《数据库系统原理》课程实验指导

openGauss 查询优化

2023 年 9 月

目录

前　言

实验环境说明

本实验环境为 virtualBOX 虚拟机 openEuler20.03 系统上的 openGauss1.1.0/openGauss2.0.0 数据库和华为云 GaussDB(openGauss)数据库，实验数据采用 TPC-H 电商数据库的八张表。

1 查询优化实验

1.1 实验目的

在 openGauss 平台下，通过观察 Select/Insert/delete/update 等 SQL 语句的查询执行计划，分析查询执行计划中连接、选择、投影等关系代数操作的实现方式及其执行成本。熟悉了解 openGauss 数据库中查询优化的使用，理解数据库查询优化的基本概念。

掌握利用 explain 命令，分析对比形式不同、执行结果等价的不同 SQL 语句的查询执行计划的执行成本和执行时间差异。

熟悉了解视图和 with 临时视图的创建，观察视图查询、with 临时视图查询的执行计划。

参照教科书中 SQL 语句查询优化相关内容，在多种情况下，对比实现方式不同但查询结果相同的等价 SQL 语句在执行计划和成本方面的差异，加深对查询优化的理解，进行书写优化 SQL 语句的初步训练，提高编写高效 SQL 语句的能力。

1.2 实验要求

涉及以下几方面：

1）复合索引左前缀；

2）多表连接属性上建立索引；

3）索引对小表查询的作用；

4）查询条件中函数对索引的影响；

5）多表嵌入式SQL查询；

6）where查询条件中复合查询条件OR对索引的影响；

7）聚集运算中索引设计；

8）Select子句中有无distinct的区别；

9）union、union all的区别；

10）from 子句中多余关系的影响。

1.3 实验总结

在实验中有哪些重要问题或者事件？你如何处理的？你的收获是什么？有何建议和意见等等。

2 实验示例

2.1 openGauss 执行计划的查看与分析

实验要求

在 openGauss 平台下，按照下述实验步骤，编写相应的 SQL 语句，观察 Select/Insert/delete/update 等 SQL 语句的查询执行计划，分析查询执行计划中连接、选择、投影等关系代数操作的实现方式，观察分析查询执行计划中关系代数操作的执行成本。

步骤 1. 编写 SQL 语句完成下面两条查询；

查询零部件表中零件零售价小于 920 且在零部件供应表中零件供应商 key 为 5 的零件 key。

查询客户表中客户账户余额小于 1000 且客户国家为 ALGERIA 的客户名称和客户电话。

步骤 2. 查看语句的执行计划。

步骤 3. 分析执行计划中各关系代数的实现方式与执行成本。

实验过程

示例：查询订单明细表中，使用了零件 key 为 1 的订单价格和订单状态。

执行如下 SQL 语句：

```
EXPLAIN
select A.l_extendedprice ,C.o_orderstatus
from lineitem as A, part as B, orders as C
where A.l_partkey = B.p_partkey and A.l_orderkey=C.o_orderkey and B.p_partkey =1;
```

查看执行计划

```
                                  QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..40181.90 rows=31 width=10)
   ->  Nested Loop  (cost=0.00..39924.19 rows=31 width=12)
         ->  Index Only Scan using part_pkey on part b  (cost=0.00..8.27 rows=1 width=4)
               Index Cond: (p_partkey = 1)
         ->  Seq Scan on lineitem a  (cost=0.00..39915.61 rows=31 width=16)
               Filter: (l_partkey = 1)
   ->  Index Scan using orders_pkey on orders c  (cost=0.00..8.30 rows=1 width=6)
         Index Cond: (o_orderkey = a.l_orderkey)
(8 rows)

total time: 0  ms
```

分析:

执行计划需要看每个字段对应的含义，理论上，通过优化开销最大的部分可以优化整个查询。

按照执行计划层级解读（纵向）：

第一层：Seq Scan on lineitem a，表扫描算子，在表 a 上执行全表顺序扫描，扫描检查的条件是 l_partkey=1,这一层的作用是把表 a 符合检查条件的数据从 buffer 或者磁盘上读上来输送给上层节点参与计算。预计执行时间（本层）为 0-39915.61 毫秒，预计输出的行数为 31 行。

第二层：Index Only Scan using part_pkey on part b，表扫描算子，在表 b 上执行主键上的索引扫描（表 b 在建立时创建过主键索引）。扫描检查的条件是 l_partkey=1，这一层的作用是把表 b 符合检查条件的数据从 buffer 或者磁盘上读上来输送给上层节点参与计算。预计执行时间（本层）为 0-8.27 毫秒，预计输出的行数为 1 行。

第三层：Nested Loop，表连接算子，主要作用是将第一层输出的数据和第二层输出的数据通过嵌套循环的方式连接，并输出结果数据。预计执行时间（整个查询的总时间）为 0-39924.19 毫秒，预计输出的行数为 31 行。

第四层：Index Scan using orders_pkey on orders c，表扫描算子，在表 c 上执行主键上的索引扫描（表 c 在建立时创建过主键索引）。扫描检查的条件是 c.o_orderkey=a.l_orderkey，这一层的作用是把表 c 符合检查条件的数据从 buffer 或者磁盘上读上来输送给上层节点参与计算。预计执行时间（本层）为 0-8.30 毫秒，预计输出的行数为 1 行。

第五层：Nested Loop，表连接算子，主要作用是将第三层输出的数据和第四层输出的数据通过嵌套循环的方式连接，并输出结果数据。预计执行时间（整个查询的总时间）为 0-40181.90 毫秒，预计输出的行数为 31 行。

2.2 观察视图查询、with 临时视图查询的执行计划

实验要求

从实验数据库中选取一张表在上面建立视图和 with 临时视图，执行一个查询，观察其执行计划。

1. 选取数据表的一张表（示例中的表除外），分别在上面建立视图和 with 临时视图；
2. 分别在两个视图上进行相同的查询；
3. 查看执行计划并分析。

实验过程

1.在orders上建立视图

```
CREATE VIEW orders_view
AS
SELECT *
FROM orders;
```

```
CREATE VIEW
```

2.查询并查看执行计划

通过视图进行查询

```
EXPLAIN ANALYZE
SELECT o_orderkey
FROM orders_view;
```

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on orders  (cost=0.00..7286.00 rows=300000 width=4) (actual time=0.017..51.848 rows=300000 loops=1)
 Total runtime: 67.418 ms
(2 rows)

total time: 68  ms
```

通过with临时视图进行查询

```
EXPLAIN ANALYZE
WITH orders_temview AS(
SELECT *
FROM orders
)
SELECT o_orderkey
FROM orders_temview;
```

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 CTE Scan on orders_temview  (cost=7286.00..13286.00 rows=300000 width=4) (actual time=0.005..110.899 rows=300000 loops=1)
   CTE orders_temview
     ->  Seq Scan on orders  (cost=0.00..7286.00 rows=300000 width=82) (actual time=0.004..33.440 rows=300000 loops=1)
 Total runtime: 131.639 ms
(4 rows)

total time: 132  ms
```

