慢 SQL 根因测试

测试中用到的：

场景一

创建表 t1 和 t2:

```
create table t1(id int, c1 text, c2 text);
create table t2(id int, c1 text, c2 text);
```

插入测试数据（提供参考）：

```
insert into t1 select generate_series(1,10000000),md5(random()::text), md5(random()::text);
insert into t2 select generate_series(1,10000000),md5(random()::text), md5(random()::text);
```

场景二

TPCH 场景，因为 TPCH 场景中的 SQL 语句，基本都可以认为是慢 SQL. 此处，我们以该类型的 SQL 语句为例，进行测试，看一下我们的功能是否可以覆盖 TPCH 中的问题场景。其中，TPCH 的 22 条语句，其对应的慢 SQL 根因有很多论文在研究。因此，该根因也比较容易对照正误。

说明：上述场景只是为了对慢 SQL 根因诊断进行讲解，用户可以使用已有的场景进行测试。

根因一（LOCK_CONTENTION）

1. 在 t1 中插入测试数据。
2. 在 session1 中执行：

   begin;

   update t1 set c1=’slowsqltest’ where id between 100 and 150;

   select pg_sleep(5); --只作为举例场景，不做参考

   end;
3. 在 session2 中执行：

   delete from t1 where id=101;

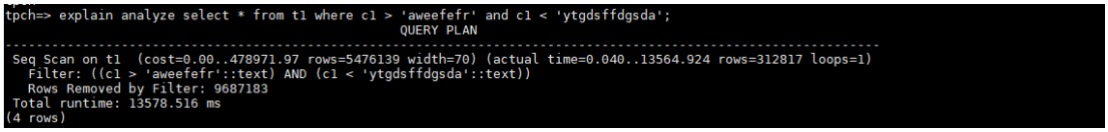
   此时因为该 SQL 被阻塞因此执行较慢。
4. 对该 SQL 执行根因诊断：

根因中指出该SQL执行期间被阻塞，阻塞其执行的SQL：update t1 set c1=’slowsqltest’ where id between 100 and 150;。

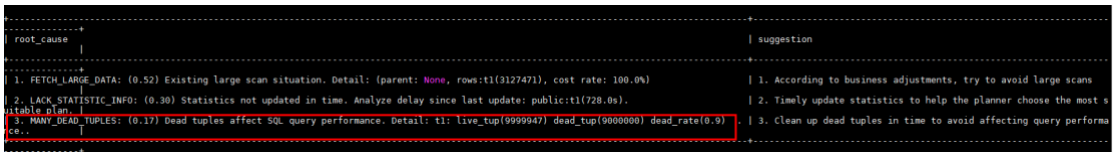
根因二（MANY_DEAD_TUPLES）：

1. 构造场景一
2. t1 表中插入数据：insert into t1 select generate_series(1,10000000),md5(random()::text), md5(random()::text);
3. 执行：update t1 set c1=’adasdadsddfdfd’ where id > 1000000;此时生成大量 dead tuples。
4. 测试语句：select * from t1 where c1 > 'aweefefr' and c1 < 'ytgdsffdgsda';

   清理死元组之前执行计划

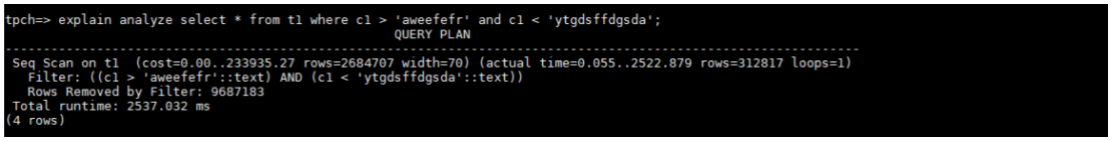


5. 进行慢 SQL 根因诊断：



其中第三个根因指出相关表存在大量死元组，影响 SQL 执行性能。

6. 清理之后执行计划



发现清理死元组后执行时间大大缩短。

根因三 FETCH_LARGE_DATA：

1. 在 t1 上，运行：select c1, c2 from t1 where c2 > 'asdsafd23324r' and c2 < 'c32454fdfegdg';

2. 执行计划：

```
tpch=> explain analyze select c1, c2 from t1 where c2 > 'asdsafd23324r' and c2 < 'c32454fdfegdg';
                                              QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Seq Scan on t1  (cost=0.00..347913.00 rows=754238 width=66) (actual time=0.028..3081.611 rows=748958 loops=1)
   Filter: ((c2 > 'asdsafd23324r'::text) AND (c2 < 'c32454fdfegdg'::text))
   Rows Removed by Filter: 9251042
 Total runtime: 3114.724 ms
(4 rows)
```

3. 执行根因诊断：

```
+------------------------------------------------------------------------------------------------------------------------------+------------------------------------------------------------------------
---------------+
| root_cause                                                                                                                   | suggestion
               |
+------------------------------------------------------------------------------------------------------------------------------+------------------------------------------------------------------------
---------------+
| 1. FETCH_LARGE_DATA: (0.52) Existing large scan situation. Detail: (parent: None, rows:t1(754238), cost rate: 100.0%)        | 1. According to business adjustments, try to avoid large scans
               |
| 2. LACK_STATISTIC_INFO: (0.30) Statistics not updated in time. Analyze delay since last update: public:t1(1389.0s).          | 2. Timely update statistics to help the planner choose the most
suitable plan. |
| 3. MANY_DEAD_TUPLES: (0.17) Dead tuples affect SQL query performance. Detail: t1: live_tup(9999947) dead_tup(18000000) dead_rate(1.8)  . | 3. Clean up dead tuples in time to avoid affecting query perform
ance..         |
+------------------------------------------------------------------------------------------------------------------------------+------------------------------------------------------------------------
```

根因中指出 **SQL** 执行期间扫描大量元组（**754238**），导致性能较差。

根因四 **UNREASONABLE_DATABASE_KNOB**：当前不考虑此根因

根因五 **UNUSED_AND_REDUNDANT_INDEX**：**SQL** 相关表存在无用索引与冗余索引，它们会对 **DML** 性能产生一定的负向作用，即存在写放大现象。
根因七 **INSERT_LARGE_DATA**：批量插入数据时执行耗时较长
根因九 **TOO_MANY_INDEX**：表中存在过多索引也会对 **DML** 产生一定的负向作用
下面对上述三个根因一起测试：

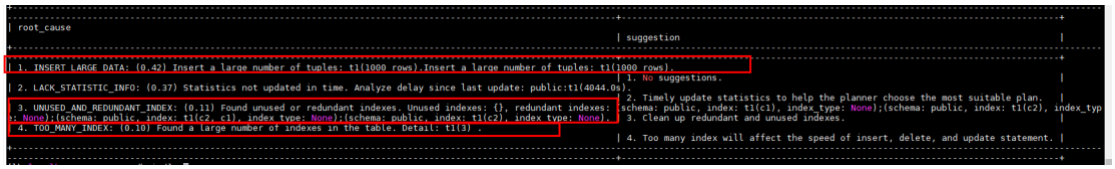
1. 在 **t1** 的 **c1** 上创建以下索引：
   create index on t1(c1);
   create index on t1(c1, c2);
   create index on t1(c2);
   create index on t1(c2, c1);
2. 执行 **SQL**，保证 **SQL** 会使用索引（用户可增加 **SQL** 数量）：
   select c1 from t1 where c1<'adsadfre' and c2<'tfdsfd';
   select c1 from t1 where c1>'adsadfre' and c1<'cfdsfd';
   select c1 from t1 where c2>'adsadfre' and c2<'cfdsfd';
3. 此时查询 **pg_stat_user_indexes** 获取索引使用情况：

```
tpch=> select * from pg_stat_user_indexes where relname='t1';
 relid | indexrelid | schemaname | relname | indexrelname | idx_scan | idx_tup_read | idx_tup_fetch
-------+------------+------------+---------+--------------+----------+--------------+---------------
 44043 |      44265 | public     | t1      | t1_c2_c1_idx |       26 |           26 |            52
 44043 |      44264 | public     | t1      | t1_c2_idx    |        1 |      1324255 |             0
 44043 |      44263 | public     | t1      | t1_c1_c2_idx |        0 |            0 |             0
 44043 |      44245 | public     | t1      | t1_c1_idx    |       36 |     11444629 |       2718122
(4 rows)
```

