openGauss AI特性创新实践课



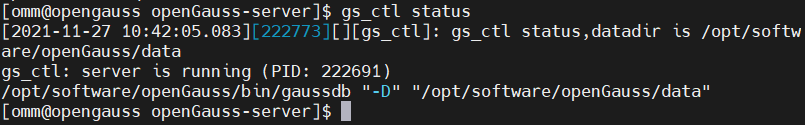
华为技术有限公司

# 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：

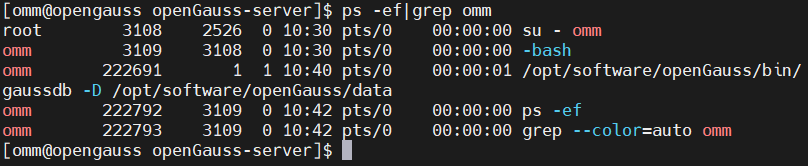
任务一：数据库状态验证

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）



任务三：简单描述关卡一中，安装数据库所需要的步骤。

1.编译前准备

(1) 创建omm用户组，给与用户组读取及执行的权限（755）；

(2) 下载openGauss源码和第三方编译库；

(3) 更新python3并安装相关依赖包。

2.数据库安装编译

(1) 设置环境变量：openGauss及其数据、第三方编译库和其他依赖（cmake、gcc、g++等）的环境变量，并source使其生效；

(2) 编译安装：生成配置文件，make编译再安装；

(3) 数据库初始化设置：

使用命令gs\_initdb -D $PGDATA --nodename=hostname --locale="en\_US.UTF-8" -Atrust -w {password}设置密码。

实验思考题：为什么需要通过源码编译，安装数据库？

1.满足不同的运行平台，Linux发型版本众多，但是每个版本采用的软件或者内核版本都不一样，因此二进制包所依赖的环境不一定能够正常运行，所以大部分软件直接提供源码。

2.方便定制，满足不同的需求，很多时候所需要的软件都是可以定制的，需要什么就安装什么，允许用户选择软件编译配置，而大多数二进制代码都是一键装全，所以自由度并不高。

3.精简，二进制文件为了保证兼容性，包含了一些不会用到的包。使用源码编译能够精简系统，减少链接库的数量。

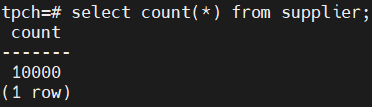
4.方便运维、开发人员维护，我们的源码是可以打包二进制的，但是对于这个软件的打包都会有一份代价不小的额外工作，包括维护，所以如果是源码的话，软件产商会直接维护，但是如果是二进制的话，一般都是Linux发行商提供。

# 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：

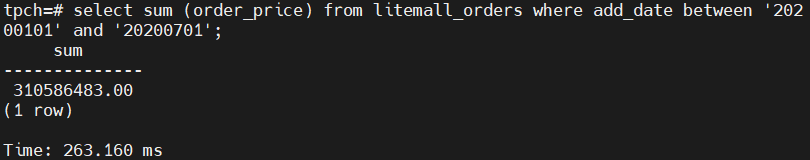
select count(\*) from supplier;;



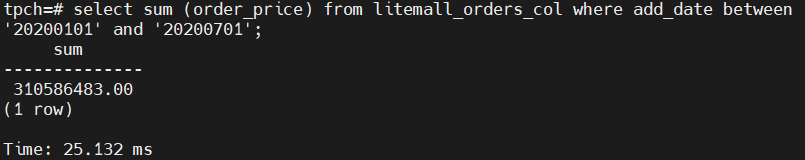
任务二：行存表与列存表执行效率对比

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';

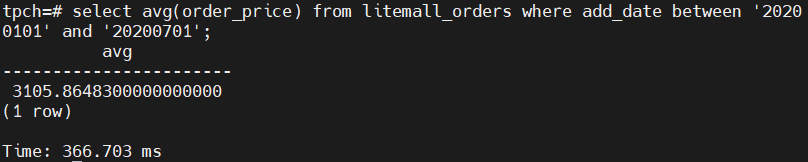


select sum (order\_price) from litemall\_orders\_col where add\_date between '20200101' and '20200701';

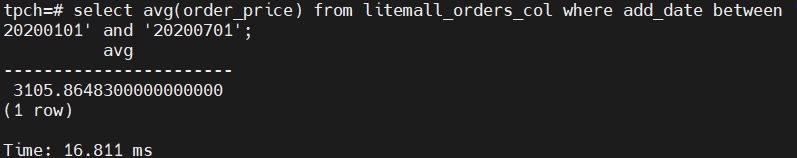


2. 2020年上半年litemall\_orders行存表与litemall\_orders\_col列存表中的order\_price的平均值查询，并对比执行效率截图

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

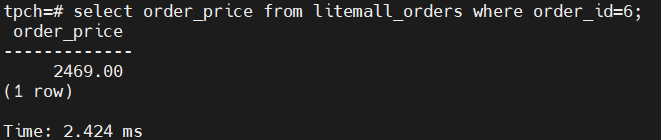


select avg (order\_price) from litemall\_orders\_col where add\_date between '20200101' and '20200701';

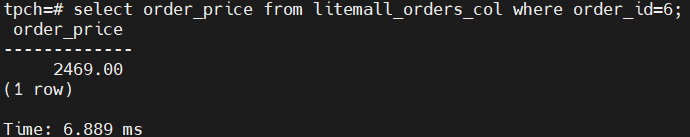


3. 查询litemall\_orders行存表与litemall\_orders\_col列存表中order\_id为6的order\_price的值，并对比执行效率截图。

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;



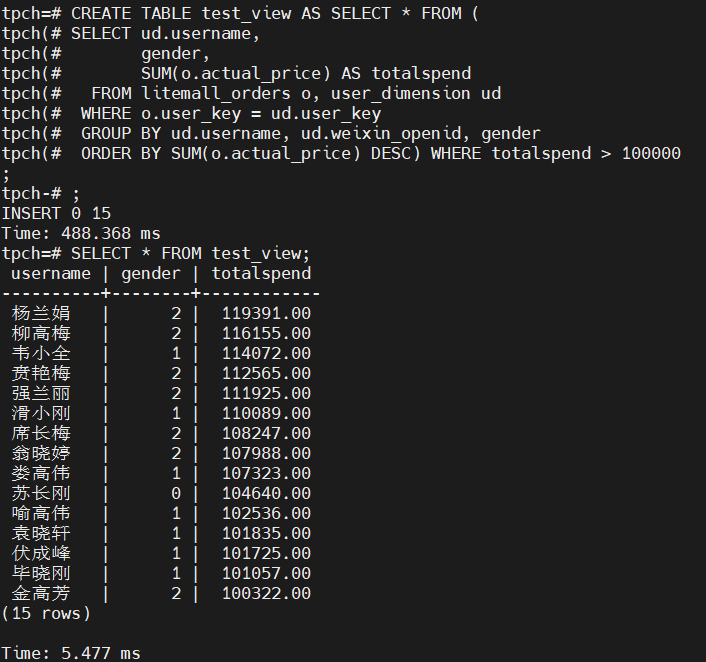
update litemall\_orders\_col set order\_price=2468 where order\_id=6;



