openGauss 安全体系创新实践课

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@opengauss openGauss-server]$ gs_ctl status
[2023-04-17 16:00:47.745][27054][][gs_ctl]: gs_ctl status,datadir is /opt/software/openGauss/
dest/data
gs_ctl: server is running (PID: 26805)
/opt/software/openGauss/dest/bin/gaussdb "-D" "/opt/software/openGauss/dest/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@opengauss openGauss-server]$ ps -ef|grep omm
root       26594   26550  0 15:39 pts/1    00:00:00 su - omm
omm        26595   26594  0 15:39 pts/1    00:00:00 -bash
omm        26805       1  1 15:53 ?        00:00:08 /opt/software/openGauss/dest/bin/gaussdb
-D /opt/software/openGauss/dest/data
omm        27059   26595  0 16:00 pts/1    00:00:00 ps -ef
omm        27060   26595  0 16:00 pts/1    00:00:00 grep --color=auto omm
```

任务三：实践思考题

思考题 1：为什么需要通过源码编译，安装数据库？

（1）平台适配性：OpenGauss 需要针对不同的操作系统和硬件平台进行适配，通过源码编译可以确保 OpenGauss 能够在目标平台上运行。

（2）自定义配置：通过源码编译可以进行更细粒度的配置，例如指定安装路径、数据库端口号、最大连接数等。

（3）安全性：源码编译可以确保我们使用的是完全自由和可验证的代码，不会受到潜在的恶意代码和漏洞的影响。

（4）源码定制化：如果需要对 OpenGauss 进行修改或添加功能，可以根据需要对 OpenGauss 进行修改，并将修改后的代码重新编译安装。

关卡二、openGauss 数据导入及行存列存

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';
```

```
tpch=# select sum (order_price) from litemall_orders where add_date between '20200101' and '20200701';
     sum
--------------
 310586483.00
(1 row)

Time: 244.640 ms
```

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

```
tpch=#  select sum (order_price) from litemall_orders_col where add_date between '20200101' and '20200701';
     sum
--------------
 310586483.00
(1 row)

Time: 23.924 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';
select avg (order_price) from litemall_orders_col where add_date between '20200101' and '20200701';
```

```
tpch=# select avg(order_price) from litemall_orders where add_date between '20200101' and '20200701';
         avg
----------------------
 3105.8648300000000000
(1 row)

Time: 328.996 ms
tpch=#  select avg(order_price) from litemall_orders_col where add_date between '20200101' and '20200701';
         avg
----------------------
 3105.8648300000000000
(1 row)

Time: 15.285 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;
```

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

```
tpch=#  select order_price from litemall_orders where order_id=6;
 order_price
-------------
     2469.00
(1 row)

Time: 1.529 ms
tpch=# select order_price from litemall_orders_col where order_id=6;
 order_price
-------------
     2469.00
(1 row)

Time: 5.038 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 litemall_orders_col set order_price=2468 where order_id=6;
```

```
tpch=# update litemall_orders set order_price=2468 where order_id=6;
UPDATE 1
Time: 4.563 ms
tpch=# update litemall_orders_col set order_price=2468 where order_id=6;
UPDATE 1
Time: 38.086 ms
```

任务三：实践思考题

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？

当执行相同的 SQL 语句时，行存表和列存表的执行时间可能会不同，这是因为它们的数据存储方式不同，导致对数据的访问方式不同，从而影响了查询的性能。对于查询需要涉及到的列，行存表需要读取整行的数据，而列存表只需要读取所需的列，因此对于需要访问的列数较少的查询，列存表的查询性能会更好。此外，行存表中的数据是按照行存储的，因此在执行聚合函数（如 SUM、COUNT）等操作时，需要对整行的数据进行扫描，而列存表中的数据是按照列存储的，因此这些操作可以直接针对某一列进行计算，从而提高了查询性能。另外，行存表的索引是基于整行数据建立的，而列存表的索引则是基于列数据建立的。因此，在需要使用索引的查询中，行存表和列存表的性能也可能会有所不同。

思考题 2：

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：

（1）在查询涉及到大部分或全部列的情况下，行存表的性能通常比列存表更好。因为行存表中的数据是按照行存储的，所以在查询时可以直接读取整行数据，避免了频繁的磁盘 I/O 操作。

（2）在更新或插入单个记录的情况下，行存表的性能通常比列存表更好。因为行存表中的数据是按照行存储的，所以单个记录的更新或插入可以直接在行中进行操作，不需要进行额外的列操作。