   发现其中 **t1_c1_c2_idx** 索引未被使用。
4. 执行 **SQL: insert into t1 select generate_series(10000000, 10100000),md5(random()::text), md5(random()::text);**执行计划：

```
tpch=> explain analyze insert into t1 select generate_series(10000000, 10100000),md5(random()::text), md5(random()::text);
                                              QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Insert on t1  (cost=0.00..15.02 rows=1000 width=68) (actual time=0.447..53173.563 rows=100001 loops=1)
   ->  Result  (cost=0.00..5.02 rows=1000 width=0) (actual time=0.032..434.470 rows=100001 loops=1)
 Total runtime: 53173.696 ms
(2 rows)
```

   此时执行时间为 **53s** 左右。
5. 对 **SQL: insert into t1 select generate_series(10000000, 10100000),md5(random()::text), md5(random()::text);**执行根因诊断：



发现根因中存在：

（1）批量插入导致性能较差；

（2）无用索引可以删除；

（3）过多索引导致插入性能较差；

6. 按照根因描述，删除其中的无用索引或冗余索引，再次执行该语句，执行计划如下：

```
tpch=> explain analyze  insert into t1 select generate_series(10000000, 10100000),md5(random()::text), md5(random()::text);
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Insert on t1  (cost=0.00..15.02 rows=1000 width=68) (actual time=0.201..13178.647 rows=100001 loops=1)
   ->  Result  (cost=0.00..5.02 rows=1000 width=0) (actual time=0.036..316.325 rows=100001 loops=1)
 Total runtime: 13178.758 ms
(3 rows)
```

此时执行时间缩短为 13s 左右，提升较大。

根因六 UPDATE_LARGE_TABLE：和 INSERT_LARGE_DATA 类似

1. 对表 t1 执行 SQL：update t1 set c1='1234567890' where id < 9000000;
2. 对 SQL 执行慢 SQL 诊断：

根因中指出存在批量更新（t1: 2999861）。

根因八 DELETE_LARGE_DATA：和 INSERT_LARGE_DATA 类似

1. 对表 t1 执行 SQL: delete from t1 where id > 9000000;
2. 对 SQL 执行慢 SQL 诊断：

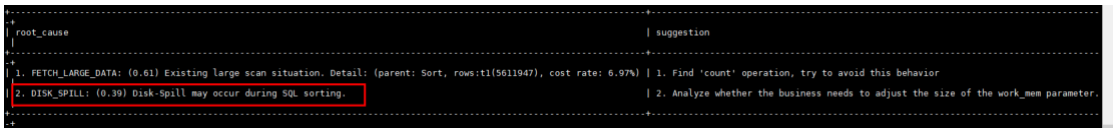
根因中指出存在批量删除（t1: 2990904）。

根因十 DISK_SPILL：SQL 执行过程中产生 SORT 或 HASH 算子产生落盘行为

1. 将数据库 work_mem 参数设置成 64kB: gs_guc reload -D /media/sdd/lk_new/data -c "work_mem=64kB"
2. 往表 t2 中插入数据：insert into t2 select generate_series(0, 10000000),md5(random()::text), md5(random()::text);
3. 执行 SQL：select c1, count(distinct c2) from t1 group by c1, c2;执行计划：

```
                                                                 QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------------
 GroupAggregate  (cost=3530183.69..3642422.63 rows=5611947 width=60) (actual time=71979.061..93188.835 rows=10281394 loops=1)
   Group By Key: c1, c2
   ->  Sort  (cost=3530183.69..3544213.56 rows=5611947 width=52) (actual time=71978.991..82145.608 rows=10300003 loops=1)
         Sort Key: c1, c2
         Sort Method: external merge  Disk: 634040kB
         ->  Seq Scan on t1  (cost=0.00..254032.47 rows=5611947 width=52) (actual time=0.026..1956.538 rows=10300003 loops=1)
 Total runtime: 93992.041 ms
(7 rows)
```

4. 对慢 SQL 执行诊断：



诊断根因中包含落盘，建议根据实际业务分析是否需要调整 work_mem 的值。

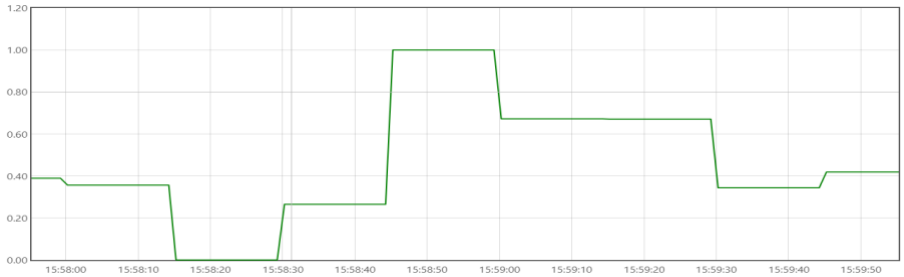
根因十一 VACUUM_EVENT 和根因十二 ANALYZE_EVENT：vacuum/analyze 操作会占用一定资源，影响 SQL 执行性能，当前不考虑

根因十三 WORKLOAD_CONTENTION：数据库负载集中影响了 SQL 执行效率，主要包括①数据库自身进程资源消耗异常（CPU、IO 等），其可能是其他大事务或批量操作导致；②数据库 TPS/QPS 较正常情况较大

根因十四 CPU_RESOURCE_CONTENTION：数据库外进程 CPU 资源抢占严重，造成数据库进程 CPU 资源不足，在具体诊断资源异常时，慢 SQL 诊断工具结合了异常检测算法，对资源突增或漂移等情况进行检测，其他根因方法类似

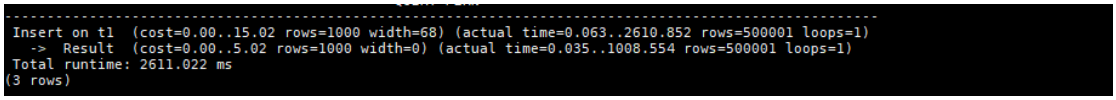
根因十五 IO_RESOURCE_CONTENTION：IO 资源紧张，导致部分 SQL 性能较差

1. 在场景一种，模拟 IO 资源抢占，此处主要使用 linux 上的 dd 插件：dd if=/dev/zero of=/media/sdc/test_dir/testfile bs=1K count=8M oflag=append conv=notrunc
2. 此时 IOUtils 情况：



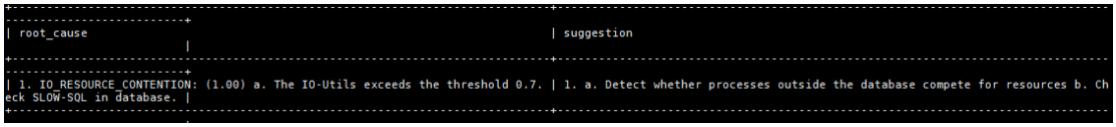
可以发现此时 IOUtils 偏高

3. 此时运行 SQL：insert into t1 select generate_series(500000, 1000000),md5(random()::text), md5(random()::text);



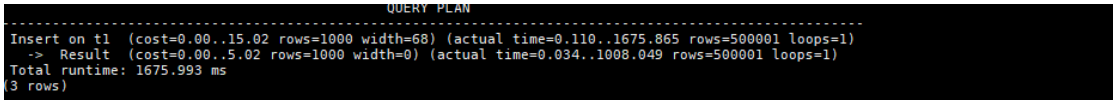
发现执行时间 2.6s 左右

4. 运用慢 SQL 诊断，



诊断结果指出当前 IOUtils 超过设定的 0.7 阈值，建议检查是否存在外部进程 IO 资源占用。

5. 将 IO 占用进程停止，再次运行 SQL：



此时 SQL 性能提升明显。

根因十六 MEMORY_RESOURCE_CONTENTION：数据库外进程 MEMORY 资源抢占严重，造成数据库进程 MEMORY 资源不足

根因十七 LARGE_NETWORK_DROP_RATE： 数据库集群网络异常，主要体现在丢包率异常

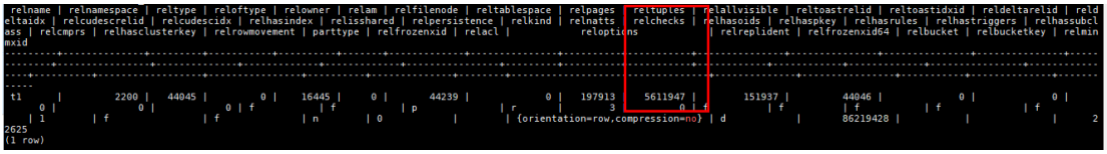
根因十八 OS_RESOURCE_CONTENTION：服务器句柄资源异常

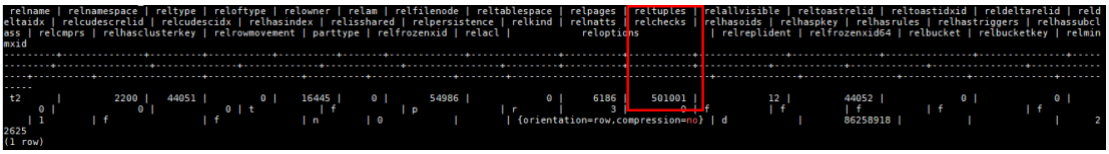
说明：根因十三到十八根因类似，文档以 IO 资源为例子进行说明，其他类似。

根因十九 WAIT_EVENT：诊断 SQL 执行期间主要时间消耗的 wait event,当前不考虑此根因

根因二十 LACK_STATISTIC_INFO：统计信息长时间未更新，导致基数估计不准确，进而影响执行计划.

