openGauss的AI特性实践

实验指导手册

教师版

目录

[1 实验环境介绍 3](#_Toc70534801)

[1.1 实验介绍 3](#_Toc70534802)

[1.1.1 关于本实验 3](#_Toc70534803)

[1.1.2 读者知识背景 3](#_Toc70534804)

[1.1.3 实验设备介绍 3](#_Toc70534805)

[2 关卡一：openGauss数据安装及基本操作 4](#_Toc70534806)

[2.1 实验介绍 4](#_Toc70534807)

[2.1.1 关于本实验 4](#_Toc70534808)

[2.1.2 实验目的 4](#_Toc70534809)

[2.2 实验任务及步骤 4](#_Toc70534810)

[2.2.1 编译前准备 4](#_Toc70534811)

[2.2.2 数据库安装编译 6](#_Toc70534812)

[2.2.3 关卡验证 9](#_Toc70534813)

[2.2.4 思考题 9](#_Toc70534814)

[3 关卡二：openGauss数据导入及基本操作 10](#_Toc70534815)

[3.1 实验介绍 10](#_Toc70534816)

[3.1.1 关于本实验 10](#_Toc70534817)

[3.1.2 实验目的 10](#_Toc70534818)

[3.2 实验任务及步骤 10](#_Toc70534819)

[3.2.1 数据初始化 10](#_Toc70534820)

[3.2.2 列存表的创建与使用 14](#_Toc70534821)

[3.2.3 物化视图的使用 18](#_Toc70534822)

[3.2.4 关卡验证 22](#_Toc70534823)

[3.2.5 思考题 22](#_Toc70534824)

[4 关卡三：openGauss的AI4DB特性应用 24](#_Toc70534825)

[4.1 实验介绍 24](#_Toc70534826)

[4.1.1 关于本实验 24](#_Toc70534827)

[4.1.2 实验目的 24](#_Toc70534828)

[4.2 实验任务及步骤 24](#_Toc70534829)

[4.2.1 将X-Tuner 安装到系统中 24](#_Toc70534830)

[4.2.2 使用X-Tuner进行参数优化 25](#_Toc70534831)

[4.2.3 Index-advisor：索引推荐 28](#_Toc70534832)

[4.2.4 关卡验证 33](#_Toc70534833)

[4.2.5 思考题 33](#_Toc70534834)

[5 关卡四：openGauss的DB4AI特性应用 34](#_Toc70534835)

[5.1 实验介绍 34](#_Toc70534836)

[5.1.1 关于本实验 34](#_Toc70534837)

[5.1.2 实验目的 34](#_Toc70534838)

[5.2 实验任务及步骤 34](#_Toc70534839)

[5.2.1 DB4AI环境部署 34](#_Toc70534840)

[5.2.2 DB4AI特性应用 39](#_Toc70534841)

[5.2.3 关卡验证 41](#_Toc70534842)

[5.2.4 思考题 41](#_Toc70534843)

# 实验环境介绍

## 实验介绍

### 关于本实验

openGauss是关系型数据库，采用客户端/服务器，单进程多线程架构，支持单机和一主多备部署方式，备机可读，支持双机高可用和读扩展。

本实验主要内容为openEuler弹性云服务上安装部署openGauss数据库，并进行简单的数据库相关操作；完成数据库相关操作后，进行列存表的使用与物化视图的使用；后续进行openGauss的AI4DB的特性实验，提升数据库性能工作；最后完成openGauss的DB4AI特性实验，完成分类测试和回归测试算法。

### 读者知识背景

本课程为创新实践课，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

具有基本的数据库知识背景，同时熟悉华为云界面，了解基本Linux知识。

### 实验设备介绍

* 组网说明

本实验环境为华为云环境，需要购买openEuler弹性云服务。

* 设备介绍

为了满足openGauss安装部署实验需要，建议每套实验环境采用以下配置：

软件类型和配置描述如下：

实验设备配套关系

|  |  |
| --- | --- |
| **设备名称** | **设备型号** |
| Linux操作系统 | openEuler20.3LTS |
| 工具 | Putty WinSCP TPCH Madlib TensorFlow |
| Python | Python 3.7.X  Python 2.7.X |

* **实验依赖安装包下载**

**访问下述链接下载本实验依赖的安装包：**

<https://practicecourse.obs.myhuaweicloud.com/openGauss%E5%88%9B%E6%96%B0%E5%AE%9E%E8%B7%B5%E8%AF%BE%E5%AE%89%E8%A3%85%E5%8C%85.zip>

# 关卡一：openGauss数据安装及基本操作

## 实验介绍

### 关于本实验

本实验通过第三方工具，完成openGauss的编译安装。

### 实验目的

掌握第三方工具库编译；

掌握openGauss编译安装。

## 实验任务及步骤

### 编译前准备

用root用户名，使用之前设置的密码登录ECS。



创建openGauss数据库的安装用户omm及其属组dbgrp。

groupadd -g 1000 dbgrp

useradd -g dbgrp -u 1000 -d /home/omm omm

修改omm用户密码。

passwd omm

输入修改的omm用户密码，建议设置成复杂密码。

创建openGauss源码存放及openGauss安装路径。

mkdir -p /opt/software/openGauss/data

下载第三方编译库，社区针对 centos\_7.6\_x86\_64、openEuler20.03 LTS\_arm、openEuler20.03 LTS\_ x86\_64 三种架构及操作系统已经提供了编译好的二进制，对于这三种系统架构，可以直接使用社区提供的编译好的文件openGauss-third\_party\_binarylibs.tar.gz。

cd /opt/software

wget https://opengauss.obs.cn-south-1.myhuaweicloud.com/1.1.0/openGauss-third\_party\_binarylibs.tar.gz

解压下载好的第三方编译库，并重命名为binarylibs。

tar -zxvf openGauss-third\_party\_binarylibs.tar.gz

mv openGauss-third\_party\_binarylibs/ binarylibs/

下载openGauss源码。

git clone https://gitee.com/opengauss/openGauss-server.git

将[cmake-3.20.5-linux-aarch64.tar.gz](https://github.com/Kitware/CMake/releases/download/v3.20.0/cmake-3.20.0-linux-aarch64.tar.gz)包，上传至服务器/usr/local/src/下进行解压（可使用winscp工具）。

cd /usr/local/src/

tar -zxvf cmake-3.20.5-linux-aarch64.tar.gz

chmod 755 /usr/local/src/cmake-3.20.5-linux-aarch64/

使用yum安装依赖包。

yum install -y libaio-devel ncurses-devel pam-devel libffi-devel libtool libtool-devel libtool-ltdl openssl-devel bison golang flex dkms-2.6.1-5.oe1.noarch python3-devel patch