（3）在事务处理方面，行存表通常比列存表更有优势。因为行存表中的数据是以行为单位进行事务处理的，一次提交可以提交多行数据。

列存表：

（1）在查询仅涉及部分列的情况下，列存表的性能通常比行存表更好。因为列存表中的数据是按照列存储的，所以只需要读取所需的列，避免了读取整行数据的开销。

（2）在对大量数据进行聚合计算时，列存表的性能通常比行存表更好。因为列存表中的数据是按照列存储的，聚合计算可以针对某一列进行操作，从而提高了计算效率。

（3）在数据仓库和分析应用方面，列存表通常比行存表更适用。因为数据仓库和分析应用通常需要大量的数据分析和聚合计算，而列存表可以提供更高效的数据读取和计算功能。

关卡三：openGauss 物化视图应用

任务一：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;
```

```
tpch=#  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)
```

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;
```

```
tpch=# SELECT * FROM v_order;
 count
-------
    15
(1 row)
```

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;
```

```
tpch=#  SELECT * FROM v_order;
 count
-------
    14
(1 row)
```

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;
```

```
tpch=# 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)
```

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;
```

```
tpch=# 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)
```

```
tpch=# 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
 马景涛   |  139391.00
(16 rows)
```

任务二：实践思考题

思考题 1：全量物化视图与增量物化视图有哪些差别？

（1）全量物化视图：全量物化视图在创建时会对原始数据进行完整的聚合和计算，并将结果存储在物化视图中。物化视图中的数据会随着源数据的更新而变化，但每次更新时都需要重新计算整个视图，因此全量物化视图的更新成本比较高，适用于数据量较小、更新频率较低的场景。

（2）增量物化视图：增量物化视图在创建时也会进行聚合和计算，但在后续的更新过程中，它只需要对新增、修改、删除的数据进行计算，而无需重新计算整个视图。因此，增量物化视图的更新成本比较低，适用于数据量较大、更新频率较高的场景。

思考题 2：物化视图适用那些使用场景？

物化视图（Materialized View）适用于需要快速访问大量复杂数据的场景，特别是在涉及联结、聚合和过滤操作时。物化视图可以将这些操作提前执行并缓存结果，从而加速后续查询的执行速度。物化视图通常用于数据仓库或业务智能系统中，用于支持复杂分析和报告。

以下是一些适合使用物化视图的情况：

（1）频繁执行的复杂查询：如果有一个查询需要执行多个联结和聚合操作，而且需要处理大量数据，物化视图可以缓存结果并加速查询。

（2）大数据量的分析：如果需要对大量数据进行分析和报告，物化视图可以在数据仓库中缓存中间结果，从而提高查询性能。

（3）需要经常更新的数据：物化视图可以在后台自动刷新，从而确保缓存的数据与源数据保持同步。

关卡四：openGauss 密态数据库特性应用

任务一：物化视图的使用

1. 通过 tcpdump 抓取数据流，此 putty 窗口暂时保持不动，将执行结果截图：

```
[root@opengauss software]# tcpdump -i any tcp port 5432 and host 127.0.0.1 -w /opt/encryption.cap
dropped privs to tcpdump
tcpdump: listening on any, link-type LINUX_SLL (Linux cooked v1), capture size 262144 bytes
```

2. 将加密表和非加密表查询结果截图：

```
openGauss=# SELECT * FROM creditcard_info_unce ORDER BY id_number;
 id_number |   name    |     credit_card
-----------+-----------+---------------------
         1 | xiaoming  | 6227 1111 1111 1111
         2 | zhangsan  | 6227 2222 2222 2222
         3 | liuhua    | 6227 3333 3333 3333
(3 rows)

openGauss=# SELECT * FROM creditcard_info ORDER BY id_number;
 id_number |   name    |     credit_card
-----------+-----------+---------------------
         1 | xiaoming  | 6227 1111 1111 1111
         2 | zhangsan  | 6227 2222 2222 2222
         3 | liuhua    | 6227 3333 3333 3333
(3 rows)
```

3. 用 wireshark 解析加密表和非加密表的差异时，非加密表 name 列和 credit_card 列是明文，加密表 name 列和 credit_card 列均是密文，将执行结果截图：

