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

```
[omm@opengauss01 openGauss-server]$ gs_ctl status
[2021-11-28 11:42:40.790][3308][][gs_ctl]: gs_ctl status,datadir is /opt/software/openGauss/d
gs_ctl: server is running (PID: 3243)
/opt/software/openGauss/bin/gaussdb "-D" "/opt/software/openGauss/data"
```

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）

```
[omm@opengauss01 openGauss-server]$ ps -ef|grep omm
root        3012    2950  0 11:27 pts/1    00:00:00 su - omm
omm         3013    3012  0 11:27 pts/1    00:00:00 -bash
omm         3243       1  1 11:41 pts/1    00:00:02 /opt/software/openGauss/bin/gaussdb -D /opt/software/openGauss/da
omm         3332    3013  0 11:44 pts/1    00:00:00 ps -ef
omm         3333    3013  0 11:44 pts/1    00:00:00 grep --color=auto omm
```

实验思考题：为什么需要通过源码编译，安装数据库？

(1) 数据库安装时需要统一的路径

(2) 编译安装可以灵活地定制软件。

(3) 通过源码编译安装可以针对数据库的硬件进行优化，以提升性能。

（4） 数据库在编译期间需要进行配置

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;;
```

```
tpch=# select count(*) from supplier;
 count
-------
 10000
(1 row)
```

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';
```

```
     sum
--------------
 310586483.00
(1 row)

Time: 256.420 ms
```

```
select sum (order_price) from litemall_orders_col where add_date between '20200101' and
'20200701';
```

```
     sum
--------------
 310586483.00
(1 row)

Time: 90.165 ms
```

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';
```

```
          avg
----------------------
 3105.8648300000000000
(1 row)

Time: 364.721 ms
```

```
select avg (order_price) from litemall_orders_col where add_date between '20200101' and '20200701';
```

```
          avg
----------------------
 3105.8648300000000000
(1 row)

Time: 77.427 ms
```

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;
```

```
 order_price
-------------
     2469.00
(1 row)

Time: 2.679 ms
```

```
select order_price from litemall_orders_col where order_id=6;
```

```
 order_price
-------------
     2469.00
(1 row)

Time: 5.507 ms
```

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 1
Time: 4.233 ms
```

```
update litemall_orders_col set order_price=2468 where order_id=6;
```

```
UPDATE 1
Time: 35.608 ms
```

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;
```

```
 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)

Time: 3.001 ms
```

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;
```

```
 count
-------
    15
(1 row)

Time: 2.878 ms
```

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。

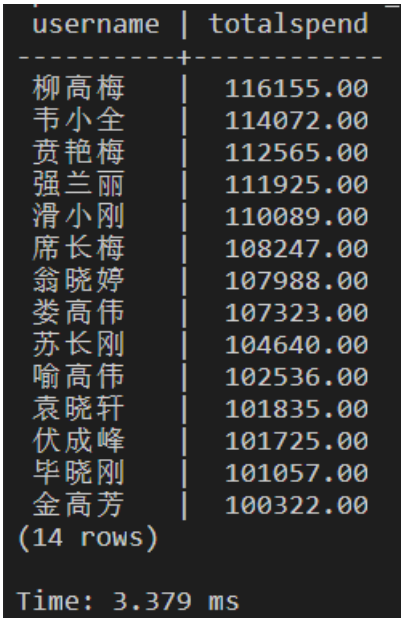
```
SELECT * FROM v_order;
```

```
 count
-------
    14
(1 row)