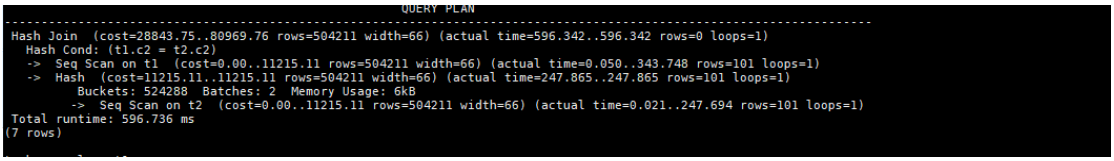
1. 在场景一中，对表 t1 和 t2 执行 analyze 操作，更新统计信息
2. 执行 SQL：delete from t1 where id > 100; delete from t2 where id > 100,查询统计信息：





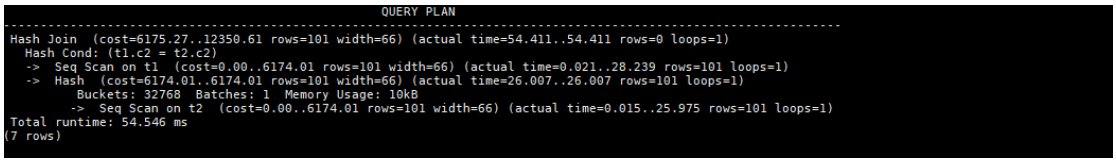
发现此时统计信息未更新，仍然为 delete 之前的状态。

3. 执行 SQL: select t1.c1, t2.c1 from t1, t2 where t1.c2=t2.c2;



可以发现执行计划中基数估计偏差较大，导致在数据量较小的情况下选择 hashjoin

4. 对表执行 analyze，更新统计信息；
5. 再次执行 SQL: select t1.c1, t2.c1 from t1, t2 where t1.c2=t2.c2;执行计划为：



此时执行计划选择了 nestloop,速度提升明显。

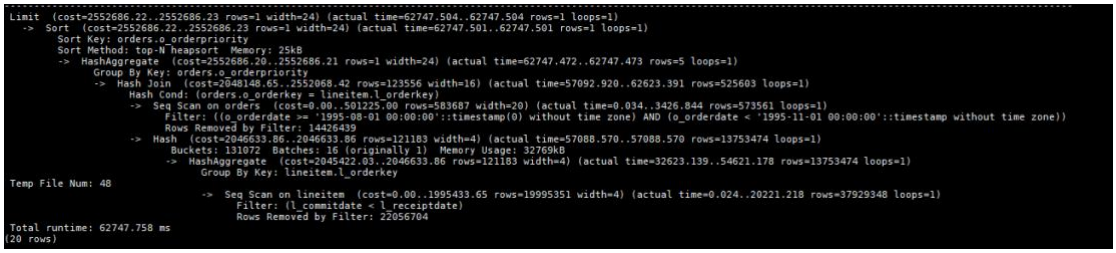
根因二十一 MISSING_INDEXES：由于 SQL 缺失索引导致性能问题

1. 在 TPCH 场景下执行 SQL：

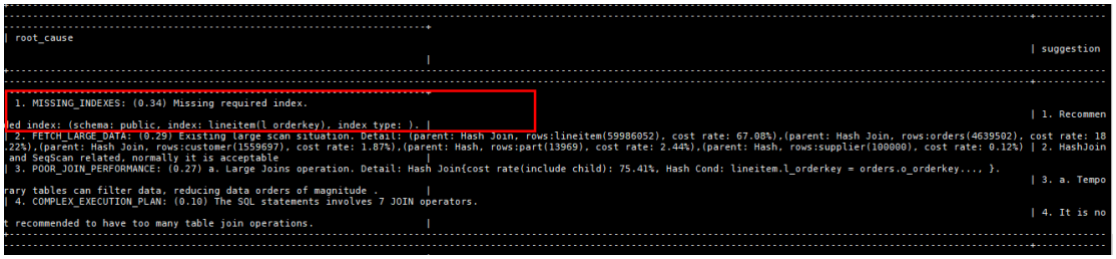
```
select o_orderpriority,
        count(*) as order_count
from
orders
where
```

```
o_orderdate >= date '1995-08-01'
and o_orderdate < date '1995-08-01' + interval '3' month
and exists (
select
*
from
lineitem
where
l_orderkey = o_orderkey
and l_commitdate < l_receiptdate
)
group by
o_orderpriority
order by
o_orderpriority
LIMIT 1;
```

其执行计划为：

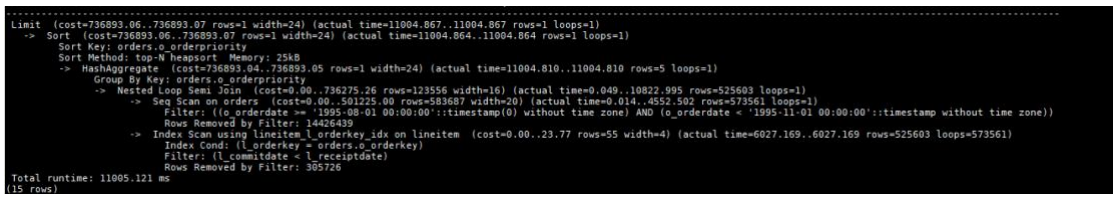


2. **对该 SQL 执行诊断：**



诊断建议在 lineitem(l_orderkey)上创建索引。

3. **按照建议创建索引并获取执行计划：**



发现索引对 SQL 提升较大

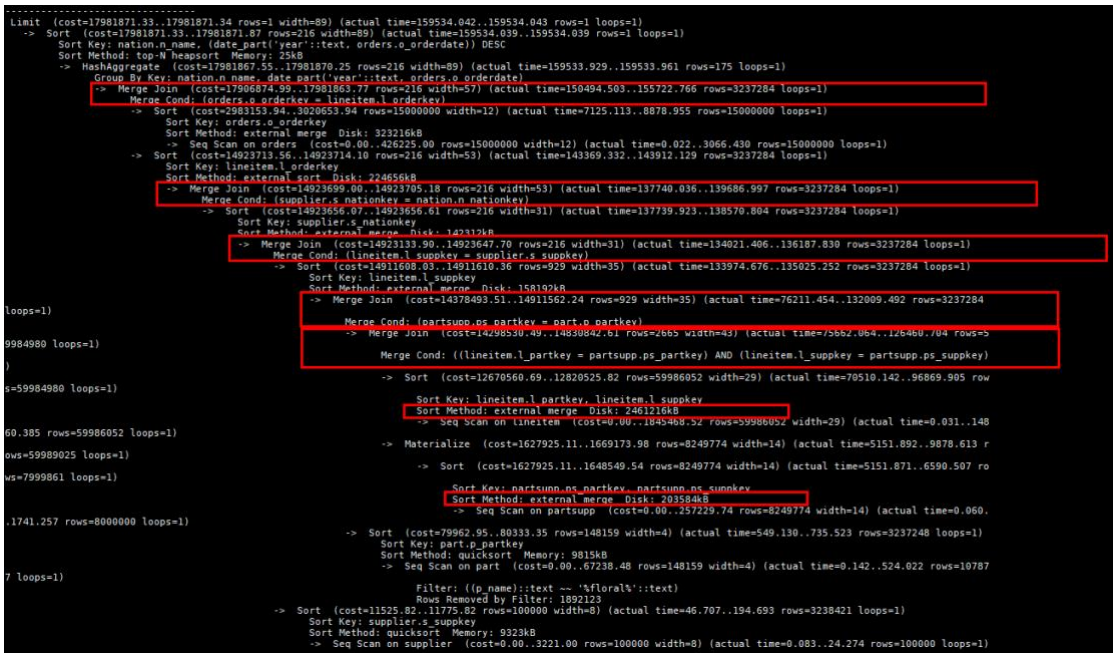
根因二十二 POOR_JOIN_PERFORMANCE: Join 算子存在以下几种情况，① 在适合使用 hashjoin 的场景，由于 GUC 参数设置或基数估计误差，导致执行计划没有使用最合适的 join 算子；② Large join，即 join 涉及元组较多；

