEGER                        ,
postgres(#      W_STREET_NUMBER           CHAR(10)                       ,
postgres(#      W_STREET_NAME             VARCHAR(60)                    ,
postgres(#      W_STREET_TYPE             CHAR(15)                       ,
postgres(#      W_SUITE_NUMBER            CHAR(10)                       ,
postgres(#      W_CITY                    VARCHAR(60)                    ,
postgres(#      W_COUNTY                  VARCHAR(30)                    ,
postgres(#      W_STATE                   CHAR(2)            DEFAULT 'GA',
postgres(#      W_ZIP                     CHAR(10)                       ,
postgres(#      W_COUNTRY                 VARCHAR(20)                    ,
postgres(#      W_GMT_OFFSET              DECIMAL(5,2)
postgres(# );
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'w_warehouse_sk' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
```

创建表，指定字段满足唯一约束。

```
postgres=> CREATE TABLE warehouse2
postgres-# (
postgres(#      W_WAREHOUSE_SK            INTEGER                NOT NULL,
postgres(#      W_WAREHOUSE_ID            CHAR(16)               NOT NULL,
postgres(#      W_WAREHOUSE_NAME          VARCHAR(20)              UNIQUE,
postgres(#      W_WAREHOUSE_SQ_FT         INTEGER                        ,
postgres(#      W_STREET_NUMBER           CHAR(10)                       ,
postgres(#      W_STREET_NAME             VARCHAR(60)                    ,
postgres(#      W_STREET_TYPE             CHAR(15)                       ,
postgres(#      W_SUITE_NUMBER            CHAR(10)                       ,
```

```
postgres(#      W_CITY                    VARCHAR(60)                ,
postgres(#      W_COUNTY                  VARCHAR(30)                ,
postgres(#      W_STATE                   CHAR(2)          DEFAULT 'GA',
postgres(#      W_ZIP                     CHAR(10)                   ,
postgres(#      W_COUNTRY                 VARCHAR(20)                ,
postgres(#      W_GMT_OFFSET              DECIMAL(5,2)
postgres(# );
NOTICE:  CREATE TABLE / UNIQUE will create implicit index
"warehouse2_w_warehouse_name_key" for table "warehouse2"
CREATE TABLE
```

创建一个有主键约束的表。

```
postgres=> CREATE TABLE warehouse3
postgres-# (
postgres(#      W_WAREHOUSE_SK            INTEGER                PRIMARY
KEY,
postgres(#      W_WAREHOUSE_ID            CHAR(16)               NOT NULL,
postgres(#      W_WAREHOUSE_NAME          VARCHAR(20)                ,
postgres(#      W_WAREHOUSE_SQ_FT         INTEGER                    ,
postgres(#      W_STREET_NUMBER           CHAR(10)                   ,
postgres(#      W_STREET_NAME             VARCHAR(60)                ,
postgres(#      W_STREET_TYPE             CHAR(15)                   ,
postgres(#      W_SUITE_NUMBER            CHAR(10)                   ,
postgres(#      W_CITY                    VARCHAR(60)                ,
postgres(#      W_COUNTY                  VARCHAR(30)                ,
postgres(#      W_STATE                   CHAR(2)          DEFAULT 'GA',
postgres(#      W_ZIP                     CHAR(10)                   ,
postgres(#      W_COUNTRY                 VARCHAR(20)                ,
postgres(#      W_GMT_OFFSET              DECIMAL(5,2)
postgres(# );
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "warehouse3_pkey" for
table "warehouse3"
CREATE TABLE
```

创建表，并指定该表数据不写入预写日志。

```
postgres=> CREATE UNLOGGED  TABLE warehouse4
postgres-# (
postgres(#      W_WAREHOUSE_SK            INTEGER                NOT NULL,
postgres(#      W_WAREHOUSE_ID            CHAR(16)               NOT NULL,
postgres(#      W_WAREHOUSE_NAME          VARCHAR(20)            UNIQUE,
postgres(#      W_WAREHOUSE_SQ_FT         INTEGER                    ,
postgres(#      W_STREET_NUMBER           CHAR(10)                   ,
postgres(#      W_STREET_NAME             VARCHAR(60)                ,
postgres(#      W_STREET_TYPE             CHAR(15)                   ,
postgres(#      W_SUITE_NUMBER            CHAR(10)                   ,
postgres(#      W_CITY                    VARCHAR(60)                ,
```

