openGauss AI特性创新实践课



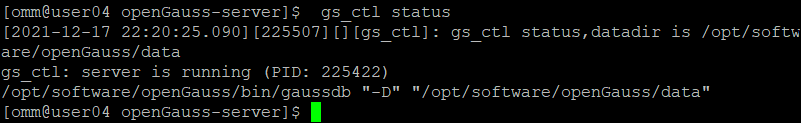
华为技术有限公司

# 关卡一、openGauss数据安装及基本操作

openGauss数据安装及基本操作, 作业提交任务如下：

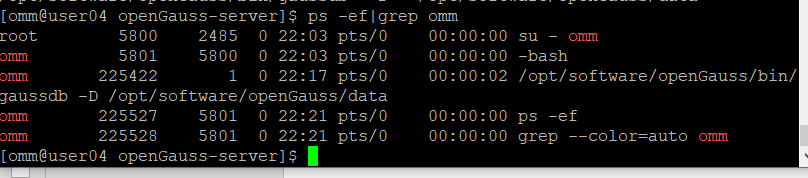
任务一：数据库状态验证

1. 查询数据库状态成功截图



任务二：数据库服务进程验证

1. 查看数据库服务进程截图（包含数据库服务器的主机名）



实验思考题：为什么需要通过源码编译，安装数据库？

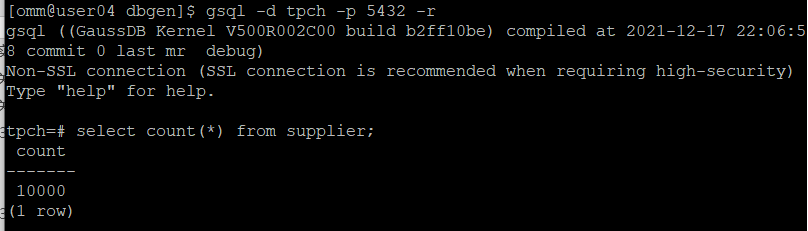
源码是源程序代码，不能直接运行，必须通过对源代码的编译把源代码转化成对应的二进制可执行程序才能运行。

# 关卡二、openGauss数据导入及基本操作

任务一：数据初始化验证

1. 查询supplier表的行数，并将结果进行图：

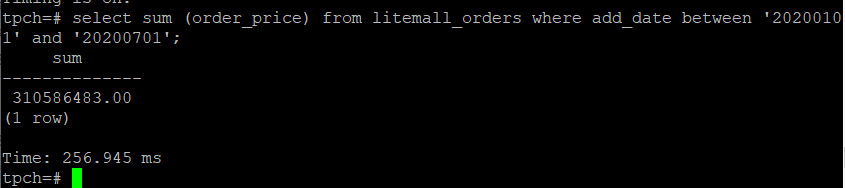
select count(\*) from supplier;;



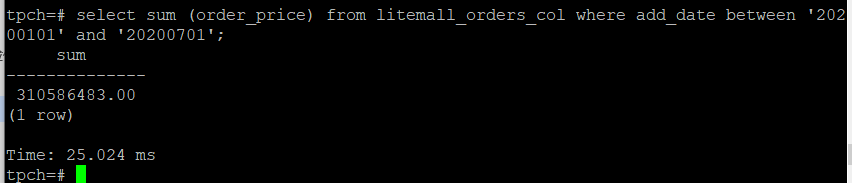
任务二：行存表与列存表执行效率对比

1. 2020年上半年litemall\_orders行存表与litemall\_orders\_col列存表中的order\_price的总和查询，并对比执行效率截图

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

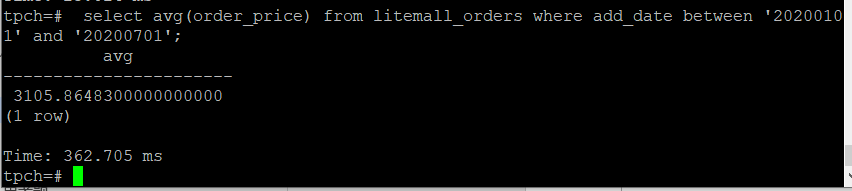


select sum (order\_price) from litemall\_orders\_col where add\_date between '20200101' and '20200701';

