

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：

root_cause	suggestion
1. LOCK_CONTENTION: (0.49) SQL was blocked by: 'update t1 set c1='slowsqltest' where id between 100 and 150;'. 2. MISSING_INDEXES: (0.34) Missing required index. 3. LACK_STATISTIC_INFO: (0.17) Statistics not updated in time. Analyze delay since last update: public:t1(1357s).	1. Adjust the business reasonably to avoid lock blocking. 2. Recommended index: (schema: public, index: t1(id), index_type:). 3. Timely update statistics to help the planner choose the most suitable plan.

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(1,1000000),md5(random())::text),md5(random())::text);
3. 执行: update t1 set c1='adasdadsddfd' where id > 1000000;此时生成大量 dead tuples。
4. 测试语句: select * from t1 where c1 > 'aweefefr' and c1 < 'ytdsfddgsda';
清理死元组之前执行计划

```
tpch=> explain analyze select * from t1 where c1 > 'aweefefr' and c1 < 'ytdsfddgsda';  
QUERY PLAN  
Seq Scan on t1 (cost=0.00..478971.97 rows=5476139 width=70) (actual time=0.040..13564.924 rows=312817 loops=1)  
Filter: ((c1 > 'aweefefr'::text) AND (c1 < 'ytdsfddgsda'::text))  
Rows Removed by Filter: 9687183  
Total runtime: 13578.516 ms  
(4 rows)
```

5. 进行慢 SQL 根因诊断：

root_cause	suggestion
1. FETCH_LARGE_DATA: (0.52) Existing large scan situation. Detail: (parent: None, rows:t1(3127471), cost rate: 100.0%) 2. LACK_STATISTIC_INFO: (0.30) Statistics not updated in time. Analyze delay since last update: public:t1(728.0s). 3. MANY_DEAD_TUPLES: (0.17) Dead tuples affect SQL query performance. Detail: t1: live_tup(9999947) dead_tup(9000000) dead_rate(0.9)...	1. According to business adjustments, try to avoid large scans 2. Timely update statistics to help the planner choose the most suitable plan. 3. Clean up dead tuples in time to avoid affecting query performance.

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

```
tpch=> explain analyze select * from t1 where c1 > 'aweefefr' and c1 < 'ytdsfddgsda';  
QUERY PLAN  
Seq Scan on t1 (cost=0.00..233935.27 rows=2684707 width=70) (actual time=0.055..2522.879 rows=312817 loops=1)  
Filter: ((c1 > 'aweefefr'::text) AND (c1 < 'ytdsfddgsda'::text))  
Rows Removed by Filter: 9687183  
Total runtime: 2537.032 ms  
(4 rows)
```

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 < 'c32454fdfeigd';

2. 执行计划:

```
tpch=> explain analyze select c1, c2 from t1 where c2 > 'adsaf2324r' and c2 < 'c32454fdfeqdg';
QUERY PLAN
-----
Seq Scan on t1 (cost=0.00..347913.00 rows=754238 width=66) (actual time=0.028..3081.011 rows=748958 loops=1)
  Filter: ((c2 > 'adsaf2324r'::text) AND (c2 < 'c32454fdfeqdg'::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 performance. |
+-----+-----+
```

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：

- 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);
- 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';
- 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。

- 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。

- 对 SQL: insert into t1 select generate_series(10000000, 10100000),md5(random()::text), md5(random()::text);执行根因诊断:

root_cause	suggestion
1. INSERT_LARGE_DATA: (0.42) Insert a large number of tuples: t1(1000 rows). Insert a large number of tuples: t1(1000 rows).	1. No suggestions.
2. LACK_STATISTIC_INFO: (0.37) Statistics not updated in time. Analyze delay since last update: public:t1(4044.0s).	2. Timely update statistics to help the planner choose the most suitable plan.
3. UNUSED_AND_REDUNDANT_INDEX: (0.11) Found unused or redundant indexes. Unused indexes: {}, redundant indexes: {schema: public, index: t1(c1), index_type: None};(schema: public, index: t1(c2), index_type: None).	3. Clean up redundant and unused indexes.
4. TOO_MANY_INDEX: (0.13) Found a large number of indexes in the table. Detail: t1(c1).	4. Too many index will affect the speed of insert, delete, and update statement.

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=60) (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:

root_cause	suggestion
1. UPDATE_LARGE_DATA: (0.54) Update a large number of tuples: t1(2999861 rows).	1. Make adjustments to the business.
2. LACK_STATISTIC_INFO: (0.46) Statistics not updated in time. Analyze delay since last update: public:t1(1358s).	2. Timely update statistics to help the planner choose the most suitable plan.

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:

root_cause	suggestion
1. DELETE_LARGE_DATA: (0.53) Delete a large number of tuples: t1(2990904 rows).	1. Make adjustments to the business.
2. LACK_STATISTIC_INFO: (0.47) Statistics not updated in time. Analyze delay since last update: public:t1(1357s).	2. Timely update statistics to help the planner choose the most suitable plan.

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=10201394 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:

root_cause	suggestion
1. FETCH_LARGE_DATA: (0.61) Existing large scan situation. Detail: (parent: Sort, rows:t1(5611947), cost rate: 6.97%)	1. Find 'count' operation, try to avoid this behavior
2. DISK_SPILL: (0.39) Disk-Spill may occur during SQL sorting.	2. Analyze whether the business needs to adjust the size of the work_mem parameter.

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。

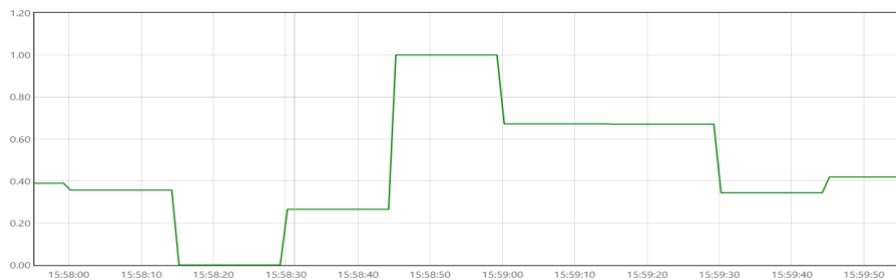
根因十一 `VACUUM_EVENT` 和根因十二 `ANALYZE_EVENT`：vacuum/analyze 操作会占用一定资源，影响 SQL 执行性能，当前不考虑

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);`

```
Insert on t1 (cost=0.00..15.02 rows=1000 width=68) (actual time=0.063..2610.852 rows=500001 loops=1)
-> Result (cost=0.00..15.02 rows=1000 width=0) (actual time=0.035..1008.554 rows=500001 loops=1)
Total runtime: 2611.022 ms
(3 rows)
```

发现执行时间 2.6s 左右

4. 运用慢 SQL 诊断，

root_cause	suggestion
1. IO_RESOURCE_CONTENTION: (1.00) a. The IO-Utils exceeds the threshold 0.7. 1. a. Detect whether processes outside the database compete for resources b. Check SLOW-SQL in database.	