任务三：物化视图的使用

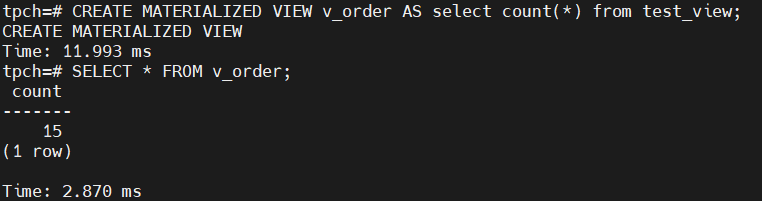
1. 创建物化视图所需要的表后，对表内容进行查询，对查询结果截图：

SELECT \* FROM test\_view;



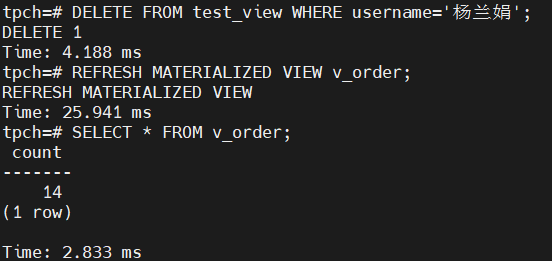
2. 使用物化视图统计人数，查询物化视图结果，将执行结果截图。

SELECT \* FROM v\_order;



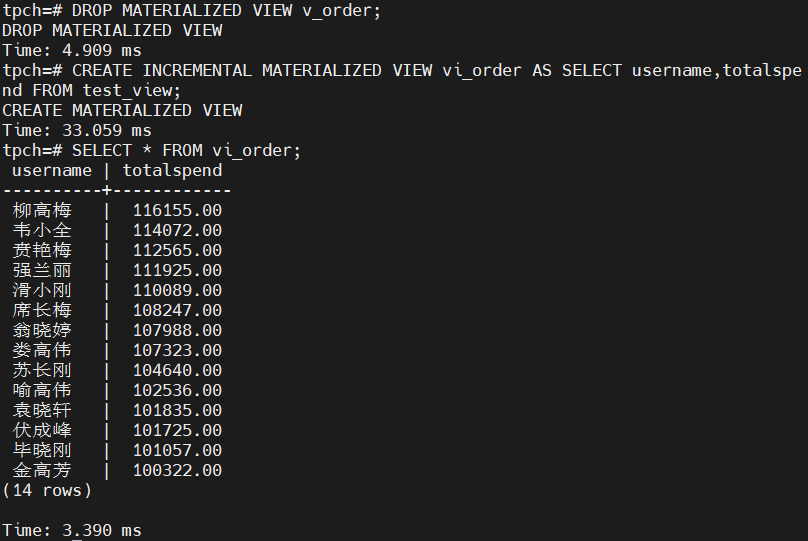
3. 对表进行操作后，刷新物化视图，查询物化视图结果，将执行结果截图。

SELECT \* FROM v\_order;



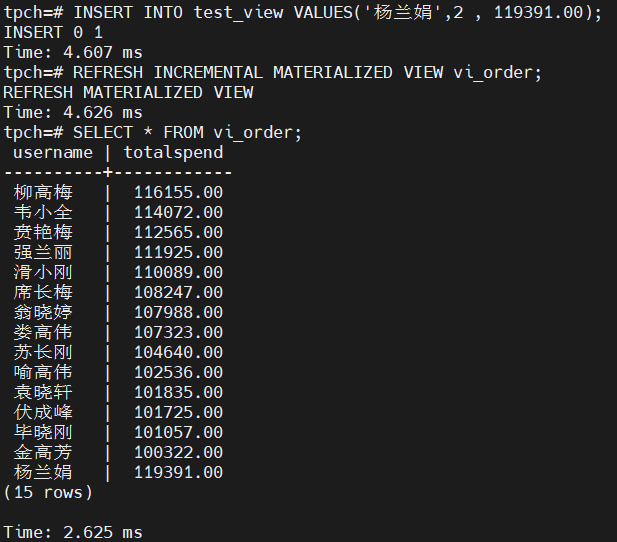
4. 创建增量物化视图，查询物化视图结果，将执行结果截图。

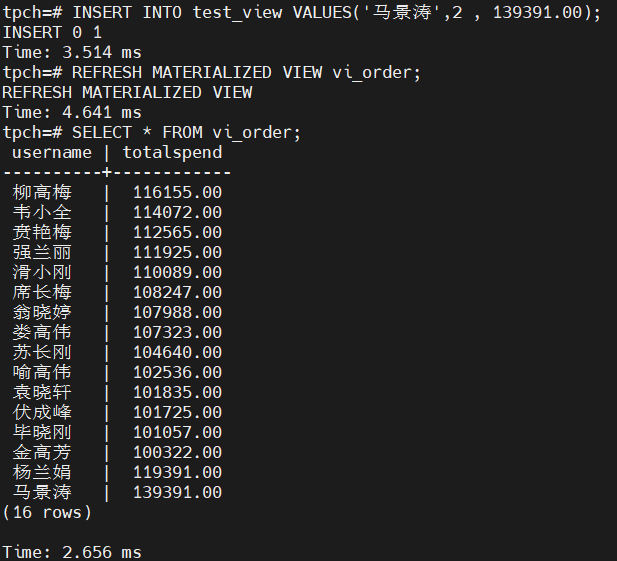
SELECT \* FROM vi\_order;



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？

测试结果（ms）：

|  |  |  |
| --- | --- | --- |
| 操作 | 行存表 | 列存表 |
| sum | 263.160 | 25.132 |
| avg | 366.703 | 16.811 |
| 选择一个属性的分量 | 2.424 | 6.889 |
| update | 4.181 | 77.362 |

因此在聚集函数操作上，列存表具有显著优势，而在更新操作上，行存表具有优势。

行存表：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。

列存表：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。

行存储的写入是一次完成，而列存储由于需要把一行记录拆分成单列保存，写入次数明显比行存储多，再加上磁头需要在盘片上移动和定位花费的时间，实际时间消耗会更大。对于数据修改，行存储是在指定位置写入一次，列存储是将磁盘定位到多个列上分别写入。所以，数据修改行存储占优。读取数据时，如果只需要其中几列数据的情况，行存储就会存在冗余列，列存储则不存在冗余，消除冗余列也会消耗资源。但是列存储的每一列数据类型是同质的，行存储则在一行记录中保存了多种类型的数据，数据解析需要在多种数据类型之间频繁转换，增加了解析的时间。所以，列存储的解析过程更有利于分析大数据。