替换python版本，将python3的链接设置为python的。

cd /usr/bin

mv python python.bak

ln -s python3 /usr/bin/python

检查python的版本。

python -V

返回值为：Python 3.7.4

修改路径的用户属组及权限。

chown omm:dbgrp -R /opt/software

chmod 755 -R /opt/software

### 数据库安装编译

切换omm用户。

su - omm

设定omm用户的环境变量。

vi ~/.bash\_profile

添加环境变量

将以下环境变量内容，红色加粗部分根据实际安装环境进行设置，并写入.bash\_profile文件中。

# GaussDB Code-

**export GAUSSHOME=/opt/software/openGauss #the path for openGauss**

export PGDATA=$GAUSSHOME/data #the path for data

**export BINARY\_PATH=/opt/software/binarylibs #the path for binarylibs**

# GCC 7.3 compiler path

**export CMAKEROOT=/usr/local/src/cmake-3.20.5-linux-aarch64/ #the path for cmake**

export GCC\_PATH=$BINARY\_PATH/buildtools/openeuler\_aarch64/gcc7.3

export CC=$GCC\_PATH/gcc/bin/gcc

export CXX=$GCC\_PATH/gcc/bin/g++

export LD\_LIBRARY\_PATH=$GCC\_PATH/gcc/lib64:$GCC\_PATH/isl/lib:$GCC\_PATH/mpc/lib/:$GCC\_PATH/mpfr/lib/:$GCC\_PATH/gmp/lib/:$LD\_LIBRARY\_PATH

export PATH=$GCC\_PATH/bin:$CMAKEROOT/bin:$PATH

# Gauss DB install path

export PATH=$GAUSSHOME/bin:$PATH

export LD\_LIBRARY\_PATH=$GAUSSHOME/lib:$LD\_LIBRARY\_PATH

export GAUSS\_WARNING\_TYPE=2

export GS\_CLUSTER\_NAME=single

使环境变量生效

source ~/.bash\_profile

进入openGauss源码下，生成配置文件。

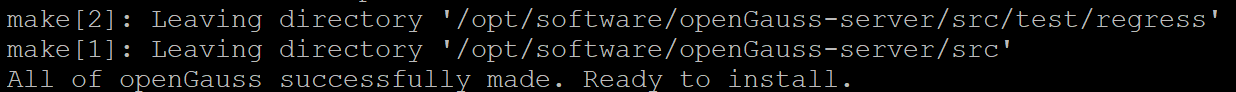
cd /opt/software/openGauss-server/

./configure --gcc-version=7.3.0 --prefix=$GAUSSHOME CFLAGS="-O0" --enable-debug --without-readline --without-zlib --enable-cassert --enable-thread-safety CC=g++ --3rd=$BINARY\_PATH --with-python

执行make命令进行编译

make -j4

当返回All of openGauss successfully made. Ready to install.时，表明编译成功。



执行make install安装。

make install

当返回openGauss installation complete.时，表明安装完成。

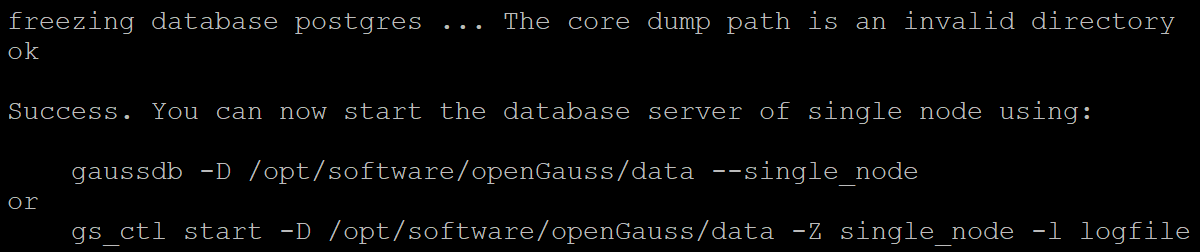
C:\Users\hwx559043\AppData\Roaming\eSpace_Desktop\UserData\hwx559043\imagefiles\E785BC09-F9C7-477E-8076-2E22F5647950.png

初始化数据库，需要设置数据库密码，请使用复杂密码，将{password}部分替换。

gs\_initdb -D $PGDATA --nodename=**hostname** --locale="en\_US.UTF-8" -Atrust -w **{password}**

例如（此处只是作为举例，建议设置为复杂密码）：

gs\_initdb -D $PGDATA --nodename=opengauss01 --locale="en\_US.UTF-8" -Atrust -w ‘Huawei#!13’



启动数据库。

gs\_ctl start -D /opt/software/openGauss/data -Z single\_node -l logfile

数据库登录。

gsql -d postgres -p 5432 -r

修改omm账号密码。

ALTER USER omm WITH PASSWORD '**自行修改密码**';

查询数据库版本。

select version();

version

----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

PostgreSQL 9.2.4 (GaussDB Kernel V500R001C20 build 4c77c0bb) compiled at 2021-03-30 17:12:43 commit 0 last mr debug on aarch64-unknown-linux-gnu, compiled by g++ (GCC) 7.3.0, 64-bit

(1 row)

退出数据库。

\q

### 关卡验证

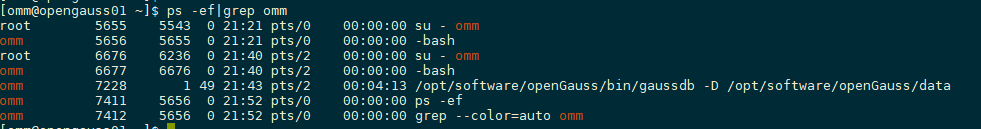
此处需要通过截图，来完成关卡的验证，首先需要对数据库状态进行验证。

gs\_ctl status



其次，对数据库进程进行截图验证，需包含数据库服务器的主机名。

ps -ef|grep omm



简单描述关卡一中，安装数据库所需要的步骤。完成以上信息，表明关卡一完成。

### 思考题

为什么需要通过源码编译，完成数据库的安装？简单描述安装数据库所需要的步骤。

答案：源码编译适用于不同的平台，如ARM架构和X86，在底层的汇编代码上就存在着差异，源码编译可以避免这种差异，同时源码编译时，可以根据具体的需求，选择不同的组件进行编译安装。

步骤请参考PPT内容。

# 关卡二：openGauss数据导入及基本操作

## 实验介绍

### 关于本实验

本实验关卡，在安装完成的openGauss数据库上，进行TPCH和target数据初始化操作；完成openGauss中的列存表的创建、修改与查询；物化视图的创建与使用。

### 实验目的

掌握openGauss数据库的基本维护方法；

掌握openGauss数据库列存表的创建与使用；

掌握物化视图的创建与使用。

## 实验任务及步骤

### 数据初始化

使用root登录，进入/opt/software目录下。

cd /opt/software

下载TPC-H测试包。

git clone <https://github.com/gregrahn/tpch-kit.git>

将tpch-hit目录的属组修改为omm用户。

chown omm:dbgrp -R /opt/software/tpch-kit/

切换至用户omm，并进入测试包的目录。

su - omm

cd /opt/software/tpch-kit/dbgen/

生成MakeFile文件。

make -f Makefile

连接openGauss数据库。

gsql -d postgres -p 5432

创建测试数据库tpch并退出。

CREATE DATABASE tpch;

\q

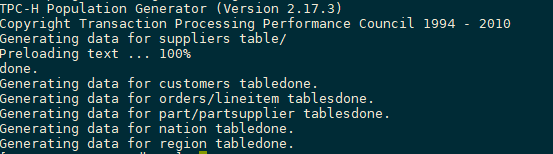
执行创建对象脚本

gsql tpch -f dss.ddl

产生测试数据

./dbgen -vf -s 1

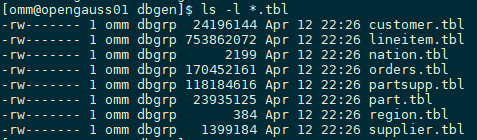
当返回如下内容，表示数据生产完成。



使用ls命令查看：

ls -l \*.tbl

回显生成的数据。



编辑加载脚本LoadData.sh。

vi LoadData.sh

将以下内容添加进LoadData.sh脚本中。

for i in `ls \*.tbl`; do

table=${i/.tbl/}

echo "Loading $table..."

sed 's/|$//' $i > /tmp/$i

gsql tpch -q -c "TRUNCATE $table"

gsql tpch -c "\\copy $table FROM '/tmp/$i' CSV DELIMITER '|'"

done

使用:wq退出并保存。

:wq

执行加载脚本LoadData.sh，将数据加载进数据库中。

sh LoadData.sh

返回结果：

Loading customer...