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：

```
Insert on t1 (cost=0.00..15.02 rows=1000 width=68) (actual time=0.110..1675.865 rows=500001 loops=1)
-> Result (cost=0.00..15.02 rows=1000 width=0) (actual time=0.034..1008.049 rows=500001 loops=1)
Total runtime: 1675.993 ms
(3 rows)
```

此时 SQL 性能提升明显。

1. 在 TPCB 场景下执行 SQL:

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

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。

发现索引对 SQL 提升较大

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:


```

Limit (cost=17981871.33..17981871.34 rows=1 width=89) (actual time=159534.042..159534.043 rows=1 loops=1)
-> Sort (cost=17981871.33..17981871.87 rows=216 width=89) (actual time=159534.039..159534.039 rows=1 loops=1)
    Sort Key: nation.n_name, (date_part('year',::text, orders.o_orderdate)) DESC
    Sort Method: top-N heapsort Memory: 25kB
-> HashAggregate (cost=17981867.55..17981870.25 rows=216 width=89) (actual time=159533.929..159533.961 rows=175 loops=1)
    Group By Key: nation.n_name, date_part('year',::text, orders.o_orderdate)
-> Merge Join (cost=17986874.99..17981863.77 rows=216 width=57) (actual time=159494.503..155722.766 rows=3237284 loops=1)
    Merge Cond: (orders.o_orderkey = lineitem.l_orderkey)
-> Sort (cost=2583153.94..2020653.94 rows=15000000 width=12) (actual time=7125.113..8878.955 rows=15000000 loops=1)
    Sort Key: orders.o_orderkey
    Sort Method: external merge Disk: 323216kB
-> Seq Scan on orders (cost=0.00..426225.00 rows=15000000 width=12) (actual time=0.022..3866.430 rows=15000000 loops=1)
-> Sort (cost=14923713.56..14923714.10 rows=216 width=53) (actual time=143389.332..143912.129 rows=3237284 loops=1)
    Sort Key: lineitem.l_orderkey
    Sort Method: external sort Disk: 724656kB
-> Merge Join (cost=14923659.00..14923705.18 rows=216 width=53) (actual time=137740.036..139686.987 rows=3237284 loops=1)
    Merge Cond: (supplier.s_nationkey = nation.n_nationkey)
-> Sort (cost=14923656.07..14923656.61 rows=216 width=31) (actual time=137739.923..138570.804 rows=3237284 loops=1)
    Sort Key: supplier.s_nationkey
    Sort Method: external merge Disk: 142312kB
-> Merge Join (cost=14923133.90..14923647.70 rows=216 width=31) (actual time=134021.406..136187.830 rows=3237284 loops=1)
    Merge Cond: (lineitem.l_suppkey = supplier.s_suppkey)
-> Sort (cost=14911608.03..14911610.36 rows=929 width=35) (actual time=133974.676..135025.252 rows=3237284 loops=1)
    Sort Key: lineitem.l_suppkey
    Sort Method: external merge Disk: 154104kB
-> Merge Join (cost=14378493.51..14911562.24 rows=929 width=35) (actual time=76211.454..132009.492 rows=3237284 loops=1)
    Merge Cond: (partsupp.ps_partkey = part.o_partkey)
-> Merge Join (cost=14298354.49..14050042.61 rows=2663 width=45) (actual time=7562.064..12640.794 rows=59984980 loops=1)
    Merge Cond: ((lineitem.l_partkey = partsupp.ps_partkey) AND (lineitem.l_suppkey = partsupp.ps_suppkey))
-> Sort (cost=12670560.69..12820525.82 rows=59986052 width=29) (actual time=70510.142..96869.905 rows=59984980 loops=1)
    Sort Key: lineitem.l_partkey, lineitem.l_suppkey
    Sort Method: external merge Disk: 2461216kB
-> Seq Scan on lineitem (cost=0.00..1845468.52 rows=59986052 width=29) (actual time=0.031..14860.385 rows=59986052 loops=1)
-> Materialize (cost=1627925.11..1669173.98 rows=8249774 width=14) (actual time=5151.892..5878.613 rows=59989025 loops=1)
-> Sort (cost=1627925.11..1648549.54 rows=8249774 width=14) (actual time=5151.871..6590.507 rows=59989025 loops=1)
    Sort Key: partsupp.ps_partkey, partsupp.ps_suppkey
    Sort Method: external merge Disk: 203584kB
-> Seq Scan on partsupp (cost=0.00..257229.74 rows=8249774 width=14) (actual time=0.060..1741.257 rows=8000000 loops=1)
-> Sort (cost=79962.95..80333.35 rows=148159 width=4) (actual time=549.130..735.523 rows=3237248 loops=1)
    Sort Key: part.p_partkey
    Sort Method: quicksort Memory: 9815KB
-> Seq Scan on part (cost=0.00..67238.48 rows=148159 width=4) (actual time=0.142..524.022 rows=107877 loops=1)
    Filter: ((p.name)::text ~> '%floral%')
    Rows Removed by Filter: 1892123
-> Sort (cost=11525.82..11775.82 rows=100000 width=8) (actual time=46.707..194.693 rows=3238421 loops=1)
    Sort Key: supplier.s_suppkey
    Sort Method: quicksort Memory: 9323kB
-> Seq Scan on supplier (cost=0.00..3221.00 rows=100000 width=8) (actual time=0.083..24.274 rows=100000 loops=1)

Total runtime: 159534.835 ms
(53 rows)