不用视图直接查询

```
EXPLAIN ANALYZE
SELECT o_orderkey
FROM orders;
```

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on orders  (cost=0.00..7286.00 rows=300000 width=4) (actual time=0.004..40.092 rows=300000 loops=1)
 Total runtime: 55.404 ms
(2 rows)

total time: 56  ms
```

3. 分析

3个执行计划对比可发现，针对以上的3个查询，通过建立视图来查询和不用视图直接查询，实际的执行计划是相同的，执行时间也相似。但是通过with临时视图查询，执行计划会有所不同，所花费的执行时间也更长。

原因在于，第一个查询中，提前建立好了orders_view视图，对orders_view视图的直接查询，在执行中，转变成了对orders的顺序扫描，而在第三个查询中，对orders的直接查询，也是对orders的顺序扫描，则两个查询中的执行过程是一样的（都只有顺序扫描这一层），执行时间自然也相似。在第二个查询中，执行计划一共有2层，第一层是对orders进行顺序扫描，并将扫描出的数据存在临时创建的通用表表达式orders_temview中(CTE它是一种临时结果集)，第二层再通过对orders_temview进行扫描来得到查询结果。由于多了第二层，执行时间也更长了。

2.3 优化 SQL 语句

实验要求

结合文档“数据库物理设计及查询优化”中相关内容，通过对以下各种不同情况下查询语句的执行情况的对比分析，巩固加深查询优化的理解，并进行书写优化 SQL 语句的初步训练，提高编写高效 SQL 语句进行数据查询的能力。

2.3.1 复合索引左前缀

实验要求

在做这部分实验时，要求使用 lineitem 表,按照下述步骤完成实验内容。对访问相同的两张表且查询需求完全一样的 2 条 select 语句实现两个要求（在 lineitem 上创建包含至少三个索引的组合索引）：

（1）比较有最左前缀索引和无索引的 2 条 select 语句的执行结果和执行计划；

（2）比较组合索引中其他索引（不包含最左前缀索引）和无索引的 2 条 select 语句的执行计划。

分别判断 2 条语句的执行结果是否一样，对比执行效果和执行速度，解释执行时间出现差异的原因

说明：在使用复合索引时要注意最左前缀的原则，以防索引失效

实验过程

1. 创建 lineitem 的备份表 lineitem_New（没有主键，没有索引，没有约束）；

```
CREATE TABLE lineitem_new AS TABLE lineitem;
```

```
INSERT 0 1199969
total time: 2982  ms
```

2. 在 lineitem_new 创建组合索引（至少包含三个索引）。

索引属性建议：可以在 l_quantity、l_extendedprice、l_tax 上设计组合索引。

```
CREATE index lineitem_index
ON lineitem_new(l_quantity, l_tax, l_extendedprice);
```

```
CREATE INDEX
```

3. 比较有最左前缀索引和无索引的 2 条 select 语句的执行结果和执行计划

编写 select 语句，使用最左前缀索引访问 lineitem_new；编写 select 语句，不使用索引方式，访问 lineitem，执行 2 条 select 语句，判断执行的结果是否一致，并观察各自执行效果、查询执行计划、时间对比。

不使用索引，查询 l_quantity=24 的全部数据

```
explain analyze
select *
from lineitem
where l_quantity=24;
```

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on lineitem  (cost=0.00..39915.61 rows=24319 width=129) (actual time=0.021..297.971 rows=24333 loops=1)
   Filter: (l_quantity = 24::numeric)
   Rows Removed by Filter: 1175636
 Total runtime: 300.017 ms
(4 rows)
```

使用最左前缀索引查询

```
explain analyze
select *
from lineitem_new
where l_quantity=24;
```

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on lineitem_new  (cost=667.41..26600.29 rows=24119 width=129) (actual time=15.083..2128.177 rows=24333 loops=1)
   Recheck Cond: (l_quantity = 24::numeric)
   ->  Bitmap Index Scan on lineitem_index  (cost=0.00..661.38 rows=24119 width=0) (actual time=12.694..12.694 rows=24333 loops=1)
         Index Cond: (l_quantity = 24::numeric)
 Total runtime: 2138.431 ms
(5 rows)
```

4. 在比较组合索引中其他索引（不包含最左前缀索引）和无索引的 2 条 select 语句的执行计划

编写 select 语句，使用不含最左前缀的索引访问 lineitem_new；编写 select 语句，不使用索引方式，访问 lineitem，执行 2 条 select 语句，判断执行的结果是否一致，并观察各自执行效果、查询执行计划、时间对比。

不使用索引，查询 l_tax=0.02 的全部数据

```
explain analyze
select *
from lineitem
where l_tax=0.02;
```

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on lineitem  (cost=0.00..39915.61 rows=132437 width=129) (actual time=0.271..514.618 rows=133336 loops=1)
   Filter: (l_tax = .02)
   Rows Removed by Filter: 1066633
 Total runtime: 522.636 ms
(4 rows)
```

使用不含最左前缀的索引查询

```
explain analyze
select *
from lineitem_new
where l_tax=0.02;
```

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on lineitem_new  (cost=0.00..39915.61 rows=135317 width=129) (actual time=0.046..304.731 rows=133336 loops=1)
   Filter: (l_tax = .02)
   Rows Removed by Filter: 1066633
 Total runtime: 312.183 ms
(4 rows)
```

分析：查询结果一致，但查询时间有很大差距，不使用索引的查询时间>使用不含最左前缀的索引的查询时间>使用最左前缀索引的查询时间。

2.3.2 多表连接操作，在连接属性上建立索引

实验要求

在做这部分实验时，要求使用 lineitem、part 和 supplier 表,按照下述步骤完成实验内容。对访问相同的两张表且查询需求完全一样的 2 条 select 语句实现两个要求之一（在连接属性上建立索引）：

（1）在 lineitem_new 表的 l_suppkey 上设计索引，列举在 supplier 中找到此供应商的手机号。

（2）在 lineitem_new 表的 l_partkey 上设计索引，列举在 part 中找到此零件的零件名称。

每个要求中，分别判断 2 条语句的执行结果是否一样，对比执行效果和执行速度，解释执行时间出现差异的原因。

实验过程

1．创建 lineitem 的备份表 lineitem_new；

2．在 lineitem_new 创建索引；

删除上个实验的索引，再在 l_suppkey 和 l_partkey 上创建 2 个索引

```
drop index lineitem_index;
create index lineitem_index1 on lineitem_new(l_suppkey);
create index lineitem_index2 on lineitem_new(l_partkey);
```

3．比较有索引和无索引的 2 条两表 select 语句的执行结果和执行计划。

编写 select 语句，使用索引访问 lineitem_new，part 和 supplier；编写 select 语句，不使用索引方式，访问 lineitem，part 和 supplier，执行 2 条 select 语句，判断执行的结果是否一致，并观察各自执行效果、查询执行计划、时间对比。

不使用 l_suppkey 索引，l_suppkey=10 ,列举在 orders 中找到的此订单的订单总价

```
EXPLAIN ANALYZE
select DISTINCT l_suppkey,s_phone
from lineitem, supplier
where l_suppkey=10 AND l_suppkey=supplier.s_suppkey;
```

```
                                   QUERY PLAN
------------------------------------------------------------------------------------------
 HashAggregate  (cost=39932.61..39932.62 rows=1 width=20) (actual time=563.874..563.875 rows=1 loops=1)
   Group By Key: lineitem.l_suppkey, supplier.s_phone
   ->  Nested Loop  (cost=0.00..39929.70 rows=582 width=20) (actual time=6.913..563.272 rows=591 loops=1)
         ->  Index Scan using supplier_index on supplier  (cost=0.00..8.27 rows=1 width=20) (actual time=4.219..4.233 rows=1 loops=1)
               Index Cond: (s_suppkey = 10)
         ->  Seq Scan on lineitem  (cost=0.00..39915.61 rows=582 width=4) (actual time=2.688..558.814 rows=591 loops=1)
               Filter: (l_suppkey = 10)
               Rows Removed by Filter: 1199378
 Total runtime: 564.167 ms
(9 rows)
```

使用 l_suppkey 索引查询

```
EXPLAIN ANALYZE
select DISTINCT l_suppkey,s_phone
from lineitem_new, supplier
where l_suppkey=10 AND l_suppkey=supplier.s_suppkey;
```

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
HashAggregate  (cost=2078.43..2078.44 rows=1 width=20) (actual time=27.040..27.040 rows=1 loops=1)
  Group By Key: lineitem_new.l_suppkey, supplier.s_phone
  ->  Nested Loop  (cost=12.89..2075.52 rows=582 width=20) (actual time=1.256..26.714 rows=591 loops=1)
        ->  Index Scan using supplier_index on supplier  (cost=0.00..8.27 rows=1 width=20) (actual time=0.006..0.007 rows=1 loops=1)
              Index Cond: (s_suppkey = 10)
        ->  Bitmap Heap Scan on lineitem_new  (cost=12.89..2061.43 rows=582 width=4) (actual time=1.247..26.535 rows=591 loops=1)
              Recheck Cond: (l_suppkey = 10)
              ->  Bitmap Index Scan on lineitem_index1  (cost=0.00..12.75 rows=582 width=0) (actual time=1.174..1.174 rows=591 loops=1)
                    Index Cond: (l_suppkey = 10)
Total runtime: 27.216 ms
(10 rows)
```