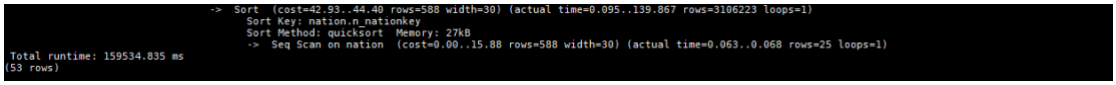
情况 1：

1. **设置 enable_hashjoin 为 off；**
2. **在 tpch 场景下执行 SQL:**

```
select
        nation,
        o_year,
        sum(amount) as sum_profit
from
        (
                select
                        n_name as nation,
                        extract(year from o_orderdate) as o_year,
                        l_extendedprice * (1 - l_discount) - ps_supplycost *
l_quantity as amount
                from
                        part,
                        supplier,
                        lineitem,
                        partsupp,
                        orders,
                        nation
                where
                        s_suppkey = l_suppkey
                        and ps_suppkey = l_suppkey
                        and ps_partkey = l_partkey
                        and p_partkey = l_partkey
                        and o_orderkey = l_orderkey
                        and s_nationkey = n_nationkey
                        and p_name like '%floral%'
        ) as profit
group by
        nation,
        o_year
order by
        nation,
        o_year desc
 LIMIT 1;
```

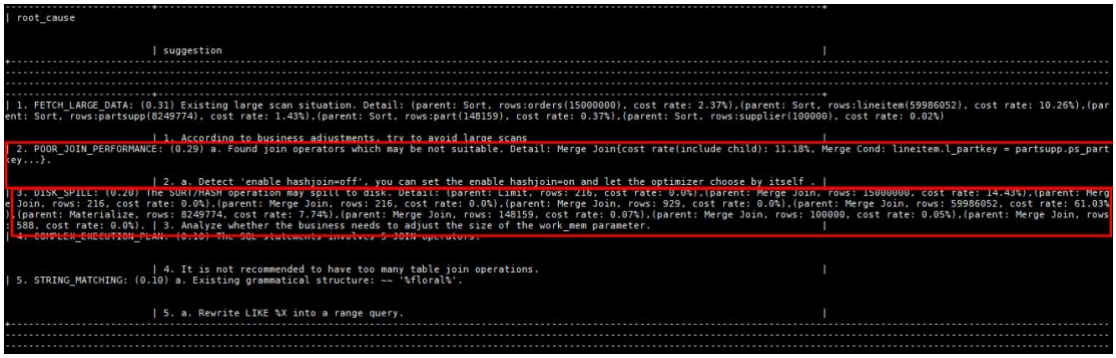
执行计划如下：





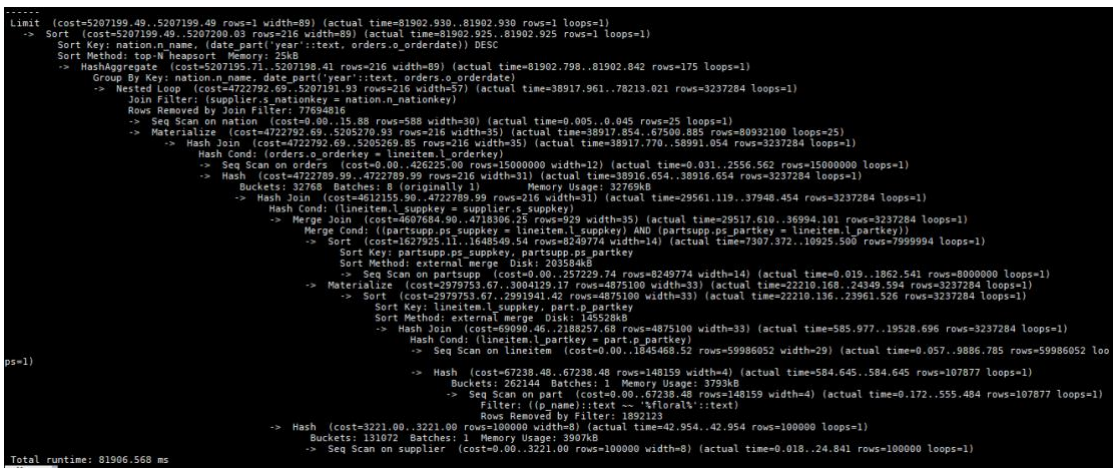
发现执行计划选择 **merge join** 算子，排序时发生落盘现象，执行效率较低；

3. 对该 **SQL** 执行诊断：



其中根因 **2** 指出 **join** 算子可能不合理，原因是 **enable_hashjoin** 参数设置不合理，建议设置为 **on**，让优化器选择具体的算子。

4. 按照建议我们将 **enable_hashjoin** 设置为 **on**，并再次执行 **SQL**，执行计划如下：



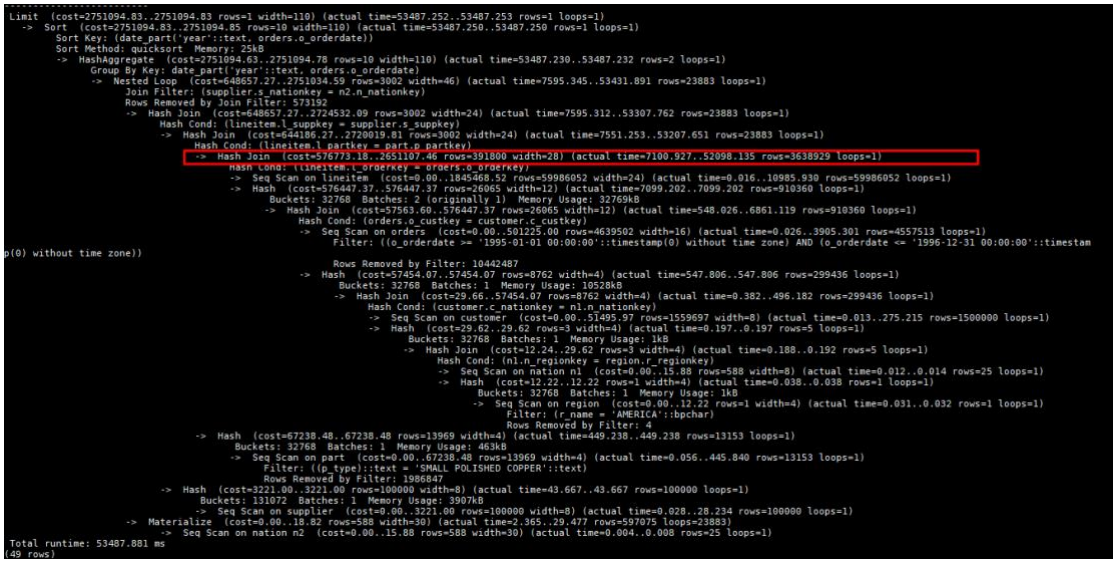
可以发现此时执行效率提升较大。

情况 2

1. 在 tpch 场景中，执行 SQL：

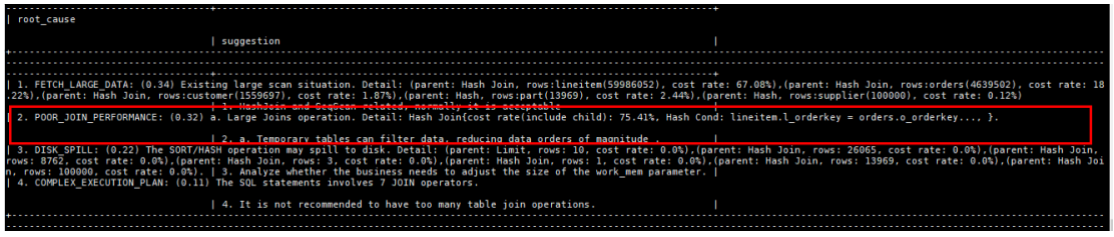
```
select
        o_year,
        sum(case
                when nation = 'BRAZIL' then volume
                else 0
        end) / sum(volume) as mkt_share
from
        (
                select
                        extract(year from o_orderdate) as o_year,
                        l_extendedprice * (1 - l_discount) as volume,
                        n2.n_name as nation
                from
                        part,
                        supplier,
                        lineitem,
                        orders,
                        customer,
                        nation n1,
                        nation n2,
                        region
                where
                        p_partkey = l_partkey
                        and s_suppkey = l_suppkey
                        and l_orderkey = o_orderkey
                        and o_custkey = c_custkey
                        and c_nationkey = n1.n_nationkey
                        and n1.n_regionkey = r_regionkey
                        and r_name = 'AMERICA'
                        and s_nationkey = n2.n_nationkey
                        and o_orderdate between date '1995-01-01' and date
'1996-12-31'
                        and p_type = 'SMALL POLISHED COPPER'
        ) as all_nations
group by
        o_year
order by
        o_year
LIMIT 1;
```

