数据仓库服务

性能调优

文档版本	10
发布日期	2024-04-26

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。
本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

安全声明

漏洞处理流程

华为公司对产品漏洞管理的规定以“漏洞处理流程”为准，该流程的详细内容请参见如下网址：
https://www.huawei.com/cn/psirt/vul-response-process
如企业客户须获取漏洞信息，请参见如下网址：
https://securitybulletin.huawei.com/enterprise/cn/security-advisory

目 录

1 优化查询性能概述

性能调优是数据库应用开发和迁移过程中的关键步骤，在整个项目实施过程中占据很大的份量。通过性能调优可以提高数据库的资源利用率，降低业务成本，还可以大大降低应用系统的运行风险，提高系统稳定性，给客户带来更大的价值。

SQL调优的唯一目的是“资源利用最大化”，即CPU、内存、磁盘IO、网络IO四种资源利用最大化。所有调优手段都是围绕资源使用开展的。所谓资源利用最大化是指SQL语句尽量高效，节省资源开销，以最小的代价实现最大的效益。比如做典型点查询的时候，可以用seqscan+filter（即读取每一条元组和点查询条件进行匹配）实现，也可以通过indexscan实现，显然indexscan可以以更小的代价实现相同的效果。

本章通过介绍性能调优最基本的数据库命令ANALYZE和EXPLAIN，来详细解读EXPLAIN展示的数据库执行计划，介绍如何通过执行计划了解数据库的执行过程、识别性能瓶颈，针对性调优。另外，通过介绍性能参数、典型应用场景、SQL诊断、SQL性能调优和SQL改写案例等性能调优的实际操作，为数据库性能调优提供全方位的参考指导。

2 Query 执行流程

SQL引擎从接受SQL语句到执行SQL语句需要经历的步骤如图2-1和表2-1所示。其中，红色字体部分为DBA可以介入实施调优的环节。

图 2-1 SQL 引擎执行查询类 SQL 语句的流程

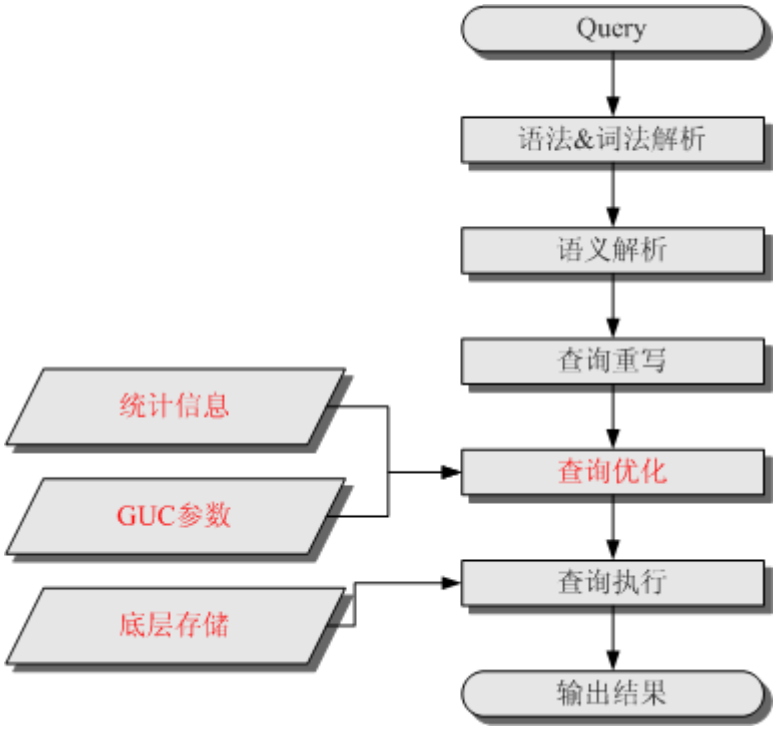


表 2-1 SQL 引擎执行查询类 SQL 语句的步骤说明

步骤	说明
1、语法&词法解析	按照约定的SQL语句规则，把输入的SQL语句从字符串转化为格式化结构(Stmt)。
2、语义解析	将“语法&词法解析”输出的格式化结构转化为数据库可以识别的对象。

步骤	说明
3、查询重写	根据规则把“语义解析”的输出等价转化为执行上更为优化的结构。
4、查询优化	根据“查询重写”的输出和数据库内部的统计信息规划SQL语句具体的执行方式，也就是执行计划。统计信息和GUC参数对查询优化（执行计划）的影响，请参见**调优手段之统计信息**和**调优手段之GUC参数**。
5、查询执行	根据“查询优化”规划的执行路径执行SQL查询语句。底层存储方式的选择合理性，将影响查询执行效率。详见**调优手段之底层存储**。

调优手段之统计信息

GaussDB(DWS)优化器是典型的基于代价的优化 (Cost-Based Optimization，简称CBO)。在这种优化器模型下，数据库根据表的元组数、字段宽度、NULL记录比率、distinct值、MCV值、HB值等表的特征值，以及一定的代价计算模型，计算出每一个执行步骤的不同执行方式的输出元组数和执行代价(cost)，进而选出整体执行代价最小/首元组返回代价最小的执行方式进行执行。这些特征值就是统计信息。从上面描述可以看出统计信息是查询优化的核心输入，准确的统计信息将帮助优化器选择最合适的查询规划，一般来说通过ANALYZE语法收集整个表或者表的若干个字段的统计信息，周期性地运行ANALYZE，或者在对表的大部分内容做了更改之后马上运行它是个好习惯。

调优手段之 GUC 参数

查询优化的主要目的是为查询语句选择高效的执行方式。

如下SQL语句:

```
SELECT count(1)
FROM customer inner join store_sales on (ss_customer_sk = c_customer_sk);
```

在执行customer inner join store_sales的时候，GaussDB(DWS)支持Nested Loop、Merge Join和Hash Join三种不同的Join方式。优化器会根据表customer和表store_sales的统计信息估算结果集的大小以及每种join方式的执行代价，然后对比选出执行代价最小的执行计划。

正如前面所说，执行代价计算都是基于一定的模型和统计信息进行估算，当因为某些原因代价估算不能反映真实的cost的时候，就需要通过guc参数设置的方式让执行计划倾向更优规划。

调优手段之底层存储

GaussDB(DWS)的表支持行存表、列存表，底层存储方式的选择严格依赖于客户的具体业务场景。一般来说计算型业务查询场景（以关联、聚合操作为主）建议使用列存表；点查询、大批量UPDATE/DELETE业务场景适合行存表。

对于每种存储方式还有对应的存储层优化手段，这部分会在后续的调优章节深入介绍。

调优手段之 SQL 重写

除了上述干预SQL引擎所生成执行计划的执行性能外，根据数据库的SQL执行机制以及大量的实践发现，有些场景下，在保证客户业务SQL逻辑的前提下，通过一定规则由DBA重写SQL语句，可以大幅度的提升SQL语句的性能。

这种调优场景对DBA的要求比较高，需要对客户业务有足够的了解，同时也需要扎实的SQL语句基本功，后续会介绍几个常见的SQL改写场景。

3 SQL 执行计划

SQL执行计划是一个节点树，显示GaussDB(DWS)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，也叫作一个执行算子。

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。

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，关于更多用法请参见**EXPLAIN语法**说明。

- 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。

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。

因此，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。

执行计划中的常见关键字说明：

1. 表访问方式
   - Seq Scan/CStore Scan

     全表顺序扫描。最基本的扫描算子，用于行/列存表的顺序扫描。
   - Index Scan/CStore Index Scan

     行/列存表的索引扫描。行/列存表上存在索引，条件列为索引列。

     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对应行。独立地

抓取数据行比顺序地读取数据的开销高很多，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，其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，需要索引扫描获取位图后再到基表上扫描。

- 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。

2. 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

     归并连接或融合连接，是先将关联表的关联列各自做排序，然后从各自的排序表中抽取数据，到另一个排序表中做匹配。

     因为Merge join需要做更多的排序，所以消耗的资源更多，因此通常情况下执行性能差于Hash Join。 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Merge Join的性能优于Hash Join。

3. 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。

执行计划显示格式

GaussDB(DWS)对执行计划提供了normal、pretty、summary、run四种显示格式。通过设置GUC参数explain_perf_mode，可以显示不同格式的执行计划。

- normal：代表使用默认的打印格式。图3-1中即为此显示格式。

  图 3-1 normal 格式执行计划示例

```
postgres=# explain select * from test where a < 1;
                            QUERY PLAN
------------------------------------------------------------------
 Streaming (type: GATHER)  (cost=0.25..19.16 rows=7 width=8)
   Node/s: All datanodes
   ->  Seq Scan on test  (cost=0.00..13.16 rows=7 width=8)
         Filter: (a < 1)
(4 rows)
```

- pretty：代表使用GaussDB(DWS)改进后的新显示格式。新的格式层次清晰，计划包含了plan node id，性能分析简单直接。如图3-2。

  图 3-2 pretty 格式执行计划示例

```
postgres=# explain select cjxh, count(1) from dwcjk group by cjxh;
 id |              operation               | E-rows | E-memory | E-width | E-costs
----+--------------------------------------+--------+----------+---------+---------
  1 | ->  Row Adapter                      |      1 |          |      52 | 58.42
  2 |    ->  Vector Streaming (type: GATHER) |      1 |          |      52 | 58.42
  3 |       ->  Vector Hash Aggregate      |      1 | 16MB     |      52 | 58.02
  4 |          ->  CStore Scan on dwcjk    |      1 | 1MB      |      44 | 58.00
(4 rows)
```

- summary：是在pretty的基础上增加了对打印信息的分析。
- run：在summary的基础上，将统计的信息输出到csv格式的文件中，以便于进一步分析。

常见类型计划

GaussDB(DWS)中当前主要存在三类分布式计划：

- FQS(fast query shipping)计划

  CN直接将原语句下发到DN，各DN单独执行，并将执行结果在CN上进行汇总。

- Stream计划

  CN根据原语句生成计划并将计划下发给DN进行执行，各DN执行过程中使用Stream算子进行数据交互。

- Remote-Query计划

  CN生成计划后，将部分原语句下发到DN，各DN单独执行，执行后将结果发送给CN，CN执行剩余计划。

现有表tt01和tt02定义如下：

```
CREATE TABLE tt01(c1 int, c2 int) DISTRIBUTE BY hash(c1);
CREATE TABLE tt02(c1 int, c2 int) DISTRIBUTE BY hash(c2);
```

类型一：FQS计划，完全下推

两表JOIN，且其连接条件为各表的分布列，在关闭stream算子的情况下，CN会直接将该语句发送至各DN执行，最后结果在CN汇总。

```
SET enable_stream_operator=off;
SET explain_perf_mode=normal;

EXPLAIN (VERBOSE on,COSTS off) SELECT * FROM tt01,tt02 WHERE tt01.c1=tt02.c2;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Data Node Scan on "__REMOTE_FQS_QUERY__"
   Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
   Node/s: All datanodes
   Remote query: SELECT tt01.c1, tt01.c2, tt02.c1, tt02.c2 FROM dbadmin.tt01, dbadmin.tt02 WHERE tt01.c1
= tt02.c2
(4 rows)
```

类型二：非FQS计划，部分语句下推

两表JOIN，且连接条件中包含非分布列，此时在关闭stream算子的情况下，CN会将基表扫描语句下发至各DN，然后在CN上进行JOIN。

```
SET enable_stream_operator=off;
SET explain_perf_mode=normal;

EXPLAIN (VERBOSE on,COSTS off) SELECT * FROM tt01,tt02 WHERE tt01.c1=tt02.c1;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Hash Join
   Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
   Hash Cond: (tt01.c1 = tt02.c1)
   ->  Data Node Scan on tt01 "_REMOTE_TABLE_QUERY_"
         Output: tt01.c1, tt01.c2
         Node/s: All datanodes
         Remote query: SELECT c1, c2 FROM ONLY dbadmin.tt01 WHERE true
   ->  Hash
         Output: tt02.c1, tt02.c2
         ->  Data Node Scan on tt02 "_REMOTE_TABLE_QUERY_"
               Output: tt02.c1, tt02.c2
               Node/s: All datanodes
               Remote query: SELECT c1, c2 FROM ONLY dbadmin.tt02 WHERE true
(13 rows)
```

类型三：Stream计划，DN之间无数据交换

两表JOIN，且连接条件为各表的分布列，因此各DN无需数据交换。CN生成stream计划后，将除Gather Stream的计划下发给DN执行，在各个DN上进行基表扫描，并进行哈希连接后，发送给CN。

```
SET enable_fast_query_shipping=off;
SET enable_stream_operator=on;

EXPLAIN (VERBOSE on,COSTS off) SELECT * FROM tt01,tt02 WHERE tt01.c1=tt02.c2;
                    QUERY PLAN
---------------------------------------------------
 Streaming (type: GATHER)
   Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
   Node/s: All datanodes
   ->  Hash Join
         Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
         Hash Cond: (tt01.c1 = tt02.c2)
         ->  Seq Scan on dbadmin.tt01
               Output: tt01.c1, tt01.c2
               Distribute Key: tt01.c1
         ->  Hash
               Output: tt02.c1, tt02.c2
               ->  Seq Scan on dbadmin.tt02
                     Output: tt02.c1, tt02.c2
                     Distribute Key: tt02.c2
(14 rows)
```

类型四：Stream计划，DN之间存在数据交换

两表JOIN，且连接条件包含非分布列，在开启stream算子(SET enable_stream_operator=on)的情况下，会生成stream计划，其DN间存在数据交换。此时对于tt02表，会在各DN进行基表扫描，扫描后会通过Redistribute Stream算子，按照JOIN条件中的tt02.c1进行哈希计算后重新发送给各DN，然后在各DN上做JOIN，最后汇总到CN。

```
postgres=> SET enable_stream_operator=on;
SET
postgres=> SET enable_fast_query_shipping=off;
SET
postgres=> SET explain_perf_mode=normal;
SET
postgres=> EXPLAIN (VERBOSE on,COSTS off) SELECT * FROM tt01,tt02 WHERE tt01.c1=tt02.c1;
                      QUERY PLAN
-------------------------------------------------
 Streaming (type: GATHER)
   Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
   Node/s: All datanodes
   ->  Hash Join
         Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
         Hash Cond: (tt02.c1 = tt01.c1)
         ->  Streaming(type: REDISTRIBUTE)
               Output: tt02.c1, tt02.c2
               Distribute Key: tt02.c1
               Spawn on: All datanodes
               Consumer Nodes: All datanodes
               ->  Seq Scan on dbadmin.tt02
                     Output: tt02.c1, tt02.c2
                     Distribute Key: tt02.c2
         ->  Hash
               Output: tt01.c1, tt01.c2
               ->  Seq Scan on dbadmin.tt01
                     Output: tt01.c1, tt01.c2
                     Distribute Key: tt01.c1
(19 rows)
```

类型五：Remote-Query计划场景

因unship_func不能下推，且不满足部分下推要求（子查询下推），所以只能发送基表扫描的语句到DN，将基表数据收集到CN上来计算。

```
postgres=> CREATE FUNCTION unship_func(integer,integer) returns integer
postgres-> AS 'select $1 + $2;'
postgres-> LANGUAGE SQL volatile
postgres-> returns null on null input;
CREATE FUNCTION
```

```
postgres=> SET explain_perf_mode=pretty;
SET
postgres=> EXPLAIN VERBOSE SELECT unship_func(tt01.c1,tt01.c2) FROM tt01 JOIN tt02 on tt01.c1=tt02.c1;
                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 id |                      operation                        | E-rows | E-distinct | E-width | E-costs
----+-------------------------------------------------------+--------+------------+---------+---------
  1 | ->  Hash Join (2,3)                                   |     30 |            |       8 | 0.86
  2 |    ->  Data Node Scan on tt01 "_REMOTE_TABLE_QUERY_"  |     30 |            |       8 | 0.00
  3 |    ->  Hash                                           |     30 |            |       4 | 0.00
  4 |       ->  Data Node Scan on tt02 "_REMOTE_TABLE_QUERY_" |     30 |            |       4 | 0.00

             SQL Diagnostic Information
 --------------------------------------------------
 SQL is not plan-shipping
         reason: Function unship_func() can not be shipped

 Predicate Information (identified by plan id)
 ---------------------------------------------
   1 --Hash Join (2,3)
         Hash Cond: (tt01.c1 = tt02.c1)

            Targetlist Information (identified by plan id)
 ---------------------------------------------------------------------
   1 --Hash Join (2,3)
         Output: (tt01.c1 + tt01.c2)
   2 --Data Node Scan on tt01 "_REMOTE_TABLE_QUERY_"
         Output: tt01.c1, tt01.c2
         Node/s: All datanodes
         Remote query: SELECT c1, c2 FROM ONLY dbadmin.tt01 WHERE true
   3 --Hash
         Output: tt02.c1
   4 --Data Node Scan on tt02 "_REMOTE_TABLE_QUERY_"
         Output: tt02.c1
         Node/s: All datanodes
         Remote query: SELECT c1 FROM ONLY dbadmin.tt02 WHERE true

 ====== Query Summary =====
 -------------------------
 Parser runtime: 0.055 ms
 Planner runtime: 0.528 ms
 Unique SQL Id: 1780774145
(37 rows)
```

EXPLAIN PERFORMANCE 详解

在SQL调优过程中经常需要执行EXPLAIN ANALYZE或EXPLAIN PERFORMANCE查看SQL语句实际执行信息，通过对比实际执行与优化器的估算之间的差别来为优化提供依据。EXPLAIN PERFORMANCE相对于EXPLAIN ANALYZE增加了每个DN上的执行信息。

表定义如下：

```
CREATE TABLE tt01(c1 int, c2 int) DISTRIBUTE BY hash(c1);
CREATE TABLE tt02(c1 int, c2 int) DISTRIBUTE BY hash(c2);
```

以如下SQL查询语句为例：

```
SELECT * FROM tt01,tt02 WHERE tt01.c1=tt02.c2;
```

执行EXPLAIN PERFORMANCE输出的显示执行信息分为以下8个部分：

1. 执行计划

```
                                                   QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------
 id |                operation                 |     A-time      | A-rows | E-rows | E-distinct | Peak Memory  | E-memory | A-width | E-width | E-costs
----+------------------------------------------+-----------------+--------+--------+------------+--------------+----------+---------+---------+---------
  1 | ->  Streaming (type: GATHER)             | 2.566           |      0 |     30 |            | 24KB         |          |         |      16 | 36.59
  2 |    ->  Hash Join (3,4)                   | [0.007, 0.009]  |      0 |     30 |            | [8KB, 8KB]   | 1MB      |         |      16 | 28.59
  3 |       ->  Seq Scan on dbadmin.tt01       | [0.002, 0.003]  |      0 |     30 | 14         | [16KB, 16KB] | 1MB      |         |       8 | 14.14
  4 |       ->  Hash                           | [0,0]           |      0 |     29 | 14         | [0, 0]       | 16MB     |         |       8 | 14.14
  5 |          ->  Seq Scan on dbadmin.tt02    | [0,0]           |      0 |     30 |            | [0, 0]       | 1MB      |         |       8 | 14.14
```

   以表格的形式将计划显示出来，包含有11个字段，分别是：id、operation、A-time、A-rows、E-rows、E-distinct、Peak Memory、E-memory、A-width、E-width和E-costs。字段含义如下**表3-1**。

表 3-1 执行字段说明

字段	描述
id	执行算子节点编号。
operation	具体的执行节点算子名称。 Vector前缀的算子是指向量化执行引擎算子，一般出现含有列存表的Query中。 Streaming是一个特殊的算子，它实现了分布式架构的核心数据shuffle功能，Streaming共有三种形态，分别对应了分布式结构下不同的数据shuffle功能： ● Streaming (type: GATHER)：作用是coordinator从DN收集数据。 ● Streaming(type: REDISTRIBUTE)：作用是DN根据选定的列把数据重分布到所有的DN。 ● Streaming(type: BROADCAST)：作用是把当前DN的数据广播给其他所有的DN。
A-time	各DN相应算子执行时间，一般DN上执行的算子的A-time是由[]括起来的两个值，分别表示此算子在所有DN上完成的最短时间和最长时间，包括下层算子执行时间。 注意：在整个计划中，除了叶子节点的执行时间是算子本身的执行时间，其余算子的执行时间均包含子节点的执行时间。
A-rows	表示相应算子输出的全局总行数。
E-rows	每个算子估算的输出行数。
E-distinct	表示hashjoin算子的distinct估计值。
Peak Memory	此算子在每个DN上执行时使用的内存峰值，[]中左侧为最小值，右侧为最大值。
E-memory	DN上每个算子估算的内存使用量，只有DN上执行的算子会显示。某些场景会在估算的内存使用量后使用括号显示该算子在内存资源充足下可以自动扩展的内存上限。
A-width	表示当前算子每行元组的实际宽度，仅对于重内存使用算子会显示，包括：(Vec)HashJoin、(Vec)HashAgg、(Vec)HashSetOp、(Vec)Sort、(Vec)Materialize算子等，其中(Vec)HashJoin计算的宽度是其右子树算子的宽度，会显示在其右子树上。
E-width	每个算子输出元组的估算宽度。

字段	描述
E-costs	每个算子估算的执行代价。 • E-costs是优化器根据成本参数定义的单位来衡量的，习惯上以磁盘页面抓取为1个单位， 其它开销参数将参照它来设置。 • 每个节点的开销（E-costs值）包括它的所有子节点的开销。 • 开销只反映了优化器关心的东西，并没有把结果行传递给客户端的时间考虑进去。虽然这个时间可能在实际的总时间里占据相当重要的分量，但是被优化器忽略了，因为它无法通过修改规划来改变。

2. SQL Diagnostic Information

   SQL自诊断信息。优化和执行过程中识别到的性能优化点，当对DML语句进行带VERBOSE属性的EXPLAIN（EXPLAIN PERFORMANCE内置自带VERBOSE属性）时，SQL自诊断信息也会输出，以辅助性能问题定位。

3. Predicate Information (identified by plan id)

```
Predicate Information (identified by plan id)
-----------------------------------------------
  2 --Hash Join (3,4)
        Hash Cond: (tt01.c1 = tt02.c2)
  3 --Seq Scan on dbadmin.tt01
        Filter: (tt01.c1 >= 202007)
  5 --Seq Scan on dbadmin.tt02
        Filter: (tt02.c2 >= 202007)
```

   谓词过滤这部分主要显示的是对应执行算子节点的过滤条件，即在整个计划执行过程中不会变的信息，主要是一些join条件和一些filter信息。

4. Memory Information (identified by plan id)

```
          Memory Information (identified by plan id)
------------------------------------------------------------------
Coordinator Query Peak Memory:
        Query Peak Memory: 2MB
DataNode Query Peak Memory
        dn_6001_6002 Query Peak Memory: 0MB
        dn_6003_6004 Query Peak Memory: 0MB
        dn_6005_6006 Query Peak Memory: 0MB
  1 --Streaming (type: GATHER)
        Peak Memory: 56KB, Estimate Memory: 512MB
  2 --Hash Join (3,4)
        dn_6001_6002 Peak Memory: 8KB, Estimate Memory: 1024KB
        dn_6003_6004 Peak Memory: 8KB, Estimate Memory: 1024KB
        dn_6005_6006 Peak Memory: 8KB, Estimate Memory: 1024KB
  3 --Seq Scan on dbadmin.tt01
        dn_6001_6002 Peak Memory: 32KB, Estimate Memory: 1024KB
        dn_6003_6004 Peak Memory: 32KB, Estimate Memory: 1024KB
        dn_6005_6006 Peak Memory: 32KB, Estimate Memory: 1024KB
  4 --Hash
        dn_6001_6002 Buckets: 0  Batches: 0  Memory Usage: 0kB
        dn_6003_6004 Buckets: 0  Batches: 0  Memory Usage: 0kB
        dn_6005_6006 Buckets: 0  Batches: 0  Memory Usage: 0kB
```


内存使用信息这部分显示的是整个计划中会将内存的使用情况打印出来的算子的内存使用信息，主要是Hash、Sort算子，包括算子峰值内存（peak memory），优化器预估的内存（estimate memory），控制内存（control memory），估算内存使用（operator memory），执行时实际宽度（width），内存使用自动扩展次数（auto spread num），是否提前下盘（early spilled），以及下盘信息，包括重复下盘次数（spill Time(s)），内外表下盘分区数（inner/outer partition spill num），下盘文件数（temp file num），下盘数据量及最小和最大分区的下盘数据量（written disk IO [min, max] ）。其中sort算子不会显示具体的下盘文件数，仅在显示排序方法时显示Disk。

5. Targetlist Information (identified by plan id)

```
  Targetlist Information (identified by plan id)
--------------------------------------------------------
  1 --Streaming (type: GATHER)
        Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
        Node/s: All datanodes
  2 --Hash Join (3,4)
        Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
  3 --Seq Scan on dbadmin.tt01
        Output: tt01.c1, tt01.c2
        Distribute Key: tt01.c1
  4 --Hash
        Output: tt02.c1, tt02.c2
  5 --Seq Scan on dbadmin.tt02
        Output: tt02.c1, tt02.c2
        Distribute Key: tt02.c2
```

这一部分显示的是每一个算子对应的输出目标列信息。

6. DataNode Information (identified by plan id)

```
                  Datanode Information (identified by plan id)
-------------------------------------------------------------------------------------
 1 --Streaming (type: GATHER)
       (actual time=12.913..12.913 rows=0 loops=1)
       (Buffers: shared hit=1)
       (CPU: ex c/r=0, ex row=0, ex cyc=645657, inc cyc=645657)
 2 --Hash Join (3,4)
       dn_6001_6002 (actual time=0.006..0.006 rows=0 loops=1) (projection time=0.000)
       dn_6003_6004 (actual time=0.007..0.007 rows=0 loops=1) (projection time=0.000)
       dn_6005_6006 (actual time=0.006..0.006 rows=0 loops=1) (projection time=0.000)
       dn_6001_6002 (Buffers: 0)
       dn_6003_6004 (Buffers: 0)
       dn_6005_6006 (Buffers: 0)
       dn_6001_6002 (CPU: ex c/r=0, ex row=0, ex cyc=231, inc cyc=296)
       dn_6003_6004 (CPU: ex c/r=0, ex row=0, ex cyc=266, inc cyc=326)
       dn_6005_6006 (CPU: ex c/r=0, ex row=0, ex cyc=252, inc cyc=308)
 3 --Seq Scan on dbadmin.tt01
       dn_6001_6002 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
       dn_6003_6004 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
       dn_6005_6006 (actual time=0.001..0.001 rows=0 loops=1) (filter time=0.000)
       dn_6001_6002 (Buffers: 0)
       dn_6003_6004 (Buffers: 0)
       dn_6005_6006 (Buffers: 0)
       dn_6001_6002 (CPU: ex c/r=0, ex row=0, ex cyc=65, inc cyc=65)
       dn_6003_6004 (CPU: ex c/r=0, ex row=0, ex cyc=60, inc cyc=60)
       dn_6005_6006 (CPU: ex c/r=0, ex row=0, ex cyc=56, inc cyc=56)
```

这部分将各个算子的执行时间（若包含过滤及投影也会显示对应的执行时间）、CPU、buffer的使用情况全部打印出来。

- 算子执行信息

```
dn_6001_6002 (actual time=0.006..0.006 rows=0 loops=1) (projection time=0.000)
dn_6003_6004 (actual time=0.007..0.007 rows=0 loops=1) (projection time=0.000)
dn_6005_6006 (actual time=0.006..0.006 rows=0 loops=1) (projection time=0.000)
```

每个算子的执行信息都包含三个部分：

- dn_6001_6002/dn_6003_6004表示具体执行的节点信息，括号中的信息是实际的执行信息。
- actual time表示实际的执行时间，第一个数字表示执行时进入当前算子到输出第一条数据所花费的时间，第二个数字表示输出所有数据的总执行时间。
- rows表示当前算子输出数据行数。
- loops表示当前算子的执行次数。需要注意，对于分区表来说，每一个分区表的扫描就是一次完整的扫描操作，当切换到下一个分区的时候，又是一次新的扫描操作。

– CPU信息

```
 dn_6001_6002 (CPU: ex c/r=0, ex row=0, ex cyc=65, inc cyc=65)
```

每个算子执行的过程都有CPU信息，其中cyc代表的是CPU的周期数，ex cyc表示的是当前算子的周期数，不包含其子节点；inc cyc是包含子节点的周期数；ex row是当前算子输出的数据行数；ex c/r则是ex cyc/ex row得到的每条数据所用的平均周期数。

– Buffer信息

```
dn_6001_6002 (Buffers: 0)
dn_6003_6004 (Buffers: 0)
dn_6005_6006 (Buffers: 0)
```

buffers显示缓冲区信息，包括共享块和临时块的读和写。

共享块包含表和索引，临时块在排序和物化中使用的磁盘块。上层节点显示出来的块数据包含了其所有子节点使用的块数。

7. User Define Profiling

```
                    User Define Profiling
------------------------------------------------------------------
Plan Node id: 1  Track name: coordinator get datanode connection
      cn_5001 (time=9.306 total_calls=1 loops=1)
Plan Node id: 1  Track name: coordinator begin transaction
      cn_5001 (time=0.002 total_calls=1 loops=1)
Plan Node id: 1  Track name: coordinator send command
      cn_5001 (time=0.113 total_calls=3 loops=1)
Plan Node id: 1  Track name: coordinator get the first tuple
      cn_5001 (time=0.091 total_calls=12 loops=1)
```

自定义信息，这一部分显示的是CN和DN、DN和DN建连的时间，以及存储层的一些执行信息。

8. Query Summary


```
                        ====== Query Summary =====
---------------------------------------------------------------------------------
Datanode executor start time [dn_6005_6006, dn_6001_6002]: [0.360 ms,0.483 ms]
Datanode executor run time [dn_6001_6002, dn_6003_6004]: [0.008 ms,0.009 ms]
Datanode executor end time [dn_6003_6004, dn_6005_6006]: [0.036 ms,0.066 ms]
Remote query poll time: 2.649 ms, Deserialze time: 0.000 ms
System available mem: 1761280KB
Query Max mem: 1761280KB
Query estimated mem: 3328KB
Enqueue time: 0.030 ms
Coordinator executor start time: 0.083 ms
Coordinator executor run time: 13.044 ms
Coordinator executor end time: 0.034 ms
Parser runtime: 0.060 ms
Planner runtime: 0.539 ms
Query Id: 218706056932222840
Unique SQL Id: 2641724793
Total runtime: 13.906 ms
```

这一部分主要打印总的执行时间和网络流量，包括了各个DN上初始化和结束阶段的最大最小执行时间、CN上的初始化、执行、结束阶段的时间，以及当前语句执行时系统可用内存、语句估算内存等信息。

- DataNode executor start time：DN执行器开始时间，[min_node_name, max_node_name] : [min_time, max_time]
- DataNode executor run time：DN执行器运行时间，[min_node_name, max_node_name] : [min_time, max_time]
- DataNode executor end time：DN执行器结束时间，[min_node_name, max_node_name] : [min_time, max_time]
- Remote query poll time：接收结果时用于poll等待的时间
- System available mem：系统可用内存
- Query Max mem：查询最大内存
- Enqueue time：入队时间
- Coordinator executor start time：CN执行器开始时间
- Coordinator executor run time：CN执行器运行时间
- Coordinator executor end time：CN执行器结束时间
- Parser runtime：解析器运行时间
- Planner runtime：优化器执行时间
- 网络流量，stream算子发送的数据量
- Query Id：查询ID
- Unique SQL ID：约束SQL ID
- Total runtime：总执行时间

须知

- A-rows和E-rows的差异体现了优化器估算和实际执行的偏差度。一般情况下两者偏差越大，则可以认为优化器生成的计划的越不可信，人工干预调优的必要性越大。
- A-time中的两个值偏差越大，表明此算子的计算偏斜(在不同DN上执行时间差异)越大，人工干预调优的必要性越大。一般来说，两个相邻的算子，上层算子的执行时间包含下层算子的执行时间，但如果上层算子为stream算子，由于各线程不存在驱动关系，上层算子执行时间可能小于下层算子的执行时间，即不存在包含关系。
- Max Query Peak Memory经常用来估算SQL语句耗费内存，也被用来作为SQL语句调优时运行态内存参数设置的重要依据。一般会以EXPLAIN ANALYZE或EXPLAIN PERFORMANCE的输出作为进一步调优的输入。

4 SQL 调优指南

4.1 调优流程

对慢SQL语句进行分析，通常包括以下步骤：

操作步骤

步骤1 收集SQL中涉及到的所有表的统计信息。在数据库中，统计信息是规划器生成计划的源数据。没有收集统计信息或者统计信息陈旧往往会造成执行计划严重劣化，从而导致性能问题。从经验数据来看，10%左右性能问题是因为没有收集统计信息。具体请参见**更新统计信息**。

步骤2 **审视和修改表定义**。

步骤3 通常情况下，有些SQL语句可以通过查询重写转换成等价的，或特定场景下等价的语句。重写后的语句比原语句更简单，且可以简化某些执行步骤达到提升性能的目的。查询重写方法在各个数据库中基本是通用的。**SQL语句改写规则**介绍了几种常用的通过改写SQL进行调优的方法。

步骤4 通过查看执行计划来查找原因。如果SQL长时间运行未结束，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查看执行计划及实际运行情况，以便更精准地定位问题原因。有关执行计划的详细介绍请参见**SQL执行计划**。

步骤5 针对EXPLAIN或EXPLAIN PERFORMANCE信息，定位SQL慢的具体原因以及改进措施，具体参见**典型SQL调优点**。

步骤6 用户可以通过指定join顺序，join、stream、scan方法，指定结果行数，指定重分布过程中的倾斜信息等多个手段来进行执行计划的调优，以提升查询的性能。详细请参见**使用Plan Hint进行调优**。

步骤7 为了保证数据库性能的持续优质，建议**例行维护表**和**例行重建索引**。

步骤8 （可选）GaussDB(DWS)支持在资源富足的情况下，通过算子并行来提升性能。详细请参见**SMP手动调优建议**。

----结束

4.2 更新统计信息

在数据库中，统计信息是规划器生成计划的源数据。没有收集统计信息或者统计信息陈旧往往会造成执行计划严重劣化，从而导致性能问题。

背景信息

ANALYZE语句可收集与数据库中表内容相关的统计信息，统计结果存储在系统表PG_STATISTIC中。查询优化器会使用这些统计数据，以生成最有效的执行计划。

建议在执行了大批量插入/删除操作后，例行对表或全库执行ANALYZE语句更新统计信息。目前默认收集统计信息的采样比例是30000行（即：guc参数default_statistics_target默认设置为100），如果表的总行数超过一定行数（大于1600000），建议设置guc参数default_statistics_target为-2，即按2%收集样本估算统计信息。

对于在批处理脚本或者存储过程中生成的中间表，也需要在完成数据生成之后显式的调用ANALYZE。

对于表中多个列有相关性且查询中有同时基于这些列的条件或分组操作的情况，可尝试收集多列统计信息，以便查询优化器可以更准确地估算行数，并生成更有效的执行计划。

生成统计信息

使用以下命令更新某个表或者整个database的统计信息。

```
ANALYZE tablename;                    --更新单个表的统计信息
ANALYZE;                              --更新全库的统计信息
```

使用以下命令进行多列统计信息相关操作。

```
ANALYZE tablename ((column_1, column_2));                    --收集tablename表的column_1、column_2列的多列统计信息

ALTER TABLE tablename ADD STATISTICS ((column_1, column_2));    --添加tablename表的column_1、column_2列的多列统计信息声明
ANALYZE tablename;                                           --收集单列统计信息，并收集已声明的多列统计信息

ALTER TABLE tablename DELETE STATISTICS ((column_1, column_2)); --删除tablename表的column_1、column_2列的多列统计信息或其声明
```

须知

- 在使用ALTER TABLE tablename ADD STATISTICS语句添加了多列统计信息声明后，系统并不会立刻收集多列统计信息，而是在下次对该表或全库进行ANALYZE时，进行多列统计信息的收集。如果想直接收集多列统计信息，请使用ANALYZE命令进行收集。
- 使用EXPLAIN查看各SQL的执行计划时，如果发现某个表SEQ SCAN的输出中rows=10，rows=10是系统给的默认值，有可能该表没有进行ANALYZE，需要对该表执行ANALYZE。

提升统计信息质量

ANALYZE是按照随机采样算法从表上采样，根据样本计算表数据特征。采样数可以通过配置参数default_statistics_target进行控制，default_statistics_target取值范围为-100~10000，默认值为100。

1）当default_statistics_target > 0时；采样的样本数为300*default_statistics_target，default_statistics_target取值越大，采样的样本也越大，样本占用的内存空间也越大，统计信息计算耗时也越长。

2）当default_statistics_target < 0时，采样的样本数为 (default_statistics_target)/100*表的总行数，default_statistics_target取值越小，采样的样本也越大。当default_statistics_target < 0时会把采样数据下盘，不存在样本占用的内存空间的问题，但是因为样本过大，计算耗时长的问题同样存在。

default_statistics_target < 0时，实际采样数是 (default_statistics_target)/100*表的总行，所以我们又称之为百分比采样。

自动收集统计信息

当配置参数autoanalyze打开时，查询语句走到优化器发现表不存在统计信息，会自动触发统计信息收集，以满足优化器的需求。

注：只有对统计信息敏感的复杂查询动作（多表关联等操作）的SQL语句执行时才会触发自动收集统计信息；简单查询(比如单点，单表聚合等) 不会触发自动收集统计信息。

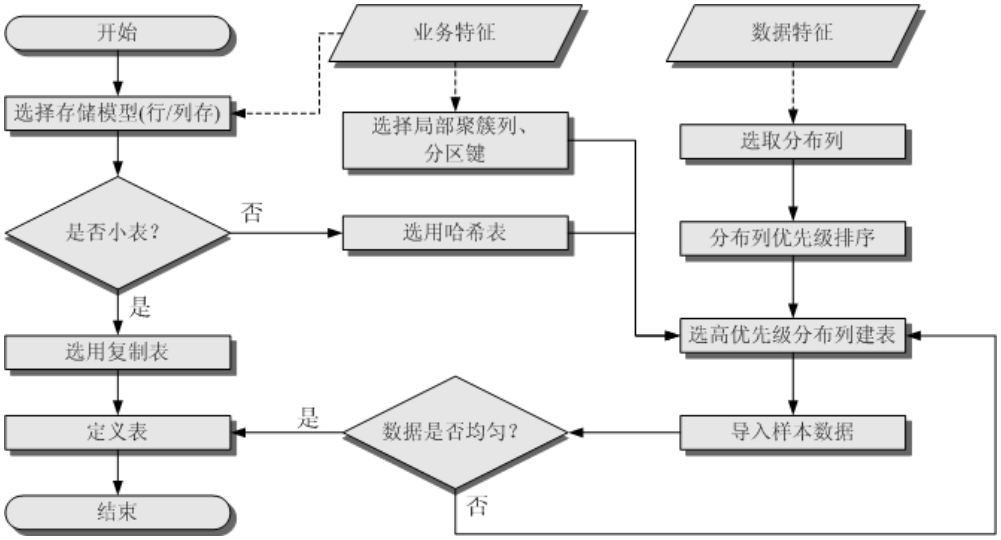
4.3 审视和修改表定义

在分布式框架下，数据分布在各个DN上。一个或者几个DN的数据存在一块物理存储设备上，好的表定义至少需要达到以下几个目标：

1. **表数据均匀分布在各个DN上**，以防止单个DN对应的存储设备空间不足造成集群有效容量下降。选择合适分布列，避免数据分布倾斜可以实现该点。
2. **表Scan压力均匀分散在各个DN上**，以避免单DN的Scan压力过大，形成Scan的单节点瓶颈。分布列不选择基表上等值filter中的列可以实现该点。
3. **减少扫描数据量**。通过分区的剪枝机制可以实现该点。
4. **尽量减少随机IO**。通过聚簇/局部聚簇可以实现该点。
5. **尽量避免数据shuffle**，减小网络压力。通过选择join-condition或者group by列为分布列可以最大程度的实现这点。

从上述描述来看表定义中最重要的一点是分布列的选择。创建表定义一般遵循**图 1 表定义流程**所示流程。表定义在数据库设计阶段创建，在SQL调优过程中进行审视和修改。

图 4-1 表定义流程



4.4 SQL 语句改写规则

根据数据库的SQL执行机制以及大量的实践，总结发现：通过一定的规则调整SQL语句，在保证结果正确的基础上，能够提高SQL执行效率。如果遵守下列规则，常常能够大幅度提升业务查询效率。

- **使用union all代替union**

  union在合并两个集合时会执行去重操作，而union all则直接将两个结果集合并、不执行去重。执行去重会消耗大量的时间，因此，在一些实际应用场景中，如果通过业务逻辑已确认两个集合不存在重叠，可用union all替代union以便提升性能。

- **join列增加非空过滤条件**

  若join列上的NULL值较多，则可以加上is not null过滤条件，以实现数据的提前过滤，提高join效率。

- **not in转not exists**

  not in语句需要使用nestloop anti join来实现，而not exists则可以通过hash anti join来实现。在join列不存在null值的情况下，not exists和not in等价。因此在确保没有null值时，可以通过将not in转换为not exists，通过生成hash join来提升查询效率。

  如下所示，如果t2.d2字段中没有null值(t2.d2字段在表定义中not null)查询可以修改为

  ```
  SELECT * FROM t1 WHERE  NOT EXISTS (SELECT * FROM t2 WHERE t1.c1=t2.d2);
  ```

  产生的计划如下：

图 4-2 not exists 执行计划

```
 id |                operation
----+----------------------------------------
  1 | ->  Streaming (type: GATHER)
  2 |     ->  Hash Right Anti Join (3, 5)
  3 |        ->  Streaming(type: REDISTRIBUTE)
  4 |           ->  Seq Scan on t2
  5 |        ->  Hash
  6 |           ->  Seq Scan on t1

Predicate Information (identified by plan id)
---------------------------------------------
  2 --Hash Right Anti Join (3, 5)
        Hash Cond: (t2.d2 = t1.c1)
(13 rows)
```

- **选择hashagg。**

  查询中GROUP BY语句如果生成了groupagg+sort的plan性能会比较差，可以通过加大work_mem的方法生成hashagg的plan，因为不用排序而提高性能。

- **尝试将函数替换为case语句。**

  GaussDB(DWS)函数调用性能较低，如果出现过多的函数调用导致性能下降很多，可以根据情况把可下推函数的函数改成CASE表达式。

- **避免对索引使用函数或表达式运算。**

  对索引使用函数或表达式运算会停止使用索引转而执行全表扫描。

- **尽量避免在where子句中使用!=或<>操作符、null值判断、or连接、参数隐式转换。**

- **对复杂SQL语句进行拆分。**

  对于过于复杂并且不易通过以上方法调整性能的SQL可以考虑拆分的方法，把SQL中某一部分拆分成独立的SQL并把执行结果存入临时表，拆分常见的场景包括但不限于：

  - 作业中多个SQL有同样的子查询，并且子查询数据量较大。
  - Plan cost计算不准，导致子查询hash bucket太小，比如实际数据1000W行，hash bucket只有1000。
  - 函数（如substr,to_number）导致大数据量子查询选择度计算不准。
  - 多DN环境下对大表做broadcast的子查询。

4.5 典型 SQL 调优点

SQL调优是一个不断分析与尝试的过程：试跑Query，判断性能是否满足要求；如果不满足要求，则通过查看执行计划分析原因并进行针对性优化；然后重新试跑和优化，直到满足性能目标。

4.5.1 SQL 自诊断

用户在执行INSERT/UPDATE/DELETE/SELECT/MERGE INTO或者CREATE TABLE AS语句时，可能会遇到性能问题。产品内置集成了性能自动诊断功能，并把相关的诊断信息保存到**实时TopSQL**中，当配置参数“enable_resource_track”为on的时候，这些诊断信息会转存到**历史TopSQL**中。通过查询**GS_WLM_SESSION_STATISTICS**，**GS_WLM_SESSION_HISTORY**，**GS_WLM_SESSION_INFO**视图的warning字段可以获得对应的性能诊断信息，为性能调优提供参考。

SQL自诊断的告警类型与**resource_track_level**的设置有关系。当“resource_track_level”设置为query时，可以诊断多列/单列统计信息未收集、分区未剪枝告警和SQL不下推的告警等非执行态的诊断信息；“resource_track_level”设置为perf或operator时，可以诊断所有的告警场景。

SQL自诊断的诊断范围与**resource_track_cost**的设置有关系。当SQL的代价大于“resource_track_cost”时，SQL才会被诊断。SQL的代价可以通过explain来确认。

执行EXPLAIN PERFORMANCE或者EXPLAIN VERBOSE的时候，除缺乏多列统计信息之外的SQL自诊断信息也会输出，具体请参考**SQL执行计划**。

告警场景

目前支持对以下9种导致性能问题的场景上报告警。

- 多列/单列统计信息未收集

  如果存在单列或者多列统计信息未收集，则上报相关告警。对于这种告警，建议的优化方案是对相关表进行ANALYZE，可参考**更新统计信息**和**统计信息调优**。

  需要特别注意的是，如果查询语句中的OBS外表和HDFS外表未收集统计信息，也会上报统计信息未收集的告警，因为OBS外表和HDFS外表的ANALYZE的性能比较差，一般不建议对其进行ANALYZE，但是建议使用ALTER FOREIGN TABLE的语法修改外表的totalrows属性，从而修正外表的行数估算。

  告警信息示例：

  整表的统计信息未收集：

  ```
  Statistic Not Collect
        schema_test.t1
  ```

  单列统计信息未收集：

  ```
  Statistic Not Collect
        schema_test.t2(c1)
  ```

  多列统计信息未收集:

  ```
  Statistic Not Collect
        schema_test.t3((c1,c2))
  ```

  单列和多列统计信息未收集：

  ```
  Statistic Not Collect
        schema_test.t4(c1)
        schema_test.t5((c1,c2))
  ```

- 分区不剪枝（该功能仅8.1.2及以上版本支持）

  分区表查询时，往往会期望通过分区键上的约束条件进行分区剪枝，从而提升分区表查询性能，但有时候会因为约束条件书写不当，导致分区表没有剪枝，出现查询性能问题，具体请参见**案例：改写SQL排除剪枝干扰**。

- SQL不下推

  对于不下推的SQL，尽可能详细上报导致不下推的原因。调优方法可参考案例**语句下推调优**。

  导致不下推的因素主要有以下两点：

  - 函数导致的不下推

    诊断信息中会给出具体的函数名称。函数下推是由函数的shippable属性决定，具体可参见CREATE FUNCTION语法。

  - 语法导致的不下推

    诊断信息中会提示导致不下推的具体语法，例如：含有With Recursive，Distinct On，row表达式，返回值为record类型的，会告警相应语法不支持下推等等。

告警信息示例：

```
SQL is not plan-shipping
        "enable_stream_operator" is off

SQL is not plan-shipping
        "Distinct On" can not be shipped

SQL is not plan-shipping
        "v_test_unshipping_log" is VIEW that will be treated as Record type can't be shipped
```

- HashJoin中大表做内表

  如果在表连接过程中使用了Hashjoin（可通过 **GS_WLM_SESSION_HISTORY**的“query_plan”字段查看），且连接的内表行数是外表行数的10倍或以上；同时内表在每个DN上的平均行数大于10万行，且发生了下盘，则上报相关告警。针对这种场景，需要调整HashJoin内外表顺序，具体调优方法参考**Join顺序的Hint**。

  告警信息示例：

  ```
  Execute diagnostic information
        PlanNode[7] Large Table is INNER in HashJoin “Vector Hash Aggregate”
  ```