不使用 l_partkey 索引，l_partkey= 928,列举在 part 中找到的此邻小区对应的主小区

```
EXPLAIN ANALYZE
select DISTINCT l_partkey,p_name
from lineitem, part
where l_partkey= 928 AND l_partkey=part.p_partkey;
```

```
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------
HashAggregate  (cost=39924.35..39924.36 rows=1 width=42) (actual time=512.591..512.591 rows=1 loops=1)
  Group By Key: lineitem.l_partkey, part.p_name
  ->  Nested Loop  (cost=0.00..39924.19 rows=31 width=42) (actual time=70.812..512.503 rows=24 loops=1)
        ->  Index Scan using part_pkey on part  (cost=0.00..8.27 rows=1 width=42) (actual time=3.674..3.676 rows=1 loops=1)
              Index Cond: (p_partkey = 928)
        ->  Seq Scan on lineitem  (cost=0.00..39915.61 rows=31 width=4) (actual time=67.134..508.759 rows=24 loops=1)
              Filter: (l_partkey = 928)
              Rows Removed by Filter: 1199945
Total runtime: 512.668 ms
(9 rows)
```

使用 l_partkey 索引查询

```
EXPLAIN ANALYZE
select DISTINCT l_partkey,p_name
from lineitem_new, part
where l_partkey= 928 AND l_partkey=part.p_partkey;
```

```
QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
HashAggregate  (cost=130.59..130.60 rows=1 width=42) (actual time=2.920..2.921 rows=1 loops=1)
  Group By Key: lineitem_new.l_partkey, part.p_name
  ->  Nested Loop  (cost=4.61..130.44 rows=30 width=42) (actual time=0.917..2.891 rows=24 loops=1)
        ->  Index Scan using part_pkey on part  (cost=0.00..8.27 rows=1 width=42) (actual time=0.005..0.006 rows=1 loops=1)
              Index Cond: (p_partkey = 928)
        ->  Bitmap Heap Scan on lineitem_new  (cost=4.61..121.87 rows=30 width=4) (actual time=0.909..2.876 rows=24 loops=1)
              Recheck Cond: (l_partkey = 928)
              ->  Bitmap Index Scan on lineitem_index2  (cost=0.00..4.61 rows=30 width=0) (actual time=0.659..0.659 rows=24 loops=1)
                    Index Cond: (l_partkey = 928)
Total runtime: 2.983 ms
(10 rows)
```

分析：查询结果一致，但不使用索引的查询时间>使用索引的查询时间，可知，在两表连接上，建立恰当的索引，可以节省查询时间。

2.3.3 索引对小表查询的作用

实验要求

在做此部分实验时要求使用元组数目少、表占用空间少的 supplier 表，按照下述步骤完成实验内容。在 s_suppkey 上创建索引，执行一个查询，查看执行计划观察是否用到索引，如果没有则强制使用并与不强制使用做比较。

实验过程

1.创建supplier的不带主键索引的备份表supplier_new；

```
CREATE TABLE supplier_new AS TABLE supplier;
```

```
INSERT 0 2000
```

2.在supplier上新建查询，观察是否用到索引；

```
explain
select s_suppkey
from supplier;
```

并没有用到索引

```
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on supplier  (cost=0.00..62.00 rows=2000 width=4)
(1 row)
```

3.编写select语句，无索引访问supplier_new，强制使用索引访问supplier，执行2条select语句，判断执行的结果是否一致，并观察各自执行效果、查询执行计划、时间对比。

无索引查询supplier_new

```
explain
select s_suppkey
from supplier_new;
```

```
                          QUERY PLAN
-------------------------------------------------------------
 Seq Scan on supplier_new  (cost=0.00..62.00 rows=2000 width=4)
(1 row)
```

强制使用索引查询 supplier

先禁止顺序扫描

```
set enable_seqscan=off;
或 set enable_seqscan=false;
```

```
SET
```

再查询 supplier

```
explain
select s_suppkey
from supplier;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Bitmap Heap Scan on supplier  (cost=42.75..104.75 rows=2000 width=4)
   ->  Bitmap Index Scan on supplier_index  (cost=0.00..42.25 rows=2000 width=0)
(2 rows)
```

分析：强制使用索引开销会增大很多，对于较小的表不建立索引反而查询会更快。

注意：在完成实验后，记得将更改的设置改回去(默认情况下，退出数据库后enable_seqscan也会自动恢复回on的状态）。

```
set enable_seqscan=on;
```

2.3.4 查询条件中函数对索引的影响

实验要求

针对同一张表，编写 2 条查询需求完全一样的等价的 select 语句，一条语句在 where 子句查询条件中使用了函数，另外一条语句则没有使用函数。

观察这 2 条等价的 SQL 语句在查询执行计划、索引使用和实现效率方面的差异，掌握如何避免函数导致查询条件索引失效的方法，编写优化的 SQL 语句。

实现以下两个要求之一：

(1) 在订单明细表 lineitem 中查询与价格为 16473.51，折扣为 0.04 的订单数量差在 20 以内订单的订单 key。

(2) 在订单明细表 lineitem 中查询与价格为 79238.70，折扣为 0.02 的订单数量差在 20 以内订单的订单 key。

实验过程

1.创建备份表并在备份表上创建索引；

先删除上面实验建立的索引

```
drop index lineitem_index1;
drop index lineitem_index2;
```

再建立索引

```
create index lineitem_index1 on lineitem_new(l_discount);
create index lineitem_index2 on lineitem_new(l_quantity);
```

2.编写select语句，完成实验要求中的查询，执行2条等价的查询语句，并查看执行计划，比较两个语句的执行结果和执行速度，观察函数对索引的影响。

以要求一为例，

查询条件中使用函数进行查询

```
explain
select distinct B.l_orderkey
FROM lineitem_new as A, lineitem_new as B
where A.l_extendedprice = 16473.51
and A.l_discount =0.04
and ABS(A.l_quantity-B.l_quantity)<20;
```

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=87479.44..89906.39 rows=242695 width=4)
   Group By Key: b.l_orderkey
   ->  Nested Loop  (cost=2016.36..86479.46 rows=399990 width=4)
         Join Filter: (abs((a.l_quantity - b.l_quantity)) < 20::numeric)
         ->  Bitmap Heap Scan on lineitem_new a  (cost=2016.36..28564.31 rows=1 width=5)
               Recheck Cond: (l_discount = .04)
               Filter: (l_extendedprice = 16473.51)
               ->  Bitmap Index Scan on lineitem_index1  (cost=0.00..2016.36 rows=108797 width=0)
                     Index Cond: (l_discount = .04)
         ->  Seq Scan on lineitem_new b  (cost=0.00..36915.69 rows=1199969 width=9)
(10 rows)
```

查询条件中不使用函数进行查询

```
explain
select distinct B.l_orderkey
FROM lineitem_new as A, lineitem_new as B
where A.l_extendedprice = 16473.51
and A.l_discount =0.04
and B.l_quantity >A.l_quantity -20
and B.l_quantity <A.l_quantity +20;
```

```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=60648.56..61981.86 rows=133330 width=4)
   Group By Key: b.l_orderkey
   ->  Nested Loop  (cost=4851.38..60315.23 rows=133330 width=4)
         ->  Bitmap Heap Scan on lineitem_new a  (cost=2016.36..28564.31 rows=1 width=5)
               Recheck Cond: (l_discount = .04)
               Filter: (l_extendedprice = 16473.51)
               ->  Bitmap Index Scan on lineitem_index1  (cost=0.00..2016.36 rows=108797 width=0)
                     Index Cond: (l_discount = .04)
         ->  Bitmap Heap Scan on lineitem_new b  (cost=2835.02..30417.62 rows=133330 width=9)
               Recheck Cond: ((l_quantity > (a.l_quantity - 20::numeric)) AND (l_quantity < (a.l_quantity + 20::numeric)))
               ->  Bitmap Index Scan on lineitem_index2  (cost=0.00..2801.69 rows=133330 width=0)
                     Index Cond: ((l_quantity > (a.l_quantity - 20::numeric)) AND (l_quantity < (a.l_quantity + 20::numeric)))
(12 rows)
```

分析：第一个执行计划使用了 l_discount 上的索引，而没有使用 l_quantity 上的索引，从而查询效率更低，第二个执行计划两个索引都使用了，从而查询效率更高。可知，很多情况下，当索引上存在函数计算时，索引就不起作用了，可转换为等价的没有函数计算的实现方式，来提高查询效率。

2.3.5 多表嵌入式 SQL 查询

实验要求

对访问相同的两张表且查询需求完全一样的 2 条 select 语句，一条使用嵌套查询，一条使用连接查询，实现两个要求之一：

(1) 根据零部件表和零部件供应表，查询零件供应商 key 为 1002 的供应商供应零件的名称。

(2) 根据订单表和客户表，查询客户 key 为 10 的订单的订单总价。

在每个要求中，分别判断 2 条语句的执行结果是否一样，对比执行效果和执行速度，解释执行时间出现差异的原因。

实验过程

以要求一为例，

1.嵌套查询