Time: 2.839 ms
```

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;
```

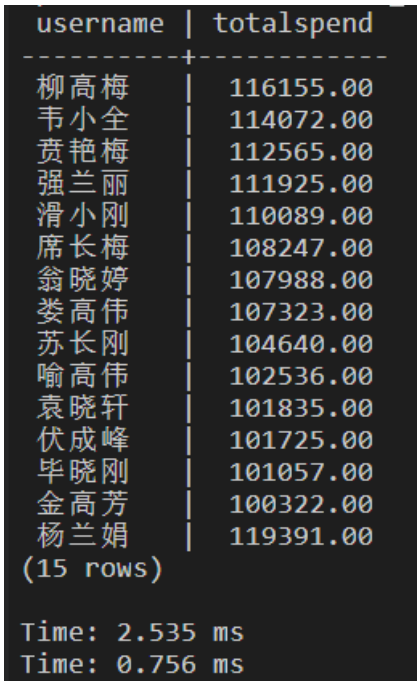


```
 username | totalspend
----------+------------
 柳高梅   |  116155.00
 韦小全   |  114072.00
 贲艳梅   |  112565.00
 强兰丽   |  111925.00
 滑小刚   |  110089.00
 席长梅   |  108247.00
 翁晓婷   |  107988.00
 娄高伟   |  107323.00
 苏长刚   |  104640.00
 喻高伟   |  102536.00
 袁晓轩   |  101835.00
 伏成峰   |  101725.00
 毕晓刚   |  101057.00
 金高芳   |  100322.00
(14 rows)

Time: 3.379 ms
```

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;
```



```
 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)

Time: 2.535 ms
Time: 0.756 ms
```

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？

(1) 在进行数据读取时，行存储通常需要将一行数据完全读出，但如果只需要其中几列数据，就会产生冗余列，而消除冗余列的过程通常是在内存中进行的，导致效率较低。

(2) 列存储每次读取的数据是集合的一段或者全部，不存在冗余性问题。

(3) 两种存储的数据分布不同。由于列存储的每一列数据类型是相同的，不存在二义性问题，这种情况使数据解析变得十分容易。而相比之下，行存储会复杂一些，在一行记录中可能保存了多种类型的数据，那么数据解析就需要在多种数据类型之间频繁转换，增加了很多解析时间，同时消耗了 CPU 资源。因此，列存储更有利于分析大数据。

由以上实验结果，发现在进行修改和选择语句时，行存储的数据库效率更高；然而在进行求和，求平均值等类似操作时，列存储数据库速度更快。

实践思考题 2：全量物化视图与增量物化视图有哪些差别？

全量物化视图只能对创建好的物化视图做全量更新，而不能做增量更新。

创建全量物化视图语法和 CREATE TABLE AS 语法一致。

增量物化视图可以对物化视图增量刷新，同时需要用户手动执行语句对物化视图进行刷新。但是目前增量物化视图所支持场景较小，目前物化视图创建语句仅支持基表扫描语句或者 UNION ALL 语句。

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
```

```
 l_returnflag | l_linestatus |   sum_qty   | sum_base_price  |  sum_disc_price
  |     sum_charge      |       avg_qty       |     avg_price      |       avg_di
sc        | count_order
--------------+--------------+-------------+-----------------+-----------------
-+---------------------+---------------------+--------------------+-------------
----------+-------------
 A            | F            | 37734107.00 |  56586554400.73 |  53758257134.8700
  |  55909065222.827692 | 25.5220058532573370 | 38273.129734621672 | .04998529583
839761162 |     1478493
 N            | F            |   991417.00 |   1487504710.38 |   1413082168.0541
  |   1469649223.194375 | 25.5164719205229835 | 38284.467760848304 | .05009342667
421629691 |       38854
 N            | O            | 74476040.00 | 111701729697.74 | 106118230307.6056
  | 110367043872.497010 | 25.5022267695849915 | 38249.117988908270 | .04999658605
370408037 |     2920374
 R            | F            | 37719753.00 |  56568041380.90 |  53741292684.6040
  |  55889619119.831932 | 25.5057936126907707 | 38250.854626099657 | .05000940583
012705647 |     1478870
(4 rows)
```

```
 13        |      888 | 6737713.99
 17        |      861 | 6460573.72
 18        |      964 | 7236687.40
 23        |      892 | 6701457.95
 29        |      948 | 7158866.63
 30        |      909 | 6808436.13
 31        |      922 | 6806670.18
(7 rows)

total time: 1192928  ms
```