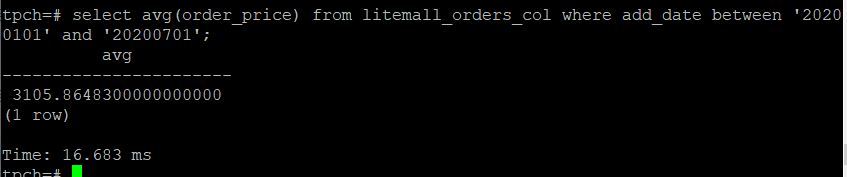


2. 2020年上半年litemall\_orders行存表与litemall\_orders\_col列存表中的order\_price的平均值查询，并对比执行效率截图

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

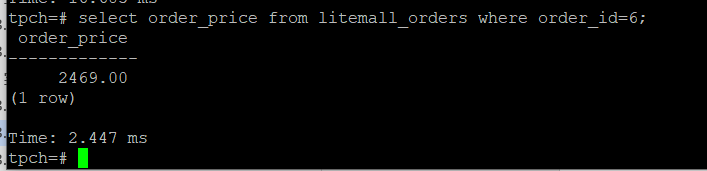


select avg (order\_price) from litemall\_orders\_col where add\_date between '20200101' and '20200701';

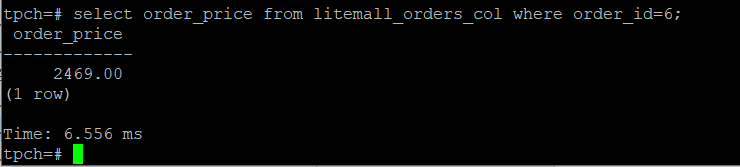


3. 查询litemall\_orders行存表与litemall\_orders\_col列存表中order\_id为6的order\_price的值，并对比执行效率截图。

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

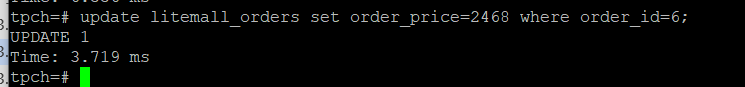


select order\_price from litemall\_orders\_col where order\_id=6;

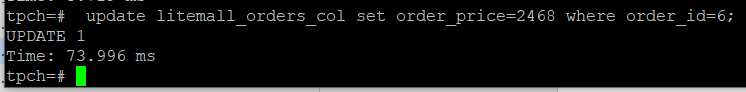


4. 将litemall\_orders行存表与litemall\_orders\_col列存表中order\_id为6的order\_price修改为2468，并对比执行效率截图。

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



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



任务三：物化视图的使用

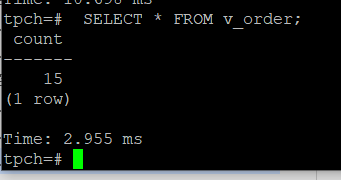
1. 创建物化视图所需要的表后，对表内容进行查询，对查询结果截图：

SELECT \* FROM test\_view;



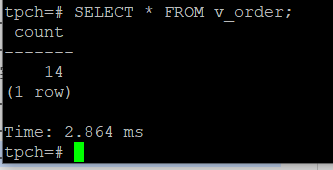
2. 使用物化视图统计人数，查询物化视图结果，将执行结果截图。

SELECT \* FROM v\_order;



3. 对表进行操作后，刷新物化视图，查询物化视图结果，将执行结果截图。

SELECT \* FROM v\_order;



4. 创建增量物化视图，查询物化视图结果，将执行结果截图。

SELECT \* FROM vi\_order;



5. 对表进行操作后，刷新增量物化视图，查询物化视图结果，将执行结果截图。

SELECT \* FROM vi\_order;



实践思考题1：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？

行存表和列存表的对数据的存储方式不同，在行存表中， 数据是按照行数据为基础逻辑存储单元进行存储的， 一行中的数据在存储介质中以连续存储形式存在。在列存表中， 数据是按照列为基础的逻辑存储单元进行存储的，一列中的数据在存储介质中以连续存储形式存在。由于储存数据的方式不同，执行SQL语句执行相关操作所需要的时间也不一样。

由于行存表是一次性读入的，如果大部分时间都是关注整张表的内容，而不是单独某几列，并且所关注的内容是不需要通过任何聚集运算的，行存表效率更高。如果比较关注的都是某几列的内容，或者有频繁聚集需要的，通过聚集之后进行数据分析的表，这时行存表会读取所有列的数据，数据读取过程中会产生冗余数据，所以列存表效率更高。

