Data Warehouse Service

Query Performance Optimization

Issue 10

Date 2024-04-26

HUAWEI CLOUD COMPUTING TECHNOLOGIES CO., LTD.

Contents

1 Overview of Query Performance Optimization

Performance optimization is a key step in database application development and migration and is a large part in the entire project implementation process. Performance optimization improves database resource utilization, reduces service costs, reduces application system running risks, improves system stability, and brings more values to customers.

The aim of SQL optimization is to maximize the utilization of resources, including CPU, memory, disk I/O, and network I/O. To maximize resource utilization is to run SQL statements as efficiently as possible to achieve the highest performance at a lower cost. For example, when performing a typical point query, you can use the seqscan and filter (that is, read every tuple and query conditions for match). You can also use an index scan, which can be implemented at a lower cost but achieve the same effect.

This chapter describes the basic database commands **analyze** and **explain** for performance optimization, and describes the explained database execution plans. This helps you understand the database execution process, identify performance bottlenecks, and optimize the database through execution plans. In addition, this document describes performance parameters, typical application scenarios, SQL diagnosis, SQL performance optimization, and SQL rewriting cases, which provide you comprehensive reference for database performance optimization.

2 Query Execution Process

The process from receiving SQL statements to the statement execution by the SQL engine is shown in **Figure 2-1** and **Table 2-1**. The texts in red are steps where database administrators can optimize queries.

Figure 2-1 Execution process of query-related SQL statements by the SQL engine

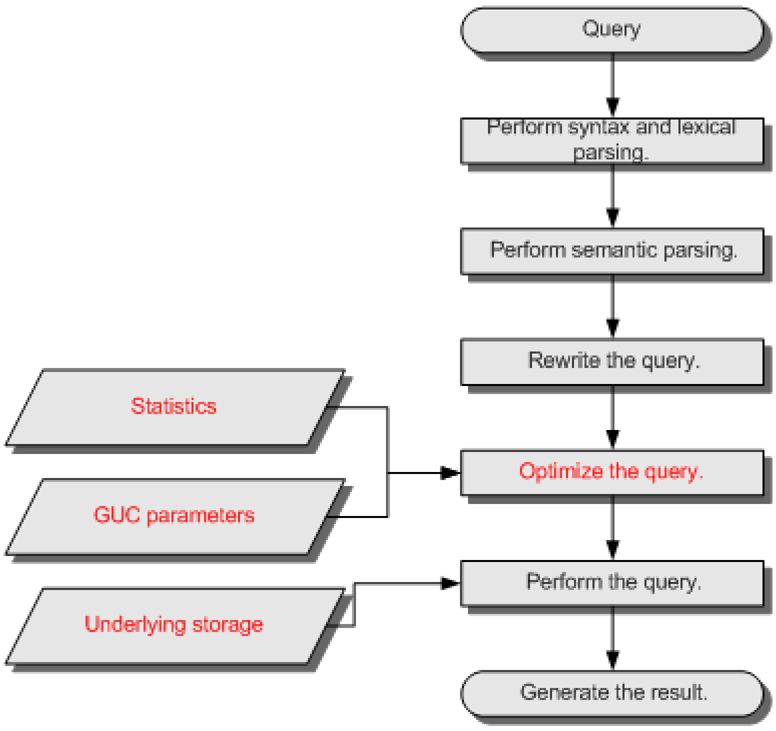


Table 2-1 Execution process of query-related SQL statements by the SQL engine

Procedure	Description
1. Perform syntax and lexical parsing.	Converts the input SQL statements from the string data type to the formatted structure stmt based on the specified SQL statement rules.

Procedure	Description
2. Perform semantic parsing.	Converts the formatted structure obtained from the previous step into objects that can be recognized by the database.
3. Rewrite the query statements.	Converts the output of the last step into the structure that optimizes the query execution.
4. Optimize the query.	Determines the execution mode of SQL statements (the execution plan) based on the result obtained from the last step and the internal database statistics. For details about the impact of statistics and GUC parameters on query optimization (execution plan), see **Optimizing Queries Using Statistics** and **Optimizing Queries Using GUC parameters**.
5. Perform the query.	Executes the SQL statements based on the execution path specified in the last step. Selecting a proper underlying storage mode improves the query execution efficiency. For details, see **Optimizing Queries Using the Underlying Storage**.

Optimizing Queries Using Statistics

The GaussDB(DWS) optimizer is a typical Cost-based Optimization (CBO). By using CBO, the database calculates the number of tuples and the execution cost for each execution step under each execution plan based on the number of table tuples, column width, NULL record ratio, and characteristic values, such as distinct, MCV, and HB values, and certain cost calculation methods. The database then selects the execution plan that takes the lowest cost for the overall execution or for the return of the first tuple. These characteristic values are the statistics, which is the core for optimizing a query. Accurate statistics helps the optimizer select the most appropriate query plan. Generally, you can collect statistics of a table or that of some columns in a table using **ANALYZE**. You are advised to periodically execute **ANALYZE** or execute it immediately after you modified most contents in a table.

Optimizing Queries Using GUC parameters

Optimizing queries aims to select an efficient execution mode.

Take the following statement as an example:

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

During execution of **customer inner join store_sales**, GaussDB(DWS) supports nested loop, merge join, and hash join. The optimizer estimates the result set value and the execution cost under each join mode based on the statistics of the **customer** and **store_sales** tables and selects the execution plan that takes the lowest execution cost.

As described in the preceding content, the execution cost is calculated based on certain methods and statistics. If the actual execution cost cannot be accurately estimated, you need to optimize the execution plan by setting the GUC parameters.

Optimizing Queries Using the Underlying Storage

GaussDB(DWS) supports row- and column-based tables. The selection of an underlying storage mode strongly depends on specific customer business scenarios. You are advised to use column-store tables for computing service scenarios (mainly involving association and aggregation operations) and row-store tables for service scenarios, such as point queries and massive **UPDATE** or **DELETE** executions.

Optimization methods of each storage mode will be described in details in the performance optimization chapter.

Optimizing Queries by Rewriting SQL Statements

Besides the preceding methods that improve the performance of the execution plan generated by the SQL engine, database administrators can also enhance SQL statement performance by rewriting SQL statements while retaining the original service logic based on the execution mechanism of the database and abundant practical experience.

This requires that the system administrators know the customer business well and have professional knowledge of SQL statements.

3 SQL Execution Plan

An SQL execution plan is a node tree that displays the detailed steps performed when the GaussDB(DWS) executes an SQL statement. Each step indicates a database operator, also called an execution operator.

You can run the **EXPLAIN** command to view the execution plan generated for each query by an optimizer. **EXPLAIN** outputs a row of information for each execution node, showing the basic node type and the expense estimate that the optimizer makes for executing the node.

Execution Plan Information

In addition to setting different display formats for an execution plan, you can use different **EXPLAIN** syntax to display execution plan information in detail. The common usages are as follows. For more usages, see **EXPLAIN Syntax**.

- EXPLAIN *statement*: only generates an execution plan and does not execute. The *statement* indicates SQL statements.
- EXPLAIN ANALYZE *statement*: generates and executes an execution plan, and displays the execution summary. Then actual execution time statistics are added to the display, including the total elapsed time expended within each plan node (in milliseconds) and the total number of rows it actually returned.
- EXPLAIN PERFORMANCE *statement*: generates and executes the execution plan, and displays all execution information.

To measure the run time cost of each node in the execution plan, the current execution of **EXPLAIN ANALYZE** or **EXPLAIN PERFORMANCE** adds profiling overhead to query execution. Running **EXPLAIN ANALYZE** or **PERFORMANCE** on a query sometimes takes longer time than executing the query normally. The amount of overhead depends on the nature of the query, as well as the platform being used.

Therefore, if an SQL statement is not finished after being running for a long time, run the **EXPLAIN** statement to view the execution plan and then locate the fault. If the SQL statement has been properly executed, run the **EXPLAIN ANALYZE** or **EXPLAIN PERFORMANCE** statement to check the execution plan and information to locate the fault.

Description of common execution plan keywords:

1. Table access modes
   - Seq Scan/CStore Scan

     Scans all rows of the table in sequence. These are basic scan operators, which are used to scan row-store and column-store tables in sequence.
   - Index Scan/CStore Index Scan

     Scans indexes of row-store and column-store tables. There are indexes in row-store or column-store tables, and the condition column is the index column.

     The optimizer uses a two-step plan: the child plan node visits an index to find the locations of rows matching the index condition, and then the upper plan node actually fetches those rows from the table itself. Fetching rows separately is much more expensive than reading them sequentially, but because not all pages of the table have to be visited, this is still cheaper than a sequential scan. The upper-layer planning node first sort the location of index identifier rows based on physical locations before reading them. This minimizes the independent capturing overhead.

     If there are separate indexes on multiple columns referenced in **WHERE**, the optimizer might choose to use an **AND** or **OR** combination of the indexes. However, this requires the visiting of both indexes, so it is not necessarily a win compared to using just one index and treating the other condition as a filter.

     The following Index scans featured with different sorting mechanisms are involved:

     - Bitmap Index Scan

       To use a bitmap index to capture a data page, you need to scan the index to obtain the bitmap and then scan the base table.

     - Index Scan using index_name

       Fetches table rows in index order, which makes them even more expensive to read. However, there are so few rows that the extra cost of sorting the row locations is unnecessary. This plan type is used mainly for queries fetching just a single row and queries having an **ORDER BY** condition that matches the index order, because no extra sorting step is needed to satisfy **ORDER BY**.
2. Table connection modes
   - Nested Loop

     Nested-loop is used for queries that have a smaller data set connected. In a Nested-loop join, the foreign table drives the internal table and each row returned from the foreign table should have a matching row in the internal table. The returned result set of all queries should not exceed 10,000. The table that returns a smaller subset will work as a foreign table, and indexes are recommended for connection fields of the internal table.
   - (Sonic) Hash Join

     A Hash join is used for large tables. The optimizer uses a hash join, in which rows of one table are entered into an in-memory hash table, after which the other table is scanned and the hash table is probed for matches to each row. Sonic and non-Sonic hash joins differ in their hash table structures, which do not affect the execution result set.

- Merge Join

  In a merge join, data in the two joined tables is sorted by join columns. Then, data is extracted from the two tables to a sorted table for matching.

  Merge join requires more resources for sorting and its performance is lower than that of hash join. If the source data has been sorted, it does not need to be sorted again when merge join is performed. In this case, the performance of merge join is better than that of hash join.

3. Operators
   - sort

     Sorts the result set.
   - filter

     The **EXPLAIN** output shows the **WHERE** clause being applied as a **Filter** condition attached to the **Seq Scan** plan node. This means that the plan node checks the condition for each row it scans, and returns only the ones that meet the condition. The estimated number of output rows has been reduced because of the **WHERE** clause. However, the scan will still have to visit all 10000 rows. As a result, the cost is not decreased. It increases a bit (by 10000 x **cpu_operator_cost**) to reflect the extra CPU time spent on checking the **WHERE** condition.
   - LIMIT

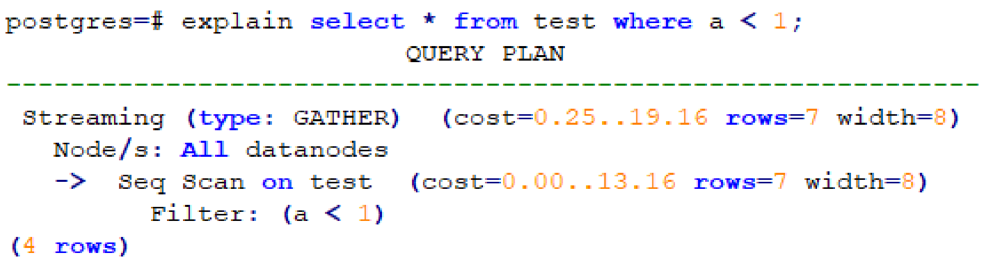
     LIMIT limits the number of output execution results. If a **LIMIT** condition is added, not all rows are retrieved.

Execution Plan Display Format

GaussDB(DWS) provides four display formats: **normal**, **pretty**, **summary**, and **run**. You can change the display format of execution plans by setting **explain_perf_mode**.

- **normal** indicates that the default printing format is used. **Figure 3-1** shows the display format.

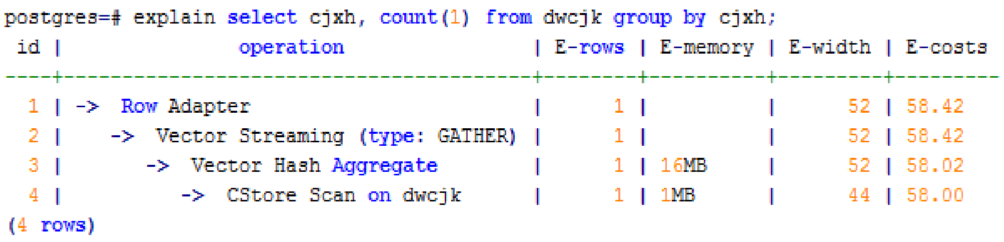
Figure 3-1 Example of an execution plan in normal format



```
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** indicates that the optimized display mode of GaussDB(DWS) is used. A new format contains a plan node ID, directly and effectively analyzing performance. **Figure 3-2** is an example.

Figure 3-2 Example of an execution plan using the pretty format



```
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** indicates that the analysis result based on such information is printed in addition to the printed information in the format specified by **pretty**.
- **run** indicates that in addition to the printed information specified by **summary**, the database exports the information as a CSV file.

Common Types of Plans

GaussDB(DWS) has three types of distributed plans:

- Fast Query Shipping (FQS) plan

  The CN directly delivers statements to DNs. Each DN executes the statements independently and summarizes the execution results on the CN.
- Stream plan

  The CN generates a plan for the statements to be executed and delivers the plan to DNs for execution. During the execution, DNs use the Stream operator to exchange data.
- Remote-Query plan

  After generating a plan, the CN delivers some statements to DNs. Each DN executes the statements independently and sends the execution result to the CN. The CN executes the remaining statements in the plan.

The existing tables **tt01** and **tt02** are defined as follows:

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

Type 1: FQS plan, all statements pushed down

Two tables are joined, and the join condition is the distribution column of each table. If the stream operator is disabled, the CN directly sends statements to each DN for execution. The result is summarized on the CN.

```
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)
```

Type 2: Non-FQS plan, some statements pushed down

Two tables are joined and the join condition contains non-distribution columns. If the stream operator is disabled, the CN delivers the base table scanning statements to each DN. Then, the JOIN operation is performed on the CN.

```
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)
```

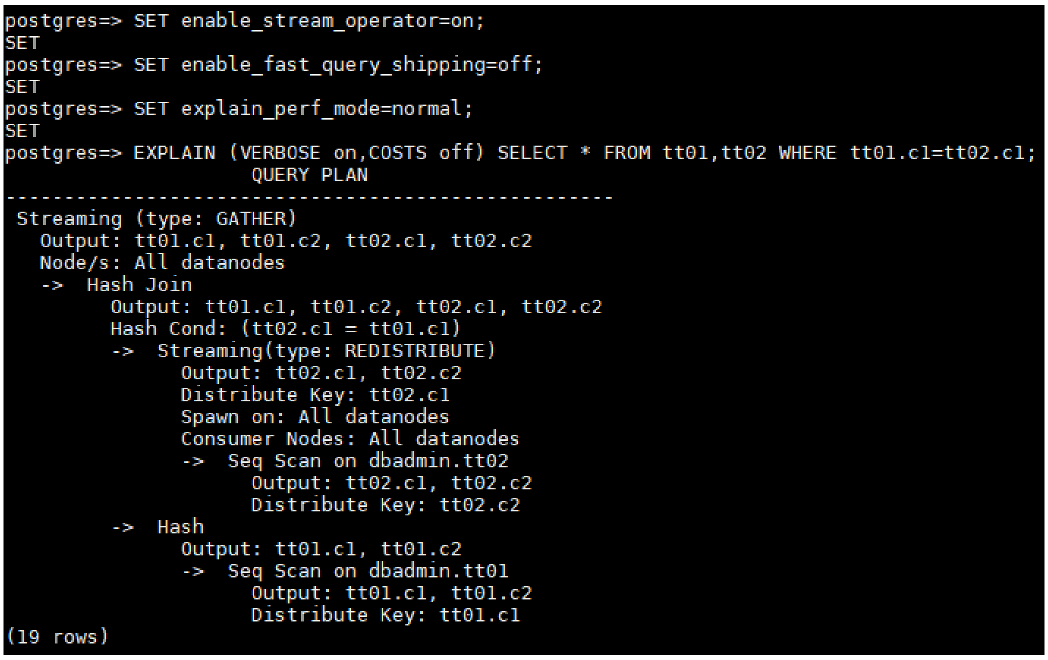
Type 3: Stream plan, no data exchange between DNs

Two tables are joined, and the join condition is the distribution column of each table. DNs do not need to exchange data. After generating a stream plan, the CN delivers the plan except the Gather Stream part to DNs for execution. The CN scans the base table on each DN, performs hash join, and sends the result to the 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)
```

Type 4: Stream plan, with data exchange between DNs

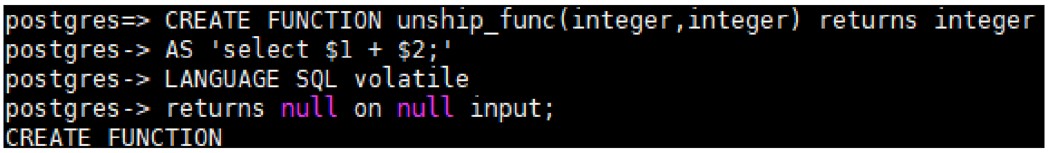
When two tables are joined and the join condition contains non-distribution columns, and the stream operator is enabled (SET enable_stream_operator=on), a stream plan is generated, which allows data exchange between DNs. For table **tt02**, the base table is scanned on each DN. After the scanning, the **Redistribute Stream** operator performs hash calculation based on **tt02.c1** in the **JOIN** condition, sends the hash calculation result to each DN, and then performs JOIN on each DN, finally, the data is summarized to the CN.



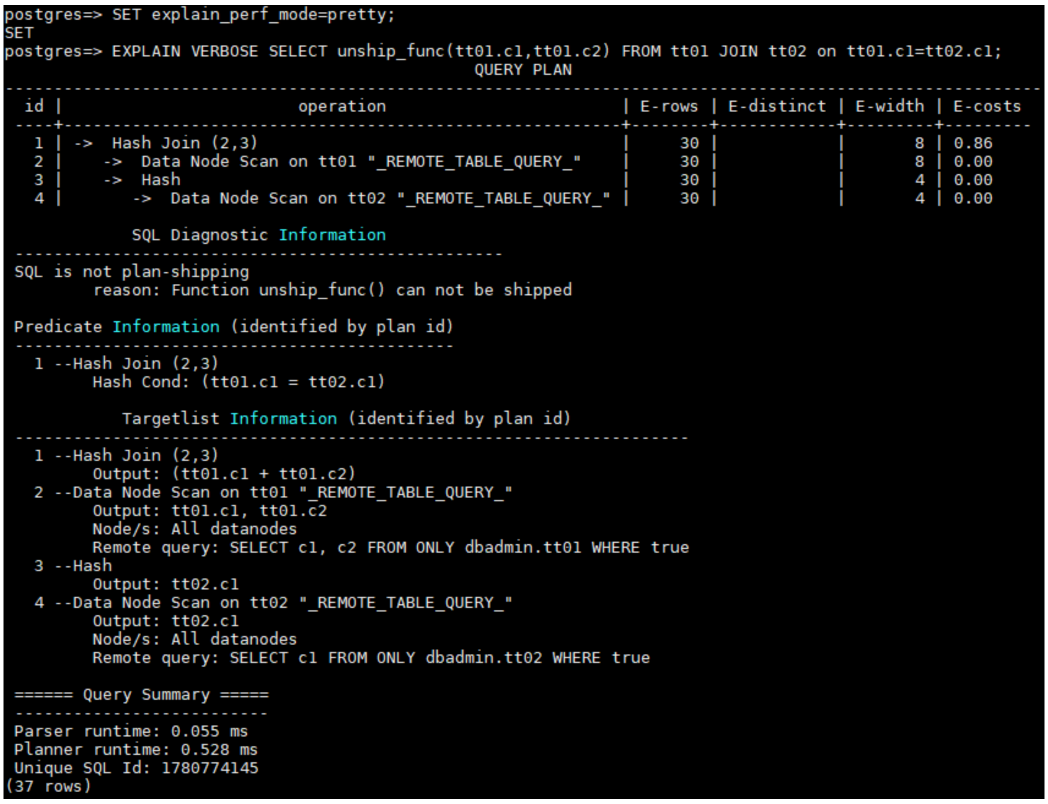
```
postgres=> SET enable_stream_operator=on;
SET
postgres=> SET enable_fast_query_shipping=off;
SET
postgres=> SET explain_perf_mode=normal;
SET
postgres=> EXPLAIN (VERBOSE on,COSTS off) SELECT * FROM tt01,tt02 WHERE tt01.c1=tt02.c1;
                     QUERY PLAN
----------------------------------------------------
 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)
```

Type 5: Remote-Query plan

unship_func cannot be pushed down and does not meet partial pushdown requirements (subquery pushdown). Therefore, you can only send base table scanning statements to DNs and collect base table data to the CN for calculation.