总的说来，由于存储的物理结构的不同，导致数据的扫描方式不同，因此引起执行效率的区别。

由于上述的区别，也就说明了两种存储方式在何种情况下执行效率高。

行存表在以下种类的操作中效率更高：

1.随机的增删改查操作。

2.需要在行中选取所有属性的查询操作。

3.需要频繁插入或更新的操作，其操作与索引和行的大小更为相关。

列存表在以下种类的操作中效率更高：

1.往往查询某几列的内容。

2.频繁进行聚集操作，需要用的聚集函数的查询。

实践思考题2：全量物化视图与增量物化视图有哪些差别？

全量物化视图仅支持对创建好的物化视图做全量更新，而不支持做增量更新。

增量物化视图支持全量更新，也支持增量更新物化视图。

物化视图实际上就是一种特殊的物理表，物化视图是相对普通视图而言的。普通视图是虚拟表，应用的局限性较大，任何对视图的查询实际上都是转换为对SQL语句的查询，性能并没有实际上提高。而物化视图实际上就是存储SQL所执行语句的结果，起到缓存的效果。

全量更新为将数据库的数据重新导入，然后更新，相当于重新创建了一个表。

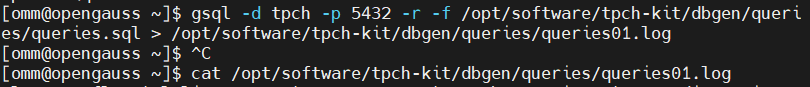
增量更新做的操作是将当前视图表中的数据和query中的数据做一个join操作，然后才将差量做填充。

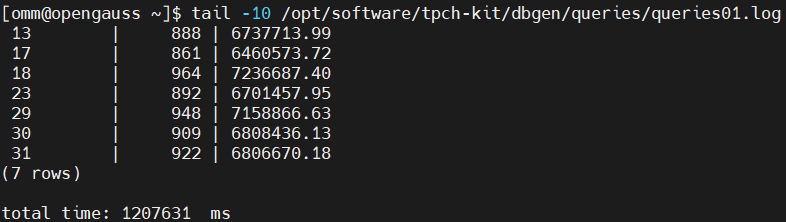
# 关卡三、openGauss的AI4DB特性应用

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：

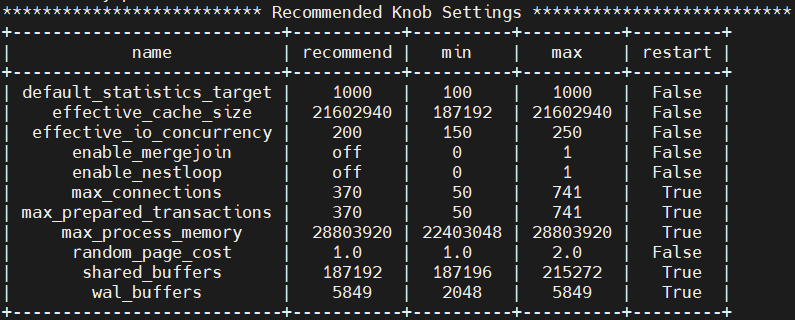
gsql -d tpch -p 5432 -r -f /opt/software/tpch-kit/dbgen/queries/queries.sql > /opt/software/tpch-kit/dbgen/queries/queries01.log