例如在任务一1）中对2020年上半年litemall\_orders行存表与litemall\_orders\_col列存表中的order\_price的总和查询：其中行存表用时256s,而列存表则用时25秒，这是因为本次操作只取了一列的数据，行存表读取所有列造成时间用的较多。

在任务一4）中将litemall\_orders行存表与litemall\_orders\_col列存表中order\_id为6的order\_price修改为2468，并对比执行效率截图：其中行存表用时3.7,而列存表则用时73.9，这就是因为列存表数据按列存储，同一行数据的地址之间的距离较远，读取时间要更长。

实践思考题2：全量物化视图与增量物化视图有哪些差别？

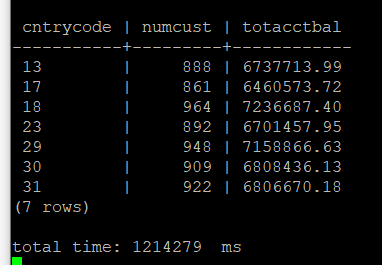
全量物化视图在原表被修改时需要更高的时间进行全量物化视图的生成，而增量物化视图需要对原本的数据进行更新，但这也意味着增量物化视图比全量物化视图占用的空间更大，在数据量大的表中全量物化视图的效率远不如增量物化视图。

# 关卡三、openGauss的AI4DB特性应用

任务一：使用X-Tuner进行参数优化

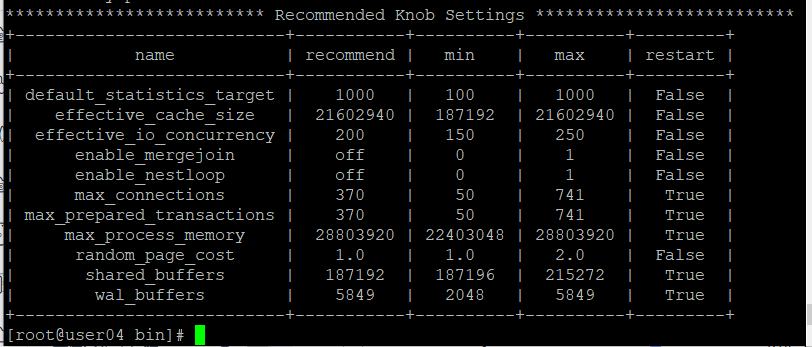
1. 执行TPCH脚本，获得测试时间，将执行结果截图：

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



2. 使用root用户，执行X-Tuner进行参数建议优化，将执行结果截图

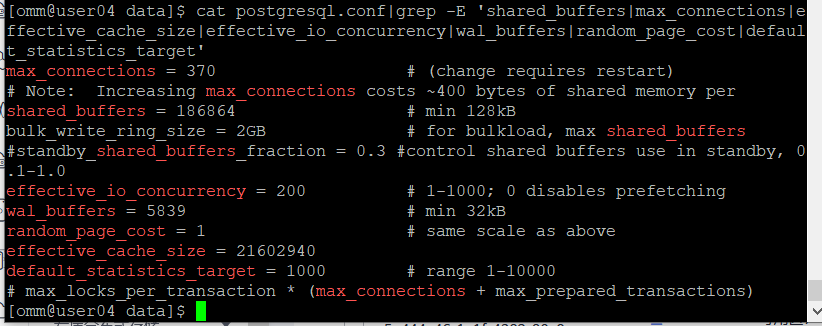
gs\_xtuner recommend --db-name tpch --db-user omm --port 5432 --host 127.0.0.1 --host-user omm



3.重启完成后，获取参数值：

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'



任务二：使用Index-advisor对select 查询语句进行优化，并通过对比执行计划，得到优化前后的不同。

1. 使用explain，对查询2020年3月订单表收入并进行排序的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