  其中，7为“query_plan”字段文本中编号为7的算子。

- 大表等值连接使用Nestloop

  如果在表连接过程中使用了nestloop（可通过**GS_WLM_SESSION_HISTORY**的query_plan字段查看），并且两个表中较大表的行数平均每个DN上的行数大于10万行、表的连接中存在等值连接，则上报相关告警。针对这种场景，需要调整表关联方式，禁止当前内外表之间使用NestLoop的关联方式，具体调优方法参考**Join方式的Hint**。

  告警信息示例：

  ```
  Execute diagnostic information
        PlanNode[5] Large Table with Equal-Condition use Nestloop"Nested Loop"
  ```

- 大表Broadcast

  如果在Broadcast算子中，平均每DN的行数大于10万行，则告警大表broadcast。针对这种场景，需要禁止BroadCast下层算子做Broadcast动作，具体调优方法参考**Stream方式的Hint**。

  告警信息示例：

  ```
  Execute diagnostic information
        PlanNode[5] Large Table in Broadcast "Streaming(type: BROADCAST dop: 1/2)"
  ```

- 数据倾斜

  某表在各DN上的分布，存在某DN上的行数是另一DN上行数的10倍或以上，且有DN中的行数大于10万行，则上报相关告警。该告警一般分为存储层倾斜和计算层倾斜，具体调优方法参考**数据倾斜调优**。

  告警信息示例：

  ```
  Execute diagnostic information
        PlanNode[6] DataSkew:"Seq Scan", min_dn_tuples:0, max_dn_tuples:524288
  ```

- 索引不合理

  在基表扫描时，满足下述条件则上报相关告警：

  - 对于行存表：
    - 使用索引扫描，输出行数/扫描行数>1/1000且输出行数>1万行。
    - 使用顺序扫描，输出行数/扫描行数<1/1000且输出行数<=1万行、扫描行数>1万行。
  - 对于列存表：

  - 使用索引扫描，输出行数/扫描行数>1/10000且输出行数>100。
  - 使用顺序扫描，输出行数/扫描行数<1/10000且输出行数<=100、扫描行数>1万行。

  调优方法可参考**算子级调优**，也可参考案例**案例：建立合适的索引**和**案例：使用partial cluster key**。

  告警信息示例：

  ```
  Execute diagnostic information
      PlanNode[4] Indexscan is not properly used:"Index Only Scan", output:524288, filtered:0, rate:1.00000
      PlanNode[5] Indexscan is ought to be used:"Seq Scan", output:1, filtered:524288, rate:0.00000
  ```

  需要注意的是，这个诊断结果只是针对当前SQL的建议，是否创建索引要结合整体业务综合分析，对于高频的过滤条件才建议创建索引。

- 估算不准

  对于平均每DN行数如果优化器的估算行数和实际行数中的较大值大于10万行，并且估算行数和实际行数中较大值是较小值的10倍或以上，则上报相关告警。针对这种场景，可以参照**行数的Hint**修正行数估算，让优化器在正确的行数基础上重新规划执行计划。

  告警信息示例：

  ```
  Execute diagnostic information
      PlanNode[5] Inaccurate Estimation-Rows: "Hash Join" A-Rows:0, E-Rows:52488
  ```

规格约束

1. 告警字符串长度上限为2048。如果告警信息超过这个长度（例如存在大量未收集统计信息的超长表名，列名等信息）则不告警，只上报warning：
   ```
   WARNING, "Planner issue report is truncated, the rest of planner issues will be skipped"
   ```
2. 如果query存在limit节点（即查询语句中包含limit），则不会上报limit节点以下的Operator级别的告警。
3. 对于“数据倾斜”和“估算不准”两种类型告警，在某一个plan树结构下，只上报下层节点的告警，上层节点不再重复告警。这主要是因为这两种类型的告警可能是因为底层触发上层的。例如，如果在scan节点已经存在数据倾斜，那么在上层的hashagg等其他算子很可能也出现数据倾斜。

4.5.2 语句下推调优

语句下推介绍

目前，GaussDB(DWS)优化器在分布式框架下制定语句的执行策略时，有三种执行计划方式：生成下推语句计划、生成分布式执行计划、生成发送语句的分布式执行计划。

- 下推语句计划：指直接将查询语句从CN发送到DN进行执行，然后将执行结果返回给CN。
- 分布式执行计划：指CN对查询语句进行编译和优化，生成计划树，再将计划树发送给DN进行执行，并在执行完毕后返回结果到CN。
- 发送语句的分布式执行计划：上述两种方式都不可行时，将可下推的查询部分组成查询语句（多为基表扫描语句）下推到DN进行执行，获取中间结果到CN，然后在CN执行剩下的部分。

在发送语句的分布式执行计划策略中，要将大量中间结果从DN发送到CN，并且要在CN运行不能下推的部分语句，会导致CN成为性能瓶颈（带宽、存储、计算等）。在进行性能调优的时候，应尽量避免只能选择该策略的查询语句。

执行语句不能下推是因为语句中含有**不支持下推的函数**或者**不支持下推的语法**。一般都可以通过等价改写规避执行计划不能下推的问题。

查看执行计划是否下推

执行计划是否下推可以依靠如下方法快速判断：

步骤1 将GUC参数“**enable_fast_query_shipping**”设置为off，使查询优化器使用分布式框架策略。

```
SET enable_fast_query_shipping = off;
```

步骤2 查看执行计划。

如果执行计划中有Data Node Scan节点，那么此执行计划为不可下推的执行计划；如果执行计划中有Streaming节点，那么计划是可以下推的。

例如如下业务SQL：

```
select
count(ss.ss_sold_date_sk order by ss.ss_sold_date_sk)c1
from store_sales ss, store_returns sr
where
sr.sr_customer_sk = ss.ss_customer_sk;
```

执行计划如下，可以看出此SQL语句不能下推。

```
                          QUERY PLAN
------------------------------------------------------------------------------
Aggregate
->  Hash Join
Hash Cond: (ss.ss_customer_sk = sr.sr_customer_sk)
->  Data Node Scan on store_sales "_REMOTE_TABLE_QUERY_"
Node/s: All datanodes
->  Hash
->  Data Node Scan on store_returns "_REMOTE_TABLE_QUERY_"
Node/s: All datanodes
(8 rows)
```

----结束

不支持下推的语法

以如下三个表定义说明不支持下推的SQL语法。

```
CREATE TABLE CUSTOMER1
(
    C_CUSTKEY     BIGINT NOT NULL
  , C_NAME        VARCHAR(25) NOT NULL
  , C_ADDRESS     VARCHAR(40) NOT NULL
  , C_NATIONKEY   INT NOT NULL
  , C_PHONE       CHAR(15) NOT NULL
  , C_ACCTBAL     DECIMAL(15,2)   NOT NULL
  , C_MKTSEGMENT  CHAR(10) NOT NULL
  , C_COMMENT     VARCHAR(117) NOT NULL
)
DISTRIBUTE BY hash(C_CUSTKEY);
CREATE TABLE test_stream(a int,b float); --float不支持重分布
CREATE TABLE sal_emp ( c1 integer[] ) DISTRIBUTE BY replication;
```

- 不支持returning语句下推

```
explain update customer1 set C_NAME = 'a' returning c_name;
                              QUERY PLAN
-----------------------------------------------------------------------
 Update on customer1  (cost=0.00..0.00 rows=30 width=187)
   Node/s: All datanodes
   Node expr: c_custkey
   ->  Data Node Scan on customer1 "_REMOTE_TABLE_QUERY_"  (cost=0.00..0.00 rows=30 width=187)
         Node/s: All datanodes
(5 rows)
```

- count(distinct expr)中的字段不支持重分布，则不支持下推

```
explain verbose select count(distinct b) from test_stream;
                                    QUERY PLAN
---------------------------------------------------------------------- Aggregate  (cost=2.50..2.51 rows=1 width=8)
   Output: count(DISTINCT test_stream.b)
   ->  Data Node Scan on test_stream "_REMOTE_TABLE_QUERY_"  (cost=0.00..0.00 rows=30 width=8)
         Output: test_stream.b
         Node/s: All datanodes
         Remote query: SELECT b FROM ONLY public.test_stream WHERE true
(6 rows)
```

- 不支持distinct on用法下推

```
explain verbose select distinct on (c_custkey) c_custkey from customer1 order by c_custkey;
                                    QUERY PLAN
---------------------------------------------------------------------- Unique  (cost=49.83..54.83 rows=30 width=8)
   Output: customer1.c_custkey
   ->  Sort  (cost=49.83..52.33 rows=30 width=8)
         Output: customer1.c_custkey
         Sort Key: customer1.c_custkey
         ->  Data Node Scan on customer1 "_REMOTE_TABLE_QUERY_"  (cost=0.00..0.00 rows=30
width=8)
               Output: customer1.c_custkey
               Node/s: All datanodes
               Remote query: SELECT c_custkey FROM ONLY public.customer1 WHERE true
(9 rows)
```

- Fulljoin的join列如果不支持重分布，则不支持下推

```
explain select * from test_stream t1 full join test_stream t2 on t1.a=t2.b;
                                      QUERY PLAN
---------------------------------------------------------------------- Hash Full Join  (cost=0.38..0.82 rows=30
width=24)
   Hash Cond: ((t1.a)::double precision = t2.b)
   ->  Data Node Scan on test_stream "_REMOTE_TABLE_QUERY_"  (cost=0.00..0.00 rows=30 width=12)
         Node/s: All datanodes
   ->  Hash  (cost=0.00..0.00 rows=30 width=12)
         ->  Data Node Scan on test_stream "_REMOTE_TABLE_QUERY_"  (cost=0.00..0.00 rows=30
width=12)
               Node/s: All datanodes
(7 rows)
```

- 不支持数组表达式下推

```
explain verbose select array[c_custkey,1] from customer1 order by c_custkey;

                         QUERY PLAN
---------------------------------------------------------------------- Sort  (cost=49.83..52.33 rows=30 width=8)
   Output: (ARRAY[customer1.c_custkey, 1::bigint]), customer1.c_custkey
   Sort Key: customer1.c_custkey
   ->  Data Node Scan on "__REMOTE_SORT_QUERY__"  (cost=0.00..0.00 rows=30 width=8)
         Output: (ARRAY[customer1.c_custkey, 1::bigint]), customer1.c_custkey
         Node/s: All datanodes
         Remote query: SELECT ARRAY[c_custkey, 1::bigint], c_custkey FROM ONLY public.customer1
WHERE true ORDER BY 2
(7 rows)
```

- With Recursive当前版本不支持下推的场景和原因如下：

序号	场景	不下推原因
1	包含外表、HDFS表的查询场景	LOG: SQL can't be shipped, reason: RecursiveUnion contains HDFS Table or ForeignScan is not shippable（LOG为CN日志中打印的不下推原因，下同） 外表、HDFS表，当前版本暂不支持下推。
2	多nodegroup场景	LOG: SQL can't be shipped, reason: With-Recursive under multi-nodegroup scenario is not shippable 基表存储nodegroup不相同，或者计算nodegroup与基表不相同，当前版本暂不支持下推。
3	WITH recursive t_result AS (SELECT dm,sj_dm,name,1 as level FROM test_rec_part WHERE sj_dm > 10 UNION SELECT t2.dm,t2.sj_dm,t2.name||' > '|| t1.name,t1.level+1 FROM t_result t1 JOIN test_rec_part t2 ON t2.sj_dm = t1.dm) SELECT * FROM t_result t;	LOG: SQL can't be shipped, reason: With-Recursive does not contain "ALL" to bind recursive & none-recursive branches UNION不带ALL，需要去重。
4	WITH RECURSIVE x(id) AS (select count(1) from pg_class where oid=1247 UNION ALL SELECT id+1 FROM x WHERE id < 5), y(id) AS (select count(1) from pg_class where oid=1247 UNION ALL SELECT id+1 FROM x WHERE id < 10) SELECT y.*, x.* FROM y LEFT JOIN x USING (id) ORDER BY 1;	LOG: SQL can't be shipped, reason: With-Recursive contains system table is not shippable 基表中有系统表。
5	WITH RECURSIVE t(n) AS (VALUES (1) UNION ALL SELECT n+1 FROM t WHERE n < 100) SELECT sum(n) FROM t;	LOG: SQL can't be shipped, reason: With-Recursive contains only values rte is not shippable 基表扫描只有VALUES子句，仅在CN上即可完成执行。

序号	场景	不下推原因
6	select a.ID,a.Name, (with recursive cte as (select ID, PID, NAME from b where b.ID = 1 union all select parent.ID,parent.PID,parent.NAME from cte as child join b as parent on child.pid=parent.id where child.ID = a.ID) select NAME from cte limit 1) cName from (select id, name, count(*) as cnt from a group by id,name) a order by 1,2;	LOG: SQL can't be shipped, reason: With-Recursive recursive term correlated only is not shippable 相关子查询的关联条件仅在递归部分，非递归部分无关联条件。
7	WITH recursive t_result AS (select * from(SELECT dm,sj_dm,name,1 as level FROM test_rec_part WHERE sj_dm < 10 order by dm limit 6 offset 2) UNION all SELECT t2.dm,t2.sj_dm,t2.name||' > '||t1.name,t1.level+1 FROM t_result t1 JOIN test_rec_part t2 ON t2.sj_dm = t1.dm) SELECT * FROM t_result t;	LOG: SQL can't be shipped, reason: With-Recursive contains conflict distribution in none-recursive(Replicate) recursive(Hash) 非递归部分带limit为Replicate计划，递归部分为Hash计划，计划存在冲突。
8	with recursive cte as (select * from rec_tb4 where id<4 union all select h.id,h.parentID,h.name from (with recursive cte as (select * from rec_tb4 where id<4 union all select h.id,h.parentID,h.name from rec_tb4 h inner join cte c on h.id=c.parentID) SELECT id ,parentID,name from cte order by parentID) h inner join cte c on h.id=c.parentID) SELECT id ,parentID,name from cte order by parentID,1,2,3;	LOG: SQL can't be shipped, reason: Recursive CTE references recursive CTE "cte" 多层Recursive嵌套，即recursive的递归部分又嵌套另一个recursive查询。

不支持下推的函数

首先介绍函数的易变性。在GaussDB(DWS)中共分三种形态：

- **IMMUTABLE**
  表示该函数在给出同样的参数值时总是返回同样的结果。
- **STABLE**

表示该函数不能修改数据库，对相同参数值，在同一次表扫描里，该函数的返回值不变，但是返回值可能在不同SQL语句之间变化。

- **VOLATILE**

  表示该函数值可以在一次表扫描内改变，因此不会做任何优化。

函数易变性可以查询pg_proc的provolatile字段获得，i代表IMMUTABLE，s代表STABLE，v代表VOLATILE。另外，在pg_proc中的proshippable字段，取值范围为t/f/NULL，这个字段与provolatile字段一起用于描述函数是否下推。

- 如果函数的provolatile属性为i，则无论proshippable的值是否为t，则函数始终可以下推。
- 如果函数的provolatile属性为s或v，则仅当proshippable的值为t时，函数可以下推。
- random如果出现CTE中，也不下推。因为这种场景下下推可能出现结果错误。

对于用户自定义函数，可以在创建函数的时候指定provolatile和proshippable属性的值，详细请参考CREATE FUNCTION。

对于函数不能下推的场景：

- 如果是系统函数，建议根据业务等价替换这个函数。
- 如果是自定义函数，建议分析客户业务场景，看函数的provolatile和proshippable属性定义是否正确。

实例分析：自定义函数

对于自定义函数，如果对于确定的输入，有确定的输出，则应将函数定义为immutable类型。

利用TPCDS的销售信息举例，需要写一个函数来获取商品的打折情况，即定义一个计算折扣的函数：

```
CREATE FUNCTION func_percent_2 (NUMERIC, NUMERIC) RETURNS NUMERIC
AS 'SELECT $1 / $2 WHERE $2 > 0.01'
LANGUAGE SQL
VOLATILE;
```

执行下列语句：

```
SELECT func_percent_2(ss_sales_price, ss_list_price)
FROM store_sales;
```

其执行计划为：

```
 Data Node Scan on store_sales "_REMOTE_TABLE_QUERY_"
   Output: func_percent_2(store_sales.ss_sales_price, store_sales.ss_list_price)
   Remote query: SELECT ss_sales_price, ss_list_price FROM ONLY store_sales WHERE true
(3 rows)
```

可见，func_percent_2并没有被下推，而是将ss_sales_price和ss_list_price收到CN上，再进行计算，消耗大量CN的资源，而且计算缓慢。

由于该自定义函数对确定的输入有确定的输出，如果将该自定义函数改为：

```
CREATE FUNCTION func_percent_1 (NUMERIC, NUMERIC) RETURNS NUMERIC
AS 'SELECT $1 / $2 WHERE $2 > 0.01'
LANGUAGE SQL
IMMUTABLE;
```

执行语句：

```
SELECT func_percent_1(ss_sales_price, ss_list_price)
FROM store_sales;
```

其执行计划为：

```
 Data Node Scan on "__REMOTE_FQS_QUERY__"  (cost=0.00..0.00 rows=0 width=0)
   Output: (func_percent_1(store_sales.ss_sales_price, store_sales.ss_list_price))
   Node/s: All datanodes
   Remote query: SELECT public.func_percent_1(ss_sales_price, ss_list_price) AS func_percent_1 FROM public.store_sales
(4 rows)
```

可见函数func_percent_1被下推到DN执行，提升了执行效率（TPCDS 1000X，3CN18DN，查询效率提升100倍以上）。

实例分析 2：使排序下推

请参考**案例：使排序下推**。

4.5.3 子查询调优

子查询背景介绍

应用程序通过SQL语句来操作数据库时会使用大量的子查询，这种写法比直接对两个表做连接操作在结构上和思路上更清晰，尤其是在一些比较复杂的查询语句中，子查询有更完整、更独立的语义，会使SQL对业务逻辑的表达更清晰更容易理解，因此得到了广泛的应用。

GaussDB(DWS)根据子查询在SQL语句中的位置把子查询分成了子查询、子链接两种形式。

- 子查询SubQuery：对应于查询解析树中的范围表RangeTblEntry，更通俗一些指的是出现在FROM语句后面的独立的SELECT语句。
- 子链接SubLink：对应于查询解析树中的表达式，更通俗一些指的是出现在where/on子句、targetlist里面的语句。

  综上，对于查询解析树而言，SubQuery的本质是范围表、而SubLink的本质是表达式。针对SubLink场景而言，由于SubLink可以出现在约束条件、表达式中，按照GaussDB(DWS)对sublink的实现，sublink可以分为以下几类：
  - exist_sublink：对应EXIST、NOT EXIST语句
  - any_sublink：对应op Any(select…)语句，其中OP可以是IN,<,>,=操作符
  - all_sublink：对应op ALL(select…)语句，其中OP可以是IN,<,>,=操作符
  - rowcompare_sublink：对应record op (select …)语句
  - expr_sublink：对应(SELECT with single targetlist item ...)语句
  - array_sublink：对应ARRAY(select…)语句
  - cte_sublink：对应with query(…)语句

  其中OLAP、HTAP场景中常用的sublink为exist_sublink、any_sublink，在GaussDB(DWS)的优化引擎中对其应用场景做了优化（子链接提升），由于SQL语句中子查询的使用的灵活性，会带来SQL子查询过于复杂而造成的性能问题。子查询从大类上来看，分为非相关子查询和相关子查询：
  - **非相关子查询None-Correlated SubQuery**

    子查询的执行不依赖于外层父查询的任何属性值。这样子查询具有独立性，可独自求解，形成一个子查询计划先于外层的查询求解。

    例如：

    ```
    select t1.c1,t1.c2
    from t1
    ```

```
where t1.c1 in (
   select c2
   from t2
   where t2.c2 IN (2,3,4)
);
                        QUERY PLAN
------------------------------------------------------------------
Streaming (type: GATHER)
  Node/s: All datanodes
  ->  Hash Right Semi Join
        Hash Cond: (t2.c2 = t1.c1)
        ->  Streaming(type: REDISTRIBUTE)
              Spawn on: All datanodes
              ->  Seq Scan on t2
                    Filter: (c2 = ANY ('{2,3,4}'::integer[]))
        ->  Hash
              ->  Seq Scan on t1
(10 rows)
```

- **相关子查询Correlated-SubQuery**

  子查询的执行依赖于外层父查询的一些属性值（如下列示例t2.c1 = t1.c1条件中的t1.c1）作为内层查询的一个AND-ed条件。这样的子查询不具备独立性，需要和外层查询按分组进行求解。

  例如：

```
select t1.c1,t1.c2
from t1
where t1.c1 in (
   select c2
   from t2
   where t2.c1 = t1.c1 AND t2.c2 in (2,3,4)
);
                              QUERY PLAN
-----------------------------------------------------------------------
Streaming (type: GATHER)
  Node/s: All datanodes
  ->  Seq Scan on t1
        Filter: (SubPlan 1)
        SubPlan 1
          ->  Result
                Filter: (t2.c1 = t1.c1)
                ->  Materialize
                      ->  Streaming(type: BROADCAST)
                            Spawn on: All datanodes
                            ->  Seq Scan on t2
                                  Filter: (c2 = ANY ('{2,3,4}'::integer[]))
(12 rows)
```

GaussDB(DWS)对 SubLink 的优化

针对SubLink的优化策略主要是让内层的子查询提升(pullup)，能够和外表直接做关联查询，从而避免生成SubPlan+Broadcast内表的执行计划。判断子查询是否存在性能风险，可以通过explain查询语句查看Sublink的部分是否被转换成SubPlan+Broadcast的执行计划。

例如：


```
select t1.c1,t1.c2
from t1
where t1.c1 in (
    select c2
    from t2
    where t2.c1 = t1.c1
);
```

```
QUERY PLAN
-------------------------------------------------------------
Streaming (type: GATHER)
  Node/s: All datanodes
  ->  Seq Scan on t1
        Filter: (SubPlan 1)
        SubPlan 1
          ->  Result
                Filter: (t2.c1 = t1.c1)
                ->  Materialize
                      ->  Streaming(type: BROADCAST)
                            Spawn on: All datanodes
                            ->  Seq Scan on t2
(11 rows)
```

- **目前GaussDB(DWS)支持的Sublink-Release场景**
  - IN-Sublink无相关条件
    - 不能包含上一层查询的表中的列（可以包含更高层查询表中的列）。
    - 不能包含易变函数。

```
select t1.c1,t1.c2
from t1
where t1.c1 in (
    select c2
    from t2
    where t2.c1 = 1
);
```

```
QUERY PLAN
-------------------------------------------------------------
Streaming (type: GATHER)
  Node/s: All datanodes
  ->  Nested Loop Semi Join
        Join Filter: (t1.c1 = t2.c2)
        ->  Seq Scan on t1
        ->  Materialize
              ->  Streaming(type: REDISTRIBUTE)
                    Spawn on: datanode1
                    ->  Seq Scan on t2
                          Filter: (c1 = 1)
(10 rows)
```

  - Exist-Sublink包含相关条件

    Where子句中必须包含上一层查询的表中的列，子查询的其它部分不能含有上层查询的表中的列。其它限制如下：
    - 子查询必须有from子句。
    - 子查询不能含有with子句。
    - 子查询不能含有聚集函数。
    - 子查询里不能包含集合操作、排序、limit、windowagg、having操作。
    - 不能包含易变函数。

```
select t1.c1,t1.c2
from t1
where exists (
    select c2
    from t2
    where t2.c1 = t1.c1
);
```

```
QUERY PLAN
-----------------------------------
Streaming (type: GATHER)
  Node/s: All datanodes
  ->  Hash Semi Join
        Hash Cond: (t1.c1 = t2.c1)
        ->  Seq Scan on t1
        ->  Hash
              ->  Seq Scan on t2
(7 rows)
```

  - 包含聚集函数的等值相关子查询的提升

    子查询的where条件中必须含有来自上一层的列，而且此列必须和子查询本层涉及表中的列做相等判断，且这些条件必须用and连接。其它地方不能包含上层的列。其它限制条件如下：
    - 子查询中where条件包含的表达式(列名)必须是表中的列。

- 子查询的Select关键字后，必须有且仅有一个输出列，此输出列必须是聚集函数(如max)，并且聚集函数的参数(t2.c2)不能是来自外层表(t1)中的列。聚集函数不能是count。

  下列示例可以提升：

  ```
  select * from t1 where c1 >(
        select max(t2.c1) from t2 where t2.c1=t1.c1
  );
  ```

  下列示例不能提升，因为子查询没有聚集函数：

  ```
  select * from t1 where c1 >(
        select  t2.c1 from t2 where t2.c1=t1.c1
  );
  ```

  下列示例不能提升，因为子查询有两个输出列：

  ```
  select * from t1 where (c1,c2) >(
        select  max(t2.c1),min(t2.c2) from t2 where t2.c1=t1.c1
  );
  ```

- 子查询必须是from子句。
- 子查询中不能有groupby、having、集合操作。
- 子查询只能是inner join。

  下列示例不能提升：

  ```
  select * from t1 where c1 >(
        select max(t2.c1) from t2 full join t3 on (t2.c2=t3.c2) where t2.c1=t1.c1
  );
  ```

- 子查询的targetlist中不能包含返回set的函数。
- 子查询的where条件中必须含有来自上一层的列，而且此列必须和子查询层涉及表中的列做相等判断，且这些条件必须用and连接。其它地方不能包含上层中的列。下列示例中的最内层子链接可以提升：

  ```
  select * from t3 where t3.c1=(
        select t1.c1
        from t1 where c1 >(
              select max(t2.c1) from t2 where t2.c1=t1.c1
  ));
  ```

  基于上面的示例，再加一个条件，则不能提升，因为最内侧子查询引用了上层中的列。示例如下：

  ```
  select * from t3 where t3.c1=(
        select t1.c1
        from t1 where c1 >(
              select max(t2.c1) from t2 where t2.c1=t1.c1 and t3.c1>t2.c2

  ));
  ```

– 提升OR子句中的SubLink

  当WHERE过滤条件中有OR连接的EXIST相关SubLink，

  例如：

  ```
  select a, c from t1
  where t1.a = (select avg(a) from t3 where t1.b = t3.b) or
  exists (select * from t4 where t1.c = t4.c);
  ```

  将OR-ed连接的EXIST相关子查询OR字句的提升过程：

  i. 提取where条件中，or子句中的opExpr。为：t1.a = (select avg(a) from t3 where t1.b = t3.b)

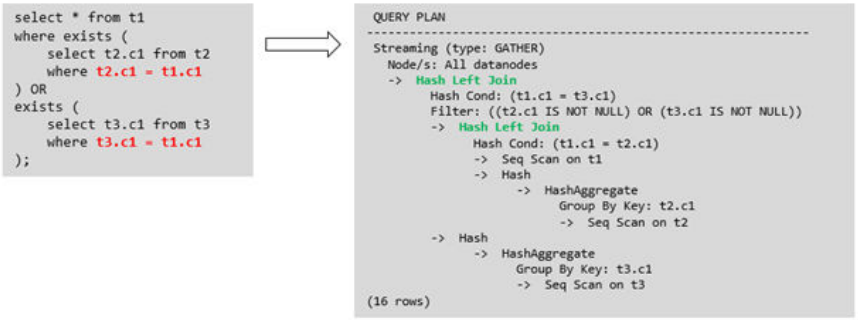
  ii. 这个op操作中包含subquery，判断是否可以提升，如果可以提升，重写subquery为：select avg(a), t3.b from t3 group by t3.b，生成not null

条件t3.b is not null，并将这个opexpr用这个not null条件替换。此时SQL变为：

```
select a, c
from t1 left join (select avg(a) avg, t3.b from t3 group by t3.b)  as t3 on (t1.a = avg and t1.b = t3.b)
where t3.b is not null or exists (select * from t4 where t1.c = t4.c);
```

iii. 再次提取or子句中的exists sublink，exists (select * from t4 where t1.c = t4.c)，判断是否可以提升，如果可以提升，转换subquery为：select t4.c from t4 group by t4.c生成NotNull条件t4.c is not null提升查询，SQL变为：

```
select a, c
from t1 left join (select avg(a) avg, t3.b from t3 group by t3.b)  as t3 on (t1.a = avg and t1.b = t3.b)
left join (select t4.c from t4 group by t4.c) where t3.b is not null or t4.c is not null;
```



- **目前GaussDB(DWS)不支持的Sublink-Release场景**

  除了以上场景之外都不支持Sublink提升，因此关联子查询会被计划成SubPlan +Broadcast的执行计划，当inner表的数据量较大时则会产生性能风险。

  如果相关子查询中跟外层的两张表做join，那么无法提升该子查询，需要通过将父SQL创建成with子句，然后再跟子查询中的表做相关子查询。

  例如：

  ```
  select distinct t1.a, t2.a
  from t1 left join t2 on t1.a=t2.a and not exists (select a,b from test1 where test1.a=t1.a and test1.b=t2.a);
  ```

  改写为

  ```
  with temp as
  (
        select * from (select t1.a as a, t2.a as b from t1 left join t2 on t1.a=t2.a)

  )
  select distinct a,b
  from temp
  where not exists (select a,b from test1 where temp.a=test1.a and temp.b=test1.b);
  ```

  – 出现在targetlist里的相关子查询无法提升(不含count)

    例如：

    ```
    explain (costs off)
    select (select c2 from t2 where t1.c1 = t2.c1) ssq, t1.c2
    from t1
    where t1.c2 > 10;
    ```

    执行计划为：

    ```
    explain (costs off)
    select (select c2 from t2 where t1.c1 = t2.c1) ssq, t1.c2
    from t1
    where t1.c2 > 10;
                      QUERY PLAN
    -----------------------------------------------------
     Streaming (type: GATHER)
    ```

```
  Node/s: All datanodes
  ->  Seq Scan on t1
        Filter: (c2 > 10)
        SubPlan 1
          ->  Result
                Filter: (t1.c1 = t2.c1)
                ->  Materialize
                      ->  Streaming(type: BROADCAST)
                            Spawn on: All datanodes
                            ->  Seq Scan on t2
(11 rows)
```

由于相关子查询出现在targetlist(查询返回列表)里，对于t1.c1=t2.c1不匹配的场景仍然需要输出值，因此使用left-outerjoin关联T1&T2确保t1.c1=t2.c1在不匹配时，子SSQ能够返回不匹配的补空值。

说明

SSQ和CSSQ的解释如下：

- SSQ：ScalarSubQuery一般指返回1行1列scalar值的sublink，简称SSQ。
- CSSQ：Correlated-ScalarSubQuery和SSQ相同不过是指包含相关条件的SSQ。

上述SQL语句可以改写为：

```
with ssq as
(
   select t2.c2 from t2
)
select ssq.c2, t1.c2
from t1 left join ssq on t1.c1 = ssq.c2
where t1.c2 > 10;
```

改写后的执行计划为：

```
                QUERY PLAN
-------------------------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
   ->  Hash Right Join
         Hash Cond: (t2.c2 = t1.c1)
         ->  Streaming(type: REDISTRIBUTE)
               Spawn on: All datanodes
               ->  Seq Scan on t2
         ->  Hash
               ->  Seq Scan on t1
                     Filter: (c2 > 10)
(10 rows)
```

可以看到出现在SSQ返回列表里的相关子查询SSQ，已经被提升成Right Join，从而避免当内表T2较大时出现SubPlan+Broadcast计划导致性能变差。

- 出现在targetlist里的相关子查询无法提升(带count)

  例如：

```
select (select count(*) from t2 where t2.c1=t1.c1) cnt, t1.c1, t3.c1
from t1,t3
where t1.c1=t3.c1 order by cnt, t1.c1;
```

  执行计划为：

```
                          QUERY PLAN
-------------------------------------------------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
   ->  Sort
         Sort Key: ((SubPlan 1)), t1.c1
         ->  Hash Join
               Hash Cond: (t1.c1 = t3.c1)
               ->  Seq Scan on t1
               ->  Hash
                     ->  Seq Scan on t3
```

```
            SubPlan 1
              ->  Aggregate
                    ->  Result
                          Filter: (t2.c1 = t1.c1)
                          ->  Materialize
                                ->  Streaming(type: BROADCAST)
                                      Spawn on: All datanodes
                                      ->  Seq Scan on t2
(17 rows)
```

由于相关子查询出现在targetlist(查询返回列表)里，对于t1.c1=t2.c1不匹配的场景仍然需要输出值，因此使用left-outerjoin关联T1&T2确保t1.c1=t2.c1在不匹配时子SSQ能够返回不匹配的补空值，但是这里带了count语句及时在t1.c1=t2.t1不匹配时需要输出0，因此可以使用一个case-when NULL then 0 else count(*)来代替。

上述SQL语句可以改写为：

```
with ssq as
(
   select count(*) cnt, c1 from t2 group by c1
)
select case when
          ssq.cnt is null then 0
          else ssq.cnt
      end cnt, t1.c1, t3.c1
from t1 left join ssq on ssq.c1 = t1.c1,t3
where t1.c1 = t3.c1
order by ssq.cnt, t1.c1;
```

改写后的执行计划为：

```
 QUERY PLAN
-----------------------------------------------------
Streaming (type: GATHER)
  Node/s: All datanodes
  ->  Sort
        Sort Key: (count(*)), t1.c1
        ->  Hash Join
              Hash Cond: (t1.c1 = t3.c1)
              ->  Hash Left Join
                    Hash Cond: (t1.c1 = t2.c1)
                    ->  Seq Scan on t1
                    ->  Hash
                          ->  HashAggregate
                                Group By Key: t2.c1
                                ->  Seq Scan on t2
              ->  Hash
                    ->  Seq Scan on t3
(15 rows)
```

- 相关条件为不等值场景

  例如：

  ```
  select t1.c1, t1.c2
  from t1
  where t1.c1 = (select agg() from t2.c2 > t1.c2);
  ```

  对于非等值相关条件的SubLink目前无法提升，从语义上可以通过做2次join（一次CorrelationKey，一次rownum自关联）达到提升改写的目的。

  改写方案有两种：

  - 子查询改写方式

    ```
    select t1.c1, t1.c2
    from t1, (
       select t1.rowid, agg() aggref
       from t1,t2
       where t1.c2 > t2.c2 group by t1.rowid
    ) dt /* derived table */
    where t1.rowid = dt.rowid AND t1.c1 = dt.aggref;
    ```

- CTE改写方式

```
WITH dt as
(
   select t1.rowid, agg() aggref
   from t1,t2
   where t1.c2 > t2.c2 group by t1.rowid
)
select t1.c1, t1.c2
from t1, derived_table
where t1.rowid = derived_table.rowid AND
t1.c1 = derived_table.aggref;
```

须知

- 目前GaussDB(DWS)尚无高效的实现表、中间结果集的全局唯一rowid因此目前此类场景很难改写，建议通过业务层进行规避，或者可以使用t1.xc_node_id + t1.ctid进行rowid关联，但是xc_node_id的重复率较高会导致join关联效率变低，而xc_node_id+ctid类型无法作为hashjoin的关联条件。
- 对于AGG类型为count(*)时需要进行CASE-WHEN对没有match的场景补0处理，非COUNT(*)场景NULL处理。
- CTE改写方式如果有sharescan支持性能上能够更优。

更多优化示例

示例1：修改基表为replicate表，并且在过滤列上创建索引。

```
create table master_table (a int);
create table sub_table(a int, b int);
select a from master_table group by a having a in (select a from sub_table);
```

上述事例中存在一个相关性子查询，为了提升查询的性能，可以将sub_table修改为一个relication表，并且在字段a上创建一个index。

示例2：修改select语句，将子查询修改为和主表的join，或者修改为可以提升的subquery，但是在修改前后需要保证语义的正确性。

```
explain (costs off)select * from master_table as t1 where t1.a in (select t2.a from sub_table as t2 where t1.a
= t2.b);
                      QUERY PLAN
------------------------------------------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
   ->  Seq Scan on master_table t1
         Filter: (SubPlan 1)
         SubPlan 1
           ->  Result
                 Filter: (t1.a = t2.b)
                 ->  Materialize
                       ->  Streaming(type: BROADCAST)
                             Spawn on: All datanodes
                             ->  Seq Scan on sub_table t2
(11 rows)
```

上面事例计划中存在一个subPlan，为了消除这个subPlan可以修改语句为：

```
explain(costs off) select * from master_table as t1 where exists (select t2.a from sub_table as t2 where t1.a
= t2.b and t1.a = t2.a);
                  QUERY PLAN
-------------------------------------------------
 Streaming (type: GATHER)
   Node/s: All datanodes
```

```
  ->  Hash Semi Join
        Hash Cond: (t1.a = t2.b)
        ->  Seq Scan on master_table t1
        ->  Hash
              ->  Streaming(type: REDISTRIBUTE)
                    Spawn on: All datanodes
                    ->  Seq Scan on sub_table t2
(9 rows)
```

从计划可以看出，subPlan消除了，计划变成了两个表的semi join，这样会大大提高执行效率。

4.5.4 统计信息调优

统计信息调优介绍

GaussDB(DWS)是基于代价估算生成的最优执行计划。优化器需要根据analyze收集的统计信息行数估算和代价估算，因此统计信息对优化器行数估算和代价估算起着至关重要的作用。通过ANALYZE收集全局统计信息，主要包括：pg_class表中的relpages和reltuples；pg_statistic表中的stadistinct、stanullfrac、stanumbersN、stavaluesN、histogram_bounds等。

实例分析 1：未收集统计信息导致查询性能差

在很多场景下，由于查询中涉及到的表或列没有收集统计信息，会对查询性能有很大的影响。

表结构如下所示：

```
CREATE TABLE LINEITEM
(
L_ORDERKEY        BIGINT        NOT NULL
, L_PARTKEY       BIGINT        NOT NULL
, L_SUPPKEY       BIGINT        NOT NULL
, L_LINENUMBER    BIGINT        NOT NULL
, L_QUANTITY      DECIMAL(15,2) NOT NULL
, L_EXTENDEDPRICE  DECIMAL(15,2) NOT NULL
, L_DISCOUNT      DECIMAL(15,2) NOT NULL
, L_TAX           DECIMAL(15,2) NOT NULL
, L_RETURNFLAG    CHAR(1)       NOT NULL
, L_LINESTATUS    CHAR(1)       NOT NULL
, L_SHIPDATE      DATE          NOT NULL
, L_COMMITDATE    DATE          NOT NULL
, L_RECEIPTDATE   DATE          NOT NULL
, L_SHIPINSTRUCT  CHAR(25)      NOT NULL
, L_SHIPMODE      CHAR(10)      NOT NULL
, L_COMMENT       VARCHAR(44)   NOT NULL
) with (orientation = column, COMPRESSION = MIDDLE) distribute by hash(L_ORDERKEY);

CREATE TABLE ORDERS
(
O_ORDERKEY        BIGINT        NOT NULL
, O_CUSTKEY       BIGINT        NOT NULL
, O_ORDERSTATUS   CHAR(1)       NOT NULL
, O_TOTALPRICE    DECIMAL(15,2) NOT NULL
, O_ORDERDATE     DATE NOT NULL
, O_ORDERPRIORITY CHAR(15)      NOT NULL
, O_CLERK         CHAR(15)      NOT NULL
, O_SHIPPRIORITY  BIGINT        NOT NULL
, O_COMMENT       VARCHAR(79)   NOT NULL
)with (orientation = column, COMPRESSION = MIDDLE) distribute by hash(O_ORDERKEY);
```

查询语句如下所示：


```
explain verbose select
count(*) as numwait
from
lineitem l1,
orders
where
o_orderkey = l1.l_orderkey
and o_orderstatus = 'F'
and l1.l_receiptdate > l1.l_commitdate
and not exists (
select
*
from
lineitem l3
where
l3.l_orderkey = l1.l_orderkey
and l3.l_suppkey <> l1.l_suppkey
and l3.l_receiptdate > l3.l_commitdate
)
order by
numwait desc;
```

当出现该问题时，可以通过如下方法确认查询中涉及到的表或列有没有做过ANALYZE收集统计信息。

1. 通过explain verbose执行query分析执行计划时会提示WARNING信息，如下所示：

   ```
   WARNING:Statistics in some tables or columns(public.lineitem(l_receiptdate,l_commitdate,l_orderkey, l_suppkey), public.orders(o_orderstatus,o_orderkey)) are not collected.
   HINT:Do analyze for them in order to generate optimized plan.
   ```

2. 可以通过在pg_log目录下的日志文件中查找以下信息来确认是当前执行的query是否由于没有收集统计信息导致查询性能变差。

   ```
   2017-06-14 17:28:30.336 CST 140644024579856 20971684 [BACKEND] LOG:Statistics in some tables or columns(public.lineitem(l_receiptdate, l_commitdate,l_orderkey,
   .l_suppkey), public.orders(o_orderstatus,o_orderkey)) are not collected.
   2017-06-14 17:28:30.336 CST 140644024579856 20971684 [BACKEND] HINT:Do analyze for them in order to generate optimized plan.
   ```

当通过以上方法查看到哪些表或列没有执行ANALYZE，可以通过对WARNING或日志中上报的表或列执行ANALYZE可以解决由于为收集统计信息导致查询变慢的问题。

实例分析 2：设置 cost_param 对查询性能优化

请参考**案例：设置cost_param对查询性能优化**。

实例分析 3：多表 join 的复杂查询存在中间结果不准调优

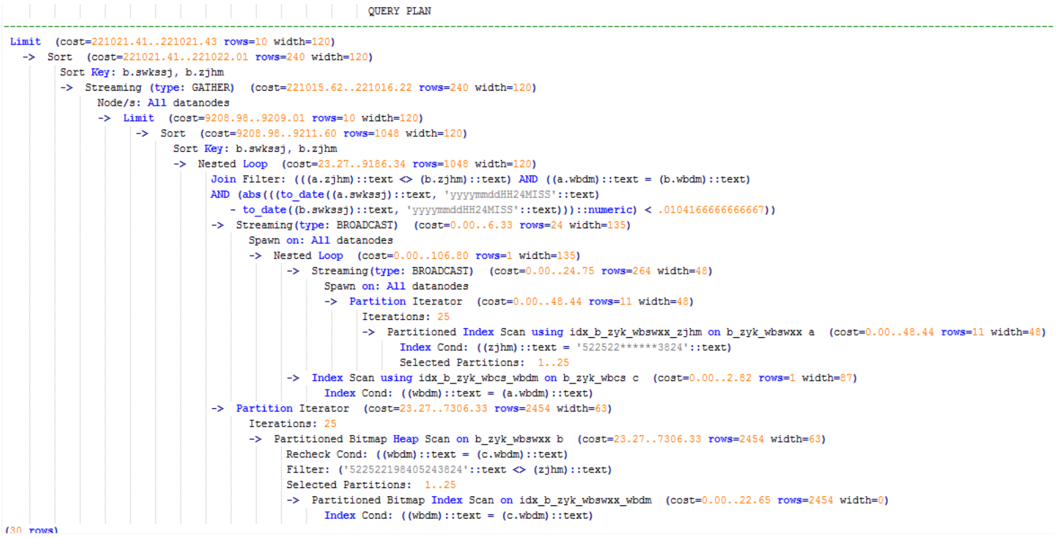
现象描述：查询与指定人在前后15分钟内、同一网吧登记上网的人员信息：

```
SELECT
C.WBM,
C.DZQH,
C.DZ,
B.ZJHM,
B.SWKSSJ,
B.XWSJ
FROM
b_zyk_wbswxx A,
b_zyk_wbswxx B,
b_zyk_wbcs C
WHERE
A.ZJHM = '522522******3824'
AND A.WBDM = B.WBDM
AND A.WBDM = C.WBDM
AND abs(to_date(A.SWKSSJ,'yyyymmddHH24MISS') - to_date(B.SWKSSJ,'yyyymmddHH24MISS')) <
```

```
INTERVAL '15 MINUTES'
ORDER BY
B.SWKSSJ,
B.ZJHM
limit 10 offset 0
;
```

执行计划如图4-3所示。该查询实际耗时约12秒。

图 4-3 应用 unlogged table 案例（一）



```
QUERY PLAN
------------------------------------------------------------------------------------------
Limit  (cost=221021.41..221021.43 rows=10 width=120)
  ->  Sort  (cost=221021.41..221022.01 rows=240 width=120)
        Sort Key: b.swkssj, b.zjhm
        ->  Streaming (type: GATHER)  (cost=221015.62..221016.22 rows=240 width=120)
              Node/s: All datanodes
              ->  Limit  (cost=9208.98..9209.01 rows=10 width=120)
                    ->  Sort  (cost=9208.98..9211.60 rows=1048 width=120)
                          Sort Key: b.swkssj, b.zjhm
                          ->  Nested Loop  (cost=23.27..9186.34 rows=1048 width=120)
                                Join Filter: (((a.zjhm)::text <> (b.zjhm)::text) AND ((a.wbdm)::text = (b.wbdm)::text)
                                AND (abs(((to_date((a.swkssj)::text, 'yyyymmddHH24MISS'::text)
                                   - to_date((b.swkssj)::text, 'yyyymmddHH24MISS'::text)))::numeric) < .0104166666666667))
                                ->  Streaming(type: BROADCAST)  (cost=0.00..6.33 rows=24 width=135)
                                      Spawn on: All datanodes
                                      ->  Nested Loop  (cost=0.00..106.80 rows=1 width=135)
                                            ->  Streaming(type: BROADCAST)  (cost=0.00..24.75 rows=264 width=48)
                                                  Spawn on: All datanodes
                                                  ->  Partition Iterator  (cost=0.00..48.44 rows=11 width=48)
                                                        Iterations: 25
                                                        ->  Partitioned Index Scan using idx_b_zyk_wbswxx_zjhm on b_zyk_wbswxx a  (cost=0.00..48.44 rows=11 width=48)
                                                              Index Cond: ((zjhm)::text = '522522******3824'::text)
                                                              Selected Partitions:  1..25
                                            ->  Index Scan using idx_b_zyk_wbcs_wbdm on b_zyk_wbcs c  (cost=0.00..2.82 rows=1 width=87)
                                                  Index Cond: ((wbdm)::text = (a.wbdm)::text)
                                ->  Partition Iterator  (cost=23.27..7306.33 rows=2454 width=63)
                                      Iterations: 25
                                      ->  Partitioned Bitmap Heap Scan on b_zyk_wbswxx b  (cost=23.27..7306.33 rows=2454 width=63)
                                            Recheck Cond: ((wbdm)::text = (c.wbdm)::text)
                                            Filter: ('522522198405243824'::text <> (zjhm)::text)
                                            Selected Partitions:  1..25
                                            ->  Partitioned Bitmap Index Scan on idx_b_zyk_wbswxx_wbdm  (cost=0.00..22.65 rows=2454 width=0)
                                                  Index Cond: ((wbdm)::text = (c.wbdm)::text)
(30 rows)
```

优化分析：分析过程如下：

1. 分析该执行计划发现，扫描节点已使用Index Scan，耗时主要在最外层Nest Loop Join的Join Filter计算中，且该计算执行了字符串的加减法和不等值比较。
2. 考虑使用unlogged table保存目标人的上网信息，且在插入时处理上网开始时间和终止时间，以避免后续进行时间加减。