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
```

```
********************** Recommended Knob Settings **********************
+---------------------------+-----------+----------+----------+---------+
|            name           | recommend |   min    |   max    | restart |
+---------------------------+-----------+----------+----------+---------+
| default_statistics_target |    1000   |   100    |   1000   |  False  |
|    effective_cache_size   |  21602940 |  187192  | 21602940 |  False  |
|  effective_io_concurrency |    200    |   150    |   250    |  False  |
|      enable_mergejoin     |    off    |    0     |    1     |  False  |
|      enable_nestloop      |    off    |    0     |    1     |  False  |
|      max_connections      |    370    |    50    |   741    |   True  |
| max_prepared_transactions |    370    |    50    |   741    |   True  |
|     max_process_memory    |  28803920 | 22403048 | 28803920 |   True  |
|      random_page_cost     |    1.0    |   1.0    |   2.0    |  False  |
|       shared_buffers      |   187192  |  187196  |  215272  |   True  |
|        wal_buffers        |    5849   |   2048   |   5849   |   True  |
+---------------------------+-----------+----------+----------+---------+
```

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'
```

```
max_connections = 370                   # (change requires restart)
# Note:  Increasing max_connections costs ~400 bytes of shared memory per
shared_buffers = 186864                 # min 128kB
bulk_write_ring_size = 2GB              # for bulkload, max shared_buffers
#standby_shared_buffers_fraction = 0.3 #control shared buffers use in standby, 0.1-1.0
effective_io_concurrency = 200          # 1-1000; 0 disables prefetching
wal_buffers = 5839                      # min 32kB
random_page_cost = 1                    # same scale as above
effective_cache_size = 21602940
default_statistics_target = 1000        # range 1-10000
# max_locks_per_transaction * (max_connections + max_prepared_transactions)
```

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;
```

```
------------------------------------------------------------------------------
---------------------------
 Sort  (cost=4593.80..4593.88 rows=31 width=47)
   Sort Key: (sum(o.actual_price)) DESC
   ->  HashAggregate  (cost=4592.72..4593.03 rows=31 width=47)
         Group By Key: ad.province
         ->  Hash Join  (cost=4354.43..4585.97 rows=1351 width=15)
               Hash Cond: (ad.address_key = o.address_key)
               ->  Seq Scan on address_dimension ad  (cost=0.00..188.02 rows
=8002 width=14)
               ->  Hash  (cost=4337.54..4337.54 rows=1351 width=9)
                     ->  Hash Join  (cost=1031.78..4337.54 rows=1351 width=9
)
                           Hash Cond: (o.add_date = dd.date_key)
                           ->  Seq Scan on litemall_orders o  (cost=0.00..30
41.00 rows=100000 width=13)
                           ->  Hash  (cost=1031.76..1031.76 rows=2 width=4)
                                 ->  Seq Scan on date_dimension dd  (cost=0.
00..1031.76 rows=2 width=4)
                                       Filter: ((year = 2020) AND ((month)::
bigint = 3))
(14 rows)
```

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');
```

```
 schema |       table        |          column
--------+--------------------+--------------------------
 public | litemall_orders    | (address_key,add_date)
 public | address_dimension  |
 public | date_dimension     | (year)
(3 rows)
```

3. 查看创建的虚拟索引列，将执行结果截图。

```
select * from hypopg_display_index();
```

```
                    indexname                    | indexrelid |      table
      |         column
---------------------------------------------------+------------+-----------
------+-------------------------
 <24576>btree_litemall_orders_address_key_add_date |      24576 | litemall_o
rders | (address_key, add_date)
 <24577>btree_date_dimension_year                  |      24577 | date_dimen
sion  | (year)
(2 rows)
```

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);
```

```
tpch=# select * from hypopg_estimate_size(24576);
 hypopg_estimate_size
----------------------
              4177920
(1 row)
```

```
tpch=# select * from hypopg_estimate_size(24577);
 hypopg_estimate_size
----------------------
               696320
(1 row)
```

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;
```

```
------------------------------------------------------------------
 Sort  (cost=3579.58..3579.65 rows=31 width=47)
   Sort Key: (sum(o.actual_price)) DESC
   ->  HashAggregate  (cost=3578.50..3578.81 rows=31 width=47)
         Group By Key: ad.province
         ->  Hash Join  (cost=3340.21..3571.74 rows=1351 width=15)
               Hash Cond: (ad.address_key = o.address_key)
               ->  Seq Scan on address_dimension ad  (cost=0.00..188.02 rows
=8002 width=14)
               ->  Hash  (cost=3323.32..3323.32 rows=1351 width=9)
                     ->  Hash Join  (cost=17.56..3323.32 rows=1351 width=9)
                           Hash Cond: (o.add_date = dd.date_key)
                           ->  Seq Scan on litemall_orders o  (cost=0.00..30
41.00 rows=100000 width=13)
                           ->  Hash  (cost=17.53..17.53 rows=2 width=4)
                                 ->  Index Scan using <24577>btree_date_dime
nsion_year on date_dimension dd  (cost=0.00..17.53 rows=2 width=4)
                                       Index Cond: (year = 2020)
                                       Filter: ((month)::bigint = 3)
(15 rows)
```