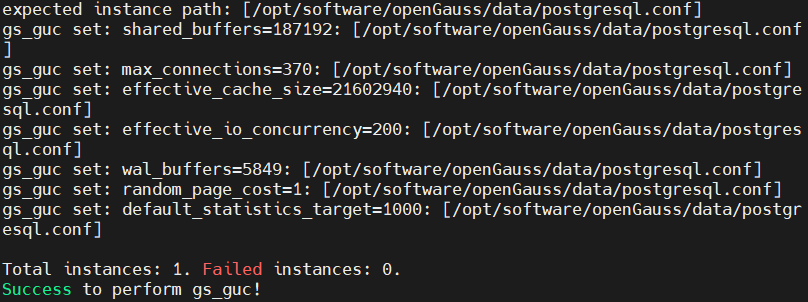




2. 使用root用户，执行X-Tuner进行参数建议优化，将执行结果截图

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

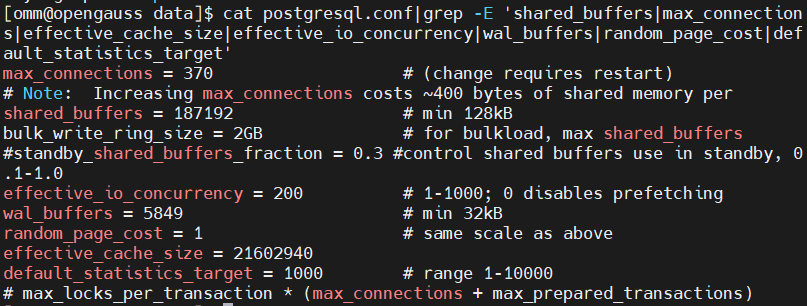




3.重启完成后，获取参数值：

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

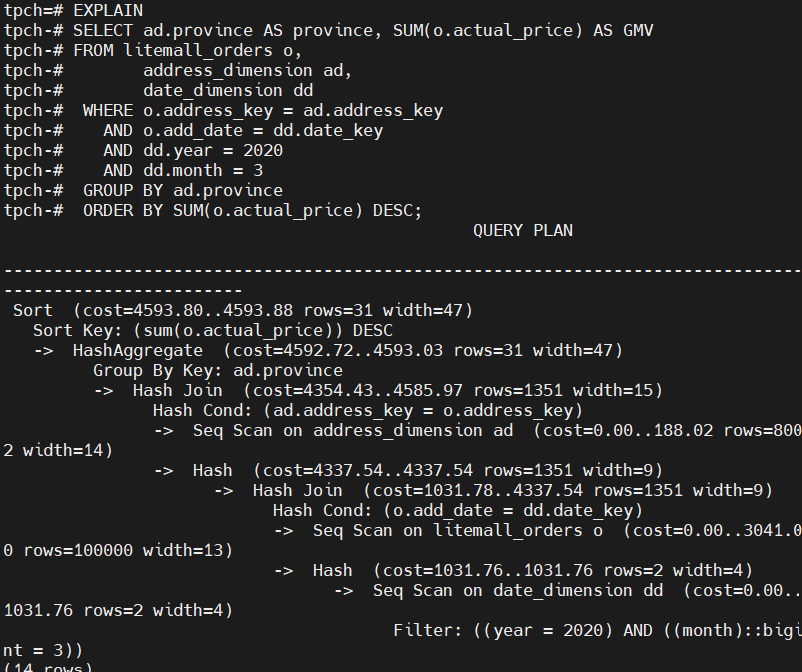
AND o.add\_date = dd.date\_key

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

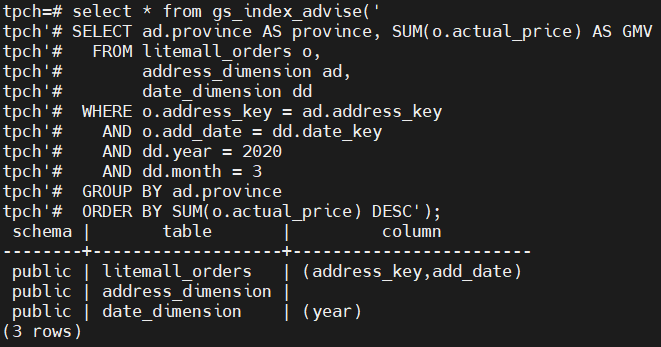
AND o.add\_date = dd.date\_key

AND dd.year = 2020

AND dd.month = 3

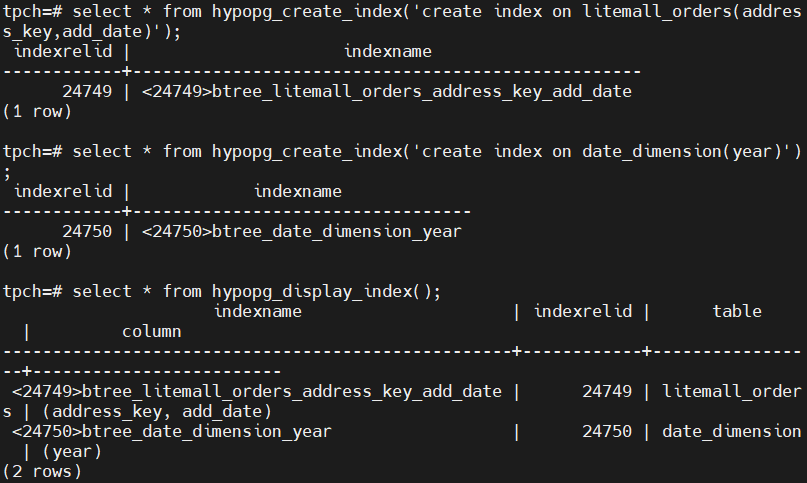
GROUP BY ad.province

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。

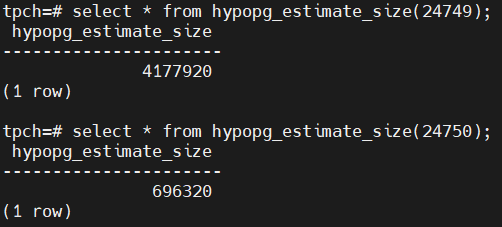
select \* from hypopg\_display\_index();



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

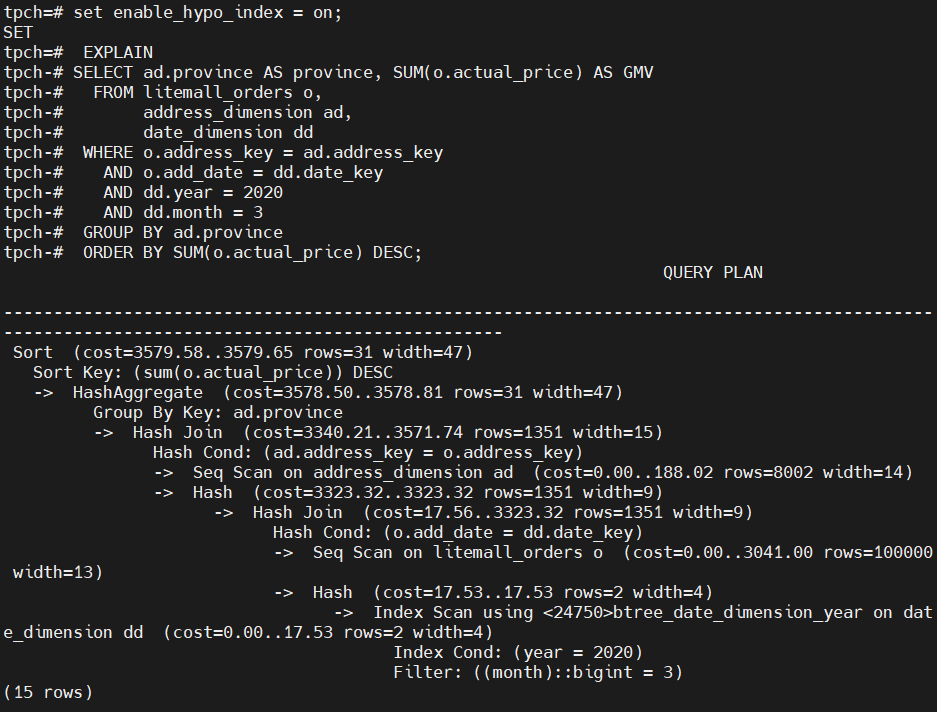
AND o.add\_date = dd.date\_key

AND dd.year = 2020

AND dd.month = 3

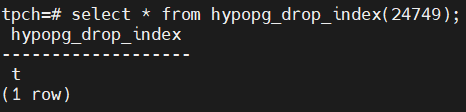
GROUP BY ad.province

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



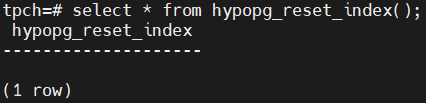
6. 删除某一个索引虚拟列，将执行结果截图。

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



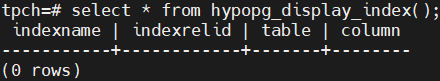
7. 删除所有索引虚拟列，将执行结果截图。

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



8. 查看索引虚拟列，将执行结果截图。

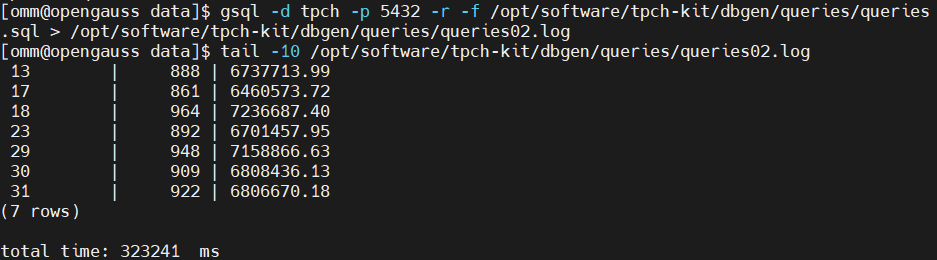
select \* from hypopg\_display\_index();



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

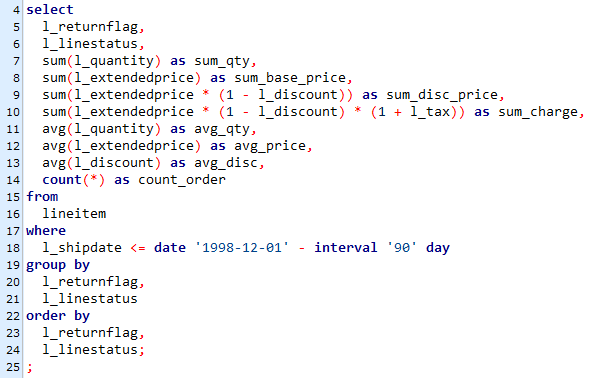


挑战一：进一步优化queries.sql中的查询语句，使得前后执行时间出现倍数级的提升。

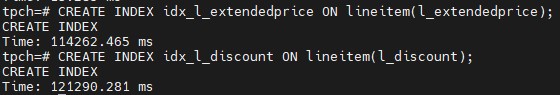
1. 重新执行queries.sql查询，将执行结果截图：

gsql -d tpch -p 5432 -r -f /opt/software/tpch-kit/dbgen/queries/queries.sql > /opt/software/tpch-kit/dbgen/queries/queries03.log