```
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 Description

You can use **EXPLAIN ANALYZE** or **EXPLAIN PERFORMANCE** to check the SQL statement execution information and compare the actual execution and the optimizer's estimation to find what to optimize. **EXPLAIN PERFORMANCE** provides the execution information on each DN, whereas **EXPLAIN ANALYZE** does not.

Tables are defined as follows:

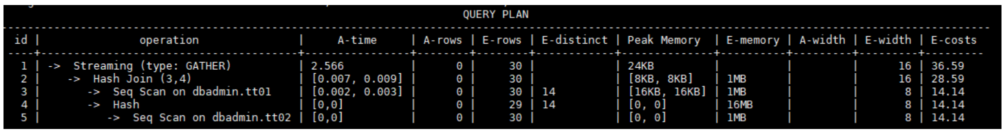
```
CREATE TABLE tt01(c1 int, c2 int) DISTRIBUTE BY hash(c1);
CREATE TABLE tt02(c1 int, c2 int) DISTRIBUTE BY hash(c2);
```

The following SQL query statement is used as an example:

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

The output of EXPLAIN PERFORMANCE consists of the following parts:

1. Execution Plan



```
                                                        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
```

The plan is displayed as a table, which contains 11 columns: **id**, **operation**, **A-time**, **A-rows**, **E-rows**, **E-distinct**, **Peak Memory**, **E-memory**, **A-width**, **E-width**, and **E-costs**. Table 3-1 describes the columns.

Table 3-1 Execution column description

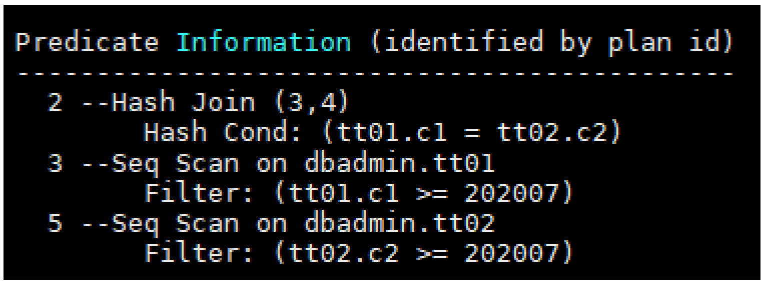
Column	Description
id	ID of an execution operator.
operation	Name of an execution operator. The operator of the Vector prefix refers to a vectorized execution engine operator, which exists in a query containing a column-store table. Streaming is a special operator. It implements the core data shuffle function of the distributed architecture. Streaming has three types, which correspond to different data shuffle functions in the distributed architecture: ● Streaming (type: GATHER): The CN collects data from DNs. ● Streaming(type: REDISTRIBUTE): Data is redistributed to all the DNs based on selected columns. ● Streaming(type: BROADCAST): Data on the current DN is broadcast to all other DNs.
A-time	Execution time of an operator on each DN. Generally, A-time of an operator is two values enclosed by square brackets ([]), indicating the shortest and longest time for completing the operator on all DNs, including the execution time of the lower-layer operators. Note: In the entire plan, the execution time of a leaf node is the execution time of the operator, while the execution time of other operators includes the execution time of its subnodes.
A-rows	Actual rows output by an operator.
E-rows	Estimated rows output by each operator.
E-distinct	Estimated distinct value of the hashjoin operator.
Peak Memory	Peak memory used when the operator is executed on each DN. The left value in [] is the minimum value, and the right value in [] is the maximum value.
E-memory	Estimated memory used by each operator on a DN. Only operators executed on DNs are displayed. In certain scenarios, the memory upper limit enclosed in parentheses will be displayed following the estimated memory usage.
A-width	The actual width of each line of tuple of the current operator. This parameter is valid only for the heavy memory operator is displayed, including: (Vec)HashJoin, (Vec)HashAgg, (Vec) HashSetOp, (Vec)Sort, and (Vec)Materialize operator. The (Vec)HashJoin calculation of width is the width of the right subtree operator, it will be displayed in the right subtree.

Column	Description
E-width	Estimated width of the output tuple of each operator.
E-costs	Estimated execution cost of each operator. ● E-costs are defined by the optimizer based on cost parameters, habitually grasping disk page as a unit. Other overhead parameters are set by referring to E-costs. ● The cost of each node (the E-costs value) includes the cost of all of its child nodes. ● Overhead reflects only what the optimizer is concerned about, but does not consider the time that the result row passed to the client. Although the time may play a very important role in the actual total time, it is ignored by the optimizer, because it cannot be changed by modifying the plan.

2. SQL Diagnostic Information

   SQL self-diagnosis information. Performance optimization points identified during optimization and execution are displayed. When **EXPLAIN** with the **VERBOSE** attribute (built-in **VERBOSE** of **EXPLAIN PERFORMANCE**) is executed on DML statements, SQL self-diagnosis information is also generated to help locate performance issues.

3. Predicate Information (identified by plan id)

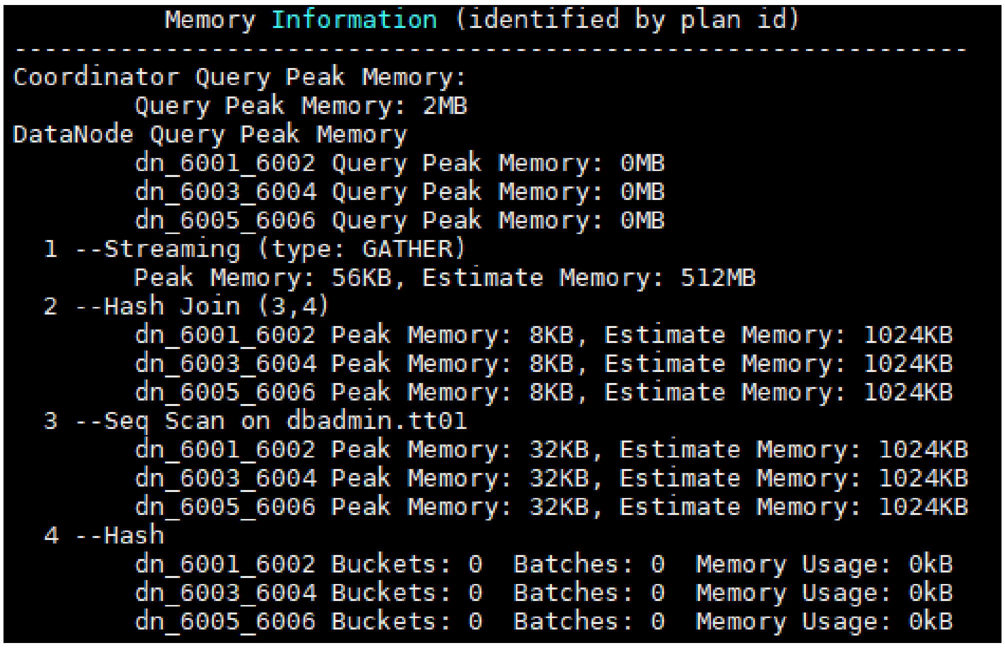


```
Predicate Information (identified by plan id)
---------------------------------------------
  2 --Hash Join (3,4)
        Hash Cond: (tt01.c1 = tt02.c2)
  3 --Seq Scan on dbadmin.tt01
        Filter: (tt01.c1 >= 202007)
  5 --Seq Scan on dbadmin.tt02
        Filter: (tt02.c2 >= 202007)
```

   This part displays the filtering conditions of the corresponding execution operator node, that is, the information that does not change during the entire plan execution, mainly the join conditions and filter information.

   8.3.0 and later cluster versions support the display the information of **CU Predicate Filter** and P**ushdown Predicate Filter(will be pruned)** related to dictionary plans.

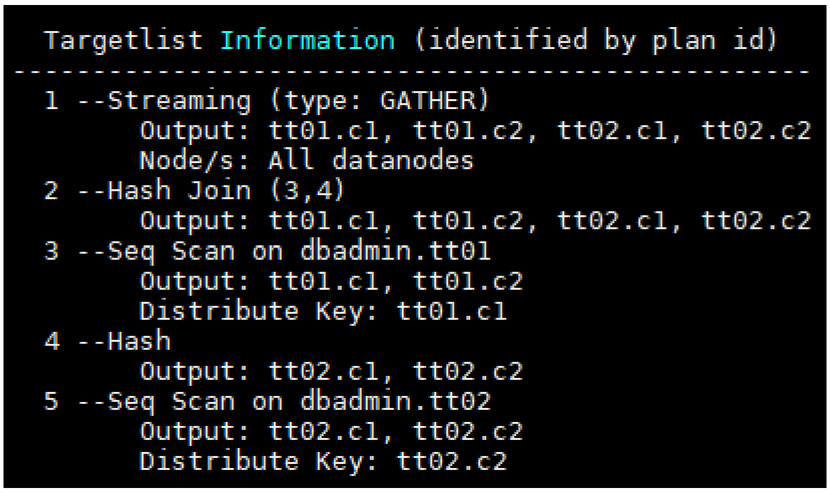
4. Memory Information (identified by plan id)



```
          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
```

Memory Usage displays the memory usage of operators in the entire plan, mainly Hash and Sort operators, including the peak memory of operators (Peak Memory), memory estimated by the optimizer (Estimate Memory), and control memory (Control Memory), estimated memory usage (operator memory), actual width during execution (Width), number of automatic memory expansion times (Auto Spread Num), whether to spill data to disks in advance (Early Spilled), and spill information which includes the number of repeated data spills (Spill Time(s)), number of internal and foreign table partitions spilled to disks (inner/outer partition spill num), number of files spilled to disks (temp file num), amount of data spilled to disks, and amount of data flushed to the minimum and maximum partitions (written disk IO [min, max]). The Sort operator does not display the number of files written to disks, and displays disks only when displaying sorting methods.

5. Targetlist Information (identified by plan id)

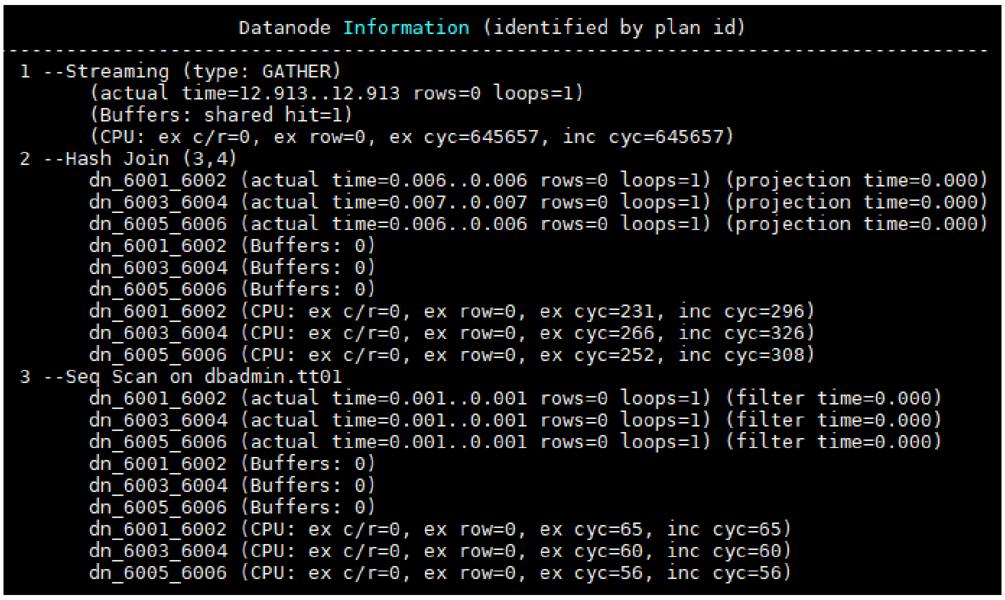


```
  Targetlist Information (identified by plan id)
-----------------------------------------------------
  1 --Streaming (type: GATHER)
        Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
        Node/s: All datanodes
  2 --Hash Join (3,4)
        Output: tt01.c1, tt01.c2, tt02.c1, tt02.c2
  3 --Seq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

This part displays the output target column information of each operator.

In 8.3.0 and later cluster versions, the dictionary parameters **Dict Optimized** and **Dict Decoded** can be displayed, indicating dictionary columns and dictionary codes, respectively.

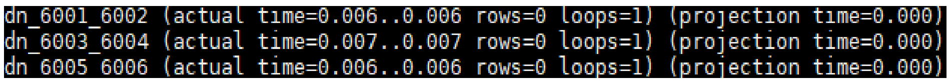
6. DataNode Information (identified by plan id)



```
                    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)
```

This part displays the execution time of each operator (including the execution time of filtering and projection, if any), CPU usage, and buffer usage.

– Operator execution information

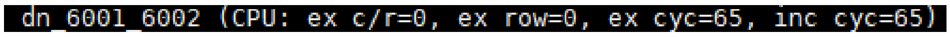


```
dn_6001_6002 (actual time=0.006..0.006 rows=0 loops=1) (projection time=0.000)
dn_6003_6004 (actual time=0.007..0.007 rows=0 loops=1) (projection time=0.000)
dn_6005_6006 (actual time=0.006..0.006 rows=0 loops=1) (projection time=0.000)
```

The execution information of each operator consists of three parts:

- **dn_6001_6002**/**dn_6003_6004** indicates the information about the execution node. The information in the brackets is the actual execution information.
- **actual time** indicates the actual execution time. The first number indicates the duration from the time when the operator is executed to the time when the first data record is output. The second number indicates the total execution time of all data records.
- **rows** indicates the number of output data rows of the operator.
- **loops** indicates the number of execution times of the operator. Note that for a partitioned table, scan on each partition is counted as a scan. Scan on a new partition is counted as a new scan.

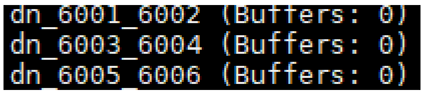
– CPU information



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

Each operator execution process has CPU information. **cyc** indicates the number of CPU cycles, and **ex cyc** indicates the number of cycles of the current operator, excluding its subnodes. **inc cyc** indicates the number of cycles, including subnodes, **ex row** indicates the number of data rows output by the current operator, and **ex c/r** indicates the mean of **ex cyc** and **ex row**.

– Buffer information

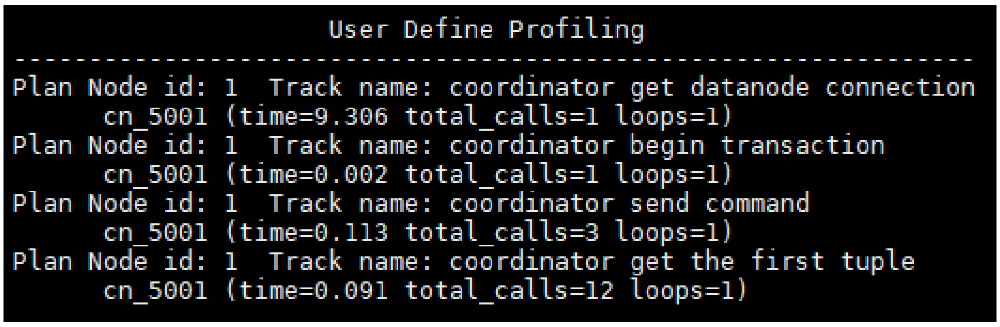


```
dn_6001_6002 (Buffers: 0)
dn_6003_6004 (Buffers: 0)
dn_6005_6006 (Buffers: 0)
```

Buffers indicates the buffer information, including the read and write operations on shared blocks and temporary blocks.

Shared blocks contain tables and indexes, and temporary blocks are disk blocks used in sorting and materialization. The number of blocks displayed on the upper-layer node contains the number of blocks used by all its subnodes.

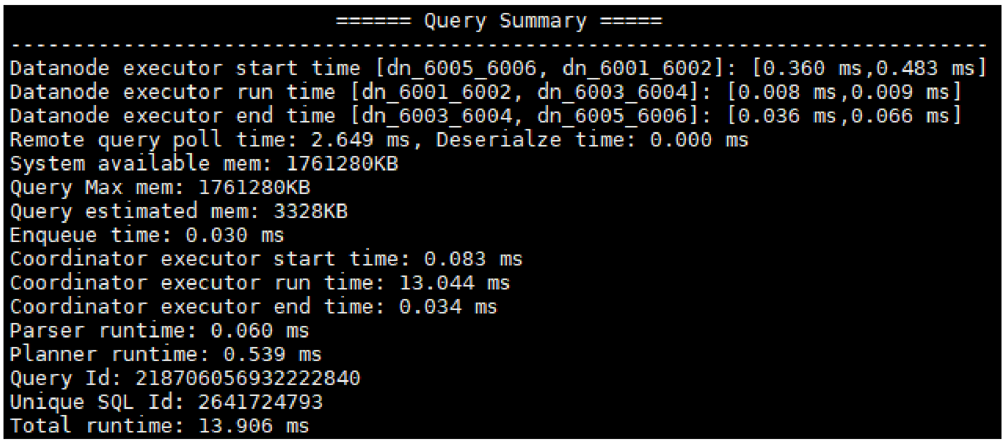
7. User Define Profiling



```
                         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)
```

User-defined information, including the time when CNs and DNs are connected, the time when DNs are connected, and some execution information at the storage layer.

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
```

The total execution time and network traffic, including the maximum and minimum execution time in the initialization and end phases on each DN, initialization, execution, and time in the end phase on each CN, and the system available memory during the current statement execution, and statement estimation memory information.