Loading lineitem...

Loading nation...

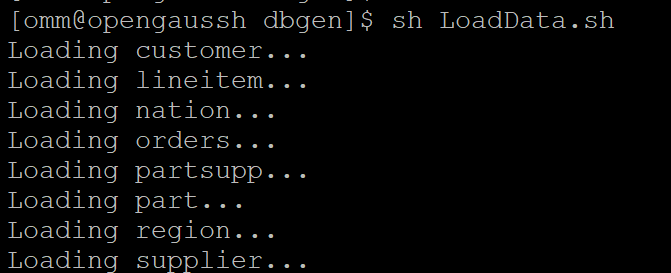
Loading orders...

Loading partsupp...

Loading part...

Loading region...

Loading supplier...



登录数据库验证，将查询结果截图。

gsql -d tpch -p 5432

select count(\*) from supplier;

此处应有返回记录，数据量为10000。

退出数据库。

\q

将相关查询拷贝进查询目录中，并进入queries/目录中。

cp dists.dss queries/

cp qgen queries/

cd queries/

生成查询语句。

vim genda.sh

将以下内容复制到genda.sh中。

for i in {1..22}; do

./qgen -d $i>$i\_new.sql

./qgen -d $i\_new | sed 's/limit -1//' | sed 's/day (3)/day/' > queries.sql

done

使用:wq退出并保存。

:wq

执行genda.sh脚本，生成查询语句

sh genda.sh

验证生成的查询语句

ls -l queries.sql

上传target.sql脚本至/opt/software目录下（可以使用winscp等工具）

切换至root用户，将脚本的属组修改为omm。

chown omm:dbgrp /opt/software/target.sql

将target.sql中的内容导入到tpch数据库中

gsql -d tpch -p 5432 -r -f /opt/software/target.sql > /opt/software/target.log

登录数据库进行验证

gsql -d tpch -p 5432

\dt

此处加上TPCH测试的表，总共有12张表。

### 列存表的创建与使用

创建与litemall\_orders表表结构相同的列存表。

gsql -d tpch -p 5432 -r

CREATE TABLE "public"."litemall\_orders\_col" (

"order\_id" INTEGER NOT NULL DEFAULT NULL,

"order\_sn" varchar(63) COLLATE "pg\_catalog"."default" DEFAULT NULL,

"order\_status" SMALLINT DEFAULT NULL,

"user\_key" INTEGER DEFAULT NULL,

"address\_key" INTEGER DEFAULT NULL,

"goods\_price" numeric(10,2) DEFAULT NULL,

"freight\_price" numeric(10,2) DEFAULT NULL,

"coupon\_price" numeric(10,2) DEFAULT NULL,

"integral\_price" numeric(10,2) DEFAULT NULL,

"groupon\_price" numeric(10,2) DEFAULT NULL,

"order\_price" numeric(10,2) DEFAULT NULL,

"actual\_price" numeric(10,2) DEFAULT NULL,

"pay\_id" varchar(63) COLLATE "pg\_catalog"."default" DEFAULT NULL,

"pay\_date" INTEGER DEFAULT NULL,

"pay\_time" INTEGER DEFAULT NULL,

"ship\_sn" varchar(63) COLLATE "pg\_catalog"."default" DEFAULT NULL,

"ship\_channel" varchar(63) COLLATE "pg\_catalog"."default" DEFAULT NULL,

"ship\_date" INTEGER DEFAULT NULL,

"ship\_time" INTEGER DEFAULT NULL,

"confirm\_date" INTEGER DEFAULT NULL,

"confirm\_time" INTEGER DEFAULT NULL,

"message" varchar(512) COLLATE "pg\_catalog"."default" DEFAULT NULL,

"comments" SMALLINT DEFAULT NULL,

"aftersale\_status" SMALLINT DEFAULT NULL,

"refund\_amount" numeric(10,2) DEFAULT NULL,

"refund\_type" varchar(63) COLLATE "pg\_catalog"."default" DEFAULT NULL,

"refund\_content" varchar(127) COLLATE "pg\_catalog"."default" DEFAULT NULL,

"refund\_date" INTEGER DEFAULT NULL,

"refund\_time" INTEGER DEFAULT NULL,

"end\_date" INTEGER DEFAULT NULL,

"end\_time" INTEGER DEFAULT NULL,

"add\_date" INTEGER DEFAULT NULL,

"add\_time" INTEGER DEFAULT NULL,

"update\_date" INTEGER DEFAULT NULL,

"update\_time" INTEGER DEFAULT NULL,

"deleted" TINYINT DEFAULT NULL

)

with (orientation=column );

将litemall\_orders表中的数据插入litemall\_orders\_col中。

insert into litemall\_orders\_col select \* from litemall\_orders;

使用\timing开启SQL执行时间的返回

\timing

使用SQL语句查询，2020年上半年litemall\_orders行存表中的order\_price的总和，并对结果和耗时截图。

select sum (order\_price) from litemall\_orders where add\_date between '20200101' and '20200701';

获得结果为：

sum

--------------

310586483.00

(1 row)

Time: 265.077 ms

使用SQL语句查询，2020年上半年litemall\_orders\_col列存表中的order\_price的总和，并对结果和耗时截图。

select sum (order\_price) from litemall\_orders\_col where add\_date between '20200101' and '20200701';

获得结果为：

sum

--------------

310586483.00

(1 row)

Time: 27.130 ms

使用SQL语句查询，2020年上半年litemall\_orders行存表中的order\_price的平均值，并对结果和耗时截图。

select avg(order\_price) from litemall\_orders where add\_date between '20200101' and '20200701';

获得结果为：

avg

-----------------------

3105.8648300000000000

(1 row)

Time: 369.490 ms

使用SQL语句查询，2020年上半年litemall\_orders\_col列存表中的order\_price的平均值，并对结果和耗时截图。

tpch=# select avg(order\_price) from litemall\_orders\_col where add\_date between '20200101' and '20200701';

获得结果为：

avg

-----------------------

3105.8648300000000000

(1 row)

Time: 16.704 ms

使用SQL语句查询，litemall\_orders行存表中order\_id为6的order\_price的值，并对结果和耗时截图。

select order\_price from litemall\_orders where order\_id=6;

获得结果为：

order\_price

-------------

2469.00

(1 row)