其执行计划为：



可以发现其中部分 **join** 算子运算的元组数非常大，导致执行效率较低。

2. 对该 **SQL** 执行诊断，结果为：



根因 **2** 说明该 **SQL** 执行过程中存在数据量较大的 **join** 操作。

根因二十三 **COMPLEX_BOOLEAN_EXPRESSIONS**：针对此类语句：**select * from t2 where c2 not in ('a', 'b', …,)**语句，如果 **NOT IN** 中元素过多则 **SQL** 性能较差，一般不建议超过 **10** 个

1. 在场景一中执行 **SQL**:

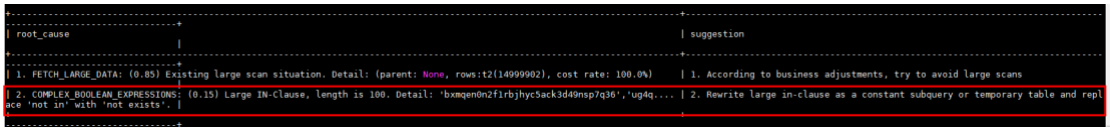
select * from t2 where c2 not in ('bxmqen0n2f1rbjhyc5ack3d49nsp7q36','ug4qxtxvkk1zp8t3ye8wbnzeprq1418n','b45gs4uzy8ftt4cxs38jok1spmvk1br7','85clfj5xnmp5jjziv6rvjcx9tb7753nk','pzleufy8nss7v8tct6i3kug4apc9p6b5','5t71g5atpw188tvhfmr91kj954vu27nx','nthr0grrfhqmmqpu0kxxzmjywbwrw6g9','aql9ofciolayembfoffquic3mufbm3ke','djtk1zpkfjugzlbkoqcwm027h8htiy14','ll1vqxkk16d0k12wtetq3zgjuuithdlt','t673fc6mu9657gwqr35c5rcy95lpapv7','chntpsxtr5cfu96x09fugg99xqw48pdy','5dtfboa5m36d7voiikhcgdigyf2a61oh','16ztkg5a23erwkadxy7602gvy37x0kiz','3r4kgaeqbuzpj2sbq6ce0tiqsh8x3ivy','21y7fvey4i26g9xyt9mhl2n3v1bgdnol','hlp189r1pggciiya80xiurg0qk9t146y','1j4n8fygto8nb6alqd0feky1qta6lcca','mboeq6h8qjm9v3h2qym1vn5k96i0okjc','rihng0fuglurx9ufrg0smro2050tzvus','sqrv4mmyjyv2bt5yb6i32qzeewkw8fh3','dnx0gf1rxhe16ytfq6v59cw5hv6fc1oz','8kdr6db46bx6zjfunn2yzqvbarfubuse','bumyeo5ihqf245wi7yvqvzu4c9cq542p','zqtzbzn47jixbvmhyg11h4a1dzgu3rol','s68a8mr5d9qqxh3udqp0ciw4of6ls8gq','3nsum4nd9d90nnwahmqm8hfay9bp8bu6','v90p0w78y6gdordbnoq74qjtplc5slos','njbmrbr32i8vbl52xaj28igqwio9glks','wgraxhjmw8pxliytrfx94ny164nuxewi','1f8dji4nsh8w86g11dzea5kyl5g8sqqi','3h066wnneylktd1d25lym05t35kr4iax','qabrzg02vemj4i54yobweqqk1zmvhmcr','ri5e090as2m47n61wp6apj3hf91mqv1o','76pk0qehwk81j0hn65xxbni9346zik9h','e05q729apzruffa4877zzuesp725myyf','833tej8ky0jcxqkn9fbpxpoem11ul6qe','t4yvhd

d93wn3adhmoaczlw4w9oej9k9k','xnsc1ds32g1t6ws4yla5z7cwaibna4qv','l0z7g6ut9n1g5b5r16np4p88us069yw4','essx3vxyb8rsr5svr01ol4xwv69drwil','hpc6n32lbx3i2jeunjzltfjmci815iy9','e8xmfexyz0r4v5dvl1axslc7z06e3vql','qilqvj63ekrjzfirxb7jqucgp76oywx7','6tdo7hasam67823srw9y46y18k525lyr','50xm88rlo4rs4b0so9pdhwqz5w2fma9c','x12mt0tsh4id82pp0np034r2coqcjt5w','8qwhcf5rhygduy12ow02fp7twnuj72lo','2gm22d9rzibly6aw3spgzkadeuyo6q4f','26jzvs5s6ok9ip2xn6o7qky95lxj5j0o','5pi9hnvyyx3gol1o9485s2g6arf7gcnv','bjmeppf154r0wkwswahye19qkuobh5kj','6g5vx15lnhwupk3zriyidmyoe78naru1','5ltt4d5qmi47ogzpqkjwhrbv6wgogwda','zeiq1qht8wco8kkpv96h91oyp21rfnxz','hjo039u7k9hp7kfbj0aj5u6w4r9hx4zv','d0cqm3a9mvrm1kox9y0dppbpfxrt1o24','d3im71k85hah0k7x4vuxau7i9i6f94b6','35wa7cjl5hslhcpuj33auogh9r1a1val','urcewbhrfcroyt969aszumky0xrbyymh','t03k6ykgkrzpdqi3n1swi102175mvi86','21gwzyi5gymqdi4si3p4g9qvwvaq6i4j','eazjyy842n324k9hs096xqznib6kb2o9','eun4vxf6yw2kvz8evjr2niix2cjm8jtr','2bmt46zc73tr9ewqzrm6bukrqgbaiwo8','a4lmiipxvfsahjyaencyekjvyp8y42ip','86m29taxlg2ijy2e1ib4gf89mtntx5zp','8s9zjffakti27fuofw93egv7u6mccdnk','ct4njx95dkmse3slh3mdhwufajrb67l1','rt45rwqgdlq2zol8goyjighdq20cfv2i','m933p3tf9x1f74qlev7p4hd1fi1f4kdo','6m2n5uh48r3191t6auovk131x1xbplqs','1yyiqsl1paxj7sx16zep7sd86tuqoqde','h7otxcktakyjsbmkx1waormcm1zze7nf','beajnf9de2az4mhdxs6nmpw747xjq2gv','4fo9dzra67kxscnxzio4qq691lnyscvs','8wa6hsrinzmkljqao8kqisgg2820l65e','hl2z00axi7gi1va71vetrttyzgug56au','zua9z2o88qichblf99ez1iz4zms6pxzc','fo1ril2u8557qogj9394oh1bpenaqomj','fz2k1agwbmatw5reompbpx399v1qwsul','1sc3p5jfacm04v6amadorvzeofubukka','1g5b4nkrmixcvbcctwgrn8nh0luhu087','hnihptwyemtmsejgy9trdr583nph1t98','9dlf46firpjgsn2wfzqmt1nzbx0mrr2z','5hg5k0bdsll99o0eg2vhf1velrbjz3ny','pmph5uuvu8g24sygaljkc7mhp432n27p','e4sldi4oh8nit3mvk9v6p2pcbpb1sa1s','w8ar7ec3ncs31exblyuu70eg5duflxcr','42oq6r0f530401y0jlhren8gejrprm5u','jyfjvbqvxs2btsskln6leuljayyvgqyg','pfrliktl93n5n45r0402re9lzkxicz1i','0jxswszqcoh6f0e5ah2647qg0hf5e046','9lrhjokxb79gjid0ihn739nboib2q8h8','kp03oux48f2n0t7npdmbu99449z8jtx7','c0ixuezekn8lifrfjm7cneqeyzlrlcpf','si5nol75dkdn6k44uyen29juh4snz7up','blt0zurq01za2062pbk577r33vve71vm','1d7a4375hw0j5ir3keh96zzfci9g574f','a2yaoc377miinzcxm0qx804z1i868oat');