```
postgres(#      W_COUNTY                  VARCHAR(30)                 ,
postgres(#      W_STATE                   CHAR(2)           DEFAULT 'GA',
postgres(#      W_ZIP                     CHAR(10)                    ,
postgres(#      W_COUNTRY                 VARCHAR(20)                 ,
postgres(#      W_GMT_OFFSET              DECIMAL(5,2)
postgres(# );
NOTICE:  CREATE TABLE / UNIQUE will create implicit index
"warehouse4_w_warehouse_name_key" for table "warehouse4"
CREATE TABLE
```

创建临时表。

```
postgres=> CREATE TEMP TABLE warehouse5
postgres-# (
postgres(#      W_WAREHOUSE_SK            INTEGER                   NOT NULL,
postgres(#      W_WAREHOUSE_ID            CHAR(16)                  NOT NULL,
postgres(#      W_WAREHOUSE_NAME          VARCHAR(20)               UNIQUE,
postgres(#      W_WAREHOUSE_SQ_FT         INTEGER                   ,
postgres(#      W_STREET_NUMBER           CHAR(10)                  ,
postgres(#      W_STREET_NAME             VARCHAR(60)               ,
postgres(#      W_STREET_TYPE             CHAR(15)                  ,
postgres(#      W_SUITE_NUMBER            CHAR(10)                  ,
postgres(#      W_CITY                    VARCHAR(60)               ,
postgres(#      W_COUNTY                  VARCHAR(30)               ,
postgres(#      W_STATE                   CHAR(2)           DEFAULT 'GA',
postgres(#      W_ZIP                     CHAR(10)                  ,
postgres(#      W_COUNTRY                 VARCHAR(20)               ,
postgres(#      W_GMT_OFFSET              DECIMAL(5,2)
postgres(# );
NOTICE:  CREATE TABLE / UNIQUE will create implicit index
"warehouse5_w_warehouse_name_key" for table "warehouse5"
CREATE TABLE
```

创建带压缩的行存表。

```
postgres=> CREATE TABLE warehouse6
postgres-# (
postgres(#      W_WAREHOUSE_SK            INTEGER                   NOT NULL,
postgres(#      W_WAREHOUSE_ID            CHAR(16)                  NOT NULL,
postgres(#      W_WAREHOUSE_NAME          VARCHAR(20)               ,
postgres(#      W_WAREHOUSE_SQ_FT         INTEGER                   ,
postgres(#      W_STREET_NUMBER           CHAR(10)                  ,
postgres(#      W_STREET_NAME             VARCHAR(60)               ,
postgres(#      W_STREET_TYPE             CHAR(15)                  ,
postgres(#      W_SUITE_NUMBER            CHAR(10)                  ,
postgres(#      W_CITY                    VARCHAR(60)               ,
postgres(#      W_COUNTY                  VARCHAR(30)               ,
postgres(#      W_STATE                   CHAR(2)           DEFAULT 'GA',
postgres(#      W_ZIP                     CHAR(10)                  ,
postgres(#      W_COUNTRY                 VARCHAR(20)               ,
```

```
postgres(#      W_GMT_OFFSET                DECIMAL(5,2)
postgres(# ) COMPRESS;
```

NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'w_warehouse_sk' as the distribution column by default.

HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.

CREATE TABLE

创建带压缩的列存表。

```
postgres=> CREATE TABLE warehouse7
postgres-# (
postgres(#      W_WAREHOUSE_SK              INTEGER                 NOT NULL,
postgres(#      W_WAREHOUSE_ID              CHAR(16)                NOT NULL,
postgres(#      W_WAREHOUSE_NAME            VARCHAR(20)                 ,
postgres(#      W_WAREHOUSE_SQ_FT           INTEGER                     ,
postgres(#      W_STREET_NUMBER             CHAR(10)                    ,
postgres(#      W_STREET_NAME               VARCHAR(60)                 ,
postgres(#      W_STREET_TYPE               CHAR(15)                    ,
postgres(#      W_SUITE_NUMBER              CHAR(10)                    ,
postgres(#      W_CITY                      VARCHAR(60)                 ,
postgres(#      W_COUNTY                    VARCHAR(30)                 ,
postgres(#      W_STATE                     CHAR(2)             DEFAULT 'GA',
postgres(#      W_ZIP                       CHAR(10)                    ,
postgres(#      W_COUNTRY                   VARCHAR(20)                 ,
postgres(#      W_GMT_OFFSET                DECIMAL(5,2)
postgres(# ) WITH (ORIENTATION = COLUMN, COMPRESSION=HIGH);
```

NOTICE: The 'DISTRIBUTE BY' clause is not specified. Using 'w_warehouse_sk' as the distribution column by default.

HINT: Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.

CREATE TABLE

创建局部聚簇存储的列存表。