6. 删除某一个索引虚拟列，将执行结果截图。

```
select * from hypopg_drop_index(16715);
```

```
 hypopg_drop_index
-------------------
 t
(1 row)
```

7. 删除所有索引虚拟列，将执行结果截图。

```
select * from hypopg_reset_index();
```

```
 hypopg_reset_index
--------------------

(1 row)
```

8. 查看索引虚拟列，将执行结果截图。

```
select * from hypopg_display_index();
```

```
 indexname | indexrelid | table | column
-----------+------------+-------+--------
(0 rows)
```

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
```

```
 13          |      888 | 6737713.99
 17          |      861 | 6460573.72
 18          |      964 | 7236687.40
 23          |      892 | 6701457.95
 29          |      948 | 7158866.63
 30          |      909 | 6808436.13
 31          |      922 | 6806670.18
(7 rows)

total time: 331855  ms
```

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？

对 name 栏中的 effective_cache_size,effective_io_concurrency、shared_buffers、max_connections、wal_buffers、random_page_cost、default_statistics_target,进行了优化。

优化参数可以提高数据库运行效率，减少资源消耗。

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？

好处：加快数据库查询速度，当 DB 执行一条 Sql 语句时，默认是根据搜索条件进行全表扫描，遇到匹配条件的就加入搜索结果集合。如果对某一字段增加索引，查询时就会先去索引列表中一次定位到特定值的行数，大大减少遍历匹配的行数，能明显增加查询的速度。

可以改变数据库结构：行存储和列存储在不同操作时可以显著增加运行速度。数据库表的范式化。提高硬件水平等。

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';
```

```
         modelname          | modelowner |          createtime
| processedtuples | discardedtuples | pre_process_time | exec_time
| iterations | outputtype |      modeltype      |
                                  query
                                 | modeldata |                  weig
ht                    |                                       hyperp
arametersnames                                                  |
    hyperparametersvalues                    |            hyperparameterso
ids              |   coefnames    |  coefvalues  | coefoids |          tra
iningscoresname           |          trainingscoresvalue            |
 modeldescribe
--------------------------+------------+----------------------------
+-----------------+-----------------+-------------------+-----------
+------------+------------+---------------------+-------------------
--------------------------------------------------------------------
------------------------------------+------------+------------------
-------------------+------------------------------------------------
---------------------------------------------------------+-----------
-------------------------------------+------------------------------
------------+---------------+---------------+----------+------------
--------------------------+---------------------------------------+
---------------
 house_binary_classifier |          10 | 2021-11-28 14:24:35.274386
|              15 |                 0 |                 0 |   .002665
|        100 |          16 | svm_classification | CREATE MODEL house
_binary_classifier USING svm_classification FEATURES tax,bath,size
TARGET price<100000 FROM houses; |            | {-.0623524,.00094606
7,.037792,.0225448} | {batch_size,decay,lambda,learning_rate,max_it
erations,max_seconds,optimizer,tolerance,seed,verbose} | {1000,.95,
.01,.8,100,0,gd,.0005,1638080675,false} | {23,701,701,701,23,23,104
3,701,23,16} | {categories} | {false,true} |           | {accuracy,f
1,precision,recall,loss} | {.666667,.666667,.625,.714286,7.10655} |

(1 row)
```