执行计划为：

发现执行计划对 t2 表走了全表扫描，代价较大。

2. 对该 SQL 进行诊断：



其中第二条根因指出 not in 的元素长度为 100，建议将 in-clause 中的元素改为临时表或子查询，并用 not exists 替换 not in；

3. 按照诊断建议，首先将 in-clause 中的元素添加到临时表 string 中，并将 SQL 修改为：select * from t2 where not exists(select info from string where string.info=t2.c2);修改后执行计划为：

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
Hash Anti Join  (cost=39.18..17513.17 rows=498703 width=70) (actual time=0.369..284.208 rows=500000 loops=1)
  Hash Cond: (t2.c2 = string.info)
  ->  Seq Scan on t2  (cost=0.00..11173.00 rows=500000 width=70) (actual time=0.013..89.603 rows=500000 loops=1)
  ->  Hash  (cost=22.97..22.97 rows=1297 width=33) (actual time=0.104..0.104 rows=100 loops=1)
        Buckets: 32768  Batches: 1  Memory Usage: 7kB
        ->  Seq Scan on string  (cost=0.00..22.97 rows=1297 width=33) (actual time=0.036..0.051 rows=100 loops=1)
Total runtime: 316.224 ms
(7 rows)
```

可以看出此时执行计划使用了 hashjojn 算子，执行效率提升较大。

根因二十四 **STRING_MATCHING**：主要包括两种情况，① **where col like '%xxx'**正则匹配，导致 **col** 字段索引失效；②**where func(col) = 'xxx'**，在索引字段上使用函数，导致 **col** 字段索引失效

情况 1：

（1） 在场景一表 **t2** 的 **c1** 字段创建索引：**create index on t1(c1);**

（2） 执行 **SQL**：**select t2.c1, t1.c1 from t1, t2 where t2.c1 like 'abc%' and t2.c2=t1.c2;**，执行计划为：

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
Hash Join  (cost=372704.78..647797.05 rows=1500 width=52) (actual time=2630.406..5250.353 rows=11 loops=1)
  Hash Cond: (t1.c2 = t2.c2)
  ->  Seq Scan on t1  (cost=0.00..254032.47 rows=5611947 width=52) (actual time=0.003..1440.138 rows=10300003 loops=1)
  ->  Hash  (cost=372686.03..372686.03 rows=1500 width=66) (actual time=2503.364..2503.364 rows=3582 loops=1)
        Buckets: 32768  Batches: 1  Memory Usage: 343kB
        ->  Seq Scan on t2  (cost=0.00..372686.03 rows=1500 width=66) (actual time=0.196..2501.369 rows=3582 loops=1)
              Filter: (c1 ~~ 'abc%'::text)
              Rows Removed by Filter: 14996420
Total runtime: 5250.557 ms
(9 rows)
```

可以看出 **t2.c1** 走了全表扫描，导致执行代价较大。

（3） 对该 **SQL** 执行诊断：

```
+-------------------------------------------------------------------------------------------------------------------------------------------+-----------------------------------------------+
| root_cause                                                                                                                                | suggestion                                    |
+-------------------------------------------------------------------------------------------------------------------------------------------+-----------------------------------------------+
| 1. FETCH_LARGE_DATA: (0.44) Existing large scan situation. Detail: (parent: Hash Join, rows:t1(5611947), cost rate: 39.21%),(parent: Hash, rows:t2(1500), cost rate: 57.53%) | 1. HashJoin and SeqScan rela |
| ted, normally it is acceptable                                                                                                            |                                               |
| 2. POOR_JOIN_PERFORMANCE: (0.41) a. Large Joins operation. Detail: Hash Join(cost rate: 42.47%, Hash Cond: (t1.c2 = t2.c2)..., ).        | 2. a. Temporary tables can f |
| ilter data, reducing data orders of magnitude.                                                                                            |                                               |
| 3. STRING_MATCHING: (0.15) a. Existing grammatical structure: like 'abc%'.                                                                | 3. a. Rewrite LIKE %X into a |
| range query.                                                                                                                              |                                               |
+-------------------------------------------------------------------------------------------------------------------------------------------+-----------------------------------------------+
```

根因中说明了该语句中存在 **like 'abc%'**，其可能导致索引失效，并将以将 **like** 改写为 **range**。

（4） 按照建议将 **SQL** 改写成 **range** 语句：**select t2.c1, t1.c1 from t1, t2 where t2.c1 between 'abc' and 'abd' and t2.c2=t1.c2;**

（5） 执行改写语句，其执行计划为：

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
Hash Join  (cost=12759.92..287872.42 rows=3523 width=52) (actual time=147.384..2811.188 rows=11 loops=1)
  Hash Cond: (t1.c2 = t2.c2)
  ->  Seq Scan on t1  (cost=0.00..254032.47 rows=5611947 width=52) (actual time=0.007..1498.375 rows=10300003 loops=1)
  ->  Hash  (cost=12715.88..12715.88 rows=3523 width=66) (actual time=21.397..21.397 rows=3582 loops=1)
        Buckets: 32768  Batches: 1  Memory Usage: 343kB
        ->  Bitmap Heap Scan on t2  (cost=140.36..12715.88 rows=3523 width=66) (actual time=2.130..20.516 rows=3582 loops=1)
              Recheck Cond: ((c1 >= 'abc'::text) AND (c1 <= 'abd'::text))
              Heap Blocks: exact=3542
              ->  Bitmap Index Scan on t2_c1_idx  (cost=0.00..139.48 rows=3523 width=0) (actual time=1.715..1.715 rows=3582 loops=1)
                    Index Cond: ((c1 >= 'abc'::text) AND (c1 <= 'abd'::text))
Total runtime: 2811.519 ms
(11 rows)
```

可以发现此时索引没有失效，执行效率提升明显。

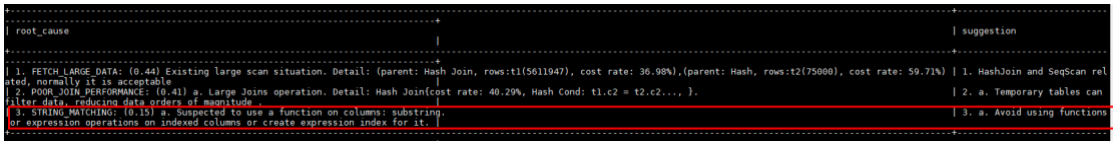
情况 2：

（1） 在场景一中执行 **SQL**（此时 **t2(c1)**索引存在）：select t1.c1, t2.c1 from t1, t2 where substring(t2.c1 from 2 for 4) = '85df' and t2.c2=t1.c2;执行计划为：