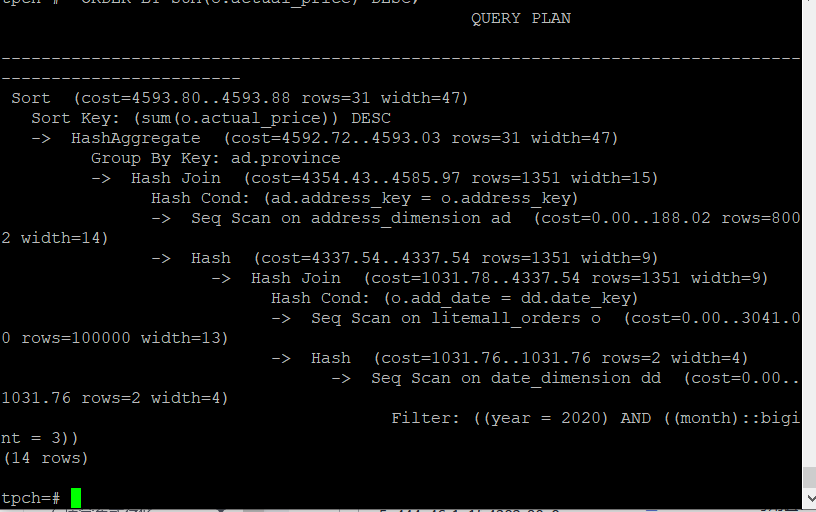
AND o.add\_date = dd.date\_key

AND dd.year = 2020

AND dd.month = 3

GROUP BY ad.province

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



2. 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

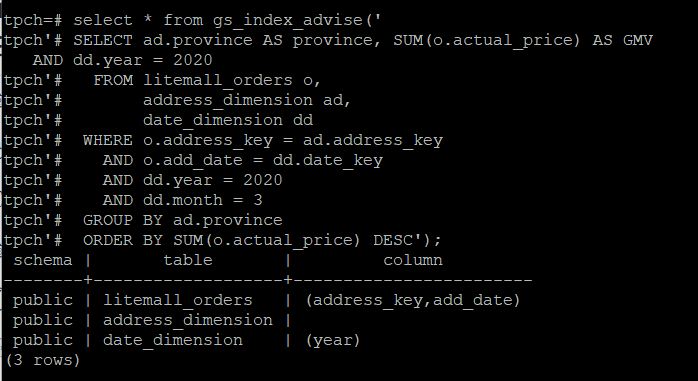
AND o.add\_date = dd.date\_key

AND dd.year = 2020

AND dd.month = 3

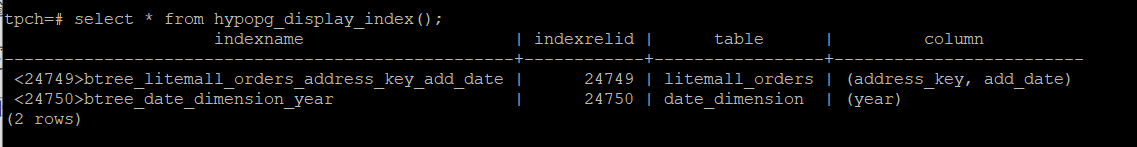
GROUP BY ad.province

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



3. 查看创建的虚拟索引列，将执行结果截图。

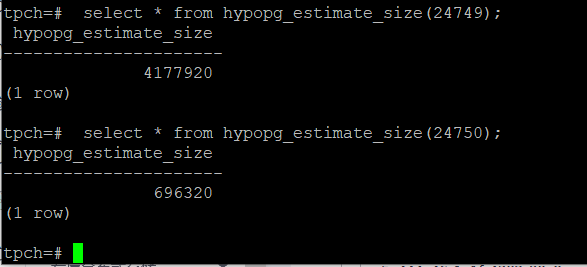
select \* from hypopg\_display\_index();