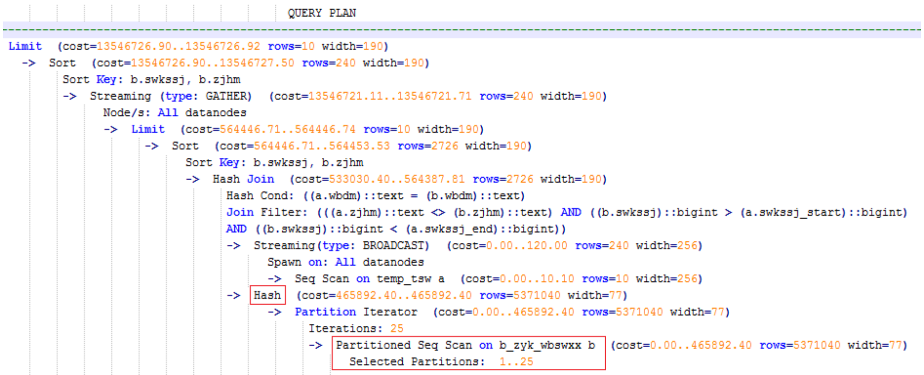
```
//创建临时unlogged table
CREATE UNLOGGED TABLE temp_tsw
(
ZJHM         NVARCHAR2(18),
WBDM         NVARCHAR2(14),
SWKSSJ_START NVARCHAR2(14),
SWKSSJ_END   NVARCHAR2(14),
WBM          NVARCHAR2(70),
DZQH         NVARCHAR2(6),
DZ           NVARCHAR2(70),
IPDZ         NVARCHAR2(39)
)
;
//插入目标人的上网记录，并处理上网开始和结束时间。
INSERT INTO
temp_tsw
SELECT
A.ZJHM,
A.WBDM,
to_char((to_date(A.SWKSSJ,'yyyymmddHH24MISS') - INTERVAL '15 
MINUTES'),'yyyymmddHH24MISS'),
to_char((to_date(A.SWKSSJ,'yyyymmddHH24MISS') + INTERVAL '15 
MINUTES'),'yyyymmddHH24MISS'),
B.WBM,B.DZQH,B.DZ,B.IPDZ
FROM
b_zyk_wbswxx A,
b_zyk_wbcs B
```

```
WHERE
A.ZJHM='522522******3824' AND A.WBDM = B.WBDM
;

//查询和目标人在前后十五分钟内在同一网吧上网的人员信息，比较大小时强制转换为int8。
SELECT
A.WBM,
A.DZQH,
A.DZ,
A.IPDZ,
B.ZJHM,
B.XM,
to_date(B.SWKSSJ,'yyyymmddHH24MISS') as SWKSSJ,
to_date(B.XWSJ,'yyyymmddHH24MISS') as XWSJ,
B.SWZDH
FROM temp_tsw A,
b_zyk_wbswxx B
WHERE
A.ZJHM <> B.ZJHM
AND A.WBDM = B.WBDM
AND (B.SWKSSJ)::int8 > (A.swkssj_start)::int8
AND (B.SWKSSJ)::int8 < (A.swkssj_end)::int8
order by
B.SWKSSJ,
B.ZJHM
limit 10 offset 0
;
```

上述查询耗时约7秒，执行计划如图4-4所示。

图 4-4 应用 unlogged table 案例（二）



```
QUERY PLAN
--------------------------------------------------------------------------------
Limit  (cost=13546726.90..13546726.92 rows=10 width=190)
  ->  Sort  (cost=13546726.90..13546727.50 rows=240 width=190)
        Sort Key: b.swkssj, b.zjhm
        ->  Streaming (type: GATHER)  (cost=13546721.11..13546721.71 rows=240 width=190)
              Node/s: All datanodes
              ->  Limit  (cost=564446.71..564446.74 rows=10 width=190)
                    ->  Sort  (cost=564446.71..564453.53 rows=2726 width=190)
                          Sort Key: b.swkssj, b.zjhm
                          ->  Hash Join  (cost=533030.40..564387.81 rows=2726 width=190)
                                Hash Cond: ((a.wbdm)::text = (b.wbdm)::text)
                                Join Filter: (((a.zjhm)::text <> (b.zjhm)::text) AND ((b.swkssj)::bigint > (a.swkssj_start)::bigint)
                                AND ((b.swkssj)::bigint < (a.swkssj_end)::bigint))
                                ->  Streaming(type: BROADCAST)  (cost=0.00..120.00 rows=240 width=256)
                                      Spawn on: All datanodes
                                      ->  Seq Scan on temp_tsw a  (cost=0.00..10.10 rows=10 width=256)
                                ->  Hash  (cost=465892.40..465892.40 rows=5371040 width=77)
                                      ->  Partition Iterator  (cost=0.00..465892.40 rows=5371040 width=77)
                                            Iterations: 25
                                            ->  Partitioned Seq Scan on b_zyk_wbswxx b  (cost=0.00..465892.40 rows=5371040 width=77)
                                                  Selected Partitions:  1..25
```

3. 分析上述执行计划，发现执行了Hash Join，对大表b_zyk_wbswxx建立了Hash Table。由于该表数据量大，创建过程耗时较长。

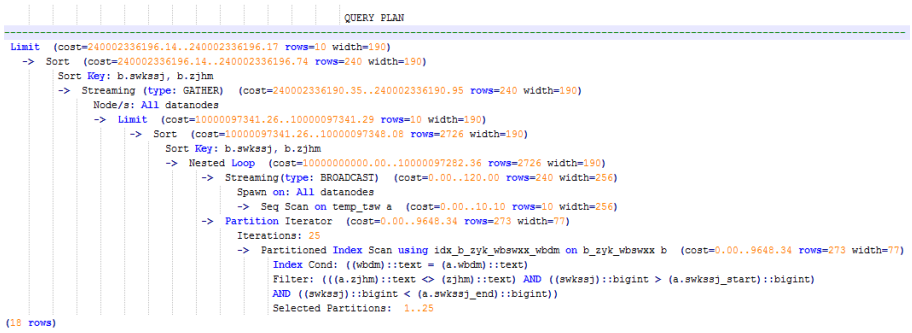
   由于temp_tsw中仅包含几百条记录，且temp_tsw和b_zyk_wbswxx均通过wbdm（网吧代码）执行等值连接。因此，如果Join方式改为Nest Loop Join，则扫描节点可以实现Index Scan，性能预计将会提升。

4. 执行如下语句，将Join方式改为Nest Loop Join。

   ```
   SET enable_hashjoin = off;
   ```

   执行计划如图4-5所示。查询耗时约3秒。

图 4-5 应用 unlogged table 案例（三）



5. 使用unlogged table保存结果集并用于分页显示。

   如果需要在上层应用页面实现分页显示，需要修改offset值确定显示目标页的结果集。按此实现，每次翻页时均执行上面查询语句，耗时较长。

   为解决上述问题，建议使用unlogged table保存结果集。

```
//创建保存结果集的unlogged table
CREATE UNLOGGED TABLE temp_result
(
WBM      NVARCHAR2(70),
DZQH     NVARCHAR2(6),
DZ       NVARCHAR2(70),
IPDZ     NVARCHAR2(39),
ZJHM     NVARCHAR2(18),
XM       NVARCHAR2(30),
SWKSSJ   date,
XWSJ     date,
SWZDH    NVARCHAR2(32)
);

//将结果集插入unlogged table，插入耗时约3秒。
INSERT INTO
temp_result
SELECT
A.WBM,
A.DZQH,
A.DZ,
A.IPDZ,
B.ZJHM,
B.XM,
to_date(B.SWKSSJ,'yyyymmddHH24MISS') as SWKSSJ,
to_date(B.XWSJ,'yyyymmddHH24MISS') as XWSJ,
B.SWZDH
FROM temp_tsw A,
b_zyk_wbswxx B
WHERE
A.ZJHM <> B.ZJHM
AND A.WBDM = B.WBDM
AND (B.SWKSSJ)::int8 > (A.swkssj_start)::int8
AND (B.SWKSSJ)::int8 < (A.swkssj_end)::int8
;

//查询结果集表进行分页显示，分页查询耗时约10ms。
SELECT
*
FROM
temp_result
ORDER BY
SWKSSJ,
ZJHM
LIMIT 10 OFFSET 0;
```


⚠ **注意**

通过ANALYZE收集全局统计信息，通常会改善查询性能。

如果遇到性能问题：可以使用plan hint来调整到之前的查询计划，详情请参见**使用Plan Hint进行调优**。

4.5.5 算子级调优

算子级调优介绍

一个查询语句要经过多个算子步骤才会输出最终的结果。由于个别算子耗时过长导致整体查询性能下降的情况比较常见。这些算子是整个查询的瓶颈算子。通用的优化手段是EXPLAIN ANALYZE/PERFORMANCE命令查看执行过程的瓶颈算子，然后进行针对性优化。

如下面的执行过程信息中，Hashagg算子的执行时间占总时间的：(51016-13535)/56476 ≈66%，此处Hashagg算子就是这个查询的瓶颈算子，在进行性能优化时应当优先考虑此算子的优化。

```
 id |                         operation                              |         A-time          |  A-rows   |  E-rows   |      Peak Memory       | E-memory | A-width | E-width |   E-costs
----+----------------------------------------------------------------+-------------------------+-----------+-----------+------------------------+----------+---------+---------+-------------
  1 | ->  Row Adapter                                                | 56476.397               |  10000000 |    237060 | 19KB                   |          |         |      20 | 20933222.75
  2 |   ->  Vector Streaming (type: GATHER)                          | 55664.220               |  10000000 |    237060 | 243KB                  |          |         |      20 | 20933222.75
  3 |     ->  Vector Hash Aggregate                                  | [55124.685,55132.180]   |  10000000 |    237060 | [29349KB, 29441KB]     | 16MB     | [20,20] |      20 | 20918406.50
  4 |       ->  Vector Streaming(type: REDISTRIBUTE)                 | [52519.781,53709.779]   | 339364604 |   4856184 | [1219KB, 1219KB]       | 1MB      |         |      20 | 10461210.85
  5 |         ->  Vector Hash Aggregate                              | [35675.636,51016.424]   | 339364604 |   4856184 | [732850KB, 746894KB]   | 16MB     | [20,20] |      20 | 10457195.65
  6 |           ->  Vector Partition Iterator                        | [9035.202,13565.884]    | 970000000 | 935838097 | [9KB, 9KB]             | 1MB      |         |      20 | 10195891.68
  7 |             ->  Partitioned CStore Scan on xuji.e_mp_day_energy_csv_1 | [9015.645,13535.346] | 970000000 | 935838097 | [845KB, 845KB]   | 1MB      |         |      20 | 10195891.68
(7 rows)
```

算子级调优示例

示例1：基表扫描时，对于点查或者范围扫描等过滤大量数据的查询，如果使用SeqScan全表扫描会比较耗时，可以在条件列上建立索引选择IndexScan进行索引扫描提升扫描效率。

```
explain (analyze on, costs off) select * from store_sales where ss_sold_date_sk = 2450944;
id |             operation          |       A-time        | A-rows | Peak Memory  | A-width
----+--------------------------------+---------------------+--------+--------------+---------
 1 | ->  Streaming (type: GATHER)   | 3666.020            |   3360 | 195KB        |
 2 |    ->  Seq Scan on store_sales | [3594.611,3594.611] |   3360 | [34KB, 34KB] |

Predicate Information (identified by plan id)
-----------------------------------------------
   2 --Seq Scan on store_sales
         Filter: (ss_sold_date_sk = 2450944)
         Rows Removed by Filter: 4968936
create index idx on store_sales_row(ss_sold_date_sk);
CREATE INDEX
explain (analyze on, costs off) select * from store_sales_row where ss_sold_date_sk = 2450944;
id |                    operation                      |     A-time      | A-rows | Peak Memory  | A-width
----+------------------------------------------------+-----------------+--------+--------------+----------
 1 | ->  Streaming (type: GATHER)                   | 81.524          |   3360 | 195KB        |
 2 |    ->  Index Scan using idx on store_sales_row | [13.352,13.352] |   3360 | [34KB, 34KB] |
```

上述例子中，全表扫描返回3360条数据，过滤掉大量数据，在ss_sold_date_sk列上建立索引后，使用IndexScan扫描效率显著提高，从3.6秒提升到13毫秒。

示例2：如果从执行计划中看，两表join选择了NestLoop，而实际行数比较大时，NestLoop Join可能执行比较慢。如下的例子中NestLoop耗时181秒，如果设置参数enable_mergejoin=off关掉Merge Join，同时设置参数enable_nestloop=off关掉NestLoop，让优化器选择HashJoin，则Join耗时提升至200多毫秒。

```
postgres=#  explain analyze select count(*) from store_sales ss, item i where ss.ss_item_sk = i.i_item_sk;
 id |                    operation                    |          A-time          |  A-rows   | E-rows  |  Peak Memory   | E-memory | A-width | E-width |   E-costs
----+-------------------------------------------------+--------------------------+-----------+---------+----------------+----------+---------+---------+-------------
  1 | ->  Row Adapter                                 | [184300.301]             |         1 |       1 | [11KB]         |          |         |       0 | [48629179.77]
  2 |    ->  Vector Aggregate                         | [184300.280]             |         1 |       1 | [181KB]        |          |         |       0 | [48629179.77]
  3 |       ->  Vector Streaming (type: GATHER)       | [184300.186]             |         4 |       4 | [188KB]        |          |         |       0 | [48629179.77]
  4 |          ->  Vector Aggregate                   | [165575.384,184252.368]  |         4 |       4 | [140KB, 140KB] | 1MB      |         |       0 | [48629179.61]
  5 |             ->  Vector Nest Loop (6,7)          | [162918.848,181438.162]  |   2880404 | 2880404 | [74KB, 74KB]   | 1MB      |         |       0 | [48627379.35]
  6 |                ->  CStore Scan on store_sales ss | [15.660,16.229]          |   2880404 | 2880404 | [490KB, 490KB] | 1MB      |         |       4 | [16683.10]
  7 |                ->  Vector Materialize           | [118314.521,132478.454]  | 12968211302 |   18000 | [869KB, 900KB] | 16MB     | [8,8]   |       4 | [3890.00]
  8 |                   ->  CStore Scan on item i     | [0.234,0.243]            |     18000 |   18000 | [476KB, 476KB] | 1MB      |         |       4 | [3867.50]
(8 rows)
```

```
postgres=#  set enable_nestloop=off;
SET
postgres=#  set enable_mergejoin=off;
SET
ppostgres=# explain analyze select count(*) fpostgres=# ales ss, item i where ss.ss_item_sk = i.i_item_sk;
 id |                    operation                    |          A-time          | A-rows  | E-rows  |  Peak Memory   | E-memory | A-width | E-width | E-costs
----+-------------------------------------------------+--------------------------+---------+---------+----------------+----------+---------+---------+-----------
  1 | ->  Row Adapter                                 | [291.066]                |       1 |       1 | [11KB]         |          |         |       0 | [32308.66]
  2 |    ->  Vector Aggregate                         | [291.052]                |       1 |       1 | [181KB]        |          |         |       0 | [32308.66]
  3 |       ->  Vector Streaming (type: GATHER)       | [290.973]                |       4 |       4 | [188KB]        |          |         |       0 | [32308.66]
  4 |          ->  Vector Aggregate                   | [220.792,234.532]        |       4 |       4 | [140KB, 140KB] | 1MB      |         |       0 | [32308.50]
  5 |             ->  Vector Hash Join (6,7)          | [209.987,223.345]        | 2880404 | 2880404 | [236KB, 241KB] | 16MB     | [8,8]   |       0 | [30508.24]
  6 |                ->  CStore Scan on store_sales ss | [13.132,13.717]          | 2880404 | 2880404 | [490KB, 490KB] | 1MB      |         |       4 | [16683.10]
  7 |                ->  CStore Scan on item i        | [0.214,0.246]            |   18000 |   18000 | [477KB, 477KB] | 1MB      |         |       4 | [3867.50]
(7 rows)
```

示例3：通常情况下Agg选择HashAgg性能较好，如果大结果集选择了Sort +GroupAgg，则需要设置enable_sort=off，HashAgg耗时明显优于Sort+GroupAgg。

```
postgres=#  explain analyze select count(*) from store_sales group by ss_item_sk;
 id |                    operation                    |          A-time          | A-rows  | E-rows  |      Peak Memory      | E-memory | A-width | E-width | E-costs
----+-------------------------------------------------+--------------------------+---------+---------+-----------------------+----------+---------+---------+-----------
  1 | ->  Row Adapter                                 | [1977.385]               |   18000 |   17644 | [20KB]                |          |         |       4 | [92875.24]
  2 |    ->  Vector Streaming (type: GATHER)          | [1973.617]               |   18000 |   17644 | [1946KB]              |          |         |       4 | [92875.24]
  3 |       ->  Vector Sort Aggregate                 | [1784.800,1883.243]      |   18000 |   17644 | [273KB, 273KB]        | 1MB      |         |       4 | [92186.02]
  4 |          ->  Vector Sort                        | [1752.270,1848.357]      | 2880404 | 2880404 | [12846466KB, 135135KB] | 16MB     | [8,8]   |       4 | [88541.40]
  5 |             ->  CStore Scan on store_sales      | [12.483,13.548]          | 2880404 | 2880404 | [490KB, 490KB]        | 1MB      |         |       4 | [16683.10]
(5 rows)
```

```
postgres=#  set enable_sort=off;
SET
postgres=#  explain analyze select count(*) from store_sales group by ss_item_sk;
 id |                    operation                    |          A-time          | A-rows  | E-rows  |      Peak Memory      | E-memory | A-width | E-width | E-costs
----+-------------------------------------------------+--------------------------+---------+---------+-----------------------+----------+---------+---------+-----------
  1 | ->  Row Adapter                                 | [838.218]                |   18000 |   17644 | [20KB]                |          |         |       4 | [21016.93]
  2 |    ->  Vector Streaming (type: GATHER)          | [834.264]                |   18000 |   17644 | [228KB]               |          |         |       4 | [21016.93]
  3 |       ->  Vector Hash Aggregate                 | [585.017,758.204]        |   18000 |   17644 | [262552KB, 262564KB] | 16MB     | [8,8]   |       4 | [20327.72]
  4 |          ->  CStore Scan on store_sales         | [12.540,13.941]          | 2880404 | 2880404 | [490KB, 490KB]        | 1MB      |         |       4 | [16683.10]
(4 rows)
```

4.5.6 数据倾斜调优

数据倾斜问题是分布式架构的重要难题，它破坏了MPP架构中各个节点对等的要求，导致单节点（倾斜节点）所存储或者计算的数据量远大于其他节点，所以会造成以下危害：

- 存储上的倾斜会严重限制系统容量，在系统容量不饱和的情况下，由于单节点倾斜的限制，使得整个系统容量无法继续增长。
- 计算上的倾斜会严重影响系统性能，由于倾斜节点所需要运算的数据量远大于其它节点，导致倾斜节点降低系统整体性能。
- 数据倾斜还严重影响了MPP架构的扩展性。由于在存储或者计算时，往往会将相同值的数据放到同一节点，因此当倾斜数据（大量数据的值相同）出现之后，即使增加节点，系统瓶颈仍然受限于倾斜节点的容量或者性能。

GaussDB(DWS)数据库针对数据倾斜问题给出了完整的解决方案，包括存储倾斜和计算倾斜两大问题，下面分别进行介绍。

存储层数据倾斜

GaussDB(DWS)数据库中，数据分布存储在各个DN上，通过分布式执行提高查询的效率。但是，如果数据分布存在倾斜，则会导致分布式执行某些DN成为瓶颈，影响查询性能。这种情况通常是由于分布列选择不合理，可以通过调整分布列的方式解决。

例如下例：

```
explain performance select count(*) from inventory;
5 --CStore Scan on lmz.inventory
        dn_6001_6002 (actual time=0.444..83.127 rows=42000000 loops=1)
        dn_6003_6004 (actual time=0.512..63.554 rows=27000000 loops=1)
        dn_6005_6006 (actual time=0.722..99.033 rows=45000000 loops=1)
        dn_6007_6008 (actual time=0.529..100.379 rows=51000000 loops=1)
```

```
      dn_6009_6010 (actual time=0.382..71.341 rows=36000000 loops=1)
      dn_6011_6012 (actual time=0.547..100.274 rows=51000000 loops=1)
      dn_6013_6014 (actual time=0.596..118.289 rows=60000000 loops=1)
      dn_6015_6016 (actual time=1.057..132.346 rows=63000000 loops=1)
      dn_6017_6018 (actual time=0.940..110.310 rows=54000000 loops=1)
      dn_6019_6020 (actual time=0.231..41.198 rows=21000000 loops=1)
      dn_6021_6022 (actual time=0.927..114.538 rows=54000000 loops=1)
      dn_6023_6024 (actual time=0.637..118.385 rows=60000000 loops=1)
      dn_6025_6026 (actual time=0.288..32.240 rows=15000000 loops=1)
      dn_6027_6028 (actual time=0.566..118.096 rows=60000000 loops=1)
      dn_6029_6030 (actual time=0.423..82.913 rows=42000000 loops=1)
      dn_6031_6032 (actual time=0.395..78.103 rows=39000000 loops=1)
      dn_6033_6034 (actual time=0.376..51.052 rows=24000000 loops=1)
      dn_6035_6036 (actual time=0.569..79.463 rows=39000000 loops=1)
```

在performance信息中，可以看到inventory表各DN的scan行数，发现各DN的行数差距较大，最大的为63000000，最小的只有15000000，差了4倍。这个差距对于数据扫描的性能影响还可以接受，但如果上层有join算子，则影响较大。

通常，数据表在各DN上是hash分布的，因此分布列的选择很重要。通过table_skewness()来查看上述inventory表在各DN的数据分布倾斜，查询结果如下：

```
select table_skewness('inventory');
              table_skewness
-------------------------------------------
 ("dn_6015_6016       ",63000000,8.046%)
 ("dn_6013_6014       ",60000000,7.663%)
 ("dn_6023_6024       ",60000000,7.663%)
 ("dn_6027_6028       ",60000000,7.663%)
 ("dn_6017_6018       ",54000000,6.897%)
 ("dn_6021_6022       ",54000000,6.897%)
 ("dn_6007_6008       ",51000000,6.513%)
 ("dn_6011_6012       ",51000000,6.513%)
 ("dn_6005_6006       ",45000000,5.747%)
 ("dn_6001_6002       ",42000000,5.364%)
 ("dn_6029_6030       ",42000000,5.364%)
 ("dn_6031_6032       ",39000000,4.981%)
 ("dn_6035_6036       ",39000000,4.981%)
 ("dn_6009_6010       ",36000000,4.598%)
 ("dn_6003_6004       ",27000000,3.448%)
 ("dn_6033_6034       ",24000000,3.065%)
 ("dn_6019_6020       ",21000000,2.682%)
 ("dn_6025_6026       ",15000000,1.916%)
(18 rows)
```

通过查询建表定义，可以发现，目前该表是以inv_date_sk作为分布列的，导致存在倾斜。通过查看各列的数据分布情况，改为inv_item_sk作为分布列，则倾斜情况分布如下：

```
select table_skewness('inventory');
              table_skewness
-------------------------------------------
 ("dn_6001_6002       ",43934200,5.611%)
 ("dn_6007_6008       ",43829420,5.598%)
 ("dn_6003_6004       ",43781960,5.592%)
 ("dn_6031_6032       ",43773880,5.591%)
 ("dn_6033_6034       ",43763280,5.589%)
 ("dn_6011_6012       ",43683600,5.579%)
 ("dn_6013_6014       ",43551660,5.562%)
 ("dn_6027_6028       ",43546340,5.561%)
 ("dn_6009_6010       ",43508700,5.557%)
 ("dn_6023_6024       ",43484540,5.554%)
 ("dn_6019_6020       ",43466800,5.551%)
 ("dn_6021_6022       ",43458500,5.550%)
 ("dn_6017_6018       ",43448040,5.549%)
 ("dn_6015_6016       ",43247700,5.523%)
 ("dn_6005_6006       ",43200240,5.517%)
 ("dn_6029_6030       ",43181360,5.515%)
```

```
 ("dn_6025_6026        ",43179700,5.515%)
 ("dn_6035_6036        ",42960080,5.487%)
(18 rows)
```

数据分布倾斜的问题得到解决。

除了table_skewness()视图外，当前版本还提供了table_distribution函数和**PGXC_GET_TABLE_SKEWNESS**视图，可以更加高效的查询各表的数据倾斜情况。

计算层数据倾斜

即使通过修改表的分布键，使得数据存储在各个节点上是均衡的，但是在执行查询的过程中，仍然可能出现数据倾斜的问题。在运算过程中某个算子在DN上输出的结果集出现倾斜，从而导致此算子上层的运算出现计算倾斜。一般来说，这是由于在执行过程中，数据重分布导致的。

在查询执行的过程中，join key、group by key等往往不是表的分布列，因此需要按照join key、group by key上数据的hash值，让数据在各个DN之间进行重新分布，这个过程对应于计划中的Redistribute算子。当重分布列上的数据存在倾斜时，就会导致运行时的数据倾斜，即重分布后部分节点的数据远远大于其他。倾斜节点需要处理更多的数据，导致倾斜节点的计算性能远远低于其他节点。

如下例中，s表和t表join，join条件中的s.x和t.x均不是表的分布列，因此需要重分布（REDISTRIBUTE算子）。其中s.x列上存在倾斜值，t.x上不存在倾斜。id=6的stream算子在datanode2节点输出的结果集是其他DN的3倍，从而导致了计算倾斜。

```
select * from skew s,test t where s.x = t.x order by s.a limit 1;
 id |                    operation                    |        A-time
----+-------------------------------------------------+-----------------------
  1 | ->  Limit                                       | 52622.382
  2 |    ->  Streaming (type: GATHER)                 | 52622.374
  3 |       ->  Limit                                 | [30138.494,52598.994]
  4 |         ->  Sort                                | [30138.486,52598.986]
  5 |           ->  Hash Join (6,8)                   | [30127.013,41483.275]
  6 |             ->  Streaming(type: REDISTRIBUTE)   | [11365.110,22024.845]
  7 |               ->  Seq Scan on public.skew s     | [2019.168,2175.369]
  8 |             ->  Hash                            | [2460.108,2499.850]
  9 |               ->  Streaming(type: REDISTRIBUTE) | [1056.214,1121.887]
 10 |                 ->  Seq Scan on public.test t   | [310.848,325.569]

6 --Streaming(type: REDISTRIBUTE)
        datanode1 (rows=5050368)
        datanode2 (rows=15276032)
        datanode3 (rows=5174272)
        datanode4 (rows=5219328)
```

和存储倾斜相比，计算倾斜更难以提前识别，因此GaussDB提出了RLBT(Runtime Load Balance Technology)方案，用以解决运行时的计算倾斜问题，该特性由参数**skew_option**控制。RLBT方案主要分为两个层面，第一步是计算倾斜识别，第二步是计算倾斜解决。下面分别进行介绍。

1. 倾斜识别

   计算倾斜的识别，即预先识别计算过程中的重分布列是否存在倾斜数据。RLBT方案中给出了三个解决手段，统计信息识别，hint方式指定以及规则识别：

   - 统计信息识别

     需要用户先执行ANALYZE收集各表的统计信息，然后优化器能够自动利用统计信息对重分布键上的倾斜数据进行提前识别，对于存在倾斜的查询，生成相应的优化计划。在重分布键有多列的情况，只有所有列都属于同一个基表才能利用统计信息进行识别。

统计信息只能给出基表的倾斜情况，当基表某一列存在倾斜，其他列上带有过滤条件，或者经过和其他表的join之后，无法准确判断倾斜列上倾斜数据是否依旧存在。当**skew_option**为normal时，这里认为倾斜数据依旧存在，仍然会对基表中识别到的倾斜进行优化；当**skew_option**为lazy时，这里认为倾斜数据已经不再存在，也就不会进行相应的优化。

- hint方式指定

  统计信息有着一定的局限性，对于较为复杂的查询，其中间结果难以通过统计信息进行估算和识别倾斜数据。对于这种情况，GaussDB(DWS)设计了hint手段，通过用户手动指定的方式，给定倾斜信息。优化器根据用户给定的倾斜信息，来对查询进行优化。详细hint使用语法参见**运行倾斜的hint**。

- 规则识别

  现在BI系统往往会产生大量带有outer join（left join、right join、full join）的SQL，outer join在匹配失败的情况下会补空产生大量NULL值，如果接下来在补空列上进行join或者group by操作，就会导致NULL值倾斜。当前RLBT技术会自动识别这种场景，并生成相应的NULL值倾斜优化计划。

2. 计算倾斜解决

   在解决倾斜时，目前针对最常见的join和agg算子进行了优化。

   - join优化

   基本思路是将倾斜数据和非倾斜数据进行隔离处理。主要分为以下三种情况：

   a. join两侧都需要做重分布：

      对倾斜侧做PART_REDISTRIBUTE_PART_ROUNDROBIN，其中对倾斜数据做roundrobin，非倾斜数据做redistribute；

      对非倾斜侧做PART_REDISTRIBUTE_PART_BROADCAST，其中对倾斜数据做broadcast，非倾斜数据做redistribute；

   b. join一侧需要重分布，另一侧不需要重分布：

      对需要重分布的一侧做PART_REDISTRIBUTE_PART_ROUNDROBIN；

      对不需要重分布的一侧做PART_LOCAL_PART_BROADCAST，其中对等于倾斜值的部分做broadcast，其余数据保留在本地。

   c. 对于有补NULL值的表：

      对该表做PART_REDISTRIBUTE_PART_LOCAL，其中将NULL值保留在本地，其余数据做redistribute。

   以前面的查询为例，s.x列上存在倾斜数据，倾斜数据的值为0。优化器通过统计信息，识别到了该倾斜数据，生成了倾斜优化计划如下：

```
 id |                                operation                                 |        A-time
----+--------------------------------------------------------------------------+-----------------------
  1 | ->  Limit                                                                | 23642.049
  2 |    ->  Streaming (type: GATHER)                                          | 23642.041
  3 |       ->  Limit                                                          | [23310.768,23618.021]
  4 |          ->  Sort                                                        | [23310.761,23618.012]
  5 |             ->  Hash Join (6,8)                                          | [20898.341,21115.272]
  6 |                ->  Streaming(type: PART REDISTRIBUTE PART ROUNDROBIN)    |
[7125.834,7472.111]
  7 |                   ->  Seq Scan on public.skew s                          | [1837.079,1911.025]
  8 |                ->  Hash                                                  | [2612.484,2640.572]
  9 |                   ->  Streaming(type: PART REDISTRIBUTE PART BROADCAST)  | [1193.548,1297.894]
 10 |                      ->  Seq Scan on public.test t                       | [314.343,328.707]

  5 --Vector Hash Join (6,8)
      Hash Cond: s.x = t.x
      Skew Join Optimizated by Statistic
  6 --Streaming(type: PART REDISTRIBUTE PART ROUNDROBIN)
      datanode1  (rows=7635968)
```

```
        datanode2 (rows=7517184)
        datanode3 (rows=7748608)
        datanode4 (rows=7818240)
```

上述执行计划中，可以看到Skew Join Optimizated by Statistic的字样，代表该计划为倾斜优化计划，其中Statistic关键字代表该倾斜优化来自于统计信息，除此之外还有Hint和Rule，分别代表倾斜优化来自于hint语句和规则。对比前面的计划可以看到，这里对于非倾斜数据和倾斜数据做了分别处理。对于s表中的非倾斜数据，依旧按照原有的方案，根据数据的hash值进行重分布；而对于倾斜数据（即等于0的数据），则通过轮询发送的方式，均衡地发送到所有节点。通过这样的方式，解决了倾斜数据分布不均衡的问题。

同时，为了保证结果的正确性，需要对t表做相应的处理。对于t表中等于0（s.x表中的倾斜值）的数据做广播，对于其他数据，依旧根据数据的hash值进行重分布。

通过这样的方式，就解决了join操作中，数据倾斜的问题。从上面的结果来看，id=6的stream算子各个DN的输出结果已经非常均衡，同时查询端到端性能提升了1倍。

如果执行计划中Stream算子类型显示为HYBRID，则表示对不同的倾斜数据所应用的stream方式不同，例如以下计划：

```
EXPLAIN (nodes OFF, costs OFF) SELECT COUNT(*) FROM skew_scol s, skew_scol1 s1 WHERE s.b =
s1.c;
QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------
id |                                                    operation
----
+-----------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------
1 | ->  Aggregate
2 |    ->  Streaming (type: GATHER)
3 |       ->  Aggregate
4 |          ->  Hash Join (5,7)
5 |             ->  Streaming(type: HYBRID)
6 |                ->  Seq Scan on skew_scol s
7 |             ->  Hash
8 |                ->  Streaming(type: HYBRID)
9 |                   ->  Seq Scan on skew_scol1 s1

Predicate Information (identified by plan id)
------------------------------------------------------------------------------------------------------------------------------------------------
--------------------
4 --Hash Join (5,7)
Hash Cond: (s.b = s1.c)
Skew Join Optimized by Statistic
5 --Streaming(type: HYBRID)
Skew Filter: (b = 1)
Skew Filter: (b = 0)
8 --Streaming(type: HYBRID)
Skew Filter: (c = 0)
Skew Filter: (c = 1)
```

数据1在skew_scol表上有倾斜，对倾斜数据做roundrobin，非倾斜数据做redistribute。

数据0在skew_scol表上是非倾斜侧，对倾斜数据做broadcast，非倾斜数据做redistribute。

由此可以看到两个stream类型分别是PART REDISTRIBUTE PART ROUNDROBIN和PART REDISTRIBUTE PART BROADCAST，这里标记stream类型为HYBRID类型。

- agg优化

对于agg操作，解决倾斜的思路与join操作不同，这里是通过首先在本DN内按照group by key进行去重操作，然后再进行重分布。因为经过DN内部去重之后，从

全局来看，每个值的数量都不会超过DN数，因此不会出现严重的数据倾斜问题。以如下query为例：

```
select c1, c2, c3, c4, c5, c6, c7, c8, c9, count(*) from t group by c1, c2, c3, c4, c5, c6, c7, c8, c9 limit 10;
```

原执行结果如下：

```
 id |               operation                |         A-time         | A-rows
----+----------------------------------------+------------------------+----------
  1 | ->  Streaming (type: GATHER)           | 130621.783             |      12
  2 |   ->  GroupAggregate                   | [85499.711,130432.341] |      12
  3 |     ->  Sort                           | [85499.509,103145.632] | 36679237
  4 |       ->  Streaming(type: REDISTRIBUTE) | [25668.897,85499.050]  | 36679237
  5 |         ->  Seq Scan on public.t       | [9835.069,10416.388]   | 36679237

  4 --Streaming(type: REDISTRIBUTE)
      datanode1 (rows=36678837)
      datanode2 (rows=100)
      datanode3 (rows=100)
      datanode4 (rows=200)
```

其中存在大量倾斜数据，导致数据按照group by key进行重分布之后，datanode1的数据量是其他节点的数十万倍。在倾斜优化之后，首先在本DN进行一次group by操作，达到数据去重的效果，然后再进行重分布，可以发现基本没有数据倾斜的问题出现。

```
 id |               operation                |         A-time
----+----------------------------------------+------------------------
  1 | ->  Streaming (type: GATHER)           | 10961.337
  2 |   ->  HashAggregate                    | [10953.014,10953.705]
  3 |     ->  HashAggregate                  | [10952.957,10953.632]
  4 |       ->  Streaming(type: REDISTRIBUTE) | [10952.859,10953.502]
  5 |         ->  HashAggregate              | [10084.280,10947.139]
  6 |           ->  Seq Scan on public.t     | [4757.031,5201.168]

 Predicate Information (identified by plan id)
-----------------------------------------------
  3 --HashAggregate
      Skew Agg Optimized by Statistic

  4 --Streaming(type: REDISTRIBUTE)
      datanode1 (rows=17)
      datanode2 (rows=8)
      datanode3 (rows=8)
      datanode4 (rows=14)
```

适用范围

- join算子
  - 支持nest loop，merge join，hash join等join方式；
  - 当倾斜数据处于join的left侧时，支持inner join，left join，semi join，anti join；当倾斜属于位于join的right侧时，支持inner join，right join，right semi join，right anti join。
  - 通过统计信息得到的倾斜优化计划，优化器会根据代价判断该计划是否为最优计划。通过hint和规则会强制生成倾斜优化计划。
- agg算子
  - array_agg、string_agg、subplan in agg qual这几种场景不支持优化；
  - 通过统计信息识别到的倾斜优化计划会受到代价、plan_mode_seed参数、best_agg_plan参数影响，而hint、规则识别到的不会。

4.6 使用 Plan Hint 进行调优

4.6.1 Plan Hint 调优概述

Plan Hint为用户提供了直接影响执行计划生成的手段，用户可以通过指定join顺序，join、stream、scan方法，指定结果行数，指定重分布过程中的倾斜信息，指定配置参数的值等多个手段来进行执行计划的调优，以提升查询的性能。

功能描述

Plan Hint支持在SELECT、INSERT、UPDATE、MERGE、DELETE关键字后通过如下形式指定：

```
/*+ <plan hint> */
```

可以同时指定多个hint，之间使用空格分隔。hint只能hint当前层的计划，对于子查询计划的hint，需要在子查询对应的关键字后指定hint。

例如：

```
select /*+ <plan_hint1> <plan_hint2> */ * from t1, (select /*+ <plan_hint3> */ from t2) where 1=1;
```

其中<plan_hint1>，<plan_hint2>为外层查询的hint，<plan_hint3>为内层子查询的hint。

须知

如果在视图定义（CREATE VIEW）时指定hint，则在该视图每次被应用时会使用该hint。

当使用random plan功能（参数plan_mode_seed不为0）时，查询指定的plan hint不会被使用。

支持范围

当前版本Plan Hint支持的范围如下，后续版本会进行增强。

- 指定Join顺序的Hint - leading hint。
- 指定Join方式的Hint，仅支持除semi/anti join，unique plan之外的常用hint。
- 指定结果集行数的Hint。
- 指定Stream方式的Hint。
- 指定Scan方式的Hint，仅支持常用的tablescan，indexscan和indexonlyscan的hint。
- 指定子链接块名的Hint。
- 指定倾斜信息的Hint，仅支持Join与HashAgg的重分布过程倾斜。
- 指定Agg重分布列Hint。仅8.1.3.100及以上集群版本支持。
- 指定子查询不提升的Hint。仅8.2.0及以上集群版本支持。
- 指定配置参数值的Hint，仅支持部分配置参数，详见**配置参数的hint**。

注意事项

- 不支持Sort、Setop和Subplan的hint。
- 不支持SMP和Node Group场景下的Hint。
- 不支持对INSERT语句的目标表使用Hint。

示例

本章节使用同一个语句进行示例，便于Plan Hint支持的各方法作对比，示例语句及不带hint的原计划如下所示：

```
explain
select i_product_name product_name
,i_item_sk item_sk
,s_store_name store_name
,s_zip store_zip
,ad2.ca_street_number c_street_number
,ad2.ca_street_name c_street_name
,ad2.ca_city c_city
,ad2.ca_zip c_zip
,count(*) cnt
,sum(ss_wholesale_cost) s1
,sum(ss_list_price) s2
,sum(ss_coupon_amt) s3
FROM   store_sales
,store_returns
,store
,customer
,promotion
,customer_address ad2
,item
WHERE  ss_store_sk = s_store_sk AND
ss_customer_sk = c_customer_sk AND
ss_item_sk = i_item_sk and
ss_item_sk = sr_item_sk and
ss_ticket_number = sr_ticket_number and
c_current_addr_sk = ad2.ca_address_sk and
ss_promo_sk = p_promo_sk and
i_color in ('maroon','burnished','dim','steel','navajo','chocolate') and
i_current_price between 35 and 35 + 10 and
i_current_price between 35 + 1 and 35 + 15
group by i_product_name
,i_item_sk
,s_store_name
,s_zip
,ad2.ca_street_number
,ad2.ca_street_name
,ad2.ca_city
,ad2.ca_zip
;
```

```
 id |                                operation                                 |   E-rows   | E-memory | E-width |  E-costs
----+--------------------------------------------------------------------------+------------+----------+---------+-----------
  1 | ->  Row Adapter                                                          |          6 |          |     273 | 3401632.49
  2 |    ->  Vector Streaming (type: GATHER)                                   |          6 |          |     273 | 3401632.49
  3 |       ->  Vector Hash Aggregate                                          |          6 | 16MB     |     273 | 3401630.82
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)                        |          6 | 1MB      |     169 | 3401630.78
  5 |             ->  Vector Hash Join (6,21)                                  |          6 | 16MB     |     169 | 3401630.42
  6 |                ->  Vector Hash Join (7,20)                               |          7 | 43MB     |     173 | 3400343.15
  7 |                   ->  Vector Streaming(type: REDISTRIBUTE)               |          7 | 1MB      |     123 | 3395775.64
  8 |                      ->  Vector Hash Join (9,19)                         |          7 | 27MB     |     123 | 3395775.48
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)         |          7 | 1MB      |     123 | 3386294.72
 10 |                            ->  Vector Hash Join (11,18)                  |          7 | 16MB     |     123 | 3386294.56
 11 |                               ->  Vector Hash Join (12,14)               |          7 | 19MB     |     112 | 3384018.02
 12 |                                  ->  Vector Partition Iterator           |  287999764 | 1MB      |      12 | 227383.99
 13 |                                     ->  Partitioned CStore Scan on store_returns |  287999764 | 1MB      |      12 | 227383.99
 14 |                                  ->  Vector Hash Join (15,17)            |    1516824 | 16MB     |     124 | 3065686.08
 15 |                                     ->  Vector Partition Iterator        | 2879987999 | 1MB      |      66 | 2756066.50
 16 |                                        ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB      |      66 | 2756066.50
 17 |                                     ->  CStore Scan on item              |        158 | 1MB      |      58 | 4051.25
 18 |                               ->  CStore Scan on store                   |      24048 | 1MB      |      19 | 2264.00
 19 |                         ->  CStore Scan on customer                      |   12000000 | 1MB      |       8 | 12923.00
 20 |                   ->  CStore Scan on customer_address ad2                |    6000000 | 1MB      |      58 | 5770.00
 21 |                ->  CStore Scan on promotion                              |      36000 | 1MB      |       4 | 1268.50
(21 rows)
```

4.6.2 Join 顺序的 Hint

功能描述

指明join的顺序，包括内外表顺序。

语法格式

- 仅指定join顺序，不指定内外表顺序。

```
leading(join_table_list)
```

- 同时指定join顺序和内外表顺序，内外表顺序仅在最外层生效。

```
leading((join_table_list))
```

参数说明

join_table_list为表示表join顺序的hint字符串，可以包含当前层的任意个表（别名），或对于子查询提升的场景，也可以包含子查询的hint别名，同时任意表可以使用括号指定优先级，表之间使用空格分隔。

须知

表只能用单个字符串表示，不能带schema。

表如果存在别名，需要优先使用别名来表示该表。

join table list中指定的表需要满足以下要求，否则会报语义错误。

- list中的表必须在当前层或提升的子查询中存在。
- list中的表在当前层或提升的子查询中必须是唯一的。如果不唯一，需要使用不同的别名进行区分。
- 同一个表只能在list里出现一次。
- 如果表存在别名，则list中的表需要使用别名。

例如：

leading(t1 t2 t3 t4 t5)表示：t1, t2, t3, t4, t5先join，五表join顺序及内外表不限。

leading((t1 t2 t3 t4 t5))表示：t1和t2先join，t2做内表；再和t3 join，t3做内表；再和t4 join，t4做内表；再和t5 join，t5做内表。

leading(t1 (t2 t3 t4) t5)表示：t2, t3, t4先join，内外表不限；再和t1, t5 join，内外表不限。

leading((t1 (t2 t3 t4) t5))表示：t2, t3, t4先join，内外表不限；在最外层，t1再和t2, t3, t4的join表join，t1为外表，再和t5 join，t5为内表。

leading((t1 (t2 t3) t4 t5)) leading((t3 t2))表示：t2, t3先join，t2做内表；然后再和t1 join，t2, t3的join表做内表；然后再依次跟t4, t5做join，t4, t5做内表。

示例

对**示例**中原语句使用如下hint:

```
explain
select /*+ leading((((((store_sales store) promotion) item) customer) ad2) store_returns) leading((store store_sales))*/ i_product_name product_name ...
```

该hint表示：表之间的join关系是：store_sales和store先join， store_sales做内表，然后依次跟promotion, item, customer, ad2, store_returns做join。生成计划如下所示：

```
WARNING:  Duplicated or conflict hint: Leading(store_sales store), will be discarded.
 id |                                operation                                |   E-rows   | E-memory | E-width |   E-costs
----+-------------------------------------------------------------------------+------------+----------+---------+-------------
  1 | ->  Row Adapter                                                         |          6 |          |     273 | 16308094.34
  2 |    ->  Vector Streaming (type: GATHER)                                  |          6 |          |     273 | 16308094.34
  3 |       ->  Vector Hash Aggregate                                         |          6 | 16MB     |     273 | 16308092.67
  4 |          ->  Vector Hash Join (5,20)                                    |          6 | 585MB    |     169 | 16308092.63
  5 |             ->  Vector Streaming(type: REDISTRIBUTE)                    |    1320811 | 1MB      |     181 | 16069870.93
  6 |                ->  Vector Hash Join (7,19)                              |    1320811 | 43MB     |     181 | 16061891.00
  7 |                   ->  Vector Streaming(type: REDISTRIBUTE)              |    1320811 | 1MB      |     131 | 16056566.78
  8 |                      ->  Vector Hash Join (9,18)                        |    1320811 | 27MB     |     131 | 16048586.85
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)        |    1383248 | 1MB      |     131 | 16038321.62
 10 |                            ->  Vector Hash Join (11,17)                 |    1383248 | 16MB     |     131 | 16029664.50
 11 |                               ->  Vector Hash Join (12,16)              | 2626366951 | 16MB     |      73 | 15751384.88
 12 |                                  ->  Vector Hash Join (13,14)           | 2750085660 | 2156MB   |      77 | 14226077.19
 13 |                                     ->  CStore Scan on store            |      24048 | 1MB      |      19 | 2264.00
 14 |                                     ->  Vector Partition Iterator       | 2879987999 | 1MB      |      66 | 2756066.50
 15 |                                        ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB | 66 | 2756066.50
 16 |                                  ->  CStore Scan on promotion           |      36000 | 1MB      |       4 | 1268.50
 17 |                               ->  CStore Scan on item                   |        158 | 1MB      |      58 | 4051.25
 18 |                         ->  CStore Scan on customer                     |   12000000 | 1MB      |       8 | 12923.00
 19 |                      ->  CStore Scan on customer_address ad2            |    6000000 | 1MB      |      58 | 5770.00
 20 |                ->  Vector Partition Iterator                            |  287999764 | 1MB      |      12 | 227383.99
 21 |                   ->  Partitioned CStore Scan on store_returns          |  287999764 | 1MB      |      12 | 227383.99
(21 rows)
```

图中计划顶端warning的提示详见**Hint的错误、冲突及告警**的说明。

4.6.3 Join 方式的 Hint

功能描述

指明Join使用的方法，可以为Nested Loop，Hash Join和Merge Join。

语法格式

```
[no] nestloop|hashjoin|mergejoin(table_list)
```

参数说明

- **no**表示hint的join方式不使用。
- **table_list**为表示hint表集合的字符串，该字符串中的表与**join_table_list**相同，只是中间不允许出现括号指定join的优先级。

例如：

no nestloop(t1 t2 t3)表示：生成t1，t2，t3三表连接计划时，不使用nestloop。三表连接计划可能是t2 t3先join，再跟t1 join，或t1 t2先join，再跟t3 join。此hint只hint最后一次join的join方式，对于两表连接的方法不hint。如果需要，可以单独指定，例如：任意表均不允许nestloop连接，且希望t2 t3先join，则增加hint：no nestloop(t2 t3)。

示例

对**示例**中原语句使用如下hint:

```
explain
select /*+ nestloop(store_sales store_returns item) */ i_product_name product_name ...
```