```
explain analyze
select A.p_name
from part as A
where A.p_partkey IN(
select B.ps_partkey
from partsupp as B
where B.ps_suppkey=1002
);
```

```
                                                     QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop Semi Join  (cost=0.00..2622.48 rows=78 width=38) (actual time=4.344..133.149 rows=80 loops=1)
   ->  Seq Scan on part a  (cost=0.00..1252.00 rows=40000 width=42) (actual time=0.706..26.975 rows=40000 loops=1)
   ->  Index Only Scan using partsupp_pkey on partsupp b  (cost=0.00..0.68 rows=21 width=4) (actual time=91.856..91.856 rows=80 loops=40000)
         Index Cond: ((ps_partkey = a.p_partkey) AND (ps_suppkey = 1002))
         Heap Fetches: 80
 Total runtime: 133.322 ms
(6 rows)
```

2.连接查询

```
explain analyze
select A.p_name
from part as A,partsupp as B
where B.ps_suppkey=1002
AND A.p_partkey =B.ps_partkey;
```

```
                                                     QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..3666.64 rows=78 width=38) (actual time=4.154..9.787 rows=80 loops=1)
   ->  Index Only Scan using partsupp_pkey on partsupp b  (cost=0.00..3112.45 rows=78 width=4) (actual time=0.014..2.657 rows=80 loops=1)
         Index Cond: (ps_suppkey = 1002)
         Heap Fetches: 80
   ->  Index Scan using part_pkey on part a  (cost=0.00..7.09 rows=1 width=42) (actual time=7.042..7.068 rows=80 loops=80)
         Index Cond: (p_partkey = b.ps_partkey)
 Total runtime: 9.854 ms
(7 rows)
```

分析：一般在使用多表查询的时候应该避免使用嵌套查询，但在实际应用中有时会出现两种情况性能相同甚至是嵌套查询性能更优的情况。

2.3.6 where 查询条件中复合查询条件 OR 对索引的影响

实验要求

在做这部分实验时，要求使用lineitem表,按照下述步骤完成实验内容。在lineitem_new上创建一个索引，查询条件中为“A OR B”，A为创建的索引条件，B为非索引条件，提交select语句并查看执行计划中是否还能用到索引。再将含有“A OR B”查询条件的语句改为等价的两条select语句的union，对比这两种实现方式的执行效率。

实验步骤

1.创建lineitem的备份表lineitem_new，并在lineitem_new上创建索引；

2.编写select语句，where查询条件为“A OR B”，A为定义在索引属性上的查询条件，B为定义在非索引属性上的查询条件，执行语句，查看执行计划是否使用了索引；

```
Select ...
From ...
Where    ... and (A OR B);
```

3.将含有“A OR B”查询条件的select语句，转换为等价的2条select语句的union：

```
Select ...
From ...
Where    ... and A
Union
Select ...
From ...
Where    ... and B;
```

分析对比这两种查询实现方式的执行效率。

示例

删除上面实验的索引，并创建索引

```
drop index lineitem_index1;
drop index lineitem_index2;
create index lineitem_index on lineitem_new(l_quantity);
```

含有“A OR B”查询条件的 select 语句

```
explain
select *
from lineitem_new
where l_tax=0.06 AND (l_quantity=36 OR l_discount=0.09);
```

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on lineitem_new  (cost=0.00..45915.46 rows=14148 width=129)
   Filter: ((l_tax = .06) AND ((l_quantity = 36::numeric) OR (l_discount = .09)))
(2 rows)
```

等价的 2 条 select 语句的 union

```
explain
(select *
from lineitem_new
where l_tax=0.06 AND l_quantity=36)
UNION
(select *
from lineitem_new
where l_tax=0.06 AND l_discount=0.09);
```

```
                                   QUERY PLAN
--------------------------------------------------------------------------------
 HashAggregate  (cost=70100.58..70244.44 rows=14386 width=129)
   Group By Key: public.lineitem_new.l_orderkey, public.lineitem_new.l_partkey, public.lineitem_new.l_suppkey, public.lineitem_new.l_linenumber, public.lineitem_new.l_quantity, public.lineitem_new.l_extendedprice, public.lineitem_new.l_discount, public.lineitem_new.l_tax, public.lineitem_new.l_returnflag, public.lineitem_new.l_linestatus, public.lineitem_new.l_shipdate, public.lineitem_new.l_commitdate, public.lineitem_new.l_receiptdate, public.lineitem_new.l_shipinstruct, public.lineitem_new.l_shipmode, public.lineitem_new.l_comment
   ->  Append  (cost=450.83..69525.14 rows=14386 width=129)
         ->  Bitmap Heap Scan on lineitem_new  (cost=450.83..26465.74 rows=2622 width=129)
               Recheck Cond: (l_quantity = 36::numeric)
               Filter: (l_tax = .06)
               ->  Bitmap Index Scan on lineitem_index  (cost=0.00..450.17 rows=24239 width=0)
                     Index Cond: (l_quantity = 36::numeric)
         ->  Seq Scan on lineitem_new  (cost=0.00..42915.54 rows=11764 width=129)
               Filter: ((l_tax = .06) AND (l_discount = .09))
(10 rows)
```

分析：含有union的语句执行效率更低，从执行计划来看，第一个语句因为where条件中引入了OR而使得索引不再起作用，但是后一个查询步骤繁多，虽然使用索引，但是却还是进行了全表扫描，效率反而更低。

2.3.7 聚集运算中的索引设计

实验要求

选定作为聚集运算查询对象的关系表，对group by操作的分组属性，建立聚集/非聚集索引，对聚集运算（如count、sum、avg）的属性，建立非聚集索引。分析比较等价的有索引聚集运算、无索引聚集运算查询在查询执行计划、执行速度方面的区别。完成以下两个要求之一：

（1）在订单明细表lineitem，计算有同一供应商订单的平均价格并按供应商分组（注意索引的建立）；

（2）在订单明细表lineitem，计算有同一供应商订单的平均数量并按供应商分组（注意索引的建立）。

实验过程

1.创建备份表，并根据查询要求在备份表上创建索引；

2.根据查询要求，在原表和备份表上编写并相同的select语句；

3.查看执行计划，分析比较等价的有索引聚集运算、无索引聚集运算查询在查询执行计划、执行速度方面的区别。

示例

以要求一为例，

删除上面实验创建的索引，并分别在l_suppkey和l_extendedprice创建索引；

```
drop index lineitem_index;
create index lineitem_index1 on lineitem_new(l_extendedprice);
create index lineitem_index2 on lineitem_new(l_suppkey);
```

查询同一l_suppkey的平均l_extendedprice并按l_suppkey分组；

无索引聚集运算

```
explain
select l_suppkey,avg(l_extendedprice) as avg_extendedprice
from lineitem
group by l_suppkey;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=42915.54..42940.54 rows=2000 width=44)
   Group By Key: l_suppkey
   ->  Seq Scan on lineitem  (cost=0.00..36915.69 rows=1199969 width=12)
(3 rows)
```

有索引聚集运算

```
explain
select l_suppkey,avg(l_extendedprice) as avg_extendedprice
from lineitem_new
group by l_suppkey;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
 HashAggregate  (cost=42915.54..42940.54 rows=2000 width=44)
   Group By Key: l_suppkey
   ->  Seq Scan on lineitem_new  (cost=0.00..36915.69 rows=1199969 width=12)
(3 rows)
```