Time: 4.434 ms

使用SQL语句查询，litemall\_orders\_col列存表中order\_id为6的order\_price的值，并对结果和耗时截图。

select order\_price from litemall\_orders\_col where order\_id=6;

获得结果为：

order\_price

-------------

2469.00

(1 row)

Time: 5.663 ms

使用SQL语句，将litemall\_orders行存表中order\_id为6的order\_price修改为2468，并对结果和耗时截图。

update litemall\_orders set order\_price=2468 where order\_id=6;

获得结果为：

UPDATE 1

Time: 3.902 ms

使用SQL语句，将litemall\_orders\_col列存表中order\_id为6的order\_price修改为2468，并对结果和耗时截图。

update litemall\_orders\_col set order\_price=2468 where order\_id=6;

获得结果为：

UPDATE 1

Time: 36.075 ms

对比在行存表与列存表中的操作，根据每种SQL执行的时间，得出结论。

### 物化视图的使用

#### 全量物化视图

创建物化视图时需要的表，内容为查看消费金额大于100000的用户。

CREATE TABLE test\_view AS SELECT \* FROM (

SELECT ud.username,

gender,

SUM(o.actual\_price) AS totalspend

FROM litemall\_orders o, user\_dimension ud

WHERE o.user\_key = ud.user\_key

GROUP BY ud.username, ud.weixin\_openid, gender

ORDER BY SUM(o.actual\_price) DESC) WHERE totalspend > 100000

;

对表内容进行查询，将执行结果截图。

SELECT \* FROM test\_view;

返回结果：

username | gender | totalspend

----------+--------+------------

杨兰娟 | 2 | 119391.00

柳高梅 | 2 | 116155.00

韦小全 | 1 | 114072.00

贲艳梅 | 2 | 112565.00

强兰丽 | 2 | 111925.00

滑小刚 | 1 | 110089.00

席长梅 | 2 | 108247.00

翁晓婷 | 2 | 107988.00

娄高伟 | 1 | 107323.00

苏长刚 | 0 | 104640.00

喻高伟 | 1 | 102536.00

袁晓轩 | 1 | 101835.00

伏成峰 | 1 | 101725.00

毕晓刚 | 1 | 101057.00

金高芳 | 2 | 100322.00

(15 rows)

创建全量物化视图。

CREATE MATERIALIZED VIEW v\_order AS select count(\*) from test\_view;

使用物化视图统计人数，查询物化视图结果，将执行结果截图。

SELECT \* FROM v\_order;

返回结果：

count

-------

15

(1 row)

删除表中的数据。

DELETE FROM test\_view WHERE username='杨兰娟';

对全量物化视图做全量刷新。

REFRESH MATERIALIZED VIEW v\_order;

使用物化视图统计人数，查询物化视图结果，将执行结果截图。

SELECT \* FROM v\_order;

返回结果：

count

-------

14

(1 row)

删除物化视图。

DROP MATERIALIZED VIEW v\_order;

#### 增量物化视图

创建增量物化视图。

CREATE INCREMENTAL MATERIALIZED VIEW vi\_order AS SELECT username,totalspend FROM test\_view;

查询物化视图结果，将执行结果截图。

SELECT \* FROM vi\_order;

返回结果：

username | totalspend

----------+------------

柳高梅 | 116155.00

韦小全 | 114072.00

贲艳梅 | 112565.00

强兰丽 | 111925.00

滑小刚 | 110089.00

席长梅 | 108247.00

翁晓婷 | 107988.00

娄高伟 | 107323.00

苏长刚 | 104640.00

喻高伟 | 102536.00

袁晓轩 | 101835.00

伏成峰 | 101725.00

毕晓刚 | 101057.00

金高芳 | 100322.00

(14 rows)

插入数据。

INSERT INTO test\_view VALUES('杨兰娟',2 , 119391.00);

增量刷新物化视图。

REFRESH INCREMENTAL MATERIALIZED VIEW vi\_order;

查询物化视图结果，将执行结果截图。

SELECT \* FROM vi\_order;

返回结果：

username | totalspend

----------+------------

柳高梅 | 116155.00

韦小全 | 114072.00

贲艳梅 | 112565.00

强兰丽 | 111925.00

滑小刚 | 110089.00

席长梅 | 108247.00

翁晓婷 | 107988.00

娄高伟 | 107323.00

苏长刚 | 104640.00

喻高伟 | 102536.00

袁晓轩 | 101835.00

伏成峰 | 101725.00

毕晓刚 | 101057.00

金高芳 | 100322.00

杨兰娟 | 119391.00

(15 rows)

再次插入数据。

INSERT INTO test\_view VALUES('马景涛',2 , 139391.00);

全量刷新物化视图。

REFRESH MATERIALIZED VIEW vi\_order;

查询物化视图结果，将执行结果截图。

SELECT \* FROM vi\_order;

返回结果：

username | totalspend

----------+------------

柳高梅 | 116155.00

韦小全 | 114072.00

贲艳梅 | 112565.00

强兰丽 | 111925.00

滑小刚 | 110089.00

席长梅 | 108247.00

翁晓婷 | 107988.00

娄高伟 | 107323.00

苏长刚 | 104640.00

喻高伟 | 102536.00

袁晓轩 | 101835.00

伏成峰 | 101725.00

毕晓刚 | 101057.00

金高芳 | 100322.00

杨兰娟 | 119391.00

马景涛 | 139391.00

(16 rows)

删除物化视图。

DROP MATERIALIZED VIEW vi\_order;

对比全量物化视图与增量物化视图，输出结论。

### 关卡验证

本关卡需要将行存与列存执行时间对比进行截图，同时通过对比，得出使用两种不同的存储所适用的场景；需要将全量与增量物化视图的查询结果进行截图，得出全量物化视图与增量物化视图的区别。

### 思考题

行存表与列存表在执行相同的SQL语句时，为何执行的时间不同？在执行哪些类型SQL时，行存表效率更高？在执行哪些类型SQL时，列存表效率更高？全量物化视图与增量物化视图有哪些差别？

答案：因为行存和列存在存储的方式不同，因此在不同的场景下，执行效率有一定的差异。

行存表在精准查询和单列增删改查时，效率更高；列存表在聚合类的查询下，效率更高。

全量物化视图可以接复杂查询，而增量物化视图，只支持简单查询。

# 关卡三：openGauss的AI4DB特性应用

## 实验介绍

### 关于本实验

本实验关卡，在安装完成的openGauss数据库上，使用AI4DB的功能，对数据库参数进行优化；对TPCH测试中的SQL语句进行优化。

### 实验目的

掌握openGauss数据库的AI4DB的功能；

掌握openGauss数据库的参数修改方法；

掌握openGauss数据库的SQL优化能力；

掌握openGauss数据库的索引创建与使用方法。

## 实验任务及步骤

### 将X-Tuner 安装到系统中

使用root用户，安装hdf5依赖包。

yum -y install hdf5-devel

将tensorflow-2.4.0-cp37-none-linux\_aarch64.whl文件上传至服务器/opt/software下，使用root用户，安装TensorFlow。

cd /opt/software

pip3 install tensorflow-2.4.0-cp37-none-linux\_aarch64.whl

出现Successfully回显，表明安装完成。

使用root用户，进入到xtuner的安装目录。

cd /opt/software/openGauss/bin/dbmind/xtuner/

安装X-Tuner。

python setup.py install

安装过程中会出现很多warning，忽略即可，出现Finished回显，表明安装完成



### 使用X-Tuner进行参数优化

使用omm用户登录，将环境变量加入omm用户的.bashrc文件中。

vi ~/.bashrc

将以下内容加入并保存退出。

export GAUSSHOME=/opt/software/openGauss

export PATH=$GAUSSHOME/bin:$PATH

export LD\_LIBRARY\_PATH=$GAUSSHOME/lib:$LD\_LIBRARY\_PATH

执行TPCH脚本，获得测试时间，将执行结果截图。

gsql -d tpch -p 5432 -r -f /opt/software/tpch-kit/dbgen/queries/queries.sql > /opt/software/tpch-kit/dbgen/queries/queries01.log