– **DataNode executor start time**: start time of the DN executor. The format is [min_node_name, max_node_name]: [min_time, max_time].

– **DataNode executor run time**: running time of the DN executor. The format is [min_node_name, max_node_name]: [min_time, max_time].

– **DataNode executor end time**: end time of the DN executor. The format is [min_node_name, max_node_name]: [min_time, max_time].

– **Remote query poll time**: poll waiting time for receiving results

- **System available mem**: available system memory
- **Query Max mem**: maximum query memory.
- **Enqueue time**: enqueuing time
- **Coordinator executor start time**: start time of the CN executor
- **Coordinator executor run time**: CN executor running time
- **Coordinator executor end time**: end time of the CN executor
- **Parser runtime**: parser running time
- **Planner runtime**: optimizer execution time
- Network traffic, or, the amount of data sent by the stream operator
- **Query Id**: query ID.
- **Unique SQL ID**: constraint SQL ID
- **Total runtime**: total execution time

NOTICE

- The difference between A-rows and E-rows shows the deviation between the optimizer estimation and actual execution. Generally, if the deviation is large, the plan generated by the optimizer cannot be trusted, and you need to modify the deviation value.
- If the difference of the A-time values is large, it indicates that the operator computing skew (difference between execution time on DNs) is large and that manual performance tuning is required. Generally, for two adjacent operators, the execution time of the upper-layer operator includes that of the lower-layer operator. However, if the upper-layer operator is a stream operator, its execution time may be less than that of the lower-layer operator, as there is no driving relationship between threads.
- **Max Query Peak Memory** is often used to estimate the consumed memory of SQL statements, and is also used as an important basis for setting a memory parameter during SQL statement optimization. Generally, the output from **EXPLAIN ANALYZE** or **EXPLAIN PERFORMANCE** is provided for the input for further optimization.

4 SQL Optimization Guide

4.1 Optimization Process

You can analyze slow SQL statements to optimize them.

Procedure

Step 1 Collect all table statistics associated with the SQL statements. In a database, statistics indicate the source data of a plan generated by a planner. If statistics are unavailable or out of date, the execution plan may seriously deteriorate, leading to low performance. According to past experience, about 10% performance problem occurred because no statistics are collected. For details, see **Updating Statistics**.

Step 2 **Review and modify the table definition.**

Step 3 Generally, some SQL statements can be converted to its equivalent statements in all or certain scenarios by rewriting queries. SQL statements are simpler after they are rewritten. Some execution steps can be simplified to improve the performance. The query rewriting method is universal in all databases. **SQL Statement Rewriting Rules** describes several optimization methods by rewriting SQL statements.

Step 4 View the execution plan to find out the cause. If the SQL statements have been running for a long period of time and not ended, run the **EXPLAIN** command to view the execution plan and then locate the fault. If the SQL statement has been executed, run the **EXPLAIN ANALYZE** or **EXPLAIN PERFORMANCE** command to check the execution plan and actual running situation and then accurately locate the fault. For details about the execution plan, see **SQL Execution Plan**.

Step 5 For details about **EXPLAIN** or **EXPLAIN PERFORMANCE**, the reason why SQL statements are slowly located, and how to solve this problem, see **Typical SQL Optimization Methods**.

Step 6 Specify a join order; join, stream, or scan operations; number of rows in a result; or redistribution skew information to optimize an execution plan, improving query performance. For details, see **Hint-based Tuning**.

Step 7 To maintain high database performance, you are advised to perform **Routinely Maintaining Tables** and **Routinely Recreating an Index**.

Step 8 (Optional) Improve performance by using operators if resources are sufficient in GaussDB(DWS). For details, see **SMP Manual Optimization Suggestions**.

----End

4.2 Updating Statistics

In a database, statistics indicate the source data of a plan generated by a planner. If no collection statistics are available or out of date, the execution plan may seriously deteriorate, leading to low performance.

Context

The **ANALYZE** statement collects statistic about table contents in databases, which will be stored in the system table **PG_STATISTIC**. Then, the query optimizer uses the statistics to work out the most efficient execution plan.

After executing batch insertion and deletions, you are advised to run the **ANALYZE** statement on the table or the entire library to update statistics. By default, 30,000 rows of statistics are sampled. That is, the default value of the GUC parameter **default_statistics_target** is **100**. If the total number of rows in the table exceeds 1,600,000, you are advised to set **default_statistics_target** to **-2**, indicating that 2% of the statistics are collected.

For an intermediate table generated during the execution of a batch script or stored procedure, you also need to run the **ANALYZE** statement.

If there are multiple inter-related columns in a table and the conditions or grouping operations based on these columns are involved in the query, collect statistics about these columns so that the query optimizer can accurately estimate the number of rows and generate an effective execution plan.

Generating Statistics

Run the following commands to update the statistics about a table or the entire database:

```
ANALYZE tablename;                      --Update statistics about a table.
ANALYZE;                           ---Update statistics about the entire database.
```

Run the following statements to perform statistics-related operations on multiple columns:

```
ANALYZE tablename ((column_1, column_2));                    --Collect statistics about column_1 and
column_2 of tablename.

ALTER TABLE tablename ADD STATISTICS ((column_1, column_2));    --Declare statistics about column_1
and column_2 of tablename.
ANALYZE tablename;                                        --Collect statistics about one or more columns.

ALTER TABLE tablename DELETE STATISTICS ((column_1, column_2)); --Delete statistics about column_1
and column_2 of tablename or their statistics declaration.
```

NOTICE

- After the statistics are declared for multiple columns by running the **ALTER TABLE** *tablename* **ADD STATISTICS** statement, the system collects the statistics about these columns next time **ANALYZE** is performed on the table or the entire database. To collect the statistics, run the **ANALYZE** statement.
- Use **EXPLAIN** to show the execution plan of each SQL statement. If **rows=10** (the default value, probably indicating the table has not been analyzed) is displayed in the **SEQ SCAN** output of a table, run the **ANALYZE** statement for this table.

Improving the Quality of Statistics

ANALYZE samples data from a table based on the random sampling algorithm and calculates table data features based on the samples. The number of samples can be specified by the **default_statistics_target** parameter. The value of **default_statistics_target** ranges from -100 to 10000, and the default value is 100.

If **default_statistics_target** > 0, the number of samples is 300 x **default_statistics_target**. This means a larger value of **default_statistics_target** indicates a larger number of samples, larger memory space occupied by samples, and longer time required for calculating statistics.

If **default_statistics_target** < 0, the number of samples is **default_statistics_target**/100 x Total number of rows in the table. A smaller value of **default_statistics_target** indicates a larger number of samples. When **default_statistics_target** < 0, the sampled data is written to the disk. In this case, the samples do not occupy memory. However, the calculation still takes a long time because the sample size is too large.

When **default_statistics_target** < 0, the actual number of samples is **default_statistics_target**/100 x Total number of rows in the table. Therefore, this sampling mode is also called percentage sampling.

Automatic Statistics Collection

When the parameter **autoanalyze** is enabled, if the query statement reaches the optimizer and finds that there are no statistics, statistics collection will be automatically triggered to meet the optimizer's requirements.

Note: Automatic statistics collection is triggered only for complex query SQL statements that are sensitive to statistics (such as multi-table association). Simple queries (such as single-point query and single-table aggregation) do not trigger automatic statistics collection.

4.3 Reviewing and Modifying a Table Definition

In a distributed framework, data is distributed on DNs. Data on one or more DNs is stored on a physical storage device. To properly define a table, you must:

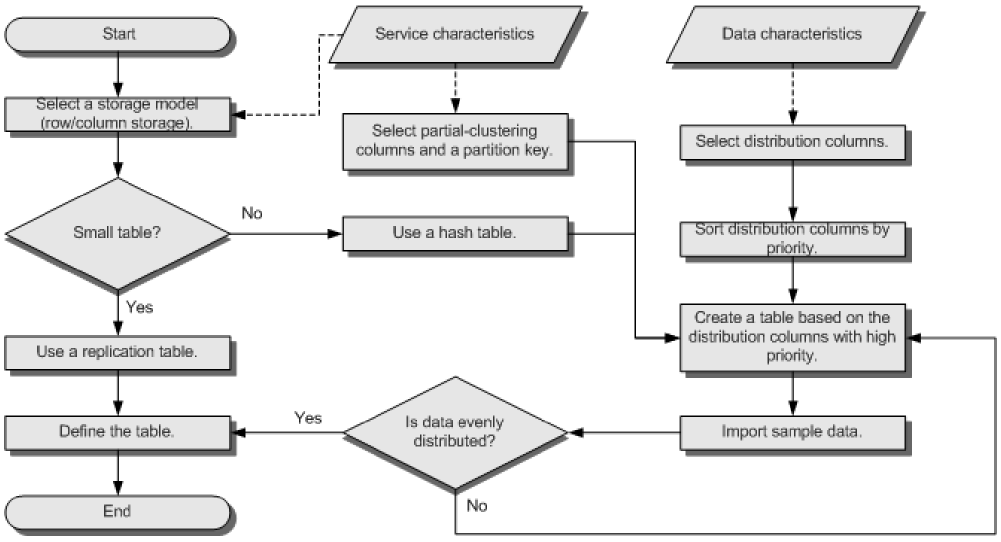
1. **Evenly distribute data on each DN** to avoid the available capacity decrease of a cluster caused by insufficient storage space of the storage device

associated with a DN. Specifically, select a proper distribution key to avoid data skew.

2. **Evenly assign table scanning tasks on each DN** to avoid that a DN is overloaded by the table scanning tasks. Specifically, do not select columns in the equivalent filter of a base table as the distribution key.
3. **Reduce the data volume scanned** by using the partition pruning mechanism.
4. **Avoid the use of random I/O** by using clustering or partial clustering.
5. **Avoid data shuffle** to reduce the network pressure by selecting the **join-condition** column or **group by** column as the distribution column.

The distribution column is the core for defining a table. The following figure shows the procedure of defining a table. The table definition is created during the database design and is reviewed and modified during the SQL statement optimization.

Figure 4-1 Procedure of defining a table


Start
Service characteristics
Data characteristics
Select a storage model (row/column storage).
Select partial-clustering columns and a partition key.
Select distribution columns.
Small table?
No
Use a hash table.
Sort distribution columns by priority.
Yes
Create a table based on the distribution columns with high priority.
Use a replication table.
Define the table.
Yes
Is data evenly distributed?
Import sample data.
No
End

4.4 SQL Statement Rewriting Rules

Based on the database SQL execution mechanism and a large number of practices, summarize finds that: using rules of a certain SQL statement, on the basis of the so that the correct test result, which can improve the SQL execution efficiency. You can comply with these rules to greatly improve service query efficiency.

- Replacing **UNION** with **UNION ALL**

  UNION eliminates duplicate rows while merging two result sets but **UNION ALL** merges the two result sets without deduplication. Therefore, replace **UNION** with **UNION ALL** if you are sure that the two result sets do not contain duplicate rows based on the service logic.

- **Adding NOT NULL to the join column**

  If there are many **NULL** values in the **JOIN** columns, you can add the filter criterion **IS NOT NULL** to filter data in advance to improve the **JOIN** efficiency.

- Converting **NOT IN** to **NOT EXISTS**

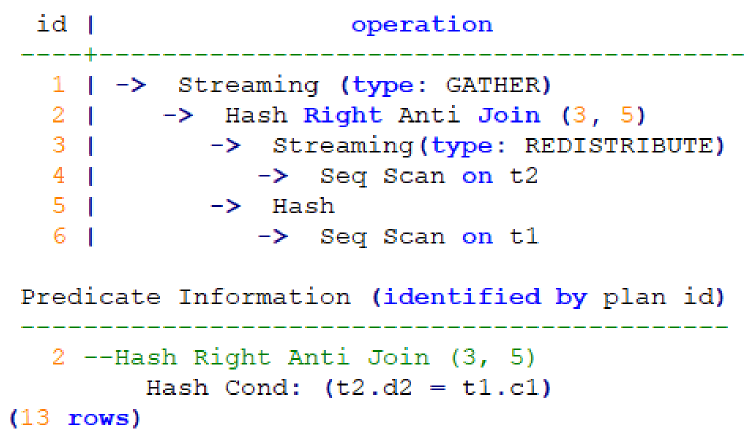
nestloop anti join must be used to implement **NOT IN**, and **Hash anti join** is required for **NOT EXISTS**. If no **NULL** value exists in the **JOIN** column, **NOT IN** is equivalent to **NOT EXISTS**. Therefore, if you are sure that no **NULL** value exists, you can convert **NOT IN** to **NOT EXISTS** to generate **hash joins** and to improve the query performance.

As shown in the following figure, the **t2.d2** column does not contain null values (it is set to **NOT NULL**) and **NOT EXISTS** is used for the query.

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

The generated execution plan is as follows:

Figure 4-2 NOT EXISTS execution plan



```
 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)
```

- Use **hashagg**.

  If a plan involving groupAgg and SORT operations generated by the **GROUP BY** statement is poor in performance, you can set **work_mem** to a larger value to generate a **hashagg** plan, which does not require sorting and improves the performance.

- Replace functions with **CASE** statements

  The GaussDB(DWS) performance greatly deteriorates if a large number of functions are called. In this case, you can modify the pushdown functions to **CASE** statements.

- **Do not use functions or expressions for indexes.**

  Using functions or expressions for indexes stops indexing. Instead, it enables scanning on the full table.

- Do not use **!=** or **<>** operators, **NULL**, **OR**, or implicit parameter conversion in **WHERE** clauses.

- **Split complex SQL statements.**

  You can split an SQL statement into several ones and save the execution result to a temporary table if the SQL statement is too complex to be tuned using the solutions above, including but not limited to the following scenarios:

  – The same subquery is involved in multiple SQL statements of a task and the subquery contains large amounts of data.

  – Incorrect **Plan cost** causes a small hash bucket of subquery. For example, the actual number of rows is 10 million, but only 1000 rows are in hash bucket.

- Functions such as **substr** and **to_number** cause incorrect measures for subqueries containing large amounts of data.
- **BROADCAST** subqueries are performed on large tables in multi-DN environment.

4.5 Typical SQL Optimization Methods

SQL optimization involves continuous analysis and adjustment. You need to test-run a query, locate and fix its performance issues (if any) based on its execution plan, and run it again, until the execution performance meet your requirements.

4.5.1 SQL Self-Diagnosis

Performance problems may occur when you run the **INSERT/UPDATE/DELETE/SELECT/MERGE INTO** or **CREATE TABLE AS** statement. The product supports automatic performance diagnosis and saves related diagnosis information to the **Real-time Top SQL**. When **enable_resource_track** is set to **on**, the diagnosis information is dumped to the **Historical Top SQL**. You can query the warning field in the views **GS_WLM_SESSION_STATISTICS**, **GS_WLM_SESSION_HISTORY**, **GS_WLM_SESSION_INFO** in *Developer Guide* to obtain the corresponding performance diagnosis information for performance optimization.

Alarms that can trigger SQL self-diagnosis depend on the settings of **resource_track_level**. When **resource_track_level** is set to **query**, you can diagnose alarms such as uncollected multi-column/single-column statistics, partitions not pruned, and failure of pushing down SQL statements. When **resource_track_level** is set to **perf** or **operator**, all alarms can be diagnosed.

Whether a SQL plan will be diagnosed depends on the settings of **resource_track_cost**. A SQL plan will be diagnosed only if its execution cost is greater than **resource_track_cost**. You can use the **EXPLAIN** keyword to check the plan execution cost.

When **EXPLAIN PERFORMANCE** or **EXPLAIN VERBOSE** is executed, SQL self-diagnosis information, except that without multi-column statistics, will be generated. For details, see **SQL Execution Plan**.

Alarms

Currently, the following performance alarms will be reported:

- Statistics of a single column or multiple columns are not collected.

  If statistics of a single column or multiple columns are not collected, an alarm is reported. To handle this alarm, you are advised to perform **ANALYZE** on related tables. For details, see **Updating Statistics** and **Optimizing Statistics**.

  If no statistics are collected for the OBS foreign table and HDFS foreign table in the query statement, an alarm indicating that statistics are not collected will be reported. Because the **ANALYZE** performance of the OBS foreign table and HDFS foreign table is poor, you are not advised to perform **ANALYZE** on these tables. Instead, you are advised to use the **ALTER FOREIGN TABLE** syntax to modify the **totalrows** attribute of the foreign table to correct the estimated number of rows.

  Example alarms:

The statistics about a table are not collected.

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

The statistics about a single column are not collected.

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

The statistics about multiple columns are not collected.

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

The statistics about a single column and multiple columns are not collected.

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

- Partitions are not pruned (supported by 8.1.2 and later versions).

  When a partitioned table is queried, the partition is pruned based on the constraints on the partition key to improve the query performance. However, the partition table may not be pruned due to improper constraints, deteriorating the query performance. For details, see **Case: Rewriting SQL Statements and Eliminating Prune Interference**.

- SQL statements are not pushed down.

  The cause details are displayed in the alarms. For details, see **Optimizing Statement Pushdown**.

  The potential causes for the pushdown failure are as follows:

  - Caused by functions

    The function name is displayed in the diagnosis information. Function pushdown is determined by the **shippable** attribute of the function. For details, see the **CREATE FUNCTION** syntax.

  - Caused by syntax

    The diagnosis information displays the syntax that causes the pushdown failure. For example, if the statement contains the **With Recursive**, **Distinct On**, or **row** expression and the return value is of the record type, an alarm is reported, indicating that the syntax does not support pushdown.

  Example alarms:

```
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
```

- In a hash join, the larger table is used as the inner table.