强制使用索引后，聚集运算

```
set enable_seqscan=off;
explain
select l_suppkey,avg(l_extendedprice) as avg_extendedprice
from lineitem_new
group by l_suppkey;
```

```
                                QUERY PLAN
-------------------------------------------------------------------------
GroupAggregate  (cost=0.00..2073781.39 rows=2000 width=44)
  Group By Key: l_suppkey
  ->  Index Scan using lineitem_index2 on lineitem_new  (cost=0.00..2067756.55 rows=1199969 width=12)
(3 rows)
```

分析：上述例子，有索引的表在查询时没有使用索引（优化器在检测到使用索引查询效率反而更低时，会自动不使用索引），第一个查询和第二个查询的执行计划基本相同，查询效率也基本相同。在有索引的表上强制使用索引后，第三个查询的执行计划采取了索引扫描，查询效率更低，查询速度更慢。注意：在完成实验后，记得将更改的设置改回去(默认情况下，退出数据库后enable_seqscan也会自动恢复回on的状态）。

```
set enable_seqscan=on;
```

2.3.8 Select 子句中有无 distinct 的区别

实验要求

在做这部分实验时，按照下述步骤完成实验内容。对访问相同的两张表且查询需求完全一样的 2 条 select 语句，一条使用 distinct，一条不使用 distinct，实现两个要求之一：

（1）根据订单表，查询订单状态为‘O’且订单优先级为‘ 4-NOT SPECIFIED’的订单 key。

（2）根据供应商表，查询供应商国家 key 为 1 且供应商账户余额小于 1000 的供应商 key。

在每个要求中，分别判断 2 条语句的执行结果是否一样，对比执行效果和执行速度，解释执行时间出现差异的原因。

示例

以要求一为例，

无distinct查询

```
explain analyze
select o_orderkey
from orders
where o_orderstatus='O'
and o_orderpriority='4-NOT SPECIFIED';
```

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on orders  (cost=0.00..8786.00 rows=29590 width=4) (actual time=0.600..146.558 rows=29224 loops=1)
   Filter: ((o_orderstatus = 'O'::bpchar) AND (o_orderpriority = '4-NOT SPECIFIED'::bpchar))
   Rows Removed by Filter: 270776
 Total runtime: 148.361 ms
(4 rows)
```

有distinct查询

```
explain analyze
select distinct o_orderkey
from orders
where o_orderstatus='O'
and o_orderpriority='4-NOT SPECIFIED';
```

```
                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=8859.98..9155.88 rows=29590 width=4) (actual time=70.249..73.308 rows=29224 loops=1)
   Group By Key: o_orderkey
   ->  Seq Scan on orders  (cost=0.00..8786.00 rows=29590 width=4) (actual time=0.067..63.239 rows=29224 loops=1)
         Filter: ((o_orderstatus = 'O'::bpchar) AND (o_orderpriority = '4-NOT SPECIFIED'::bpchar))
         Rows Removed by Filter: 270776
 Total runtime: 75.260 ms
(6 rows)
```

分析：以上两个查询的查询结果是一样的（因为o_orderkey是主键，本身就没有重复的），但是有distinct的查询的执行计划，比无distinct的查询的执行计划，多了按o_orderkey进行散列聚集的操作，从而使得有distinct的查询效率要比无distinct的查询效率更低，使用distinct有时反而会增大开销。

2.3.9 union、union all 的区别

实验要求

在做这部分实验时，按照下述步骤完成实验内容。对访问相同的两张表且查询需求完全一样的 2 条 select 语句，一条使用 union，一条使用 union all，实现两个要求之一：

（1）找出所有零部件表，零部件供应表的零件 key。

（2）找出客户表中的客户 key 和订单表的客户 key 并总和到一起。

在每个要求中，分别判断 2 条语句的执行结果是否一样，对比执行效果和执行速度，解释执行时间出现差异的原因。

示例

以要求一为例。

使用union进行查询

```
explain analyze
select p_partkey from part
union
select ps_partkey from partsupp;
```

```
                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=8849.00..10849.00 rows=200000 width=4) (actual time=78.856..83.826 rows=40000 loops=1)
   Group By Key: part.p_partkey
   ->  Append  (cost=0.00..8349.00 rows=200000 width=4) (actual time=0.006..47.116 rows=200000 loops=1)
         ->  Seq Scan on part  (cost=0.00..1252.00 rows=40000 width=4) (actual time=0.006..6.501 rows=40000 loops=1)
         ->  Seq Scan on partsupp  (cost=0.00..5097.00 rows=160000 width=4) (actual time=0.003..24.444 rows=160000 loops=1)
 Total runtime: 86.477 ms
(6 rows)
```

使用union all进行查询

```
explain analyze
select p_partkey from part
union all
select ps_partkey from partsupp;
```

```
                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Result  (cost=0.00..6349.00 rows=200000 width=4) (actual time=0.006..62.890 rows=200000 loops=1)
   ->  Append  (cost=0.00..6349.00 rows=200000 width=4) (actual time=0.006..45.724 rows=200000 loops=1)
         ->  Seq Scan on part  (cost=0.00..1252.00 rows=40000 width=4) (actual time=0.005..5.865 rows=40000 loops=1)
         ->  Seq Scan on partsupp  (cost=0.00..5097.00 rows=160000 width=4) (actual time=0.005..23.782 rows=160000 loops=1)
 Total runtime: 72.306 ms
(5 rows)
```

分析：两次查询的查询结果不一样，第一次查询结果的数据要远远少于第二次查询结果的数据，第二次查询结果中有大量的重复数据，但是第二次查询效率要高于第一次的查询效率，union all的性能要优于union。

2.3.10 from 中存在多余的关系表，即查询非最简化

实验要求

在做这部分实验时，按照下述步骤完成实验内容。对访问相同的两张表且查询需求完全一样的 2 条等价 select 语句，一条在 from 后面增加一个多余的关系表（但在 where 子句中增加连接条件），另一条 from 后面最简化。

可自行选择查询语句，要求至少两表连接查询。分别判断 2 条语句的执行效果和执行速度，解释执行时间出现差异的原因。

示例

没有多余关系表

```
explain analyze
select distinct orders.o_custkey
from lineitem,orders
where lineitem.l_orderkey=orders.o_orderkey;
```

```
                                                              QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=67451.19..67637.66 rows=18647 width=4) (actual time=1167.030..1169.327 rows=19999 loops=1)
   Group By Key: orders.o_custkey
   ->  Hash Join  (cost=11036.00..64451.26 rows=1199969 width=4) (actual time=101.742..962.625 rows=1199969 loops=1)
         Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
         ->  Seq Scan on lineitem  (cost=0.00..36915.69 rows=1199969 width=4) (actual time=3.392..565.529 rows=1199969 loops=1)
         ->  Hash  (cost=7286.00..7286.00 rows=300000 width=8) (actual time=97.416..97.416 rows=300000 loops=1)
                Buckets: 524288  Batches: 1  Memory Usage: 11719kB
               ->  Seq Scan on orders  (cost=0.00..7286.00 rows=300000 width=8) (actual time=0.032..46.362 rows=300000 loops=1)
 Total runtime: 1184.905 ms
(9 rows)
```

增加了一个多余的关系表tbCell和对应的连接条件

```
explain analyze
select distinct orders.o_custkey
from lineitem,orders,part
where lineitem.l_orderkey=orders.o_orderkey and lineitem.l_partkey=part.p_partkey;
```