```
postgres=> CREATE TABLE warehouse8
postgres-# (
postgres(#      W_WAREHOUSE_SK              INTEGER                 NOT NULL,
postgres(#      W_WAREHOUSE_ID              CHAR(16)                NOT NULL,
postgres(#      W_WAREHOUSE_NAME            VARCHAR(20)                 ,
postgres(#      W_WAREHOUSE_SQ_FT           INTEGER                     ,
postgres(#      W_STREET_NUMBER             CHAR(10)                    ,
postgres(#      W_STREET_NAME               VARCHAR(60)                 ,
postgres(#      W_STREET_TYPE               CHAR(15)                    ,
postgres(#      W_SUITE_NUMBER              CHAR(10)                    ,
postgres(#      W_CITY                      VARCHAR(60)                 ,
postgres(#      W_COUNTY                    VARCHAR(30)                 ,
postgres(#      W_STATE                     CHAR(2)             DEFAULT 'GA',
postgres(#      W_ZIP                       CHAR(10)                    ,
postgres(#      W_COUNTRY                   VARCHAR(20)                 ,
postgres(#      W_GMT_OFFSET                DECIMAL(5,2),
postgres(#      PARTIAL CLUSTER KEY(W_WAREHOUSE_SK, W_WAREHOUSE_ID)
```

```
postgres(# ) WITH (ORIENTATION = COLUMN);
NOTICE:  The 'DISTRIBUTE BY' clause is not specified. Using 'w_warehouse_sk' as the distribution column by default.
HINT:  Please use 'DISTRIBUTE BY' clause to specify suitable data distribution column.
CREATE TABLE
# 定义一个检查列约束。
postgres=> CREATE TABLE warehouse9
postgres-# (
postgres(#     W_WAREHOUSE_SK            INTEGER    PRIMARY KEY CHECK (W_WAREHOUSE_SK > 0),
postgres(#     W_WAREHOUSE_ID            CHAR(16)                NOT NULL,
postgres(#     W_WAREHOUSE_NAME          VARCHAR(20) CHECK (W_WAREHOUSE_NAME IS NOT NULL),
postgres(#     W_WAREHOUSE_SQ_FT         INTEGER                         ,
postgres(#     W_STREET_NUMBER           CHAR(10)                        ,
postgres(#     W_STREET_NAME             VARCHAR(60)                     ,
postgres(#     W_STREET_TYPE             CHAR(15)                        ,
postgres(#     W_SUITE_NUMBER            CHAR(10)                        ,
postgres(#     W_CITY                    VARCHAR(60)                     ,
postgres(#     W_COUNTY                  VARCHAR(30)                     ,
postgres(#     W_STATE                   CHAR(2)           DEFAULT 'GA',
postgres(#     W_ZIP                     CHAR(10)                        ,
postgres(#     W_COUNTRY                 VARCHAR(20)                     ,
postgres(#     W_GMT_OFFSET              DECIMAL(5,2)
postgres(# );
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index "warehouse9_pkey" for table "warehouse9"
CREATE TABLE
# 增加列。
postgres=> ALTER TABLE warehouse9 ADD W_GOODS_CATEGORY varchar(30);
ALTER TABLE
# 在一个操作中改变两个现存字段的类型。
postgres=> ALTER TABLE warehouse9
postgres-# ALTER COLUMN W_GOODS_CATEGORY TYPE varchar(80),
postgres-# ALTER COLUMN W_STREET_NAME TYPE varchar(100);
ALTER TABLE
# 给一个已存在字段添加非空约束。
postgres=> ALTER TABLE warehouse9 ALTER COLUMN W_GOODS_CATEGORY SET NOT NULL;
ALTER TABLE
# 移除已存在字段的非空约束。
postgres=> ALTER TABLE warehouse9 ALTER COLUMN W_GOODS_CATEGORY DROP NOT NULL;
ALTER TABLE
# 表中删除一个字段。
postgres=> ALTER TABLE warehouse9 DROP COLUMN W_STREET_NAME;
```