  An alarm will be reported if the number of rows in the inner table reaches or exceeds 10 times of that in the foreign table, more than 100,000 inner-table rows are processed on each DN in average, and data has been flushed to disks. You can view the **query_plan** column in **GS_WLM_SESSION_HISTORY** to check whether hash joins are used. In this scenario, you need to adjust the sequence of the HashJoin internal and foreign tables. For details, see **Join Order Hints**.

  Example alarm:

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

In the preceding command, **7** indicates the operator whose ID is **7** in the **query_plan** column.

- **nestloop** is used in a large-table equivalent join.

  An alarm will be reported if nested loop is used in an equivalent join where more than 100,000 larger-table rows are processed on each DN in average. You can view the **query_plan** column of **GS_WLM_SESSION_HISTORY** to check whether nested loop is used. In this scenario, you need to adjust the table join mode and disable the NestLoop join mode between the current internal and foreign tables. For details, see **Join Operation Hints**.

  Example alarm:

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

- A large table is broadcasted.

  An alarm will be reported if more than 100 thousand of rows are broadcasted on each DN in average. In this scenario, the broadcast operation of the BroadCast lower-layer operator needs to be disabled. For details, see **Stream Operation Hints**.

  Example alarm:

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

- Data skew occurs.

  An alarm will be reported if the number of rows processed on any DN exceeds 100 thousand, and the number of rows processed on a DN reaches or exceeds 10 times of that processed on another DN. Generally, this alarm is generated due to storage layer skew or computing layer skew. For details, see **Optimizing Data Skew**.

  Example alarm:

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

- The index is improper.

  During base table scanning, an alarm is reported if the following conditions are met:

  – For row-store tables:

    ■ When the index scanning is used, the ratio of the number of output lines to the number of scanned lines is greater than 1/1000 and the number of output lines is greater than 10,000.

    ■ When sequential scanning is used, the number of output lines to the number of scanned lines is less than 1/1000, the number of output lines is less than or equal to 10,000, and the number of scanned lines is greater than 10,000.

  – For column-store tables:

    ■ When the index scanning is used, the ratio of the number of output lines to the number of scanned lines is greater than 1/10000 and the number of output lines is greater than 100.

  - ▪ When sequential scanning is used, the number of output lines to the number of scanned lines is less than 1/10,000, the number of output lines is less than or equal to 100, and the number of scanned lines is greater than 10,000.

  For details, see **Optimizing Operators**. You can also refer to **Case: Creating an Appropriate Index** and **Case: Setting Partial Cluster Keys**.

  Example alarms:

```
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
```

  The diagnosis result is only a suggestion for the current SQL statement. You are advised to create an index only for frequently used filter criteria.

- Estimation is inaccurate.

  An alarm will be reported if the maximum number or the estimated maximum number of rows processed on a DN is over 100,000, and the larger number reaches or exceeds 10 times of the smaller one. In this scenario, you can refer to **Rows Hints** to correct the estimation on the number of rows, so that the optimizer can re-design the execution plan based on the correct number.

  Example alarm:

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

Restrictions

1. An alarm contains a maximum of 2048 characters. If the length of an alarm exceeds this value (for example, a large number of long table names and column names are displayed in the alarm when their statistics are not collected), a warning instead of an alarm will be reported.
```
WARNING, "Planner issue report is truncated, the rest of planner issues will be skipped"
```
2. If a query statement contains the **Limit** operator, alarms of operators lower than **Limit** will not be reported.
3. For alarms about data skew and inaccurate estimation, only alarms on the lower-layer nodes in a plan tree will be reported. This is because the same alarms on the upper-level nodes may be triggered by problems on the lower-layer nodes. For example, if data skew occurs on the **Scan** node, data skew may also occur in operators (for example, **Hashagg**) at the upper layer.

4.5.2 Optimizing Statement Pushdown

Statement Pushdown

Currently, the GaussDB(DWS) optimizer can use three methods to develop statement execution policies in the distributed framework: generating a statement pushdown plan, a distributed execution plan, or a distributed execution plan for sending statements.

- A statement pushdown plan pushes query statements from a CN down to DNs for execution and returns the execution results to the CN.
- In a distributed execution plan, a CN compiles and optimizes query statements, generates a plan tree, and then sends the plan tree to DNs for

execution. After the statements have been executed, execution results will be returned to the CN.

- A distributed execution plan for sending statements pushes queries that can be pushed down (mostly base table scanning statements) to DNs for execution. Then, the plan obtains the intermediate results and sends them to the CN, on which the remaining queries are to be executed.

The third policy sends many intermediate results from the DNs to a CN for further execution. In this case, the CN performance bottleneck (in bandwidth, storage, and computing) is caused by statements that cannot be pushed down to DNs. Therefore, you are not advised to use the query statements that only the third policy is applicable to.

Statements cannot be pushed down to DNs if they have **Functions That Do Not Support Pushdown** or **Syntax That Does Not Support Pushdown**. Generally, you can rewrite the execution statements to solve the problem.

Viewing Whether the Execution Plan Has Been Pushed Down to DNs

Perform the following procedure to quickly determine whether the execution plan can be pushed down to DNs:

Step 1 Set the GUC parameter **enable_fast_query_shipping** to **off** to use the distributed framework policy for the query optimizer.

```
SET enable_fast_query_shipping = off;
```

Step 2 View the execution plan.

If the execution plan contains Data Node Scan, the SQL statements cannot be pushed down to DNs. If the execution plan contains Streaming, the SQL statements can be pushed down to DNs.

For example:

```
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;
```

The execution plan is as follows, which indicates that the SQL statement cannot be pushed down.

```
                          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)
```

----End

Syntax That Does Not Support Pushdown

SQL syntax that does not support pushdown is described using the following table definition examples:

```
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 does not support redistribution.
CREATE TABLE sal_emp ( c1 integer[] ) DISTRIBUTE BY replication;
```

- The **returning** statement cannot be pushed down.

```
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)
```

- If columns in **count(distinct expr)** do not support redistribution, they do not support pushdown.

```
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)
```

- Statements using **distinct on** cannot be pushed down.

```
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)
```

- In a statement using **FULL JOIN**, if the column specified using **JOIN** does not support redistribution, the statement does not support pushdown.

```
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)
```

- Does not support array expression pushdown.

```
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)
```

- The following table describes the scenarios where a statement containing **WITH RECURSIVE** cannot be pushed down in the current version, as well as the causes.

No.	Scenario	Cause of Not Supporting Pushdown
1	The query contains foreign tables or HDFS tables.	LOG: SQL can't be shipped, reason: RecursiveUnion contains HDFS Table or ForeignScan is not shippable (In this table, **LOG** describes the cause of not supporting pushdown.) In the current version, queries containing foreign tables or HDFS tables do not support pushdown.
2	Multiple Node Groups	LOG: SQL can't be shipped, reason: With-Recursive under multi-nodegroup scenario is not shippable In the current version, pushdown is supported only when all base tables are stored and computed in the same Node Group.

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
		
		ALL is not used for **UNION**. In this case, the return result is deduplicated.

No.	Scenario	Cause of Not Supporting Pushdown
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 A base table contains the system catalog.

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
		
		Only **VALUES** is used for scanning base tables. In this case, the statement can be executed on the CN, and DNs are unnecessary.

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 The correlation conditions of correlated subqueries are only in the recursion part, and the non-recursion part has no correlation condition.
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) The **replicate** plan is used for **limit** in the non-recursion part but the **hash** plan is used in the recursion part, resulting in conflicts.

No.	Scenario	Cause of Not Supporting Pushdown
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" **recursive** of multiple-layers are nested. That is, a **recursive** is nested in the recursion part of another **recursive**.

Functions That Do Not Support Pushdown

This module describes the variability of functions. The function variability in GaussDB(DWS) is as follows:

- **IMMUTABLE**

  Indicates that the function always returns the same result if the parameter values are the same.
- **STABLE**

  Indicates that the function cannot modify the database, and that within a single table scan it will consistently return the same result for the same parameter values, but that its result varies by SQL statements.
- **VOLATILE**

  Indicates that the function value can change even within a single table scan, so no optimizations can be made.

The volatility of a function can be obtained by querying its **provolatile** column in **pg_proc**. The value **i** indicates immutable, **s** indicates stable, and **v** indicates volatile. The valid values of the **proshippable** column in **pg_proc** are **t**, **f**, and **NULL**. This column and the **provolatile** column together describe whether a function is pushed down.

- If the **provolatile** of a function is **i**, the function can be pushed down regardless of the value of **proshippable**.
- If the **provolatile** of a function is **s** or **v**, the function can be pushed only if the value of **proshippable** is **t**.
- CTEs containing random are not pushed down, because pushdown may lead to incorrect results.

For a UDF, you can specify the values of **provolatile** and **proshippable** during its creation. For details, see CREATE FUNCTION.

In scenarios where a function does not support pushdown, perform one of the following as required:

- If it is a system function, replace it with a functionally equivalent one.
- If it is a UDF function, check whether its **provolatile** and **proshippable** are correctly defined.

Example: UDF

Define a user-defined function that generates fixed output for a certain input as the **immutable** type.

Use the TPCDS sales information as an example. You need to define a function to obtain the discount information.

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

Run the following statement:

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

The execution plan is as follows:

```
 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)
```

func_percent_2 is not pushed down, and **ss_sales_price** and **ss_list_price** are executed on a CN. In this case, a large amount of resources on the CN is consumed, and the performance deteriorates as a result.

In this example, the function returns certain output when certain input is entered. Therefore, we can modify the function to the following one:

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

Run the following statement:

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

The execution plan is as follows:

```
 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)
```

func_percent_1 is pushed down to DNs for quicker execution. (In TPCDS 1000X, where three CNs and 18 DNs are used, the query efficiency is improved by over 100 times).

Example 2: Pushing Down the Sorting Operation

For details, see **Case: Pushing Down Sort Operations to DNs**.

4.5.3 Optimizing Subqueries

What Is a Subquery

When an application runs a SQL statement to operate the database, a large number of subqueries are used because they are more clear than table join. Especially in complicated query statements, subqueries have more complete and independent semantics, which makes SQL statements clearer and easy to understand. Therefore, subqueries are widely used.

In GaussDB(DWS), subqueries can also be called sublinks based on the location of subqueries in SQL statements.

- Subquery: corresponds to a scope table (RangeTblEntry) in the query parse tree. That is, a subquery is a **SELECT** statement following immediately after the **FROM** keyword.
- Sublink: corresponds to an expression in the query parsing tree. That is, a sublink is a statement in the **WHERE** or **ON** clause or in the target list.

  In conclusion, a subquery is a scope table and a sublink is an expression in the query parsing tree. A sublink can be found in constraint conditions and expressions. In GaussDB(DWS), sublinks can be classified into the following types:

  – exist_sublink: corresponding to the **EXIST** and **NOT EXIST** statements.
  – any_sublink: corresponding to the **OP ANY(SELECT...)** statement. **OP** can be the **IN**, **<**, **>**, or **=** operator.
  – all_sublink: corresponding to the **OP ALL(SELECT...)** statement. **OP** can be the **IN**, **<**, **>**, or **=** operator.
  – rowcompare_sublink: corresponding to the **RECORD OP (SELECT...)** statement.
  – expr_sublink: corresponding to the **(SELECT** with a single target list item**)** statement.
  – array_sublink: corresponding to the **ARRAY(SELECT...)** statement.
  – cte_sublink: corresponding to the **WITH(...)** statement.

  The sublinks commonly used in OLAP and HTAP are exist_sublink and any_sublink. The sublinks are pulled up by the optimization engine of GaussDB(DWS). Because of the flexible use of subqueries in SQL statements, complex subqueries may affect query performance. Subqueries are classified into non-correlated subqueries and correlated subqueries.

  – **Non-correlated subquery**

    The execution of a subquery is independent from any attribute of outer queries. In this way, a subquery can be executed before outer queries.

    Example:

```
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)
```

– **Correlated subquery**

  The execution of a subquery depends on some attributes of outer queries which are used as **AND** conditions of the subquery. In the following example, **t1.c1** in the **t2.c1 = t1.c1** condition is a dependent attribute. Such a subquery depends on outer queries and needs to be executed once for each outer query.

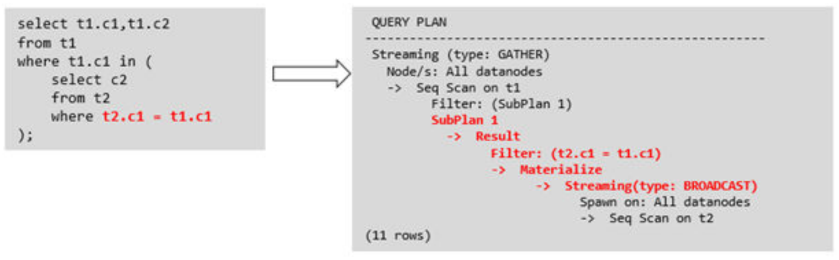
  Example:

```
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) SubLink Optimization

A subquery is pulled up to join with tables in outer queries, preventing the subquery from being converted into the combination of a subplan and broadcast. You can run the **EXPLAIN** statement to check whether a subquery is converted into the combination of a subplan and broadcast.

Example:



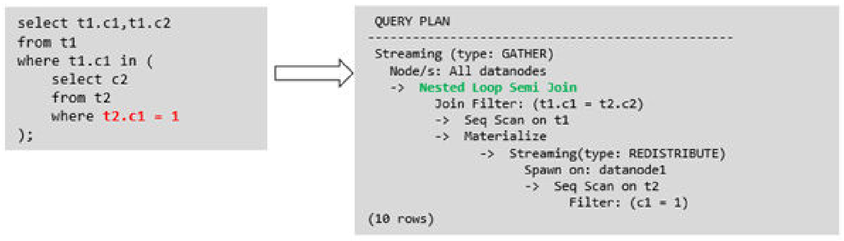
```
select t1.c1,t1.c2
from t1
where t1.c1 in (
    select c2
    from t2
    where t2.c1 = t1.c1
);
```

```
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)
```

● Sublink-release supported by GaussDB(DWS)

– Pulling up the **IN** sublink

- The subquery cannot contain columns in the outer query (columns in more outer queries are allowed).
- The subquery cannot contain volatile functions.



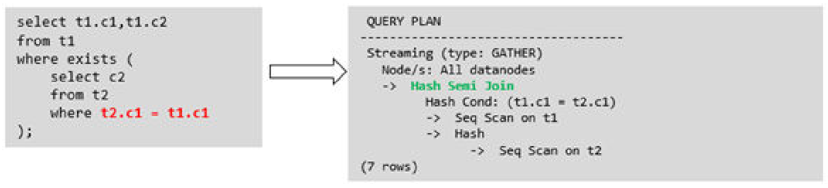
```
select t1.c1,t1.c2
from t1
where t1.c1 in (
     select c2
     from t2
     where t2.c1 = 1
);
```

```
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)
```

– Pulling up the **EXISTS** sublink

The **WHERE** clause must contain a column in the outer query. Other parts of the subquery cannot contain the column. Other restrictions are as follows:

- The subquery must contain the **FROM** clause.
- The subquery cannot contain the **WITH** clause.
- The subquery cannot contain aggregate functions.
- The subquery cannot contain a **SET**, **SORT**, **LIMIT**, **WindowAgg**, or **HAVING** operation.
- The subquery cannot contain volatile functions.



```
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)
```

– Pulling up an equivalent query containing aggregation functions

The **WHERE** condition of the subquery must contain a column from the outer query. Equivalence comparison must be performed between this column and related columns in tables of the subquery. These conditions must be connected using **AND**. Other parts of the subquery cannot contain the column. Other restrictions are as follows:

- The expression in the **WHERE** condition of the subquery must be table columns.
- After the **SELECT** keyword of the subquery, there must be only one output column. The output column must be an aggregation function (for example, **MAX**), and the parameter (for example, **t2.c2**) of the aggregate function cannot be columns of a table (for example, **t1**) in outer quires. The aggregate function cannot be **COUNT**.

  For example, the following subquery can be pulled up:

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

The following subquery cannot be pulled up because the subquery has no aggregation function.

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

The following subquery cannot be pulled up because the subquery has two output columns:

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

- The subquery must be a **FROM** clause.
- The subquery cannot contain a **GROUP BY**, **HAVING**, or **SET** operation.
- The subquery can only be inner join.

  For example, the following subquery can be pulled up:

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

- The target list of the subquery cannot contain the function that returns a set.
- The **WHERE** condition of the subquery must contain a column from the outer query. Equivalence comparison must be performed between this column and related columns in tables of the subquery. These conditions must be connected using **AND**. Other parts of the subquery cannot contain the column. For example, the following subquery can be pulled up:

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

  If another condition is added to the subquery in the previous example, the subquery cannot be pulled up because the subquery references to the column in the outer query. Example:

  ```
  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

  ));
  ```

– Pulling up a sublink in the **OR** clause

  If the **WHERE** condition contains a **EXIST**-related sublink connected by **OR**,

  for example,

  ```
  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);
  ```

  the process of pulling up such a sublink is as follows:

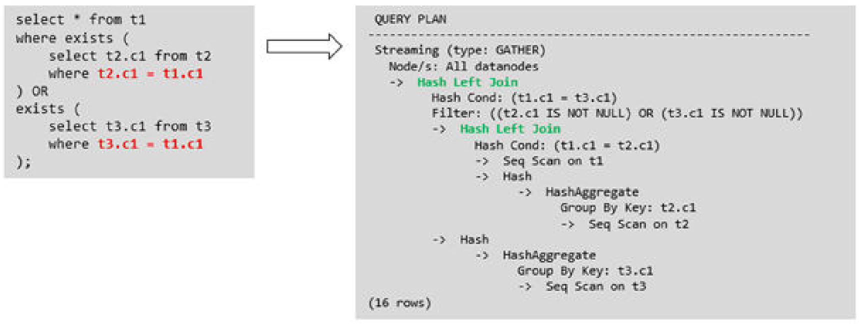
i. Extract **opExpr** from the **OR** clause in the **WHERE** condition. The value is **t1.a = (select avg(a) from t3 where t1.b = t3.b)**.

ii. The **opExpr** contains a subquery. If the subquery can be pulled up, the subquery is rewritten as **elect avg(a), t3.b from t3 group by t3.b**, generating the **NOT NULL** condition **t3.b is not null**. The **opExpr** is replaced with this **NOT NULL** condition. In this case, the SQL statement changes to:

```
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. Extract the **EXISTS** sublink **exists (select * from t4 where t1.c = t4.c)** from the **OR** clause to check whether the sublink can be pulled up. If it can be pulled up, it is converted into **select t4.c from t4 group by t4.c**, generating the **NOT NULL** condition **t4.c is not null**. In this case, the SQL statement changes to:

```
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;
```



```
select * from t1
where exists (
    select t2.c1 from t2
    where t2.c1 = t1.c1
) OR
exists (
    select t3.c1 from t3
    where t3.c1 = t1.c1
);
```

```
QUERY PLAN
-----------------------------------------------------------------
Streaming (type: GATHER)
  Node/s: All datanodes
  ->  Hash Left Join
        Hash Cond: (t1.c1 = t3.c1)
        Filter: ((t2.c1 IS NOT NULL) OR (t3.c1 IS NOT NULL))
        ->  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
              ->  HashAggregate
                    Group By Key: t3.c1
                    ->  Seq Scan on t3
(16 rows)
```

- Sublink-release not supported by GaussDB(DWS)

  Except the sublinks described above, all the other sublinks cannot be pulled up. In this case, a join subquery is planned as the combination of a subplan and broadcast. As a result, if tables in the subquery have a large amount of data, query performance may be poor.