查看queries01.log。

tail -10 /opt/software/tpch-kit/dbgen/queries/queries01.log

得到结果如下，将结果截图：

total time: 581509 ms

使用root用户，执行X-Tuner进行参数建议优化，将执行结果截图。

需要输入正确的omm数据库用户和omm操作系统用户密码（密码为关卡一中创建omm时设置的用户密码）。

gs\_xtuner recommend --db-name tpch --db-user omm --port 5432 --host 127.0.0.1 --host-user omm

得到结果如下：

INFO:

+---------------------------------------+-----------------------+

| Metric | Value |

+---------------------------------------+-----------------------+

| workload\_type | ap |

| dirty\_background\_bytes | 0 |

| current\_locks\_count | 0.0 |

| current\_prepared\_xacts\_count | 0.0 |

| rollback\_commit\_ratio | 0.0 |

| write\_tup\_speed | 0.0 |

| average\_connection\_age | 0.014932 |

| checkpoint\_avg\_sync\_time | 0.0333333333333333 |

| checkpoint\_proactive\_triggering\_ratio | 0.0666666666666667 |

| uptime | 0.479763446944444 |

| fetched\_returned\_ratio | 0.5852240193485715 |

| cache\_hit\_rate | 0.9854683811359706 |

| current\_connections | 1.0 |

| search\_modify\_ratio | 1084000.0 |

| read\_write\_ratio | 10840000.0 |

| all\_database\_size | 1539913.41015625 |

| temp\_file\_size | 1617.81002869898 |

| os\_cpu\_count | 4 |

| checkpoint\_dirty\_writing\_time\_window | 450.0 |

| read\_tup\_speed | 6.2826238027896 |

| current\_free\_mem | 6078976 |

| shared\_buffer\_toast\_hit\_rate | 61.904761904761905 |

| os\_mem\_total | 6976512 |

| max\_processes | 74 |

| track\_activity\_size | 74.0 |

| used\_mem | 77602911.75 |

| ap\_index | 8.0 |

| block\_size | 8.0 |

| shared\_buffer\_tidx\_hit\_rate | 92.72727272727273 |

| shared\_buffer\_heap\_hit\_rate | 98.56054191363252 |

| shared\_buffer\_idx\_hit\_rate | 99.14454277286136 |

| is\_hdd | False |

| enable\_autovacuum | True |

| is\_64bit | True |

| load\_average | [0.115, 0.16, 0.1375] |

+---------------------------------------+-----------------------+

p.s: The unit of storage is kB.

WARN:

[0]. The total size of all databases is less than the memory size. Therefore, it is unnecessary to set shared\_buffers to a large value.

[1]. The number of CPU cores is a little small. Please do not run too high concurrency. You are recommended to set max\_connections based on the number of CPU cores. If your job does not consume much CPU, you can also increase it.

[2]. The value of wal\_buffers is a bit high. Generally, an excessively large value does not bring better performance. You can also set this parameter to -1. The database automatically performs adaptation.

[3]. The database runs for a short period of time, and the database description may not be accumulated. The recommendation result may be inaccurate.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Recommended Knob Settings \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

+---------------------------+-----------+--------+---------+---------+

| name | recommend | min | max | restart |

+---------------------------+-----------+--------+---------+---------+

| shared\_buffers | 192488 | 130809 | 221361 | True |

| max\_connections | 71 | 15 | 143 | True |

| effective\_cache\_size | 5232384 | 192488 | 5232384 | False |

| effective\_io\_concurrency | 200 | 150 | 250 | False |

| wal\_buffers | 6015 | 2048 | 6015 | True |

| random\_page\_cost | 1.0 | 1.0 | 2.0 | False |

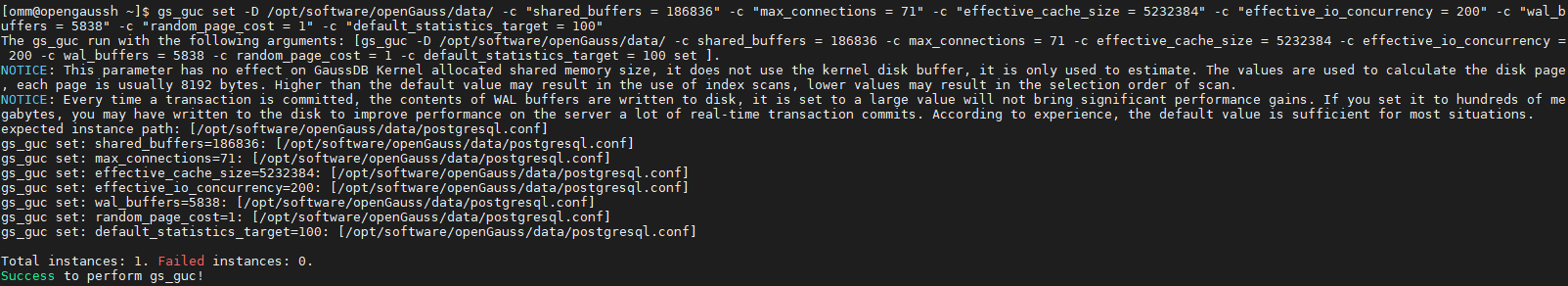
| default\_statistics\_target | 100 | 80 | 300 | False |

+---------------------------+-----------+--------+---------+---------+

使用omm用户，对以上参数进行优化。

gs\_guc set -D /opt/software/openGauss/data/ -c "shared\_buffers = 192488" -c "max\_connections = 71" -c "effective\_cache\_size = 5232384" -c "effective\_io\_concurrency = 200" -c "wal\_buffers = 6015" -c "random\_page\_cost = 1" -c "default\_statistics\_target = 100"

返回结果为：



使用omm用户，重启数据库。

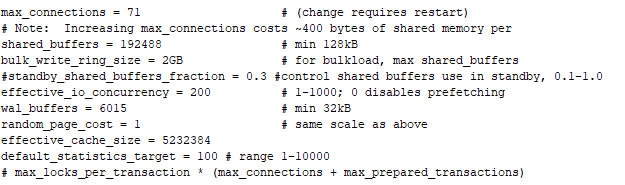
gs\_ctl stop

gs\_ctl start -D /opt/software/openGauss/data -Z single\_node -l logfile

获取参数值，将结果截图。

cd /opt/software/openGauss/data

cat postgresql.conf|grep -E 'shared\_buffers|max\_connections|effective\_cache\_size|effective\_io\_concurrency|wal\_buffers|random\_page\_cost|default\_statistics\_target'