```
ALTER TABLE
```

修改表名。

```
postgres=> ALTER TABLE warehouse9 RENAME TO warehouse9_1;
ALTER TABLE
```

步骤 4 索引操作

对行存普通表创建 btree 索引。

```
postgres=> CREATE INDEX index_warehouse_row_1 ON warehouse_row_1 USING BTREE(w_warehouse_id);
CREATE INDEX
```

对行存分区表创建 btree 索引。

```
postgres=> CREATE INDEX index_warehouse_row_2 ON warehouse_row_2 USING BTREE(w_warehouse_id)
LOCAL
postgres-# (
postgres(#       PARTITION p1_index_par,
postgres(#       PARTITION p2_index_par,
postgres(#       PARTITION p3_index_par,
postgres(#       PARTITION p4_index_par,
postgres(#       PARTITION p5_index_par,
postgres(#       PARTITION p6_index_par
postgres(# );
CREATE INDEX
```

对列存表创建 psort 索引。

```
postgres=> CREATE INDEX index_warehouse_col ON warehouse_col USING PSORT (w_warehouse_id);
CREATE INDEX
```

步骤 5 创建视图

基于表创建视图。

```
postgres=> CREATE VIEW warehouse_view AS
postgres-#       SELECT * FROM warehouse_col WHERE w_warehouse_sk <=100;
CREATE VIEW
```

步骤 6 清理环境

```
postgres=> DROP VIEW warehouse_view ;
DROP VIEW
postgres=> DROP TABLE warehouse;
DROP TABLE
postgres=> DROP TABLE warehouse2;
DROP TABLE
postgres=> DROP TABLE warehouse3;
DROP TABLE
postgres=> DROP TABLE warehouse4;
DROP TABLE
postgres=> DROP TABLE warehouse6;
DROP TABLE
```

```
postgres=> DROP TABLE warehouse7;
DROP TABLE
postgres=> DROP TABLE warehouse8;
DROP TABLE
postgres=> DROP TABLE warehouse9_1;
DROP TABLE
postgres=> DROP TABLE warehouse_col;
DROP TABLE
postgres=> DROP TABLE warehouse_row_1;
DROP TABLE
postgres=> DROP TABLE warehouse_row_2;
DROP TABLE
```

8.2 数据导入

8.2.1 实验介绍

8.2.1.1 关于本实验

本实验通过 INSERT 和外表方式，实现将上传到 OBS 桶中的数据导入到 GaussDB(DWS)数据库中。

8.2.1.2 实验目的

- 掌握 OBS 桶导入数据库的方式
- 掌握外表导入数据的方法

8.2.2 实验任务

步骤 1　数据文件上传至 OBS

通过 OBS 客户端，将数据文件 customer.dat 上传至 OBS 桶。

步骤 2　创建外表

```
postgres=> create foreign table foreigntb_customer( c_customer_sk
integer                         ,
postgres(> c_customer_id          char(16)                ,
postgres(> c_current_cdemo_sk     integer                 ,
postgres(> c_current_hdemo_sk     integer                 ,
postgres(> c_current_addr_sk      integer                 ,
postgres(> c_first_shipto_date_sk integer                 ,
postgres(> c_first_sales_date_sk  integer                 ,
postgres(> c_salutation           char(10)                ,
postgres(> c_first_name           char(20)                ,
postgres(> c_last_name            char(30)                ,
postgres(> c_preferred_cust_flag  char(1)                 ,
```