该hint表示：生成store_sales，store_returns和item三表的结果集时，最后的两表关联使用nestloop。生成计划如下所示：

```
 id |                                  operation                                  |   E-rows   | E-memory | E-width |     E-costs
----+-----------------------------------------------------------------------------+------------+----------+---------+-----------------
  1 | ->  Row Adapter                                                             |          6 |          |     273 | 100061693161.06
  2 |    ->  Vector Streaming (type: GATHER)                                      |          6 |          |     273 | 100061693161.06
  3 |       ->  Vector Hash Aggregate                                             |          6 | 16MB     |     273 | 100061693159.40
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)                           |          6 | 1MB      |     169 | 100061693159.36
  5 |             ->  Vector Hash Join (6,22)                                     |          6 | 43MB     |     169 | 100061693158.99
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                     |          6 | 1MB      |     119 | 100061688591.48
  7 |                   ->  Vector Hash Join (8,21)                               |          6 | 16MB     |     119 | 100061688591.30
  8 |                      ->  Vector Hash Join (9,20)                            |          7 | 27MB     |     123 | 100061687304.04
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)            |          7 | 1MB      |     123 | 100061677823.27
 10 |                            ->  Vector Hash Join (11,19)                     |          7 | 16MB     |     123 | 100061677823.12
 11 |                               ->  Vector Nest Loop (12,17)                  |          7 | 1MB      |     112 | 100061675546.57
 12 |                                  ->  Vector Hash Join (13,15)               |      13670 | 585MB    |      62 | 6163443.54
 13 |                                     ->  Vector Partition Iterator           | 2879987999 | 1MB      |      66 | 2756066.50
 14 |                                        ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB |      66 | 2756066.50
 15 |                                     ->  Vector Partition Iterator           |  287999764 | 1MB      |      12 | 227383.99
 16 |                                        ->  Partitioned CStore Scan on store_returns |  287999764 | 1MB |      12 | 227383.99
 17 |                                  ->  Vector Materialize                     |        158 | 16MB     |      58 | 4051.28
 18 |                                     ->  CStore Scan on item                 |        158 | 1MB      |      58 | 4051.25
 19 |                               ->  CStore Scan on store                      |      24048 | 1MB      |      19 | 2264.00
 20 |                            ->  CStore Scan on customer                      |   12000000 | 1MB      |       8 | 12923.00
 21 |                         ->  CStore Scan on promotion                        |      36000 | 1MB      |       4 | 1268.50
 22 |                      ->  CStore Scan on customer_address ad2                |    6000000 | 1MB      |      58 | 5770.00
(22 rows)
```

4.6.4 行数的 Hint

功能描述

指明中间结果集的大小，支持绝对值和相对值的hint。

语法格式

```
rows(table_list #|+|-|* const)
```

参数说明

- **#,+,-,***，进行行数估算hint的四种操作符号。#表示直接使用后面的行数进行hint。+,-,*表示对原来估算的行数进行加、减、乘操作，运算后的行数最小值为1行。table_list为hint对应的单表或多表join结果集，与**Join方式的Hint**中**table_list**相同。
- **const**可以是任意非负数，支持科学计数法。

例如：

rows(t1 #5)表示：指定t1表的结果集为5行。

rows(t1 t2 t3 *1000)表示：指定t1, t2, t3 join完的结果集的行数乘以1000。

建议

- 推荐使用两个表*的hint。对于两个表的采用*操作符的hint，只要两个表出现在join的两端，都会触发hint。例如：设置hint为rows(t1 t2 * 3)，对于(t1 t3 t4)和(t2 t5 t6)join时，由于t1和t2出现在join的两端，所以其join的结果集也会应用该hint规则乘以3。
- rows hint支持在单表、多表、function table及subquery scan table的结果集上指定hint。

示例

对**示例**中原语句使用如下hint:

```
explain
select /*+ rows(store_sales store_returns *50) */ i_product_name product_name ...
```

该hint表示：store_sales，store_returns关联的结果集估算行数在原估算行数基础上乘以50。生成计划如下所示：

```
 id |                                operation                                 |   E-rows   | E-memory | E-width |  E-costs
----+--------------------------------------------------------------------------+------------+----------+---------+-----------
  1 | ->  Row Adapter                                                          |        312 |          |     273 | 3401656.58
  2 |    ->  Vector Streaming (type: GATHER)                                   |        312 |          |     273 | 3401656.58
  3 |       ->  Vector Hash Aggregate                                          |        312 | 16MB     |     273 | 3401634.91
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)                        |        313 | 1MB      |     169 | 3401634.39
  5 |             ->  Vector Hash Join (6,21)                                  |        313 | 43MB     |     169 | 3401633.06
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                  |        313 | 1MB      |     119 | 3397065.38
  7 |                   ->  Vector Hash Join (8,20)                            |        313 | 27MB     |     119 | 3397064.31
  8 |                      ->  Vector Streaming(type: REDISTRIBUTE)            |        328 | 1MB      |     119 | 3387583.37
  9 |                         ->  Vector Hash Join (10,19)                     |        328 | 16MB     |     119 | 3387582.18
 10 |                            ->  Vector Hash Join (11,18)                  |        344 | 16MB     |     123 | 3386294.74
 11 |                               ->  Vector Hash Join (12,14)               |        360 | 19MB     |     112 | 3384018.02
 12 |                                  ->  Vector Partition Iterator           |  287999764 | 1MB      |      12 | 227383.99
 13 |                                     ->  Partitioned CStore Scan on store_returns |  287999764 | 1MB      |      12 | 227383.99
 14 |                                  ->  Vector Hash Join (15,17)            |    1516824 | 16MB     |     124 | 3065686.08
 15 |                                     ->  Vector Partition Iterator        | 2879987999 | 1MB      |      66 | 2756066.50
 16 |                                        ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB      |      66 | 2756066.50
 17 |                                     ->  CStore Scan on item              |        158 | 1MB      |      58 | 4051.25
 18 |                               ->  CStore Scan on store                   |      24048 | 1MB      |      19 | 2264.00
 19 |                            ->  CStore Scan on promotion                  |      36000 | 1MB      |       4 | 1268.50
 20 |                      ->  CStore Scan on customer                         |   12000000 | 1MB      |       8 | 12923.00
 21 |                ->  CStore Scan on customer_address ad2                   |    6000000 | 1MB      |      58 | 5770.00
(21 rows)
```

第11行算子的估算行数修正为360行，原估算行数为7行（四舍五入后取值）。

4.6.5 Stream 方式的 Hint

功能描述

指明stream使用的方法，可以为broadcast和redistribute以及指定AGG重分布的分布键。

说明

指定Agg重分布列Hint，仅8.1.3.100及以上集群版本支持。

语法格式

```
[no] broadcast | redistribute(table_list) | redistribute ((*) (columns))
```

参数说明

- **no**表示hint的stream方式不使用，当指定AGG分布列的hint时指定no关键字无效。

- **table_list**为进行stream操作的单表或多表join结果集，见**参数说明**。
- 当指定分布列的hint时，*为固定写法，不支持指定表名。
- **columns** 指定group by子句中的一个或者多个列，没有group by子句也可以指定distinct子句中的列。

说明

- 指定的分布列，需要用group by或 distinct中的列序号或列名来表示，count(distinct)中的列只能通过列名指定。
- 对于多层的查询，可以在每层指定对应层的分布列hint，只在当前层生效。
- 指定的count（ distinct ）列仅针对生成双层hashagg的计划时才生效，否则指定的分布列无效。
- 指定了分布列，如果优化器估算后发现不需要重分布，则指定的分布列无效。

建议

- 通常优化器会根据统计信息选择一组不倾斜的分布键进行数据重分布。当默认选择的分布键有倾斜时，可以手动指定重分布的列，避免数据倾斜。
- 在选择分布键的时候，通常要根据数据分布特征选取一组distinct值比较高的列做为分布列，这样可以保证重分布后，数据均匀的分布到各个DN。
- 在编写好hint后，可以通过explain verbose+SQL打印执行计划，查看指定的分布键是否有效，如果指定的分布键无效会有warning提示。

示例

- 对**示例**中原语句使用如下hint：

```
explain
select /*+ no redistribute(store_sales store_returns item store) leading(((store_sales store_returns item store) customer)) */ i_product_name product_name ...
```

原计划中，(store_sales store_returns item store)和customer做join时，前者做了重分布，此hint表示禁止前者混合表做重分布，但仍然保持join顺序，则生成计划如下所示：

```
 id |                                operation                                 |   E-rows   | E-memory | E-width |  E-costs
----+--------------------------------------------------------------------------+------------+----------+---------+------------
  1 | ->  Row Adapter                                                          |          6 |          |     273 | 5718448.94
  2 |    ->  Vector Streaming (type: GATHER)                                   |          6 |          |     273 | 5718448.94
  3 |       ->  Vector Hash Aggregate                                          |          6 | 16MB     |     273 | 5718447.27
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)                        |          6 | 1MB      |     169 | 5718447.23
  5 |             ->  Vector Hash Join (6,21)                                  |          6 | 16MB     |     169 | 5718446.86
  6 |                ->  Vector Hash Join (7,20)                               |          7 | 43MB     |     173 | 5717159.60
  7 |                   ->  Vector Streaming(type: REDISTRIBUTE)               |          7 | 1MB      |     123 | 5712592.09
  8 |                      ->  Vector Hash Join (9,18)                         |          7 | 585MB    |     123 | 5712591.93
  9 |                         ->  Vector Hash Join (10,17)                     |          7 | 16MB     |     123 | 3386294.56
 10 |                            ->  Vector Hash Join (11,13)                  |          7 | 19MB     |     112 | 3384018.02
 11 |                               ->  Vector Partition Iterator              |  287999764 | 1MB      |      12 | 227383.99
 12 |                                  ->  Partitioned CStore Scan on store_returns |  287999764 | 1MB      |      12 | 227383.99
 13 |                               ->  Vector Hash Join (14,16)               |    1516824 | 16MB     |     124 | 3065686.08
 14 |                                  ->  Vector Partition Iterator           | 2879987999 | 1MB      |      66 | 2756066.50
 15 |                                     ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB      |      66 | 2756066.50
 16 |                                  ->  CStore Scan on item                 |        158 | 1MB      |      58 | 4051.25
 17 |                            ->  CStore Scan on store                      |      24048 | 1MB      |      19 | 2264.00
 18 |                         ->  Vector Streaming(type: BROADCAST)            |  288000000 | 1MB      |       8 | 2176297.36
 19 |                            ->  CStore Scan on customer                   |   12000000 | 1MB      |       8 | 12923.00
 20 |                      ->  CStore Scan on customer_address ad2             |    6000000 | 1MB      |      58 | 5770.00
 21 |                   ->  CStore Scan on promotion                           |      36000 | 1MB      |       4 | 1268.50
(21 rows)
```

- 指定Agg重分布的分布列。

```
explain (verbose on, costs off, nodes off)
select /*+ redistribute ((*) (2 3)) */ a1, b1, c1, count(c1)  from t1 group by a1, b1, c1 having count(c1) > 10 and sum(d1) > 100
```

通过下面的示例可以看到指定的group by列的后两列做为分布键。

```
                              QUERY PLAN
-------------------------------------------------------------------
  id |                 operation
 ----+-------------------------------------------
   1 | ->  Streaming (type: GATHER)
   2 |     ->  HashAggregate
   3 |        ->  Streaming(type: REDISTRIBUTE)
   4 |           ->  Seq Scan on public.t1

         Predicate Information (identified by plan id)
 ------------------------------------------------------------
   2 --HashAggregate
         Filter: ((count(t1.c1) > 10) AND (sum(t1.d1) > 100))

 Targetlist Information (identified by plan id)
 ----------------------------------------------
   1 --Streaming (type: GATHER)
         Output: a1, b1, c1, (count(c1))
   2 --HashAggregate
         Output: a1, b1, c1, count(c1)
         Group By Key: t1.a1, t1.b1, t1.c1
   3 --Streaming(type: REDISTRIBUTE)
         Output: a1, b1, c1, d1
         Distribute Key: b1, c1
   4 --Seq Scan on public.t1
         Output: a1, b1, c1, d1

    ====== Query Summary =====
 ---------------------------------
 System available mem: 24862720KB
 Query Max mem: 24862720KB
 Query estimated mem: 3138KB
(30 rows)
```

- 当语句中不包含group by 子句时，指定distinct列作为重分布列。

```
explain (verbose on, costs off, nodes off)
select /*+ redistribute ((*) (3 1)) */ distinct a1, b1, c1 from t1;
```

```
                    QUERY PLAN
-------------------------------------------------
  id |                 operation
 ----+-----------------------------------------
   1 | ->  Streaming (type: GATHER)
   2 |     ->  HashAggregate
   3 |        ->  Streaming(type: REDISTRIBUTE)
   4 |           ->  Seq Scan on public.t1

 Targetlist Information (identified by plan id)
 -----------------------------------------------
   1 --Streaming (type: GATHER)
         Output: a1, b1, c1
   2 --HashAggregate
         Output: a1, b1, c1
         Group By Key: t1.a1, t1.b1, t1.c1
   3 --Streaming(type: REDISTRIBUTE)
         Output: a1, b1, c1
         Distribute Key: c1, a1
   4 --Seq Scan on public.t1
         Output: a1, b1, c1

    ====== Query Summary =====
 ---------------------------------
 System available mem: 24862720KB
 Query Max mem: 24862720KB
 Query estimated mem: 3136KB
(25 rows)
```

4.6.6 Scan 方式的 Hint

功能描述

指明scan使用的方法，可以是tablescan、indexscan和indexonlyscan。

语法格式

```
[no] tablescan|indexscan|indexonlyscan(table [index])
```

参数说明

- **no**表示hint的scan方式不使用。
- **table**表示hint指定的表，只能指定一个表，如果表存在别名应优先使用别名进行hint。
- **index**表示使用indexscan或indexonlyscan的hint时，指定的索引名称，当前只能指定一个。

说明

对于indexscan或indexonlyscan，只有hint的索引属于hint的表时，才能使用该hint。

scan hint支持在行列存表、hdfs内外表、obs表、子查询表上指定。对于hdfs内表，由主表和delta表组成，delta表对用户不可见，故hint仅作用在主表上。

指定indexscan可生效indexscan或indexonlyscan，indexscan或indexonlyscan也可同时出现。当indexscan和indexonlyscan hint同时出现，优先生效indexonlyscan。

示例

为了hint使用索引扫描，需要首先在表item的i_item_sk列上创建索引，名称为i：

```
create index i on item(i_item_sk);
```

对**示例**中原语句使用如下hint：

```
explain
select /*+ indexscan(item i) */ i_product_name product_name ...
```

该hint表示：item表使用索引i进行扫描。生成计划如下所示：

```
 id |                                  operation                                  |   E-rows   | E-memory | E-width |     E-costs
----+-----------------------------------------------------------------------------+------------+----------+---------+-----------------
  1 | ->  Row Adapter                                                             |          6 |          |     273 | 100061674938.26
  2 |    ->  Vector Streaming (type: GATHER)                                      |          6 |          |     273 | 100061674938.26
  3 |       ->  Vector Hash Aggregate                                             |          6 | 16MB     |     273 | 100061674936.59
  4 |          ->  Vector Streaming(type: REDISTRIBUTE)                           |          6 | 1MB      |     169 | 100061674936.55
  5 |             ->  Vector Hash Join (6,21)                                     |          6 | 43MB     |     169 | 100061674936.19
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                     |          6 | 1MB      |     119 | 100061670368.67
  7 |                   ->  Vector Hash Join (8,20)                               |          6 | 16MB     |     119 | 100061670368.50
  8 |                      ->  Vector Hash Join (9,19)                            |          7 | 27MB     |     123 | 100061669081.23
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)            |          7 | 1MB      |     123 | 100061659600.47
 10 |                            ->  Vector Hash Join (11,18)                     |          7 | 16MB     |     123 | 100061659600.31
 11 |                               ->  Vector Nest Loop (12,17)                  |          7 | 1MB      |     112 | 100061657323.77
 12 |                                  ->  Vector Hash Join (13,15)               |      13670 | 585MB    |      62 | 6163443.54
 13 |                                     ->  Vector Partition Iterator           | 2879987999 | 1MB      |      66 | 2756066.50
 14 |                                        ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB |      66 | 2756066.50
 15 |                                     ->  Vector Partition Iterator           |  287999764 | 1MB      |      12 | 227383.99
 16 |                                        ->  Partitioned CStore Scan on store_returns |  287999764 | 1MB |      12 | 227383.99
 17 |                                  ->  CStore Index Scan using i on item      |          1 | 1MB      |      58 | 4.01
 18 |                               ->  CStore Scan on store                      |      24048 | 1MB      |      19 | 2264.00
 19 |                            ->  CStore Scan on customer                      |   12000000 | 1MB      |       8 | 12923.00
 20 |                         ->  CStore Scan on promotion                        |      36000 | 1MB      |       4 | 1268.50
 21 |                   ->  CStore Scan on customer_address ad2                   |    6000000 | 1MB      |      58 | 5770.00
(21 rows)
```

4.6.7 子链接块名的 hint

功能描述

指明子链接块的名称。

语法格式

```
blockname (table)
```

参数说明

- **table**表示为该子链接块hint的别名的名称。

说明

- **blockname hint**仅在对应的子链接块提升时才会被上层查询使用。目前支持的子链接提升包括IN子链接提升、EXISTS子链接提升和包含Agg等值相关子链接提升。该hint通常会和前面章节提到的hint联合使用。
- 对于FROM关键字后的子查询，则需要使用子查询的别名进行hint，blockname hint不会被用到。
- 如果子链接中含有多个表，则提升后这些表可与外层表以任意优化顺序连接，hint也不会被用到。

示例

```
explain select /*+nestloop(store_sales tt) */ * from store_sales where ss_item_sk in (select /*
+blockname(tt)*/ i_item_sk from item group by 1);
```

该hint表示：子链接的别名为tt，提升后与上层的store_sales表关联时使用nestloop。生成计划如下所示：

```
 id |                       operation                        |   E-rows   | E-memory | E-width |     E-costs
----+--------------------------------------------------------+------------+----------+---------+-----------------
  1 | ->  Row Adapter                                        | 1439994000 |          |     216 | 325105765847.91
  2 |    ->  Vector Streaming (type: GATHER)                 | 1439994000 |          |     216 | 325105765847.91
  3 |       ->  Vector Nest Loop Semi Join (4, 6)            | 1439994000 | 1MB      |     216 | 325026664615.00
  4 |          ->  Vector Partition Iterator                 | 2879987999 | 1MB      |     216 | 2756066.50
  5 |             ->  Partitioned CStore Scan on store_sales | 2879987999 | 1MB      |     216 | 2756066.50
  6 |          ->  Vector Materialize                        |     300000 | 16MB     |       4 | 4176.25
  7 |             ->  Vector Hash Aggregate                  |     300000 | 16MB     |       4 | 3988.75
  8 |                ->  CStore Scan on item                 |     300000 | 1MB      |       4 | 3832.50
(8 rows)
```

4.6.8 运行倾斜的 hint

功能描述

指明查询运行时重分布过程中存在倾斜的重分布键和倾斜值，针对Join和HashAgg运算中的重分布进行优化。

语法格式

- 指定单表倾斜：
  ```
  skew(table (column) [(value)])
  ```
- 指定中间结果倾斜：
  ```
  skew((join_rel) (column) [(value)])
  ```

参数说明

- **table**表示存在倾斜的单个表名。
- **join_rel**表示参与join的两个或多个表，如（t1 t2）表示t1和t2join后的结果存在倾斜。
- **column**表示倾斜表中存在倾斜的一个或多个列。
- **value**表示倾斜的列中存在倾斜的一个或多个值。

说明

- skew hint仅在需要重分布且指定的倾斜信息与查询执行过程中的重分布信息相匹配时才会被使用。
- skew hint受GUC参数**skew_option**限制，如果参数处于关闭状态，则无法进行skew hint倾斜调优。
- skew hint目前仅处理普通表和子查询类型的表关系，支持基表hint、子查询hint、with as子句hint。对于子查询，无论提升与否都支持在skew hint中使用，这点与其它hint不一样。
- 对于倾斜表，如果定义了别名，则在hint中必须使用别名。
- 对于倾斜列，在不产生歧义的情况下，可以使用原名也可以使用别名。skew hint的column不支持表达式，如果需要指定采用分布键为表达式的重分布存在倾斜，需要将重分布键指定为新的列，以新的列进行hint。
- 对于倾斜值，个数需为列数的整数倍并按列的顺序进行组合，组合的个数不能超过10个。如果各倾斜列的倾斜值的个数不一样，为了满足按列组合，值可以重复指定。如，表t1的c1和c2存在倾斜，c1列的倾斜值只有a1，而c2列的倾斜有b1和b2，则skew hint如下：skew(t1 (c1 c2) ((a1 b1)(a1 b2)))。例中(a1 b1)为一个值组合，NULL可以作为倾斜值出现，每个hint中的值组合不超过十个， 且需为列的整数倍。
- 在Join的重分布优化中，skew hint中的value不可缺省，在HashAgg中可以缺省。
- 对于表、列、值中若指定多个，则同类间需以空格分离。
- 对于倾斜值，不支持在hint中进行类型强转；对于string类型，需要使用单引号。

例如：

- 指定单表倾斜

  每一个skew hint用来表示一个表关系存在的倾斜信息，如果想要指定在查询中的多个表关系存在的倾斜信息，则通过指定多个skew hint实现。

  在指定skew时，包括以下四个场景的用法：

  – 单列单值： skew(t (c1) (v1))

    说明：表关系t的c1列中的v1值在查询执行中存在倾斜。

  – 单列多值：skew(t (c1) (v1 v2 v3 ...))

    说明：表关系t的c1列中的v1、v2、v3…等值在查询执行中存在倾斜。

  – 多列单值：skew(t (c1 c2) (v1 v2))

    说明：表关系t的c1列的v1值和c2列的v2值在查询执行中存在倾斜。

  – 多列多值：skew(t (c1 c2) ((v1 v2) (v3 v4) (v5 v6) ...))

    说明：表关系t的c1列的v1、v3、v5…值和c2列的v2、v4、v6…值在查询执行中存在倾斜。

  须知

  多列多值时，各组倾斜值间也可以不使用括号，如：skew(t (c1 c2) (v1 v2 v3 v4 v5 v6 ...))。是否使用括号必须统一，不可混合，

  如：skew(t (c1 c2) (v1 v2 v3 v4 （ v5 v6 ） ...)) 将会产生语法报错。

- 指定中间结果倾斜

  如果基表不存在倾斜，而是查询执行中的中间结果出现倾斜，则需要通过指定中间结果倾斜的skew hint来进行倾斜的调优。skew((t1 t2) (c1) (v1))

  说明：表关系t1和t2 Join后的结果存在倾斜，倾斜的是t1表的c1列，c1列的倾斜值是v1。

  为了避免产生歧义，“c1”只能存在于join_rel的一个表关系中，如果存在同名列则通过别名进行规避。

建议

- 如果查询具有多层，则哪一层出现倾斜，则将hint写在哪一层中。
- 对于提升的子查询，skew hint支持直接使用子查询名进行hint。如果明确子查询提升后的哪一个基表存在倾斜，则直接使用基表进行hint的可用性更高。
- 无论对于表或列，若存在别名，则优先使用别名进行hint。

示例

指定单表倾斜

- 原query中进行hint。

  采用如下查询进行skew hint倾斜调优的举例，查询语句及不带hint的原计划如下所示：

```
explain
with customer_total_return as
(select sr_customer_sk as ctr_customer_sk
,sr_store_sk as ctr_store_sk
,sum(SR_FEE) as ctr_total_return
from store_returns
```



```
,date_dim
where sr_returned_date_sk = d_date_sk
and d_year =2000
group by sr_customer_sk
,sr_store_sk)
 select  c_customer_id
from customer_total_return ctr1
,store
,customer
where ctr1.ctr_total_return > (select avg(ctr_total_return)*1.2
from customer_total_return ctr2
where ctr1.ctr_store_sk = ctr2.ctr_store_sk)
and s_store_sk = ctr1.ctr_store_sk
and s_state = 'NM'
and ctr1.ctr_customer_sk = c_customer_sk
order by c_customer_id
limit 100;
```

```
 id |                                  operation                                  |  E-rows   |    E-memory    | E-width |  E-costs
----+-----------------------------------------------------------------------------+-----------+----------------+---------+-----------
  1 | ->  Row Adapter                                                             |       100 |                |      20 | 911254.47
  2 |    ->  Vector Limit                                                         |       100 |                |      20 | 911254.47
  3 |       ->  Vector Streaming (type: GATHER)                                   |      2400 |                |      20 | 911325.75
  4 |          ->  Vector Limit                                                   |      2400 | 1MB            |      20 | 911247.62
  5 |             ->  Vector Sort                                                 |   3684816 | 16MB           |      20 | 911631.21
  6 |                ->  Vector Hash Join (7,29)                                  |   3684817 | 41MB(12374MB)  |      20 | 905379.41
  7 |                   ->  Vector Streaming(type: REDISTRIBUTE)                  |   3684817 | 384KB          |       4 | 883010.31
  8 |                      ->  Vector Hash Join (9,19)                            |   3684817 | 16MB           |       4 | 861302.05
  9 |                         ->  Vector Hash Join (10,18)                        |  11054450 | 16MB           |      44 | 427109.71
 10 |                            ->  Vector Hash Aggregate                        |  50247501 | 397MB(12671MB) |      54 | 395302.57
 11 |                               ->  Vector Streaming(type: REDISTRIBUTE)      |  50247501 | 384KB          |      22 | 358663.76
 12 |                                  ->  Vector Hash Join (13,15)               |  50247501 | 16MB           |      22 | 294300.51
 13 |                                     ->  Vector Partition Iterator           | 287999764 | 1MB            |      26 | 227383.99
 14 |                                        ->  Partitioned CStore Scan on store_returns | 287999764 | 1MB |      26 | 227383.99
 15 |                                     ->  Vector Streaming(type: BROADCAST)   |      8712 | 384KB          |       4 | 975.56
 16 |                                        ->  Vector Partition Iterator        |       363 | 1MB            |       4 | 910.65
 17 |                                           ->  Partitioned CStore Scan on date_dim |   363 | 1MB            |       4 | 910.65
 18 |                            ->  CStore Scan on store                         |        44 | 1MB            |       4 | 1006.39
 19 |                         ->  Vector Hash Aggregate                           |       192 | 16MB           |      68 | 426707.38
 20 |                            ->  Vector Subquery Scan on ctr2                 |  50247501 | 1MB            |      36 | 416239.03
 21 |                               ->  Vector Hash Aggregate                     |  50247501 | 397MB(12671MB) |      54 | 395302.57
 22 |                                  ->  Vector Streaming(type: REDISTRIBUTE)   |  50247501 | 384KB          |      22 | 358663.76
 23 |                                     ->  Vector Hash Join (24,26)            |  50247501 | 16MB           |      22 | 294300.51
 24 |                                        ->  Vector Partition Iterator        | 287999764 | 1MB            |      26 | 227383.99
 25 |                                           ->  Partitioned CStore Scan on store_returns | 287999764 | 1MB |      26 | 227383.99
 26 |                                        ->  Vector Streaming(type: BROADCAST)|      8712 | 384KB          |       4 | 975.56
 27 |                                           ->  Vector Partition Iterator     |       363 | 1MB            |       4 | 910.65
 28 |                                              ->  Partitioned CStore Scan on date_dim |  363 | 1MB            |       4 | 910.65
 29 |                   ->  CStore Scan on customer                               |  12000000 | 1MB            |      24 | 12923.00
(29 rows)
```

对内层with子句中的HashAgg和外层的Hash Join进行hint指定，带hint的查询如下：

```
explain
with customer_total_return as
(select /*+ skew(store_returns(sr_store_sk sr_customer_sk)) */sr_customer_sk as ctr_customer_sk
,sr_store_sk as ctr_store_sk
,sum(SR_FEE) as ctr_total_return
from store_returns
,date_dim
where sr_returned_date_sk = d_date_sk
and d_year =2000
group by sr_customer_sk
,sr_store_sk)
 select  /*+ skew(ctr1(ctr_customer_sk)(11))*/  c_customer_id
from customer_total_return ctr1
,store
,customer
where ctr1.ctr_total_return > (select avg(ctr_total_return)*1.2
from customer_total_return ctr2
where ctr1.ctr_store_sk = ctr2.ctr_store_sk)
and s_store_sk = ctr1.ctr_store_sk
and s_state = 'NM'
and ctr1.ctr_customer_sk = c_customer_sk
order by c_customer_id
limit 100;
```

该hint表示：内层with子句中的group by在做HashAgg中进行重分布时存在倾斜，对应原计划的10和21号Hash Agg算子；外层ctr1表的ctr_customer_sk列在做Hash Join中进行重分布时存在倾斜，对应原计划的6号算子。生成计划如下所示：


```
 id |                                operation                                 |  E-rows   |  E-memory   | E-width |  E-costs
----+--------------------------------------------------------------------------+-----------+-------------+---------+-----------
  1 | ->  Row Adapter                                                          |       100 |             |      20 | 1061778.14
  2 |    ->  Vector Limit                                                      |       100 |             |      20 | 1061778.14
  3 |       ->  Vector Streaming (type: GATHER)                                |      2400 |             |      20 | 1061849.41
  4 |          ->  Vector Limit                                                |      2400 | 1MB         |      20 | 1061771.29
  5 |             ->  Vector Sort                                              |   3684816 | 16MB        |      20 | 1062154.87
  6 |                ->  Vector Hash Join (7,31)                               |   3684817 | 41MB(12344MB) |      20 | 1055903.08
  7 |                   ->  Vector Streaming(type: PART REDISTRIBUTE PART ROUNDROBIN) |   3684817 | 384KB       |       4 | 1013056.49
  8 |                      ->  Vector Hash Join (9,20)                         |   3684817 | 16MB        |       4 | 1000006.10
  9 |                         ->  Vector Hash Join (10,19)                     |  11054450 | 16MB        |      44 | 496461.73
 10 |                            ->  Vector Hash Aggregate                     |  50247501 | 397MB(12010MB) |      54 | 464654.59
 11 |                               ->  Vector Streaming(type: REDISTRIBUTE)   |  50247501 | 384KB       |      54 | 428015.79
 12 |                                  ->  Vector Hash Aggregate               |  50247501 | 397MB(12010MB) |      54 | 330939.31
 13 |                                     ->  Vector Hash Join (14,16)         |  50247501 | 16MB        |      22 | 294300.51
 14 |                                        ->  Vector Partition Iterator     | 287999764 | 1MB         |      26 | 227383.99
 15 |                                           ->  Partitioned CStore Scan on store_returns | 287999764 | 1MB |      26 | 227383.99
 16 |                                        ->  Vector Streaming(type: BROADCAST) |      8712 | 384KB       |       4 | 975.56
 17 |                                           ->  Vector Partition Iterator  |       363 | 1MB         |       4 | 910.65
 18 |                                              ->  Partitioned CStore Scan on date_dim |       363 | 1MB |       4 | 910.65
 19 |                               ->  CStore Scan on store                   |        44 | 1MB         |       4 | 1006.39
 20 |                            ->  Vector Hash Aggregate                     |       192 | 16MB        |      68 | 496059.40
 21 |                               ->  Vector Subquery Scan on ctr2           |  50247501 | 1MB         |      36 | 485591.05
 22 |                                  ->  Vector Hash Aggregate               |  50247501 | 397MB(12010MB) |      54 | 464654.59
 23 |                                     ->  Vector Streaming(type: REDISTRIBUTE) |  50247501 | 384KB       |      54 | 428015.79
 24 |                                        ->  Vector Hash Aggregate         |  50247501 | 397MB(12010MB) |      54 | 330939.31
 25 |                                           ->  Vector Hash Join (26,28)   |  50247501 | 16MB        |      22 | 294300.51
 26 |                                              ->  Vector Partition Iterator | 287999764 | 1MB |      26 | 227383.99
 27 |                                                 ->  Partitioned CStore Scan on store_returns | 287999764 | 1MB |      26 | 227383.99
 28 |                                              ->  Vector Streaming(type: BROADCAST) |      8712 | 384KB       |       4 | 975.56
 29 |                                                 ->  Vector Partition Iterator |       363 | 1MB         |       4 | 910.65
 30 |                                                    ->  Partitioned CStore Scan on date_dim |       363 | 1MB |       4 | 910.65
 31 |                      ->  Vector Streaming(type: PART LOCAL PART BROADCAST) |  12000000 | 384KB       |      24 | 34485.50
 32 |                         ->  CStore Scan on customer                      |  12000000 | 1MB         |      24 | 12923.00
(32 rows)
```

从优化后的计划可以看出：①对于Hash Agg，由于其重分布存在倾斜，所以优化为双层Agg；②对于Hash Join，同样由于其重分布存在倾斜，所以优化为采用新的重分布算子。

- 需要改写query后进行hint

  不带hint的查询和计划如下：

```
explain select count(*) from store_sales_1 group by round(ss_list_price);
```

```
 id |                              operation                               |  E-rows  |  E-memory  | E-width |  E-costs
----+----------------------------------------------------------------------+----------+------------+---------+----------
  1 | ->  Row Adapter                                                      |    16672 |            |      14 | 62261.28
  2 |    ->  Vector Streaming (type: GATHER)                               |    16672 |            |      14 | 62261.28
  3 |       ->  Vector Streaming(type: LOCAL GATHER dop: 1/2)              |    16672 | 32KB       |      14 | 61479.78
  4 |          ->  Vector Hash Aggregate                                   |    16672 | 16MB       |      14 | 61452.00
  5 |             ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 2/2)  | 3112836 | 128KB      |       6 | 57498.43
  6 |                ->  CStore Scan on store_sales_1                      | 3112836 | 1MB        |       6 | 21810.25
(6 rows)
```

由于hint中列不支持表达式，在进行倾斜优化时需要借助subquery改写查询，改写后的查询和计划如下：

```
explain
select count(*)
from (select round(ss_list_price),ss_hdemo_sk
from store_sales_1)tmp(a,ss_hdemo_sk)
group by a;
```

```
 id |                              operation                               |  E-rows  |  E-memory  | E-width |  E-costs
----+----------------------------------------------------------------------+----------+------------+---------+----------
  1 | ->  Row Adapter                                                      |    16672 |            |      14 | 62261.28
  2 |    ->  Vector Streaming (type: GATHER)                               |    16672 |            |      14 | 62261.28
  3 |       ->  Vector Streaming(type: LOCAL GATHER dop: 1/2)              |    16672 | 32KB       |      14 | 61479.78
  4 |          ->  Vector Hash Aggregate                                   |    16672 | 16MB       |      14 | 61452.00
  5 |             ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 2/2)  | 3112836 | 128KB      |       6 | 57498.43
  6 |                ->  CStore Scan on store_sales_1                      | 3112836 | 1MB        |       6 | 21810.25
(6 rows)
```

改写注意不要影响到业务逻辑。

采用改写后的查询进行hint，带hint的查询和计划如下：

```
explain
select /*+ skew(tmp(a)) */ count(*)
from (select round(ss_list_price),ss_hdemo_sk
from store_sales_1)tmp(a,ss_hdemo_sk)
group by a;
```

```
 id |                              operation                               |  E-rows  | E-memory | E-width |  E-costs
----+----------------------------------------------------------------------+----------+----------+---------+-----------
  1 | ->  Row Adapter                                                      |    16672 |          |      14 | 27771.82
  2 |    ->  Vector Streaming (type: GATHER)                               |    16672 |          |      14 | 27771.82
  3 |       ->  Vector Streaming(type: LOCAL GATHER dop: 1/2)              |    16672 | 32KB     |      14 | 26990.32
  4 |          ->  Vector Hash Aggregate                                   |    16671 | 16MB     |      14 | 26962.54
  5 |             ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 2/2)  |    66216 | 128KB    |      14 | 26838.09
  6 |                ->  Vector Hash Aggregate                             |    66216 | 16MB     |      14 | 25949.61
  7 |                   ->  CStore Scan on store_sales_1                   | 3112836 | 1MB      |       6 | 21810.25
(7 rows)
```

从计划可以看出，对Hash Agg进行倾斜优化后，采用了双层agg实现，大大过滤了进行重分布时的数据量，减少了重分布时间。

此外，需要说明的是，对于子查询，支持使用查询内部的列进行hint，如：

```
explain
select /*+ skew(tmp(b)) */ count(*)
```

```
from (select round(ss_list_price) b,ss_hdemo_sk
from store_sales_1)tmp(a,ss_hdemo_sk)
group by a;
```

4.6.9 指定子查询不提升的 hint

功能描述

优化器在对查询进行逻辑优化时通常会将可以提升的子查询提升到上层以避免嵌套执行，但对于某些场景，嵌套执行不会导致性能下降过多，而提升之后扩大了查询路径的搜索范围，可能导致性能变差。对于此类情况，可以使用no merge hint指定子查询不提升进行调试。大多数情况下不建议使用此hint。

语法格式

```
no merge [(subquery_name)]
```

参数说明

subquery_name为目标子查询名，亦可以是view或cte名，表示该子查询在逻辑优化时不会进行提升；当不指定subquery_name时，表示当前查询不提升。

示例

创建表t1、t2、t3：

```
create table t1(a1 int,b1 int,c1 int,d1 int);
create table t2(a2 int,b2 int,c2 int,d2 int);
create table t3(a3 int,b3 int,c3 int,d3 int);
```

原语句为：

```
explain select * from t3, (select a1,b2,c1,d2 from t1,t2 where t1.a1=t2.a2) s1 where t3.b3=s1.b2;
```

```
 id |           operation           | E-rows | E-width | E-costs
----+-------------------------------+--------+---------+---------
  1 | ->  Hash Join (2,6)           |  44450 |      32 | 754.31
  2 |     ->  Hash Join (3,4)       |   8885 |      28 | 182.11
  3 |        ->  Seq Scan on t3     |   1776 |      16 | 27.76
  4 |        ->  Hash               |   1776 |      12 | 27.76
  5 |           ->  Seq Scan on t2  |   1776 |      12 | 27.76
  6 |     ->  Hash                  |   1776 |       8 | 27.76
  7 |        ->  Seq Scan on t1     |   1776 |       8 | 27.76
```

上述查询中，可以使用以下两种方式禁止子查询s1进行提升：

- 方式一：
  ```
  explain select /*+ no merge(s1) */ * from t3, (select a1,b2,c1,d2 from t1,t2 where t1.a1=t2.a2) s1 where t3.b3=s1.b2;
  ```
- 方式二：
  ```
  explain select * from t3, (select /*+ no merge */ a1,b2,c1,d2 from t1,t2 where t1.a1=t2.a2) s1 where t3.b3=s1.b2;
  ```

提升后效果：


```
 id |           operation           | E-rows | E-width | E-costs
----+-------------------------------+--------+---------+---------
  1 | ->  Hash Join (2,6)           |   8880 |      32 | 443.03
  2 |    ->  Hash Join (3,4)        |   8885 |      16 | 182.11
  3 |       ->  Seq Scan on t1      |   1776 |       8 | 27.76
  4 |       ->  Hash                |   1776 |      12 | 27.76
  5 |          ->  Seq Scan on t2   |   1776 |      12 | 27.76
  6 |    ->  Hash                   |   1776 |      16 | 27.76
  7 |       ->  Seq Scan on t3      |   1776 |      16 | 27.76
```

4.6.10 配置参数的 hint

功能描述

指明计划生成时配置参数的值，又称作guc hint。

语法格式

```
set [global](guc_name guc_value)
```

参数说明

- **global**表示hint设置的配置参数在语句级别生效，不加global表示hint设置的配置参数在子查询级别生效，即仅在hint所在的子查询中生效，在该语句的其它子查询中不生效。
- **guc_name**表示hint指定的配置参数的名称。
- **guc_value**表示hint指定的配置参数的值。

说明

- 如果hint设置的配置参数在语句级别生效，则该hint必须写在顶层查询中，而不能写在子查询中。对于UNION、INTERSECT、EXCEPT和MINUS语句，可以将在语句级别的guc hint写在参与集合运算的任意一个SELECT子句上，该guc hint设置的配置参数会在参与集合运算的每个SELECT子句上生效。
- 子查询提升时，该子查询上的所有guc hint会被丢弃。
- 如果一个配置参数既被语句级别的guc hint设置，又被子查询级别的guc hint设置，则子查询级别的guc hint在对应的子查询中生效，语句级别的guc hint在语句的其它子查询中生效。

guc hint当前仅支持部分配置参数，并且有些配置参数不支持在子查询级别设置，只能在语句级别设置，以下为支持的参数列表：

表 4-1 guc hint 支持的配置参数

配置参数名	是否支持在子查询级别设置
agg_max_mem	是
agg_redistribute_enhancement	是
best_agg_plan	是
cost_model_version	否

配置参数名	是否支持在子查询级别设置
cost_param	否
enable_bitmapscan	是
enable_broadcast	是
enable_redistribute	是
enable_extrapolation_stats	是
enable_fast_query_shipping	否
enable_force_vector_engine	否
enable_hashagg	是
enable_hashjoin	是
enable_index_nestloop	是
enable_indexscan	是
enable_join_pseudoconst	是
enable_nestloop	是
enable_nodegroup_debug	否
enable_partition_dynamic_pruning	是
enable_sort	是
enable_stream_ctescan	否
enable_value_redistribute	是
enable_vector_engine	否
expected_computing_nodegroup	否
force_bitmapand	是
from_collapse_limit	是
join_collapse_limit	是
join_num_distinct	是
outer_join_max_rows_multipler	是
prefer_hashjoin_path	否
qrw_inlist2join_optmode	是
qual_num_distinct	是
query_dop	否
query_max_mem	否

配置参数名	是否支持在子查询级别设置
query_mem	否
rewrite_rule	否
setop_optmode	是
skew_option	是
index_selectivity_cost	是
index_cost_limit	是
enable_accelerate_select	否

示例

对**示例**中原语句使用如下hint：

```
explain
select /*+ set global(query_dop 0) */ i_product_name product_name
...
```

该hint表示：在整个语句的计划生成过程中，将配置参数query_dop设置为0，即打开SMP自适应功能。生成的计划如下图所示：

```
 id |                                      operation                                       | E-rows  | E-memory | E-width | E-costs
----+--------------------------------------------------------------------------------------+---------+----------+---------+----------
  1 | ->  Row Adapter                                                                      |       1 |          |     230 | 19595.89
  2 |    ->  Vector Sonic Hash Aggregate                                                   |       1 |          |     230 | 19595.89
  3 |       ->  Vector Streaming (type: GATHER)                                            |       3 |          |     230 | 19595.89
  4 |          ->  Vector Sonic Hash Aggregate                                             |       3 | 16MB     |     230 | 19595.66
  5 |             ->  Vector Nest Loop (6,28)                                              |       3 | 1MB      |     126 | 19595.62
  6 |                ->  Vector Nest Loop (7,27)                                           |       3 | 1MB      |     130 | 19291.57
  7 |                   ->  Vector Streaming(type: LOCAL GATHER dop: 1/2)                  |       3 | 4MB      |     118 | 19279.41
  8 |                      ->  Vector Nest Loop (9,24)                                    |       3 | 1MB      |     118 | 19279.38
  9 |                         ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 2/2)     |       3 | 4MB      |      82 | 18117.66
 10 |                            ->  Vector Nest Loop (11,21)                            |       3 | 1MB      |      82 | 18117.61
 11 |                               ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 2/2) |       3 | 4MB      |      82 | 16195.20
 12 |                                  ->  Vector Sonic Hash Join (13,15)               |       3 | 16MB     |      82 | 16195.15
 13 |                                     ->  Vector Partition Iterator                 |  287514 | 1MB      |      12 | 1110.42
 14 |                                        ->  Partitioned CStore Scan on store_returns |  287514 | 1MB      |      12 | 1110.42
 15 |                                     ->  Vector Streaming(type: LOCAL BROADCAST dop: 2/2) |    2764 | 4MB      |      94 | 14718.42
 16 |                                        ->  Vector Sonic Hash Join (17,19)        |    1382 | 16MB     |      94 | 14699.69
 17 |                                           ->  Vector Partition Iterator          | 2880404 | 1MB      |      39 | 11541.07
 18 |                                              ->  Partitioned CStore Scan on store_sales | 2880404 | 1MB      |      39 | 11541.07
 19 |                                           ->  Vector Streaming(type: LOCAL BROADCAST dop: 2/2) |      16 | 4MB      |      55 | 1947.12
 20 |                                              ->  CStore Scan on item              |       8 | 1MB      |      55 | 1947.00
 21 |                                  ->  Vector Materialize                           |  100000 | 16MB     |       8 | 1797.41
 22 |                                     ->  Vector Streaming(type: LOCAL REDISTRIBUTE dop: 2/2) |  100000 | 4MB      |       8 | 1714.07
 23 |                                        ->  CStore Scan on customer               |  100000 | 1MB      |       8 | 703.67
 24 |                            ->  Vector Materialize                                |   50000 | 16MB     |      44 | 1099.22
 25 |                               ->  Vector Streaming(type: LOCAL REDISTRIBUTE dop: 2/2) |   50000 | 4MB      |      44 | 1057.55
 26 |                                  ->  CStore Scan on customer_address ad2          |   50000 | 1MB      |      44 | 552.33
 27 |                      ->  CStore Scan on store                                     |      36 | 1MB      |      20 | 12.01
 28 |                   ->  CStore Scan on promotion                                    |     900 | 1MB      |       4 | 300.30
(28 rows)
```

4.6.11 Hint 的错误、冲突及告警

Plan Hint的结果会体现在计划的变化上，可以通过explain来查看变化。

Hint中的错误不会影响语句的执行，只是不能生效，该错误会根据语句类型以不同方式提示用户。对于explain语句，hint的错误会以warning形式显示在界面上，对于非explain语句，会以debug1级别日志显示在日志中，关键字为PLANHINT。

hint 的错误类型

- 语法错误
  语法规则树归约失败，会报错，指出出错的位置。

例如：hint关键字错误，leading hint或join hint指定2个表以下，其它hint未指定表等。一旦发现语法错误，则立即终止hint的解析，所以此时只有错误前面的解析完的hint有效。

例如：

```
leading((t1 t2)) nestloop(t1) rows(t1 t2 #10)
```

nestloop(t1)存在语法错误，则终止解析，可用hint只有之前解析的leading((t1 t2))。

- 语义错误
  - 表不存在，存在多个，或在leading或join中出现多次，均会报语义错误。
  - scanhint中的index不存在，会报语义错误。
  - 另外，如果子查询提升后，同一层出现多个名称相同的表，且其中某个表需要被hint，hint会存在歧义，无法使用，需要为相同表增加别名规避。
- hint重复或冲突

  如果存在hint重复或冲突，只有第一个hint生效，其它hint均会失效，会给出提示。
  - hint重复是指，hint的方法及表名均相同。例如：nestloop(t1 t2) nestloop(t1 t2)。
  - hint冲突是指，table list一样的hint，存在不一样的hint，hint的冲突仅对于每一类hint方法检测冲突。

    例如：nestloop (t1 t2) hashjoin (t1 t2)，则后面与前面冲突，此时hashjoin的hint失效。注意：nestloop(t1 t2)和no mergejoin(t1 t2)不冲突。

    须知

    leading hint中的多个表会进行拆解。例如：leading ((t1 t2 t3))会拆解成：leading((t1 t2)) leading(((t1 t2) t3))，此时如果存在leading((t2 t1))，则两者冲突，后面的会被丢弃。（例外：指定内外表的hint若与不指定内外表的hint重复，则始终丢弃不指定内外表的hint。）
- 子链接提升后hint失效