```

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：

```

| root_cause
|
| suggestion
|
|-----|
| 1. FETCH_LARGE_DATA: (0.31) Existing large scan situation. Detail: (parent: Sort, rows:orders(15000000), cost rate: 2.37%),(parent: Sort, rows:lineitem(59986052), cost rate: 10.26%),(parent: Sort, rows:partsupp(8249774), cost rate: 1.43%),(parent: Sort, rows:part(148159), cost rate: 0.37%),(parent: Sort, rows:supplier(100000), cost rate: 0.02%)
|
| 2. POOR_JOIN_PERFORMANCE: (0.29) a. According to business adjustments, try to avoid large scans.
|
| 2. a. Detect 'enable hashjoin=off', you can set the enable hashjoin=on and let the optimizer choose by itself.
|
| 3. DISK_SPILL: (0.20) The SORT/HASH operation may spill to disk. Detail: (parent: Limit, rows: 216, cost rate: 0.04%),(parent: Merge Join, rows: 15000000, cost rate: 14.43%),(parent: Merge Join, rows: 216, cost rate: 0.04%),(parent: Merge Join, rows: 216, cost rate: 0.03%),(parent: Merge Join, rows: 929, cost rate: 0.03%),(parent: Merge Join, rows: 59986052, cost rate: 61.03%),(parent: Materialize, rows: 8249774, cost rate: 7.74%),(parent: Merge Join, rows: 148159, cost rate: 0.07%),(parent: Merge Join, rows: 100000, cost rate: 0.05%),(parent: Merge Join, rows: 588, cost rate: 0.05%)
|
| 4. COMPLEX_EXPRESSION_PLAN: (0.10) The SQL statements involves 5 JOIN operators.
|
| 5. STRING_MATCHING: (0.10) a. Existing grammatical structures: ~> '%floral%'.
|
| 5. a. Rewrite LIKE %X into a range query.
|
|-----|

```

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：

```

Limit (cost=5207199.49..5207199.49 rows=1 width=89) (actual time=81902.938..81902.938 rows=1 loops=1)
-> Sort (cost=5207200.83 rows=216 width=89) (actual time=81902.935..81902.935 rows=1 loops=1)
    Sort Key: nation.n_name, (date_part('year',::text, orders.o_orderdate)) DESC
    Sort Method: top-N heapsort Memory: 25kB
-> HashAggregate (cost=5207199.71..5207199.71 rows=216 width=89) (actual time=81902.798..81902.842 rows=175 loops=1)
    Group By Key: nation.n_name, date_part('year',::text, orders.o_orderdate)
-> Nested Loop (cost=4722792.69..5207191.93 rows=216 width=57) (actual time=38917.961..78213.021 rows=3237284 loops=1)
    Join Filter: (supplier.s_nationkey = nation.n_nationkey)
    Rows Removed by Join Filter: 77694816
-> Seq Scan on nation (cost=0.00..15.88 rows=588 width=30) (actual time=0.005..0.045 rows=25 loops=1)
-> Materialize (cost=4722792.69..5208270.99 rows=216 width=35) (actual time=38917.854..67500.885 rows=80932100 loops=25)
-> Hash Join (cost=4722792.69..5208269.85 rows=216 width=35) (actual time=38917.770..58991.054 rows=3237284 loops=1)
    Hash Cond: (orders.o_orderkey = lineitem.l_orderkey)
-> Seq Scan on orders (cost=0.00..426225.00 rows=15000000 width=12) (actual time=0.031..2556.562 rows=15000000 loops=1)
-> Hash (cost=4722789.99..4722789.99 rows=216 width=31) (actual time=38916.654..38916.654 rows=3237284 loops=1)
    Buckets: 32768 Batches: 8 (originally 1) Memory Usage: 32769kB
-> Hash Join (cost=4612155.90..4722789.99 rows=216 width=31) (actual time=29561.119..37948.454 rows=3237284 loops=1)
    Hash Cond: (lineitem.l_suppkey = supplier.s_suppkey)
-> Seq Scan on partsupp (cost=0.00..257229.74 rows=8249774 width=14) (actual time=0.019..1862.541 rows=8000000 loops=1)
-> Merge Join (cost=4607684.90..4718306.25 rows=929 width=35) (actual time=29517.610..36994.101 rows=3237284 loops=1)
    Merge Cond: ((partsupp.ps_suppkey = lineitem.l_suppkey) AND (partsupp.ps_partkey = lineitem.l_partkey))
-> Sort (cost=1627925.11..1648549.54 rows=8249774 width=14) (actual time=7307.372..10925.508 rows=9999994 loops=1)
    Sort Key: lineitem.l_suppkey, part.o_partkey
    Sort Method: external merge Disk: 203584kB
-> Seq Scan on partsupp (cost=0.00..1845468.52 rows=8249774 width=14) (actual time=0.019..1862.541 rows=8000000 loops=1)
-> Materialize (cost=2979753.67..3004129.17 rows=4875100 width=33) (actual time=22210.168..24349.594 rows=3237284 loops=1)
-> Sort (cost=2979753.67..2991941.42 rows=4875100 width=33) (actual time=22210.136..23961.526 rows=3237284 loops=1)
    Sort Key: lineitem.l_suppkey, part.o_partkey
    Sort Method: external merge Disk: 145528kB
-> Hash Join (cost=69090.46..2180257.68 rows=4875100 width=33) (actual time=585.977..19528.696 rows=3237284 loops=1)
    Hash Cond: (lineitem.l_partkey = part.p_partkey)
-> Seq Scan on lineitem (cost=0.00..1845468.52 rows=59986052 width=29) (actual time=0.057..9886.785 rows=59986052 loops=1)
-> Hash (cost=67238.48..67238.48 rows=148159 width=4) (actual time=584.645..584.645 rows=107877 loops=1)
    Buckets: 262144 Batches: 1 Memory Usage: 3793kB
-> Seq Scan on part (cost=0.00..67238.48 rows=148159 width=4) (actual time=0.172..555.484 rows=107877 loops=1)
    Filter: ((p.name)::text ~> '%floral%')
    Rows Removed by Filter: 1892123
-> Hash (cost=3221.00..3221.00 rows=100000 width=8) (actual time=42.954..42.954 rows=100000 loops=1)
    Buckets: 131072 Batches: 1 Memory Usage: 3097kB
-> Seq Scan on supplier (cost=0.00..3221.00 rows=100000 width=8) (actual time=0.018..24.841 rows=100000 loops=1)

Total runtime: 81906.568 ms