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';
```

```
          modelname         | modelowner |        createtime         | processedtuples | discardedtuples | pre_process_time | exec_time | iterations | outputtype |
odeltype        |                                                                                    query
            | modeldata |                                weight                        |                                                    hyperparametersnames
           |                   hyperparametersvalues                      |            hyperparametersoids             |  coefnames   |  coefvalues  | coefoids |         traini
resname         |          trainingscoresvalue           | modeldescribe
----------------------------+-------------+---------------------------+-----------------+-----------------+------------------+-----------+------------+------------+--
--------------+------------------------------------------------------------------------------------------------------------------------------------------------------------
------------+-----------+-------------------------------------------------+---------------------------------------------------------------------------------------------
-----------+----------------------------------------------------------------+----------------------------+--------------+--------------+----------+-----------------
-----------------+--------------------------------------+---------------
 house_binary_classifier |          10 | 2021-11-28 14:27:25.691 |              15 |               0 |                0 |   .011667 |        100 |         16 | s
lassification | CREATE MODEL house_binary_classifier USING svm_classification FEATURES tax, bath, size TARGET price < 100000 FROM houses WITH batch_size=1, lear
_rate=0.001; |           | {-.00232914,1.09564e-05,.00199586,.000433053} | {batch_size,decay,lambda,learning_rate,max_iterations,max_seconds,optimizer,toleranc
d,verbose} | {1,.95,.01,.001,100,0,gd,.0005,1638080845,false} | {23,701,701,701,23,23,1043,701,23,16} | {categories} | {false,true} |          | {accuracy,f1,pr
ion,recall,loss} | {.8,.769231,.833333,.714286,9.7806} |
(1 row)
```

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;
```

```
 tax  | bath | size | price  | price_actual | price_svm_pred | price_logistic_pre
------+------+------+--------+--------------+----------------+-------------------
  590 |    1 |  770 |  50000 | t            | t              | t
 1050 |    2 | 1410 |  85000 | t            | t              | t
   20 |    1 | 1060 |  22500 | t            | t              | t
  870 |    2 | 1300 |  90000 | t            | t              | t
 1320 |    2 | 1500 | 133000 | f            | f              | t
 1350 |    1 |  820 |  90500 | t            | f              | f
 2790 |  2.5 | 2130 | 260000 | f            | f              | f
  680 |    1 | 1170 | 142500 | f            | t              | t
 1840 |    2 | 1500 | 160000 | f            | f              | f
 3680 |    2 | 2790 | 240000 | f            | f              | f
 1660 |    1 | 1030 |  87000 | t            | f              | f
 1620 |    2 | 1250 | 118600 | f            | f              | f
 3100 |    2 | 1760 | 140000 | f            | f              | f
 2070 |    3 | 1550 | 148000 | f            | f              | f
  650 |  1.5 | 1450 |  65000 | t            | t              | t
(15 rows)
```

实践思考题 1：分类模型与回归模型有何不同？

回归是定量输出，是连续变量预测；
分类是定性输出，是离散变量预测。

分类是关于标签的预测，而回归是关于数量的预测。

实践思考题 2：什么是 SVM 算法？

SVM 是一种二分类模型，它将实例的特征向量映射为空间中的一些点，SVM 的目的就是想要画出一条线，以 “最好地” 区分这两类点，以至如果以后有了新的点，这条线也能做出很好的分类。SVM 适合中小型数据样本、非线性、高维的分类问题。

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？

准确率 = 算法分类正确的数据个数/输入算法的数据的个数 A=(TP+TN)/(T+F)

精确率 = 预测为正的样本占所有正样本的比重 P=TP/(TP+FP)

召回率 = 正确预测的数据在总样本中的比重 R=TP/(TP+FN)=TP/T

F1 值 = 精确率和召回率的兼顾指标，是精确率和召回率的调和平均数。调和平均数的性质，只有当精确率和召回率二者都非常高的时候，它们的调和平均才会高。如果其中之一很低，调和平均就会被拉得接近于那个很低的数

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？

RMSE：均方根误差衡量观测值与真实值之间的偏差。常用来作为机器学习模型预测结果衡量的标准。

MSE：均方误差 MSE 是真实值与预测值的差值的平方然后求和平均。

通过平方的形式便于求导，所以常被用作线性回归的损失函数。

MAE：平均绝对误差是绝对误差的平均值。可以更好地反映预测值误差的实际情况。

SD：标准差方差的算术平均根。用于衡量一组数值的离散程度。