  If a correlated subquery joins with two tables in outer queries, the subquery cannot be pulled up. You need to change the parent query into a **WITH** clause and then perform the join.

  Example:

```
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);
```

  The parent query is changed into:

```
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);
```

  – The subquery (without **COUNT**) in the target list cannot be pulled up.

    Example:

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

```
from t1
where t1.c2 > 10;
```

The execution plan is as follows:

```
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)
```

The correlated subquery is displayed in the target list (query return list). Values need to be returned even if the condition **t1.c1=t2.c1** is not met. Therefore, use left outer join to join **T1** and **T2** so that SSQ can return padding values when the condition **t1.c1=t2.c1** is not met.

NOTE

ScalarSubQuery (SSQ) and Correlated-ScalarSubQuery (CSSQ) are described as follows:

- SSQ: a sublink that returns only a single row and column scalar value
- CSSQ: an SSQ containing conditions

The preceding SQL statement can be changed into:

```
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;
```

The execution plan after the change is as follows:

```
               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)
```

In the preceding example, the SSQ is pulled up to right join, preventing poor performance caused by the combination of a subplan and broadcast when the table (**T2**) in the subquery is too large.

– The subquery (with **COUNT**) in the target list cannot be pulled up.

Example:

```
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;
```

The execution plan is as follows:

```
                         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)
```

The correlated subquery is displayed in the target list (query return list). Values need to be returned even if the condition **t1.c1=t2.c1** is not met. Therefore, use left outer join to join **T1** and **T2** so that SSQ can return padding values when the condition **t1.c1=t2.c1** is not met. However, **COUNT** is used to ensure that **0** is returned when the condition is note met. Therefore, **case-when NULL then 0 else count(*)** can be used.

The preceding SQL statement can be changed into:

```
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;
```

The execution plan after the change is as follows:

```
  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)
```

– Pulling up nonequivalent subqueries

Example:

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

Nonequivalent subqueries cannot be pulled up. You can perform join twice (one CorrelationKey and one rownum self-join) to rewrite the statement.

You can rewrite the statement in either of the following ways:

- Subquery rewriting

```
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 rewriting

```
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;
```

NOTICE

- Currently, GaussDB(DWS) does not have an effective way to provide globally unique row IDs for tables and intermediate result sets. Therefore, the rewriting is difficult. It is recommended that this issue is avoided at the service layer or by using **t1.xc_node_id + t1.ctid** to associate row IDs. However, the high repetition rate of **xc_node_id** leads to low association efficiency, and **xc_node_id+ctid** cannot be used as the join condition of hash join.
- If the AGG type is **COUNT(*)**, **0** is used for data padding if **CASE-WHEN** is not matched. If the type is not **COUNT(*)**, **NULL** is used.
- CTE rewriting works better by using share scan.

More Optimization Examples

1. Change the base table to a replication table and create an index on the filter column.

```
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);
```

In this example, a correlated subquery is contained. To improve the query performance, you can change **sub_table** to a replication table and create an index on the **a** column.

2. Modify the **SELECT** statement, change the subquery to a **JOIN** relationship between the primary table and the parent query, or modify the subquery to

improve the query performance. Ensure that the subquery to be used is semantically correct.

```
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)
```

In the preceding example, a subplan is used. To remove the subplan, you can modify the statement as follows:

```
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)
```

In this way, the subplan is replaced by the semi-join between the two tables, greatly improving the execution efficiency.

4.5.4 Optimizing Statistics

What Is Statistic Optimization

GaussDB(DWS) generates optimal execution plans based on the cost estimation. Optimizers need to estimate the number of data rows and the cost based on statistics collected using **ANALYZE**. Therefore, the statistics is vital for the estimation of the number of rows and cost. Global statistics are collected using **ANALYZE**: **relpages** and **reltuples** in the **pg_class** table; **stadistinct**, **stanullfrac**, **stanumbersN**, **stavaluesN**, and **histogram_bounds** in the **pg_statistic** table.

Example 1: Poor Query Performance Due to the Lack of Statistics

In most cases, the lack of statistics in tables or columns involved in the query greatly affects the query performance.

The table structure is as follows:

```
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);
```

The query statements are as follows:

```
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;
```

If such an issue occurs, you can use the following methods to check whether statistics in tables or columns has been collected using **ANALYZE**.

1. Execute **EXPLAIN VERBOSE** to analyze the execution plan and check the warning information:

```
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. Check whether the following information exists in the log file in the **pg_log** directory. If it does, the poor query performance was caused by the lack of statistics in some tables or columns.

```
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.
```

By using any of the preceding methods, you can identify tables or columns whose statistics have not been collected using **ANALYZE**. You can execute **ANALYZE** to warnings or tables and columns recorded in logs to resolve the problem.

Example 2: Setting cost_param to Optimize Query Performance

For details, see **Case: Configuring cost_param for Better Query Performance**.

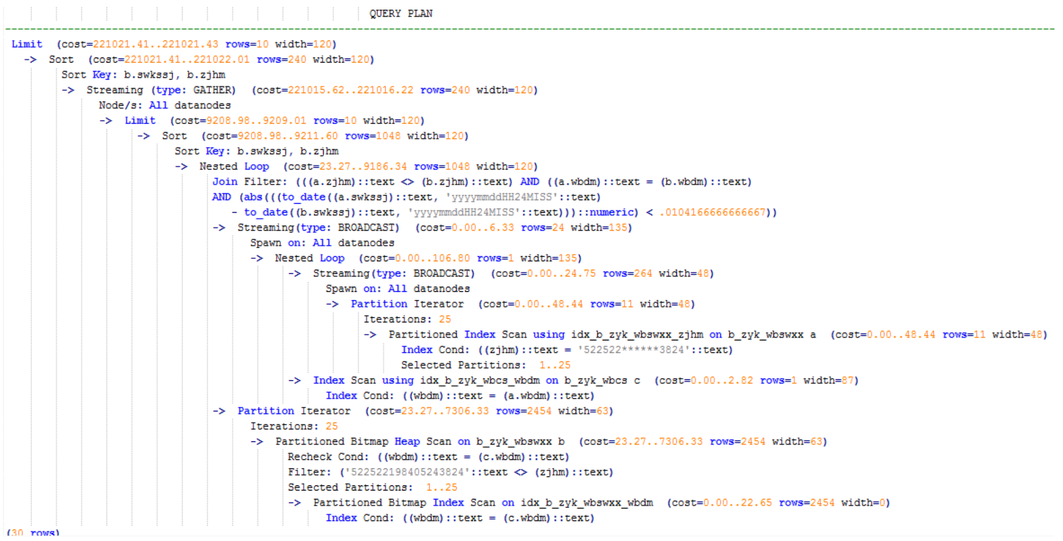
Example 3: Optimization is Not Accurate When Intermediate Results Exist in the Query Where JOIN Is Used for Multiple Tables

Symptom: Query the personnel who have checked in an Internet cafe within 15 minutes before and after the check-in of a specified person.

```
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
;
```

Figure 4-3 shows the execution plan. This query takes about 12s.

Figure 4-3 Using an unlogged table (1)



```
                                                                    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)
```

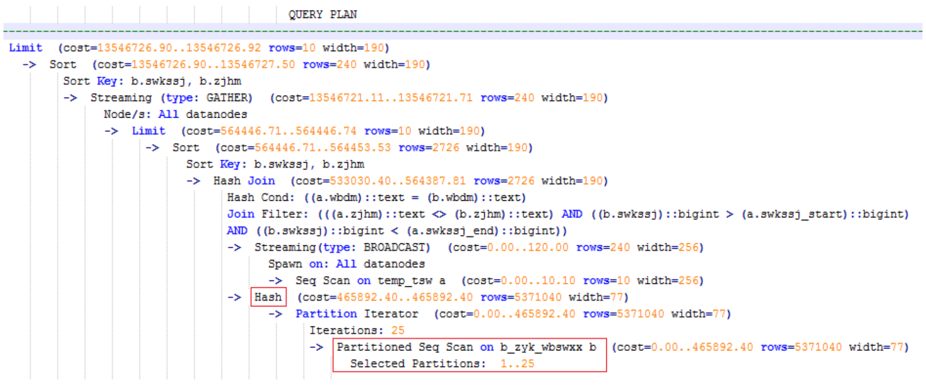
Optimization analysis:

1. In the execution plan, index scan is used for node scanning, the **Join Filter** calculation in the external **NEST LOOP IN** statement consumes most of the query time, and the calculation uses the string addition and subtraction, and unequal-value comparison.
2. Use an unlogged table to record the Internet access time of the specified person. The start time and end time are processed during data insertion, and this reduces subsequent addition and subtraction operations.

```
//Create a temporary 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)
)
;
//Insert the Internet access record of the specified person, and process the start time and end time.
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
;

//Query the personnel who have check in an Internet cafe before and after 15 minutes of the check-in
of the specified person. Convert their ID card number format to int8 in comparison.
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
;
```

The query takes about 7s. **Figure 4-4** shows the execution plan.

Figure 4-4 Using an unlogged table (2)



```
                              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. In the previous plan, **Hash Join** has been executed, and a Hash table has been created for the large table **b_zyk_wbswxx**. The table contains large amounts of data, so the creation takes long time.

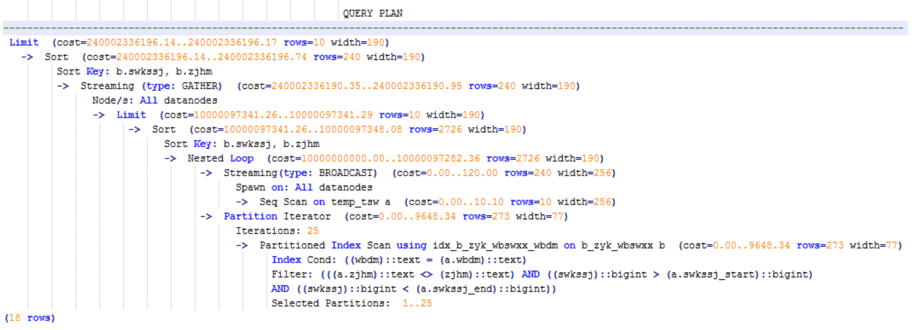
   temp_tsw contains only hundreds of records, and an equal-value connection is created between **temp_tsw** and **b_zyk_wbswxx** using wbdm (the Internet cafe code). Therefore, if **JOIN** is changed to **NEST LOOP JOIN**, index scan can be used for node scanning, and the performance will be boosted.

4. Execute the following statement to change **JOIN** to **NEST LOOP JOIN**.

   ```
   SET enable_hashjoin = off;
   ```

   Figure 4-5 shows the execution plan. The query takes about 3s.

Figure 4-5 Using an unlogged table (3)



```
                                      QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
Limit  (cost=240002336196.14..240002336196.17 rows=10 width=190)
  ->  Sort  (cost=240002336196.14..240002336196.74 rows=240 width=190)
        Sort Key: b.swkssj, b.zjhm
        ->  Streaming (type: GATHER)  (cost=240002336190.35..240002336190.95 rows=240 width=190)
              Node/s: All datanodes
              ->  Limit  (cost=10000097341.26..10000097341.29 rows=10 width=190)
                    ->  Sort  (cost=10000097341.26..10000097348.08 rows=2726 width=190)
                          Sort Key: b.swkssj, b.zjhm
                          ->  Nested Loop  (cost=10000000000.00..10000097282.36 rows=2726 width=190)
                                ->  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)
                                ->  Partition Iterator  (cost=0.00..9648.34 rows=273 width=77)
                                      Iterations: 25
                                      ->  Partitioned Index Scan using idx_b_zyk_wbswxx_wbdm on b_zyk_wbswxx b  (cost=0.00..9648.34 rows=273 width=77)
                                            Index Cond: ((wbdm)::text = (a.wbdm)::text)
                                            Filter: (((a.zjhm)::text <> (zjhm)::text) AND ((swkssj)::bigint > (a.swkssj_start)::bigint)
                                            AND ((swkssj)::bigint < (a.swkssj_end)::bigint))
                                            Selected Partitions:  1..25
(18 rows)
```

5. Save the query result set in the unlogged table for paging display.

   If paging display needs to be achieved on the upper-layer application page, change the **offset** value to determine the result set on the target page. In this way, the previous query statement will be executed every time after a page turning operation, which causes long response latency.

   To resolve this problem, you are advised to use the unlogged table to save the result set.

   ```
   //Create an unlogged table to save the result set.
   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)
);

//Insert the result set to the unlogged table. The insertion takes about 3s.
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
;

//Perform paging query on the result set. The paging query takes about 10 ms.
SELECT
*
FROM
temp_result
ORDER BY
SWKSSJ,
ZJHM
LIMIT 10 OFFSET 0;
```

CAUTION

Collecting global statistics using ANALYZE improves query performance.

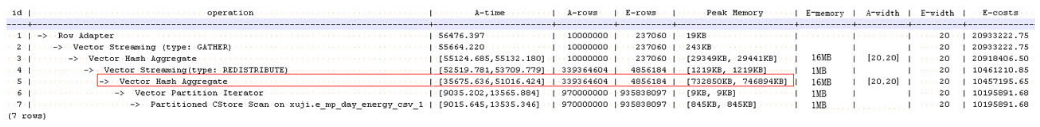
If a performance problem occurs, you can use plan hint to adjust the query plan to the previous one. For details, see **Hint-based Tuning**.

4.5.5 Optimizing Operators

What Is Operator Optimization

A query statement needs to go through multiple operator procedures to generate the final result. Sometimes, the overall query performance deteriorates due to long execution time of certain operators, which are regarded as bottleneck operators. In this case, you need to execute the **EXPLAIN ANALYZE/ PERFORMANCE** command to view the bottleneck operators, and then perform optimization.

For example, in the following execution process, the execution time of the **Hashagg** operator accounts for about 66% [(51016-13535)/56476 ≈ 66%] of the total execution time. Therefore, the **Hashagg** operator is the bottleneck operator for this query. Optimize this operator first.



```
 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)
```

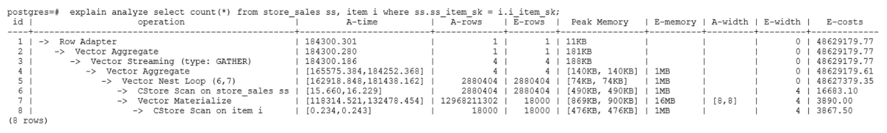
Operator Optimization Example

1. Scan the base table. For queries requiring large volume of data filtering, such as point queries or queries that need range scanning, a full table scan using SeqScan will take a long time. To facilitate scanning, you can create indexes on the condition column and select IndexScan for index scanning.

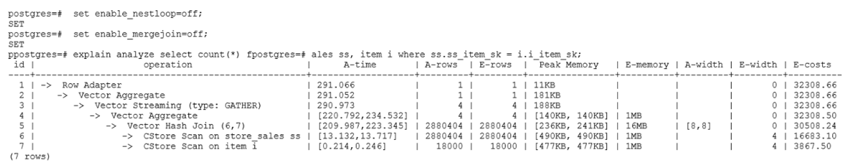
```
 explain (analyze on, costs off) select * from store_sales where ss_sold_date_sk = 2450944;
 id |             operation          |       A-time        | A-rows | Peak Memory  | A-width
----+--------------------------------+---------------------+--------+--------------+---------
  1 | ->  Streaming (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] |
```

In this example, the full table scan filters much data and returns 3360 records. After an index has been created on the **ss_sold_date_sk** column, the scanning efficiency is significantly boosted from 3.6s to 13 ms by using **IndexScan**.

2: If NestLoop is used for joining tables with a large number of rows, the join may take a long time. In the following example, NestLoop takes 181s. If **enable_mergejoin=off** is set to disable merge join and **enable_nestloop=off** is set to disable NestLoop so that the optimizer selects hash join, the join takes more than 200 ms.

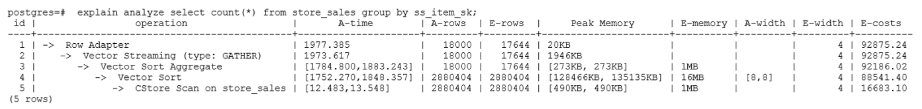


```
postgres=#  explain analyze select count(*) from store_sales ss, item i where ss.ss_item_sk = i.i_item_sk;
 id |                 operation                 |         A-time          |    A-rows   | E-rows  |  Peak Memory   | E-memory | A-width | E-width |    E-costs
----+-------------------------------------------+-------------------------+-------------+---------+----------------+----------+---------+---------+-------------
  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)
```

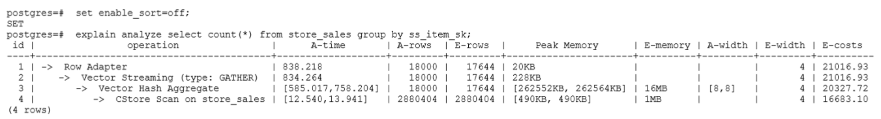


```
postgres=#  set enable_nestloop=off;
SET
postgres=#  set enable_mergejoin=off;
SET
ppostgres=# explain analyze select count(*) fpostgres=# ales ss, item i where ss.ss_item_sk = i.i_item_sk;
 id |                   operation                   |       A-time       | A-rows  | E-rows  |  Peak Memory   | 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)
```

3. Generally, query performance can be improved by selecting **HashAgg**. If **Sort** and **GroupAgg** are used for a large result set, you need to set **enable_sort** to **off**. **HashAgg** consumes less time than **Sort** and **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 | [128466KB, 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 Optimizing Data Skew

Data skew breaks the balance among nodes in the distributed MPP architecture. If the amount of data stored or processed by a node is much greater than that by other nodes, the following problems may occur:

- Storage skew severely limits the system capacity. The skew on a single node hinders system storage utilization.
- Computing skew severely affects performance. The data to be processed on the skew node is much more than that on other nodes, deteriorating overall system performance.
- Data skew severely affects the scalability of the MPP architecture. During storage or computing, data with the same values is often placed on the same node. Therefore, even if we add nodes after a data skew occurs, the skew data (data with the same values) is still placed on a single node, which become the capacity and performance bottleneck of the entire system.

GaussDB(DWS) provides a complete solution for data skew, including storage and computing skew.

Data Skew in the Storage Layer

In the GaussDB(DWS) database, data is distributed and stored on each DN. You can improve the query efficiency by using distributed execution. However, if data skew occurs, bottlenecks exist on some DNs during distribution execution, affecting the query performance. This is because the distribution column is not properly selected. This can be solved by adjusting the distribution column.

For example:

```
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)
```

In the performance information, you can view the number of scan rows of each DN in the inventory table. The number of rows of each DN differs a lot, the biggest is 63000000 and the smallest value is 15000000. This value difference on the performance of data scan is acceptable, but if the join operator exists in the upper-layer, the impact on the performance cannot be ignored.

Generally, the data table is hash distributed on each DN; therefore, it is important to choose a proper distribution column. Run table_skewness() to view data skew of each DN in the inventory table. The query result is as follows:

```
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)
```

The table definition indicates that the table uses the **inv_date_sk** column as the distribution column, which causes a data skew. Based on the data distribution of each column, change the distribution column to **inv_item_sk**. The skew status is as follows:

```
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)
```

Data skew is solved.

In addition to the **table_skewness()** view, you can use the **table_distribution** function and the PGXC_GET_TABLE_SKEWNESS view to efficiently query the data skew of each table.

Data Skew in the Computing Layer

Even if data is balanced across nodes after you change the distribution key of a table, data skew may still occur during a query. If data skew occurs in the result set of an operator on a DN, skew will also occur during the computing that involves the operator. Generally, this is caused by data redistribution during the execution.

During a query, JOIN keys and GROUP BY keys are not used as distribution columns. Data is redistributed among DNs based on the hash values of data on the keys. The redistribution is implemented using the Redistribute operator in an execution plan. Data skew in redistribution columns can lead to data skew during system operation. After the redistribution, some nodes will have much more data, process more data, and will have much lower performance than others.

In the following example, the **s** and **t** tables are joined, and **s.x** and **t.x** columns in the join condition are not their distribution keys. Table data is redistributed using the **REDISTRIBUTE** operator. Data skew occurs in the **s.x** column and not in the **t.x** column. The result set of the **Streaming** operator (**id** being **6**) on datanode2 has data three times that of other DNs and causes a skew.