```
                                                              QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=85702.76..85889.23 rows=18647 width=4) (actual time=1456.390..1458.458 rows=19999 loops=1)
   Group By Key: orders.o_custkey
   ->  Hash Join  (cost=12788.00..82702.84 rows=1199969 width=4) (actual time=118.903..1250.760 rows=1199969 loops=1)
         Hash Cond: (lineitem.l_partkey = part.p_partkey)
         ->  Hash Join  (cost=11036.00..64451.26 rows=1199969 width=8) (actual time=105.822..932.917 rows=1199969 loops=1)
               Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
               ->  Seq Scan on lineitem  (cost=0.00..36915.69 rows=1199969 width=8) (actual time=0.348..515.267 rows=1199969 loops=1)
               ->  Hash  (cost=7286.00..7286.00 rows=300000 width=8) (actual time=104.587..104.587 rows=300000 loops=1)
                      Buckets: 524288  Batches: 1  Memory Usage: 11719kB
                     ->  Seq Scan on orders  (cost=0.00..7286.00 rows=300000 width=8) (actual time=0.005..46.141 rows=300000 loops=1)
         ->  Hash  (cost=1252.00..1252.00 rows=40000 width=4) (actual time=12.937..12.937 rows=40000 loops=1)
                Buckets: 65536  Batches: 1  Memory Usage: 1407kB
               ->  Seq Scan on part  (cost=0.00..1252.00 rows=40000 width=4) (actual time=0.005..7.233 rows=40000 loops=1)
 Total runtime: 1460.069 ms
(14 rows)
```

分析：两次查询的结果完全一致，但是第一次查询的查询效率远远高于第二次查询，增加多余的表会大大的增加开销。

3 附录

3.1 openGauss SQL 执行计划概述

SQL 执行计划是一个节点树，显示 openGauss 执行一条 SQL 语句时执行的详细步骤。每一个步骤为一个数据库运算符。使用 EXPLAIN 命令可以查看优化器为每个查询生成的具体执行计划。EXPLAIN 给每个执行节点都输出一行，显示基本的节点类型和优化器为执行这个节点预计的开销值。如下图所示：

```
postgres=# explain select * from t1, t2 where t1.c1=t2.c2;
                           QUERY PLAN
-----------------------------------------------------------------
 Hash Join  (cost=58.35..355.67 rows=23091 width=16)
   Hash Cond: (t1.c1 = t2.c2)
   ->  Seq Scan on t1  (cost=0.00..31.49 rows=2149 width=8)
   ->  Hash  (cost=31.49..31.49 rows=2149 width=8)
         ->  Seq Scan on t2  (cost=0.00..31.49 rows=2149 width=8)
(5 rows)
```

说明：

- 最底层节点是表扫描节点，它扫描表并返回原始数据行。不同的表访问模式有不同的扫描节点类型：顺序扫描、索引扫描等。最底层节点的扫描对象也可能是非表行数据（不是直接从表中读取的数据），如 VALUES 子句和返回行集的函数，它们有自己的扫描节点类型。
- 如果查询需要连接、聚集、排序、或者对原始行做其它操作，那么就会在扫描节点之上添加其它节点。 并且这些操作通常都有多种方法，因此在这些位置也有可能出现不同的执行节点类型。
- 第一行(最上层节点)是执行计划总执行开销的预计。这个数值就是优化器试图最小化的数值。

执行计划显示信息

除了设置不同的执行计划显示格式外，还可以通过不同的 EXPLAIN 用法，显示不同详细程度的执行计划信息。常见有如下几种：

EXPLAIN statement：只生成执行计划，不实际执行。其中 statement 代表 SQL 语句。

EXPLAIN ANALYZE statement：生成执行计划，进行执行，并显示执行的概要信息。显示中加入了实际的运行时间统计，包括在每个规划节点内部花掉的总时间(以毫秒计)和它实际返回的行数。

EXPLAIN PERFORMANCE statement：生成执行计划，进行执行，并显示执行期间的全部信息。

说明：

- 为了测量运行时在执行计划中每个节点的开销，EXPLAIN ANALYZE 或 EXPLAIN PERFORMANCE 会在当前查询执行上增加性能分析的开销。在一个查询上运行 EXPLAIN ANALYZE 或 EXPLAIN PERFORMANCE 有时会比普通查询明显的花费更多的时间。超支的数量依赖于查询的本质和使用的平台。
- 当定位 SQL 运行慢问题时，如果 SQL 长时间运行未结束，建议通过 EXPLAIN 命令查看执行计划，进行初步定位。如果 SQL 可以运行出来，则推荐使用 EXPLAIN ANALYZE 或 EXPLAIN PERFORMANCE 查看执行计划及其实际的运行信息，以便更精准地定位问题原因。

3.2 EXPLAIN

功能描述

显示 SQL 语句的执行计划。执行计划将显示 SQL 语句所引用的表会采用什么样的扫描方式，如：简单的顺序扫描、索引扫描等。如果引用了多个表，执行计划还会显示用到的 JOIN 算法。

执行计划的最关键的部分是语句的预计执行开销，这是计划生成器估算执行该语句将花费多长的时间。

若指定了 ANALYZE 选项，则该语句会被执行，然后根据实际的运行结果显示统计数据，包括每个计划节点内时间总开销（毫秒为单位）和实际返回的总行数。这对于判断计划生成器的估计是否接近现实非常有用。

注意：

在指定 ANALYZE 选项时，语句会被执行。如果用户想使用 EXPLAIN 分析 INSERT，UPDATE，DELETE，CREATE TABLE AS 或 EXECUTE 语句，而不想改动数据（执行这些语句会影响数据），请使用这种方法（关于事务的详细信息请见实验指导书 08）：

```
START TRANSACTION;
EXPLAIN ANALYZE ...;
ROLLBACK;
```

语法格式

共有 2 种格式：

- 显示 SQL 语句的执行计划，支持多种选项，对选项顺序无要求。

```
EXPLAIN [ (    option    [, ...] )    ] statement;
```

其中选项 option 子句的语法为。

```
ANALYZE [ boolean ] |
      ANALYSE [ boolean ] |
      VERBOSE [ boolean ] |
      COSTS [ boolean ] |
      CPU [ boolean ] |
      DETAIL [ boolean ] |
      NODES [ boolean ] |
      NUM_NODES [ boolean ] |
      BUFFERS [ boolean ] |
      TIMING [ boolean ] |
      PLAN [ boolean ] |
```

FORMAT { TEXT | XML | JSON | YAML }
- 显示 SQL 语句的执行计划，且要按顺序给出选项。

EXPLAIN { [{ ANALYZE | ANALYSE }] [VERBOSE] | PERFORMANCE } statement;

参数说明

statement
指定要分析的 SQL 语句。

ANALYZE boolean | ANALYSE boolean
显示实际运行时间和其他统计数据。
取值范围：
TRUE（缺省值）：显示实际运行时间和其他统计数据。
FALSE：不显示。

VERBOSE boolean
显示有关计划的额外信息。
取值范围：
TRUE（缺省值）：显示额外信息。
FALSE：不显示。

COSTS boolean
包括每个规划节点的估计总成本，以及估计的行数和每行的宽度。
取值范围：
TRUE（缺省值）：显示估计总成本和宽度。
FALSE：不显示。

CPU boolean
打印 CPU 的使用情况的信息。
取值范围：
TRUE（缺省值）：显示 CPU 的使用情况。
FALSE：不显示。

DETAIL boolean
打印数据库节点上的信息。
取值范围：
TRUE（缺省值）：打印数据库节点的信息。
FALSE：不打印。

NODES boolean
打印 query 执行的节点信息。
取值范围：
TRUE（缺省值）：打印执行的节点的信息。

FALSE：不打印。

NUM_NODES boolean
打印执行中的节点的个数信息。
取值范围：
TRUE（缺省值）：打印数据库节点个数的信息。
FALSE：不打印。

BUFFERS boolean
包括缓冲区的使用情况的信息。
取值范围：
TRUE：显示缓冲区的使用情况。
FALSE（缺省值）：不显示。

TIMING boolean
包括实际的启动时间和花费在输出节点上的时间信息。
取值范围：
TRUE（缺省值）：显示启动时间和花费在输出节点上的时间信息。
FALSE：不显示。

PLAN
是否将执行计划存储在 plan_table 中。当该选项开启时，会将执行计划存储在 PLAN_TABLE 中，不打印到当前屏幕，因此该选项为 on 时，不能与其他选项同时使用。
取值范围：
ON（缺省值）：将执行计划存储在 plan_table 中，不打印到当前屏幕。执行成功返回 EXPLAIN SUCCESS。
OFF：不存储执行计划，将执行计划打印到当前屏幕。

FORMAT
指定输出格式。
取值范围：TEXT，XML，JSON 和 YAML。
默认值：TEXT。

PERFORMANCE
使用此选项时，即打印执行中的所有相关信息。

3.3 EXPLAIN-PLAN

功能描述

通过 EXPLAIN PLAN 命令可以将查询执行的计划信息存储于 PLAN_TABLE 表中。与 EXPLAIN 命令不同的是，EXPLAIN PLAN 仅将计划信息进行存储，而不会打印到屏幕。

语法格式