```

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:
```


d93wn3adhmoaczlW4w9oej9k9k','xnsclds32gt1t6ws4yla5z7cwaibna4qv','l0z7g6ut9n1g5b5
r16np4p88us069yw4','essx3vxyb8rsr5svr01ol4xwv69drwil','hpc6n32lxb3i2jeunjzltfjmci815
iy9','e8xmfexyz0r4v5dvl1axslc7z06e3vql','qilqvj63ekrjzfrxb7jqucgp76oywx7','6tdo7hasam
67823srw9y46y18k525lyr','50xm88rlo4rs4b0so9pdhwqz5w2fma9c','x12mt0tsh4id82pp0n
p034r2coqcjt5w','8qwhcf5rhygduy12ow02fp7twnuj72lo','2gm22d9rzibly6aw3spgzkadeuy
o6q4f','26jzvs5s6ok9ip2xn6o7qky95ljx5jo','5pi9hnnvyyx3gol1o9485s2g6arf7gcnv','bjmepp
f154r0wkswahye19qkuobh5kj','6g5vx15lnhwupk3zriyidmyoe78naru1','5ltt4d5qmi47ogz
pqkijwhrbv6wggogwda','zeiq1qht8wco8kpkp96h91oyp21rfnxz','hjo039u7k9hp7kfbj0aj5u6w
4r9hx4zv','d0cqmq3a9mvrml1kox9y0dppbpfxrt1o24','d3im71k85hah0k7x4vuxau7i9i6f94b6',
'35wa7cjl5hslhcupj33auogh9r1a1val','urcewbhrfrcroyt969aszumky0xrbyymh','t03k6ykgkrzp
dqj3n1swi102175mvi86','21gwzyi5gymqdi4si3p4g9qvwwaq6i4j','eazjy9842n324k9hs096xq
znib6kb2o9','eun4vx6yw2kvz8evjr2niix2cjm8jtr','2bmt46zc73tr9ewqzrm6bukrqgbaiwo8',
a4lmiipxvfhajyaencyekjvyp8y42ip','86m29taxlg2ijy2e1ib4gf89mntntx5zp','8s9zjffakti27fuof
w93egv7u6mccdnk','ct4njx95dkmse3slh3mdhwufajrb67l1','rt45rwqgdllq2zol8goyjghdq20c
fv2i','m933p3tf9x1f74qliev7p4hd1fi1f4kdo','6m2n5uh48r3191t6auovk131x1xbplqs','1yyiqs
l1paxj7sx16zep7sd86tuqqode','h7otxcktakysbmkmx1waormcm1zze7nf','beajnf9de2az4mhd
xs6nmpw747xjq2gv','4fo9dzra67kxscnxzio4qq691lnyscv','8wa6hsrinzmkljqao8kqisgg2820
l65e','hl200axi7gi1va71vetrttyzgug56au','zua9z2o88qichblf99ez1iz4zms6pxzc','fo1ril2u85
57qogj9394oh1bpenaqomj','fz2k1agwbmatw5reompbpx399v1qwsul','1sc3p5jfacm04v6a
madorvzeofubukka','1g5b4nkrmixcvbctwgrn8nh0luhu087','hnhptwyemtmejgy9trdr583
nph1t98','9dlf46firpjgsn2wzfzqmt1nzbX0mr2z','5hg5k0bdsl199o0eg2vhf1velrbjz3ny','pmp
h5uuuv8g24sygaljkc7mhp432n27p','e4sldi4oh8nit3mvk9v6p2pcbpb1sa1s','w8ar7ec3ncs31e
xblyuu70eg5duflxcr','42oq6r0f530401y0jlhren8gejrprpm5u','jyfvbqvxs2btsskl6leuljayyvgq
yg','pfrliktl93n5n45r0402re9lzkxici1','0jxswszqcoh6f0e5ah2647qg0hf5e046','9lRhjokxb79g
jid0ihh739nboib2q8h8','kp03oux48f2n0t7npgdmBu99449z8jtx7','c0ixuezekn8lifrfrjm7cneq
yzlrlcpf','si5nol75dkdn6k44uyen29juh4snz7up','blt0zurq01za2062pbk577r33vve71vm','1d
7a4375hw0j5ir3keh96zzfcig574f','a2yaoc377miinzcxm0qx804z1i868oat');

执行计划为：