  子链接提升后的hint失效，会给出提示。通常出现在子链接中存在多个表连接的场景。提升后，子链接中的多个表不再作为一个整体出现在join中。
- 列类型不支持重分布
  - 对于skew hint来说，目的是为了进行重分布时的调优，所以当hint列的类型不支持重分布时，hint将无效。
- hint未被使用
  - 非等值join使用hashjoin hint或mergejoin hint
  - 不包含索引的表使用indexscan hint或indexonlyscan hint
  - 通常只有在索引列上使用过滤条件才会生成相应的索引路径，全表扫描将不会使用索引，因此使用indexscan hint或indexonlyscan hint将不会使用
  - indexonlyscan只有输出列仅包含索引列才会使用，否则指定时hint不会被使用
  - 多个表存在等值连接时，仅尝试有等值连接条件的表的连接，此时没有关联条件的表之间的路径将不会生成，所以指定相应的leading，join，rows hint将不使用，例如：t1 t2 t3表join，t1和t2, t2和t3有等值连接条件，则t1和t3不会优先连接，leading(t1 t3)不会被使用。

- 生成stream计划时，如果表的分布列与join列相同，则不会生成redistribute的计划；如果不同，且另一表分布列与join列相同，只能生成redistribute的计划，不会生成broadcast的计划，指定相应的hint则不会被使用。
- 对于AGG重分布列的hint，hint未被使用的可能原因如下：
  - 指定的分布键包含不支持重分布的数据类型。
  - 执行计划中不需要重分布。
  - 执行的分布键的序号有误。
  - 对于使用grouping sets子句和cube子句的AP函数，window agg中的分布键，不支持hint。

    说明

    指定Agg重分布列Hint，仅8.1.3.100及以上集群版本支持。
- 如果子链接未被提升，则blockname hint不会被使用。
- 对于skew hint，hint未被使用的可能原因如下：
  - 计划中不需要进行重分布。
  - hint指定的列为包含分布键。
  - hint指定倾斜信息有误或不完整，如对于join优化未指定值。
  - 倾斜优化的GUC参数处于关闭状态。
- 对于guc hint，hint未被使用的可能原因如下：
  - 配置参数不存在。
  - 配置参数不支持guc hint。
  - 配置参数的值无效。
  - 语句级别的guc hint没有被写在顶层查询中。
  - 子查询级别的guc hint设置的配置参数不支持在子查询级别设置。
  - guc hint所在的子查询被提升。

4.6.12 Plan Hint 实际调优案例

本节以TPC-DS标准测试的Q24的部分语句为例，在1000X，24DN环境上，说明使用plan hint进行实际调优的过程。示例如下：

```
select avg(netpaid) from
(select c_last_name
,c_first_name
,s_store_name
,ca_state
,s_state
,i_color
,i_current_price
,i_manager_id
,i_units
,i_size
,sum(ss_sales_price) netpaid
```

```
from store_sales
,store_returns
,store
,item
,customer
,customer_address
where ss_ticket_number = sr_ticket_number
and ss_item_sk = sr_item_sk
and ss_customer_sk = c_customer_sk
and ss_item_sk = i_item_sk
and ss_store_sk = s_store_sk
and c_birth_country = upper(ca_country)
and s_zip = ca_zip
and s_market_id=7
group by c_last_name
,c_first_name
,s_store_name
,ca_state
,s_state
,i_color
,i_current_price
,i_manager_id
,i_units
,i_size);
```

1. 该语句的初始计划如下，运行时间110s：

```
 id |                                    operation                                    |        A-time         |  A-rows   |  E-rows   |
----+---------------------------------------------------------------------------------+-----------------------+-----------+-----------+
  1 | ->  Row Adapter                                                                 | 110324.107            |         1 |         1 |
  2 |    ->  Vector Aggregate                                                         | 110324.093            |         1 |         1 |
  3 |       ->  Vector Streaming (type: GATHER)                                       | 110323.958            |        24 |        24 |
  4 |          ->  Vector Aggregate                                                   | [110179.302,110309.653] |        24 |        24 |
  5 |             ->  Vector Hash Aggregate                                           | [110178.388,110308.515] |    647824 |     16656 |
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                         | [77616.177,96478.771]   | 666834733 |     16664 |
  7 |                   ->  Vector Hash Join (8,22)                                   | [81727.257,84728.519]   | 666834733 |     16664 |
  8 |                      ->  Vector Streaming(type: REDISTRIBUTE)                   | [78770.520,82021.087]   | 666834733 |     16664 |
  9 |                         ->  Vector Hash Join (10,21)                            | [88066.755,90701.860]   | 666834733 |     16664 |
 10 |                            ->  Vector Streaming(type: BROADCAST)                | [7940.962,21430.725]    | 591882336 |     51360 |
 11 |                               ->  Vector Hash Join (12,20)                      | [2419.995,5319.606]     |  24661764 |      2140 |
 12 |                                  ->  Vector Streaming(type: REDISTRIBUTE)       | [1750.448,4659.581]     |  25258268 |      2241 |
 13 |                                     ->  Vector Hash Join (14,18)                | [15240.666,17159.616]   |  25258268 |      2241 |
 14 |                                        ->  Vector Hash Join (15,17)             | [12112.913,13563.366]   | 252564412 | 472070592 |
 15 |                                           ->  Vector Partition Iterator         | [11148.731,12473.230]   | 2879987999 | 2879987999 |
 16 |                                              ->  Partitioned CStore Scan on public.store_sales | [11097.921,12412.596] | 2879987999 | 2879987999 |
 17 |                                           ->  CStore Scan on public.store       | [0.447,0.689]           |      2064 |      2064 |
 18 |                                        ->  Vector Partition Iterator            | [296.805,319.014]       | 287999764 | 287999764 |
 19 |                                           ->  Partitioned CStore Scan on public.store_returns | [292.938,314.787] | 287999764 | 287999764 |
 20 |                                     ->  CStore Scan on public.customer          | [114.358,144.462]       |  12000000 |  12000000 |
 21 |                                  ->  CStore Scan on public.customer_address     | [38.426,56.753]         |   6000000 |   6000000 |
 22 |                            ->  CStore Scan on public.item                       | [3.160,5.026]           |    300000 |    300000 |
(22 rows)
```

该计划中，第10层算子使用broadcast性能较差，由于第11层算子估算行数为2140，比实际行数严重低估。错误行数估算主要来源于第13层算子的行数低估，根因是第13层hashjoin中，使用store_sales的(ss_ticket_number, ss_item_sk)列和store_returns的(sr_ticket_number, sr_item_sk)列进行关联，由于缺少多列相关性的估算导致行数严重低估。

2. 使用如下的rows hint进行调优后，计划如下，运行时间318s：

```
select avg(netpaid) from
(select /*+rows(store_sales store_returns * 11270)*/ c_last_name ...
```



```
 id |                                operation                                |        A-time         |   A-rows   |   E-rows   |
----+-------------------------------------------------------------------------+-----------------------+------------+------------+
  1 | ->  Row Adapter                                                         | 318585.246            |          1 |          1 |
  2 |    ->  Vector Aggregate                                                 | 318585.232            |          1 |          1 |
  3 |       ->  Vector Streaming (type: GATHER)                               | 318585.082            |         24 |         24 |
  4 |          ->  Vector Aggregate                                           | [318323.324,318499.290] |         24 |         24 |
  5 |             ->  Vector Hash Aggregate                                   | [318320.813,318497.054] |     647824 |  187770504 |
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                 | [288074.860,305601.698] |  666834733 |  187770507 |
  7 |                   ->  Vector Hash Join (8,22)                           | [253642.468,315808.664] |  666834733 |  187770507 |
  8 |                      ->  Vector Hash Join (9,18)                        | [250904.317,315684.018] |  666834733 |  187770507 |
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)        | [4552.500,310602.307]   |  275042158 |  147106999 |
 10 |                            ->  Vector Hash Join (11,17)                 | [7658.951,14053.823]    |  275042158 |  147106999 |
 11 |                               ->  Vector Streaming(type: REDISTRIBUTE)  | [3953.255,10264.943]    |  287999764 |  154060900 |
 12 |                                  ->  Vector Hash Join (13,15)           | [28196.188,32838.794]   |  287999764 |  154060900 |
 13 |                                     ->  Vector Partition Iterator       | [11477.673,12324.583]   | 2879987999 | 2879987999 |
 14 |                                        ->  Partitioned CStore Scan on public.store_sales | [11411.382,12250.209] | 2879987999 | 2879987999 |
 15 |                                     ->  Vector Partition Iterator       | [304.188,403.205]       |  287999764 |  287999764 |
 16 |                                        ->  Partitioned CStore Scan on public.store_returns | [299.838,398.255] |  287999764 |  287999764 |
 17 |                                  ->  CStore Scan on public.customer     | [122.246,170.128]       |   12000000 |   12000000 |
 18 |                         ->  Vector Streaming(type: REDISTRIBUTE)        | [57.558,117.461]        |     492915 |     146467 |
 19 |                            ->  Vector Hash Join (20,21)                 | [45.554,96.238]         |     492915 |     146467 |
 20 |                               ->  CStore Scan on public.customer_address | [39.738,89.412]       |    6000000 |    6000000 |
 21 |                               ->  CStore Scan on public.store           | [0.361,1.095]         |       2064 |       2064 |
 22 |                      ->  Vector Streaming(type: BROADCAST)              | [48.986,91.170]       |    7200000 |    7200000 |
 23 |                         ->  CStore Scan on public.item                  | [4.506,6.602]         |     300000 |     300000 |
(23 rows)
```

时间反而劣化了，原因是第8层hashjoin过慢引起第9层redistribute时间过慢导致，其中第9层redistribute并没有数据倾斜，hashjoin慢的原因是由于第18层redistribute后数据倾斜导致。

3. 经过实际数据查证，customer_address的两个join列的不同值数目较少，使用其进行join容易出现数据倾斜，故把customer_address放到最后进行join。使用如下的hint进行调优后，计划如下，运行时间116s：

```
select avg(netpaid) from
(select /*+rows(store_sales store_returns *11270)
leading((store_sales store_returns store item customer) customer_address)*/
c_last_name ...
```

```
 id |                                operation                                |        A-time         |   A-rows   |   E-rows   |
----+-------------------------------------------------------------------------+-----------------------+------------+------------+
  1 | ->  Row Adapter                                                         | 116326.597            |          1 |          1 |
  2 |    ->  Vector Aggregate                                                 | 116326.590            |          1 |          1 |
  3 |       ->  Vector Streaming (type: GATHER)                               | 116326.473            |         24 |         24 |
  4 |          ->  Vector Aggregate                                           | [116157.161,116236.494] |         24 |         24 |
  5 |             ->  Vector Hash Aggregate                                   | [116155.328,116233.946] |     647824 |  187770504 |
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                 | [84103.951,102052.326]  |  666834733 |  187770507 |
  7 |                   ->  Vector Hash Join (8,10)                           | [23229.469,47484.697]   |  666834733 |  187770507 |
  8 |                      ->  Vector Streaming(type: REDISTRIBUTE)           | [38.367,74.930]         |    6000000 |    6000000 |
  9 |                         ->  CStore Scan on public.customer_address      | [69.877,121.460]        |    6000000 |    6000000 |
 10 |                      ->  Vector Streaming(type: REDISTRIBUTE)           | [17404.744,17567.550]   |   24661764 |   24112909 |
 11 |                         ->  Vector Hash Join (12,22)                    | [16123.627,16397.246]   |   24661764 |   24112909 |
 12 |                            ->  Vector Streaming(type: REDISTRIBUTE)     | [15320.663,15741.646]   |   25258268 |   25252751 |
 13 |                               ->  Vector Hash Join (14,21)              | [14962.342,16375.458]   |   25258268 |   25252751 |
 14 |                                  ->  Vector Hash Join (15,19)           | [14449.031,15825.949]   |   25258268 |   25252751 |
 15 |                                     ->  Vector Hash Join (16,18)        | [11439.959,12510.065]   |  252564412 |  472070592 |
 16 |                                        ->  Vector Partition Iterator    | [10531.986,11536.213]   | 2879987999 | 2879987999 |
 17 |                                           ->  Partitioned CStore Scan on public.store_sales | [10483.634,11474.944] | 2879987999 | 2879987999 |
 18 |                                        ->  CStore Scan on public.store  | [0.347,0.463]           |       2064 |       2064 |
 19 |                                     ->  Vector Partition Iterator       | [293.977,365.021]       |  287999764 |  287999764 |
 20 |                                        ->  Partitioned CStore Scan on public.store_returns | [289.936,360.808] |  287999764 |  287999764 |
 21 |                                  ->  CStore Scan on public.item         | [3.109,5.245]           |     300000 |     300000 |
 22 |                               ->  CStore Scan on public.customer        | [113.871,141.791]       |   12000000 |   12000000 |
(22 rows)
```

发现时间基本花在了第6层redistribute算子上，需要进一步优化。

4. 由于最后一层redistribute包含倾斜，所以时间较长。为了避免倾斜，需要将item表放在最后join，由于item表的join并不能使行数减少。修改hint如下并执行，计划如下，运行时间120s：

```
select avg(netpaid) from
(select /*+rows(store_sales store_returns *11270)
leading((customer_address (store_sales store_returns store customer) item))
c_last_name ...
```

```
 id |                              operation                                 |         A-time          |   A-rows   |   E-rows   |
----+------------------------------------------------------------------------+-------------------------+------------+------------+
  1 | ->  Row Adapter                                                        | 120377.258              |          1 |          1 |
  2 |    ->  Vector Aggregate                                                | 120377.245              |          1 |          1 |
  3 |       ->  Vector Streaming (type: GATHER)                              | 120377.091              |         24 |         24 |
  4 |          ->  Vector Aggregate                                          | [120184.884,120301.704] |         24 |         24 |
  5 |             ->  Vector Hash Aggregate                                  | [120183.119,120297.845] |     647824 |  187770504 |
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                | [87775.682,106070.878]  |  666834733 |  187770507 |
  7 |                   ->  Vector Hash Join (8,22)                          | [22323.764,49878.523]   |  666834733 |  187770507 |
  8 |                      ->  Vector Hash Join (9,11)                       | [21129.236,45208.255]   |  666834733 |  187770507 |
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)       | [37.859,75.412]         |    6000000 |    6000000 |
 10 |                            ->  CStore Scan on public.customer_address  | [74.798,114.449]        |    6000000 |    6000000 |
 11 |                         ->  Vector Streaming(type: REDISTRIBUTE)       | [15714.458,15824.928]   |   24661764 |   24112909 |
 12 |                            ->  Vector Hash Join (13,21)                | [14637.516,14955.464]   |   24661764 |   24112909 |
 13 |                               ->  Vector Streaming(type: REDISTRIBUTE) | [13898.593,14333.200]   |   25258268 |   25252751 |
 14 |                                  ->  Vector Hash Join (15,19)          | [14166.917,15378.244]   |   25258268 |   25252751 |
 15 |                                     ->  Vector Hash Join (16,18)       | [11272.239,12052.532]   |  252564412 |  472070592 |
 16 |                                        ->  Vector Partition Iterator   | [10409.566,11127.981]   | 2879987999 | 2879987999 |
 17 |                                           ->  Partitioned CStore Scan on public.store_sales | [10365.838,11077.601] | 2879987999 | 2879987999 |
 18 |                                        ->  CStore Scan on public.store | [0.431,0.609]           |       2064 |       2064 |
 19 |                                     ->  Vector Partition Iterator      | [343.780,408.254]       |  287999764 |  287999764 |
 20 |                                        ->  Partitioned CStore Scan on public.store_returns | [339.844,403.923] | 287999764 | 287999764 |
 21 |                               ->  CStore Scan on public.customer       | [117.234,163.598]       |   12000000 |   12000000 |
 22 |                      ->  Vector Streaming(type: BROADCAST)             | [44.571,130.129]        |    7200000 |    7200000 |
 23 |                         ->  CStore Scan on public.item                 | [4.169,6.347]           |     300000 |     300000 |
(23 rows)
```

该计划中的redistribute问题并没有解决，因为第22层item表做了broadcast，导致与customer_address表join后的倾斜并没有被消除掉。

5. 增加如下禁止item表做broadcast的hint，使与customer_address join的表做redistribute（也可以进行join表redistribute的hint），计划如下，运行时间105s：

```
select avg(netpaid) from
(select /*+rows(store_sales store_returns *11270)
leading((customer_address (store_sales store_returns store customer) item))
no broadcast(item)*/
c_last_name ...
```

```
 id |                              operation                                 |         A-time          |   A-rows   |   E-rows   |
----+------------------------------------------------------------------------+-------------------------+------------+------------+
  1 | ->  Row Adapter                                                        | 105854.957              |          1 |          1 |
  2 |    ->  Vector Aggregate                                                | 105854.948              |          1 |          1 |
  3 |       ->  Vector Streaming (type: GATHER)                              | 105854.825              |         24 |         24 |
  4 |          ->  Vector Aggregate                                          | [105706.709,105776.135] |         24 |         24 |
  5 |             ->  Vector Hash Aggregate                                  | [105705.061,105773.013] |     647824 |  187770504 |
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                | [70701.966,89973.672]   |  666834733 |  187770507 |
  7 |                   ->  Vector Hash Join (8,23)                          | [71759.500,79018.433]   |  666834733 |  187770507 |
  8 |                      ->  Vector Streaming(type: REDISTRIBUTE)          | [69794.307,77269.178]   |  666834733 |  187770507 |
  9 |                         ->  Vector Hash Join (10,12)                   | [21443.307,46714.378]   |  666834733 |  187770507 |
 10 |                            ->  Vector Streaming(type: REDISTRIBUTE)    | [41.295,83.419]         |    6000000 |    6000000 |
 11 |                               ->  CStore Scan on public.customer_address | [70.405,166.072]      |    6000000 |    6000000 |
 12 |                            ->  Vector Streaming(type: REDISTRIBUTE)    | [15689.053,15788.475]   |   24661764 |   24112909 |
 13 |                               ->  Vector Hash Join (14,22)             | [14517.847,14712.929]   |   24661764 |   24112909 |
 14 |                                  ->  Vector Streaming(type: REDISTRIBUTE) | [13806.733,14089.770] |  25258268 |   25252751 |
 15 |                                     ->  Vector Hash Join (16,20)       | [13709.384,15095.449]   |   25258268 |   25252751 |
 16 |                                        ->  Vector Hash Join (17,19)    | [10944.796,11827.285]   |  252564412 |  472070592 |
 17 |                                           ->  Vector Partition Iterator | [10070.316,10884.728]  | 2879987999 | 2879987999 |
 18 |                                              ->  Partitioned CStore Scan on public.store_sales | [10018.966,10828.990] | 2879987999 | 2879987999 |
 19 |                                           ->  CStore Scan on public.store | [0.447,0.568]        |       2064 |       2064 |
 20 |                                        ->  Vector Partition Iterator   | [293.042,329.056]       |  287999764 |  287999764 |
 21 |                                           ->  Partitioned CStore Scan on public.store_returns | [288.631,324.782] | 287999764 | 287999764 |
 22 |                                  ->  CStore Scan on public.customer    | [113.735,138.235]       |   12000000 |   12000000 |
 23 |                      ->  CStore Scan on public.item                    | [3.127,5.357]           |     300000 |     300000 |
(23 rows)
```

6. 发现最后一层使用单层Agg，但行数缩减较多。使用相同的hint，同时结合参数best_agg_plan=3进行双层Agg调优，最终计划如下图所示，运行时间94s，完成调优。

```
 id |                                                operation                                                 |        A-time         |  A-rows  |  E-rows   |
----+----------------------------------------------------------------------------------------------------------+-----------------------+----------+-----------+
  1 | ->  Row Adapter                                                                                          | 94004.670             |        1 |         1 |
  2 |    ->  Vector Aggregate                                                                                  | 94004.655             |        1 |         1 |
  3 |       ->  Vector Streaming (type: GATHER)                                                                | 94004.504             |       24 |        24 |
  4 |          ->  Vector Aggregate                                                                            | [93833.832,93928.052] |       24 |        24 |
  5 |             ->  Vector Hash Aggregate                                                                    | [93832.460,93926.412] |   647824 | 187770507 |
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                                                  | [93640.866,93787.939] |   647824 | 183912384 |
  7 |                   ->  Vector Hash Aggregate                                                              | [93687.544,93791.242] |   647824 | 183912384 |
  8 |                      ->  Vector Hash Join (9,24)                                                         | [70025.469,72773.161] |   666834733 | 187770507 |
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)                                         | [68242.223,71275.972] |   666834733 | 187770507 |
 10 |                            ->  Vector Hash Join (11,13)                                                  | [21421.136,44830.306] |   666834733 | 187770507 |
 11 |                               ->  Vector Streaming(type: REDISTRIBUTE)                                   | [35.444,71.328]       |   6000000 |   6000000 |
 12 |                                  ->  CStore Scan on public.customer_address                              | [67.246,119.224]      |   6000000 |   6000000 |
 13 |                               ->  Vector Streaming(type: REDISTRIBUTE)                                   | [16089.853,16212.570] |   24661764 |  24112909 |
 14 |                                  ->  Vector Hash Join (15,23)                                            | [14822.972,15188.942] |   24661764 |  24112909 |
 15 |                                     ->  Vector Streaming(type: REDISTRIBUTE)                             | [14061.867,14604.162] |   25258268 |  25252751 |
 16 |                                        ->  Vector Hash Join (17,21)                                      | [13949.756,15492.311] |   25258268 |  25252751 |
 17 |                                           ->  Vector Hash Join (18,20)                                   | [10935.742,12160.719] |   252564412 | 472070592 |
 18 |                                              ->  Vector Partition Iterator                               | [10052.958,11194.962] |   2879987999 | 2879987999 |
 19 |                                                 ->  Partitioned CStore Scan on public.store_sales        | [10008.415,11143.984] |   2879987999 | 2879987999 |
 20 |                                              ->  CStore Scan on public.store                             | [0.452,0.839]         |   2064 |      2064 |
 21 |                                           ->  Vector Partition Iterator                                  | [298.235,332.736]     |   287999764 | 287999764 |
 22 |                                              ->  Partitioned CStore Scan on public.store_returns         | [294.067,327.629]     |   287999764 | 287999764 |
 23 |                                     ->  CStore Scan on public.customer                                   | [114.377,145.156]     |   12000000 |  12000000 |
 24 |                         ->  CStore Scan on public.item                                                   | [3.150,3.530]         |   300000 |    300000 |
(24 rows)
```

如果有统计信息变更引起的查询劣化，可以考虑用plan hint来调整到之前的查询计划。这里以TPCH-Q17为例，在收集default_statistics_target设置为–2的统计信息之后，计划相比于默认统计信息发生劣化。

1. 默认统计信息（default_statistics_target设置为100）的计划如下：

```
 id |                                                operation                                                 |          A-time
----+----------------------------------------------------------------------------------------------------------+---------------------------
  1 | ->  Row Adapter                                                                                          | 265006.779
  2 |    ->  Vector Aggregate                                                                                  | 265006.764
  3 |       ->  Vector Streaming (type: GATHER)                                                                | 265006.071
  4 |          ->  Vector Aggregate                                                                            | [263699.512,264503.084]
  5 |             ->  Vector Hash Join (6,17)                                                                  | [263676.665,264477.932]
  6 |                ->  Vector Streaming(type: LOCAL GATHER dop: 1/4)                                         | [1.998,7.594]
  7 |                   ->  Vector Hash Aggregate                                                              | [201775.393,202432.672]
  8 |                      ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 4/4)                             | [201567.130,202231.524]
  9 |                         ->  Vector Hash Join (10,12)                                                     | [170675.231,199908.410]
 10 |                            ->  Vector Partition Iterator                                                 | [34847.797,51968.266]
 11 |                               ->  Partitioned CStore Scan on tpch10wx_col.lineitem                       | [33805.013,51137.657]
 12 |                            ->  Vector Hash Aggregate                                                     | [23283.387,25359.493]
 13 |                               ->  Vector Streaming(type: SPLIT BROADCAST dop: 4/4)                       | [12850.624,14608.515]
 14 |                                  ->  Vector Hash Aggregate                                               | [2690.439,3616.623]
 15 |                                     ->  Vector Partition Iterator                                        | [2659.700,3579.390]
 16 |                                        ->  Partitioned CStore Scan on tpch10wx_col.part                  | [2642.213,3559.093]
 17 |                ->  Vector Streaming(type: REDISTRIBUTE dop: 1/4)                                         | [262300.732,262961.078]
 18 |                   ->  Vector Hash Join (19,21)                                                           | [225749.727,260990.322]
 19 |                      ->  Vector Partition Iterator                                                       | [40046.078,56220.694]
 20 |                         ->  Partitioned CStore Scan on tpch10wx_col.lineitem                             | [39204.414,55328.448]
 21 |                      ->  Vector Streaming(type: SPLIT BROADCAST dop: 4/4)                                | [55748.177,61987.136]
 22 |                         ->  Vector Partition Iterator                                                    | [3042.864,3873.942]
 23 |                            ->  Partitioned CStore Scan on tpch10wx_col.part                              | [3027.023,3848.159]
(23 rows)
```

2. 统计信息变更（default_statistics_target设置为–2）的计划如下：

```
 id |                                                operation                                                 |          A-time
----+----------------------------------------------------------------------------------------------------------+---------------------------
  1 | ->  Row Adapter                                                                                          | 1440492.994
  2 |    ->  Vector Aggregate                                                                                  | 1440492.982
  3 |       ->  Vector Streaming (type: GATHER)                                                                | 1440491.021
  4 |          ->  Vector Streaming(type: LOCAL GATHER dop: 1/6)                                               | [1439737.284,1440008.568]
  5 |             ->  Vector Aggregate                                                                         | [1439008.369,1439854.148]
  6 |                ->  Vector Hash Join (7,18)                                                               | [1439006.016,1439851.619]
  7 |                   ->  Vector Streaming(type: LOCAL BROADCAST dop: 6/6)                                   | [2.932,139.405]
  8 |                      ->  Vector Hash Aggregate                                                           | [190452.312,195910.748]
  9 |                         ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 6/6)                          | [190171.929,195653.119]
 10 |                            ->  Vector Hash Join (11,13)                                                  | [161076.195,178831.123]
 11 |                               ->  Vector Partition Iterator                                              | [27306.318,45564.565]
 12 |                                  ->  Partitioned CStore Scan on tpch10wx_col.lineitem                    | [26752.444,44912.020]
 13 |                               ->  Vector Hash Aggregate                                                  | [35601.624,39812.058]
 14 |                                  ->  Vector Streaming(type: SPLIT BROADCAST dop: 6/6)                    | [23096.460,27057.137]
 15 |                                     ->  Vector Hash Aggregate                                            | [2372.587,3052.445]
 16 |                                        ->  Vector Partition Iterator                                     | [2345.381,3012.732]
 17 |                                           ->  Partitioned CStore Scan on tpch10wx_col.part               | [2329.874,2989.393]
 18 |                   ->  Vector Hash Join (19,22)                                                           | [1437388.414,1438470.781]
 19 |                      ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 6/6)                             | [1392693.529,1408571.859]
 20 |                         ->  Vector Partition Iterator                                                    | [29065.204,41264.514]
 21 |                            ->  Partitioned CStore Scan on tpch10wx_col.lineitem                          | [28212.219,40133.491]
 22 |                      ->  Vector Streaming(type: LOCAL REDISTRIBUTE dop: 6/6)                             | [2570.841,3438.567]
 23 |                         ->  Vector Partition Iterator                                                    | [2447.569,3276.369]
 24 |                            ->  Partitioned CStore Scan on tpch10wx_col.part                              | [2432.124,3263.641]
(24 rows)
```

3. 经过对比，劣化的原因主要为lineitem和part表join时stream类型由BroadCast变更为Redistribute导致。可以对语句进行stream方式的hint来调整到之前的计划，例如：

```
select /*+ no redistribute(part lineitem) */
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    lineitem,
    part
where
    p_partkey = l_partkey
    and p_brand = 'Brand#23'
    and p_container = 'MED BOX'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem
        where
            l_partkey = p_partkey
    );
```

4.6.13 Plan Hint 实际调优案例

本节以TPC-DS标准测试的Q24的部分语句为例，在1000X，24DN环境上，说明使用plan hint进行实际调优的过程。示例如下：

```
select avg(netpaid) from
(select c_last_name
,c_first_name
,s_store_name
,ca_state
,s_state
,i_color
,i_current_price
,i_manager_id
,i_units
,i_size
,sum(ss_sales_price) netpaid
from store_sales
,store_returns
,store
,item
,customer
,customer_address
where ss_ticket_number = sr_ticket_number
and ss_item_sk = sr_item_sk
and ss_customer_sk = c_customer_sk
and ss_item_sk = i_item_sk
and ss_store_sk = s_store_sk
and c_birth_country = upper(ca_country)
and s_zip = ca_zip
and s_market_id=7
group by c_last_name
,c_first_name
,s_store_name
,ca_state
,s_state
,i_color
,i_current_price
,i_manager_id
,i_units
,i_size);
```

1. 该语句的初始计划如下，运行时间110s：

```
 id |                                operation                                 |        A-time         |  A-rows   |  E-rows
----+--------------------------------------------------------------------------+-----------------------+-----------+-----------+
  1 | ->  Row Adapter                                                          | 110324.107            |         1 |         1 |
  2 |    ->  Vector Aggregate                                                  | 110324.093            |         1 |         1 |
  3 |       ->  Vector Streaming (type: GATHER)                                | 110323.958            |        24 |        24 |
  4 |          ->  Vector Aggregate                                            | [110179.302,110309.653] |        24 |        24 |
  5 |             ->  Vector Hash Aggregate                                    | [110178.388,110308.515] |    647824 |     16656 |
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                  | [77616.177,96478.771] |    666834733 |     16664 |
  7 |                   ->  Vector Hash Join (8,22)                            | [81727.257,84728.519] |    666834733 |     16664 |
  8 |                      ->  Vector Streaming(type: REDISTRIBUTE)            | [78770.520,82021.087] |    666834733 |     16664 |
  9 |                         ->  Vector Hash Join (10,21)                     | [88066.755,90701.860] |    666834733 |     16664 |
 10 |                            ->  Vector Streaming(type: BROADCAST)         | [7940.962,21430.725]  |    591882336 |     51360 |
 11 |                               ->  Vector Hash Join (12,20)               | [2419.995,5319.606]   |     24661764 |      2140 |
 12 |                                  ->  Vector Streaming(type: REDISTRIBUTE) | [1750.448,4659.581]   |     25258268 |      2241 |
 13 |                                     ->  Vector Hash Join (14,18)         | [15240.666,17159.616] |     25258268 |      2241 |
 14 |                                        ->  Vector Hash Join (15,17)      | [12112.913,13563.366] |    252564412 |    472070592 |
 15 |                                           ->  Vector Partition Iterator  | [11148.731,12473.230] |   2879987999 |   2879987999 |
 16 |                                              ->  Partitioned CStore Scan on public.store_sales | [11097.921,12412.596] |   2879987999 |   2879987999 |
 17 |                                           ->  CStore Scan on public.store | [0.447,0.689]         |      2064 |      2064 |
 18 |                                        ->  Vector Partition Iterator     | [296.805,319.014]     |    287999764 |    287999764 |
 19 |                                           ->  Partitioned CStore Scan on public.store_returns | [292.938,314.787] |    287999764 |    287999764 |
 20 |                                  ->  CStore Scan on public.customer      | [114.358,144.462]     |     12000000 |     12000000 |
 21 |                               ->  CStore Scan on public.customer_address | [38.426,56.753]       |      6000000 |      6000000 |
 22 |                         ->  CStore Scan on public.item                   | [3.160,5.026]         |       300000 |       300000 |
(22 rows)
```

该计划中，第10层算子使用broadcast性能较差，由于第11层算子估算行数为2140，比实际行数严重低估。错误行数估算主要来源于第13层算子的行数低估，根因是第13层hashjoin中，使用store_sales的(ss_ticket_number, ss_item_sk)列和store_returns的(sr_ticket_number, sr_item_sk)列进行关联，由于缺少多列相关性的估算导致行数严重低估。

2. 使用如下的rows hint进行调优后，计划如下，运行时间318s：

```
select avg(netpaid) from
(select /*+rows(store_sales store_returns * 11270)*/ c_last_name ...
```

```
 id |                                operation                                 |        A-time         |  A-rows   |  E-rows
----+--------------------------------------------------------------------------+-----------------------+-----------+-----------+
  1 | ->  Row Adapter                                                          | 318585.246            |         1 |         1 |
  2 |    ->  Vector Aggregate                                                  | 318585.232            |         1 |         1 |
  3 |       ->  Vector Streaming (type: GATHER)                                | 318585.082            |        24 |        24 |
  4 |          ->  Vector Aggregate                                            | [318323.324,318499.290] |        24 |        24 |
  5 |             ->  Vector Hash Aggregate                                    | [318320.813,318497.054] |    647824 |  187770504 |
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                  | [288074.860,305601.698] |    666834733 |  187770507 |
  7 |                   ->  Vector Hash Join (8,22)                            | [253642.468,315808.664] |    666834733 |  187770507 |
  8 |                      ->  Vector Hash Join (9,18)                         | [250904.317,315684.018] |    666834733 |  187770507 |
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)         | [4552.500,310602.307] |    275042158 |  147106999 |
 10 |                            ->  Vector Hash Join (11,17)                  | [7658.951,14053.823]  |    275042158 |  147106999 |
 11 |                               ->  Vector Streaming(type: REDISTRIBUTE)   | [3953.255,10264.943]  |    287999764 |  154060900 |
 12 |                                  ->  Vector Hash Join (13,15)            | [28196.188,32838.794] |    287999764 |  154060900 |
 13 |                                     ->  Vector Partition Iterator        | [11477.673,12324.583] |   2879987999 |   2879987999 |
 14 |                                        ->  Partitioned CStore Scan on public.store_sales | [11411.382,12250.209] |   2879987999 |   2879987999 |
 15 |                                     ->  Vector Partition Iterator        | [304.188,403.205]     |    287999764 |    287999764 |
 16 |                                        ->  Partitioned CStore Scan on public.store_returns | [299.838,398.255] |    287999764 |    287999764 |
 17 |                                  ->  CStore Scan on public.customer      | [122.246,170.128]     |     12000000 |     12000000 |
 18 |                               ->  Vector Streaming(type: REDISTRIBUTE)   | [57.558,117.461]      |       492915 |       146467 |
 19 |                                  ->  Vector Hash Join (20,21)            | [45.554,96.238]       |       492915 |       146467 |
 20 |                                     ->  CStore Scan on public.customer_address | [39.738,89.412]   |      6000000 |      6000000 |
 21 |                                     ->  CStore Scan on public.store      | [0.361,1.095]         |         2064 |         2064 |
 22 |                            ->  Vector Streaming(type: BROADCAST)         | [48.986,91.170]       |      7200000 |      7200000 |
 23 |                               ->  CStore Scan on public.item             | [4.506,6.602]         |       300000 |       300000 |
(23 rows)
```

时间反而劣化了，原因是第8层hashjoin过慢引起第9层redistribute时间过慢导致，其中第9层redistribute并没有数据倾斜，hashjoin慢的原因是由于第18层redistribute后数据倾斜导致。

3. 经过实际数据查证，customer_address的两个join列的不同值数目较少，使用其进行join容易出现数据倾斜，故把customer_address放到最后进行join。使用如下的hint进行调优后，计划如下，运行时间116s：

```
select avg(netpaid) from
(select /*+rows(store_sales store_returns *11270)
leading((store_sales store_returns store item customer) customer_address)*/
c_last_name ...
```

```
 id |                                operation                                 |        A-time         |  A-rows   |  E-rows
----+--------------------------------------------------------------------------+-----------------------+-----------+-----------
  1 | ->  Row Adapter                                                          | 116326.597            |         1 |         1
  2 |    ->  Vector Aggregate                                                  | 116326.590            |         1 |         1
  3 |       ->  Vector Streaming (type: GATHER)                                | 116326.473            |        24 |        24
  4 |          ->  Vector Aggregate                                            | [116157.161,116236.494] |        24 |        24
  5 |             ->  Vector Hash Aggregate                                    | [116155.328,116233.946] |    647824 |  187770504
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                  | [84103.951,102052.326] |    666834733 |  187770507
  7 |                   ->  Vector Hash Join (8,10)                            | [23229.469,47484.697] |    666834733 |  187770507
  8 |                      ->  Vector Streaming(type: REDISTRIBUTE)            | [38.367,74.930]       |   6000000 |    6000000
  9 |                         ->  CStore Scan on public.customer_address       | [69.877,121.460]      |   6000000 |    6000000
 10 |                      ->  Vector Streaming(type: REDISTRIBUTE)            | [17404.744,17567.550] |   24661764 |   24112909
 11 |                         ->  Vector Hash Join (12,22)                     | [16123.627,16397.246] |   24661764 |   24112909
 12 |                            ->  Vector Streaming(type: REDISTRIBUTE)      | [15320.663,15741.646] |   25258268 |   25252751
 13 |                               ->  Vector Hash Join (14,21)               | [14962.342,16375.458] |   25258268 |   25252751
 14 |                                  ->  Vector Hash Join (15,19)            | [14449.031,15825.949] |   25258268 |   25252751
 15 |                                     ->  Vector Hash Join (16,18)         | [11439.959,12510.065] |  252564412 |  472070592
 16 |                                        ->  Vector Partition Iterator     | [10531.986,11536.213] | 2879987999 | 2879987999
 17 |                                           ->  Partitioned CStore Scan on public.store_sales | [10483.634,11474.944] | 2879987999 | 2879987999
 18 |                                        ->  CStore Scan on public.store   | [0.347,0.463]         |      2064 |       2064
 19 |                                     ->  Vector Partition Iterator        | [293.977,365.021]     |  287999764 |  287999764
 20 |                                        ->  Partitioned CStore Scan on public.store_returns | [289.936,360.808] |  287999764 |  287999764
 21 |                                  ->  CStore Scan on public.item          | [3.109,5.245]         |    300000 |     300000
 22 |                               ->  CStore Scan on public.customer         | [113.871,141.791]     |  12000000 |   12000000
(22 rows)
```

发现时间基本花在了第6层redistribute算子上，需要进一步优化。

4. 由于最后一层redistribute包含倾斜，所以时间较长。为了避免倾斜，需要将item表放在最后join，由于item表的join并不能使行数减少。修改hint如下并执行，计划如下，运行时间120s：

```
select avg(netpaid) from
(select /*+rows(store_sales store_returns *11270)
leading((customer_address (store_sales store_returns store customer) item))
c_last_name ...
```

```
 id |                                operation                                 |        A-time         |  A-rows   |  E-rows
----+--------------------------------------------------------------------------+-----------------------+-----------+-----------
  1 | ->  Row Adapter                                                          | 120377.258            |         1 |         1
  2 |    ->  Vector Aggregate                                                  | 120377.245            |         1 |         1
  3 |       ->  Vector Streaming (type: GATHER)                                | 120377.091            |        24 |        24
  4 |          ->  Vector Aggregate                                            | [120184.884,120301.704] |        24 |        24
  5 |             ->  Vector Hash Aggregate                                    | [120183.119,120297.845] |    647824 |  187770504
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                  | [87775.682,106070.878] |    666834733 |  187770507
  7 |                   ->  Vector Hash Join (8,22)                            | [22323.764,49878.523] |    666834733 |  187770507
  8 |                      ->  Vector Hash Join (9,11)                         | [21129.236,45208.255] |    666834733 |  187770507
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)         | [37.859,75.412]       |   6000000 |    6000000
 10 |                            ->  CStore Scan on public.customer_address    | [74.798,114.449]      |   6000000 |    6000000
 11 |                         ->  Vector Streaming(type: REDISTRIBUTE)         | [15714.458,15824.928] |   24661764 |   24112909
 12 |                            ->  Vector Hash Join (13,21)                  | [14637.516,14955.464] |   24661764 |   24112909
 13 |                               ->  Vector Streaming(type: REDISTRIBUTE)   | [13898.593,14333.200] |   25258268 |   25252751
 14 |                                  ->  Vector Hash Join (15,19)            | [14166.917,15378.244] |   25258268 |   25252751
 15 |                                     ->  Vector Hash Join (16,18)         | [11272.239,12052.532] |  252564412 |  472070592
 16 |                                        ->  Vector Partition Iterator     | [10409.566,11127.981] | 2879987999 | 2879987999
 17 |                                           ->  Partitioned CStore Scan on public.store_sales | [10365.838,11077.601] | 2879987999 | 2879987999
 18 |                                        ->  CStore Scan on public.store   | [0.431,0.609]         |      2064 |       2064
 19 |                                     ->  Vector Partition Iterator        | [343.780,408.254]     |  287999764 |  287999764
 20 |                                        ->  Partitioned CStore Scan on public.store_returns | [339.844,403.923] |  287999764 |  287999764
 21 |                                  ->  CStore Scan on public.customer      | [117.234,163.598]     |  12000000 |   12000000
 22 |                      ->  Vector Streaming(type: BROADCAST)               | [44.571,130.129]      |   7200000 |    7200000
 23 |                         ->  CStore Scan on public.item                   | [4.169,6.347]         |    300000 |     300000
(23 rows)
```

该计划中的redistribute问题并没有解决，因为第22层item表做了broadcast，导致与customer_address表join后的倾斜并没有被消除掉。

5. 增加如下禁止item表做broadcast的hint，使与customer_address join的表做redistribute（也可以进行join表redistribute的hint），计划如下，运行时间105s：

```
select avg(netpaid) from
(select /*+rows(store_sales store_returns *11270)
leading((customer_address (store_sales store_returns store customer) item))
no broadcast(item)*/
c_last_name ...
```

```
 id |                                                operation                                                 |        A-time         |  A-rows  |  E-rows
----+----------------------------------------------------------------------------------------------------------+-----------------------+----------+----------
  1 | ->  Row Adapter                                                                                          | 105854.957            |        1 |        1 |
  2 |    ->  Vector Aggregate                                                                                  | 105854.948            |        1 |        1 |
  3 |       ->  Vector Streaming (type: GATHER)                                                                | 105854.825            |       24 |       24 |
  4 |          ->  Vector Aggregate                                                                            | [105706.709,105776.135] |       24 |       24 |
  5 |             ->  Vector Hash Aggregate                                                                    | [105705.061,105773.013] |   647824 |  187770504 |
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                                                  | [70701.966,89973.672] |   666834733 |  187770507 |
  7 |                   ->  Vector Hash Join (8,23)                                                            | [71759.500,79018.433] |   666834733 |  187770507 |
  8 |                      ->  Vector Streaming(type: REDISTRIBUTE)                                            | [69794.307,77269.178] |   666834733 |  187770507 |
  9 |                         ->  Vector Hash Join (10,12)                                                     | [21443.307,46714.378] |   666834733 |  187770507 |
 10 |                            ->  Vector Streaming(type: REDISTRIBUTE)                                      | [41.295,83.419]       |  6000000 |  6000000 |
 11 |                               ->  CStore Scan on public.customer_address                                 | [70.405,166.072]      |  6000000 |  6000000 |
 12 |                            ->  Vector Streaming(type: REDISTRIBUTE)                                      | [15689.053,15788.475] |  24661764 |  24112909 |
 13 |                               ->  Vector Hash Join (14,22)                                               | [14517.847,14712.929] |  24661764 |  24112909 |
 14 |                                  ->  Vector Streaming(type: REDISTRIBUTE)                                | [13806.733,14089.770] |  25258268 |  25252751 |
 15 |                                     ->  Vector Hash Join (16,20)                                         | [13709.384,15095.449] |  25258268 |  25252751 |
 16 |                                        ->  Vector Hash Join (17,19)                                      | [10944.796,11827.285] |  252564412 |  472070592 |
 17 |                                           ->  Vector Partition Iterator                                  | [10070.316,10884.728] | 2879987999 | 2879987999 |
 18 |                                              ->  Partitioned CStore Scan on public.store_sales           | [10018.966,10828.990] | 2879987999 | 2879987999 |
 19 |                                           ->  CStore Scan on public.store                                | [0.447,0.568]         |     2064 |     2064 |
 20 |                                        ->  Vector Partition Iterator                                     | [293.042,329.056]     |  287999764 |  287999764 |
 21 |                                           ->  Partitioned CStore Scan on public.store_returns            | [288.631,324.782]     |  287999764 |  287999764 |
 22 |                                  ->  CStore Scan on public.customer                                      | [113.735,138.235]     |  12000000 |  12000000 |
 23 |                      ->  CStore Scan on public.item                                                      | [3.127,5.357]         |   300000 |   300000 |
(23 rows)
```

6. 发现最后一层使用单层Agg，但行数缩减较多。使用相同的hint，同时结合参数best_agg_plan=3进行双层Agg调优，最终计划如下图所示，运行时间94s，完成调优。

```
 id |                                                operation                                                 |        A-time         |  A-rows  |  E-rows
----+----------------------------------------------------------------------------------------------------------+-----------------------+----------+----------
  1 | ->  Row Adapter                                                                                          | 94004.670             |        1 |        1 |
  2 |    ->  Vector Aggregate                                                                                  | 94004.655             |        1 |        1 |
  3 |       ->  Vector Streaming (type: GATHER)                                                                | 94004.504             |       24 |       24 |
  4 |          ->  Vector Aggregate                                                                            | [93833.832,93928.052] |       24 |       24 |
  5 |             ->  Vector Hash Aggregate                                                                    | [93832.460,93926.412] |   647824 |  187770507 |
  6 |                ->  Vector Streaming(type: REDISTRIBUTE)                                                  | [93640.866,93787.939] |   647824 |  183912384 |
  7 |                   ->  Vector Hash Aggregate                                                              | [93687.544,93791.242] |   647824 |  183912384 |
  8 |                      ->  Vector Hash Join (9,24)                                                         | [70025.469,72773.161] |   666834733 |  187770507 |
  9 |                         ->  Vector Streaming(type: REDISTRIBUTE)                                         | [68242.223,71275.972] |   666834733 |  187770507 |
 10 |                            ->  Vector Hash Join (11,13)                                                  | [21421.136,44830.306] |   666834733 |  187770507 |
 11 |                               ->  Vector Streaming(type: REDISTRIBUTE)                                   | [35.444,71.328]       |  6000000 |  6000000 |
 12 |                                  ->  CStore Scan on public.customer_address                              | [67.246,119.224]      |  6000000 |  6000000 |
 13 |                               ->  Vector Streaming(type: REDISTRIBUTE)                                   | [16089.853,16212.570] |  24661764 |  24112909 |
 14 |                                  ->  Vector Hash Join (15,23)                                            | [14822.972,15188.942] |  24661764 |  24112909 |
 15 |                                     ->  Vector Streaming(type: REDISTRIBUTE)                             | [14061.867,14604.162] |  25258268 |  25252751 |
 16 |                                        ->  Vector Hash Join (17,21)                                      | [13949.756,15492.311] |  25258268 |  25252751 |
 17 |                                           ->  Vector Hash Join (18,20)                                   | [10935.742,12160.719] |  252564412 |  472070592 |
 18 |                                              ->  Vector Partition Iterator                               | [10052.958,11194.962] | 2879987999 | 2879987999 |
 19 |                                                 ->  Partitioned CStore Scan on public.store_sales        | [10008.415,11143.984] | 2879987999 | 2879987999 |
 20 |                                              ->  CStore Scan on public.store                             | [0.452,0.839]         |     2064 |     2064 |
 21 |                                           ->  Vector Partition Iterator                                  | [298.235,332.736]     |  287999764 |  287999764 |
 22 |                                              ->  Partitioned CStore Scan on public.store_returns         | [294.067,327.629]     |  287999764 |  287999764 |
 23 |                                        ->  CStore Scan on public.customer                                | [114.377,145.156]     |  12000000 |  12000000 |
 24 |                      ->  CStore Scan on public.item                                                      | [3.150,3.530]         |   300000 |   300000 |
(24 rows)
```