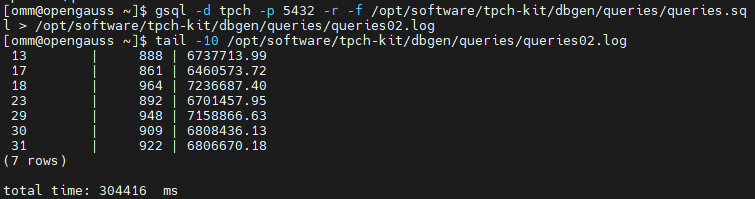
打开路径为/opt/software/tpch-kit/dbgen/queries/queries.sql的sql文件，第一条sql语句如下图所示，该语句执行时间较其他的语句长：



发现涉及到了l\_extendedprice和l\_discount两列的聚集函数，因此可以建两个索引，如下所示：



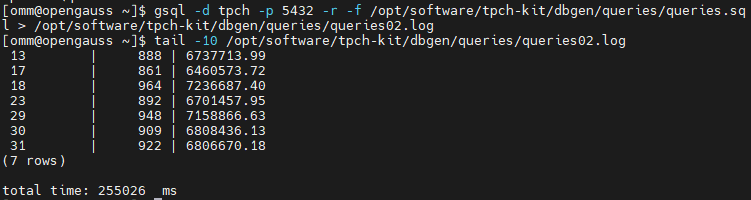
此时查询时间如下所示，时间降低约30s：



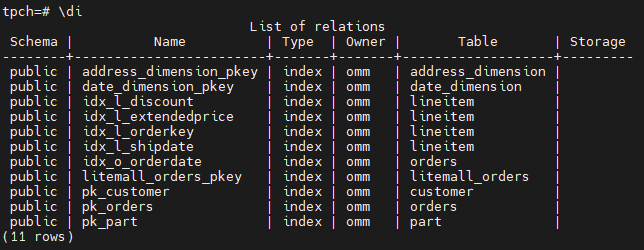
此外，通过分析发现，lineitem由于表比较大，选择操作会消耗大量的时间，因此在该表的l\_orderkey上建立索引，如下所示：



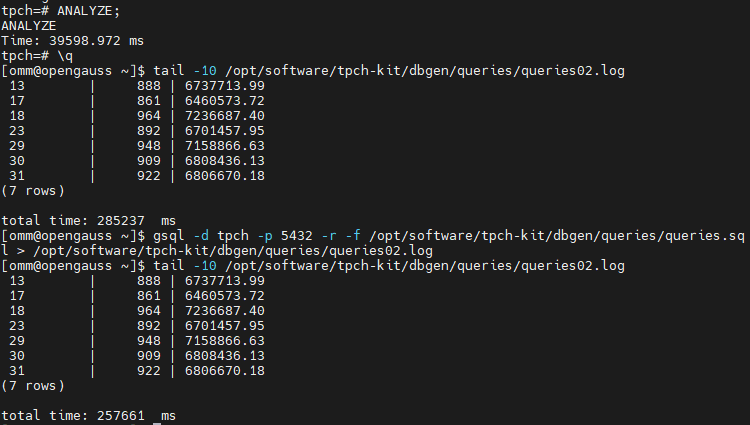
然后执行时间降低至250s左右，如下图所示：



至此数据库的索引存在五条索引及三条主键索引，如下所示：



当使用ANALYZE语句更新全库的统计信息后，再次运行，性能无明显提升。



最后，我们尝试使用X-Tuner自调优策略。首先是使用迭代推荐命令。

存在问题：python执行的xtuner并不位于路径/opt/software/openGauss-server/src/gausskernel/dbmind/tools/xtuner/下，而是应该前往python的xtuner包的路径下，即/usr/local/lib/python3.7/site-packages/openGauss\_xtuner-2.1.0-py3.7.egg/tuner/，将该路径下的xtuner.conf文件更改以下两行：

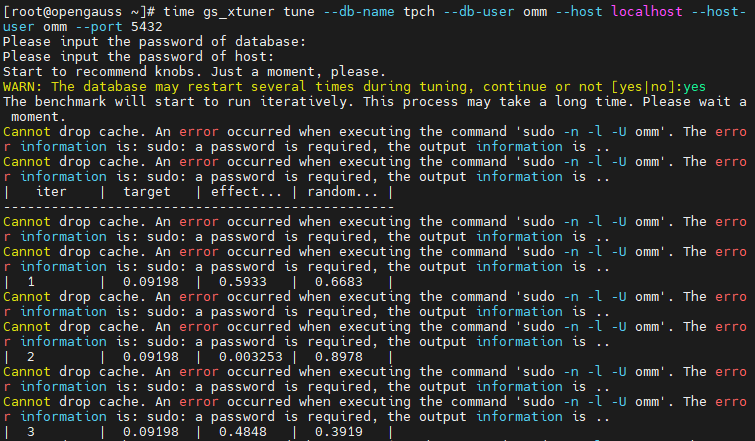
max\_iterations = 10 （从100轮改为10）

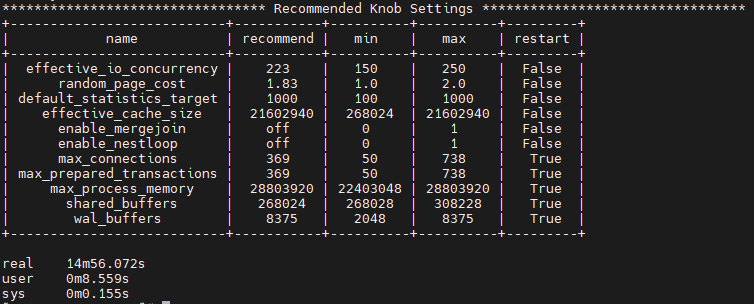
benchmark\_path = /opt/software/tpch-kit/dbgen/queries/

由于每轮迭代耗时约2分钟，为了避免耗时过长，我们可以将默认的100轮改为10论，同时引入自己的数据集。使用以下命令开始迭代。

time gs\_xtuner tune --db-name tpch --db-user omm --host localhost --host-user omm --port 5432

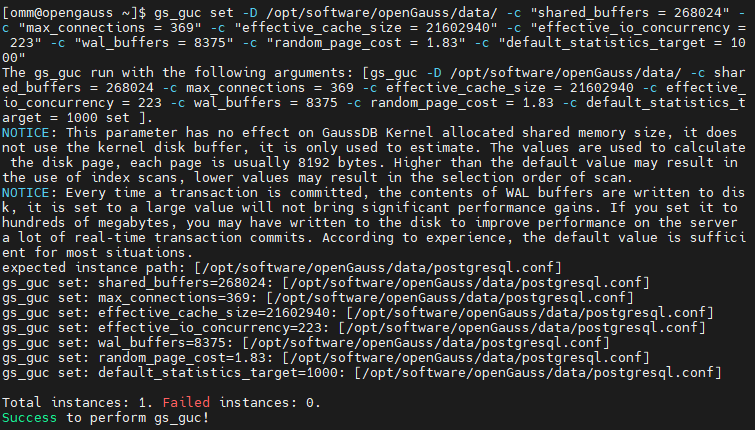
但是，迭代的过程中，会出现报错，目前仍未找到原因，当执行结束后，推荐的参数为：



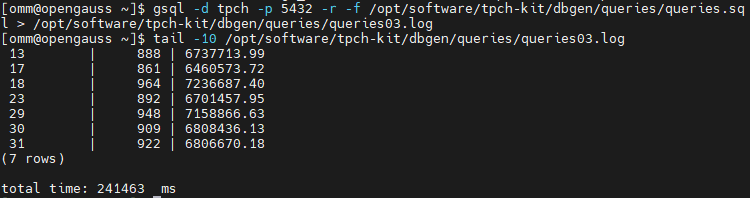


因此使用命令：

**gs\_guc set -D /opt/software/openGauss/data/ -c "shared\_buffers = 268024" -c "max\_connections = 369" -c "effective\_cache\_size = 21602940" -c "effective\_io\_concurrency = 223" -c "wal\_buffers = 8375" -c "random\_page\_cost = 1.83" -c "default\_statistics\_target = 1000"**