```
postgres(> c_birth_day                integer                ,
postgres(> c_birth_month              integer                ,
postgres(> c_birth_year               integer                ,
postgres(> c_birth_country            varchar(20)            ,
postgres(> c_login                    char(13)               ,
postgres(> c_email_address            char(50)               ,
postgres(> c_last_review_date         char(10))
postgres-> SERVER gsmpp_server
postgres-> OPTIONS (
postgres(> location 'obs://shujuqianyi/hcipgaussdbolap/customer',
postgres(> format 'text',
postgres(> encoding 'utf8',
postgres(> ACCESS_KEY 'FBXTUKQZUXYCEIGMMYHI',
postgres(> SECRET_ACCESS_KEY '6x3W05yfmtK7Y1NjuWI2RdxVo6ButqPzxDm75G82',delimiter ',') read only;
```

结果：CREATE FOREIGN TABLE

创建成功

步骤 3　创建目标表

```
postgres=> CREATE TABLE customer
postgres-# (     c_customer_sk             integer                 not null    ,
postgres(#       c_customer_id             char(16)                not null    ,
postgres(#       c_current_cdemo_sk        integer                             ,
postgres(#       c_current_hdemo_sk        integer                             ,
postgres(#       c_current_addr_sk         integer                             ,
postgres(#       c_first_shipto_date_sk    integer                             ,
postgres(#       c_first_sales_date_sk     integer                             ,
postgres(#       c_salutation              char(10)                            ,
postgres(#       c_first_name              char(20)                            ,
postgres(#       c_last_name               char(30)                            ,
postgres(#       c_preferred_cust_flag     char(1)                             ,
postgres(#       c_birth_day               integer                             ,
postgres(#       c_birth_month             integer                             ,
postgres(#       c_birth_year              integer                             ,
postgres(#       c_birth_country           varchar(20)                         ,
postgres(#       c_login                   char(13)                            ,
postgres(#       c_email_address           char(50)                            ,
postgres(#       c_last_review_date        char(10)
postgres(# ) DISTRIBUTE BY HASH(c_customer_sk);
```

结果：CREATE TABLE

创建成功

步骤 4　查询外表数据量

```
postgres=> select count(*) from foreigntb_customer;
```

结果：

count

```
--------
 100000
(1 row)
```

步骤 5　数据导入目标表

```
postgres=> INSERT INTO customer SELECT * FROM foreigntb_customer;
```

结果：INSERT 0 100000

数据导入成功

步骤 6　清理环境

```
postgres=> drop FOREIGN table foreigntb_customer;
DROP FOREIGN TABLE
postgres=>
postgres=> drop table customer;
DROP TABLE
```

8.3 增删改查

8.3.1 实验介绍

8.3.1.1 关于本实验

通过操作数据库表数据的增删改查(INSERT/DELETE/UPDATE/SELECT)，掌握 GaussDB(DWS)的 SQL 语法。

8.3.1.2 实验目的

- 掌握 GaussDB(DWS)的 SQL 语法

8.3.2 实验任务

步骤 1　创建表

```
create table emp(
        gh number(4),          --工号
        xm varchar2(20),      --姓名
        sex char(10),          --性别
        birthday date,         --生日
        sal number(7, 2),     --薪水
        level int                  --级别
)
with (orientation = column,compression=middle)
distribute by hash (gh);
```

步骤 2　插入数据

```
insert into emp values(1, 'Jack', 'male', '1989-05-01', 12500, 14);
insert into emp values(2, 'Rose', 'female', '1986-09-01', 15000, 17);
insert into emp values(3, 'Tom', 'male', '1987-02-01', 11500, 16);
insert into emp values(4, 'Edward', 'male', '1988-03-01', 13000, 15);
insert into emp values(5, 'Lily', 'female', '1989-07-01', 12000, 15);
insert into emp values(6, 'Ken', 'male', '1990-10-01', 11000, 13);
insert into emp values(7, 'Leon', 'male', '1984-11-01', 11000, 17);
insert into emp values(8, 'John', 'male', '1990-05-01', 11000, 16);
insert into emp values(9, 'Mary', 'female', '1985-09-01', 18000, 18);
insert into emp values(10, 'Owen', 'male', '1990-10-01', 13500, 15);
```

步骤 3　删除数据

```
delete from emp where gh=3 and xm='Tom';
```

步骤 4　修改数据

```
postgres=> update emp set level=16 where gh=5 and xm='Lily';
```

步骤 5　查询数据

查询工号为 2 的人员信息；查询所有 16 级以上的人员信息，并按出生年月排序；统计 15 级以上人员平均薪水。