```
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)
```

It is more difficult to detect skew in computing than in storage. To solve skew in computing, GaussDB provides the Runtime Load Balance Technology (RLBT) solution controlled by the **skew_option** parameter. The RLBT solution addresses how to detect and solve data skew.

1. Detect data skew.

   The solution first checks whether skew data exists in redistribution columns used for computing. RLBT can detect data skew based on statistics, specified hints, or rules.

   – Detection based on statistics

     Run the **ANALYZE** statement to collect statistics on tables. The optimizer will automatically identify skew data on redistribution keys based on the statistics and generate optimization plans for queries having potential skew. When the redistribution key has multiple columns, statistics information can be used for identification only when all columns belong to the same base table.

     The statistics information can only provide the skew of the base table. If a column in the base table is skewed, or other columns have filtering conditions, or after the join of other tables, we cannot determine whether the skewed data still exists on the skewed column. If **skew_option** is set to **normal**, it indicates that data skew persists and the base tables will be optimized to solve the skew. If **skew_option** is set to **lazy**, it indicates that data skew is solved and the optimization will stop.

   – Detection based on specified hints

The intermediate results of complex queries are difficult to estimate based on statistics. In this case, you can specify hints to provide the skew information, based on which the optimizer optimizes queries. For details about the syntax of hints, see **Skew Hints**.

– Detection based on rules

  In a business intelligence (BI) system, a large number of SQL statements having outer joins (including left joins, right joins, and full joins) are generated, and many NULL values will be generated in empty columns that have no match for outer joins. If JOIN or GROUP BY operations are performed on the columns, data skew will occur. RLBT can automatically identify this scenario and generate an optimization plan for NULL value skew.

2. Solve computing skew.

   Join and **Aggregate** operators are optimized to solve skew.

   – **Join** optimization

   Skew and non-skew data is separately processed. Details are as follows:

   a. When redistribution is required on both sides of a join:

      Use **PART_REDISTRIBUTE_PART_ROUNDROBIN** on the side with skew. Specifically, perform round-robin on skew data and redistribution on non-skew data.

      Use **PART_REDISTRIBUTE_PART_BROADCAST** on the side with no skew. Specifically, perform broadcast on skew data and redistribution on non-skew data.

   b. When redistribution is required on only one side of a join:

      Use **PART_REDISTRIBUTE_PART_ROUNDROBIN** on the side where redistribution is required.

      Use **PART_LOCAL_PART_BROADCAST** on the side where redistribution is not required. Specifically, perform broadcast on skew data and retain other data locally.

   c. When a table has **NULL** values padded:

      Use **PART_REDISTRIBUTE_PART_LOCAL** on the table. Specifically, retain the **NULL** values locally and perform redistribution on other data.

   In the example query, the **s.x** column contains skewed data and its value is **0**. The optimizer identifies the skew data in statistics and generates the following optimization plan:

```
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)
```

In the preceding execution plan, **Skew Join Optimized by Statistic** indicates that this is an optimized plan used for handling data skew. The **Statistic** keyword indicates that the plan optimization is based on statistics; **Hint** indicates that the optimization is based on hints; **Rule** indicates that the optimization is based on rules. In this plan, skew and non-skew data is separately processed. Non-skew data in the **s** table is redistributed based on its hash values, and skew data (whose value is **0**) is evenly distributed on all nodes in round-robin mode. In this way, data skew is solved.

To ensure result correctness, the **t** table also needs to be processed. In the **t** table, the data whose value is **0** (skew value in the **s.x** table) is broadcast and other data is redistributed based on its hash values.

In this way, data skew in JOIN operations is solved. The above result shows that the output of the **Streaming** operator (**id** being **6**) is balanced and the end-to-end performance of the query is doubled.

If the stream operator type in the execution plan is **HYBRID**, the stream mode varies depending on the skew data. The following plan is an example:

```
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)
```

Data 1 has skew in the **skew_scol** table. Perform **ROUNDROBIN** on skew data and **REDISTRIBUTE** on non-skew data.

Data 0 is the side with no skew in the **skew_scol** table. Perform **BROADCAST** on skew data and **REDISTRIBUTE** on non-skew data.

As shown in the preceding figure, the two stream types are **PART REDISTRIBUTE PART ROUNDROBIN** and **PART REDISTRIBUTE PART BROADCAST**. In this example, the stream type is **HYBRID**.

– **Aggregate** optimization

For aggregation, data on each DN is deduplicated based on the **GROUP BY** key and then redistributed. After the deduplication on DNs, the global occurrences of each value will not be greater than the number of DNs. Therefore, no serious data skew will occur. Take the following query as an example:

```
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;
```

The command output is as follows:

```
 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)
```

A large amount of skew data exists. As a result, after data is redistributed based on its **GROUP BY** key, the data volume of datanode1 is hundreds of thousands of times that of others. After optimization, a GROUP BY operation is performed on the DN to deduplicate data. After redistribution, no data skew occurs.

```
 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)
```

Applicable scope

– **Join** operator

- **nest loop**, **merge join**, and **hash join** can be optimized.
- If skew data is on the left to the join, **inner join**, **left join**, **semi join**, and **anti join** are supported. If skew data is on the right to the join, **inner join**, **right join**, **right semi join**, and **right anti join** are supported.
- For an optimization plan generated based on statistics, the optimizer checks whether it is optimal by estimating its cost. Optimization plans based on hints or rules are forcibly generated.

- **Aggregate** operator
  - **array_agg**, **string_agg**, and **subplan in agg qual** cannot be optimized.
  - A plan generated based on statistics is affected by its cost, the **plan_mode_seed** parameter, and the **best_agg_plan** parameter. A plan generated based on hints or rules are not affected by them.

4.6 Hint-based Tuning

4.6.1 Plan Hint Optimization

In plan hints, you can specify a join order, join, stream, and scan operations, the number of rows in a result, and redistribution skew information to tune an execution plan, improving query performance.

Function

Plan hints can be specified using the keywords such as **SELECT**, **INSERT**, **UPDATE**, **MERGE**, and **DELETE**, in the following format:

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

You can specify multiple hints for a query plan and separate them by spaces. A hint specified for a query plan does not apply to its subquery plans. To specify a hint for a subquery, add the hint following the keyword of this subquery.

For example:

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

In the preceding command, <*plan_hint1*> and <*plan_hint2*> are the hints of a query, and <*plan_hint3*> is the hint of its subquery.

NOTICE

If a hint is specified in the **CREATE VIEW** statement, the hint will be applied each time this view is used.

If the random plan function is enabled (**plan_mode_seed** is set to a value other than 0), the specified hint will not be used.

Supported Hints

Currently, the following hints are supported:

- Join order hints (**leading**)
- Join operation hints, excluding the **semi join**, **anti join**, and **unique plan** hints
- Rows hints
- Stream operation hints
- Scan operation hints, supporting only **tablescan**, **indexscan**, and **indexonlyscan**

- Sublink name hints
- Skew hints, supporting only the skew in the redistribution involving Join or HashAgg
- Hint used for **Agg** distribution columns Only clusters of 8.1.3.100 and later versions support this function.
- Hint that disables subquery pull-up. Only clusters of 8.2.0 and later versions support this function.
- Configuration parameter hints. For details about supported parameters, see **Configuration Parameter Hints**.

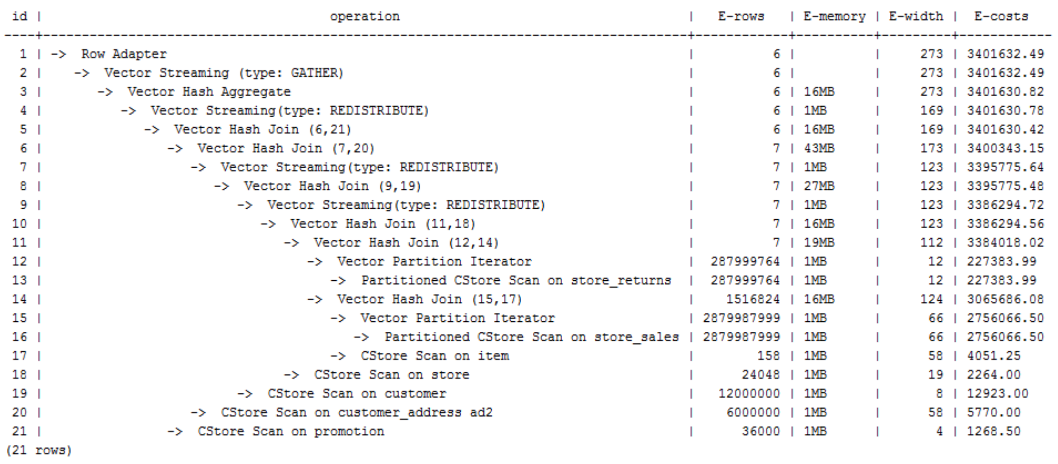
Precautions

- **Sort**, **Setop**, and **Subplan** hints are not supported.
- Hints do not support SMP or Node Groups.
- Hints cannot be used for the target table of the **INSERT** statement.

Examples

The following is the original plan and is used for comparing with the optimized ones:

```
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 Order Hints

Function

Theses hints specify the join order and outer/inner tables.

Syntax

- Specify only the join order.

```
leading(join_table_list)
```

- Specify the join order and outer/inner tables. The outer/inner tables are specified by the outermost parentheses.

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

Parameter Description

join_table_list specifies the tables to be joined. The values can be table names or table aliases. If a subquery is pulled up, the value can also be the subquery alias. Separate the values with spaces. You can add parentheses to specify the join priorities of tables.

NOTICE

A table name or alias can only be a string without a schema name.

An alias (if any) is used to represent a table.

To prevent semantic errors, tables in the list must meet the following requirements:

- The tables must exist in the query or its subquery to be pulled up.
- The table names must be unique in the query or subquery to be pulled up. If they are not, their aliases must be unique.
- A table appears only once in the list.
- An alias (if any) is used to represent a table.

For example:

leading(t1 t2 t3 t4 t5): **t1**, **t2**, **t3**, **t4**, and **t5** are joined. The join order and outer/inner tables are not specified.

leading(t1 t2 t3 t4 t5): **t1**, **t2**, **t3**, **t4**, and **t5** are joined in sequence. The table on the right is used as the inner table in each join.

leading(t1 (t2 t3 t4) t5): First, **t2**, **t3**, and **t4** are joined and the outer/inner tables are not specified. Then, the result is joined with **t1** and **t5**, and the outer/inner tables are not specified.

leading(t1 (t2 t3 t4) t5): First, **t2**, **t3**, and **t4** are joined and the outer/inner tables are not specified. Then, the result is joined with **t1**, and **(t2 t3 t4)** is used as the inner table. Finally, the result is joined with **t5**, and **t5** is used as the inner table.

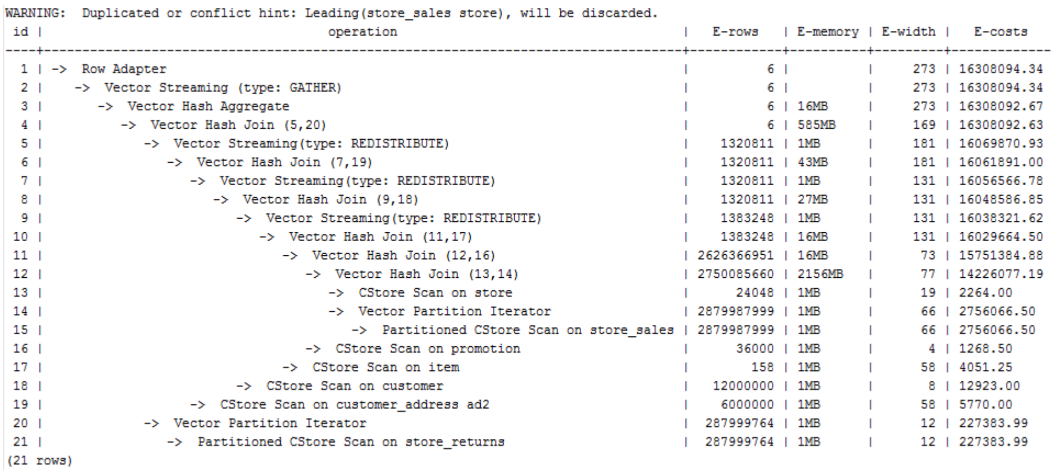
leading((t1 (t2 t3) t4 t5)) leading((t3 t2)): First, **t2** and **t3** are joined and **t2** is used as the inner table. Then, the result is joined with **t1**, and **(t2 t3)** is used as the inner table. Finally, the result is joined with **t4** and then **t5**, and the table on the right in each join is used as the inner table.

Examples

Hint the query plan in **Examples** as follows:

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

First, **store_sales** and **store** are joined and **store_sales** is the inner table. Then, The result is joined with **promotion**, **item**, **customer**, **ad2**, and **store_returns** in sequence. The optimized plan is as follows:



```
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)
```

For details about the warning at the top of the plan, see **Hint Errors, Conflicts, and Other Warnings**.

4.6.3 Join Operation Hints

Function

Specifies the join method. It can be nested loop join, hash join, or merge join.

Syntax

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

Parameter Description

- **no** indicates that the specified hint will not be used for a join.
- *table_list* specifies the tables to be joined. The values are the same as those of **join_table_list** but contain no parentheses.

For example:

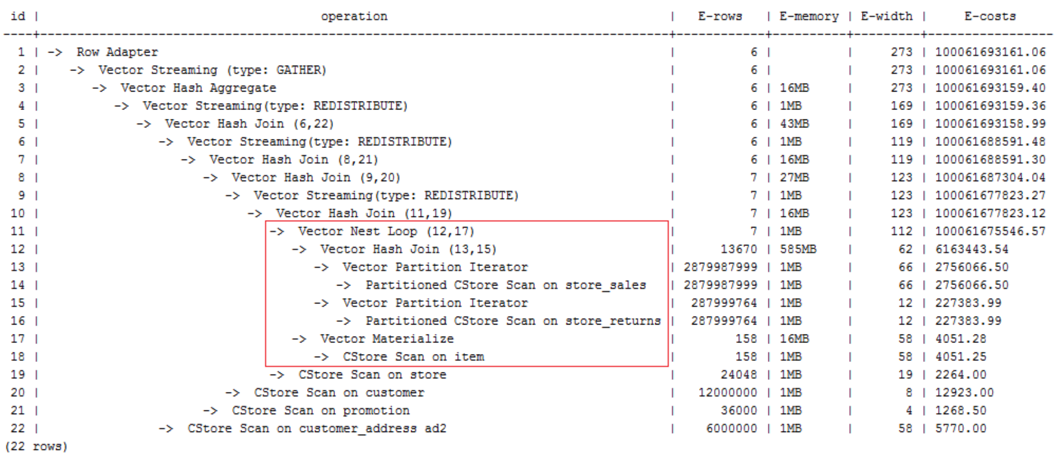
no nestloop(t1 t2 t3): **nestloop** is not used for joining **t1**, **t2**, and **t3**. The three tables may be joined in either of the two ways: Join **t2** and **t3**, and then **t1**; join **t1** and **t2**, and then **t3**. This hint takes effect only for the last join. If necessary, you can hint other joins. For example, you can add **no nestloop(t2 t3)** to join **t2** and **t3** first and to forbid the use of **nestloop**.