如果有统计信息变更引起的查询劣化，可以考虑用plan hint来调整到之前的查询计划。这里以TPCH-Q17为例，在收集default_statistics_target设置为–2的统计信息之后，计划相比于默认统计信息发生劣化。

1. 默认统计信息（ default_statistics_target设置为100 ）的计划如下：

```
 id |                                                operation                                                 |        A-time
----+----------------------------------------------------------------------------------------------------------+-----------------------
  1 | ->  Row Adapter                                                                                          | 265006.779
  2 |    ->  Vector Aggregate                                                                                  | 265006.764
  3 |       ->  Vector Streaming (type: GATHER)                                                                | 265006.071
  4 |          ->  Vector Aggregate                                                                            | [263699.512,264503.084]
  5 |             ->  Vector Hash Join (6,17)                                                                  | [263676.665,264477.932]
  6 |                ->  Vector Streaming(type: LOCAL GATHER dop: 1/4)                                         | [1.998,7.594]
  7 |                   ->  Vector Hash Aggregate                                                              | [201775.393,202432.672]
  8 |                      ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 4/4)                             | [201567.130,202231.524]
  9 |                         ->  Vector Hash Join (10,12)                                                     | [170675.231,199908.410]
 10 |                            ->  Vector Partition Iterator                                                 | [34847.797,51968.266]
 11 |                               ->  Partitioned CStore Scan on tpch10wx_col.lineitem                       | [33805.013,51137.657]
 12 |                            ->  Vector Hash Aggregate                                                     | [23283.387,25359.493]
 13 |                               ->  Vector Streaming(type: SPLIT BROADCAST dop: 4/4)                       | [12850.624,14608.515]
 14 |                                  ->  Vector Hash Aggregate                                               | [2690.439,3616.623]
 15 |                                     ->  Vector Partition Iterator                                        | [2659.700,3579.390]
 16 |                                        ->  Partitioned CStore Scan on tpch10wx_col.part                  | [2642.213,3559.093]
 17 |                ->  Vector Streaming(type: REDISTRIBUTE dop: 1/4)                                         | [262300.732,262961.078]
 18 |                   ->  Vector Hash Join (19,21)                                                           | [225749.727,260990.322]
 19 |                      ->  Vector Partition Iterator                                                       | [40046.078,56220.694]
 20 |                         ->  Partitioned CStore Scan on tpch10wx_col.lineitem                             | [39204.414,55328.448]
 21 |                      ->  Vector Streaming(type: SPLIT BROADCAST dop: 4/4)                                | [55748.177,61987.136]
 22 |                         ->  Vector Partition Iterator                                                    | [3042.864,3873.942]
 23 |                            ->  Partitioned CStore Scan on tpch10wx_col.part                              | [3027.023,3848.159]
(23 rows)
```

2. 统计信息变更（ default_statistics_target设置为–2 ）的计划如下：

```
id |                                operation                                 |          A-time
---+--------------------------------------------------------------------------+---------------------------
 1 | ->  Row Adapter                                                          | 1440492.994
 2 |    ->  Vector Aggregate                                                  | 1440492.982
 3 |       ->  Vector Streaming (type: GATHER)                                | 1440491.021
 4 |          ->  Vector Streaming(type: LOCAL GATHER dop: 1/6)               | [1439737.284,1440008.568]
 5 |             ->  Vector Aggregate                                         | [1439008.369,1439854.148]
 6 |                ->  Vector Hash Join (7,18)                               | [1439006.016,1439851.619]
 7 |                   ->  Vector Streaming(type: LOCAL BROADCAST dop: 6/6)    | [2.932,139.405]
 8 |                      ->  Vector Hash Aggregate                           | [190452.312,195910.748]
 9 |                         ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 6/6) | [190171.929,195653.119]
10 |                            ->  Vector Hash Join (11,13)                  | [161076.195,178831.123]
11 |                               ->  Vector Partition Iterator              | [27306.318,45564.565]
12 |                                  ->  Partitioned CStore Scan on tpch10wx_col.lineitem | [26752.444,44912.020]
13 |                               ->  Vector Hash Aggregate                  | [35601.624,39812.058]
14 |                                  ->  Vector Streaming(type: SPLIT BROADCAST dop: 6/6) | [23096.460,27057.137]
15 |                                     ->  Vector Hash Aggregate            | [2372.587,3052.445]
16 |                                        ->  Vector Partition Iterator     | [2345.381,3012.732]
17 |                                           ->  Partitioned CStore Scan on tpch10wx_col.part | [2329.874,2989.393]
18 |                   ->  Vector Hash Join (19,22)                           | [1437388.414,1438470.781]
19 |                      ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 6/6) | [1392693.529,1408571.859]
20 |                         ->  Vector Partition Iterator                    | [29065.204,41264.514]
21 |                            ->  Partitioned CStore Scan on tpch10wx_col.lineitem | [28212.219,40133.491]
22 |                      ->  Vector Streaming(type: LOCAL REDISTRIBUTE dop: 6/6) | [2570.841,3438.567]
23 |                         ->  Vector Partition Iterator                    | [2447.569,3276.369]
24 |                            ->  Partitioned CStore Scan on tpch10wx_col.part | [2432.124,3263.641]
(24 rows)
```

3. 经过对比，劣化的原因主要为lineitem和part表join时stream类型由BroadCast变更为Redistribute导致。可以对语句进行stream方式的hint来调整到之前的计划，例如：

```
select /*+ no redistribute(part lineitem) */
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    lineitem,
    part
where
    p_partkey = l_partkey
    and p_brand = 'Brand#23'
    and p_container = 'MED BOX'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem
        where
            l_partkey = p_partkey
    );
```

4.7 例行维护表

为了保证数据库的有效运行，数据库必须在插入/删除操作后，基于客户场景，定期做VACUUM FULL和ANALYZE，更新统计信息，以便获得更优的性能。

相关概念

使用VACUUM、VACUUM FULL和ANALYZE命令定期对每个表进行维护，主要有以下原因：

- VACUUM FULL可回收已更新或已删除的数据所占据的磁盘空间，同时将小数据文件合并。
- VACUUM对每个表维护了一个可视化映射来跟踪包含对别的活动事务可见的数组的页。一个普通的索引扫描首先通过可视化映射来获取对应的数组，来检查是否对当前事务可见。若无法获取，再通过堆数组抓取的方式来检查。因此更新表的可视化映射，可加速唯一索引扫描。

- VACUUM可避免执行的事务数超过数据库阈值时，事务ID重叠造成的原有数据丢失。
- ANALYZE可收集与数据库中表内容相关的统计信息。统计结果存储在系统表PG_STATISTIC中。查询优化器会使用这些统计数据，生成最有效的执行计划。

操作步骤

步骤1 使用VACUUM或VACUUM FULL命令，进行磁盘空间回收。

- **VACUUM**：

  对表执行VACUUM操作

  ```
  VACUUM customer;
  ```

  可以与数据库操作命令并行运行。（执行期间，可正常使用的语句：SELECT、INSERT、UPDATE和DELETE。不可正常使用的语句：ALTER TABLE）。

  对表分区执行VACUUM操作

  ```
  VACUUM customer_par PARTITION ( P1 );
  ```

- **VACUUM FULL**：

  ```
  VACUUM FULL customer;
  ```

  需要向正在执行的表增加排他锁，且需要停止其他所有数据库操作。

  在进行磁盘空间回收时，用户可以使用如下命令查询集群中最早事务对应session，再根据需要结束最早执行的长事务，从而更加高效的利用磁盘空间。

  a. 使用命令从GTM上查询oldestxmin

  ```
  select * from pgxc_gtm_snapshot_status();
  ```

  b. 从CN上查询对应的session的pid，此处xmin为上一步的oldestxmin。

  ```
  select * from pgxc_running_xacts() where xmin=1400202010;
  ```

步骤2 使用ANALYZE语句更新统计信息。

```
ANALYZE customer;
```

使用ANALYZE VERBOSE语句更新统计信息，并输出表的相关信息。

```
ANALYZE VERBOSE customer;
```

也可以同时执行VACUUM ANALYZE命令进行查询优化。

```
VACUUM ANALYZE customer;
```

说明

VACUUM和ANALYZE会导致I/O流量的大幅增加，这可能会影响其他活动会话的性能。因此，建议通过“vacuum_cost_delay”参数设置。

步骤3 删除表

```
DROP TABLE customer;
DROP TABLE customer_par;
DROP TABLE part;
```

----结束

维护建议

- 定期对部分大表做VACUUM FULL，在性能下降后为全库做VACUUM FULL，目前暂定每月做一次VACUUM FULL。
- 定期对系统表做VACUUM FULL，主要是PG_ATTRIBUTE。
- 启用系统自动清理进程（AUTOVACUUM）自动执行VACUUM和ANALYZE，回收被标识为删除状态的记录空间，并更新表的统计数据。

4.8 例行重建索引

背景信息

数据库经过多次删除操作后，索引页面上的索引键将被删除，造成索引膨胀。例行重建索引，可有效的提高查询效率。

数据库支持的索引类型包含B-tree索引、GIN索引和PSORT索引。

- 对于B-tree索引，例行重建索引可有效的提高查询效率。
  - 如果数据发生大量删除后，索引页面上的索引键将被删除，导致索引页面数量的减少，造成索引膨胀。重建索引可回收浪费的空间。
  - 新建的索引中逻辑结构相邻的页面，通常在物理结构中也是相邻的，所以一个新建的索引比更新了多次的索引访问速度要快。
- 对于非B-tree索引，不建议例行重建。

重建索引

重建索引有以下两种方式：

- 先删除索引（DROP INDEX），再创建索引（CREATE INDEX）。

  在删除索引过程中，会在父表上增加一个短暂的排他锁，阻止相关读写操作。在创建索引过程中，会锁住写操作但是不会锁住读操作，此时读操作只能使用顺序扫描。
- 使用REINDEX语句重建索引。
  - 使用REINDEX TABLE语句重建索引，会在重建过程中增加排他锁，阻止相关读写操作。
  - 使用REINDEX INTERNAL TABLE语句重建desc表（包括）的索引，会在重建过程中增加排他锁，阻止相关读写操作。

操作步骤

假定在导入表“areaS”上的“area_id”字段上存在普通索引“areaS_idx”。重建索引有以下两种方式：

- 先删除索引（DROP INDEX），再创建索引（CREATE INDEX）
  a. 删除索引。
     ```
     DROP INDEX areaS_idx;
     ```
  b. 创建索引。
     ```
     CREATE INDEX areaS_idx ON areaS (area_id);
     ```
- 使用REINDEX重建索引。
  - 使用REINDEX TABLE语句重建索引。
    ```
    REINDEX TABLE areaS;
    ```
  - 使用REINDEX INTERNAL TABLE重建desc表（包括）的索引。
    ```
    REINDEX INTERNAL TABLE areaS;
    ```

4.9 SQL 调优关键参数调整

本节将介绍影响GaussDB(DWS) SQL调优性能的关键CN配置参数，配置方法参见**设置GUC参数**。

表 4-2 CN 配置参数

参数/参考值	描述
enable_nestloop=on	控制查询优化器对嵌套循环连接（Nest Loop Join）类型的使用。当设置为“on”后，优化器优先使用Nest Loop Join；当设置为“off”后，优化器在存在其他方法时将优先选择其他方法。 **说明** 如果只需要在当前数据库连接（即当前Session）中临时更改该参数值，则只需要在SQL语句中执行如下命令： `SET enable_nestloop to off;` 此参数默认设置为“on”，但实际调优中应根据情况选择是否关闭。一般情况下，在三种join方式（Nested Loop、Merge Join和Hash Join）里，Nested Loop性能较差，实际调优中可以选择关闭。
enable_bitmapscan=on	控制查询优化器对位图扫描规划类型的使用。设置为“on”，表示使用；设置为“off”，表示不使用。 **说明** 如果只需要在当前数据库连接（即当前Session）中临时更改该参数值，则只需要在SQL语句中执行命令如下命令： `SET enable_bitmapscan to off;` bitmapscan扫描方式适用于“where a > 1 and b > 1”且a列和b列都有索引这种查询条件，但有时其性能不如indexscan。因此，现场调优如发现查询性能较差且计划中有bitmapscan算子，可以关闭bitmapscan，看性能是否有提升。
enable_fast_query_shipping=on	控制查询优化器是否使用分布式框架，执行快速执行计划。设置为“on”，表示执行计划在CN和DN上各自生成；设置为“off”，表示使用分布式框架，即执行计划在CN上生成，然后发送到DN中执行。 **说明** 如果只需要在当前数据库连接（即当前Session）中临时更改该参数值，则只需要在SQL语句中执行如下命令： `SET enable_fast_query_shipping to off;`
enable_hashagg=on	控制优化器对Hash聚集规划类型的使用。
enable_hashjoin=on	控制优化器对Hash连接规划类型的使用。
enable_mergejoin=on	控制优化器对融合连接规划类型的使用。

参数/参考值	描述
enable_indexscan=on	控制优化器对索引扫描规划类型的使用。
enable_indexonlyscan=on	控制优化器对仅索引扫描规划类型的使用。
enable_seqscan=on	控制优化器对顺序扫描规划类型的使用。完全消除顺序扫描是不可能的，但是关闭这个变量会让优化器在存在其他方法的时候优先选择其他方法。
enable_sort=on	控制优化器使用的排序步骤。该设置不可能完全消除明确的排序，但是关闭这个变量可以让优化器在存在其他方法的时候优先选择其他方法。
enable_broadcast=on	控制查询优化器对于broadcast广播模式数据传输的使用。此方式网络传输数据量较大，因此当网络传输节点（Stream）实际数据量较大而估算不准时，可以将该参数设置为off，看性能是否有提升。
enable_redistribute=on （该参数仅8.2.1.300及以上集群版本支持）	控制查询优化器对于local redistribute和split redistribute重分布模式数据传输的使用。此参数与enable_broadcast是对应关系。优化器可能会对local broadcast和split broadcast代价估计偏大，从而选择了local redistribute或者split redistribute重分布的计划。这有可能导致性能劣化。因此当网络传输节点（Stream）实际数据量较小时，可以将该参数设置为off，让优化器优先选择broadcast广播模式，看性能是否有提升。
rewrite_rule	控制优化器是否启用特定组合的重写规则。

4.10 配置 SMP

4.10.1 SMP 适用场景与限制

背景信息

SMP特性通过算子并行来提升性能，同时会占用更多的系统资源，包括CPU、内存、网络、I/O等等。本质上SMP是一种以资源换取时间的方式，在合适的场景以及资源充足的情况下，能够起到较好的性能提升效果；但是如果在不合适的场景下，或者资源不足的情况下，反而可能引起性能的劣化。同时，生成SMP需要考虑更多的候选计划，将会导致生成时间较长，相比串行场景也会引起性能的劣化。

适用场景

- 支持并行的算子

  计划中存在以下算子支持并行：

  a. Scan：支持行存普通表和行存分区表顺序扫描、列存普通表和列存分区表顺序扫描、HDFS内外表顺序扫描；支持GDS数据导入的外表扫描并行。以上均不支持复制表。

b. Join：HashJoin、NestLoop

c. Agg：HashAgg、SortAgg、PlainAgg、WindowAgg(只支持partition by，不支持order by)

d. Stream：Redistribute、Broadcast

e. 其他：Result、Subqueryscan、Unique、Material、Setop、Append、VectoRow、RowToVec

- SMP特有算子

  为了实现并行，新增了并行线程间的数据交换Stream算子供SMP特性使用。以下新增的算子可以看做Stream算子的子类：

  a. Local Gather：实现DN内部并行线程的数据汇总

  b. Local Redistribute：在DN内部各线程之间，按照分布键进行数据重分布

  c. Local Broadcast：将数据广播到DN内部的每个线程

  d. Local RoundRobin：在DN内部各线程之间实现数据轮询分发

  e. Split Redistribute：在集群跨DN的并行线程之间实现数据重分布

  f. Split Broadcast：将数据广播到集群所有DN的并行线程

  上述新增算子可以分为Local与非Local两类，Local类算子实现了DN内部并行线程间的数据交换，而非Local类算子实现了跨DN的并行线程间的数据交换。

- 示例说明

  以TPCH Q1的并行计划为例：

```
 id |                                operation                                 |
----+--------------------------------------------------------------------------+
  1 | ->  Row Adapter                                                          |
  2 |    ->  Vector Streaming (type: GATHER)                                   |
  3 |       ->  Vector Sort                                                    |
  4 |          ->  Vector Streaming(type: LOCAL GATHER dop: 1/4)               |
  5 |             ->  Vector Hash Aggregate                                    |
  6 |                ->  Vector Streaming(type: SPLIT REDISTRIBUTE dop: 4/4)   |
  7 |                   ->  Vector Hash Aggregate                              |
  8 |                      ->  Vector Append(9, 10)                            |
  9 |                         ->  Dfs Scan on lineitem                         |
 10 |                         ->  Vector Adapter                               |
 11 |                            ->  Seq Scan on pg_delta_1423863972 lineitem  |
(11 rows)
```

  在这个计划中，实现了Hdfs Scan以及HashAgg算子的并行，并且新增了Local Gather和Split Redistribute数据交换算子。

  其中6号算子为Split Redistribute算子，上面标有的“dop: 4/4”表明Split Redistribute的发送端和接收端线程的并行度均为4。4号算子为Local Gather，上面标有“dop: 1/4”，该算子的发送端线程并行度为4，而接收端线程并行度为1，即下层的5号Hash Aggregate算子按照4并行度执行，而上层的1~3号算子按照串行执行，4号算子实现了DN内并行线程的数据汇总。

  通过计划Stream算子上标明的dop信息即可看出各个算子的并行情况。

非适用场景

1. 生成计划时间占比很高的短查询场景。
2. 不支持CN上的算子并行。
3. 不支持不能下推的查询并行执行。
4. 不支持子查询subplan的并行，以及包含子查询的算子并行。

4.10.2 资源对 SMP 性能的影响

SMP架构是一种利用富余资源来换取时间的方案，计划并行之后必定会引起资源消耗的增加，包括CPU、内存、I/O和网络带宽等资源的消耗都会出现明显的增长，而且随着并行度的增大，资源消耗也随之增大。当上述资源成为瓶颈的情况下，SMP无法提升性能，反而可能导致集群整体性能的劣化。SMP支持自适应特性，该特性会根据当前资源和查询特征，动态选取最优的并行度。下面对各种资源对SMP性能的影响情况分别进行说明：

- **CPU资源**

  在一般客户场景中，系统CPU利用率不高的情况下，利用SMP并行架构能够更充分地利用系统CPU资源，提升系统性能。但当数据库服务器的CPU核数较少，CPU利用率已经比较高的情况下，如果打开SMP并行，不仅性能提升不明显，反而可能因为多线程间的资源竞争而导致性能劣化。

- **内存资源**

  查询并行后会导致内存使用量的增长，但每个算子使用内存上限仍受到work_mem等参数的限制。假设work_mem为4GB，并行度为2，那么每个并行线程所分到的内存上限为2GB。在work_mem较小或者系统内存不充裕的情况下，使用SMP并行后，可能出现数据下盘，导致查询性能劣化的问题。

- **网络带宽资源**

  为了实现查询并行执行，会新增并行线程间的数据交换算子。对于Local类Stream算子，所需要进行数据交换的线程在同一个DN内，通过内存交换，不会增加网络负担。而非Local类算子，需要通过网络进行数据交换，因此会加重网络负担。当网络资源成为瓶颈的情况下，并行可能会导致一定程度的劣化。

- **I/O资源**

  要实现并行扫描必定会增加I/O的资源消耗，因此只有在I/O资源充足的情况下，并行扫描才能够提高扫描性能。

4.10.3 其他因素对 SMP 性能的影响

除了资源因素外，还有一些因素也会对SMP并行性能造成影响。例如分区表中分区数据不均，以及系统并发度等因素。

- **数据倾斜对SMP性能的影响**

  当数据中存在严重数据倾斜时，并行效果较差。例如某表join列上某个值的数据量远大于其他值，开启并行后，根据join列的值对该表数据做hash重分布，使得某个并行线程的数据量远多于其他线程，造成长尾问题，导致并行后效果差。

- **系统并发度对SMP性能的影响**

  SMP特性会增加资源的使用，而在高并发场景下资源剩余较少。所以，如果在高并发场景下，开启SMP并行，会导致各查询之间严重的资源竞争问题。一旦出现了资源竞争的现象，无论是CPU、I/O、内存或者网络资源，都会导致整体性能的下降。因此在高并发场景下，开启SMP往往不能达到性能提升的效果，甚至可能引起性能劣化。

4.10.4 SMP 相关参数配置建议

如果要打开SMP自适应功能，要设置query_dop=0，需同步调整以下相关参数值，以获取更佳的dop选择：

- comm_usable_memory

当系统内存较大时，max_process_memory设置较大，可适当调大该值，建议设置为max_process_memory的5%，默认值为4GB。

- comm_max_stream

  设置建议值为：comm_max_stream=Min(dop_limit * dop_limit * 20 * 2, max_process_memory(字节数) * 0.025 / 总DN数 / 260)，且该值在comm_max_stream取值范围内。

- max_connections

  设置建议值为：max_connections=dop_limit * 20 * 6 + 24，且该值在max_connections取值范围内。

注意

公式中的dop_limit为集群中每个DN对应的CPU数，计算公式为：dop_limit = 单机器的CPU逻辑核数 / 单机器的DN数。

4.10.5 SMP 手动调优建议

如果想手动进行SMP调优，需要熟练掌握**SMP相关参数配置建议**，并了解本节内容。

使用限制

系统的CPU、内存、I/O和网络带宽等资源充足。SMP架构是一种利用富余资源来换取时间的方案，计划并行之后必定会引起资源消耗的增加，当上述资源成为瓶颈的情况下，SMP无法提升性能，反而可能导致性能的劣化。同时，SMP计划的生成时间较串行要长。因此，在短查询为主的TP类业务中，或者出现资源瓶颈的情况下，建议关闭SMP，即设置query_dop=1。

配置步骤

1. 观察当前系统负载情况，如果系统资源充足（资源利用率小于50%），执行步骤2；否则退出。
2. 设置query_dop=1，利用explain打出执行计划，观察计划是否符合**SMP适用场景与限制**小节中的适用场景。如果符合，进入下一步。
3. 设置query_dop=-value，在考虑资源情况和计划特征基础上，限制dop选取的范围为[1,value]。
4. 设置query_dop=value，不考虑资源情况和计划特征，强制选取dop为1或value。
5. 在符合条件的查询语句执行前设置合适的query_dop值，在语句执行结束后关闭query_dop。例如，

```
SET query_dop = 0;
SELECT COUNT(*) FROM t1 GROUP BY a;
......
SET query_dop = 1;
```

说明

- 资源许可的情况下，并行度越高，性能提升效果越好。
- SMP并行度支持会话级设置，推荐客户在执行符合要求的查询前，打开smp，执行结束后，关闭smp。以免在业务峰值时，对业务造成冲击。
- SMP自适应（query_dop<=0）依赖资源管理，如果资源管理禁用（use_workload_manager为off），那么只会产生1或2并行度的计划。

4.11 查询最耗性能的 SQL

系统中有些SQL语句运行了很长时间还没有结束，这些语句会消耗很多的系统性能，请根据本章内容查询长时间运行的SQL语句。

操作步骤

步骤1 查询系统中长时间运行的查询语句。

```
SELECT current_timestamp - query_start AS runtime, datname, usename, query FROM pg_stat_activity
where state != 'idle' ORDER BY 1 desc;
```

查询后会按执行时间从长到短顺序返回查询语句列表，第一条结果就是当前系统中执行时间最长的查询语句。返回结果中包含了系统调用的SQL语句和用户执行SQL语句，请根据实际找到用户执行时间长的语句。

若当前系统较为繁忙，可以通过限制current_timestamp - query_start大于某一阈值来查看执行时间超过此阈值的查询语句。

```
SELECT query FROM pg_stat_activity WHERE current_timestamp - query_start > interval '1 days';
```

步骤2 设置参数track_activities为on。

```
SET track_activities = on;
```

当此参数为on时，数据库系统才会收集当前活动查询的运行信息。

步骤3 查看正在运行的查询语句。

以查看视图pg_stat_activity为例：

```
SELECT datname, usename, state FROM pg_stat_activity;
 datname  | usename | state  |
----------+---------+--------+
 postgres |   omm   | idle   |
 postgres |   omm   | active |
(2 rows)
```

如果state字段显示为idle，则表明此连接处于空闲，等待用户输入命令。

如果仅需要查看非空闲的查询语句，则使用如下命令查看：

```
SELECT datname, usename, state FROM pg_stat_activity WHERE state != 'idle';
```

步骤4 分析长时间运行的查询语句状态。

- 若查询语句处于正常状态，则等待其执行完毕。
- 若查询语句阻塞，则通过如下命令查看当前处于阻塞状态的查询语句：
  ```
  SELECT datname, usename, state, query FROM pg_stat_activity WHERE waiting = true;
  ```
  查询结果中包含了当前被阻塞的查询语句，该查询语句所请求的锁资源可能被其他会话持有，正在等待持有会话释放锁资源。

  说明

  只有当查询阻塞在系统内部锁资源时，waiting字段才显示为true。尽管等待锁资源是数据库系统最常见的阻塞行为，但是在某些场景下查询也会阻塞在等待其他系统资源上，例如写文件、定时器等。但是这种情况的查询阻塞，不会在视图pg_stat_activity中体现。

----结束

4.12 分析作业是否被阻塞

数据库系统运行时，在某些业务场景下查询语句会被阻塞，导致语句运行时间过长，可以强制结束有问题的会话。

操作步骤

步骤1 查看阻塞的查询语句及阻塞查询的表、模式信息。

```
SELECT w.query as waiting_query,
w.pid as w_pid,
w.usename as w_user,
l.query as locking_query,
l.pid as l_pid,
l.usename as l_user,
t.schemaname || '.' || t.relname as tablename
from pg_stat_activity w join pg_locks l1 on w.pid = l1.pid
and not l1.granted join pg_locks l2 on l1.relation = l2.relation
and l2.granted join pg_stat_activity l on l2.pid = l.pid join pg_stat_user_tables t on l1.relation = t.relid
where w.waiting;
```

该查询返回线程ID、用户信息、查询状态，以及导致阻塞的表、模式信息。

步骤2 使用如下命令结束相应的会话。其中，139834762094352为线程ID。

```
SELECT PG_TERMINATE_BACKEND(139834762094352);
```

显示类似如下信息，表示结束会话成功。

```
 PG_TERMINATE_BACKEND
----------------------
 t
(1 row)
```

显示类似如下信息，表示用户正在尝试结束当前会话，此时仅会重连会话，而不是结束会话。

```
FATAL:  terminating connection due to administrator command
FATAL:  terminating connection due to administrator command
The connection to the server was lost. Attempting reset: Succeeded.
```

说明

gsql客户端使用PG_TERMINATE_BACKEND函数终止本会话后台线程时，客户端不会退出而是自动重连。

----结束

5 实际调优案例

5.1 案例：选择合适的分布列

分布列用于将数据分布到不同的节点上，划分均衡可以避免数据倾斜。

在进行关联查询时，尽量选择查询中的关联条件作为分布键。当关联条件作为分布键时，相关数据都分布在DN本地，将减少DN之间的数据流动代价，提升查询速度。

优化前

将a作为t1和t2的分布列，表定义如下：

```
CREATE TABLE t1 (a int, b int) DISTRIBUTE BY HASH (a);
CREATE TABLE t2 (a int, b int) DISTRIBUTE BY HASH (a);
```

执行如下查询：

```
SELECT * FROM t1, t2 WHERE t1.a = t2.b;
```

则执行计划存在“Streaming(type: REDISTRIBUTE)”，即DN根据选定的列把数据重分布到所有的DN，这将导致DN之间存在较大通信数据量，如图5-1所示。

图 5-1 选择合适的分布列案例（一）

```
EXPLAIN PERFORMANCE SELECT * FROM t1, t2 WHERE t1.a = t2.b;
                                              QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
 id |              operation                 |      A-time      | A-rows | E-rows | E-distinct |  Peak Memory   | E-memory | A-width | E-width | E-costs
----+----------------------------------------+------------------+--------+--------+------------+----------------+----------+---------+---------+---------
  1 | ->  Streaming (type: GATHER)           | 8.760            |      0 |     30 |            | 24KB           |          |         |      16 | 37.96
  2 |    ->  Hash Join (3,5)                 | [0.396, 0.428]   |      0 |     30 |            | [8KB, 8KB]     | 1MB      |         |      16 | 29.96
  3 |       ->  Streaming(type: REDISTRIBUTE) | [0,0]            |      0 |     30 | 10         | [0, 0]         | 2MB      |         |       8 | 15.49
  4 |          ->  Seq Scan on dbadmin.t2    | [0.001, 0.002]   |      0 |     30 |            | [32KB, 32KB]   | 1MB      |         |       8 | 14.14
  5 |       ->  Hash                         | [0.003, 0.003]   |      0 |     29 | 14         | [264KB, 264KB] | 16MB     |         |       8 | 14.14
  6 |          ->  Seq Scan on dbadmin.t1    | [0.001, 0.002]   |      0 |     30 |            | [32KB, 32KB]   | 1MB      |         |       8 | 14.14

 Predicate Information (identified by plan id)
 ---------------------------------------------
   2 --Hash Join (3,5)
         Hash Cond: (t2.b = t1.a)
```

优化后

将查询中的关联条件作为分布键，执行下列语句修改b作为t2的分布列：

```
ALTER TABLE t2 DISTRIBUTE BY HASH (b);
```

将表t2的分布列改为b列之后，执行计划将不再包含“Streaming(type: REDISTRIBUTE)”，减少了DN之间存在的通信数据量的同时，执行时间也从8.7毫秒降低至2.7毫秒，从而提升查询性能，如图5-2所示。

图 5-2 选择合适的分布列案例（二）

```
 EXPLAIN PERFORMANCE SELECT * FROM t1, t2 WHERE t1.a = t2.b;
                                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 id |              operation             |      A-time      | A-rows | E-rows | E-distinct | Peak Memory  | E-memory | A-width | E-width | E-costs
----+------------------------------------+------------------+--------+--------+------------+--------------+----------+---------+---------+---------
  1 | ->  Streaming (type: GATHER)       | 2.727            |      0 |     30 |            | 24KB         |          |         |      16 | 36.59
  2 |    ->  Hash Join (3,4)             | [0.006, 0.007]   |      0 |     30 |            | [8KB, 8KB]   | 1MB      |         |      16 | 28.59
  3 |       ->  Seq Scan on dbadmin.t1   | [0.001, 0.002]   |      0 |     30 | 14         | [16KB, 16KB] | 1MB      |         |       8 | 14.14
  4 |       ->  Hash                     | [0,0]            |      0 |     29 | 14         | [0, 0]       | 16MB     |         |       8 | 14.14
  5 |          ->  Seq Scan on dbadmin.t2 | [0,0]           |      0 |     30 |            | [0, 0]       | 1MB      |         |       8 | 14.14

 Predicate Information (identified by plan id)
 ---------------------------------------------
   2 --Hash Join (3,4)
         Hash Cond: (t1.a = t2.b)
```

5.2 案例：建立合适的索引

创建合适的索引可以加速对表中数据行的检索。索引占用磁盘空间，并且降低添加、删除和更新行的速度。如果需要非常频繁地更新数据或磁盘空间有限，则需要限制索引的数量。在表较大时再建立索引，表中的数据越多，索引的优越性越明显。建议仅在匹配如下某条原则时创建索引：

- 需要经常执行查询的字段。
- 对于存在多字段连接的查询，建议在连接条件字段上建立组合索引。例如select * from t1 join t2 on t1.a=t2.a and t1.b=t2.b，可以在t1表上的a，b字段上建立组合索引。
- where子句过滤条件的字段（尤其是范围条件）。
- 经常出现在order by、group by和distinct后的字段。

优化前

列存分区表orders表定义如下：

```
                                                 pg_get_tabledef
-------------------------------------------------------------------------------------------------------------------------------------
 SET search_path = dbadmin;                                                                                                         +
 CREATE  TABLE orders (                                                                                                              +
         o_orderkey bigint NOT NULL,                                                                                                 +
         o_custkey bigint NOT NULL,                                                                                                  +
         o_orderstatus character(1) NOT NULL,                                                                                        +
         o_totalprice numeric(15,2) NOT NULL,                                                                                        +
         o_orderdate timestamp(0) without time zone NOT NULL,                                                                        +
         o_orderpriority character(15) NOT NULL,                                                                                     +
         o_clerk character(15) NOT NULL,                                                                                             +
         o_shippriority bigint NOT NULL,                                                                                             +
         o_comment character varying(79) NOT NULL                                                                                    +
 )                                                                                                                                   +
 WITH (orientation=column, compression=low, colversion=2.0, enable_delta=false)                                                      +
 DISTRIBUTE BY HASH(o_orderkey)                                                                                                      +
 TO GROUP group_version1                                                                                                             +
 PARTITION BY RANGE (o_orderdate)                                                                                                    +
 (                                                                                                                                   +
          PARTITION o_orderdate_1 VALUES LESS THAN ('1993-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default,+
          PARTITION o_orderdate_2 VALUES LESS THAN ('1994-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default,+
          PARTITION o_orderdate_3 VALUES LESS THAN ('1995-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default,+
          PARTITION o_orderdate_4 VALUES LESS THAN ('1996-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default,+
          PARTITION o_orderdate_5 VALUES LESS THAN ('1997-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default,+
          PARTITION o_orderdate_6 VALUES LESS THAN ('1998-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default,+
          PARTITION o_orderdate_7 VALUES LESS THAN ('1999-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default +
 )                                                                                                                                   +
 ENABLE ROW MOVEMENT;
(1 row)
```

执行SQL语句查询没有建立索引情况下的执行计划，发现执行时间为48毫秒。

```
EXPLAIN PERFORMANCE SELECT * FROM orders WHERE o_custkey = '1106459';
```

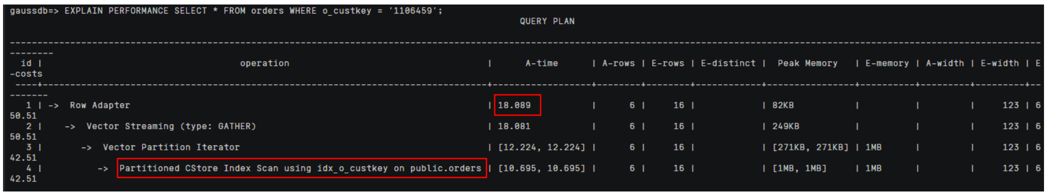
```
gaussdb=>  EXPLAIN PERFORMANCE SELECT * FROM orders WHERE o_custkey = '1106459';
                                                   QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
 id |                      operation                       |      A-time      | A-rows | E-rows | E-distinct | Peak Memory  | E-memory | A-width | E-width | E-costs
----+------------------------------------------------------+------------------+--------+--------+------------+--------------+----------+---------+---------+----------
  1 | ->  Row Adapter                                      | 48.500           |      6 |     16 |            | 82KB         |          |         |     123 | 94931.00
  2 |    ->  Vector Streaming (type: GATHER)               | 48.491           |      6 |     16 |            | 249KB        |          |         |     123 | 94931.00
  3 |       ->  Vector Partition Iterator                  | [45.479, 45.479] |      6 |     16 |            | [17KB, 17KB] | 1MB      |         |     123 | 94923.00
  4 |          ->  Partitioned CStore Scan on public.orders | [45.157, 45.157] |      6 |     16 |            | [1MB, 1MB]   | 1MB      |         |     123 | 94923.00
```

优化后

where子句过滤条件的字段是o_custkey，在o_custkey字段上添加一个索引：

```
CREATE INDEX idx_o_custkey ON orders (o_custkey) LOCAL;
```

执行SQL语句查询建立索引后的执行计划，发现执行时间为18毫秒。



```
gaussdb=> EXPLAIN PERFORMANCE SELECT * FROM orders WHERE o_custkey = '1106459';
                                   QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
--------
 id |                        operation                        |      A-time      | A-rows | E-rows | E-distinct |  Peak Memory     | E-memory | A-width | E-width | E
-costs
----+---------------------------------------------------------+------------------+--------+--------+------------+------------------+----------+---------+---------+--
-------
  1 | -> Row Adapter                                          | 18.089           |      6 |     16 |            | 82KB             |          |         |     123 | 6
50.51
  2 |   -> Vector Streaming (type: GATHER)                    | 18.081           |      6 |     16 |            | 249KB            |          |         |     123 | 6
50.51
  3 |     -> Vector Partition Iterator                        | [12.224, 12.224] |      6 |     16 |            | [271KB, 271KB]   | 1MB      |         |     123 | 6
42.51
  4 |       -> Partitioned CStore Index Scan using idx_o_custkey on public.orders | [10.695, 10.695] |      6 |     16 |            | [1MB, 1MB]       | 1MB      |         |     123 | 6
42.51
```

5.3 案例：增加 JOIN 列非空条件

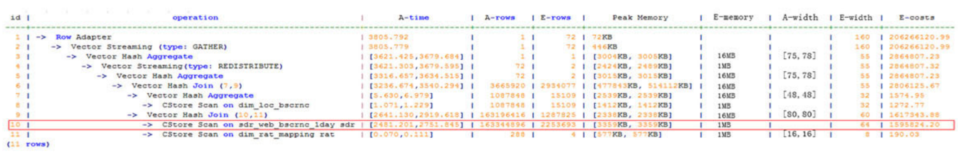
若Join列上的NULL值较多，可以加上is not null过滤条件，以实现数据的提前过滤，提高Join效率。

优化前

```
SELECT
 *
FROM
( ( SELECT
  STARTTIME STTIME,
  SUM(NVL(PAGE_DELAY_MSEL,0)) PAGE_DELAY_MSEL,
  SUM(NVL(PAGE_SUCCEED_TIMES,0)) PAGE_SUCCEED_TIMES,
  SUM(NVL(FST_PAGE_REQ_NUM,0)) FST_PAGE_REQ_NUM,
  SUM(NVL(PAGE_AVG_SIZE,0)) PAGE_AVG_SIZE,
  SUM(NVL(FST_PAGE_ACK_NUM,0)) FST_PAGE_ACK_NUM,
  SUM(NVL(DATATRANS_DW_DURATION,0)) DATATRANS_DW_DURATION,
  SUM(NVL(PAGE_SR_DELAY_MSEL,0)) PAGE_SR_DELAY_MSEL
 FROM
  PS.SDR_WEB_BSCRNC_1DAY SDR
  INNER JOIN (SELECT
      BSCRNC_ID,
      BSCRNC_NAME,
      ACCESS_TYPE,
      ACCESS_TYPE_ID
     FROM
      nethouse.DIM_LOC_BSCRNC
     GROUP BY
      BSCRNC_ID,
      BSCRNC_NAME,
      ACCESS_TYPE,
      ACCESS_TYPE_ID) DIM
  ON SDR.BSCRNC_ID = DIM.BSCRNC_ID
  AND DIM.ACCESS_TYPE_ID IN (0,1,2)
  INNER JOIN nethouse.DIM_RAT_MAPPING RAT
  ON (RAT.RAT = SDR.RAT)
 WHERE
  ( (STARTTIME >= 1461340800
  AND STARTTIME < 1461427200) )
  AND RAT.ACCESS_TYPE_ID IN (0,1,2)
 GROUP BY STTIME ) ) ;
```

执行计划如图5-3所示。

图 5-3 增加 JOIN 列非空条件（一）



```
 id |                     operation                      |        A-time         |   A-rows  |  E-rows  |      Peak Memory        | E-memory | A-width  | E-width |    E-costs
----+----------------------------------------------------+-----------------------+-----------+----------+-------------------------+----------+----------+---------+--------------
  1 | -> Row Adapter                                     | 3805.792              |         1 |       72 | 72KB                    |          |          |     160 | 206266120.99
  2 |   -> Vector Streaming (type: GATHER)               | 3805.779              |         1 |       72 | 446KB                   |          |          |     160 | 206266120.99
  3 |     -> Vector Hash Aggregate                       | [3621.425,3679.684]   |         1 |        1 | [3004KB, 3005KB]        | 16MB     | [75, 78] |      55 | 2864807.23
  4 |       -> Vector Streaming(type: REDISTRIBUTE)      | [3621.303,3679.595]   |        72 |        2 | [2424KB, 2489KB]        | 1MB      |          |      55 | 2864807.32
  5 |         -> Vector Hash Aggregate                   | [3316.657,3634.515]   |        72 |        2 | [3015KB, 3015KB]        | 16MB     | [75, 78] |      55 | 2864807.23
  6 |           -> Vector Hash Join (7,9)                | [3236.674,3540.294]   |   3665920 |  2934077 | [477843KB, 514112KB]    | 16MB     |          |      55 | 2806125.67
  7 |             -> Vector Hash Aggregate               | [5.630,6.979]         |   1087848 |    15109 | [2539KB, 2539KB]        | 16MB     | [48, 48] |      32 | 1574.95
  8 |               -> CStore Scan on dim_loc_bscrnc     | [1.071,1.229]         |   1087848 |    15109 | [1412KB, 1412KB]        | 1MB      |          |      32 | 1272.77
  9 |             -> Vector Hash Join (10,11)            | [2641.130,2919.618]   | 163196416 |  1287825 | [2338KB, 2338KB]        | 16MB     | [80, 80] |      60 | 1617343.88
 10 |               -> CStore Scan on sdr_web_bscrnc_1day sdr | [2481.201,2751.845] | 163344896 | 2253693 | [3359KB, 3359KB]   | 1MB      |          |      64 | 1595824.20
 11 |               -> CStore Scan on dim_rat_mapping rat | [0.070,0.111]        |       288 |        4 | [577KB, 577KB]          | 1MB      | [16, 16] |       8 | 190.03
(11 rows)
```

优化后

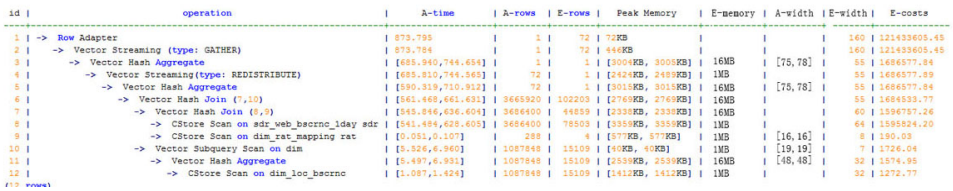
1. 分析执行计划图5-3可知，在顺序扫描阶段耗时较多。
2. 多表JOIN中，由于表PS.SDR_WEB_BSCRNC_1DAY的JOIN列“BSCRNC_ID”存在大量空值，JOIN性能差。

   建议在语句中手动添加JOIN列的非空判断，修改后的语句如下所示。

```
SELECT
 *
FROM
( ( SELECT
  STARTTIME STTIME,
  SUM(NVL(PAGE_DELAY_MSEL,0)) PAGE_DELAY_MSEL,
  SUM(NVL(PAGE_SUCCEED_TIMES,0)) PAGE_SUCCEED_TIMES,
  SUM(NVL(FST_PAGE_REQ_NUM,0)) FST_PAGE_REQ_NUM,
  SUM(NVL(PAGE_AVG_SIZE,0)) PAGE_AVG_SIZE,
  SUM(NVL(FST_PAGE_ACK_NUM,0)) FST_PAGE_ACK_NUM,
  SUM(NVL(DATATRANS_DW_DURATION,0)) DATATRANS_DW_DURATION,
  SUM(NVL(PAGE_SR_DELAY_MSEL,0)) PAGE_SR_DELAY_MSEL
 FROM
  PS.SDR_WEB_BSCRNC_1DAY SDR
  INNER JOIN (SELECT
      BSCRNC_ID,
      BSCRNC_NAME,
      ACCESS_TYPE,
      ACCESS_TYPE_ID
     FROM
      nethouse.DIM_LOC_BSCRNC
     GROUP BY
      BSCRNC_ID,
      BSCRNC_NAME,
      ACCESS_TYPE,
      ACCESS_TYPE_ID) DIM
  ON SDR.BSCRNC_ID = DIM.BSCRNC_ID
  AND DIM.ACCESS_TYPE_ID IN (0,1,2)
  INNER JOIN nethouse.DIM_RAT_MAPPING RAT
  ON (RAT.RAT = SDR.RAT)
 WHERE
  ( (STARTTIME >= 1461340800
  AND STARTTIME < 1461427200) )
  AND RAT.ACCESS_TYPE_ID IN (0,1,2)
  and SDR.BSCRNC_ID is not null
 GROUP BY
  STTIME ) ) A;
```

执行计划如图5-4所示。

图 5-4 增加 JOIN 列非空条件（二）



5.4 案例：使排序下推

在做场景性能测试时，发现某场景大部分时间是CN端在做window agg，占到总执行时间95%以上，系统资源不能充分利用。研究发现该场景的特点是：将两列分别求sum作为一个子查询，外层对两列的和再求和后做trunc，然后排序。可以尝试将语句改写为子查询，使排序下推。

优化前

表结构如下所示：

```
CREATE TABLE public.test(imsi int,L4_DW_THROUGHPUT int,L4_UL_THROUGHPUT int)
with (orientation = column) DISTRIBUTE BY hash(imsi);
```

查询语句如下所示：

```
SELECT COUNT(1) over() AS DATACNT,
IMSI AS IMSI_IMSI,
CAST(TRUNC(((SUM(L4_UL_THROUGHPUT) + SUM(L4_DW_THROUGHPUT))), 0) AS
DECIMAL(20)) AS TOTAL_VOLOME_KPIID
FROM public.test AS test
GROUP BY IMSI
order by TOTAL_VOLOME_KPIID DESC;
```

执行计划如下：

```
Row Adapter  (cost=10.70..10.70 rows=10 width=12)
  ->  Vector Sort  (cost=10.68..10.70 rows=10 width=12)
        Sort Key: ((trunc((((sum(l4_ul_throughput)) + (sum(l4_dw_throughput))))::numeric,
0))::numeric(20,0))
        ->  Vector WindowAgg  (cost=10.09..10.51 rows=10 width=12)
              ->  Vector Streaming (type: GATHER)  (cost=242.04..246.84 rows=240 width=12)
                    Node/s: All datanodes
                    ->  Vector Hash Aggregate  (cost=10.09..10.29 rows=10 width=12)
                          Group By Key: imsi
                          ->  CStore Scan on test  (cost=0.00..10.01 rows=10 width=12)
```

可以看到window agg和sort全部在CN端执行，耗时非常严重。

优化后

尝试将语句改写为子查询：

```
SELECT COUNT(1) over() AS DATACNT, IMSI_IMSI, TOTAL_VOLOME_KPIID
FROM (SELECT IMSI AS IMSI_IMSI,
CAST(TRUNC(((SUM(L4_UL_THROUGHPUT) + SUM(L4_DW_THROUGHPUT))),
0) AS DECIMAL(20)) AS TOTAL_VOLOME_KPIID
FROM public.test AS test
GROUP BY IMSI
ORDER BY TOTAL_VOLOME_KPIID DESC);
```

将trunc两列的和作为一个子查询，然后在子查询的外面做window agg，这样排序就可以下推了，执行计划如下：

```
Row Adapter  (cost=10.70..10.70 rows=10 width=24)
  ->  Vector WindowAgg  (cost=10.45..10.70 rows=10 width=24)
        ->  Vector Streaming (type: GATHER)  (cost=250.83..253.83 rows=240 width=24)
              Node/s: All datanodes
              ->  Vector Sort  (cost=10.45..10.48 rows=10 width=12)
                    Sort Key: ((trunc(((sum(test.l4_ul_throughput) + sum(test.l4_dw_throughput)))::numeric,
0))::numeric(20,0))
                    ->  Vector Hash Aggregate  (cost=10.09..10.29 rows=10 width=12)
                          Group By Key: test.imsi
                          ->  CStore Scan on test  (cost=0.00..10.01 rows=10 width=12)
```