```
See Scan on t2 (cost=0.00..73673.00 rows=499900 width=70) (actual time=0.021..1715.958 rows=500000 loops=1)
  Filter: (t2.c1 <= ALL ((0+sum(m2.f1rbjhyck3ack3dmsp7q36,mdxtxtk4zpmtyehdceprq14lbn,145gpg4zyf1t4cex38j0k1pnmk1br7,85clfi5wmp5jiziv6rjvc;sdh7753nk,glzleufydnas7;dtc1613kug4pc;9p6b5,5k71g5atpu18btv
  hmf91k1j569u2zn,nthrgrrfthmmpdqkx3zjydwv9v9,ad19pfc101aywbf7f9qut3m1fbsk6,91tk1p41jup1k1qccmw778mhty1y4,111vqk16dsk12wtet3arajau1hdt,1673fcmu9657qwr35c5rcy951paw7,chnptxtscfupx09fupg
  99wq48dp9,5dtfboas3ad7v01kkgcdygf7a61oh,161tk5a23erakadv7662vy37x0kzr,3rdkgueqduzp124bqce0t1qphk31vy,21y7fvey4126gxyt9ml2n3vlbgdml,hl18091pggc11y80xiurg0k0t146y,1y4e8fyt08b6balq0f0ky1qt06l
  cca,mdueedh9jnv32qym1vck0428k3r,rlv6p9glrlyfouf1qpmec2056t2vay,gcrcmmyjy2t5726132zccmwad1f0,dwvqy1r1m0y1f0q63kcdwdf1cz,0d16d6dbm3j1um2yqubafubay,huwye53mqf12dml1yvezukc;pg42p,zst
  zbn471jxbvhyj1lhaldzq3rol,968a8ar5d9qgh3dugdc1u0f018gg,3mumad9d09mwa0mqehfaybpb0b,vg06w7fy6gdrdnog74qjtplc5los,njbmbr3218wb152aj281pguic9lks,wgraxhjm0p1yrtf0k4ny164uxwvi,1f8d14mh
  b6dgl1dzaaky15g8sqi,3hd0dwmyel1td1251ym05135kr4ias,qabzr02vwy4154ybwuqql1zvhnrc,r14e00a02m7n1m0dapi3hfi0mqv1o,76pkqghk41j0hm55xmb1934621ksh,em572bapzruffa077zuzmyf,033te18ky0jzcxkn9
  fhpqpm1ul1qz,4y4yhd93h3adhmec1w4w6y9k8k,msc16j2jttf0w5y1657cwnh3adv,10z7gub01pbb1fmpg08hu069w4,es03xvay8r352vrdl14w66dml,hpc6n32lxb3i2jeunjzltfjmci815iy9,edctwxy0r4v5dvl1axslc7
  z063vql,q1lqv163krjzfr7jqucgp76oywx7,6tdo7hasam7823srw9y46y18k525lyr,50xm88rlo4rs4b0so9pdhwqz5w2fma9c,x12mt0tsh4id82pp0n34r2coqcjt5w,8qwhcf5rhygduy12ow02fp7twnuj72lo,2gm22d9rzibly6aw3spgzkadeuy06q4f
  ,26jzvs5s6ok9ip2xn6o7qky95ljx5jo,5pi9hnnvyyx3gol1o9485s2g6arf7gcnv,bjmeppf154r0wkswahye19qkuobh5kj,6g5vx15lnhwupk3zriyidmyoe78naru1,5ltt4d5qmi47ogzpqkijwhrbv6wggogwda,zeiq1qht8wco8kpkp96h91oyp21rfnxz,hjo039u7k9hp7kfbj0aj5u6w4r9hx4zv,d0cqmq3a9mvrml1kox9y0dppbpfxrt1o24,d3im71k85hah0k7x4vuxau7i9i6f94b6,35wa7cjl5hslhcupj33auogh9r1a1val,urcewbhrfrcroyt969aszumky0xrbyymh,t03k6ykgkrzpqdqj3n1swi102175mvi86,21gwzyi5gymqdi4si3p4g9qvwwaq6i4j,eazjy9842n324k9hs096xqznib6kb2o9,eun4vx6yw2kvz8evjr2niix2cjm8jtr,2bmt46zc73tr9ewqzrm6bukrqgbaiwo8,a4lmiipxvfhajyaencyekjvyp8y42ip,86m29taxlg2ijy2e1ib4gf89mntntx5zp,8s9zjffakti27fuofw93egv7u6mccdnk,ct4njx95dkmse3slh3mdhwufajrb67l1,rt45rwqgdllq2zol8goyjghdq20cfv2i,m933p3tf9x1f74qliev7p4hd1fi1f4kdo,6m2n5uh48r3191t6auovk131x1xbplqs,1yyiqsl1paxj7sx16zep7sd86tuqqode,h7otxcktakysbmkmx1waormcm1zze7nf,beajnf9de2az4mhdxs6nmpw747xjq2gv,4fo9dzra67kxscnxzio4qq691lnyscv,8wa6hsrinzmkljqao8kqisgg2820l65e,hl200axi7gi1va71vetrttyzgug56au,zua9z2o88qichblf99ez1iz4zms6pxzc,fo1ril2u8557qogj9394oh1bpenaqomj,fz2k1agwbmatw5reompbpx399v1qwsul,1sc3p5jfacm04v6amadorvzeofubukka,1g5b4nkrmixcvbctwgrn8nh0luhu087,hnhptwyemtmejgy9trdr583nph1t98,9dlf46firpjgsn2wzfzqmt1nzbX0mr2z,5hg5k0bdsl199o0eg2vhf1velrbjz3ny,pmp
  h5uuuv8g24sygaljkc7mhp432n27p,e4sldi4oh8nit3mvk9v6p2pcbpb1sa1s,w8ar7ec3ncs31exblyuu70eg5duflxcr,42oq6r0f530401y0jlhren8gejrprpm5u,jyfvbqvxs2btsskl6leuljayyvgqyg,pfrliktl93n5n45r0402re9lzkxici1,0jxswszqcoh6f0e5ah2647qg0hf5e046,9lRhjokxb79gjid0ihh739nboib2q8h8,kp03oux48f2n0t7npgdmBu99449z8jtx7,c0ixuezekn8lifrfrjm7cneqyzlrlcpf,si5nol75dkdn6k44uyen29juh4snz7up,blt0zurq01za2062pbk577r33vve71vm,1d7a4375hw0j5ir3keh96zzfcig574f,a2yaoc377miinzcxm0qx804z1i868oat));
Total runtime: 1750.646 ms
(3 rows)
```

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：

root cause	suggestion
1. FETCH_LARGE_DATA: (0.85) Existing large scan situation. Detail: (parent: None, rows:t1(4999902), cost rate: 100.0%)	1. According to business adjustments, try to avoid large scans
2. COMPLEX_BOOLEAN_EXPRESSIONS: (0.15) Large IN-Clause, length is 100. Detail: 'bxmqent2f1rbjhyck3ack3dmsp7q36','uq4g.... 1. Rewrite large in-clause as a constant subquery or temporary table and rep	

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 hashjoin 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. RETU LARGE DATA: (0.40) 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 related, 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)...), b. STRING MATCHING: (0.15) a. Existing grammatical structure: like 'abc%'.	2. a. Temporary tables can filter data, reducing data orders of magnitude. 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: 24kB
      -> 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 substr(t2.c1 from 2 for 4) = '85df' and t2.c2=t1.c2;执行计划为：

```

QUERY PLAN
-----
Hash Join (cost=41123.53..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: ('substr'(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:

root_cause	suggestion
1. FETCH LARGE DATA: (0.44) Existing large scan situation. Detail: (parent: Hash Join, rows:t1(5611947), cost rate: 36.98%),(parent: Hash, rows:t2(75000), cost rate: 59.71%) 1. HashJoin and SeqScan related, normally it is acceptable	
2. HASH JOIN PERFORMANCE: (0.41) a. Large Joins operation. Detail: Hash Join(cost rate: 40.29%, Hash Cond: t1.c2 = t2.c2...,).	2. a. Temporary tables can filter data, reducing data orders of magnitude.
3. STRING MATCHING: (0.15) a. Suspected to use a function on columns: substring.	3. a. Avoid using functions or expression operations on indexed columns or create expression index for it.

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:

QUERY PLAN
Hash Join (cost=144365.65..420192.92 rows=75000 width=52) (actual time=1753.386..2853.201 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=0.022..1492.779 rows=10300003 loops=1) -> Hash (cost=143428.15..143428.15 rows=75000 width=66) (actual time=2.126..2.126 rows=246 loops=1) Buckets: 131072 Batches: 1 Memory Usage: 24kB -> Bitmap Heap Scan on t2 (cost=1489.58..143428.15 rows=75000 width=66) (actual time=0.144..2.019 rows=246 loops=1) Recheck Conds: (*substring*(c1, 2, 4) = '85df'::text) Heap Blocks: exact=246 -> Bitmap Index Scan on t2_substring_idx (cost=0.00..1390.75 rows=75000 width=0) (actual time=0.101..0.101 rows=246 loops=1) Index Cond: (*substring*(c1, 2, 4) = '85df'::text) Total runtime: 2853.577 ms (11 rows)