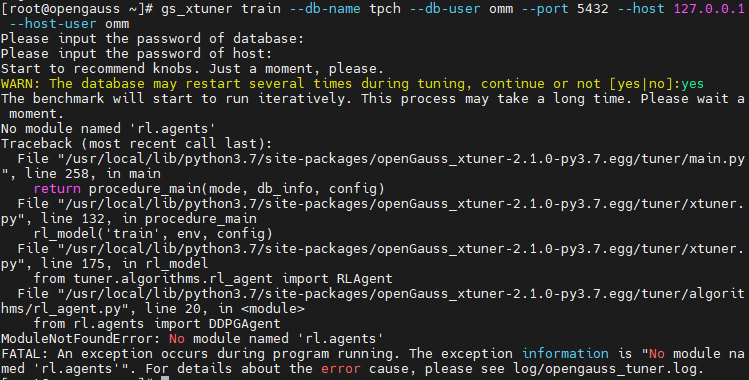
****

重新运行得到如下结果：



查询时间降低至240s左右，略有降低

此外，还可以使用xtuner的训练模式，但是安装好rl模块后，在运行时，仍会报模块未找到的错误，目前仍未找到原因：



总结：数据库调优经过了建立索引，然后再使用xtuner自调优的过程，最终将查询时间从330s降低到240s左右。

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？

|  |  |  |  |
| --- | --- | --- | --- |
| 参数 | 描述 | 默认值 | 优化值 |
| **shared\_buffers** | 设置数据库服务器将使用的共享内存缓冲区量 | 1024(8MB) | 187192(1462.4375MB) |
| **max\_connections** | 决定数据库的最大并发连接数 | 100 | 370 |
| **effective\_cache\_size** | 设置规划器对一个单一查询可用的有效磁盘缓冲区尺寸的假设 | 524288 (4GB) | 21602940 |
| **effective\_io\_concurrency** | 设置PostgreSQL可以同时被执行的并发磁盘I/O操作的数量 | 1 | 200 |
| **wal\_buffers** | 用于还未写入磁盘的WAL数据的共享内存量 | -1 | 5849 |
| **random\_page\_cost** | 设置规划器对一次非顺序获取磁盘页面的代价估计 | 4 | 1 |
| **default\_statistics\_target** | 为没有通过ALTER TABLE SET STATISTICS设置列相关目标的表列设置默认统计目标 | 100 | 1000 |

**shared\_buffers：**在数据库系统中，我们主要关注磁盘IO，大多数OLTP工作负载都是随机IO，因此从磁盘获取非常慢。因此可以将数据缓存在RAM中，以此来提高性能。但是shared\_buffers设置较大的值会导致很多等待清除的脏数据的问题，设置太小则无法重新对write进行排序从而优化IO。

**max\_connections：**增加并行连接数量，如果连接数越多，介于MySQL会为每个连接提供连接缓冲区，就会开销越多的内存，降低数据库效率。

**effective\_cache\_size：**这个参数会被考虑在使用一个索引的代价估计中，更高的数值会使得索引扫描更可能被使用，更低的数值会使得顺序扫描更可能被使用。

**effective\_io\_concurrency：**该参数对大量数据页访问时效果显著。如果数据库经常忙于在并发会话中发出的多个查询，较低的值可能足以使磁盘阵列繁忙，而比保持磁盘繁忙所需的值更高的值只会造成额外的CPU开销。

**wal\_buffers：**为预写日志(wal)缓冲区，在每次事务提交时，WAL缓冲区的内容才被写出到磁盘，提高该值能提高性能，但因此极大的值不会提供显著的收益。

**random\_page\_cost：**设置乱序读取要比顺序读取慢多少（倍），减少这个值将导致系统更倾向于索引扫描；提高它将让索引扫描看起来相对更昂贵。

**default\_statistics\_target：**更大的值增加了需要做ANALYZE的时间，扫描的数据行的个数也就更多，得到的优化器统计数据就更准确。

总的说来，对这些参数的优化主要是为了改善并发性、合理分配内存资源，以及提高IO的效率等。

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？

查询是数据库中执行得最多的操作，而索引的引入就是为了提升数据库的检索速度。除此之外，索引还有其他好处：

1.创建唯一性索引，保证数据库表中每一行数据的唯一性;

2.加速表和表之间的连接;