经过SQL改写，性能由120s提升7s，优化效果明显。

5.5 案例：设置 cost_param 对查询性能优化

cost_param参数用于控制在特定的客户场景中，使用不同的估算方法使得估算值与真实值更接近。此参数可以同时控制多种方法，与某一方法对应的位做与操作，不为0表示该方法被选择。

场景一：优化前

cost_param的bit0(set cost_param=1)值为1时，表示对于求!=连接的选择率时选择一种改良机制，此方法在自连接（两个相同的表之间连接）的估算中更加准确。下面查询的例子是cost_param的bit0为1时的优化场景。V300R002C00版本开始已弃用cost_param & 1不为0时的路径，默认选择已优化的估算公式。

说明

选择率是两表join时，满足join条件的行数在join结果集中所占的比率。

表结构如下所示：

```
CREATE TABLE LINEITEM
(
L_ORDERKEY BIGINT NOT NULL
, L_PARTKEY BIGINT NOT NULL
, L_SUPPKEY BIGINT NOT NULL
, L_LINENUMBER BIGINT NOT NULL
, L_QUANTITY DECIMAL(15,2) NOT NULL
, L_EXTENDEDPRICE DECIMAL(15,2) NOT NULL
, L_DISCOUNT DECIMAL(15,2) NOT NULL
, L_TAX DECIMAL(15,2) NOT NULL
, L_RETURNFLAG CHAR(1) NOT NULL
, L_LINESTATUS CHAR(1) NOT NULL
, L_SHIPDATE DATE NOT NULL
, L_COMMITDATE DATE NOT NULL
, L_RECEIPTDATE DATE NOT NULL
, L_SHIPINSTRUCT CHAR(25) NOT NULL
, L_SHIPMODE CHAR(10) NOT NULL
, L_COMMENT VARCHAR(44) NOT NULL
) with (orientation = column, COMPRESSION = MIDDLE) distribute by hash(L_ORDERKEY);

CREATE TABLE ORDERS
(
O_ORDERKEY BIGINT NOT NULL
, O_CUSTKEY BIGINT NOT NULL
, O_ORDERSTATUS CHAR(1) NOT NULL
, O_TOTALPRICE DECIMAL(15,2) NOT NULL
, O_ORDERDATE DATE NOT NULL
, O_ORDERPRIORITY CHAR(15) NOT NULL
, O_CLERK CHAR(15) NOT NULL
, O_SHIPPRIORITY BIGINT NOT NULL
, O_COMMENT VARCHAR(79) NOT NULL
)with (orientation = column, COMPRESSION = MIDDLE) distribute by hash(O_ORDERKEY);
```

查询语句如下所示：

```
explain verbose select
count(*) as numwait
from
lineitem l1,
orders
where
o_orderkey = l1.l_orderkey
and o_orderstatus = 'F'
and l1.l_receiptdate > l1.l_commitdate
and not exists (
select
*
from
lineitem l3
where
l3.l_orderkey = l1.l_orderkey
and l3.l_suppkey <> l1.l_suppkey
and l3.l_receiptdate > l3.l_commitdate
)
```



```
order by
numwait desc;
```

执行计划如下图所示（verbose条件下，新增distinct列，受cost off/on控制，hashjoin行显示内外表的distinct估值，其他行为空）：

```
 id |                          operation                          | E-rows | E-distinct | E-width | E-costs
----+-------------------------------------------------------------+--------+------------+---------+---------
  1 | ->  Row Adapter                                             |      1 |            |       8 | 39.36
  2 |    ->  Vector Sort                                          |      1 |            |       8 | 39.36
  3 |       ->  Vector Aggregate                                  |      1 |            |       8 | 39.34
  4 |          ->  Vector Streaming (type: GATHER)                |      2 |            |       8 | 39.34
  5 |             ->  Vector Aggregate                            |      2 |            |       8 | 39.25
  6 |                ->  Vector Hash Anti Join (7, 10)            |      2 | 4, 5       |       0 | 39.24
  7 |                   ->  Vector Hash Join (8,9)                |      2 | 200, 1     |      16 | 26.12
  8 |                      ->  CStore Scan on public.lineitem 11  |      7 |            |      16 | 13.05
  9 |                      ->  CStore Scan on public.orders       |      1 |            |       8 | 13.05
 10 |                   ->  CStore Scan on public.lineitem 13     |      7 |            |      16 | 13.05
```

场景一：优化后

以上查询为lineitem表自连接的Anti Join，当使用cost_param的bit0为0时，估算Anti Join的行数与实际行数相差很大，导致查询性能下降。可以通过设置cost_param的bit0为1时，使Anti Join的行数估算更准确，从而提高查询性能。优化后的执行计划如下：

```
id |                          operation                          |  E-rows   | E-memory | E-width |  E-costs
---+-------------------------------------------------------------+-----------+----------+---------+-----------
 1 |  -> Row Adapter                                             |         1 |          |       0 | 9104892.37 9
 2 |    -> Vector Sort                                           |         1 |          |       0 | 9104892.37 9
 3 |      -> Vector Aggregate                                    |         1 |          |       0 | 9104892.35 8
 4 |        -> Vector Streaming (type: GATHER)                   |        48 |          |       0 | 9104892.35 8
 5 |          -> Vector Aggregate                                |        48 | 1MB      |       0 | 9104890.82 5
 6 |            -> Vector Hash Join (7.12)                       | 2526630903 | 929MB    |       0 | 8973295.45 4
 7 |              -> Vector Hash Anti Join (8. 10)               | 1999996587 | 3178MB   |       8 | 7198231.14
 8 |                -> Vector Partition Iterator                 | 1999996587 | 1MB      |      16 | 3000158.25
 9 |                  -> Partitioned CStore Scan on public.lineitem 11 | 1999996587 | 1MB | 16 | 3000158.25 1
10 |                -> Vector Partition Iterator                 | 1999996587 | 1MB      |      16 | 3000158.25
11 |                  -> Partitioned CStore Scan on public.lineitem 13 | 1999996587 | 1MB | 16 | 3000158.25
12 |              -> Vector Partition Iterator                   |  730839014 | 1MB      |       8 | 589611.00
13 |                -> Partitioned CStore Scan on public.orders  |  730839014 | 1MB      |       8 | 589611.00
```

场景二：优化前

当cost_param的bit1(set cost_param=2)为1时，表示求多个过滤条件（Filter）的选择率时，选择最小的作为总的选择率，而非两者乘积，此方法在过滤条件的列之间关联性较强时估算更加准确。下面查询的例子是cost_param的bit1为1时的优化场景。

表结构如下所示：

```
CREATE TABLE NATION
(
N_NATIONKEYINT NOT NULL
, N_NAMECHAR(25) NOT NULL
, N_REGIONKEYINT NOT NULL
, N_COMMENTVARCHAR(152)
) distribute by replication;
CREATE TABLE SUPPLIER
(
S_SUPPKEYBIGINT NOT NULL
, S_NAMECHAR(25) NOT NULL
, S_ADDRESSVARCHAR(40) NOT NULL
, S_NATIONKEYINT NOT NULL
, S_PHONECHAR(15) NOT NULL
, S_ACCTBALDECIMAL(15,2) NOT NULL
, S_COMMENTVARCHAR(101) NOT NULL
) distribute by hash(S_SUPPKEY);
CREATE TABLE PARTSUPP
```



```
(
PS_PARTKEYBIGINT NOT NULL
, PS_SUPPKEYBIGINT NOT NULL
, PS_AVAILQTYBIGINT NOT NULL
, PS_SUPPLYCOSTDECIMAL(15,2)NOT NULL
, PS_COMMENTVARCHAR(199) NOT NULL
)distribute by hash(PS_PARTKEY);
```

查询语句如下所示：

```
set cost_param=2;
explain verbose select
nation,
sum(amount) as sum_profit
from
(
select
n_name as nation,
l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
from
supplier,
lineitem,
partsupp,
nation
where
s_suppkey = l_suppkey
and ps_suppkey = l_suppkey
and ps_partkey = l_partkey
and s_nationkey = n_nationkey
) as profit
group by nation
order by nation;
```

当cost_param的bit1为0时，执行计划如下图所示：

```
 id |                           operation                           | E-rows | E-distinct | E-width | E-costs
----+---------------------------------------------------------------+--------+------------+---------+---------
  1 | ->  Sort                                                      |      1 |            |     208 | 61.52
  2 |    ->  HashAggregate                                          |      1 |            |     208 | 61.51
  3 |       ->  Streaming (type: GATHER)                            |      2 |            |     208 | 61.51
  4 |          ->  HashAggregate                                    |      2 |            |     208 | 61.36
  5 |             ->  Hash Join (6,7)                               |      2 | 20, 15     |     176 | 61.33
  6 |                ->  Seq Scan on public.nation                  |     40 |            |     108 | 20.20
  7 |                ->  Hash                                       |      2 |            |      76 | 41.04
  8 |                   ->  Hash Join (9,16)                        |      2 | 10, 13     |      76 | 41.04
  9 |                      ->  Streaming(type: REDISTRIBUTE)        |      2 |            |      88 | 27.73
 10 |                         ->  Hash Join (11,14)                 |      2 | 10, 13     |      88 | 27.62
 11 |                            ->  Streaming(type: REDISTRIBUTE)  |     20 |            |      70 | 14.19
 12 |                               ->  Row Adapter                 |     21 |            |      70 | 13.01
 13 |                                  ->  CStore Scan on public.lineitem |     20 |            |      70 | 13.01
 14 |                            ->  Hash                           |     21 |            |      34 | 13.13
 15 |                               ->  Seq Scan on public.partsupp |     20 |            |      34 | 13.13
 16 |                         ->  Hash                              |     21 |            |      12 | 13.13
 17 |                            ->  Seq Scan on public.supplier    |     20 |            |      12 | 13.13
```

场景二：优化后

在以上查询中，supplier、lineitem、partsupp三表做hashjoin的条件为(lineitem.l_suppkey = supplier.s_suppkey) AND (lineitem.l_partkey = partsupp.ps_partkey)，此hashjoin条件中存在两个过滤条件，这前一个过滤条件中的lineitem.l_suppkey和后一个过滤条件中的lineitem.l_partkey同为lineitem表的两列，这两列存在强相关的关联关系。在这种情况，估算hashjoin条件的选择率时，如果使用cost_param的bit1为0时，实际是将AND的两个过滤条件分别计算的2个选择率的值相乘来得到hashjoin条件的选择率，导致行数估算不准确，查询性能较差。所以需要将cost_param的bit1为1时，选择最小的选择率作为总的选择率估算行数比较准确，查询性能较好，优化后的计划如下图所示：


```
 id |                          operation                          | E-rows | E-distinct | E-width | E-costs
----+-------------------------------------------------------------+--------+------------+---------+---------
  1 | -> Sort                                                     |     10 |            |     208 | 64.42
  2 |    -> HashAggregate                                         |     10 |            |     208 | 64.23
  3 |       -> Streaming (type: GATHER)                           |     20 |            |     208 | 64.23
  4 |          -> HashAggregate                                   |     20 |            |     208 | 62.71
  5 |             -> Hash Join (6,7)                              |     20 | 20, 10     |     176 | 62.46
  6 |                -> Seq Scan on public.nation                 |     40 |            |     108 | 20.20
  7 |                -> Hash                                      |     20 |            |      76 | 41.97
  8 |                   -> Hash Join (9,16)                       |     20 | 10, 13     |      76 | 41.97
  9 |                      -> Streaming(type: REDISTRIBUTE)       |     20 |            |      82 | 28.54
 10 |                         -> Hash Join (11,14)               |     20 | 10, 13     |      82 | 27.63
 11 |                            -> Streaming(type: REDISTRIBUTE) |     20 |            |      70 | 14.19
 12 |                               -> Row Adapter                |     21 |            |      70 | 13.01
 13 |                                  -> CStore Scan on public.lineitem |     20 |     |      70 | 13.01
 14 |                            -> Hash                          |     21 |            |      12 | 13.13
 15 |                               -> Seq Scan on public.supplier |     20 |           |      12 | 13.13
 16 |                      -> Hash                                |     21 |            |      34 | 13.13
 17 |                         -> Seq Scan on public.partsupp     |     20 |            |      34 | 13.13
```

5.6 案例：调整局部聚簇键

局部聚簇（Partial Cluster Key, 简称PCK），列存储下一种通过min/max稀疏索引实现基表快速扫描的索引技术。Partial Cluster Key可以指定多列，但是一般不建议超过2列。PCK适用于列存大表点查询加速。

优化前

创建一个无局部聚簇（以下称为PCK）的列存表orders_no_pck，表定义如下：

```
                                pg_get_tabledef
---------------------------------------------------------------------------------
 SET search_path = dbadmin;                                                     +
 CREATE  TABLE orders_no_pck (                                                  +
         o_orderkey bigint NOT NULL,                                            +
         o_custkey bigint NOT NULL,                                             +
         o_orderstatus character(1) NOT NULL,                                   +
         o_totalprice numeric(15,2) NOT NULL,                                   +
         o_orderdate timestamp(0) without time zone NOT NULL,                   +
         o_orderpriority character(15) NOT NULL,                                +
         o_clerk character(15) NOT NULL,                                        +
         o_shippriority bigint NOT NULL,                                        +
         o_comment character varying(79) NOT NULL                               +
 )                                                                              +
 WITH (orientation=column, compression=low, colversion=2.0, enable_delta=false)+
 DISTRIBUTE BY HASH(o_orderkey)                                                 +
 TO GROUP group_version1;
(1 row)
```

执行以下SQL语句，查询某个点查询的执行计划：

```
EXPLAIN PERFORMANCE
SELECT * FROM orders_no_pck
WHERE o_orderkey = '13095143'
ORDER BY o_orderdate;
```

由下图可知执行时间为48毫秒，查看Datanode Information发现filter时间为19毫秒，CUNone比例为0。

```
gaussdb=>  EXPLAIN PERFORMANCE
gaussdb-> SELECT * FROM orders_no_pck
gaussdb-> WHERE o_orderkey = '13095143'
gaussdb-> ORDER BY o_orderdate;
                                                        QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------
 id |                 operation                 |       A-time       | A-rows | E-rows | E-distinct |  Peak Memory   | E-memory | A-width | E-width | E-costs
----+-------------------------------------------+--------------------+--------+--------+------------+----------------+----------+---------+---------+----------
  1 | ->  Row Adapter                           | 48.182             |      1 |      3 |            | 82KB           |          |         |     123 | 94838.01
  2 |    ->  Vector Streaming (type: GATHER)    | 48.175             |      1 |      3 |            | 825KB          |          |         |     123 | 94838.01
  3 |       ->  Vector Sort                     | [44.260, 44.772]   |      1 |      3 |            | [330KB, 411KB] | 16MB     | [0,167] |     123 | 94830.01
  4 |          ->  CStore Scan on public.orders_no_pck | [44.157, 44.669] |  1 |      1 |            | [1MB, 1MB]     | 1MB      |         |     123 | 94830.00

         Predicate Information (identified by plan id)
```

```
                    Datanode Information (identified by plan id)
---------------------------------------------------------------------------------------------------------------
  1 --Row Adapter
        (actual time=48.181..48.182 rows=1 loops=1)
        (CPU: ex c/r=676, ex row=1, ex cyc=676, inc cyc=4818100)
  2 --Vector Streaming (type: GATHER)
        (actual time=48.174..48.175 rows=1 loops=1)
        (Buffers: 0)
        (CPU: ex c/r=4817424, ex row=1, ex cyc=4817424, inc cyc=4817424)
  3 --Vector Sort
        dn_6001_6002 (actual time=44.461..44.461 rows=0 loops=1)
        dn_6003_6004 (actual time=44.259..44.260 rows=1 loops=1)
        dn_6005_6006 (actual time=44.772..44.772 rows=0 loops=1)
        dn_6001_6002 (Buffers: shared hit=389)
        dn_6003_6004 (Buffers: shared hit=389)
        dn_6005_6006 (Buffers: shared hit=389)
        dn_6001_6002 (CPU: ex c/r=0, ex row=0, ex cyc=11343, inc cyc=4446062)
        dn_6003_6004 (CPU: ex c/r=10101, ex row=1, ex cyc=10101, inc cyc=4425810)
        dn_6005_6006 (CPU: ex c/r=0, ex row=0, ex cyc=10257, inc cyc=4477201)
  4 --CStore Scan on public.orders_no_pck
        dn_6001_6002 (actual time=44.348..44.348 rows=0 loops=1) (filter time=19.721) (RoughCheck CU: CUNone: 0, CUSome: 84)
        dn_6003_6004 (actual time=38.704..44.157 rows=1 loops=1) (filter time=19.739) (RoughCheck CU: CUNone: 0, CUSome: 84)
        dn_6005_6006 (actual time=44.669..44.669 rows=0 loops=1) (filter time=19.568) (RoughCheck CU: CUNone: 0, CUSome: 84)
        dn_6001_6002 (Buffers: shared hit=389)
        dn_6003_6004 (Buffers: shared hit=389)
        dn_6005_6006 (Buffers: shared hit=389)
        dn_6001_6002 (CPU: ex c/r=0, ex row=5007635, ex cyc=4434719, inc cyc=4434719)
        dn_6003_6004 (CPU: ex c/r=0, ex row=5002975, ex cyc=4415709, inc cyc=4415709)
        dn_6005_6006 (CPU: ex c/r=0, ex row=4989390, ex cyc=4466944, inc cyc=4466944)
```

优化后

创建的列存表orders_pck。表定义如下：

```
                          pg_get_tabledef
--------------------------------------------------------------------------------
 SET search_path = dbadmin;                                                    +
 CREATE  TABLE orders_pck (                                                    +
         o_orderkey bigint NOT NULL,                                           +
         o_custkey bigint NOT NULL,                                            +
         o_orderstatus character(1) NOT NULL,                                  +
         o_totalprice numeric(15,2) NOT NULL,                                  +
         o_orderdate timestamp(0) without time zone NOT NULL,                  +
         o_orderpriority character(15) NOT NULL,                               +
         o_clerk character(15) NOT NULL,                                       +
         o_shippriority bigint NOT NULL,                                       +
         o_comment character varying(79) NOT NULL                              +
 )                                                                             +
 WITH (orientation=column, compression=low, colversion=2.0, enable_delta=false)+
 DISTRIBUTE BY HASH(o_orderkey)                                                +
 TO GROUP group_version1;
(1 row)
```

使用ALTER TABLE将字段o_orderkey设置为PCK：

```
postgres=> ALTER TABLE orders_pck ADD PARTIAL CLUSTER KEY(o_orderkey);
ALTER TABLE
```

执行以下SQL语句，再次查询同样的点查询SQL语句的执行计划：

```
EXPLAIN PERFORMANCE
SELECT * FROM orders_pck
WHERE o_orderkey = '13095143'
ORDER BY o_orderdate;
```

由下图可知执行时间为5毫秒，查看Datanode Information发现filter时间为0.5毫秒，CUNone比例为82。CUNone比列越高，PCK的性能收益越明显。

```
gaussdb=> EXPLAIN PERFORMANCE
gaussdb-> SELECT * FROM orders_pck
gaussdb-> WHERE o_orderkey = '13095143'
gaussdb-> ORDER BY o_orderdate;
                                                    QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------
 id |                 operation                  |     A-time     | A-rows | E-rows | E-distinct |  Peak Memory   | E-memory | A-width | E-width | E-costs
----+--------------------------------------------+----------------+--------+--------+------------+----------------+----------+---------+---------+----------
  1 | ->  Row Adapter                            | 5.597          |      1 |      3 |            | 82KB           |          |         |     123 | 94838.01
  2 |    ->  Vector Streaming (type: GATHER)     | 5.589          |      1 |      3 |            | 825KB          |          |         |     123 | 94838.01
  3 |       ->  Vector Sort                      | [1.858, 1.929] |      1 |      3 |            | [330KB, 411KB] | 16MB     | [0,167] |     123 | 94830.01
  4 |          ->  CStore Scan on public.orders_pck | [1.742, 1.804] |   1 |      1 |            | [1MB, 1MB]     | 1MB      |         |     123 | 94830.00

      Predicate Information (identified by plan id)
 -----------------------------------------------------------------
```

```
                                Datanode Information (identified by plan id)
-------------------------------------------------------------------------------------------------------------------------
  1 --Row Adapter
        (actual time=5.597..5.597 rows=1 loops=1)
        (CPU: ex c/r=815, ex row=1, ex cyc=815, inc cyc=559741)
  2 --Vector Streaming (type: GATHER)
        (actual time=5.589..5.589 rows=1 loops=1)
        (Buffers: shared hit=3)
        (CPU: ex c/r=558926, ex row=1, ex cyc=558926, inc cyc=558926)
  3 --Vector Sort
        dn_6001_6002 (actual time=1.858..1.858 rows=0 loops=1)
        dn_6003_6004 (actual time=1.914..1.914 rows=1 loops=1)
        dn_6005_6006 (actual time=1.929..1.929 rows=0 loops=1)
        dn_6001_6002 (Buffers: shared hit=395)
        dn_6003_6004 (Buffers: shared hit=396)
        dn_6005_6006 (Buffers: shared hit=396)
        dn_6001_6002 (CPU: ex c/r=0, ex row=0, ex cyc=11573, inc cyc=185784)
        dn_6003_6004 (CPU: ex c/r=12187, ex row=1, ex cyc=12187, inc cyc=191420)
        dn_6005_6006 (CPU: ex c/r=0, ex row=0, ex cyc=12455, inc cyc=192864)
  4 --CStore Scan on public.orders_pck
        dn_6001_6002 (actual time=1.742..1.742 rows=0 loops=1) (filter time=0.497) (RoughCheck CU: CUNone: 82, CUSome: 2)
        dn_6003_6004 (actual time=1.694..1.793 rows=1 loops=1) (filter time=0.509) (RoughCheck CU: CUNone: 82, CUSome: 2)
        dn_6005_6006 (actual time=1.804..1.804 rows=0 loops=1) (filter time=0.509) (RoughCheck CU: CUNone: 82, CUSome: 2)
        dn_6001_6002 (Buffers: shared hit=392)
        dn_6003_6004 (Buffers: shared hit=393)
        dn_6005_6006 (Buffers: shared hit=393)
        dn_6001_6002 (CPU: ex c/r=0, ex row=5007635, ex cyc=174211, inc cyc=174211)
        dn_6003_6004 (CPU: ex c/r=0, ex row=5002975, ex cyc=179233, inc cyc=179233)
        dn_6005_6006 (CPU: ex c/r=0, ex row=4989390, ex cyc=180409, inc cyc=180409)
```

5.7 案例：调整中间表存储方式

在GaussDB(DWS)中行存表使用行执行引擎，列存表使用列执行引擎。如果一个SQL语句涉及的表既有行存表又有列存表，系统会自动选择行执行引擎。由于列执行引擎的性能(除indexscan相关的算子)比行执行引擎性能要好很多，因此一般建议使用列存表。特别是对一些中间结果集转储的表，一定要分析清楚，使用合适的表存储类型。

优化前

某局点测试过程遇到如下的执行计划，客户希望将性能提升至3s内返回结果。

```
 id |                                                operation                                                 |          A-time          | A-rows | E-rows |  Peak Memory  | E-memory | A-width | E-width | E-costs
----+----------------------------------------------------------------------------------------------------------+--------------------------+--------+--------+---------------+----------+---------+---------+----------
  1 | ->  Streaming (type: GATHER)                                                                             | 4651.039                 |      7 |     17 | 304KB         | 1MB      |         |      41 | 101740.13
  2 |    ->  Hash Join (3,7)                                                                                   | [4.501,4629.889]         |      7 |     17 | [8KB, 8KB]    | 1MB      |         |      41 | 101739.10
  3 |       ->  Append                                                                                         | [3474.514,3840.836]      |  34752433 |  108109436 | [1KB, 1KB]    | 1MB      |         |      49 | 88969.75
  4 |          ->  Row Adapter                                                                                 | [2727.901,3040.924]      |  33011417 |  99098596 | [49KB, 49KB]  | 1MB      |         |      49 | 70615.19
  5 |             ->  Partitioned Dfs Scan on sd_data.act_account_his ta                                       | [2288.421,2568.707]      |  33011417 |  99098596 | [1002KB, 1012KB] | 1MB      |         |      49 | 70615.19
  6 |          ->  Seq Scan on cstore.pg_delta_2425217623 ta                                                   | [163.977,169.707]        |   1741016 |   9010840 | [15KB, 15KB]  | 1MB      |         |      50 | 18354.56
  7 |       ->  Hash                                                                                           | [4.385,7.999]            |      9 |     32 | [260KB, 292KB] | 16MB     | [0, 36] |      30 | 100.17
  8 |          ->  Streaming(type: REDISTRIBUTE)                                                               | [4.384,7.997]            |      9 |     32 | [1054KB, 1055KB] | 1MB      |         |      30 | 100.17
  9 |             ->  Hash Join (10,11)                                                                        | [0.162,1.043]            |      9 |     32 | [5KB, 5KB]    | 1MB      |         |      30 | 100.06
 10 |                ->  Seq Scan on pg_temp_cn_5001_140148717123328.input_acct_id_tbl tb                      | [0.005,0.176]            |   1030 |  31968 | [11KB, 11KB]  | 1MB      |         |      11 | 16.99
 11 |                ->  Hash                                                                                  | [0.001,0.849]            |      9 |     32 | [288KB, 290KB] | 16MB     | [0, 37] |      19 | 80.31
 12 |                   ->  HashAggregate                                                                      | [0.001,0.849]            |      9 |     32 | [10KB, 13KB]  | 1MB      |         |      19 | 80.30
 13 |                      ->  Seq Scan on public.row_unlogged_table                                           | [0.000,0.847]            |    449 |    449 | [13KB, 13KB]  | 1MB      |         |      19 | 78.70
(13 rows)
```

优化后

经过分析发现计划走了行引擎。根本原因是：临时计划表input_acct_id_tbl和中间结果转储表row_unlogged_table使用了行存表。

修改这两个表为列存表之后，性能提升至1.6s。

```
 id |                                                operation                                                 |          A-time          | A-rows | E-rows |  Peak Memory  | E-memory | A-width | E-width | E-costs
----+----------------------------------------------------------------------------------------------------------+--------------------------+--------+--------+---------------+----------+---------+---------+----------
  1 | ->  Row Adapter                                                                                          | 1567.367                 |      7 |     17 | 39KB          |          |         |      41 | 101758.52
  2 |    ->  Vector Streaming (type: GATHER)                                                                   | 1567.349                 |      7 |     17 | 393KB         |          |         |      41 | 101758.52
  3 |       ->  Vector Hash Join (4,8)                                                                         | [8.130,1629.101]         |      7 |     17 | [2362KB, 2446KB] | 16MB     |         |      41 | 101757.48
  4 |          ->  Vector Append                                                                               | [542.823,1452.479]       |  5681770 |  108109436 | [1KB, 1KB]    | 1MB      |         |      49 | 88969.75
  5 |             ->  Partitioned Dfs Scan on sd_data.act_account_his ta                                       | [295.796,1195.830]       |  3940754 |  99098596 | [861KB, 1012KB] | 1MB      |         |      49 | 70615.19
  6 |             ->  Vector Adapter                                                                           | [236.065,260.284]        |   1741016 |   9010840 | [129KB, 129KB] | 1MB      |         |      50 | 18354.56
  7 |                ->  Seq Scan on cstore.pg_delta_2425217623 ta                                             | [152.595,168.048]        |   1741016 |   9010840 | [15KB, 15KB]  | 1MB      |         |      50 | 18354.56
  8 |          ->  Vector Streaming(type: REDISTRIBUTE)                                                        | [7.727,12.981]           |      9 |     32 | [1092KB, 1141KB] | 1MB      | [0, 40] |      30 | 118.56
  9 |             ->  Vector Hash Join (10,11)                                                                 | [0.132,4.958]            |      9 |     32 | [2217KB, 2217KB] | 16MB     |         |      30 | 118.45
 10 |                ->  CStore Scan on pg_temp_cn_5001_140148155066112.input_acct_id_tbl tb                   | [4.372,4.372]            |    999 |  31968 | [207KB, 207KB] | 1MB      |         |      11 | 81.00
 11 |                ->  Vector Hash Aggregate                                                                 | [0.062,0.209]            |      9 |     32 | [2225KB, 2225KB] | 16MB     | [0, 35] |      19 | 33.67
 12 |                   ->  CStore Scan on public.col_unlogged_table                                           | [0.011,0.107]            |    449 |    449 | [541KB, 598KB] | 1MB      |         |      19 | 32.08
```

5.8 案例：改建分区表

逻辑上的一张表根据某种策略分成多个物理块进行存储，这张逻辑上的表称之为分区表，每个物理块则称为一个分区。一般对数据和查询都有明显区间段特征的表使用分区策略可通过较小不必要的数据扫描，从而提升查询性能

在查询时，可通过分区剪枝技术尽可能减少底层数据扫描，即缩小表的扫描范围。分区剪枝是指对于分区表或分区索引来说，优化器可以自动从FROM和WHERE字句里根据分区键提取出需要扫描的分区，从而避免全表扫描，减少扫描的数据块，提高性能。

优化前

创建一个非分区表orders_no_part，表定义如下：

```
                          pg_get_tabledef
--------------------------------------------------------------------------------
 SET search_path = dbadmin;
 CREATE  TABLE orders_no_part (
         o_orderkey bigint NOT NULL,
         o_custkey bigint NOT NULL,
         o_orderstatus character(1) NOT NULL,
         o_totalprice numeric(15,2) NOT NULL,
         o_orderdate timestamp(0) without time zone NOT NULL,
         o_orderpriority character(15) NOT NULL,
         o_clerk character(15) NOT NULL,
         o_shippriority bigint NOT NULL,
         o_comment character varying(79) NOT NULL
 )
 WITH (orientation=column, compression=low, colversion=2.0, enable_delta=false)
 DISTRIBUTE BY HASH(o_orderkey)
 TO GROUP group_version1;
(1 row)
```

执行以下SQL语句查询非分区表的执行计划：

```
EXPLAIN PERFORMANCE
SELECT count(*) FROM orders_no_part WHERE
o_orderdate >= '1996-01-01 00:00:00'::timestamp(0);
```

由下图可知执行时间为73毫秒，其中全表扫描的时间为44~45毫秒。

```
gaussdb=> EXPLAIN PERFORMANCE
gaussdb-> SELECT count(*) FROM orders_no_part WHERE
gaussdb-> o_orderdate >= '1996-01-01 00:00:00'::timestamp(0);
                                                       QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------
 id |                 operation                  |      A-time      | A-rows  | E-rows  | E-distinct |  Peak Memory   | E-memory | A-width | E-width | E-costs
----+--------------------------------------------+------------------+---------+---------+------------+----------------+----------+---------+---------+----------
  1 | ->  Row Adapter                            | 73.623           |       1 |       1 |            | 10KB           |          |         |       8 | 99791.27
  2 |    ->  Vector Aggregate                    | 73.611           |       1 |       1 |            | 177KB          |          |         |       8 | 99791.27
  3 |       ->  Vector Streaming (type: GATHER)  | 73.575           |       3 |       3 |            | 89KB           |          |         |       8 | 99791.27
  4 |          ->  Vector Aggregate              | [54.963, 55.561] |       3 |       3 |            | [138KB, 138KB] | 1MB      |         |       8 | 99783.27
  5 |             ->  CStore Scan on public.orders_no_part | [44.572, 45.077] | 5890663 | 5943908 |            | [308KB, 308KB] | 1MB      |         |       0 | 94830.00
```

优化后

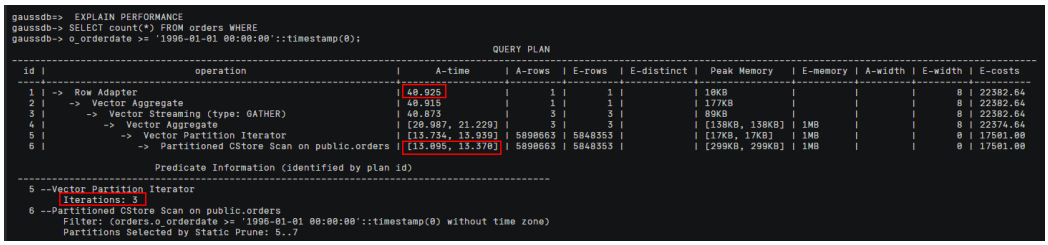
创建一个分区表orders。表定义如下：

```
                                    pg_get_tabledef
------------------------------------------------------------------------------------------------------------
 SET search_path = dbadmin;                                                                                +
 CREATE  TABLE orders (                                                                                    +
         o_orderkey bigint NOT NULL,                                                                       +
         o_custkey bigint NOT NULL,                                                                        +
         o_orderstatus character(1) NOT NULL,                                                              +
         o_totalprice numeric(15,2) NOT NULL,                                                              +
         o_orderdate timestamp(0) without time zone NOT NULL,                                              +
         o_orderpriority character(15) NOT NULL,                                                           +
         o_clerk character(15) NOT NULL,                                                                   +
         o_shippriority bigint NOT NULL,                                                                   +
         o_comment character varying(79) NOT NULL                                                          +
 )                                                                                                         +
 WITH (orientation=column, compression=low, colversion=2.0, enable_delta=false)                            +
 DISTRIBUTE BY HASH(o_orderkey)                                                                            +
 TO GROUP group_version1                                                                                   +
 PARTITION BY RANGE (o_orderdate)                                                                          +
 (                                                                                                         +
          PARTITION o_orderdate_1 VALUES LESS THAN ('1993-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default,+
          PARTITION o_orderdate_2 VALUES LESS THAN ('1994-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default,+
          PARTITION o_orderdate_3 VALUES LESS THAN ('1995-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default,+
          PARTITION o_orderdate_4 VALUES LESS THAN ('1996-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default,+
          PARTITION o_orderdate_5 VALUES LESS THAN ('1997-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default,+
          PARTITION o_orderdate_6 VALUES LESS THAN ('1998-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default,+
          PARTITION o_orderdate_7 VALUES LESS THAN ('1999-01-01 00:00:00'::timestamp(0) without time zone) TABLESPACE pg_default +
 )                                                                                                         +
 ENABLE ROW MOVEMENT;
(1 row)
```


再次执行SQL语句，查询分区表的执行计划。执行时间为40毫秒，其中表扫描时间仅为13毫秒，另Iterations越小，分区剪枝效果越好。

```
EXPLAIN PERFORMANCE
SELECT count(*) FROM orders_no_part WHERE
o_orderdate >= '1996-01-01 00:00:00'::timestamp(0);
```

由下图可知执行时间为40毫秒，其中表扫描时间仅为13毫秒。另外Iterations越小，分区剪枝效果越好。



5.9 案例：调整 GUC 参数 best_agg_plan

现象描述

t1的表定义为：

```
create table t1(a int, b int, c int) distribute by hash(a);
```

假设agg下层算子所输出结果集的分布列为setA，agg操作的group by列为setB，则在Stream框架下，Agg操作可以分为两个场景。

场景一：setA是setB的一个子集。

对于这种场景，直接对下层结果集进行汇聚的结果就是正确的汇聚结果，上层算子直接使用即可。如下图所示：

```
explain select a, count(1) from t1 group by a;
 id |         operation          | E-rows | E-width | E-costs
----+----------------------------+--------+---------+---------
  1 | ->  Streaming (type: GATHER) |     30 |       4 | 15.56
  2 |    ->  HashAggregate       |     30 |       4 | 14.31
  3 |       ->  Seq Scan on t1   |     30 |       4 | 14.14
(3 rows)
```

场景二：setA不是setB的一个子集。

对于这种场景，Stream执行框架分为如下三种计划形态：

hashagg+gather(redistribute)+hashagg

redistribute+hashagg(+gather)

hashagg+redistribute+hashagg(+gather)

GaussDB(DWS)提供了guc参数best_agg_plan来干预执行计划，强制其生成上述对应的执行计划，此参数取值范围为0，1，2，3

- 取值为1时，强制生成第一种计划。
- 取值为2时，如果group by列可以重分布，强制生成第二种计划，否则生成第一种计划。

- 取值为3时，如果group by列可以重分布，强制生成第三种计划，否则生成第一种计划。
- 取值为0时，优化器会根据以上三种计划的估算代价选择最优的一种计划生成。

具体影响如下：

```
set best_agg_plan to 1;
SET
explain select b,count(1) from t1 group by b;
 id |           operation           | E-rows | E-width | E-costs
----+-------------------------------+--------+---------+---------
  1 | ->  HashAggregate             |      8 |       4 | 15.83
  2 |    ->  Streaming (type: GATHER) |     25 |       4 | 15.83
  3 |       ->  HashAggregate       |     25 |       4 | 14.33
  4 |          ->  Seq Scan on t1   |     30 |       4 | 14.14
(4 rows)
set best_agg_plan to 2;
SET
explain select b,count(1) from t1 group by b;
 id |              operation              | E-rows | E-width | E-costs
----+-------------------------------------+--------+---------+---------
  1 | ->  Streaming (type: GATHER)        |     30 |       4 | 15.85
  2 |    ->  HashAggregate                |     30 |       4 | 14.60
  3 |       ->  Streaming(type: REDISTRIBUTE) |     30 |       4 | 14.45
  4 |          ->  Seq Scan on t1         |     30 |       4 | 14.14
(4 rows)
set best_agg_plan to 3;
SET
explain select b,count(1) from t1 group by b;
 id |              operation              | E-rows | E-width | E-costs
----+-------------------------------------+--------+---------+---------
  1 | ->  Streaming (type: GATHER)        |     30 |       4 | 15.84
  2 |    ->  HashAggregate                |     30 |       4 | 14.59
  3 |       ->  Streaming(type: REDISTRIBUTE) |     25 |       4 | 14.59
  4 |          ->  HashAggregate          |     25 |       4 | 14.33
  5 |             ->  Seq Scan on t1      |     30 |       4 | 14.14
(5 rows)
```

总结

通常优化器总会选择最优的执行计划，但是众所周知代价估算，尤其是中间结果集的代价估算一般会有比较大的偏差，这种比较大的偏差就可能会导致agg的计算方式出现比较大的偏差，这时候就需要通过best_agg_plan进行agg计算模型的干预。

一般来说，当agg汇聚的收敛度很小时，即结果集的个数在agg之后并没有明显变少时（经验上以5倍为临界点），选择redistribute+hashagg执行方式，否则选择hashagg+redistribute+hashagg执行方式。

5.10 案例：改写 SQL 消除子查询（案例 1）

现象描述

```
select
    1,
    (select count(*) from customer_address_001 a4 where a4.ca_address_sk = a.ca_address_sk) as GZCS
from customer_address_001 a;
```

此SQL性能较差，查看发现执行计划中存在SubPlan，引用SubPlan结果的算子可能需要反复的调用获取这个SubPlan的值，即SubPlan以下的结果要重复执行很多次。具体如下：

```
postgres=# explain select 1,(select count(*)
postgres(#           from customer_address_001 a4
postgres(#          where a4.ca_address_sk = a.ca_address_sk
postgres(#         ) as GZCS from customer_address_001 a;
 id |                          operation                          | E-rows | E-width | E-costs
----+-------------------------------------------------------------+--------+---------+---------
  1 | ->  Streaming (type: GATHER)                                |    320 |       4 | 4529.27
  2 |    ->  Seq Scan on customer_address_001 a                   |    320 |       4 | 4496.27
  3 |       ->  Aggregate  [2, SubPlan 1]                         |     32 |       4 | 139.50
  4 |          ->  Result                                         |  10240 |       4 | 138.69
  5 |             ->  Materialize                                 |  10240 |       4 | 138.69
  6 |                ->  Streaming(type: BROADCAST)               |  10240 |       4 | 137.09
  7 |                   ->  Seq Scan on customer_address_001 a4   |    320 |       4 | 32.32
(7 rows)
```

优化说明

此优化的核心就是消除子查询。那么从SQL语义出发，可以等价改写SQL为：

```
select
   1,
   coalesce(a4.c1, 0)
from
   (select count(*) c1, a4.ca_address_sk from customer_address_001 a4 group by a4.ca_address_sk) a4
right join customer_address_001 a on a4.ca_address_sk = a.ca_address_sk;
```

5.11 案例：改写 SQL 消除子查询（案例 2）

现象描述

某局点客户反馈如下SQL语句的执行时间超过1天未结束：

```
UPDATE calc_empfyc_c_cusr1 t1
SET ln_rec_count =
 (
   SELECT CASE WHEN current_date - ln_process_date + 1 <= 12 THEN 0 ELSE t2.ln_rec_count END
   FROM calc_empfyc_c1_policysend_tmp t2
   WHERE t1.ln_branch = t2.ln_branch AND t1.ls_policyno_cusr1 = t2.ls_policyno_cusr1
)
WHERE dsign = '1'
AND flag = '1'
AND EXISTS
   (SELECT 1
   FROM calc_empfyc_c1_policysend_tmp t2
   WHERE t1.ln_branch = t2.ln_branch AND t1.ls_policyno_cusr1 = t2.ls_policyno_cusr1
   );
```

对应的执行计划如下：

```
Streaming (type: GATHER)  (cost=44693.26..19548819558.34 rows=4058158 width=1061)
  Node/s: All datanodes
  ->  Update on channel.calc_empfyc_c_cusr1 t1  (cost=44689.26..19546717163.01 rows=4058158 width=1061)
        ->  Hash Join  (cost=44689.26..19546717163.01 rows=4058158 width=1061)
              Hash Cond: (((t1.ln_branch)::text = ((t2.ln_branch)::text)) AND ((t1.ls_policyno_cusr1)::text = ((t2.ls_policyno_cusr1)::text)))
              ->  Seq Scan on channel.calc_empfyc_c_cusr1 t1  (cost=0.00..28692.39 rows=7105667 width=1055)
                    Filter: ((t1.dsign = '1'::bpchar) AND (t1.flag = '1'::bpchar))
              ->  Hash  (cost=2112.16..2112.16 rows=108998016 width=37)
                    ->  Unique  (cost=2112.06..2112.16 rows=108998016 width=37)
                          ->  Sort  (cost=2112.06..2112.09 rows=775 width=37)
                                Sort Key: ((t2.ln_branch)::text), ((t2.ls_policyno_cusr1)::text)
                                ->  Streaming(type: BROADCAST)  (cost=2109.81..2111.85 rows=775 width=37)
                                      Spawn on: All datanodes
                                      ->  HashAggregate  (cost=2109.81..2109.82 rows=12 width=37)
                                            Group By Key: (t2.ln_branch)::text, (t2.ls_policyno_cusr1)::text
                                            ->  Seq Scan on channel.calc_empfyc_c1_policysend_tmp t2  (cost=0.00..1406.87 rows=1703094 width=37)
              SubPlan 1
                ->  Result  (cost=0.00..308262.89 rows=108998016 width=44)
                      Filter: (((t1.ln_branch)::text = (t2.ln_branch)::text) AND ((t1.ls_policyno_cusr1)::text = (t2.ls_policyno_cusr1)::text))
                      ->  Materialize  (cost=0.00..295489.68 rows=108998016 width=44)
                            ->  Streaming(type: BROADCAST)  (cost=0.00..286974.21 rows=108998016 width=44)
                                  Spawn on: All datanodes
                                  ->  Seq Scan on channel.calc_empfyc_c1_policysend_tmp t2  (cost=0.00..1406.87 rows=1703094 width=44)
```

优化说明

很明显，执行计划中存在SubPlan，并且SubPlan中的运算相当重，即此SubPlan是一个明确的性能瓶颈点。

根据SQL语意等价改写SQL消除SubPlan如下：

```
UPDATE calc_empfyc_c_cusr1 t1
SET ln_rec_count = CASE WHEN current_date - ln_process_date + 1 <= 12 THEN 0 ELSE t2.ln_rec_count END
FROM calc_empfyc_c1_policysend_tmp t2
WHERE
t1.dsign = '1' AND t1.flag = '1'
AND t1.ln_branch = t2.ln_branch AND t1.ls_policyno_cusr1 = t2.ls_policyno_cusr1;
```

改写之后SQL语句在50S内执行完成

5.12 案例：改写 SQL 排除剪枝干扰

分区表查询中表达式一侧不是单纯的分区键、而是包含分区键的表达式的Filter条件，这种类型的Filter条件是不能用来剪枝的。

优化前

t_ddw_f10_op_cust_asset_mon为分区表，分区键为year_mth，此字段是由年月两个值拼接而成的整数。

测试SQL如下：

```
SELECT
    count(1)
FROM t_ddw_f10_op_cust_asset_mon b1
WHERE b1.year_mth  < substr('20200722',1 ,6 )
AND b1.year_mth + 1 >= substr('20200722',1 ,6 );
```

测试结果显示此SQL的表Scan耗时长达10s，查询SQL语句的执行计划如下

```
EXPLAIN (ANALYZE ON, VERBOSE ON)
SELECT
    count(1)
FROM t_ddw_f10_op_cust_asset_mon b1
WHERE b1.year_mth < substr('20200722',1 ,6 )
AND b1.year_mth + 1 >= cast(substr('20200722',1 ,6 ) AS int);
                                                                                    QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
  id |                                   operation                                   |       A-time         | A-rows  | E-rows  | E-distinct | Peak Memory  | E-memory | A-width | E-width |  E-costs
 ----+--------------------------------------------------------------------------------+-----------------------+----------+----------+------------+--------------+----------+---------+---------+-----------
   1 | ->  Aggregate                                                   | 10662.260            |        1 |        1 |            | 32KB         |          |         |       8 | 593656.42
   2 |    ->  Streaming (type: GATHER)                                 | 10662.172            |        4 |        4 |            | 136KB        |          |         |       8 | 593656.42
   3 |       ->  Aggregate                                             | [9692.785, 10656.068] |        4 |        4 |            | [24KB, 24KB] | 1MB      |         |       8 | 593646.42
   4 |          ->  Partition Iterator                                 | [8787.198, 9629.138]  | 16384000 | 32752850 |            | [16KB, 16KB] | 1MB      |         |       0 | 573175.88
   5 |             ->  Partitioned Seq Scan on public.t_ddw_f10_op_cust_asset_mon b1 | [8365.655, 9152.115]  | 16384000 | 32752850 |            | [32KB, 32KB] | 1MB      |         |       0 | 573175.88

                                 SQL Diagnostic Information
 ------------------------------------------------------------------------------------------
 Partitioned table unprunable Qual
        table public.t_ddw_f10_op_cust_asset_mon b1:
```

```
        left side of expression "((year_mth + 1) > 202008)" invokes function-call/type-conversion

                Predicate Information (identified by plan id)
-------------------------------------------------------------------------------
  4 --Partition Iterator
        Iterations: 6
  5 --Partitioned Seq Scan on public.t_ddw_f10_op_cust_asset_mon b1
        Filter: ((b1.year_mth < 202007::bigint) AND ((b1.year_mth + 1) >= 202007))
        Rows Removed by Filter: 81920000
        Partitions Selected by Static Prune: 1..6
```

优化后

分析语句的执行计划，查看执行计划中的SQL自诊断信息，发现如下诊断信息：

```
                              SQL Diagnostic Information
--------------------------------------------------------------------------------------
Partitioned table unprunable Qual
        table public.t_ddw_f10_op_cust_asset_mon b1:
        left side of expression "((year_mth + 1) > 202008)" invokes function-call/type-conversion
```