4. 获取索引虚拟列大小结果（单位为：字节），将执行结果截图。

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

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



5.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

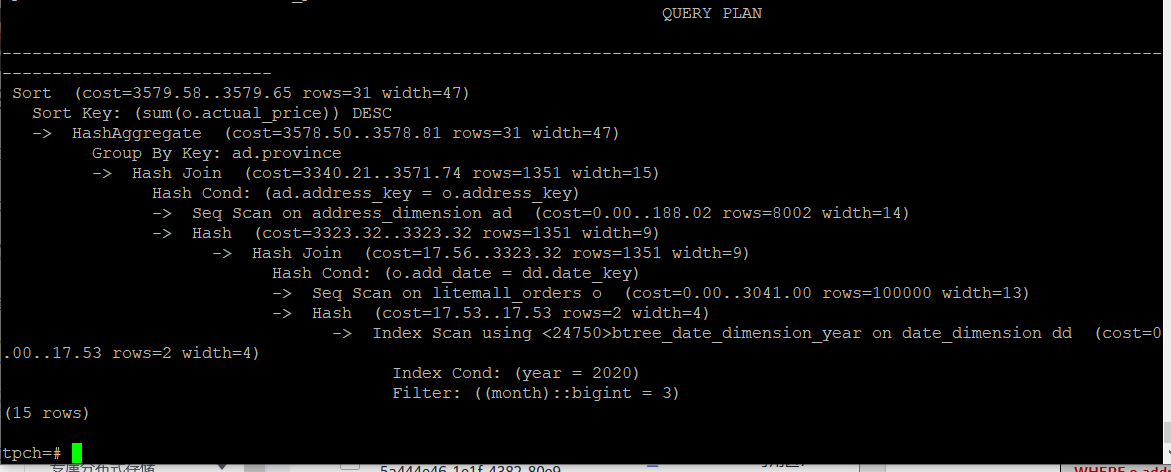
AND o.add\_date = dd.date\_key

AND dd.year = 2020

AND dd.month = 3

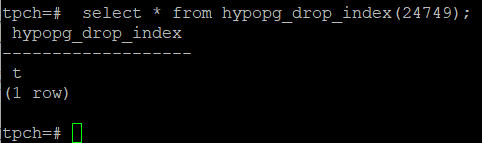
GROUP BY ad.province

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



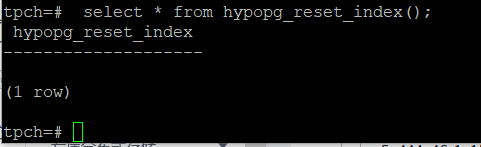
6. 删除某一个索引虚拟列，将执行结果截图。

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



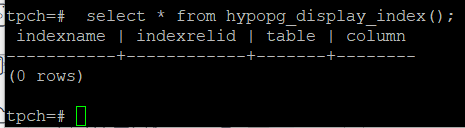
7. 删除某一个索引虚拟列，将执行结果截图。

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



8. 查看索引虚拟列，将执行结果截图。

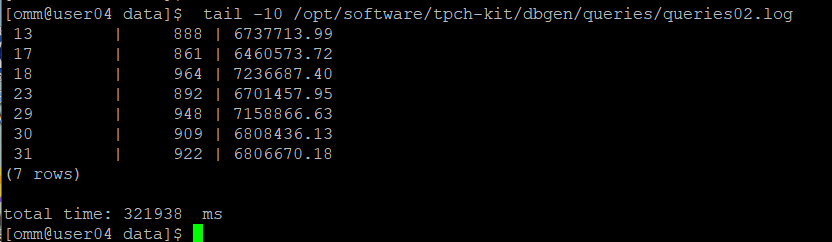
select \* from hypopg\_display\_index();



任务三：通过创建索引，对queries.sql中的SQL语句进行优化，并对比优化前后queries.sql执行的时间。

1. 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



挑战一：进一步优化queries.sql中的查询语句，使得前后执行时间出现倍数级的提升。

1. 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/queries03.log

实践思考题1：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？

X-Tuner是一款数据库集成的参数调优工具，通过结合深度强化学习和全局搜索算法等AI技术，实现在无需人工干预的情况下，获取最佳数据库参数配置。优化了default\_statistics\_target, effective-cache-size, effective\_io\_concurrency, enable\_mergejoin, enable\_nestloop, max\_connections, max\_prepared\_transactions, max\_process\_memory, random\_page\_cost, shared\_buffers, wal\_buffers这些参数。提高了数据库的效率。