3.在使用分组和排序子句进行数据检索时，可以显著减少查询中分组和排序的时间。

但是索引也有一些问题，诸如需要占物理空间，建立过多后维护难度大，甚至建立后还反而会导致某些查询操作效率降低。

数据库的优化方法还有：

1.优化查询语句；

2.扩展硬件，加大物理内存，提高文件系统性能；

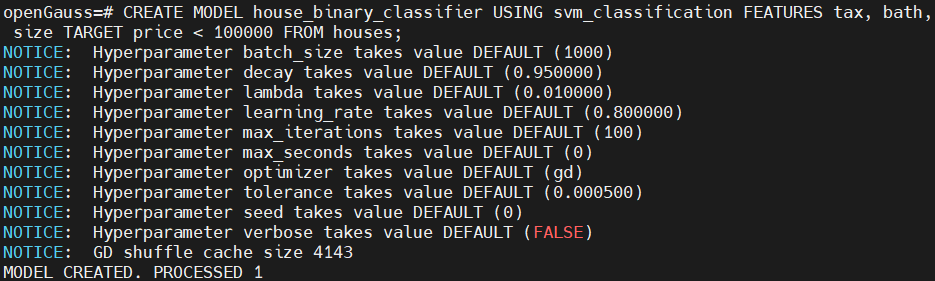
3.给数据库增加缓存系统，把热数据缓存到内存中；

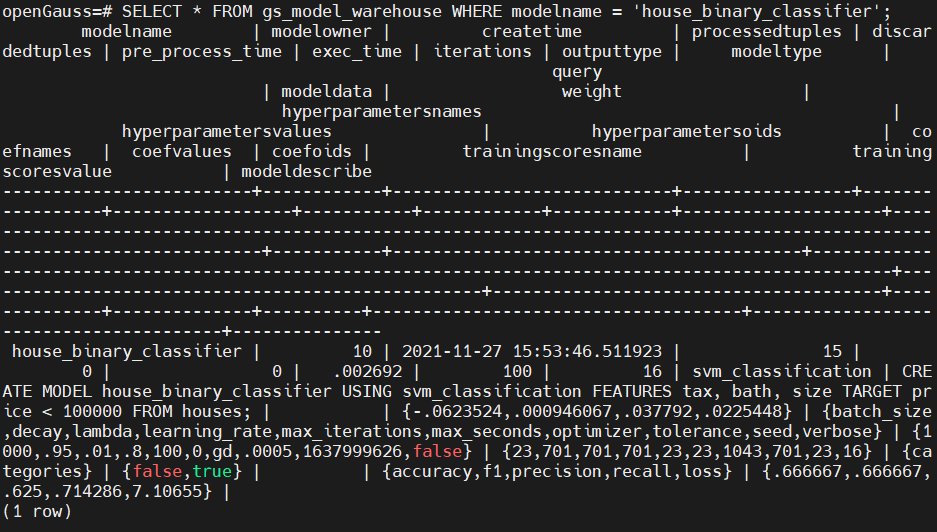
4.分库、分表、分区；

# 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：

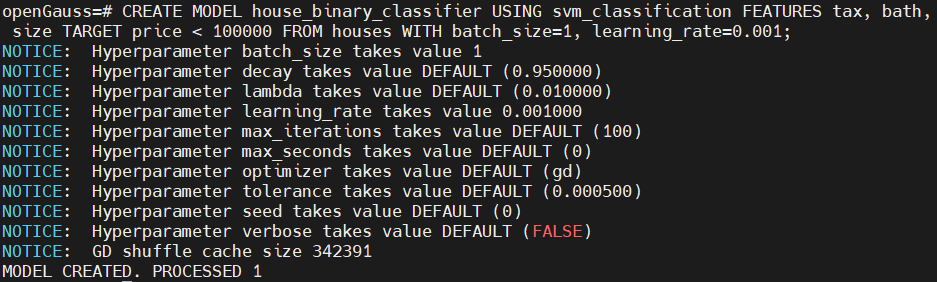
postgres=# SELECT \* FROM gs\_model\_warehouse WHERE modelname = 'house\_binary\_classifier';

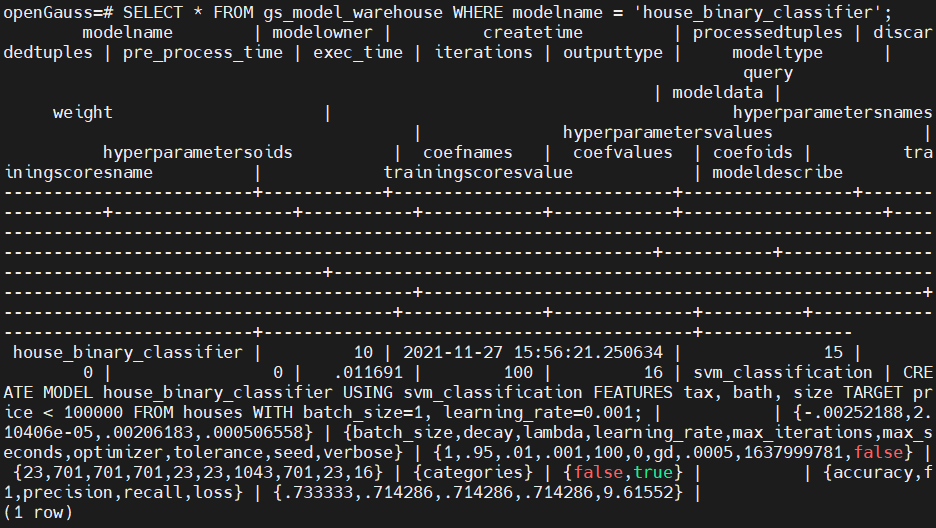




任务二：观察新模型的信息，将执行结果截图。

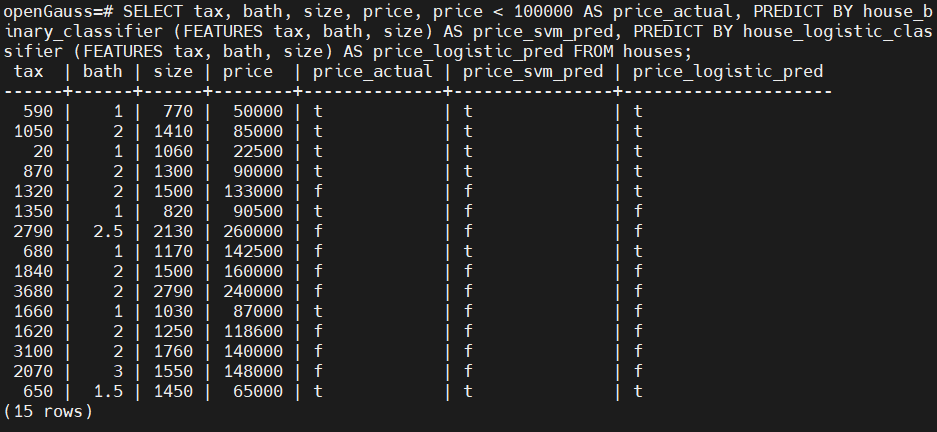
postgres=# SELECT \* FROM gs\_model\_warehouse WHERE modelname = 'house\_binary\_classifier';





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？

监督学习从数据中学习一个分类模型或者分类决策函数，称为分类器。分类器对新的输入预测其属于哪一类别，称为分类。

回归用于预测输入变量和输出变量之间的关系，特别是当输入变量的值发生变化时，输出变量的值也会跟着变化。回归模型正是表示输入变量到输出变量之间的映射函数，回归问题的学习等价于函数拟合。

因此分类问题与回归问题都是监督学习问题，区别在于学习函数的预测输出是类别还是值。此外，分类是确定边界，回归是寻找映射。总结不同之处如下：

|  |  |  |
| --- | --- | --- |
| 特性 | 分类（监督学习） | 回归 |
| 输出类型 | 离散数据 | 连续数据 |
| 目的 | 寻找决策边界 | 寻找最优拟合 |

实践思考题2：什么是SVM算法？

SVM是建立在统计学习理论的VC维理论和结构风险最小原理基础上的，根据有限的样本信息在模型的复杂性和学习能力之间寻求最佳折衷，以期获得最好的泛化能力。其思想主要是找到分隔的超平面离两边的数据尽可能大，即最大化间隔。同时，SVM采取了核函数，核函数的作用就是把低维映射到高维中，便于分类，同时使其能够处理非线性可分的数据。

SVM算法主要是为了最终得到决策函数，这也是分类面：



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？

混淆矩阵是用来总结一个分类器结果的矩阵，对于二分类问题，混淆矩阵是的矩阵，如下所示：

|  |  |  |  |
| --- | --- | --- | --- |
| A\Pr | 预测值=1 | 预测值=0 |  |
| **真实值=1** | TP | FN | P |
| **真实值=0** | FP | TN | N |
|  | P’ | N’ | All |