```
SHOW.Z....I.Q...;SELECT * FROM creditcard_info_unce ORDER BY
id_number;.T...W..id_number...@..............name...@..............credit_card...@..............D....
......1....xiaoming....6227 1111 1111 1111D.........2....zhangsan....6227 2222 2222 2222D...,......
3....liuhua....6227 3333 3333 3333C...
SELECT 3.Z....I.Q...6SELECT * FROM creditcard_info ORDER BY
id_number;.T...W..id_number...@..............name...@......2........credit_card...@......2........D...
[......1....
\x0193f3522dc60e46d2756dca799c09828f4b9fc3a5751dda731f22f259aa7671e8e2423f92310000005407034c13f1c996fd85
ec642f97a43e12fe4893f8e09382331475b04fe6eb92....
\x0193f3522de20094ca8173cb1bf734c8ac4a63813f811e4699482c10881d306c7711918d3331000000ec64d3f9862acca208dd
10970b83b9f1dbf75080345312d3409d8b0bddfdc80abb7d1343b89f1c506fc72115c92df36aD...[......2....
\x0193f3522dda9415d2ccc2ec67fd6aa11733eab5b97790ddbbd9aed641716df54413c618eb3100000059c8bf7c12546bb6e760
253b92b903956d7b169624bc0610305dcf9f93bce0c1....
\x0193f3522d12fab4b96bb3e6d405f16a09716970595bbaf98320787a9525de6a14dbd2830031000000d67fee9600794913e74f
7646e90fe9f39ecaa9d2d5a11b20a9da861b4b71288ef7609268e151151e76f6a45e0560e828D...[......3....
\x0193f3522da702c55393d153a88dc0ebef0239dc39aee7ac1e864c84d0435e11fc7a5c4de3310000003e11c247f1b1e8961db4
54394b7adb68521a2a6b649051a036a96baee6b0b8ad....
\x0193f3522d71fe0be211b5f5b7caf719c9be12ca59861bc191c85f9e85a5932ae5e69c809431000000fa29f8aedfb337df0328
9a19cfa668a4faeac4f260931bc18f7d9ea3716c7bb1705fccab8aa0dd3937d0782e94018da8C...
SELECT 3.Z....I.
```

分组 41，18 客户端 分组，18 服务器 分组，35 turn(s). 点击选择。

4. 查询加密表，查询到的结果为密文，将执行结果截图：

```
openGauss=# SELECT * FROM creditcard_info ORDER BY id_number;
 id_number |                                                      name
                                                      |
                         credit_card

-----------+----------------------------------------------------------------------------------
---------------------------------------------------+----------------------------------------------
------------------------------------------------------------------------------------------------
--------------------
         1 | \x0193f3522dc60e46d2756dca799c09828f4b9fc3a5751dda731f22f259aa7671e8e2423f92310000005407034c13f
1c996fd85ec642f97a43e12fe4893f8e09382331475b04fe6eb92 | \x0193f3522de20094ca8173cb1bf734c8ac4a63813f811e4699
482c10881d306c7711918d3331000000ec64d3f9862acca208dd10970b83b9f1dbf75080345312d3409d8b0bddfdc80abb7d1343b89f
1c506fc72115c92df36a
         3 | \x0193f3522da702c55393d153a88dc0ebef0239dc39aee7ac1e864c84d0435e11fc7a5c4de3310000003e11c247f1b
1e8961db454394b7adb68521a2a6b649051a036a96baee6b0b8ad | \x0193f3522d71fe0be211b5f5b7caf719c9be12ca59861bc191
c85f9e85a5932ae5e69c809431000000fa29f8aedfb337df03289a19cfa668a4faeac4f260931bc18f7d9ea3716c7bb1705fccab8aa0
dd3937d0782e94018da8
(2 rows)
```

任务二：实践思考题

思考题 1：

数据实际存储在物理磁盘上的时候是明文还是密文？数据的加解密的动作是在客户端完成的还是服务端完成的？

数据在物理磁盘上存储时通常是以明文形式存储的。这是因为数据加密和解密需要消耗大量的计算资源，如果所有数据都以密文形式存储，会对存储和访问数据造成很大的性能开销。

在一些安全要求较高的场景下，数据可能会被加密存储，这通常会采用硬件加密的方式，即使用专用的加密芯片对数据进行加密，从而在保证数据安全的同时尽可能地减少性能开销。

至于数据的加解密动作是在客户端完成还是服务端完成，则取决于具体的应用场景和加密方案。在一些应用中，数据可能在客户端进行加密后再传输到服务端存储，而在访问数据时，则需要先从服务端获取加密数据，再在客户端进行解密。而在另一些应用中，数据则可能是在服务端存储时进行加密，访问时则由服务端进行解密后再返回给客户端。