```
                                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------------
Hash Join  (cost=411123.53..686950.80 rows=75000 width=52) (actual time=6963.425..8038.800 rows=3 loops=1)
  Hash Cond: (t1.c2 = t2.c2)
  ->  Seq Scan on t1  (cost=0.00..254032.47 rows=5611947 width=52) (actual time=0.006..1443.034 rows=10300003 loops=1)
  ->  Hash  (cost=410186.03..410186.03 rows=75000 width=66) (actual time=5240.900..5240.900 rows=246 loops=1)
        Buckets: 131072  Batches: 1  Memory Usage: 24kB
        ->  Seq Scan on t2  (cost=0.00..410186.03 rows=75000 width=66) (actual time=0.053..5240.373 rows=246 loops=1)
              Filter: ("substring"(c1, 2, 4) = '85df'::text)
              Rows Removed by Filter: 14999756
Total runtime: 8039.135 ms
(9 rows)
```

此时 **t2** 中的索引未使用。

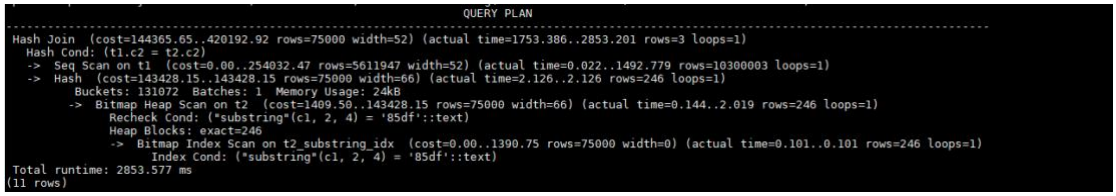
（2） 对该 SQL 使用诊断工具：



发现根因中说明 **substring(t2.c1 from 2 for 4)**可能导致索引失效。同时建议根据业务创建表达式索引。

（3） 创建表达式索引：CREATE INDEX ON t2 (substring(t2.c1 from 2 for 4));

（4） 再次执行 **SQL**，执行计划：



发现 **SQL** 性能提升明显。

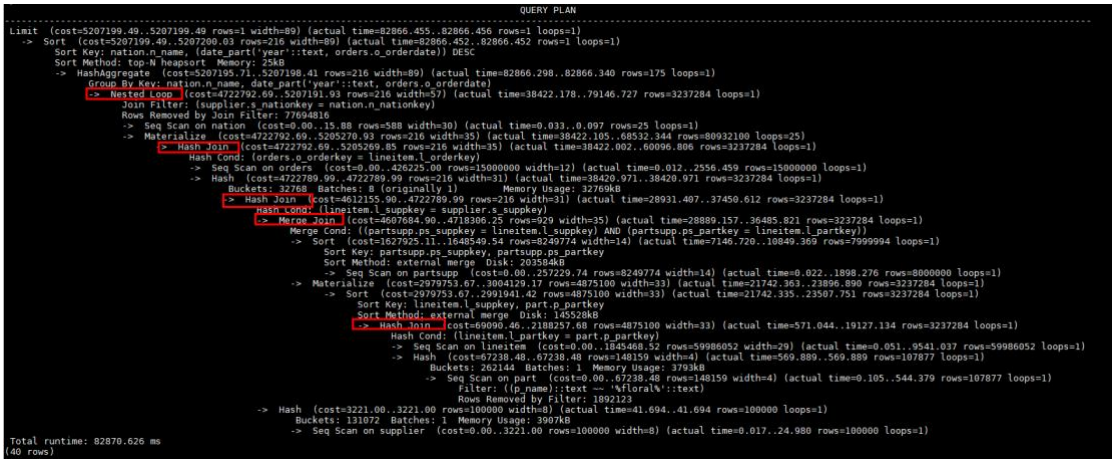
根因二十五 **COMPLEX_EXECUTION_PLAN**：典型场景为 **SQL** 中 **join** 或者子查询过多

1. 在 **tpch** 场景下执行 **SQL**：

```
select
        nation,
        o_year,
        sum(amount) as sum_profit
from
        (
                select
                        n_name as nation,
                        extract(year from o_orderdate) as o_year,
                        l_extendedprice * (1 - l_discount) - ps_supplycost *
l_quantity as amount
                from
                        part,
                        supplier,
                        lineitem,
                        partsupp,
                        orders,
                        nation
                where
                        s_suppkey = l_suppkey
                        and ps_suppkey = l_suppkey
                        and ps_partkey = l_partkey
                        and p_partkey = l_partkey
                        and o_orderkey = l_orderkey
                        and s_nationkey = n_nationkey
                        and p_name like '%floral%'
```

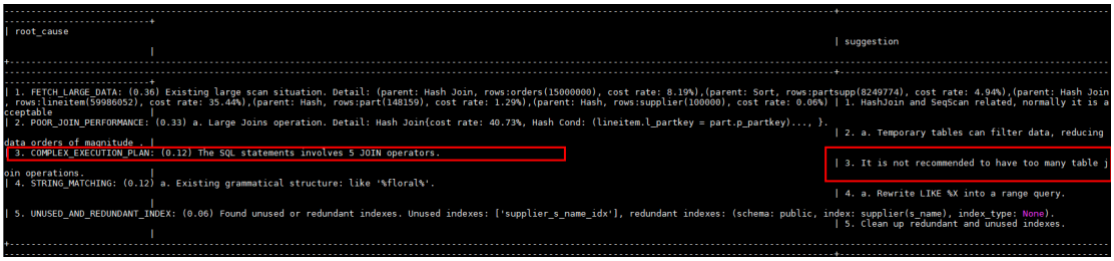
```
        ) as profit
    group by
        nation,
        o_year
    order by
        nation,
        o_year desc
LIMIT 1;
```

执行计划为：



发现执行计划中存在 5 个 join 算子。

2. 对该 SQL 执行诊断：

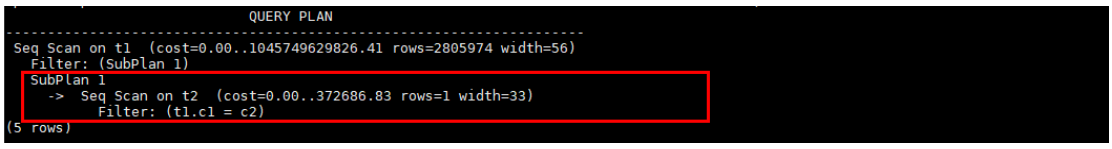


根因中指出该 SQL 包含 5 个 join 算子，一般建议一个 SQL 语句中不超过 2 个 join 算子。

根因二十六 CORRELATED_SUBQUERY：SQL 中存在子查询不能提升的场景（todo 为啥没改写成功）

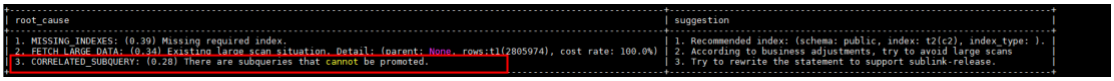
1. **在场景一中执行 SQL：** select * from t1 where t1.c1 in (select t2.c1 from t2 where t1.c1 = t2.c2)；

   其执行计划为：



   由于执行时间太长，因此此处不实际执行，但也可以看出其子查询不能提升，导致最终代价非常大。

2. **对该 SQL 执行诊断：**



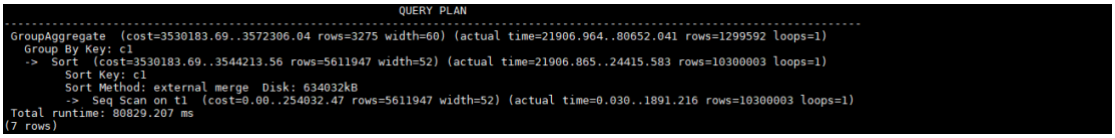
可以发现根因中说明了 **SQL** 中存在不能提升的子查询。

根因二十七 **POOR_AGGREGATION_PERFORMANCE**：**AGG** 算子代价较大，其主要包括以下几个情况，① **SQL** 结构不优导致优化器不能选择最优的 **AGG** 算子；②错误的参数设置（**enable_hashagg,enable_groupagg**）导致优化器不能选着正确的算子；③**AGG** 算子本生代价较大

情况一：

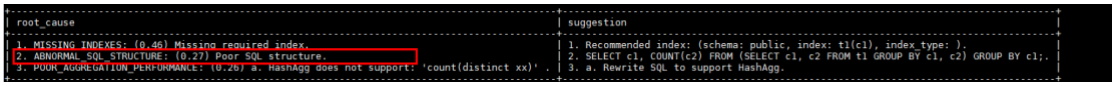
（1） 场景一中执行 **SQL:** select c1, count(distinct c2) from t1 group by c1;

其执行计划为：



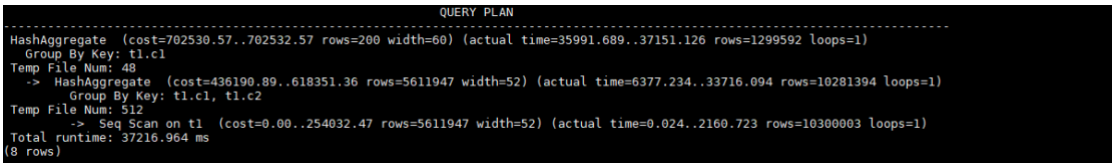
其中 **sort+GroupAgg** 算子代价较大。

（2） 对该 **SQL** 进行诊断，根因如下：



根因中说明了该 **SQL** 结构不优，并给出了对应的改写语句：SELECT c1, COUNT(c2) FROM (SELECT c1, c2 FROM t1 GROUP BY c1, c2) GROUP BY c1;

（3） 运行改写语句，执行计划为：



发现改写后性能提升较大。

情况二：

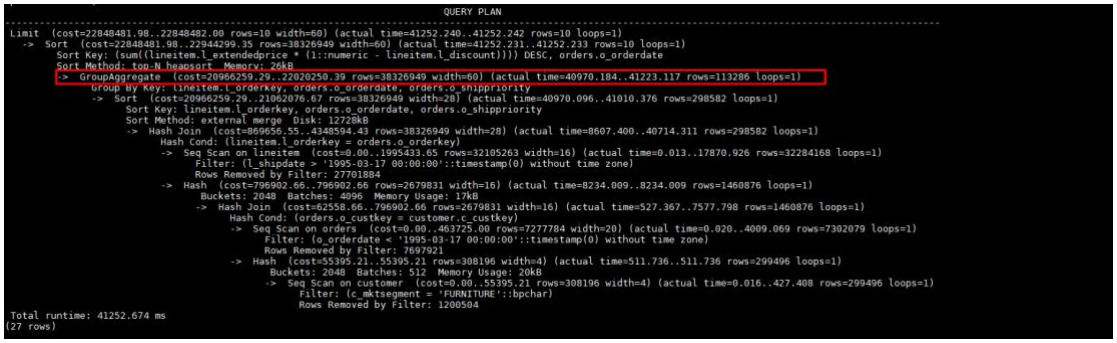
（1） 先手动设置 **enable_hashagg=off;**

（2） 在 **tpch** 场景下执行 **SQL:**