Examples

Hint the query plan in **Examples** as follows:

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

nestloop is used for the last join between **store_sales**, **store_returns**, and **item**. The optimized plan is as follows:



```
 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 Rows Hints

Function

These hints specify the number of rows in an intermediate result set. Both absolute values and relative values are supported.

Syntax

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

Parameter Description

- **#**,**+**,**-**, and ***** are operators used for hinting the estimation. **#** indicates that the original estimation is used without any calculation. **+**,**-**, and ***** indicate that the original estimation is calculated using these operators. The minimum calculation result is 1. *table_list* specifies the tables to be joined. The values are the same as those of **table_list** in **Join Operation Hints**.
- *const* can be any non-negative number and supports scientific notation.

For example:

rows(t1 #5): The result set of **t1** is five rows.

rows(t1 t2 t3 *1000): Multiply the result set of joined **t1**, **t2**, and **t3** by 1000.

Suggestion

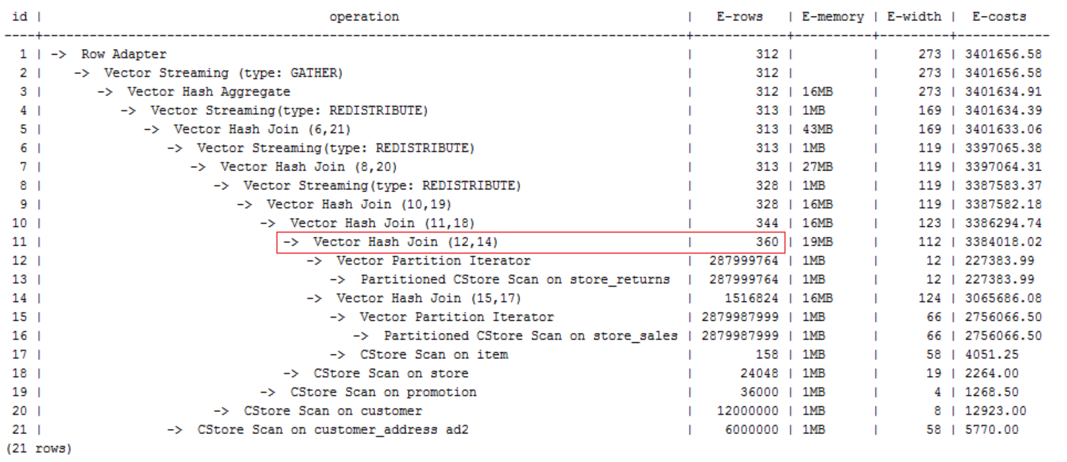
- The hint using ***** for two tables is recommended, because this hint will take effect for a join as long as the two tables appear on both sides of this join. For example, if the hint is **rows(t1 t2 * 3)**, the join result of **(t1 t3 t4)** and **(t2 t5 t6)** will be multiplied by 3 because **t1** and **t2** appear on both sides of the join.
- **rows** hints can be specified for the result sets of a single table, multiple tables, function tables, and subquery scan tables.

Examples

Hint the query plan in **Examples** as follows:

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

Multiply the result set of joined **store_sales** and **store_returns** by 50. The optimized plan is as follows:



```
 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)
```

The estimation value after the hint in row 11 is **360**, and the original value is rounded off to 7.

4.6.5 Stream Operation Hints

Function

Specifies the stream method, which can be broadcast, redistribute, or specifying the distribution key for **Agg** redistribution.

NOTE

Specifies the hint for the distribution column during the Agg process. This parameter is supported only by clusters of version 8.1.3.100 or later.

Syntax

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

Parameter Description

- **no** indicates that the hinted stream method is not used. When the hint is specified for the distribution columns in the **Agg** redistribution, **no** is invalid.
- *table_list* specifies the tables to be joined. For details, see **Parameter Description**.
- When hints are specified for distribution columns, the asterisk (*) is fixed and the table name cannot be specified.
- **columns** specifies one or more columns in the **GROUP BY** clause. When there are no **GROUP BY** clauses, it can specify the columns in the **DISTINCT** clause.

NOTE

- The specified distribution column must be specified using the column sequence number or column name in **group by** or **distinct**. The columns in **count(distinct)** can only be specified using column names.
- For a multi-layer query, you can specify the distribution column hint at each layer. The hint takes effect only at the corresponding layer.
- The column specified in **count(distinct)** takes effect only for two-level hashagg plans. Otherwise, the specified distribution column is invalid.
- If the optimizer finds that redistribution is not required after estimation, the specified distribution column is invalid.

Tips

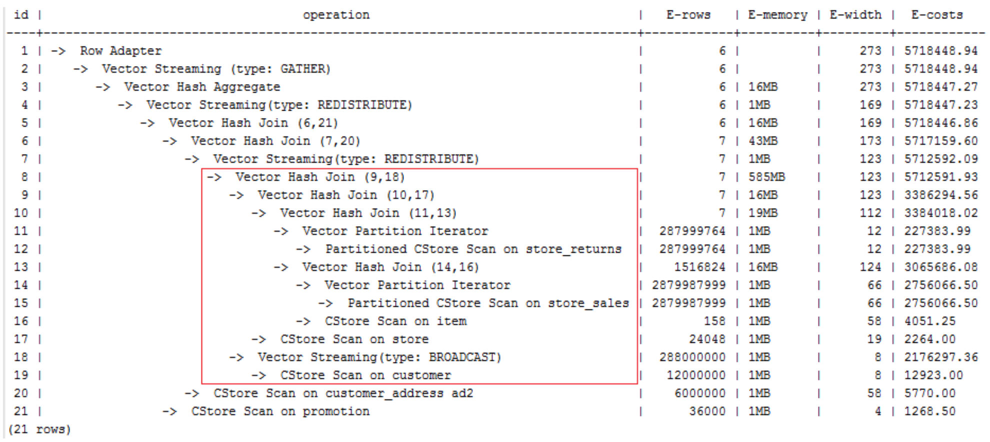
- Generally, the optimizer selects a group of non-skew distribution keys for data redistribution based on statistics. If the default distribution keys have data skew, you can manually specify the distribution columns to avoid data skew.
- When selecting a distribution key, select a group of columns with high distinct values as the distribution key based on data distribution features. In this way, data can be evenly distributed to each DN after redistribution.
- After writing hints, you can run **explain verbose** to print the execution plan and check whether the specified distribution key is valid. If the specified distribution key is invalid, a warning is displayed.

Example

- Hint the query plan in **Examples** as follows:

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

  In the original plan, the join result of **store_sales**, **store_returns**, **item**, and **store** is redistributed before it is joined with **customer**. After the hinting, the redistribution is disabled and the join order is retained. The optimized plan is as follows:

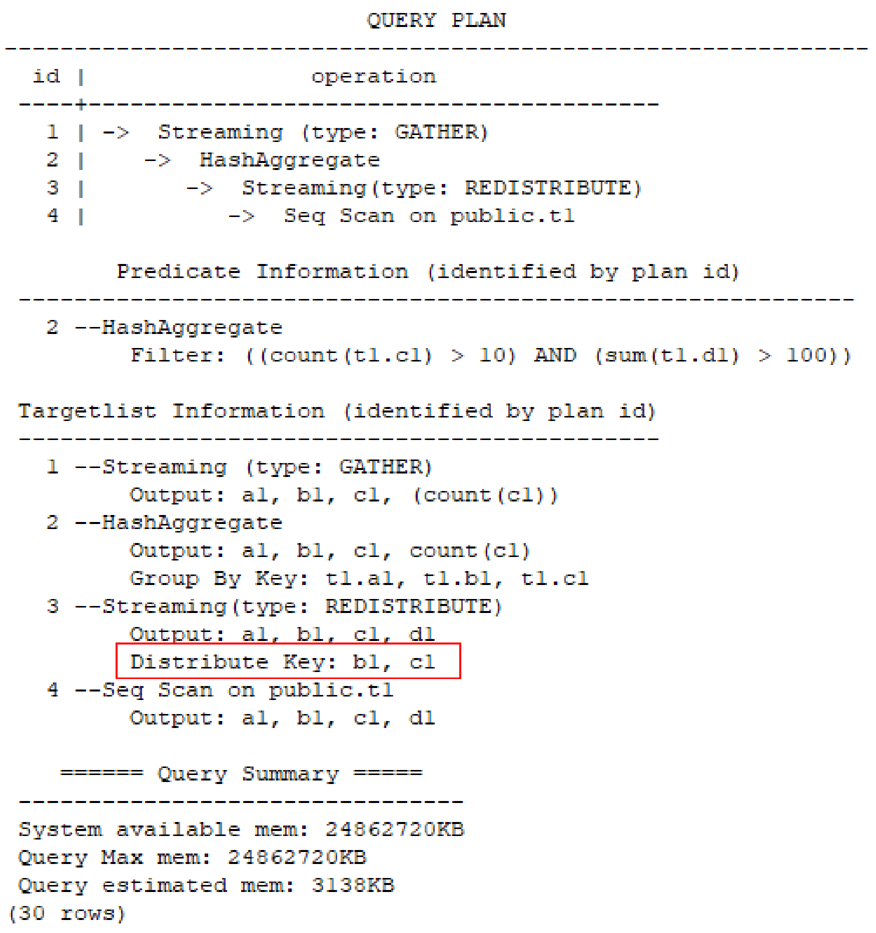


```
 id |                              operation                                        |   E-rows   | E-memory | E-width |  E-costs
----+-------------------------------------------------------------------------------+------------+----------+---------+------------
  1 | ->  Row Adapter                                                               |          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)
```

- Specifies the distribution columns for Agg redistribution.

```
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
```

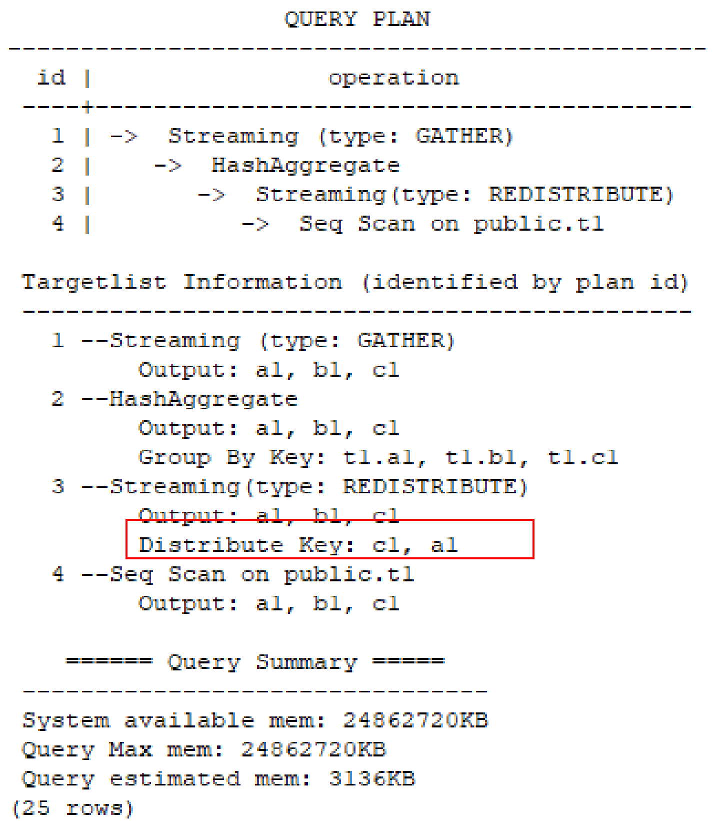
  In the following example, the last two columns of the specified **GROUP BY** columns are used as distribution keys.



```
                          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)
```

- If the statement does not contain the **GROUP BY** clause, specify the distinct column as the distribution columns.

```
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 Operation Hints

Function

These hints specify a scan operation, which can be **tablescan**, **indexscan**, or **indexonlyscan**.

Syntax

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

Parameter Description

- **no** indicates that the specified hint will not be used for a join.
- *table* specifies the table to be scanned. You can specify only one table. Use a table alias (if any) instead of a table name.
- *index* indicates the index for **indexscan** or **indexonlyscan**. You can specify only one index.

NOTE

indexscan and **indexonlyscan** hints can be used only when the specified index belongs to the table.

Scan operation hints can be used for row-store tables, column-store tables, HDFS tables, HDFS foreign tables, OBS tables, and subquery tables. HDFS tables include primary tables and delta tables. The delta tables are invisible to users. Therefore, scan operation hints are used only for primary tables.

If **indexscan** is specified, **indexscan** or **indexonlyscan** takes effect. **indexscan** and **indexonlyscan** can also take effect at the same time. When **indexscan** and **indexonlyscan hints** appear at the same time, **indexonlyscan** takes effect first.

Example

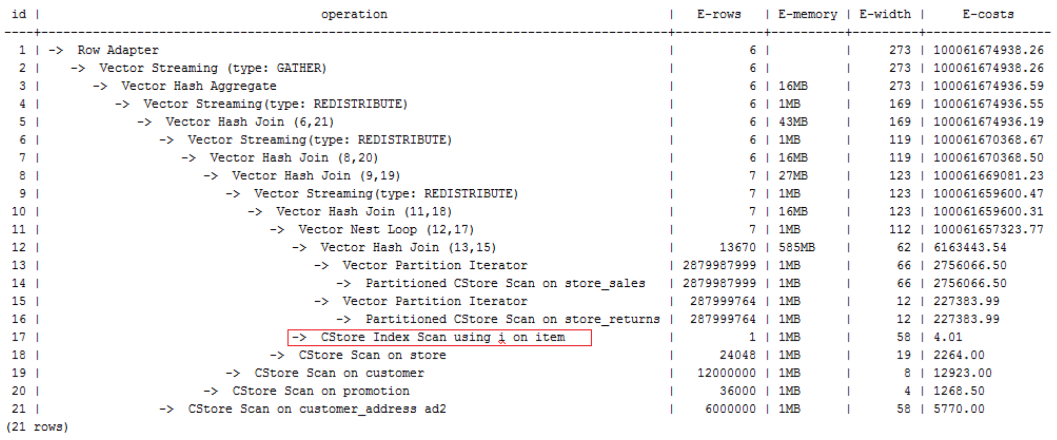
To specify an index-based hint for a scan, create an index named **i** on the **i_item_sk** column of the **item** table.

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

Hint the query plan in **Examples** as follows:

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

item is scanned based on an index. The optimized plan is as follows:



```
 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 Sublink Name Hints

Function

These hints specify the name of a sublink block.

Syntax

```
blockname (table)
```

Parameter Description

- *table* indicates the name you have specified for a sublink block.

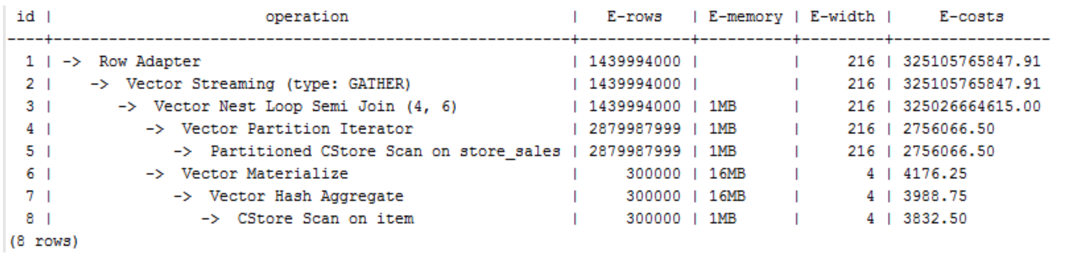
NOTE

- This hint is used by an outer query only when a sublink is pulled up. Currently, only the **Agg** equivalent join, **IN**, and **EXISTS** sublinks can be pulled up. This hint is usually used together with the hints described in the previous sections.
- The subquery after the **FROM** keyword is hinted by using the subquery alias. In this case, **blockname** becomes invalid.
- If a sublink contains multiple tables, the tables will be joined with the outer-query tables in a random sequence after the sublink is pulled up. In this case, **blockname** also becomes invalid.

Examples

```
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);
```

tt indicates the sublink block name. After being pulled up, the sublink is joined with the outer-query table **store_sales** by using **nestloop**. The optimized plan is as follows:



```
 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 Skew Hints

Function

Theses hints specify redistribution keys containing skew data and skew values, and are used to optimize redistribution involving Join or HashAgg.

Syntax

- Specify single-table skew.
  ```
  skew(table (column) [(value)])
  ```
- Specify intermediate result skew.
  ```
  skew((join_rel) (column) [(value)])
  ```

Parameter Description

- **table** specifies the table where skew occurs.
- **join_rel** specifies two or more joined tables. For example, **(t1 t2)** indicates that the result of joining **t1** and **t2** tables contains skew data.
- **column** specifies one or more columns where skew occurs.
- **value** specifies one or more skew values.

NOTE

- Skew hints are used only if redistribution is required and the specified skew information matches the redistribution information.
- Skew hints are controlled by the GUC parameter **skew_option**. If the parameter is disabled, skew hints cannot be used for solving skew.
- Currently, skew hints support only the table relationships of the ordinary table and subquery types. Hints can be specified for base tables, subqueries, and **WITH ... AS** clauses. Unlike other hints, a subquery can be used in skew hints regardless of whether it is pulled up.
- Use an alias (if any) to specify a table where data skew occurs.
- You can use a name or an alias to specify a skew column as long as it is not ambiguous. The columns in skew hints cannot be expressions. If data skew occurs in the redistribution that uses an expression as a redistribution key, set the redistribution key as a new column and specify the column in skew hints.
- The number of skew values must be an integer multiple of the number of columns. Skew values must be grouped based on the column sequence, with each group containing a maximum of 10 values. You can specify duplicate values to group skew columns having different number of skew values. For example, the **c1** and **c2** columns of the **t1** table contains skew data. The skew value of the **c1** column is **a1**, and the skew values of the **c2** column are **b1** and **b2**. In this case, the skew hint is **skew(t1 (c1 c2) ((a1 b1)(a1 b2)))**. **(a1 b1)** is a value group, where **NULL** is allowed as a skew value. Each hint can contain a maximum of 10 groups and the number of groups should be an integer multiple of the number of columns.
- In the redistribution optimization of Join, a skew value must be specified for skew hints. The skew value can be left empty for HashAgg.
- If multiple tables, columns, or values are specified, separate items of the same type with spaces.
- The type of skew values cannot be forcibly converted in hints. To specify a string, enclose it with single quotation marks (' ').

Example:

- Specify single-table skew.

  Each skew hint describes the skew information of one table relationship. To describe the skews of multiple table relationships in a query, specify multiple skew hints.

  Skew hints have the following formats:

  – One skew value in one column: **skew(t (c1) (v1))**

    Description: The **v1** value in the **c1** column of the **t** table relationship causes skew in query execution.

  – Multiple skew values in one column: **skew(t (c1) (v1 v2 v3 ...))**

    Description: Values including **v1, v2**, and **v3** in the **c1** column of the **t** table relationship cause skew in query execution.

  – Multiple columns, each having one skew value: **skew(t (c1 c2) (v1 v2))**

    Description: The **v1** value in the **c1** column and the **v2** value in the **c2** column of the **t** table relationship cause skew in query execution.

  – Multiple columns, each having multiple skew values: **skew(t (c1 c2) ((v1 v2) (v3 v4) (v5 v6) ...))**

    Description: Values including **v1, v3**, and **v5** in the **c1** column and values including **v2, v4**, and **v6** in the **c2** column of the **t** table relationship cause skew in query execution.

NOTICE

In the last format, parentheses for skew value groups can be omitted, for example, **skew(t (c1 c2) (v1 v2 v3 v4 v5 v6 ...))**. In a skew hint, either use parentheses for all skew value groups or for none of them.

Otherwise, a syntax error will be generated. For example, **skew(t (c1 c2) (v1 v2 v3 v4 (v5 v6) ...))** will generate an error.

- Specify intermediate result skew.

  If data skew does not occur in base tables but in an intermediate result during query execution, specify skew hints of the intermediate result to solve the skew. The format is **skew((t1 t2) (c1) (v1))**.

  Description: Data skew occurs after the table relationships **t1** and **t2** are joined. The **c1** column of the **t1** table contains skew data and its skew value is **v1**.

  c1 can exist only in a table relationship of **join_rel**. If there is another column having the same name, use aliases to avoid ambiguity.