### Index-advisor：索引推荐

登录数据库。

gsql -d tpch -p 5432

使用SQL查询2020年3月订单收入，并进行排序。

SELECT ad.province AS province, SUM(o.actual\_price) AS GMV

FROM litemall\_orders o,

address\_dimension ad,

date\_dimension dd

WHERE o.address\_key = ad.address\_key

AND o.add\_date = dd.date\_key

AND dd.year = 2020

AND dd.month = 3

GROUP BY ad.province

ORDER BY SUM(o.actual\_price) DESC;

使用explain，对该SQL加以分析，将结果截图。

EXPLAIN

SELECT ad.province AS province, SUM(o.actual\_price) AS GMV

FROM litemall\_orders o,

address\_dimension ad,

date\_dimension dd

WHERE o.address\_key = ad.address\_key

AND o.add\_date = dd.date\_key

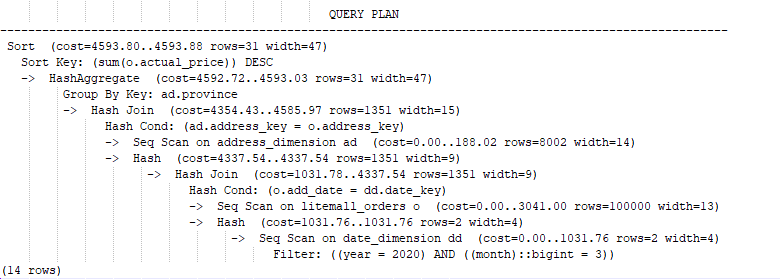
AND dd.year = 2020

AND dd.month = 3

GROUP BY ad.province

ORDER BY SUM(o.actual\_price) DESC;

获得执行计划结果为：



使用索引推荐功能，对查询语句进行推荐，将执行结果截图。

select \* from gs\_index\_advise('

SELECT ad.province AS province, SUM(o.actual\_price) AS GMV

FROM litemall\_orders o,

address\_dimension ad,

date\_dimension dd

WHERE o.address\_key = ad.address\_key

AND o.add\_date = dd.date\_key

AND dd.year = 2020

AND dd.month = 3

GROUP BY ad.province

ORDER BY SUM(o.actual\_price) DESC');

获得索引推荐结果如下：

table | column

-------------------+------------------------

litemall\_orders | (address\_key,add\_date)

address\_dimension |

date\_dimension | (year)

在litemall\_orders和date\_dimension表上创建虚拟索引列。

select \* from hypopg\_create\_index('create index on litemall\_orders(address\_key,add\_date)');

select \* from hypopg\_create\_index('create index on date\_dimension(year)');

查看创建的虚拟索引列，将执行结果截图。

select \* from hypopg\_display\_index();

获得创建虚拟列的结果：

indexname | indexrelid | table | column

---------------------------------------------------+------------+-----------------+-------------------------

<16715>btree\_litemall\_orders\_address\_key\_add\_date | 16715 | litemall\_orders | (address\_key, add\_date)

<16716>btree\_date\_dimension\_year | 16716 | date\_dimension | (year)

获取索引虚拟列大小结果（单位为：字节），将执行结果截图。

其中16715和16716为上一步中查询到的indexrelid，此处查询需要进行替换成自己系统中查询到的值。

select \* from hypopg\_estimate\_size(16715);

select \* from hypopg\_estimate\_size(16716);

返回结果为：

hypopg\_estimate\_size

----------------------

4177920

(1 row)

hypopg\_estimate\_size

----------------------

696320

(1 row)

开启GUC参数enable\_hypo\_index，

该参数控制数据库的优化器进行EXPLAIN时是否考虑创建的虚拟索引。通过对特定的查询语句执行explain，用户可根据优化器给出的执行计划评估该索引是否能够提升该查询语句的执行效率。

set enable\_hypo\_index = on;

再次使用explain，对该SQL加以分析，将执行结果截图。

EXPLAIN

SELECT ad.province AS province, SUM(o.actual\_price) AS GMV

FROM litemall\_orders o,

address\_dimension ad,

date\_dimension dd

WHERE o.address\_key = ad.address\_key

AND o.add\_date = dd.date\_key

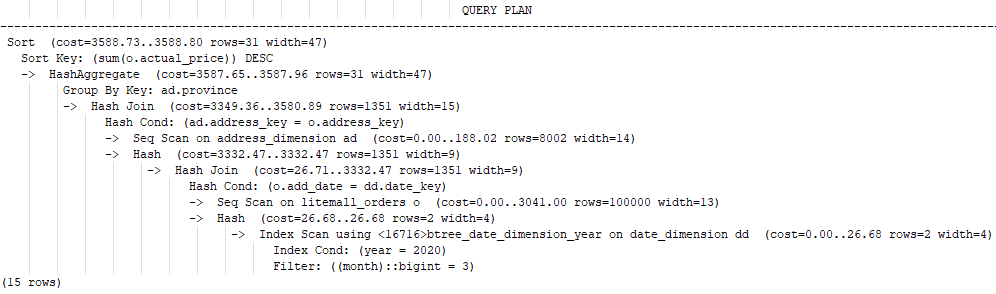
AND dd.year = 2020

AND dd.month = 3

GROUP BY ad.province

ORDER BY SUM(o.actual\_price) DESC;

获得执行计划：



删除某一个索引虚拟列，将执行结果截图。

select \* from hypopg\_drop\_index(16715);

返回结果：

hypopg\_drop\_index

-------------------

f

(1 row)

删除所有索引虚拟列，将执行结果截图。

select \* from hypopg\_reset\_index();

返回结果：

hypopg\_reset\_index

--------------------

(1 row)

查看索引虚拟列，将执行结果截图。

select \* from hypopg\_display\_index();

返回结果：

indexname | indexrelid | table | column

-----------+------------+-------+--------

(0 rows)

针对执行的queries.sql进行优化，为customer、order、part表增加主键索引，学员也可以针对queries.sql中的查询内容进行进一步优化。

ALTER TABLE "public"."customer" ADD CONSTRAINT "pk\_customer" PRIMARY KEY ("c\_custkey");

ALTER TABLE "public"."orders" ADD CONSTRAINT "pk\_orders" PRIMARY KEY ("o\_orderkey");

ALTER TABLE "public"."part" ADD CONSTRAINT "pk\_part" PRIMARY KEY ("p\_partkey");

在order表的o\_orderdate列，以及lineitem表的l\_shipdate列上创建索引，可以通过分析queries.sql查询，进行更深入的优化。

CREATE INDEX idx\_o\_orderdate ON orders(o\_orderdate);

CREATE INDEX idx\_l\_shipdate ON lineitem(l\_shipdate);

重新执行queries.sql查询，将执行结果截图。

gsql -d tpch -p 5432 -r -f /opt/software/tpch-kit/dbgen/queries/queries.sql > /opt/software/tpch-kit/dbgen/queries/queries02.log