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:

```

QUERY PLAN
-----
Limit (cost=5207199.49..5207199.49 rows=1 width=89) (actual time=82866.455..82866.456 rows=1 loops=1)
-> Sort (cost=5207199.49..5207200.03 rows=216 width=89) (actual time=82866.452..82866.452 rows=1 loops=1)
    Sort Key: nation_n_name, (date_part('year'::text, orders_o_orderdate)) DESC
    Sort Method: top-N heapsort  Memory: 258K
-> HashAggregate (cost=5207195.71..5207198.41 rows=216 width=89) (actual time=82866.298..82866.340 rows=175 loops=1)
    Group By key: nation_n_name, date_part('year'::text, orders_o_orderdate)
    <- Nested Loop (cost=4222192.69..5207191.93 rows=216 width=31) (actual time=38422.178..79146.727 rows=3237284 loops=1)
        Join Filter: (supplier_s_nationkey = nation_n_nationkey)
        Rows Removed by Join Filter: 77694816
        >- Seq Scan on nation (cost=0.00..15.88 rows=588 width=30) (actual time=0.033..0.097 rows=25 loops=1)
        >- Hash Join (cost=4722792.69..5205270.93 rows=216 width=35) (actual time=38422.105..68532.344 rows=80932100 loops=25)
            Hash Cond: (orders_o_orderkey = lineitem_l_orderkey)
            >- Seq Scan on orders (cost=0.00..426225.00 rows=15000000 width=12) (actual time=0.012..2556.459 rows=15000000 loops=1)
            >- Hash (cost=4722789.99..4722789.99 rows=216 width=31) (actual time=38420.971..38420.971 rows=3237284 loops=1)
                Buckets: 32768  Batches: 8 (originally 1)  Memory Usage: 32768kB
                >- Hash Join (cost=4612155.99..4722789.99 rows=216 width=31) (actual time=28931.407..37450.612 rows=3237284 loops=1)
                    Hash Cond: (lineitem_l_supplier_s = supplier_s_supplier_s)
                    <- Merge Join (cost=4607684.99..4713306.25 rows=229 width=35) (actual time=28889.157..36485.821 rows=3237284 loops=1)
                        Merge Cond: ((partsupp_p_s_supplier_s = lineitem_l_supplier_s) AND (partsupp_p_s_partkey = lineitem_l_partkey))
                        >- Sort (cost=1627925.11..1648540.54 rows=8249774 width=14) (actual time=7146.720..10849.369 rows=7999994 loops=1)
                            Sort Key: partsupp_p_s_supplier_s, partsupp_p_s_partkey
                            Sort Method: external merge  Disk: 203584kB
                            >- Seq Scan on partsupp (cost=0.00..257229.74 rows=8249774 width=14) (actual time=0.022..1898.276 rows=8800800 loops=1)
                            >- Materialize (cost=3979753.67..3804129.17 rows=4875100 width=33) (actual time=21742.363..23896.890 rows=3237284 loops=1)
                                Sort Key: lineitem_l_supplier_s, part_p_partkey
                                Sort Method: external merge  Disk: 16528kB
                                >- Hash Join (cost=69090.46..2188257.68 rows=4875100 width=33) (actual time=571.044..19127.134 rows=3237284 loops=1)
                                    Hash Cond: (lineitem_l_partkey = part_p_partkey)
                                    >- Seq Scan on lineitem (cost=0.00..1845468.52 rows=59986052 width=29) (actual time=0.051..9541.037 rows=59986052 loops=1)
                                    >- Hash (cost=67238.48..67238.48 rows=148159 width=4) (actual time=569.889..569.889 rows=107877 loops=1)
                                        Buckets: 262144  Batches: 1  Memory Usage: 3793kB
                                        >- Seq Scan on part (cost=0.00..67238.48 rows=148159 width=4) (actual time=0.105..544.379 rows=107877 loops=1)
                                            Filter: ((p_name)::text = 'floralV'::text)
                                            Rows Removed by Filter: 1892123
                                >- Hash (cost=3221.00..3221.00 rows=100000 width=8) (actual time=41.694..41.694 rows=100000 loops=1)
                                    Buckets: 131072  Batches: 1  Memory Usage: 3907kB
                                    >- Seq Scan on supplier (cost=0.00..3221.00 rows=100000 width=8) (actual time=0.017..24.980 rows=100000 loops=1)
Total runtime: 82870.626 ms
(40 rows)

```

发现执行计划中存在 5 个 join 算子。

2. 对该 SQL 执行诊断:

```

root_cause | suggestion
-----|-----
1. FETCH_LARGE_DATA: (0.36) Existing large scan situation. Detail: (parent: Hash Join, rows:orders(15000000), cost rate: 8.19%),(parent: Sort, rows:partsupp(8249774), cost rate: 4.94%),(parent: Hash Join, rows:lineitem(59986052), cost rate: 35.44%),(parent: Hash, rows:part(148159), cost rate: 1.29%),(parent: Hash, rows:supplier(100000), cost rate: 0.06%) | 1. HashJoin and SeqScan related, normally it is a acceptable
2. POOR_JOIN_PERFORMANCE: (0.33) a. Large Joins operation. Detail: Hash Join(cost rate: 40.73%, Hash Cond: (lineitem_l_partkey = part_p_partkey)... ). | 2. a. Temporary tables can filter data, reducing data orders of magnitude
3. COMPLEX_EXECUTION_PLAN: (0.12) The SQL statements involves 5 JOIN operators. | 3. It is not recommended to have too many table
4. STRING_MATCHING: (0.12) a. Existing grammatical structure: like 'floralV'. | 4. a. Rewrite LIKE %X into a range query.
5. UNUSED_AND_REDUNDANT_INDEX: (0.06) Found unused or redundant indexes. Unused indexes: ['supplier_s_name_idx'], redundant indexes: (schema: public, index: supplier(s_name), index_type: None). | 5. Clean up redundant and unused indexes.

```

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:

```

QUERY PLAN
-----
Seq Scan on t1 (cost=0.00..1045749629826.41 rows=2805974 width=56)
Filter: (SubPlan 1)
SubPlan 1
-> Seq Scan on t2 (cost=0.00..372686.83 rows=1 width=33)
Filter: (t1.c1 = c2)
(5 rows)

```

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:

root_cause	suggestion
1. MISSING INDEXES: (0.30) Missing required index.	1. Recommended index: (schema: public, index: t2(c2), index_type:).
2. FETCH LARGE DATA: (0.34) Existing large scan situation. Detail: (parent: None, rows:t1(2005974), cost rate: 100.0%)	2. According to business adjustments, try to avoid large scans
3. CORRELATED SUBQUERY: (0.28) There are subqueries that cannot be promoted.	3. Try to rewrite the statement to support sublink-release.