```
EXPLAIN PLAN
[ SET STATEMENT_ID = string ]
FOR statement ;
```

参数说明

EXPLAIN 中的 PLAN 选项表示需要将计划信息存储于 PLAN_TABLE 中，存储成功将返回“EXPLAIN SUCCESS”。

STATEMENT_ID 用户可以对查询设置标签，输入的标签信息也将存储于 PLAN_TABLE 中。

注意:

- 用户在执行 EXPLAIN PLAN 时，如果没有进行 SET STATEMENT_ID，则默认为空值。同时，用户可输入的 STATEMENT_ID 最大长度为 30 个字节，超过长度将会产生报错。
- 对于执行错误的 SQL 无法进行计划信息的收集。
- EXPLAIN PLAN 不支持在数据库节点上执行。
- PLAN_TABLE 中的数据是 session 级生命周期并且 session 隔离和用户隔离，用户只能看到当前 session、当前用户的数据。

PLAN_TABLE

PLAN_TABLE 显示用户通过执行 EXPLAIN PLAN 收集到的计划信息。计划信息的生命周期是 session 级别，session 退出后相应的数据将被清除。同时不同 session 和不同 user 间的数据是相互隔离的。

名称	类型	描述
statement_id	varchar2(30)	用户输入的查询标签。
plan_id	bigint	查询标识。
id	int	查询生成的计划中的每一个执行算子的编号。
operation	varchar2(30)	计划中算子的操作描述。
options	varchar2(255)	操作选项。
object_name	name	操作对应的对象名，非查询中使用到的对象别名。来自于用户定义。
object_type	varchar2(30)	对象类型。

object_owner	name	对象所属 schema，来自于用户定义。
projection	varchar2(4000)	操作输出的列信息。

字段描述

说明：

- object_type 取值范围为 PG_CLASS 中定义的 relkind 类型（TABLE 普通表，INDEX 索引，SEQUENCE 序列，VIEW 视图，COMPOSITE TYPE 复合类型，TOASTVALUE TOAST 表)和计划使用到的 rtekind(SUBQUERY, JOIN, FUNCTION, VALUES, CTE, REMOTE_QUERY)。

- object_owner 对于 RTE 来说是计划中使用的对象描述，非用户定义的类型不存在 object_owner。

- statement_id、object_name、object_owner、projection 字段内容遵循用户定义的大小写存储，其它字段内容采用大写存储。

- 支持用户对 PLAN_TABLE 进行 SELECT 和 DELETE 操作，不支持其它 DML 操作。

3.4 执行计划中的关键字

表访问方式

Seq Scan

全表顺序扫描。

Index Scan

优化器决定使用两步的规划：最底层的规划节点访问一个索引，找出匹配索引条件的行的位置，然后上层规划节点真实地从表中抓取出那些行。独立地抓取数据行比顺序地读取它们的开销高很多，但是因为并非所有表的页面都被访问了，这么做实际上仍然比一次顺序扫描开销要少。使用两层规划的原因是，上层规划节点在读取索引标识出来的行位置之前，会先将它们按照物理位置排序，这样可以最小化独立抓取的开销。

如果在 WHERE 里面使用的好几个字段上都有索引，那么优化器可能会使用索引的 AND 或 OR 的组合。但是这么做要求访问两个索引，因此与只使用一个索引，而把另外一个条件只当作过滤器相比，这个方法未必是更优。

索引扫描可以分为以下几类，他们之间的差异在于索引的排序机制。

Bitmap Index Scan

使用位图索引抓取数据页。

Index Scan using index_name

使用简单索引搜索，该方式表的数据行是以索引顺序抓取的，这样就令读取它们的开销更大，但是这里的行少得可怜，因此对行位置的额外排序并不值得。最常见的就是看到这种规划类型只抓取一行，以及那些要求 ORDER BY 条件匹配索引顺序的查询。因为那时候没有多余的排序步骤是必要的以满足 ORDER BY。

表连接方式

Nested Loop

嵌套循环，适用于被连接的数据子集较小的查询。在嵌套循环中，外表驱动内表，外表返回的每一行都要在内表中检索找到它匹配的行，因此整个查询返回的结果集不能太大（不能大于 10000），要把返回子集较小的表作为外表，而且在内表的连接字段上建议要有索引。

(Sonic) Hash Join

哈希连接，适用于数据量大的表的连接方式。优化器使用两个表中较小的表，利用连接键在内存中建立 hash 表，然后扫描较大的表并探测散列，找到与散列匹配的行。Sonic 和非 Sonic 的 Hash Join 的区别在于所使用 hash 表结构不同，不影响执行的结果集。

Merge Join

归并连接，通常情况下执行性能差于哈希连接。如果源数据已经被排序过，在执行融合连接时，并不需要再排序，此时融合连接的性能优于哈希连接。

运算符

sort

对结果集进行排序。

filter

EXPLAIN 输出显示 WHERE 子句当作一个"filter"条件附属于顺序扫描计划节点。这意味着规划节点为它扫描的每一行检查该条件，并且只输出符合条件的行。预计的输出行数降低了，因为有 WHERE 子句。不过，扫描仍将必须访问所有 10000 行，因此开销没有降低；实际上它还增加了一些（确切的说，通过 10000 * cpu_operator_cost）以反映检查 WHERE 条件的额外 CPU 时间。

LIMIT

LIMIT 限定了执行结果的输出记录数。如果增加了 LIMIT，那么不是所有的行都会被检索到。

3.5 SQL 语句性能优化策略

1、对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

2、应尽量避免在 where 子句中对字段进行 null 值判断，创建表时 NULL 是默认值，但大多数时候应该使用 NOT NULL，或者使用一个特殊的值，如 0，-1 作为默认值。

3、应尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，可以使用 UNION 合并查询：select id from t where num=10 union all select id from t where num=20。

4、in 和 not in 也要慎用，否则会导致全表扫描，对于连续的数值，能用 between 就不要用 in 了：Select id from t where num between 1 and 3。

5、下面的查询也将导致全表扫描：select id from t where name like ‘%abc%’或者 select id from t where name like ‘%abc’若要提高效率，可以考虑全文检索。而 select id from t where name like ‘abc%’才用到索引。

6、如果在 where 子句中使用参数，也会导致全表扫描。

7、应尽量避免在 where 子句中对字段进行表达式操作，应尽量避免在 where 子句中对字段进行函数操作。

8、很多时候用 exists 代替 in 是一个好的选择：select num from a where num in(select num from b)。用下面的语句替换：select num from a where exists(select 1 from b where num=a.num)。

9、索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过 6 个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

10、应尽可能的避免更新 clustered 索引数据列， 因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

11、尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。

12、尽可能的使用 varchar/nvarchar 代替 char/nchar，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

13、最好不要使用“*”返回所有：select * from t ，用具体的字段列表代替“*”，不要返回用不到的任何字段。

14、尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

15、使用表的别名：当在 SQL 语句中连接多个表时，请使用表的别名并把别名前缀于每个 Column 上。这样一来，就可以减少解析的时间并减少那些由 Column 歧义引起的语法错误。

16、使用“临时表”暂存中间结果 ：

简化 SQL 语句的重要方法就是采用临时表暂存中间结果，但是临时表的好处远远不止这些，将临时结果暂存在临时表，后面的查询就在 tempdb 中了，这可以避免程序中多次扫描主表，也大大减少了程序执行中“共享锁”阻塞“更新锁”，减少了阻塞，提高了并发性能。

17、一些 SQL 查询语句应加上 nolock，读、写是会相互阻塞的，为了提高并发性能，对于一些查询，可以加上 nolock，这样读的时候可以允许写，但缺点是可能读到未提交的脏数据。

使用 nolock 有 3 条原则：

查询的结果用于“插、删、改”的不能加 nolock；

查询的表属于频繁发生页分裂的，慎用 nolock ；

使用临时表一样可以保存“数据前影”，能采用临时表提高并发性能的，不要用 nolock。

18、常见的简化规则如下：

不要有超过 5 个以上的表连接（JOIN），考虑使用临时表或表变量存放中间结果。少用子查询，视图嵌套不要过深，一般视图嵌套不要超过 2 个为宜。

19、将需要查询的结果预先计算好放在表中，查询的时候再 Select，例如医院的住院费计算。

20、用 OR 的字句可以分解成多个查询，并且通过 UNION 连接多个查询。他们的速度只同是否使用索引有关，如果查询需要用到联合索引，用 UNION all 执行的效率更高。多个 OR 的字句没有用到索引，改写成 UNION 的形式再试图与索引匹配。一个关键的问题是否用到索引。

21、在 IN 后面值的列表中，将出现最频繁的值放在最前面，出现得最少的放在最后面，减少判断的次数。

22、下列 SQL 条件语句中的列都建有恰当的索引，但执行速度却非常慢：

SELECT * FROM record WHERE substrINg(card_no,1,4)=’5378’ (13 秒)

SELECT * FROM record WHERE amount/30< 1000 (11 秒)

SELECT * FROM record WHERE convert(char(10),date,112)=’19991201’ (10 秒)

分析：

WHERE 子句中对列的任何操作结果都是在 SQL 运行时逐列计算得到的，因此它不得不进行表搜索，而没有使用该列上面的索引。

如果这些结果在查询编译时就能得到，那么就可以被 SQL 优化器优化，使用索引，避免表搜索，因此将 SQL 重写成下面这样：

SELECT * FROM record WHERE card_no like ‘5378%’ (< 1 秒)

SELECT * FROM record WHERE amount< 1000*30 (< 1 秒)

SELECT * FROM record WHERE date= ‘1999/12/01’ (< 1 秒)

23、尽量将数据的处理工作放在服务器上，减少网络的开销，如使用存储过程。

存储过程是编译好、优化过、并且被组织到一个执行规划里、且存储在数据库中的 SQL 语句，是控制流语言的集合，速度当然快。反复执行的动态 SQL，可以使用临时存储过程，该过程（临时表）被放在 Tempdb 中。

24、当有一批处理的插入或更新时，用批量插入或批量更新，绝不会一条条记录的去更新。

25、在所有的存储过程中，能够用 SQL 语句的，我绝不会用循环去实现。

例如：列出上个月的每一天，我会用 connect by 去递归查询一下，绝不会去用循环从上个月第一天到最后一天。

26、尽量使用 exists 代替 select count(1)来判断是否存在记录，count 函数只有在统计表中所有行数时使用，而且 count(1)比 count(*)更有效率。

27、尽量使用“>=”，不要使用“>”。

28、索引的使用规范：

索引的创建要与应用结合考虑，建议大的 OLTP 表不要超过 6 个索引；

尽可能的使用索引字段作为查询条件，尤其是聚簇索引，必要时可以通过 index index_name 来强制指定索引；

避免对大表查询时进行 table scan，必要时考虑新建索引；

在使用索引字段作为条件时，如果该索引是联合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用；

要注意索引的维护，周期性重建索引，重新编译存储过程。

29、提高 GROUP BY 语句的效率，可以通过将不需要的记录在 GROUP BY 之前过滤掉。下面两个查询返回相同结果，但第二个明显就快了许多。

低效：