查询queries02.log文件：

tail -10 /opt/software/tpch-kit/dbgen/queries/queries02.log

得到结果：

total time: 474229 ms

### 关卡验证

使用X-Tuner完成参数优化建议，并针对建议值完成openGauss数据库参数优化，对结果进行截图；完成索引推荐功能的优化，使用索引推荐以及虚拟索引列的创建，对创建的索引虚拟列前后的执行时间进行对比截图；完成对queries查询的优化，对比优化前后的区别。

### 思考题

根据X-Tuner给出的参数优化，在哪些参数上进行了优化，为何要对这些参数进行优化？索引的使用，对于执行SQL有什么好处？除了使用索引和参数外，还有哪些方面可以对数据库进行优化？

答案：在对数据库参数优化时，主要针对数据库的内存参数、连接数等参数上进行优化，主要是围绕当前数据库，能够更加充分利用主机资源做的优化。

索引的使用能够加快检索速度。

一方面可以通过增加主机资源，如增加CPU、提高IO能力、增加带宽等方式，还可以通过表分区、分片、历史数据归档等方式，来降低数据量。

# 关卡四：openGauss的DB4AI特性应用

## 实验介绍

### 关于本实验

本实验关卡，在安装完成的openGauss数据库上，使用DB4AI的功能，利用openGauss的DB4AI性能进行某产品销售情况的预测；利用openGauss的DB4AI性能进行客户区域的分类。

### 实验目的

掌握部署madlib框架；

掌握openGauss数据库的DB4AI的功能。

## 实验任务及步骤

### DB4AI环境部署

使用omm用户，清理openGauss环境，关闭数据库。

因为DB4AI特性是在python2下编译安装，所以数据库需要重新进行编译安装，同时数据也需要重新导入。首先使用omm用户关闭当前的数据库。

gs\_ctl stop

使用root用户，清理openGauss安装目录。

rm -rf /opt/software/openGauss/{bin,data,etc,include,jre,lib,logs,share}

更换python版本，将python版本更改回python2。

cd /usr/bin

mv python python.bak3

mv python.bak python

验证python版本。

python -V

返回结果为：

Python 2.7.16

使用omm用户进入源码目录，并使用git clean命令清理变量。

cd /opt/software/openGauss-server

git clean -xdf

进入数据库源码，重新生成配置文件，并编译安装。

./configure --gcc-version=7.3.0 --prefix=$GAUSSHOME CFLAGS="-O0" --enable-debug --without-readline --without-zlib --enable-cassert --enable-thread-safety CC=g++ --3rd=$BINARY\_PATH --with-python --with-includes=/usr/local/include/python2.7

make -j4

make install

检查数据库安装版本。

cd /opt/software/openGauss/lib/postgresql

ls -l plpython\*

返回结果为（表明使用的python2编译安装的）：



初始化数据库，并启动数据库。

gs\_initdb -D $PGDATA --nodename=hostname --locale="en\_US.UTF-8" -Atrust -w {password}

例如：

gs\_initdb -D $PGDATA --nodename=opengauss01 --locale="en\_US.UTF-8" -Atrust -w ‘Huawei#!13’

启动数据库：

gs\_ctl start -D /opt/software/openGauss/data -Z single\_node -l logfile

登录数据库，并创建tpch数据库。

gsql -d postgres -p 5432

CREATE DATABASE tpch dbcompatibility='B';

恢复数据，可以参考关卡二中，将tpch和target脚本的数据进行恢复。

使用root用户安装Madlib，上传安装包。

将apache-madlib-1.17.0-src.tar.gz文件上传到服务器/opt/software下解压。

cd /opt/software

tar -zxvf apache-madlib-1.17.0-src.tar.gz

mkdir -p /opt/software/apache-madlib-1.17.0-src/build/third\_party/downloads

将madlib.patch上传到/opt/software/apache-madlib-1.17.0-src路径下。

将PyXB-1.2.6.tar.gz和boost\_1\_61\_0.tar.gz上传至/opt/software/apache-madlib-1.17.0-src/build/third\_party/downloads路径下。

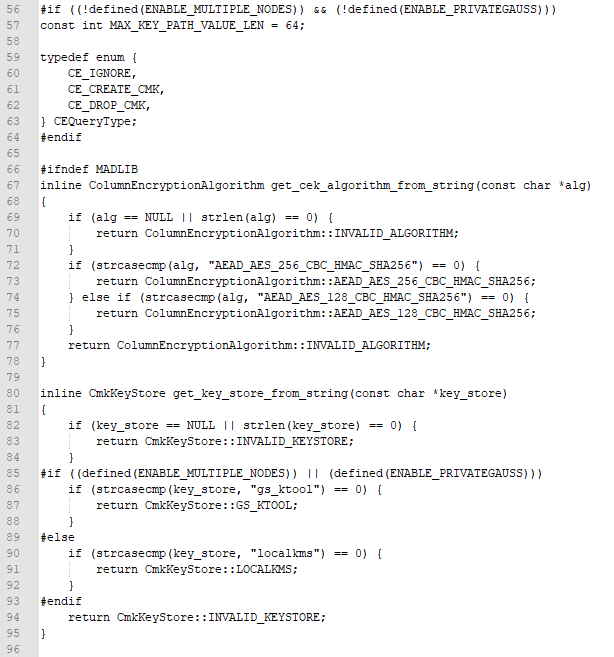
chown -R omm.dbgrp /opt/software/apache-madlib-1.17.0-src

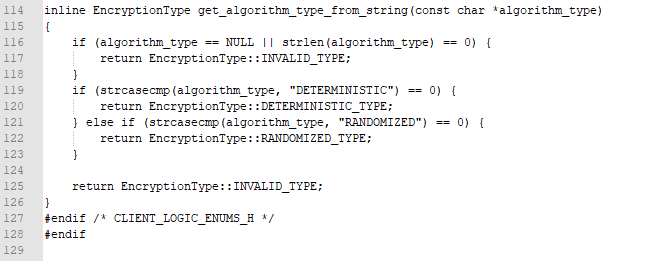
cd /opt/software/apache-madlib-1.17.0-src/build/third\_party/downloads

ls -l \*.gz

使用omm账户修改client\_logic\_enums.h文件，在inline ColumnEncryptionAlgorithm定义前，添加#ifndef MADLIB内容，以及在脚本尾部添加#endif。

vi /opt/software/openGauss/include/postgresql/server/client\_logic/client\_logic\_enums.h





使用omm用户，patch补丁。

su - omm

cd /opt/software/apache-madlib-1.17.0-src

patch -p1 < madlib.patch

使用omm账号在apache-madlib-1.17.0-src目录下进行编译安装。

./configure -DCMAKE\_INSTALL\_PREFIX=/opt/software/madlib -DPOSTGRESQL\_EXECUTABLE=$GAUSSHOME/bin/ -DPOSTGRESQL\_9\_2\_EXECUTABLE=$GAUSSHOME/bin/ -DPOSTGRESQL\_9\_2\_CLIENT\_INCLUDE\_DIR=$GAUSSHOME/bin/ -DPOSTGRESQL\_9\_2\_SERVER\_INCLUDE\_DIR=$GAUSSHOME/bin/

make

make install

进入Madlib的安装目录bin下，将算法安装到数据库中。

Madlib初始化的命令如下：

./madpack -s <SCHEMA\_NAME> -p opengauss -c <USER\_NAME>@127.0.0.1:<PORT>/<DATABASE\_NAME> install