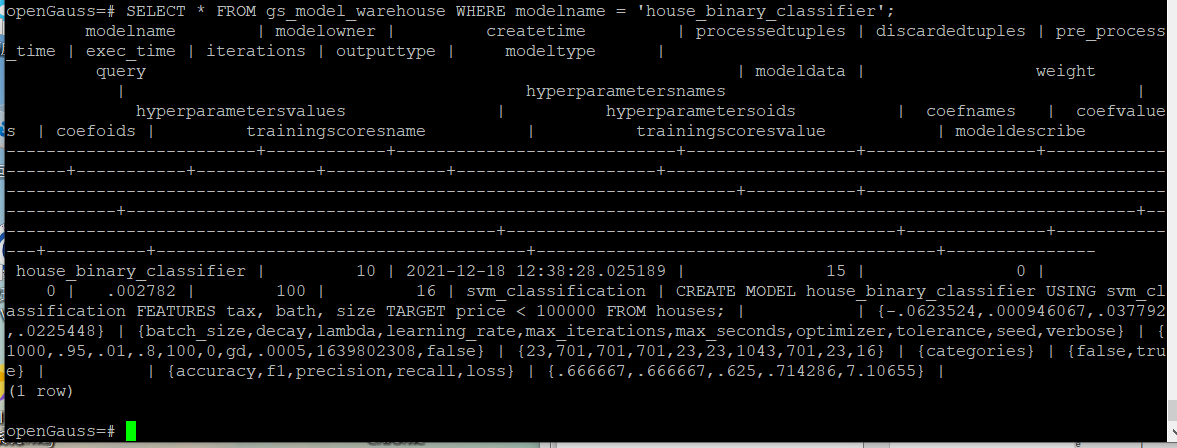
实践思考题2：索引的使用，对于执行SQL有什么好处？除了使用索引和参数外，还有哪些方面可以对数据库进行优化？

索引的使用可以大幅提高数据库查找效率，除了索引和参数外，还可以通过优化数据库的数据存储结构，根据数据库的用途不同选择行存储或者列存储的方式。可以给数据库添加缓存表，一些使用较多的数据库保存在缓存表中，提高查询效率等。

# 关卡四、openGauss的DB4AI特性应用

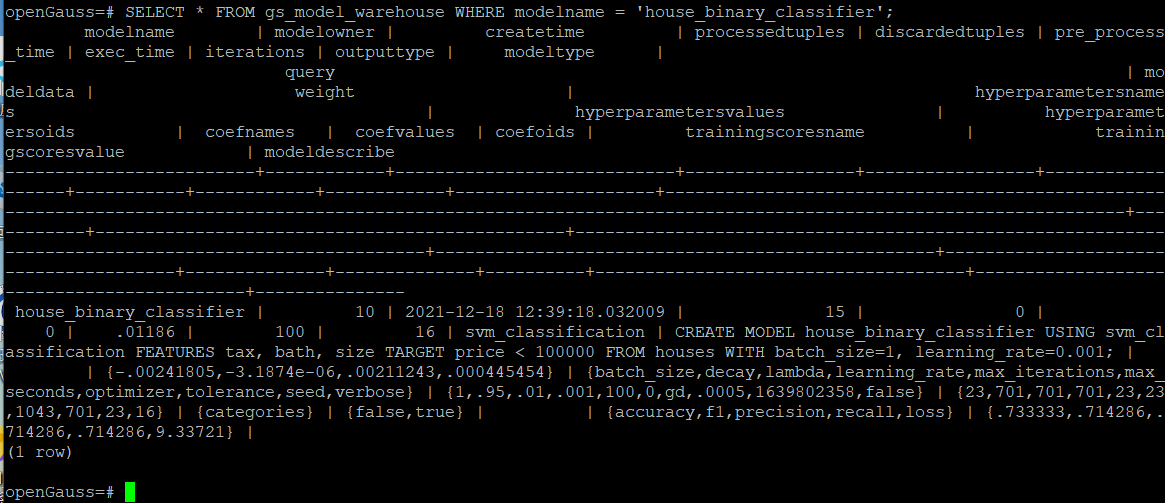
任务一：在gs\_model\_warehouse系统表中查看训练后的模型信息，将执行结果截图：

postgres=# SELECT \* FROM gs\_model\_warehouse WHERE modelname = 'house\_binary\_classifier';