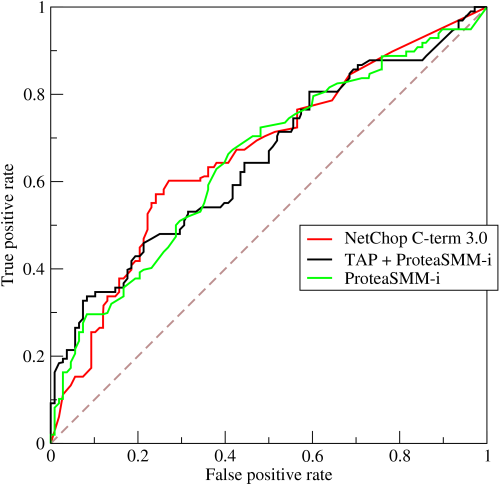
符号含义：TP为真阳性，FP为假阳性，FN为假阴性，TN为真阴性。

根据上面的混淆矩阵，分类问题就有以下评价指标：

|  |  |  |
| --- | --- | --- |
| 指标名 | 含义 | 公式 |
| 准确度/识别度 | 正确的比例 |  |
| 误差率 | 错误的比例 |  |
| 精度 | 被分类器标记为正的样本实际为正的比例 |  |
| 召回率 | 正样本标记为正的比例 |  |
| F度量//F-分数 | 精度和召回率的调和平均值 |  |
|  | 精度和召回率的加权值 |  |
| 灵敏度 | 正确识别的正样本比例 |  |
| 特效性 | 正确识别的负样本比例 |  |

后两个评价指标主要是根据类不平衡问题所提出，避免了类不平衡，对评价指标的影响。

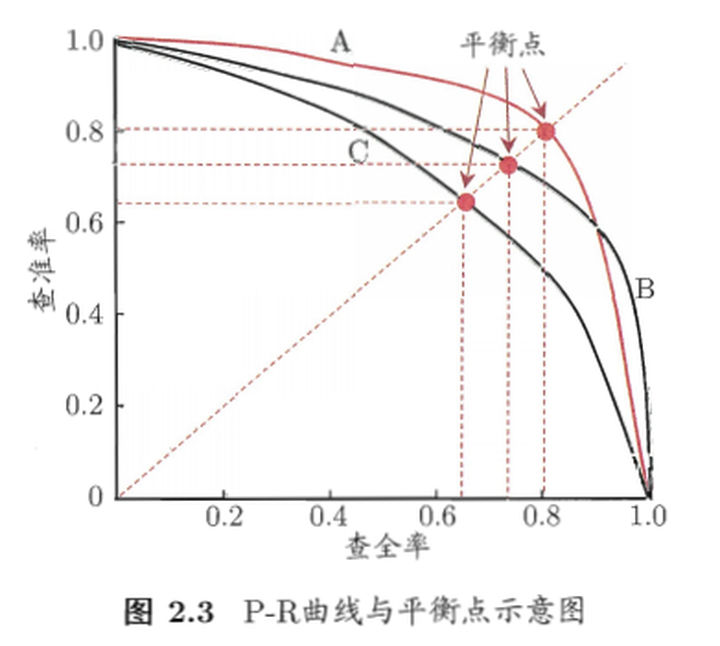
除此之外，还会使用ROC曲线，如图所示：



分类器对应的ROC曲线应该尽可能靠近坐标轴的左上角， 而对角线的位置意味着分类器的效果和随机猜测一样的差。

对应ROC曲线，AUC为ROC曲线下的面积， 这个面积的数值介于0到1之间， 能够直观的评价出分类器的好坏，AUC的值越大， 分类器效果越好。

另外，P-R曲线也是一评价方法。



曲线越靠上面，效果越好，当二者存在交叉的情况，此时采用曲线下面积大小衡量性能，面积越大，性能越好。

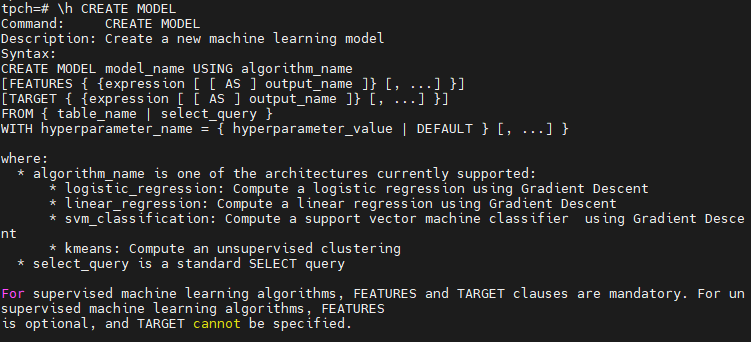
实践思考题4：回归问题有哪些评价指标，请分别说明他们的含义？

评价指标如下表所示。

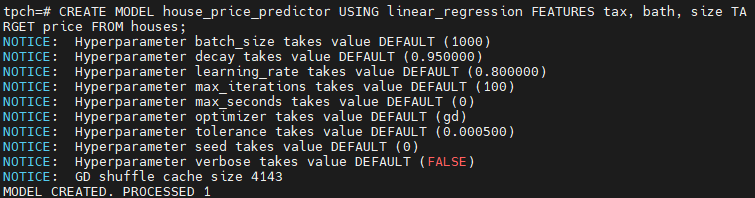
|  |  |  |
| --- | --- | --- |
| 指标 | 含义 | 公式 |
| 均方误差（MSE） | 描述了样本真实值与预测值差方求和的平均值，即为线性回归的损失函数 |  |
| RMSE | 对MSE开平方根 |  |
| 平均绝对误差（MAE） | 真实值与预测值差的绝对值和求平均 |  |
| MAPE | 所有样本真实值与预测值差绝对值与真实值的比例的和求平均 |  |
| 平均平方百分比误差（MSPE） | 所有样本的真实值与预测值的差与真实值的比例求平方的和求平均 |  |
| R-Square | 评估该模型线性拟合度 |  |

挑战二：使用Madlib中的其他功能，完成某个数据挖掘工作（如决策树）。

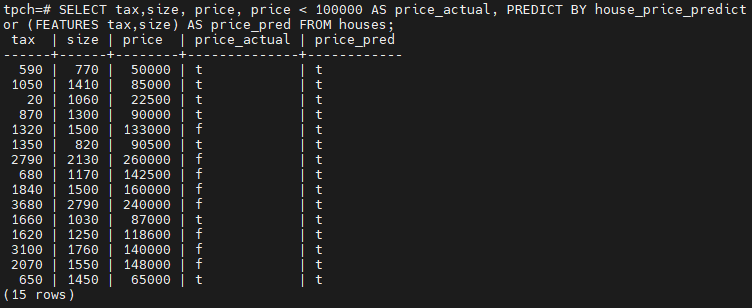
使用\h CREATE MODEL命令查看支持的算法，此处我们选择线性回归算法，目的是对房价进行线性拟合。



使用命令CREATE MODEL house\_price\_predictor USING linear\_regression FEATURES tax,size TARGET price<100000 FROM houses WITH batch\_size=1;



使用命令SELECT tax,size, price, price < 100000 AS price\_actual, PREDICT BY house\_price\_predictor (FEATURES tax,size) AS price\_pred FROM houses;查看结果，如下所示。



发现预测效果比较差。