命令中参数说明如下：

-s：schema的名称。

-p：数据库平台，使用opengauss即可。

-c：连接数据库的参数。包括用户名、‘@’、IP地址、端口号和目标数据库名称。

install为安装的命令，除此之外，还有reinstall（重新安装），uninstall（卸载）等命令可用。

cd /opt/software/madlib/bin

./madpack -s madlib -p opengauss -c omm@127.0.0.1:5432/tpch install

登录tpch数据库，查看schema，存在madlib表示安装完成，将执行结果截图。

gsql -d tpch -p 5432

\dn

返回结果为：

List of schemas

Name | Owner

-------------+-------

cstore | omm

dbe\_perf | omm

madlib | omm

pkg\_service | omm

public | omm

snapshot | omm

### DB4AI特性应用

#### 分类算法

设置schema和兼容性参数。

SET search\_path="$user",public,madlib;

SET behavior\_compat\_options = 'bind\_procedure\_searchpath';

使用SVM对TPCH的orders表进行分类，使用默认的参数进行训练，分类的条件为o\_totalprice < 100000'。

DROP TABLE IF EXISTS order\_svm, orders\_pred,order\_svm\_summary;

SELECT madlib.svm\_classification('public.orders','public.order\_svm','o\_totalprice < 100000','ARRAY[1,o\_orderkey,o\_totalprice]');

查看训练模型，将执行结果截图。

打开列显示：

\x on

查看模型order\_svm。

select \* from order\_svm;

返回结果为：

-[ RECORD 1 ]------+---------------------------------------------

coef | {545.171945772,6.16170372745,-174.130614912}

loss | 1530442.71157333

norm\_of\_gradient | 794563661853.824

num\_iterations | 100

num\_rows\_processed | 1500000

num\_rows\_skipped | 0

dep\_var\_mapping | {f,t}

关闭列显示：

\x off

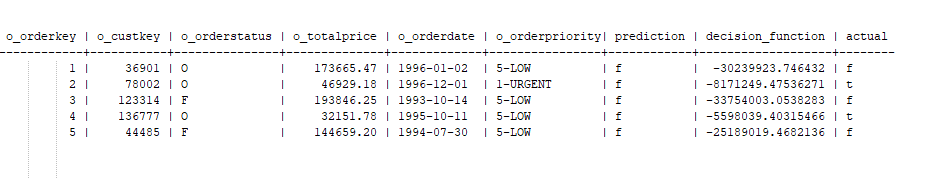
使用模型，进行预测。

SELECT madlib.svm\_predict('public.order\_svm','public.orders','o\_orderkey','public.orders\_pred');

将结果与原表进行关联查询，将执行结果截图。

SELECT \*, o\_totalprice < 100000 AS actual FROM orders JOIN orders\_pred USING (o\_orderkey) ORDER BY o\_orderkey;

结果部分内容如下：



#### 回归算法

准备训练模型，依旧复用TPCH的order表。

SET search\_path="$user",public,madlib;

SET behavior\_compat\_options = 'bind\_procedure\_searchpath';

使用linregr\_train对TPCH的orders表进回归预测。

DROP TABLE IF EXISTS orders\_linregr, orders\_linregr\_summary;

SELECT madlib.linregr\_train( 'public.orders', 'public.orders\_linregr', 'o\_totalprice', 'ARRAY[1,o\_orderkey,o\_totalprice]');

查看模型内容，将执行结果截图。

打开列显示：

\x on

查看模型orders\_linregr。

select \* from orders\_linregr;

返回结果为：

-[ RECORD 1 ]------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

coef | {-4.30019147970648e-11,1.92349564722592e-17,.999999999999999}

r2 | .999999999999997

std\_err | {9.91297854737603e-06,2.17732654879598e-12,4.25545086131727e-11}

t\_stats | {-4.33794087130829e-06,8.83420839326823e-06,23499272640.8889}

p\_values | {.999996538823953,.999992951321516,0}

condition\_no | 9112137.41175339

num\_rows\_processed | 1500000

num\_missing\_rows\_skipped | 0

variance\_covariance | {{9.82671436807373e-11,-1.42169141680179e-17,-2.73736264520016e-16},{-1.42169141680179e-17,4.74075090009183e-24,-3.50367152150851e-26},{-2.73736264520016e-16,-3.50367152150701e-26,1.81088620330859e-21}}

关闭列显示：

\x off

预测，并对比结果，将执行结果截图。

SELECT orders.o\_orderkey,orders.o\_custkey,orders.o\_orderstatus,orders.o\_totalprice,orders.o\_orderdate,orders.o\_orderpriority,

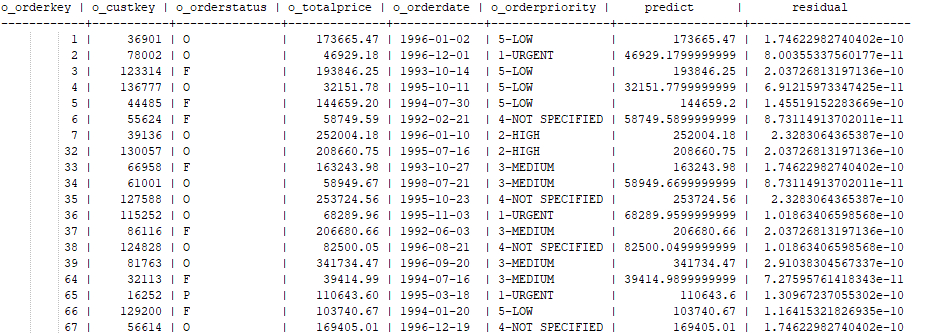
madlib.linregr\_predict( m.coef, ARRAY[1,o\_orderkey,o\_totalprice]) as predict,

o\_totalprice - madlib.linregr\_predict( m.coef, ARRAY[1,o\_orderkey,o\_totalprice]) as residual

FROM public.orders, public.orders\_linregr AS m

ORDER BY o\_orderkey;

获得预测结果：



### 关卡验证

完成分类算法和回归算法的测试，并截图。

### 思考题

分类模型与回归模型有何不同？什么是SVM算法？

答案：分类模型是指，根据训练集推断它所对应的类别，是一种定性输出，也叫离散变量预测；回归模型是指，根据训练集推断它所对应的输出值（实数）是多少，是一种定量输出，也叫连续变量预测。

在机器学习中，支持向量机（英语：support vector machine，常简称为SVM，又名支持向量网络）是在分类与回归分析中分析数据的监督式学习模型与相关的学习算法。给定一组训练实例，每个训练实例被标记为属于两个类别中的一个或另一个，SVM训练算法创建一个将新的实例分配给两个类别之一的模型，使其成为非概率二元线性分类器。SVM模型是将实例表示为空间中的点，这样映射就使得单独类别的实例被尽可能宽的明显的间隔分开。然后，将新的实例映射到同一空间，并基于它们落在间隔的哪一侧来预测所属类别。

除了进行线性分类之外，SVM还可以使用所谓的核技巧有效地进行非线性分类，将其输入隐式映射到高维特征空间中。