Filter条件中存在表达式(year_mth + 1) > 202008，这种**表达式一侧不是单纯的分区键、而是包含分区键的表达式的Filter条件是不能用来剪枝的**，因而导致查询语句扫描了几乎整个分区表的数据。

跟原始SQL语句对比，可以确定表达式 '(year_mth + 1) > 202008' 是从表达式 'b1.year_mth + 1 > substr('20200822',1 ,6)' 衍生而来，按照诊断信息把修改SQL语句为如下方式：

```
SELECT
    count(1)
FROM t_ddw_f10_op_cust_asset_mon b1
WHERE b1.year_mth <= substr('20200822',1 ,6 )
AND b1.year_mth > cast(substr('20200822',1 ,6 ) AS int) - 1;
```

改写之后，SQL语句的执行信息如下，可以看到不剪枝告警已经消除，剪枝后需要扫描分区数为1，执行时间从10s提升至3s。

```
EXPLAIN (analyze ON, verbose ON)
SELECT
    count(1)
FROM t_ddw_f10_op_cust_asset_mon b1
WHERE b1.year_mth < substr('20200722',1 ,6 )
AND b1.year_mth >= cast(substr('20200722',1 ,6 ) AS int) - 1;
                                                  QUERY PLAN
----------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------
 id |                            operation                             |       A-time        | A-rows  | E-rows  | E-
distinct | Peak Memory  | E-memory | A-width | E-width |  E-costs
----+------------------------------------------------------------------------------+----------------------+----------
+----------+------------+--------------+----------+---------+---------+-----------
  1 | ->  Aggregate                                                     | 3009.796            |       1 |       1 |           |
32KB         |          |         |       8 | 501541.70
  2 |    ->  Streaming (type: GATHER)                                    | 3009.718            |       4 |       4
|           | 136KB        |          |         |       8 | 501541.70
  3 |       ->  Aggregate                                               | [2675.509, 3003.298] |       4 |       4
|           | [24KB, 24KB] | 1MB      |         |       8 | 501531.70
  4 |          ->  Partition Iterator                                   | [1820.725, 2053.836] | 16384000 |
16380697 |            | [16KB, 16KB] | 1MB      |         |       0 | 491293.75
  5 |             ->  Partitioned Seq Scan on public.t_ddw_f10_op_cust_asset_mon b1 | [1420.972, 1590.083] |
16384000 | 16380697 |            | [16KB, 16KB] | 1MB      |         |       0 | 491293.75

                Predicate Information (identified by plan id)
--------------------------------------------------------------------------
  4 --Partition Iterator
        Iterations: 1
  5 --Partitioned Seq Scan on public.t_ddw_f10_op_cust_asset_mon b1
```

```
Filter: ((b1.year_mth < 202007::bigint) AND (b1.year_mth >= 202006))
Partitions Selected by Static Prune: 6
```

5.13 案例：改写 SQL 消除 in-clause

优化前

in-clause/any-clause是常见的SQL语句约束条件，有时in或any后面的clause都是常量，类似于：

```
select
count(1)
from calc_empfyc_c1_result_tmp_t1
where ls_pid_cusr1 in (‘20120405’, ‘20130405’);
```

或者

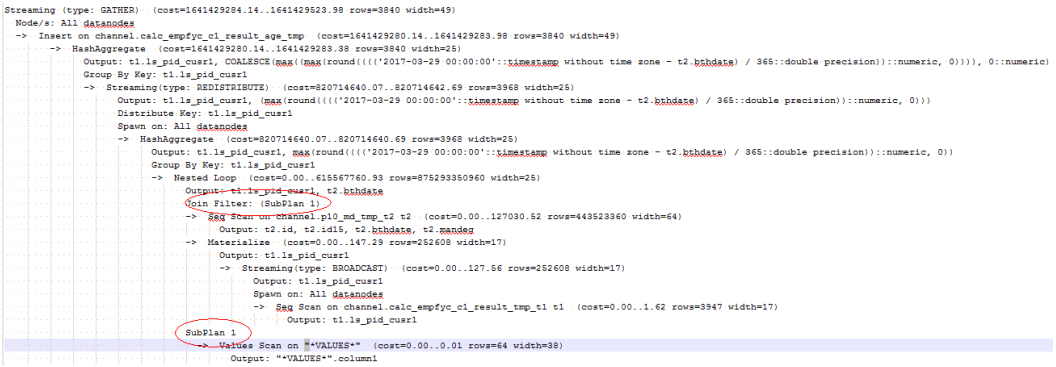
```
select
count(1)
from calc_empfyc_c1_result_tmp_t1
where ls_pid_cusr1 in any(‘20120405’, ‘20130405’);
```

但是也有一些如下的特殊用法：

```
SELECT
ls_pid_cusr1,COALESCE(max(round((current_date-bthdate)/365)),0)
FROM calc_empfyc_c1_result_tmp_t1 t1,p10_md_tmp_t2 t2
WHERE t1.ls_pid_cusr1 = any(values(id),(id15))
GROUP BY ls_pid_cusr1;
```

其中，id、id15为p10_md_tmp_t2中的两列，“t1.ls_pid_cusr1 = any(values(id),(id15))”等价于“t1.ls_pid_cusr1 = id or t1.ls_pid_cusr1 = id15”。

因此join-condition实质上是一个不等式，这种不等值的join操作必须走nestloop，对应执行计划如下：



优化后

测试发现由于两表结果集过大，导致nestloop耗时过长，超过一小时未返回结果，因此性能优化的关键是消除nestloop，让join走更高效的hashjoin。从语义等价的角度消除any-clause，SQL改写如下：

```
select
ls_pid_cusr1,COALESCE(max(round(ym/365)),0)
from
(
	(
		SELECT
```

```
                ls_pid_cusr1,(current_date-bthdate) as ym
        FROM calc_empfyc_c1_result_tmp_t1 t1,p10_md_tmp_t2 t2
        WHERE t1.ls_pid_cusr1 = t2.id and t1.ls_pid_cusr1 != t2.id15
    )
    union all
    (
        SELECT
            ls_pid_cusr1,(current_date-bthdate) as ym
        FROM calc_empfyc_c1_result_tmp_t1 t1,p10_md_tmp_t2 t2
        WHERE t1.ls_pid_cusr1 = id15
    )
)
GROUP BY ls_pid_cusr1;
```

注意：尽量使用union all代替union。union在合并两个集合时会执行去重操作，而union all则直接将两个结果集合并、不执行去重。执行去重会消耗大量的时间，因此，在一些实际应用场景中，如果通过业务逻辑已确认两个集合不存在重叠，可用union all替代union以便提升性能。

优化后的SQL查询由两个等值join的子查询构成，而每个子查询都可以走更适合此场景的hashjoin。优化后的执行计划如下

```
 id |                                operation                                |        A-time        |  A-rows   |  E-rows   |   Peak Memory    | E-memory |  A-width
----+-------------------------------------------------------------------------+----------------------+-----------+-----------+------------------+----------+-----------
  1 | ->  Streaming (type: GATHER)                                            | 6737.281             |         0 |       192 | 292KB            |          |
  2 |   ->  Insert on channel.calc_empfyc_c1_result_age_tmp                   | [4665.024,4990.666]  |         0 |       192 | [1108KB, 1108KB] | 1MB      |
  3 |     ->  HashAggregate                                                   | [4664.996,4990.641]  |         0 |       192 | [12KB, 12KB]     | 16MB     |
  4 |       ->  Streaming(type: REDISTRIBUTE)                                 | [4664.991,4990.637]  |         0 |      3392 | [2090KB, 2090KB] | 1MB      |
  5 |         ->  HashAggregate                                               | [3416.939,4958.348]  |         0 |      3392 | [14KB, 14KB]     | 16MB     |
  6 |           ->  Append                                                    | [3416.936,4958.340]  |         0 |      4011 | [1KB, 1KB]       | 1MB      |
  7 |             ->  Hash Join (8,9)                                         | [2011.226,3080.697]  |         0 |      3947 | [6KB, 6KB]       | 1MB      |
  8 |               ->  Seq Scan on channel.p10_md_tmp_t2 t2                  | [803.782,1238.984]   | 443525717 | 443523360 | [12KB, 12KB]     | 1MB      |
  9 |               ->  Hash                                                  | [4.357,328.979]      |    252608 |    252608 | [482KB, 482KB]   | 16MB     | [35,39]
 10 |                 ->  Streaming(type: BROADCAST)                          | [2.345,326.320]      |    252608 |    252608 | [2090KB, 2090KB] | 1MB      |
 11 |                   ->  Seq Scan on channel.calc_empfyc_c1_result_tmp_t1 t1 | [0.011,0.030]        |      3947 |      3947 | [11KB, 11KB]     | 1MB      |
 12 |           ->  Hash Join (13,14)                                         | [1376.258,2066.110]  |         0 |        64 | [5KB, 5KB]       | 1MB      |
 13 |             ->  Seq Scan on channel.p10_md_tmp_t2 t2                    | [777.552,1388.499]   | 443525717 | 443523360 | [12KB, 12KB]     | 1MB      |
 14 |             ->  Hash                                                    | [2.812,4.217]        |    252608 |    252608 | [482KB, 482KB]   | 16MB     | [23,27]
 15 |               ->  Streaming(type: BROADCAST)                            | [1.276,1.868]        |    252608 |    252608 | [2090KB, 2090KB] | 1MB      |
 16 |                 ->  Seq Scan on channel.calc_empfyc_c1_result_tmp_t1 t1 | [0.010,0.033]        |      3947 |      3947 | [11KB, 11KB]     | 1MB      |
(16 rows)
```

优化后，从超过1个小时未返回结果优化到7s返回结果。

5.14 案例：使用 partial cluster key

列存表可以选取某一列或几列设置为partial cluster key(column_name[, ...])。在导入数据时，按设置的列进行局部排序（默认每70个CU即420万行排序一次），生成的CU会聚集在一起，即CU的min,max会在一个较小的区间内。当查询时，where条件含有这些列时，可产生良好的过滤效果。

优化前

未使用partial cluster key。表定义如下：

```
CREATE TABLE lineitem
(
L_ORDERKEY    BIGINT NOT NULL
, L_PARTKEY     BIGINT NOT NULL
, L_SUPPKEY     BIGINT NOT NULL
, L_LINENUMBER  BIGINT NOT NULL
, L_QUANTITY    DECIMAL(15,2) NOT NULL
, L_EXTENDEDPRICE  DECIMAL(15,2) NOT NULL
, L_DISCOUNT    DECIMAL(15,2) NOT NULL
, L_TAX         DECIMAL(15,2) NOT NULL
, L_RETURNFLAG  CHAR(1) NOT NULL
, L_LINESTATUS  CHAR(1) NOT NULL
, L_SHIPDATE    DATE NOT NULL
, L_COMMITDATE  DATE NOT NULL
, L_RECEIPTDATE DATE NOT NULL
, L_SHIPINSTRUCT CHAR(25) NOT NULL
, L_SHIPMODE     CHAR(10) NOT NULL
, L_COMMENT      VARCHAR(44) NOT NULL
```

```
)
with (orientation = column)
distribute by hash(L_ORDERKEY);

select
sum(l_extendedprice * l_discount) as revenue
from
lineitem
where
l_shipdate >= '1994-01-01'::date
and l_shipdate < '1994-01-01'::date + interval '1 year'
and l_discount between 0.06 - 0.01 and 0.06 + 0.01
and l_quantity < 24;
```

导入数据后执行查询，查看执行时间：

图 5-5 未使用 partial cluster key

```
 id |                operation                 |       A-time        | A-rows | E-rows |  Peak Memory   | A-width | E-width |  E-costs
----+------------------------------------------+---------------------+--------+--------+----------------+---------+---------+-----------
  1 | ->  Row Adapter                          | 1653.156            |      1 |      1 | 12KB           |         |      44 | 205803.90
  2 |    ->  Vector Aggregate                  | 1653.146            |      1 |      1 | 184KB          |         |      44 | 205803.90
  3 |       ->  Vector Streaming (type: GATHER)| 1653.070            |      1 |      1 | 174KB          |         |      44 | 205803.90
  4 |          ->  Vector Aggregate            | [1481.497,1481.497] |      1 |      1 | [225KB, 225KB] |         |      44 | 205803.84
  5 |             ->  CStore Scan on public.lineitem | [1405.004,1405.004] | 114160 | 111485 | [792KB, 792KB] |         |      12 | 205246.40
(5 rows)
```

图 5-6 未使用 partial cluster key 后 CU 加载情况

```
  5 --CStore Scan on public.lineitem
        datanode1 (actual time=40.623..1405.004 rows=114160 loops=1)
        datanode1 (RoughCheck CU: CUNone: 0, CUSome: 101)
        datanode1 (LLVM Optimized)
        datanode1 (Buffers: shared hit=18385 read=23)
        datanode1 (CPU: ex c/r=31917, ex cyc=3643646206, inc cyc=3643646206)
```

优化后

where条件中l_shipdate和l_quantity的distinct值数量较少且可以做min max过滤，将字段l_shipdate、l_quantity设置为PCK修改表定义如下：

```
CREATE TABLE lineitem
(
L_ORDERKEY    BIGINT NOT NULL
, L_PARTKEY     BIGINT NOT NULL
, L_SUPPKEY     BIGINT NOT NULL
, L_LINENUMBER  BIGINT NOT NULL
, L_QUANTITY    DECIMAL(15,2) NOT NULL
, L_EXTENDEDPRICE  DECIMAL(15,2) NOT NULL
, L_DISCOUNT    DECIMAL(15,2) NOT NULL
, L_TAX         DECIMAL(15,2) NOT NULL
, L_RETURNFLAG  CHAR(1) NOT NULL
, L_LINESTATUS  CHAR(1) NOT NULL
, L_SHIPDATE    DATE NOT NULL
, L_COMMITDATE  DATE NOT NULL
, L_RECEIPTDATE DATE NOT NULL
, L_SHIPINSTRUCT CHAR(25) NOT NULL
, L_SHIPMODE     CHAR(10) NOT NULL
, L_COMMENT      VARCHAR(44) NOT NULL
, partial cluster key(l_shipdate, l_quantity)
)
with (orientation = column)
distribute by hash(L_ORDERKEY);
```

重新导入数据后执行查询，查看执行时间：

图 5-7 使用 partial cluster key

```
id |                operation                |      A-time       | A-rows | E-rows |  Peak Memory   | A-width | E-width |  E-costs
---+-----------------------------------------+-------------------+--------+--------+----------------+---------+---------+----------
 1 | ->  Row Adapter                         | 459.539           |      1 |      1 | 12KB           |         |      44 | 205693.85
 2 |    ->  Vector Aggregate                 | 459.528           |      1 |      1 | 184KB          |         |      44 | 205693.85
 3 |       ->  Vector Streaming (type: GATHER) | 459.452         |      1 |      1 | 174KB          |         |      44 | 205693.85
 4 |          ->  Vector Aggregate           | [285.177,285.177] |      1 |      1 | [225KB, 225KB] |         |      44 | 205693.79
 5 |             ->  CStore Scan on public.lineitem | [249.757,249.757] | 114160 |  89475 | [792KB, 792KB] |         |      12 | 205246.40
(5 rows)
```

图 5-8 使用 partial cluster key 后 CU 加载情况

```
  5 --CStore Scan on public.lineitem
        datanode1 (actual time=23.017..249.757 rows=114160 loops=1)
        datanode1 (RoughCheck CU: CUNone: 84, CUSome: 17)
        datanode1 (LLVM Optimized)
        datanode1 (Buffers: shared hit=2853 read=23)
        datanode1 (CPU: ex c/r=5673, ex cyc=647656146, inc cyc=647656146)
```

使用partial cluster key后，5-- CStore Scan on public.lineitem的时间减少了1.2s，得益于有84个CU被过滤掉了。

优化建议

- 选取partial cluster key列。
  - 列存表支持创建partial cluster key的类型character varying(n), varchar(n), character(n), char(n), text, nvarchar2, timestamp with time zone, timestamp without time zone, date, time without time zone, time with time zone。
  - 数据的distinct值数量较少，这样能产生较好的过滤效果。
  - 出现在查询where条件中，优先选取能过滤大量数据的列。
  - partial cluster key中设置多个列时，是先按第一个列排序，当第一个列值相同时，使用第二列比较，后续列依次类推。推荐不要超出3个列。
- 添加partial cluster key后，优化导入性能。

  由于添加了partial cluster key，在导入时会增加排序计算，会对导入性能产生影响。当排序完全在内存中进行时影响较小，如果无法在内存中完成排序时，会下盘写临时文件，这时就会产生较大的影响。

  排序使用的内存通过GUC参数psort_work_mem来设置，可以设置较大的值来使用更大的内存进行排序。

  排序的数据量是通过表的存储参数PARTIAL_CLUSTER_ROWS来设置，降低这个数值，可减少一次排序的数据量。这个参数通常与存储参数MAX_BATCHROW配置使用。PARTIAL_CLUSTER_ROWS设置值必须是MAX_BATCHROW的整数倍，MAX_BATCHROW是设置单个CU中数据的最大行数。

5.15 案例：NOT IN 转 NOT EXISTS

NOT IN语句需要使用nestloop anti join来实现，而NOT EXISTS则可以通过hash anti join来实现。在join列不存在null值的情况下，not exists和not in等价。因此在确保没有null值时，可以通过将not in转换为not exists，通过生成hash join来提升查询效率。

优化前

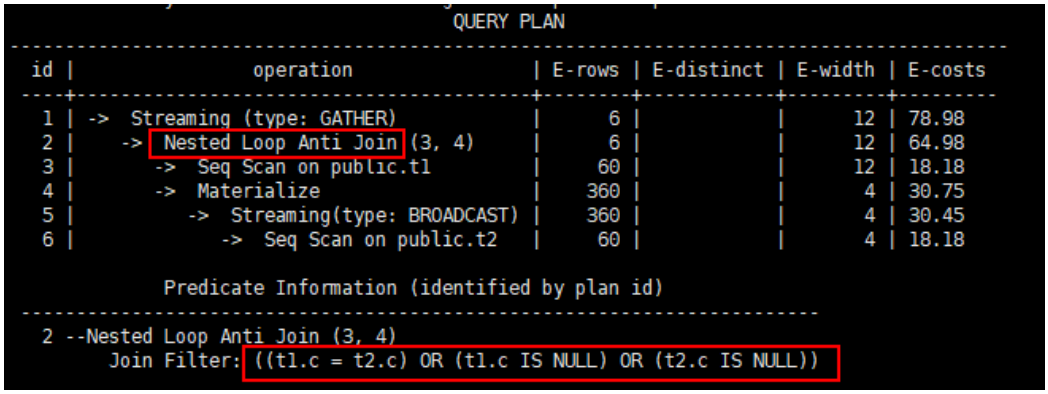
创建两个基表t1、t2：


```
CREATE TABLE t1(a int, b int, c int not null) WITH(orientation=row);
CREATE TABLE t2(a int, b int, c int not null) WITH(orientation=row);
```

执行下列SQL语句，查询NOT IN的执行计划：

```
EXPLAIN VERBOSE SELECT * FROM t1 WHERE t1.c NOT IN (SELECT t2.c FROM t2);
```

返回结果如图：



```
                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 id |               operation                | E-rows | E-distinct | E-width | E-costs
----+----------------------------------------+--------+------------+---------+---------
  1 | ->  Streaming (type: GATHER)           |      6 |            |      12 | 78.98
  2 |    ->  Nested Loop Anti Join (3, 4)    |      6 |            |      12 | 64.98
  3 |       ->  Seq Scan on public.t1        |     60 |            |      12 | 18.18
  4 |       ->  Materialize                  |    360 |            |       4 | 30.75
  5 |          ->  Streaming(type: BROADCAST)|    360 |            |       4 | 30.45
  6 |             ->  Seq Scan on public.t2  |     60 |            |       4 | 18.18

            Predicate Information (identified by plan id)
-------------------------------------------------------------------------
  2 --Nested Loop Anti Join (3, 4)
        Join Filter: ((t1.c = t2.c) OR (t1.c IS NULL) OR (t2.c IS NULL))
```

从返回结果可知执行计划走NestLoop，因为NULL值跟任意值的OR运算结果都是NULL，WHERE条件表达式：

```
t1.c NOT IN (SELECT t2.c FROM t2)
```

等价于：

```
t1.c <> ANY(t2.c) AND t1.c IS NOT NULL AND ANY(t2.c) IS NOT NULL
```

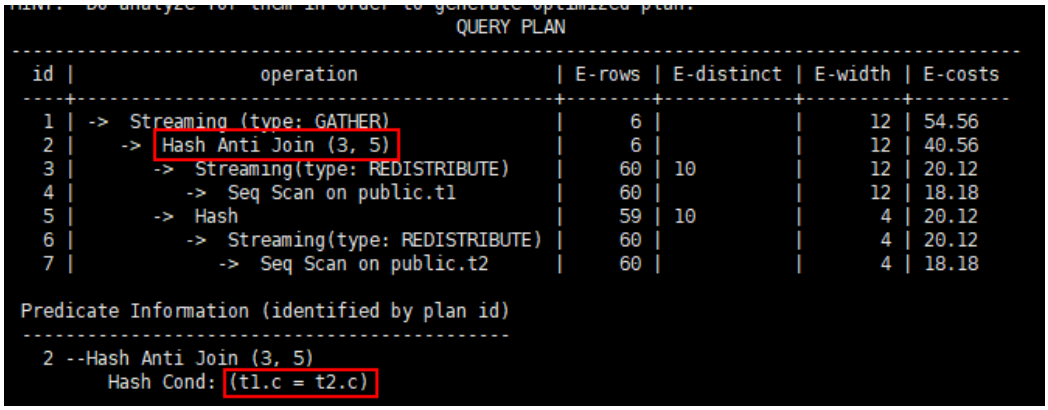
优化后

可将查询可以修改为：

```
SELECT * FROM t1 WHERE NOT EXISTS (SELECT * FROM t2 WHERE t2.c = t1.c);
```

执行下列语句，查询NOT EXISTS的执行计划：

```
EXPLAIN VERBOSE SELECT * FROM t1 WHERE NOT EXISTS (SELECT 1 FROM t2 WHERE t2.c = t1.c);
```



```
                                  QUERY PLAN
-------------------------------------------------------------------------------------------
 id |               operation                | E-rows | E-distinct | E-width | E-costs
----+----------------------------------------+--------+------------+---------+---------
  1 | ->  Streaming (type: GATHER)           |      6 |            |      12 | 54.56
  2 |    ->  Hash Anti Join (3, 5)           |      6 |            |      12 | 40.56
  3 |       ->  Streaming(type: REDISTRIBUTE)|     60 | 10         |      12 | 20.12
  4 |          ->  Seq Scan on public.t1     |     60 |            |      12 | 18.18
  5 |       ->  Hash                         |     59 | 10         |       4 | 20.12
  6 |          ->  Streaming(type: REDISTRIBUTE)|  60 |            |       4 | 20.12
  7 |             ->  Seq Scan on public.t2  |     60 |            |       4 | 18.18

Predicate Information (identified by plan id)
-----------------------------------------------
  2 --Hash Anti Join (3, 5)
        Hash Cond: (t1.c = t2.c)
```


6 SQL 执行 troubleshooting

6.1 分析查询效率异常降低的问题

通常在几十毫秒内完成的查询，有时会突然需要几秒的时间完成；而通常需要几秒完成的查询，有时需要半小时才能完成。如何分析这种查询效率异常降低的问题呢?

处理步骤

通过下列的操作步骤，可以分析出查询效率异常降低的原因。

步骤1 使用ANALYZE命令分析数据库。

ANALYZ命令更新所有表中数据大小以及属性等相关统计信息，该命令为轻量级，可以经常执行。如果此命令执行后性能恢复或者有所提升，则表明autovacuum未能很好的完成它的工作，有待进一步分析。

步骤2 检查查询语句是否返回了多余的数据信息。

例如，如果查询语句先查询一个表中所有的记录，而只用到结果中的前10条记录。对于包含50条记录的表，查询起来是很快的；但是，当表中包含的记录达到50000条，查询效率将会有所下降。

若业务应用中存在只需要部分数据信息，但是查询语句却是返回所有信息的情况，建议修改查询语句，增加LIMIT子句来限制返回的记录数。这样至少使数据库优化器有了一定的优化空间，一定程度上会提升查询效率。

步骤3 检查查询语句单独运行时是否仍然较慢。

尝试在数据库没有其他查询或查询较少的时候运行查询语句，并观察运行效率。如果效率较高，则说明可能是由于之前运行数据库系统的主机负载过大导致查询低效。此外，还可能是执行计划比较低效，但是由于主机硬件较快使得查询效率较高。

步骤4 检查相同查询语句重复执行的效率。

查询效率低的一个重要原因是查询所需信息没有缓存在内存中，这可能是由于内存资源紧张，缓存信息被其他查询处理覆盖。

重复执行相同的查询语句，如果后续执行的查询语句效率提升，则可能是由于上述原因导致。

----结束

6.2 不同用户查询同表显示数据不同

问题现象

2个用户登录相同数据库human_resource，分别执行的查询语句如下：SELECT count(*) FROM areas，查询同一张表areas时，查询结果却不一致。

原因分析

请先判断同名的表是否确实是同一张表。在关系型数据库中，确定一张表通常需要3个因素：database，schema，table。从问题现象描述看，database，table已经确定，分别是human_resource、areas。接着，需要检查schema。使用dbadmin，user01分别登录发现，search_path依次是public和"$user"。dbadmin作为集群管理员，默认不会创建dbadmin同名的schema，即不指定schema的情况下所有表都会建在public下。而对于普通用户如user01，则会在创建用户时，默认创建同名的schema，即不指定schema时表都会创建在user01的schema下。最终确定该案例发生时，确实因为2个用户之间交错对表进行操作，导致了同名不同表的情况。

处理方法

在操作表时加上schema引用，格式：schema.table。

6.3 业务运行时整数转换错误

问题现象

在转换整数时报出以下错误：

```
Invalid input syntax for integer: "13."
```

原因分析

有些数据类型不能转换成目标数据类型。

处理办法

逐步缩小SQL范围来确定。

6.4 SQL 语句出错自动重试

GaussDB(DWS)支持在SQL语句执行出错时自动重试（下文简称CN Retry）。对于来自gsql客户端、JDBC、ODBC驱动的SQL语句，在SQL语句执行失败时，CN端能够自动识别语句执行过程中的报错，并重新下发任务进行自动重试。

该功能的限制和约束如下：

- 功能范围限制：
  - 仅能提高故障发生时SQL语句执行成功率，不能保证100%的执行成功。

  - CN Retry默认开启，开启后temp表会记录日志，关闭CN Retry后，temp表不会记录日志，因此不能在使用temp表时反复打开/关闭CN Retry开关，否则主备切换后再CN Retry会造成数据不一致。
  - CN Retry默认开启，开启后新创建的unlogged表会忽视unlogged关键字，建成普通表。关闭CN Retry后，unlogged表不会记录日志，因此不能在使用unlogged表时反复打开/关闭CN Retry开关，否则主备切换后再CN Retry会造成数据不一致。
  - 在使用gds进行数据导出时，支持CN Retry。现有机制导出时会对重复文件进行检测并删除相同的文件，因此建议不要对相同的外表重复导出数据，除非确定数据目录中相同文件名的文件需要删除。

- 错误类型约束：

  SQL语句出错时能够被识别和重试的错误，仅限在错误类型列表（请参考**表6-1**）中定义的错误。

- 语句类型约束：

  支持单语句CN Retry、存储过程、函数、匿名块。不支持事务块中的语句。

- 存储过程语句约束：
  - 包含EXCEPTION的存储过程，如果在执行过程中（包含语句块执行和EXCEPTION中的语句执行）错误被抛出，可以retry，且系统内部错误发生时，retry会先于EXCEPTION被执行，而如果报错被EXCEPTION捕获则不能retry。
  - 不支持使用全局变量的package。
  - 不支持DBMS_JOB。
  - 不支持UTL_FILE。
  - 如果存储过程中有输出打印信息（如dbms_output.put_line或raise info等），则发生retry时会重复输出已打印的消息，并会在重复消息前输出“Notice：Retry triggered, some message may be duplicated.”加以提示。
- 集群状态约束：
  - 仅支持DN、GTM实例故障。
  - CN Retry有次数限制，如果在CN Retry达到最大尝试次数（最大次数由max_query_retry_times控制）之前，集群状态无法从故障状态恢复到正常状态，那么CN Retry不能保证执行成功。
  - 扩容时不支持CN Retry。
- 数据导入约束：
  - 不支持COPY FROM STDIN语句。
  - 不支持gsql \copy from元命令。
  - 不支持JDBC CopyManager copyIn导入数据。

CN Retry支持的错误类型列表和对应的错误码信息见**表6-1**， 可以通过GUC参数retry_ecode_list设置CN Retry支持的错误类型列表，但不建议用户直接修改该参数，如有修改需求请联系技术工程师协助处理。

表 6-1 CN Retry 支持的错误类型列表

错误类型	错误码	备注
对端连接重置（CONNECTION_RESET_BY_PEER）	YY001	TCP通信错误：Connection reset by peer（CN和DN间通信）
对端流重置（STREAM_CONNECTION_RESET_BY_PEER）	YY002	TCP通信错误：Stream connection reset by peer（DN和DN间通信）
锁等待超时（LOCK_WAIT_TIMEOUT）	YY003	锁超时，Lock wait timeout
连接超时（CONNECTION_TIMED_OUT）	YY004	TCP通信错误，Connection timed out
查询设置错误（SET_QUERY_ERROR）	YY005	SET命令发送失败，Set query
超出逻辑内存（OUT_OF_LOGICAL_MEMORY）	YY006	内存申请失败，Out of logical memory
通信库内存分配（SCTP_MEMORY_ALLOC）	YY007	SCTP通信错误，Memory allocate error
无通信库缓存数据（SCTP_NO_DATA_IN_BUFFER）	YY008	SCTP通信错误，SCTP no data in buffer
通信库释放内存关闭（SCTP_RELEASE_MEMORY_CLOSE）	YY009	SCTP通信错误，Release memory close
SCTP、TCP断开（SCTP_TCP_DISCONNECT）	YY010	SCTP通信错误，TCP disconnect
通信库断开（SCTP_DISCONNECT）	YY011	SCTP通信错误，SCTP disconnect
通信库远程关闭（SCTP_REMOTE_CLOSE）	YY012	SCTP通信错误，Stream closed by remote
等待未知通信库通信（SCTP_WAIT_POLL_UNKNOW）	YY013	等待未知通信库通信，SCTP wait poll unknow
无效快照（SNAPSHOT_INVALID）	YY014	快照非法，Snapshot invalid
通讯接收信息错误（ERRCODE_CONNECTION_RECEIVE_WRONG）	YY015	连接获取错误，Connection receive wrong
内存耗尽（OUT_OF_MEMORY）	53200	内存耗尽，Out of memory

错误类型	错误码	备注
连接失败（CONNECTION_FAILURE）	08006	GTM出错，Connection failure
连接异常（CONNECTION_EXCEPTION）	08000	连接出现错误，和DN的通讯失败，Connection exception
管理员关闭系统（ADMIN_SHUTDOWN）	57P01	管理员关闭系统，Admin shutdown
关闭远程流接口（STREAM_REMOTE_CLOSE_SOCKET）	XX003	关闭远程套接字，Stream remote close socket
重复查询编号（ERRCODE_STREAM_DUPLICATE_QUERY_ID）	XX009	重复查询，Duplicate query id
stream查询并发更新同一行（ERRCODE_STREAM_CONCURRENT_UPDATE）	YY016	stream查询并发更新同一行，Stream concurrent update
LLVM内存分配错误（ERRCODE_LLVM_BAD_ALLOC_ERROR ）	CG003	内存分配错误， Allocate error
LLVM致命错误（ERRCODE_LLVM_FATAL_ERROR ）	CG004	致命错误，Fatal error
HashJoin临时文件读取错误(ERRCODE_HASHJOIN_TEMP_FILE_ERROR)	F0011	临时文件读取错误，File error
Buffer文件读取错误(ERRCODE_BUFFER_FILE_ERROR)	F0012	文件读取错误，File error
分区个数发生变化(ERRCODE_PARTITION_NUM_CHANGED)	45003	在扫描LIST分区表时，发现此时的分区个数和优化阶段的分区个数不一致，一般出现在查询和ADD/DROP分区并发时。（此错误类型仅8.1.3及以上集群版本支持）
节点间对象SCHEMA名称不一致(ERRCODE_UNMATCH_OBJECT_SCHEMA)	42P30	对象SCHEMA名称不一致，Unmatched schema name

开启CN Retry功能需要设置如下GUC参数：

- 必选的GUC参数（CN和DN都需设置）
  max_query_retry_times

⚠ 注意

CN Retry功能开启时会为临时表数据记录日志，为保证数据一致性，在使用临时表时不能切换CN Retry开关状态，保持使用临时表的会话中CN Retry开关始终处于打开状态或者关闭状态。

- 可选的GUC参数

  cn_send_buffer_size

  max_cn_temp_file_size

7 query_band 负载识别

概述

GaussDB(DWS)实现基于query_band的负载识别和队列内优先级控制，一方面提供了更为灵活的负载识别手段，可根据作业类型、应用名称、脚本名称等识别负载队列，使用户根据业务场景可灵活配置query_band识别队列；另一方面实现了队列内作业下发优先级控制，后续将逐步实现队列内资源优先级控制。

管理员用户可根据业务场景及作业类别配置query_band所关联队列及估算内存限制等实现更为灵活的负载控制与资源管控。如果业务未配置query_band或用户未将query_band关联行为时，作业会默认使用用户关联队列和队列内优先级。

query_band 支持的负载行为

query_band是一个session级别的GUC参数，作为作业标识符本身没有特殊含义，数据类型为字符型，支持赋值任何字符串。但是为方便区分和设置，query_band负载识别仅支持识别键值对形式的query_band，示例：

```
SET query_band='JobName=abc;AppName=test;UserName=user';
```

其中，‘JobName=abc’，‘AppName=test’以及‘UserName=user’都是一个独立的键值对。query_band键值对规格：

- query_band使用键值对方式设置，即'key=value'；session内支持设置多个query_band键值对，多个键值对之间使用分号分隔。query_band键值对和query_band参数值长度的上限均为1024个字符。
- query_band键值对支持的有效字符包括：数字0~9、大写字母A~Z、小写字母a-z、'.'、'-'、'_' 以及'#'。

query_band负载识别以键值对为单位设置和识别负载行为，目前支持的负载行为，如表7-1所示：

表 7-1 QUERY_BAND 支持负载行为

类别	行为	行为表现
负载管理(workload)	资源池(respool)	query_band关联资源池

类别	行为	行为表现
负载管理(workload)	优先级(priority)	队列内优先级
次序(order)	队列（respool） 目前为无效字段，主要用于后续扩展。	query_band搜索次序

其中，行为类别用于负载行为归类，不同的负载行为可能属于同一个类别，比如资源池和优先级同属于负载管理类别；负载行为代表query_band键值对关联的是哪种负载行为；行为表现用于记录具体的负载行为是什么；负载类别中的次序用于标记query_band负载行为识别的优先级，一个session内设置多个query_band键值对时，优先使用次序较小的query_band键值对识别负载行为。每一个query_band键值对都可以对应0个或多个负载行为，但是一种负载行为只能关联一个。query_band负载行为详细说明如下：

- 资源池：query_band支持关联资源池，作业执行时，若query_band指定了资源池，则使用query_band关联的资源池，否则使用用户关联的资源池。
  - query_band关联资源池时，资源池不存在报错退出，关联失败。
  - query_band关联资源池时，记录query_band与资源池依赖关系。
  - query_band关联资源池删除时，提示有query_band依赖，资源池删除失败。
- 队列内优先级：query_band支持关联作业优先级，支持高中低(High/Medium/Low)三个优先级，同时提供Rush作为特殊优先级（绿色通道），默认优先级为Medium。正常实践过程中，大部分作业使用Medium优先级，优先级较低作业使用Low优先级，特权作业使用High优先级，High作业不建议过多。Rush优先级作为特殊场景下应急使用，平时不建议使用。

  队列内优先级实现队列内排队优先级：
  - 静态负载管理场景下，CN并发不足时，触发CN全局队列排队，CN全局队列为优先级队列。
  - 动态负载管理场景下，DN内存不足时，触发CCN全局排队，CCN全局队列为优先级队列。
  - 资源池并发或内存不足时，触发资源池排队，资源池队列为优先级队列。

  以上优先级队列均遵守以下调度规则：
  - 优先级高作业优先调度。
  - 优先级高作业全部调度完之后调度优先级低作业。
  - 动态负载管理场景下，CN全局队列不支持query_band优先级。
- 次序：支持设置query_band识别次序，未设置识别次序的使用默认次序-1。除默认次序外，不存在次序相同的两个query_band。设置次序时对query_band次序进行校验，存在相同次序时，已存在次序递归+1直到不存在相同次序为止。
  - session内设置多个query_band键值对时，使用次序较小的query_band键值对作为负载识别的query_band。
  - 次序最小为0，默认次序-1为最大次序。
  - 次序都为默认次序时，使用设置靠前的query_band作为负载识别的query_band。

  - 示例：set query_band='b=1;a=3;c=1'; b=1，其中b=1次序-1，a=3次序4，c=1次序1，则使用c=1作为负载识别的query_band，此设计可提供负载管理员对负载调度调整能力。

query_band 应用与配置

- pg_workload_action跨库系统表用于存储query_band行为和次序，详见**PG_WORKLOAD_ACTION**。
- 默认行为和次序在系统表pg_workload_action不存储，query_band有设置非默认行为的，查询其行为默认行为也显示；查询行为和次序都为默认的query_band行为时，显示<query_band information not found>。
- gs_wlm_set_queryband_action函数用于设置query_band行为：其中第一个参数即queryband键值对的长度上限为63个字符；第二个参数action不区分大小写，多个action使用分号分隔；order为缺省参数，默认为-1。具体请参见**gs_wlm_set_queryband_action**。
- gs_wlm_set_queryband_order函数设置query_band次序：其中第一个参数即queryband键值对的长度上限为63个字符；query_band次序必须大于等于-1，除默认次序-1外不存在两个次序相同的query_band。设置query_band次序时如果存在相同次序的query_band，则将原query_band次序+1。具体请参见**gs_wlm_set_queryband_order**。
- gs_wlm_get_queryband_action函数用于查询query_band行为，具体请参见**gs_wlm_set_queryband_action**。
- pg_queryband_action系统视图用于查询所有query_band行为，具体请参见**PG_QUERYBAND_ACTION** 。
- query_band优先级在负载管理视图**PG_SESSION_WLMSTAT** ）中显示为int型，数字和优先级的对应关系如下：
  - 0：表示该作业不受负载管理管控；
  - 1：LOW；
  - 2：MEDIUM；
  - 4：HIGH；
  - 8：RUSH；
- 权限控制：除初始用户外，被授权用户才具有设置和查询query_band权限。

说明

批量取消所有运行作业或队列并发上限是1且只有一个队列有作业运行的情况下，可能会触发CN自动唤醒作业导致作业不按照优先级下发。

示例

步骤1 设置query_band“JobName=abc”关联资源池p1、队列内优先级Rush、次序为1。

```
SELECT * FROM gs_wlm_set_queryband_action('JobName=abc','respool=p1;priority=rush',1);
gs_wlm_set_queryband_action
----------------------------
 t
(1 row)
```

步骤2 修改query_band“JobName=abc”的关联资源池为p2。

```
SELECT * FROM gs_wlm_set_queryband_action('JobName=abc','respool=p2');
gs_wlm_set_queryband_action
----------------------------
```



```
 t
(1 row)
```

步骤3 修改query_band“JobName=abc”的队列内优先级为High。

```
SELECT * FROM gs_wlm_set_queryband_action('JobName=abc','priority=high');
gs_wlm_set_queryband_action
-----------------------------
 t
(1 row)
```

步骤4 修改query_band“JobName=abc”的次序为3。

```
SELECT * FROM gs_wlm_set_queryband_order('JobName=abc',3);
gs_wlm_set_queryband_order
-----------------------------
 t
(1 row)
```

步骤5 查询query_band关联的负载行为。

```
SELECT * FROM pg_queryband_action;
    qband      | respool_id | respool | priority | qborder
---------------+------------+---------+----------+---------
 AppName=test |      16974 | p1      | low      |      -1
 JobName=abc  |      17119 | p2      | high     |       1
(2 rows)
```

----结束

8 常见性能参数调优设计

在使用数据库的时候，为了提高集群的性能，有多种方式去调优，从硬件配置到软件驱动升级，再到数据库的内部参数调整。本章节旨在介绍一些常用参数以及推荐配置。

1. query_dop

   设置用户自定义的查询并行度。

   SMP架构是一种利用富余资源来换取时间的方案，计划并行之后必定会引起资源消耗的增加，包括CPU、内存、I/O和网络带宽等资源的消耗都会出现明显的增长，而且随着并行度的增大，资源消耗也随之增大。

   - 当资源成为瓶颈的情况下，SMP无法提升性能，反而可能导致性能的劣化。在出现资源瓶颈的情况下，建议关闭SMP。
   - 当资源许可的情况下，并行度越高，性能提升效果越好。

   SMP并行度支持会话级设置，推荐在执行符合要求的查询前，打开SMP，执行结束后，关闭SMP。以免在业务峰值时，对业务造成冲击。

   可通过set query_dop=10，在会话中打开SMP。

2. enable_dynamic_workload

   开启动态负载管理。动态负载管理指数据库内部根据用户负载情况，自动对复杂查询进行队列控制，不再需要手动设置参数，做到系统参数免调优。

   该参数默认打开，需要注意以下几点：

   - 集群有一个CN会作为中心协调节点（CCN），用于收集和调度作业执行，Central Coordinator State会显示其状态。当CCN不存在时，作业不再受动态负载管理控制。
   - 简单查询作业（估算值<32MB）、非DML（即非INSERT、UPDATE、DELETE和SELECT）语句，不走自适应负载，需要通过max_active_statements来进行单CN的上限控制。
   - 在自适应负载特性下，参数work_mem的默认数值不能变大，否则会引起内存不受控（例如未做Analyze的语句）。
   - 以下场景或语句由于内存使用的特殊性和不确定性，可能导致大并发场景内存不受控，遇到需要降低并发数：
     - 单条元组占用内存过大，例如，基表包含超过MB级别的宽列。
     - 完全下推语句的查询。

- 需要在CN上耗费大量内存的语句，例如，不能下推的语句、withhold cursor场景。
- 由于计划生成不当导致hashjoin算子建立的hash表重复值过多，占用大量内存。
- 包含UDF，且UDF中使用大量内存。

该参数可配合query_dop=0使用，当query_dop设置为0（自适应），系统会根据资源情况和计划特征，动态为每个查询选取[1,8]之间的最优的并行度。enable_dynamic_workload参数会动态分配内存。

3. max_active_statements

   设置全局的最大并发数量。此参数只应用到CN，且针对一个CN上的执行作业。

   需根据系统资源（如CPU资源、IO资源和内存资源）情况，调整此数值大小，使得系统支持最大限度的并发作业，且防止并发执行作业过多，引起系统崩溃。

   - 当取值-1或者0时，不限制全局并发数。
   - 在点查询的场景下，参数建议设置为100。
   - 在分析类查询的场景下，参数的值设置为CPU的核数除以DN个数，一般可以设置5~8个。

4. session_timeout

   缺省情况下，客户端连接数据库后处于空闲状态时会根据参数的默认值自动断开连接。

   取值范围：整型，0-86400，最小单位为秒（s），0表示关闭超时设置。一般不建议设置为0。

5. 影响数据库内存的五大参数:

   max_process_memory、shared_buffers、cstore_buffers、work_mem和maintenance_work_mem。

   - max_process_memory

     max_process_memory是逻辑内存管理参数，主要功能是控制单个CN/DN上可用内存的最大峰值。

     非从备DN节点自动适配，公式为（物理内存大小）* 0.8 / (1+主DN个数)，当结果不足2GB时，默认取2GB。从备DN默认为12GB。

   - shared_buffers

     设置DWS使用的共享内存大小。增加此参数的值会使DWS比系统默认设置需要更多的System V共享内存。

     建议设置shared_buffers值为内存的40%以内。主要用于行存表scan。计算公式：shared_buffers=(单服务器内存/单服务器DN个数)*0.4*0.25

   - cstore_buffers

     设置列存和OBS、HDFS外表列存格式（orc、parquet、carbondata）所使用的共享缓冲区的大小。

     列存表使用cstore_buffers设置的共享缓冲区，几乎不用shared_buffers。因此在列存表为主的场景中，应减少shared_buffers，增加cstore_buffers。

     OBS、HDFS外表使用cstore_buffers设置ORC、Parquet、Carbondata的元数据和数据的缓存，元数据缓存大小为cstore_buffers的1/4，最大不超过2GB，其余缓存空间为列存数据和外表列存格式数据共享使用。

   - work_mem

     设置内部排序操作和Hash表在开始写入临时磁盘文件之前使用的内存大小。

ORDER BY，DISTINCT和merge joins都要用到排序操作。Hash表在散列连接、散列为基础的聚集、散列为基础的IN子查询处理中都要用到。

对于复杂的查询，可能会同时并发运行好几个排序或者散列操作，每个都可以使用此参数所声明的内存量，不足时会使用临时文件。同样，好几个正在运行的会话可能会同时进行排序操作。因此使用的总内存可能是work_mem的好几倍。

计算公式：

对于串行无并发的复杂查询场景，平均每个查询有5-10关联操作，建议work_mem=50%内存/10。

对于串行无并发的简单查询场景，平均每个查询有2-5个关联操作，建议work_mem=50%内存/5。

对于并发场景，建议work_mem=串行下的work_mem/物理并发数。

- maintenance_work_mem

  设置维护性操作（比如VACUUM、CREATE INDEX、ALTER TABLE ADD FOREIGN KEY等）中可使用的最大的内存。

  设置建议：

  建议设置此参数的值大于work_mem，可以改进清理和恢复数据库转储的速度。因为在一个数据库会话里，任意时刻只有一个维护性操作可以执行，并且在执行维护性操作时不会有太多的会话。

  当自动清理进程运行时，autovacuum_max_workers倍数的内存将会被分配，所以此时设置maintenance_work_mem的值应该不小于work_mem。

6. bulk_write_ring_size

   数据并行导入使用的环形缓冲区大小。

   该参数主要影响入库性能，建议导入压力大的场景增加DN上的该参数配置。

7. 影响数据库连接的两大参数:

   max_connections和max_prepared_transactions

   - max_connections

     允许和数据库连接的最大并发连接数。此参数会影响集群的并发能力。

     设置建议：

     CN中此参数建议保持默认值。DN中此参数建议设置为CN的个数乘以CN中此参数的值。

     增大这个参数可能导致GaussDB(DWS)要求更多的System V共享内存或者信号量，可能超过操作系统缺省配置的最大值。这种情况下，请酌情对数值加以调整。

   - max_prepared_transactions

     设置可以同时处于"预备"状态的事务的最大数目。增加此参数的值会使GaussDB(DWS)比系统默认设置需要更多的System V共享内存。

   须知

   max_connections取值的设置受max_prepared_transactions的影响，在设置max_connections之前，应确保max_prepared_transactions的值大于或等于max_connections的值，这样可确保每个会话都有一个等待中的预备事务。

8. checkpoint_completion_target

指定检查点完成的目标。

含义是每个checkpoint需要在checkpoints间隔时间的50%内完成。

默认值为0.5，为提高性能可改成0.9。

9. data_replicate_buffer_size

   发送端与接收端传递数据页时，队列占用内存的大小。此参数会影响主备之间复制的缓冲大小。

   默认值为128MB，若服务器内存为256G，可适当增大到512MB。