```
SELECT JOB , AVG(SAL)
FROM EMP
GROUP BY JOB
HAVING JOB =’ PRESIDENT’
OR JOB =’ MANAGER’
```

高效:

```
SELECT JOB , AVG(SAL)
FROM EMP
WHERE JOB =’ PRESIDENT’
OR JOB =’ MANAGER’
GROUP BY JOB
```

30、别名的使用，别名是大型数据库的应用技巧，就是表名、列名在查询中以一个字母为别名，查询速度要比建连接表快 1.5 倍。

31、避免死锁，在你的存储过程和触发器中访问同一个表时总是以相同的顺序；事务应经可能地缩短，在一个事务中应尽可能减少涉及到的数据量；永远不要在事务中等待用户输入。

32、避免使用临时表，除非却有需要，否则应尽量避免使用临时表，相反，可以使用表变量代替；大多数时候(99%)，表变量驻扎在内存中，因此速度比临时表更快，临时表驻扎在 TempDb 数据库中，因此临时表上的操作需要跨数据库通信，速度自然慢。

33、最好不要使用触发器：

触发一个触发器，执行一个触发器事件本身就是一个耗费资源的过程；

如果能够使用约束实现的，尽量不要使用触发器；

不要为不同的触发事件(Insert，Update 和 Delete)使用相同的触发器；

不要在触发器中使用事务型代码。

34、索引创建规则：

表的主键、外键必须有索引；

数据量超过 300 的表应该有索引；

经常与其他表进行连接的表，在连接字段上应该建立索引；

经常出现在 Where 子句中的字段，特别是大表的字段，应该建立索引；

索引应该建在选择性高的字段上；

索引应该建在小字段上，对于大的文本字段甚至超长字段，不要建索引；

复合索引的建立需要进行仔细分析，尽量考虑用单字段索引代替；

正确选择复合索引中的主列字段，一般是选择性较好的字段；

复合索引的几个字段是否经常同时以 AND 方式出现在 Where 子句中？单字段查询是否极少甚至没有？如果是，则可以建立复合索引；否则考虑单字段索引；

如果复合索引中包含的字段经常单独出现在 Where 子句中，则分解为多个单字段索引；

如果复合索引所包含的字段超过 3 个，那么仔细考虑其必要性，考虑减少复合的字段；

如果既有单字段索引，又有这几个字段上的复合索引，一般可以删除复合索引；

频繁进行数据操作的表，不要建立太多的索引；

删除无用的索引，避免对执行计划造成负面影响；

表上建立的每个索引都会增加存储开销，索引对于插入、删除、更新操作也会增加处理上的开销。另外，过多的复合索引，在有单字段索引的情况下，一般都是没有存在价值的；相反，还会降低数据增加删除时的性能，特别是对频繁更新的表来说，负面影响更大。

尽量不要对数据库中某个含有大量重复的值的字段建立索引。

35、查询缓冲并不自动处理空格，因此，在写 SQL 语句时，应尽量减少空格的使用，尤其是在 SQL 首和尾的空格（因为查询缓冲并不自动截取首尾空格）。

36、我们应该为数据库里的每张表都设置一个 ID 做为其主键，而且最好的是一个 INT 型的（推荐使用 UNSIGNED），并设置上自动增加的 AUTO_INCREMENT 标志。

37、在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON，在结束时设置 SET NOCOUNT OFF。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE_IN_PROC 消息。

38、EXPLAIN SELECT 查询用来跟踪查看效果：

使用 EXPLAIN 关键字可以让你知道 openGauss 是如何处理你的 SQL 语句的。这可以帮你分析你的查询语句或是表结构的性能瓶颈。EXPLAIN 的查询结果还会告诉你你的索引主键被如何利用的，你的数据表是如何被搜索和排序的。

39、当只要一行数据时使用 LIMIT 1 ：

当你查询表的有些时候，你已经知道结果只会有一条结果，但因为你可能需要去 fetch 游标，或是你也许会去检查返回的记录数。

在这种情况下，加上 LIMIT 1 可以增加性能。这样一来，MySQL 数据库引擎会在找到一条数据后停止搜索，而不是继续往后查少下一条符合记录的数据。

40、优化表的数据类型，选择合适的数据类型：

原则：更小通常更好，简单就好，所有字段都得有默认值，尽量避免 null。 在创建表的时候，为了获得更好的性能，我们可以将表中字段的宽度设得尽可能小。

例如：在定义邮政编码这个字段时，如果将其设置为 CHAR(255)，显然给数据库增加了不必要的空间。甚至使用 VARCHAR 这种类型也是多余的，因为 CHAR(6)就可以很好的完成任务了。

41、任何对列的操作都将导致表扫描，它包括数据库函数、计算表达式等等，查询时要尽可能将操作移至等号右边。