```
select
        l_orderkey,
        sum(l_extendedprice * (1 - l_discount)) as revenue,
        o_orderdate,
        o_shippriority
from
        customer,
        orders,
        lineitem
where
        c_mktsegment = 'FURNITURE'
        and c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and o_orderdate < date '1995-03-17'
```

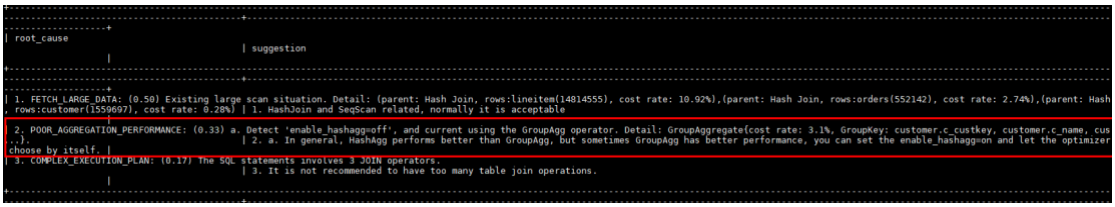
```
        and l_shipdate > date '1995-03-17'
group by
        l_orderkey,
        o_orderdate,
        o_shippriority
order by
        revenue desc,
        o_orderdate
LIMIT 10;
```

其执行计划为：



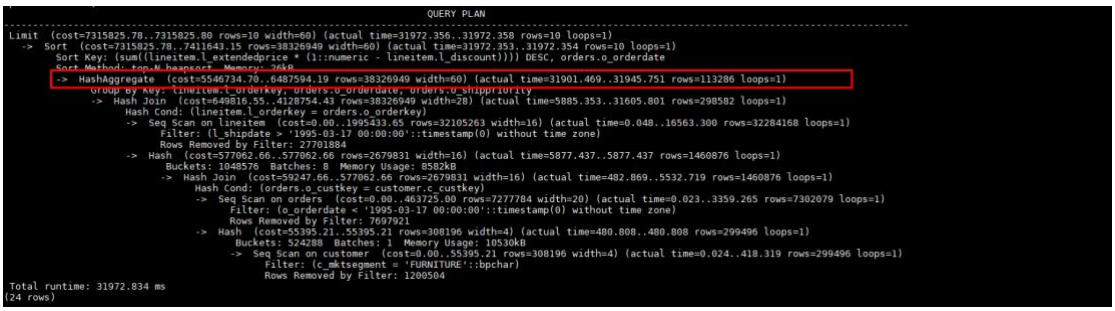
可以发现执行计划选择了 sort + GroupAgg 算子。

（3） **对该 SQL 进行诊断，根因如下：**



诊断发现 enable_hashagg 参数为 off，可能影响优化器执行计划选择。

（4） 将 enable_hashagg 设置成 on，并运行该 SQL，执行计划为：



此时优化器选择了 HashAGG 算子，执行效率提升较大。

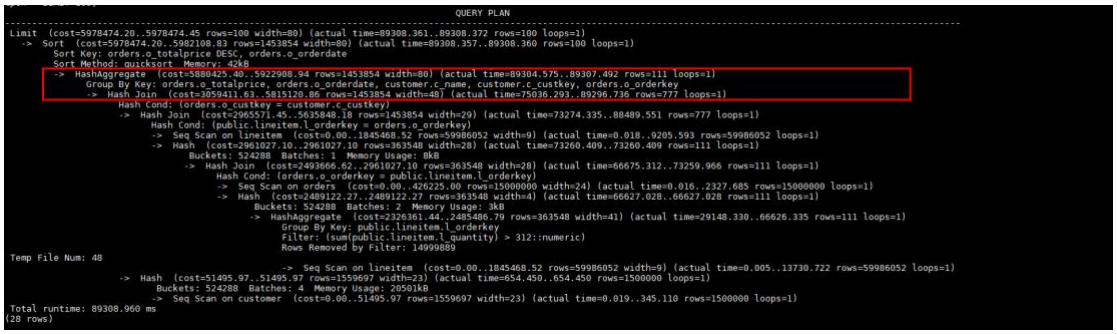
情况三：

（1） 在 tpch 场景下执行 SQL:

```
select
        c_name,
        c_custkey,
        o_orderkey,
        o_orderdate,
```

```
        o_totalprice,
        sum(l_quantity)
    from
        customer,
        orders,
        lineitem
    where
        o_orderkey in (
            select
                l_orderkey
            from
                lineitem
            group by
                l_orderkey having
                    sum(l_quantity) > 312
        )
        and c_custkey = o_custkey
        and o_orderkey = l_orderkey
    group by
        c_name,
        c_custkey,
        o_orderkey,
        o_orderdate,
        o_totalprice
    order by
        o_totalprice desc,
        o_orderdate
```

LIMIT 100; 执行计划为：



其 HashAGG 代价较大。

（2） 对该 SQL 执行诊断，结果为：

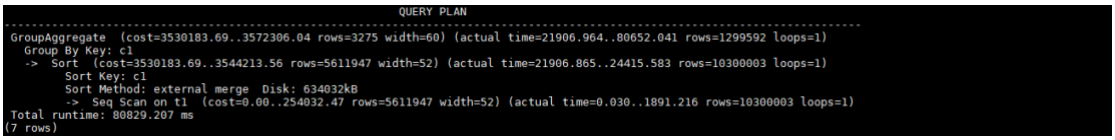
发现根因中指出 HashAGG 算子占整个代价的 49.47%。

根因二十八 ABNORMAL_SQL_STRUCTURE：SQL 结构不优（内核不支持优化场景），进而导致执行计划不优

1. **场景一中，测试 SQL：**

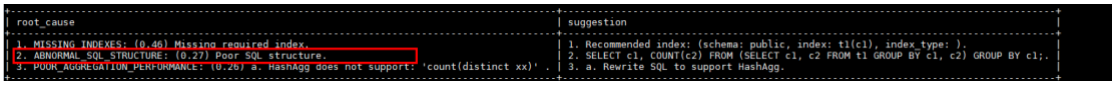
   select c1, count(distinct c2) from t1 group by c1;

2. **执行该 SQL，执行计划为：**



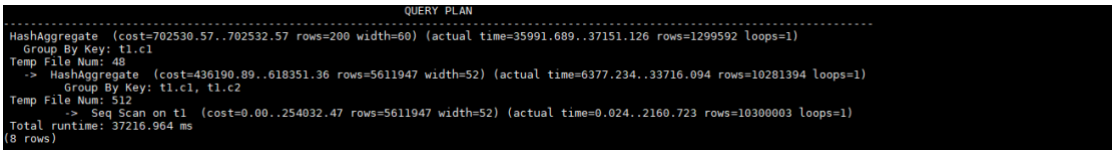
   我们发现该 SQL 走了 sort + GroupAGG 方式，导致代价较大（执行时间 80s）

3. **对该 SQL 执行慢 SQL 诊断，结果为：**



   其中第二个根因诊断出该 SQL 结构不优，第三条根因提出 HashAgg 不支持 count(distinct xx)结构，因此我们按照第二个根因的建议对 SQL 进行改写。

4. **执行改写的 SQL，其执行计划为：**



   最终 SQL 性能提升较大，执行时间缩短为 37s。

根因二十九 TIMED_TASK_CONFLICT：定时任务冲突

此场景当前不考虑