```
postgres=> select * from emp where gh=2;
postgres=> select * from emp where level >= 16 order by birthday;
postgres=> select avg(sal) from emp where level = 15;
```

步骤 6　清理环境

```
drop table emp;
```

8.4 用户权限设计

8.4.1 实验介绍

8.4.1.1 关于本实验

通过对用户角色和权限的管理，为不同权限用户提供不同的服务，防止数据库系统、数据被泄露、篡改、破坏。

8.4.1.2 实验目的

理解用户标识和认证

掌握用户角色管理

理解数据库访问控制

8.4.2 实验任务

步骤 1　创建角色

创建角色 A，拥有 createdb 系统权限和 tpcds.web_returns 表的查询权限。

```
postgres=> CREATE ROLE role_a WITH CREATEDB PASSWORD 'Bigdata@123';
CREATE ROLE
postgres=> GRANT USAGE ON SCHEMA tpcds TO role_a;
GRANT
postgres=> GRANT SELECT ON TABLE tpcds.web_returns to role_a;
GRANT
```

步骤 2　创建用户

创建用户 A，用户 B，将角色的权限赋予用户 A，B。

```
postgres=> CREATE USER user_a PASSWORD 'Bigdata@123';
CREATE ROLE
postgres=> CREATE USER user_b PASSWORD 'Bigdata@123';
CREATE ROLE
postgres=> GRANT role_a TO user_a;
GRANT ROLE
postgres=> GRANT role_a TO user_b;
GRANT ROLE
```

步骤 3　创建角色 B

同样的步骤，创建角色 B，拥有 sysadmin 权限；角色 C，拥有 auditadmin 权限和 tpcds.catalog_page 表的查询权限；用户 C，拥有角色 B 和 C 的权限。

```
postgres=> CREATE ROLE role_b WITH SYSADMIN PASSWORD 'Bigdata@123';
CREATE ROLE
postgres=> CREATE ROLE role_c WITH AUDITADMIN PASSWORD 'Bigdata@123';
CREATE ROLE
postgres=> GRANT USAGE ON SCHEMA tpcds TO role_c;
GRANT
postgres=> GRANT SELECT ON TABLE tpcds.catalog_page to role_c;
GRANT
postgres=>
postgres=> CREATE USER user_c PASSWORD 'Bigdata@123';
CREATE ROLE
postgres=> GRANT role_b TO user_c;
GRANT ROLE
postgres=> GRANT role_c TO user_c;
GRANT ROLE;
```

步骤 4　创建用户 SOD1

创建用户 SOD1，授予权限 SYSADMIN 和 CREATEROLE

```
postgres=> CREATE USER sod1 WITH SYSADMIN CREATEROLE PASSWORD 'Bigdata@123';
```

步骤 5　创建用户 SOD2

创建用户 SOD2：授予权限 SYSADMIN 和 AUDITADMIN

```
postgres=> CREATE USER sod2 WITH SYSADMIN AUDITADMIN PASSWORD 'Bigdata@123';
```

步骤 6　创建用户 SOD3/4/5

创建用户 SOD3、SOD4、SOD5：分别授予 SYSADMIN、CREATEROLE、AUDITADMIN

```
postgres=> CREATE USER sod3 WITH SYSADMIN PASSWORD 'Bigdata@123';
CREATE ROLE
postgres=> CREATE USER sod4 WITH CREATEROLE PASSWORD 'Bigdata@123';
CREATE ROLE
postgres=> CREATE USER sod5 WITH AUDITADMIN PASSWORD 'Bigdata@123';
CREATE ROLE
```

步骤 7　清理环境

```
postgres=> DROP USER user_a;
DROP ROLE
postgres=> DROP USER user_b;
DROP ROLE
postgres=> DROP USER user_c;
DROP ROLE
postgres=> DROP USER role_a cascade;
DROP ROLE
postgres=> DROP USER role_b cascade;
DROP ROLE
postgres=> DROP USER role_c cascade;
DROP ROLE
postgres=> DROP USER sod3;
DROP ROLE
postgres=> DROP USER sod4;
DROP ROLE
postgres=> DROP USER sod5;
DROP ROLE
```