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；③AGG 算子本身代价较大

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：

```

QUERY PLAN
-----
GroupAggregate (cost=3530183.69..3572306.04 rows=3275 width=60) (actual time=21906.964..80652.041 rows=1299592 loops=1)
  Group By Key: c1
    -> Sort (cost=3530183.69..3544213.56 rows=5611947 width=52) (actual time=21906.865..24415.583 rows=10300003 loops=1)
        Sort Key: c1
        Sort Method: external merge  Disk: 634032kB
        -> Seq Scan on t1 (cost=0.00..254032.47 rows=5611947 width=52) (actual time=0.030..1891.216 rows=10300003 loops=1)
Total runtime: 80829.207 ms
(7 rows)

```

其中 sort+GroupAgg 算子代价较大。

- （2） 对该 SQL 进行诊断，根因如下：

root_cause	suggestion
1. MISSING INDEXES: (0.46) Missing required index.	1. Recommended index: (schema: public, index: t1(c1), index_type:).
2. ABNORMAL SQL STRUCTURE: (0.27) Poor SQL structure.	2. SELECT c1, COUNT(c2) FROM (SELECT c1, c2 FROM t1 GROUP BY c1, c2) GROUP BY c1;
3. POOR_AGGREGATION_PERFORMANCE: (0.26) a. HashAgg does not support: 'count(distinct xx)'	3. a. Rewrite SQL to support HashAgg.

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：

```

QUERY PLAN
-----
HashAggregate (cost=702530.57..702532.57 rows=200 width=60) (actual time=35991.689..37151.126 rows=1299592 loops=1)
  Group By Key: t1.c1
  Temp File Num: 48
    -> HashAggregate (cost=436190.89..618351.36 rows=5611947 width=52) (actual time=6377.234..33716.094 rows=10281394 loops=1)
        Group By Key: t1.c1, t1.c2
        Temp File Num: 512
        -> Seq Scan on t1 (cost=0.00..254032.47 rows=5611947 width=52) (actual time=0.024..2160.723 rows=10300003 loops=1)
Total runtime: 37216.964 ms
(8 rows)

```

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:

```

```

QUERY PLAN
-----
Limit (cost=22848481.98..22848482.00 rows=10 width=60) (actual time=41252.240..41252.242 rows=10 loops=1)
-> Sort (cost=22848481.98..22944299.35 rows=38326949 width=60) (actual time=41252.231..41252.233 rows=10 loops=1)
    Sort Key: (sum((lineitem.l_extendedprice * (1::numeric - lineitem.l_discount)))) DESC, orders.o_orderdate
    Sort Method: top-N heapsort  Memory: 2648
-> GroupAggregate (cost=20966259.29..22020250.39 rows=38326949 width=60) (actual time=40970.184..41223.117 rows=113286 loops=1)
    Group by Key: lineitem.l_orderkey, orders.o_orderdate, orders.o_shippriority
    Sort Key: lineitem.l_orderkey, orders.o_orderdate, orders.o_shippriority
    Sort Method: external merge  Disk: 12728kB
    Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
    Seq Scan on lineitem (cost=0.00..1995433.65 rows=32105263 width=16) (actual time=0.013..17870.926 rows=32284168 loops=1)
    Filter: (l_shipdate > '1995-03-17 00:00:00'::timestamp(0) without time zone)
    Rows Removed by Filter: 27701884
    Hash Cond: (orders.o_custkey = customer.c_custkey)
    Seq Scan on orders (cost=0.00..463725.00 rows=7277784 width=20) (actual time=0.020..4009.069 rows=7302079 loops=1)
    Filter: (o_orderdate < '1995-03-17 00:00:00'::timestamp(0) without time zone)
    Rows Removed by Filter: 7697921
    Hash Cond: (c_mktsegment = 'FURNITURE'::bpchar)
    Seq Scan on customer (cost=0.00..55395.21 rows=308196 width=4) (actual time=0.016..427.408 rows=299496 loops=1)
    Filter: (c_mktsegment = 'FURNITURE'::bpchar)
    Rows Removed by Filter: 1200504
Total runtime: 41252.674 ms
(27 rows)

```

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：

```

root_cause
+-----+
| suggestion |
+-----+
1. FETCH_LARGE_DATA: (0.50) Existing large scan situation. Detail: (parent: Hash Join, rows:lineitem[14814555], cost rate: 10.92%),(parent: Hash Join, rows:orders[552142], cost rate: 2.74%),(parent: Hash Join, rows:customer[1559697], cost rate: 0.28%) | 1. HashJoin and SeqScan related, normally it is acceptable
2. POOR_AGGREGATION_PERFORMANCE: (0.33) a. Detect 'enable_hashagg=off', and current using the GroupAgg operator. Detail: GroupAggregate(cost rate: 3.1%, GroupKey: customer.c_custkey, customer.c_name, customer.c_mktsegment) | 2. a. In general, HashAgg performs better than GroupAgg, but sometimes GroupAgg has better performance, you can set the enable_hashagg=on and let the optimizer choose by itself.
3. COMPLEX_EXECUTION_PLAN: (0.17) The SQL statements involves 3 JOIN operators. | 3. It is not recommended to have too many table join operations.

```

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：

```

QUERY PLAN
-----
Limit (cost=7315825.78..7315825.80 rows=10 width=60) (actual time=31972.356..31972.358 rows=10 loops=1)
-> Sort (cost=7315825.78..7411643.15 rows=38326949 width=60) (actual time=31972.353..31972.354 rows=10 loops=1)
    Sort Key: (sum((lineitem.l_extendedprice * (1::numeric - lineitem.l_discount)))) DESC, orders.o_orderdate
    Sort Method: top-N heapsort  Memory: 2648
-> HashAggregate (cost=546734.76..6487594.10 rows=38326949 width=60) (actual time=31901.469..31945.751 rows=113286 loops=1)
    Group by Key: lineitem.l_orderkey, orders.o_orderdate, orders.o_shippriority
    Hash Cond: (lineitem.l_orderkey = orders.o_orderkey)
    Seq Scan on lineitem (cost=0.00..1995433.65 rows=32105263 width=16) (actual time=0.048..16563.300 rows=32284168 loops=1)
    Filter: (l_shipdate > '1995-03-17 00:00:00'::timestamp(0) without time zone)
    Rows Removed by Filter: 27701884
    Hash Cond: (orders.o_custkey = customer.c_custkey)
    Seq Scan on orders (cost=0.00..463725.00 rows=7277784 width=20) (actual time=0.023..3359.265 rows=7302079 loops=1)
    Filter: (o_orderdate < '1995-03-17 00:00:00'::timestamp(0) without time zone)
    Rows Removed by Filter: 7697921
    Hash Cond: (c_mktsegment = 'FURNITURE'::bpchar)
    Seq Scan on customer (cost=0.00..55395.21 rows=308196 width=4) (actual time=0.024..418.319 rows=299496 loops=1)
    Filter: (c_mktsegment = 'FURNITURE'::bpchar)
    Rows Removed by Filter: 1200504
Total runtime: 31972.834 ms
(24 rows)

```

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:

```

```

QUERY PLAN
-----
Limit (cost=5978474.20..5978474.45 rows=100 width=80) (actual time=89308.361..89308.372 rows=100 loops=1)
-> Sort (cost=5978474.20..5982108.83 rows=1453854 width=80) (actual time=89308.357..89308.360 rows=100 loops=1)
    Sort Key: orders.o_totalprice DESC, orders.o_orderdate
    Sort Method: quicksort  Memory: 424kB
-> HashAggregate (cost=3888429.40..5922988.04 rows=1453854 width=80) (actual time=89308.375..89309.402 rows=111 loops=1)
    Group By Key: orders.o_totalprice, orders.o_orderdate, customer.c_name, customer.c_custkey, orders.o_orderkey
    -> Hash Join (cost=3059411.63..5815120.85 rows=1453854 width=48) (actual time=75036.293..89296.736 rows=777 loops=1)
        Hash Cond: (orders.o_custkey = customer.c_custkey)
        -> Hash Join (cost=2965571.45..5635848.18 rows=1453854 width=29) (actual time=73274.335..88489.551 rows=777 loops=1)
            Hash Cond: (public.lineitem.l_orderkey = orders.o_orderkey)
            -> Seq Scan on lineitem (cost=0.00..1845468.52 rows=59986052 width=9) (actual time=0.018..9205.593 rows=59986052 loops=1)
            -> Hash (cost=2961027.19..2961027.19 rows=363548 width=28) (actual time=73260.409..73260.409 rows=111 loops=1)
                Buckets: 524288 Batches: 1 Memory Usage: 0kB
                -> Hash Join (cost=2483866.42..2961027.19 rows=363548 width=28) (actual time=66675.312..73259.956 rows=111 loops=1)
                    Hash Cond: (orders.o_orderkey = public.lineitem.l_orderkey)
                    -> Seq Scan on orders (cost=0.00..426225.00 rows=15000000 width=24) (actual time=0.016..2327.485 rows=15000000 loops=1)
                    -> Hash (cost=2480122.21..2480122.27 rows=363548 width=14) (actual time=66627.028..66627.628 rows=111 loops=1)
                        Buckets: 524288 Batches: 2 Memory Usage: 3kB
                        -> HashAggregate (cost=2328361.44..2485466.79 rows=363548 width=14) (actual time=29140.330..66626.335 rows=111 loops=1)
                            Group By Key: public.lineitem.l_orderkey
                            Filter: (sum(public.lineitem.l_quantity) > 312::numeric)
                            Rows Removed by Filter: 14990089
        Temp File Num: 48
        -> Hash (cost=51495.97..51495.97 rows=1559697 width=23) (actual time=0.005..13730.722 rows=59986052 loops=1)
            Buckets: 524288 Batches: 4 Memory Usage: 20501kB
            -> Seq Scan on customer (cost=0.00..51495.97 rows=1559697 width=23) (actual time=0.019..345.110 rows=15000000 loops=1)
Total runtime: 89308.960 ms
(28 rows)

```

其 HashAGG 代价较大。

(2) 对该 SQL 执行诊断, 结果为:

```

+-----+
| root_cause | suggestion |
+-----+
+-----+
| 1. FETCH_LARGE_DATA: (0.34) Existing large scan situation. Detail: (parent: Hash Join, rows=lineitem(59986052), cost rate: 38.67%, (parent: Hash Join, rows=orders(15000000), cost rate: 7.13%), (parent: HashAggregate, rows=lineitem(59986052), cost rate: 38.87%), (parent: Hash, rows=customer(1559697), cost rate: 0.86%) | 1. HashJoin and SeqScan related, normally it is acceptable |
| 2. POOR_JOIN_PERFORMANCE: (0.32) a. Large Joins operation. Detail: HashJoin(cost rate: 47.66%, Hash Cond: (orders.o_custkey = customer.c_custkey)...), HashJoin(cost rate: 44.74%, Hash Cond: (public.lineitem.l_orderkey = orders.o_orderkey)...), | 2. a. Temporary tables can filter data, reducing data orders of magnitude |
| 3. POOR_AGGREGATION_PERFORMANCE: (0.23) a. The HashAgg operator cost is too expensive. Detail: HashAggregate(cost rate(include children): 49.47%, GroupKey: orders.o_totalprice, orders.o_orderdate,...), | 3. a. If the number of group keys or NDV is large, the hash table may be larger and lead to spill to disk. |
| 4. COMPLEX_EXECUTION_PLAN: (0.11) The SQL statements involves 3 JOIN operators. | 4. It is not recommended to have too many table join operations. |
+-----+

```

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：

```
QUERY PLAN
-----
GroupAggregate (cost=3530183.69..3572306.04 rows=3275 width=60) (actual time=21906.964..80652.041 rows=1299592 loops=1)
  Group By Key: c1
    -> Sort (cost=3530183.69..3544213.56 rows=5611947 width=52) (actual time=21906.865..24415.583 rows=10300003 loops=1)
        Sort Key: c1
        Sort Method: external merge  Disk: 634032kB
        -> Seq Scan on t1 (cost=0.00..254032.47 rows=5611947 width=52) (actual time=0.030..1891.216 rows=10300003 loops=1)
Total runtime: 80829.207 ms
(7 rows)
```

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：

root cause	suggestion
1. MISSING INDEXES: [0.46] Missing required index.	1. Recommended index: (schema: public, index: t1(c1), index_type:).
2. ABNORMAL SQL STRUCTURE: [0.27] Poor SQL structure.	2. SELECT c1, COUNT(c2) FROM (SELECT c1, c2 FROM t1 GROUP BY c1, c2) GROUP BY c1;.
3. HASH_AGG_NOT_SUPPORTED: [0.27] HashAgg does not support: 'count(distinct xx)'.	3. a. Rewrite SQL to support HashAgg.

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：

```
QUERY PLAN
-----
HashAggregate (cost=702530.57..702532.57 rows=200 width=60) (actual time=35991.689..37151.126 rows=1299592 loops=1)
  Group By Key: t1.c1
  Temp File Num: 48
    -> HashAggregate (cost=436190.89..618351.36 rows=5611947 width=52) (actual time=6377.234..33716.094 rows=10281394 loops=1)
        Group By Key: t1.c1, t1.c2
        Temp File Num: 512
        -> Seq Scan on t1 (cost=0.00..254032.47 rows=5611947 width=52) (actual time=0.024..2160.723 rows=10300003 loops=1)
Total runtime: 37216.964 ms
(8 rows)
```

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