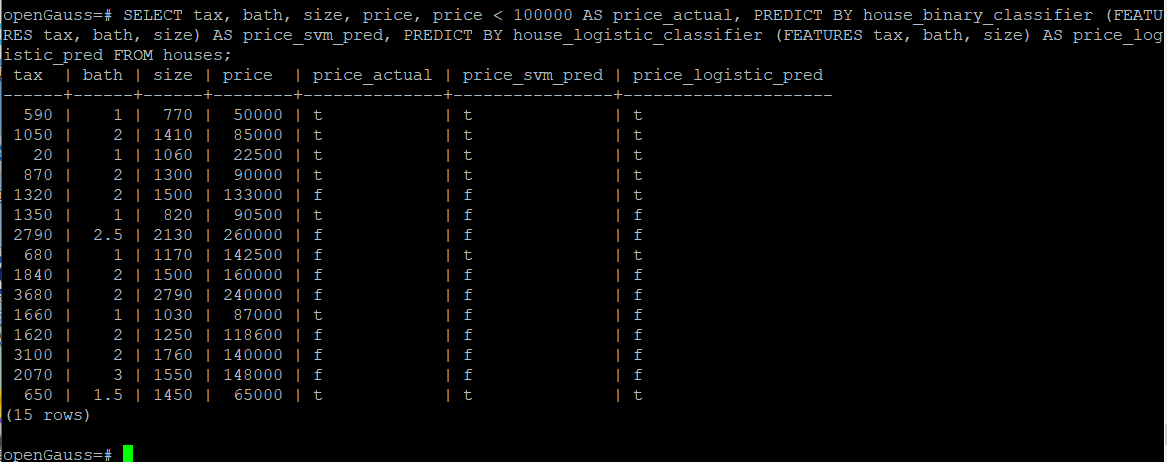
任务二：观察新模型的信息，将执行结果截图。

postgres=# SELECT \* FROM gs\_model\_warehouse WHERE modelname = 'house\_binary\_classifier';



任务三：利用训练好的逻辑回归模型预测数据，并与SVM算法进行比较，将执行结果截图。

postgres=# SELECT tax, bath, size, price, price < 100000 AS price\_actual, PREDICT BY house\_binary\_classifier (FEATURES tax, bath, size) AS price\_svm\_pred, PREDICT BY house\_logistic\_classifier (FEATURES tax, bath, size) AS price\_logistic\_pred FROM houses;



实践思考题1：分类模型与回归模型有何不同？

分类模型可以将回归模型的输出离散化，回归模型也可以将分类模型输出连续化。分类模型用于处理分类问题，回归模型用于处理回归问题。

实践思考题2：什么是SVM算法？

VM是一个二元分类算法，支持线性分类和非线性分类。经过演进，现在也可以支持多元分类，同时经过扩展，也能应用于回归问题。SVM学习的基本想法是求解能够正确划分训练数据集并且几何间隔最大的分离超平面。

实践思考题3：分类问题有哪些评价指标，请分别说明他们的含义？

**精确率和召回率**

精确率是计算我们预测某类样本中，有多少被正确预测的，是针对样本而言的

精确率 P=TP真阳性/TP真阳性+FP真阳性

召回率是针对原来实际样本而言的，看有多少样本被正确预测出来 召回率 R=TP真阳性/TP真阳性+FN假阴性

**F1分数（F1 Score）**，是统计学中用来衡量二分类模型精确度的一种指标。它同时兼顾了分类模型的精确率和召回率。F1分数可以看作是模型精确率和召回率的一种加权平均，它的最大值是1，最小值是0。

**Micro-F1和Macro-F1**

在第一个多标签分类任务中，可以对每个“类”，计算F1，显然我们需要把所有类的F1合并起来考虑。

**这里有两种合并方式：**

* **第一种**计算出所有类别总的Precision和Recall，然后计算F1。

例如依照最上面的表格来计算:Precison=5/(5+4)=0.556,Recall=5/(5+4)=0.556，然后带入F1的公式求出F1，这种方式被称为Micro-F1微平均。

* **第二种**方式是计算出每一个类的Precison和Recall后计算F1，最后将F均。

例如上式A类：P=2/(2+0)=1.0，R=2/(2+2)=0.5，F1=(2\*1\*0.5)/1+0.5=0.667。同理求出B类C类的F1，最后求平均值，这种范式叫做Macro-F1宏平均。

实践思考题4：回归问题有哪些评价指标，请分别说明他们的含义？

点对点误差

MSE均方误差

均方误差描述了样本真实值与预测值差方求和的平均值，在机器学习中，利用均方误差最小化优化模型的方法称为“最小二乘法”

RMSE

RMSE即对MSE开平方根

MAE平均绝对误差

MAE（Mean Absolute Erroe）即真实值与预测值差的绝对值和求平均：