Suggestion

- For a multi-level query, write the hint on the layer where data skew occurs.
- For a listed subquery, you can specify the subquery name in a hint. If you know data skew occurs on which base table, directly specify the table.
- Aliases are preferred when you specify a table or column in a hint.

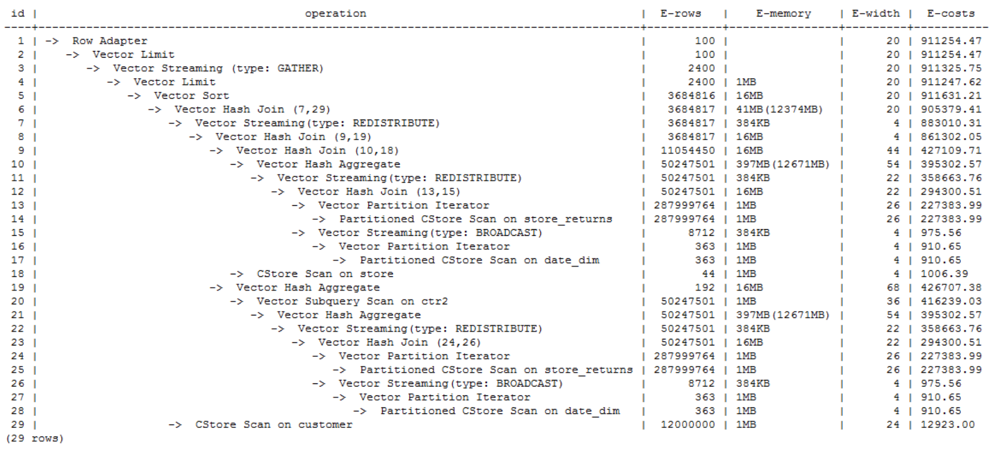
Examples

Specify single-table skew.

- Specify hints in the original query.

  For example, the original query is as follows:

```
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;
```

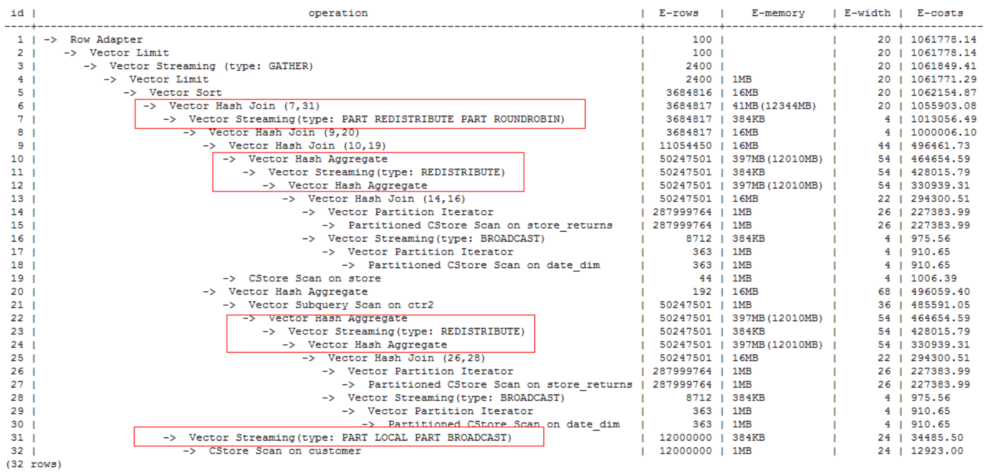


```
 id |                                 operation                                  |  E-rows   |    E-memory     | E-width |  E-costs
----+----------------------------------------------------------------------------+-----------+-----------------+---------+-----------
  1 | ->  Row Adapter                                                            |       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)
```

Specify the hints of HashAgg in the inner **with** clause and of the outer Hash Join. The query containing hints is as follows:

```
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;
```

The hints indicate that the **group by** in the inner **with** clause contains skew data during redistribution by HashAgg, corresponding to the original Hash Agg operators 10 and 21; and that the **ctr_customer_sk** column in the outer **ctr1** table contains skew data during redistribution by Hash Join, corresponding to operator 6 in the original plan. The optimized plan is as follows:



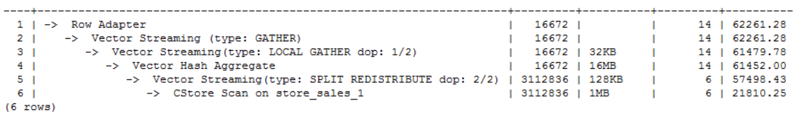
```
 id |                                     operation                                     |  E-rows   |    E-memory    | E-width |  E-costs
----+-----------------------------------------------------------------------------------+-----------+----------------+---------+-----------
  1 | ->  Row Adapter                                                                    |       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)
```

To solve data skew in the redistribution, Hash Agg is changed to double-level Agg operators and the redistribution operators used by Hash Join are changed in the optimized plan.

- Modify the query and then specify hints.

  For example, the original query and its plan are as follows:

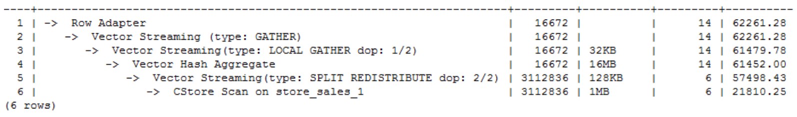
```
explain select count(*) from store_sales_1 group by round(ss_list_price);
```



```
----+---------------------------------------------------------------+---------+----------+---------+----------
  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)
```

  Columns in hints do not support expressions. To specify hints, rewrite the query as several subqueries. The rewritten query and its plan are as follows:

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

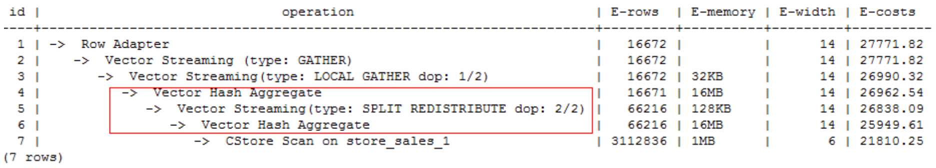


```
----+---------------------------------------------------------------+---------+----------+---------+----------
  1 | ->  Row Adapter                                               |   16672 |          |      14 | 62261.28
  2 |    ->  Vector Streaming (type: GATHER)                        |   16672 |          |      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)
```

  Ensure that the service logic is not changed during the rewriting.

  Specify hints in the rewritten query as follows:

```
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)
```

  The plan shows that after Hash Agg is changed to double-layer Agg operators, redistributed data is greatly reduced and redistribution time shortened.

  You can specify hints in columns in a subquery, for example:

```
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 Hint That Disables Subquery Pull-up

Function

To optimize query logic, the optimizer usually pulls up subqueries for execution. However, sometimes the pulled up subqueries do not run much faster than others, and may even be slower due to enlarged search scope. In this case, you can specify the **no merge** hint to disable pull-up. This hint is not recommended in most cases.

Syntax

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

Description

subquery_name indicates the name of a subquery. It can also be a view or CTE name. The specified subquery will not be unnested during logic optimization. If **subquery_name** is not specified, the current query will not be unnested.

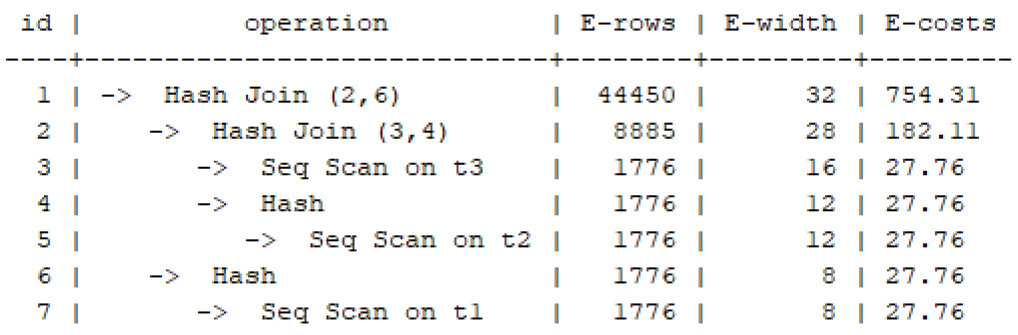
Example

Create tables **t1**, **t2**, and **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);
```

The original statement is as follows:

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

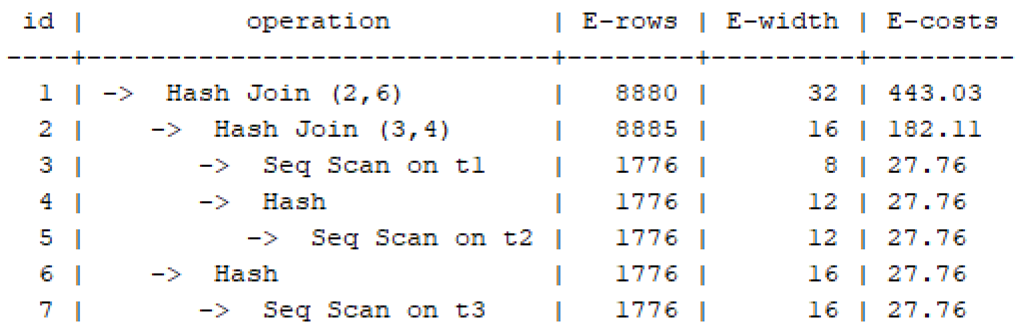


```
 id |          operation           | E-rows | E-width | E-costs
----+------------------------------+--------+---------+---------
  1 | ->  Hash Join (2,6)          |  44450 |      32 | 754.31
  2 |    ->  Hash Join (3,4)       |   8885 |      28 | 182.11
  3 |       ->  Seq Scan on t3     |   1776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

In this query, you can use the following methods to disable the pull-up of subquery **s1**:

- Method 1:
  ```
  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;
  ```
- Method 2:
  ```
  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;
  ```

Outcome:



```
 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 Configuration Parameter Hints

Function

A hint, or a GUC hint, specifies a configuration parameter value when a plan is generated.

Syntax

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

Parameters

- **global** indicates that the parameter set by hint takes effect at the statement level. If **global** is not specified, the parameter takes effect only in the subquery where the hint is located.
- **guc_name** indicates the name of the configuration parameter specified by hint.
- **guc_value** indicates the value of a configuration parameter specified by hint.

NOTE

- If a parameter set by hint takes effect at the statement level, the hint must be written to the top-level query instead of the subquery. For **UNION**, **INTERSECT**, **EXCEPT**, and **MINUS** statements, you can write the GUC hint at the statement level to any **SELECT** clause that participates in the set operation. The configuration parameters set by the GUC hint take effect on each **SELECT** clause that participates in the set operation.
- When a subquery is pulled up, all GUC hints on the subquery are discarded.
- If a parameter is set by both the statement-level GUC hint and the subquery-level GUC hint, the subquery-level GUC hint takes effect in the corresponding subquery, and the statement-level GUC hint takes effect in other subqueries of the statement.

Currently, GUC hints support only some configuration parameters. Some parameters cannot be configured at the subquery level and can only be configured at the statement level. The following table lists the supported parameters.

Table 4-1 Configuration parameters supported by GUC hints

Parameter	Configured at the Subquery Level (Yes/No)
agg_max_mem	Yes
agg_redistribute_enhancement	Yes
best_agg_plan	Yes
cost_model_version	No
cost_param	No
enable_bitmapscan	Yes
enable_broadcast	Yes
enable_redistribute	Yes
enable_extrapolation_stats	Yes
enable_fast_query_shipping	No
enable_force_vector_engine	No
enable_hashagg	Yes
enable_hashjoin	Yes
enable_index_nestloop	Yes
enable_indexscan	Yes
enable_join_pseudoconst	Yes
enable_nestloop	Yes
enable_nodegroup_debug	No
enable_partition_dynamic_pruning	Yes
enable_sort	Yes
enable_stream_ctescan	No
enable_value_redistribute	Yes
enable_vector_engine	No
expected_computing_nodegroup	No
force_bitmapand	Yes
from_collapse_limit	Yes
join_collapse_limit	Yes
join_num_distinct	Yes
outer_join_max_rows_multipler	Yes

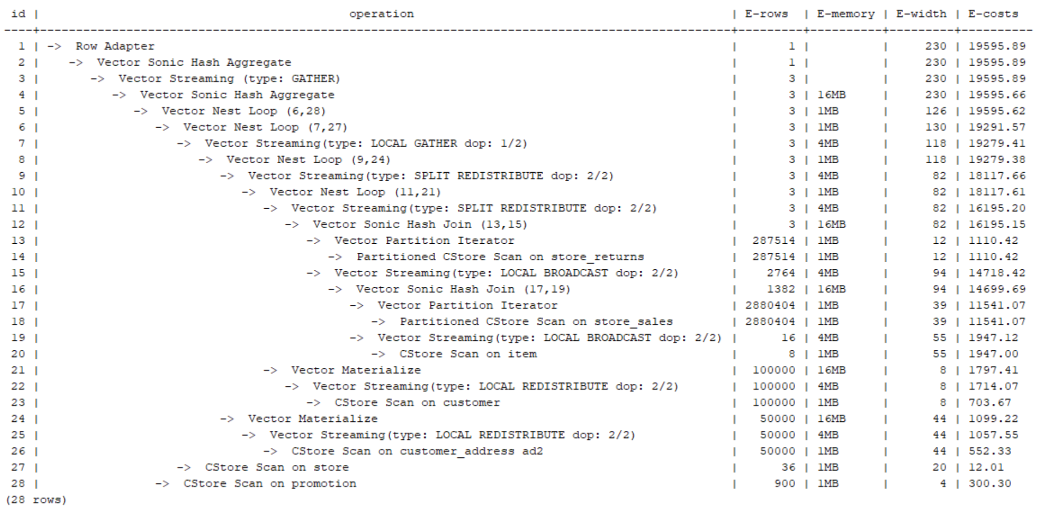
Parameter	Configured at the Subquery Level (Yes/No)
prefer_hashjoin_path	No
qrw_inlist2join_optmode	Yes
qual_num_distinct	Yes
query_dop	No
query_max_mem	No
query_mem	No
rewrite_rule	No
setop_optmode	Yes
skew_option	Yes
index_cost_limit	Yes

Examples

Hint the query plan in **Examples** as follows:

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

This hint indicates that the **query_dop** parameter is set to **0** when the plan for a statement is generated, which means the SMP adaptation function is enabled. The generated plan is as follows:



```
 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 Errors, Conflicts, and Other Warnings

Plan hints change an execution plan. You can run **EXPLAIN** to view the changes.

Hints containing errors are invalid and do not affect statement execution. The errors will be displayed in different ways based on statement types. Hint errors in an **EXPLAIN** statement are displayed as a warning on the interface. Hint errors in other statements will be recorded in debug1-level logs containing the **PLANHINT** keyword.

Hint Error Types

- Syntax errors.

  An error will be reported if the syntax tree fails to be reduced. The No. of the row generating an error is displayed in the error details.

  For example, the hint keyword is incorrect, no table or only one table is specified in the **leading** or **join** hint, or no tables are specified in other hints. The parsing of a hint is terminated immediately after a syntax error is detected. Only the hints that have been parsed successfully are valid.

  For example:

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

  The syntax of **nestloop(t1)** is wrong and its parsing is terminated. Only **leading(t1 t2)** that has been successfully parsed before **nestloop(t1)** is valid.

- Semantic errors.
  - An error will be reported if the specified tables do not exist, multiple tables are found based on the hint setting, or a table is used more than once in the **leading** or **join** hint.
  - An error will be reported if the index specified in a scan hint does not exist.
  - If multiple tables with the same name exist after a subquery is pulled up and some of them need to be hinted, add aliases for them to avoid name duplication.

- Duplicated or conflicted hints.

  If hint duplication or conflicts occur, only the first hint takes effect. A message will be displayed to describe the situation.

  - Hint duplication indicates that a hint is used more than once in the same query, for example, **nestloop(t1 t2) nestloop(t1 t2)**.
  - A hint conflict indicates that the functions of two hints with the same table list conflict with each other.

    For example, if **nestloop (t1 t2) hashjoin (t1 t2)** is used, **hashjoin (t1 t2)** becomes invalid. **nestloop(t1 t2)** does not conflict with **no mergejoin(t1 t2)**.

    NOTICE

    The table list in the **leading** hint is disassembled. For example, **leading (t1 t2 t3)** will be disassembled as **leading(t1 t2) leading((t1 t2) t3)**, which will conflict with **leading(t2 t1)** (if any). In this case, the latter **leading(t2 t1)** becomes invalid. If two hints use duplicated table lists and only one of them has the specified outer/inner table, the one without a specified outer/inner table becomes invalid.

- A hint becomes invalid after a sublink is pulled up.

In this case, a message will be displayed. Generally, such invalidation occurs if a sublink contains multiple tables to be joined, because the table list in the sublink becomes invalid after the sublink is pulled up.

- Unsupported column types.
  - Skew hints are specified to optimize redistribution. They will be invalid if their corresponding columns do not support redistribution.
- Specified hints are not used.
  - If **hashjoin** or **mergejoin** is specified for non-equivalent joins, it will not be used.
  - If **indexscan** or **indexonlyscan** is specified for a table that does not have an index, it will not be used.
  - If **indexscan hint** or **indexonlyscan** is specified for a full-table scan or for a scan whose filtering conditions are not set on index columns, it will not be used.
  - The specified **indexonlyscan** hint is used only when the output column contains only indexes.
  - In equivalent joins, only the joins containing equivalence conditions are valid. Therefore, the **leading**, **join**, and **rows** hints specified for the joins without an equivalence condition will not be used. For example, **t1**, **t2**, and **t3** are to be joined, and the join between **t1** and **t3** does not contain an equivalence condition. In this case, **leading(t1 t3)** will not be used.
  - To generate a streaming plan, if the distribution key of a table is the same as its join key, **redistribute** specified for this table will not be used. If the distribution key and join key are different for this table but the same for the other table in the join, **redistribute** specified for this table will be used but **broadcast** will not.
  - If a hint for an **Agg** distribution column is not used, the possible causes are as follows:
    - The specified distribution key contains data types that do not support redistribution.
    - Redistribution is not required in the execution plan.
    - Wrong distribution key sequence numbers are executed.
    - For AP functions that use the GROUPING SETS and CUBE clauses, hints are not supported for distribution keys in window aggregate functions .

      NOTE

      Specifies the hint for the distribution column druing the Agg process.. This parameter is supported only by clusters of version 8.1.3.100 or later.
  - If no sublink is pulled up, the specified **blockname** hint will not be used.
  - For unused skew hints, the possible causes are:
    - The plan does not require redistribution.
    - The columns specified by hints contain distribution keys.

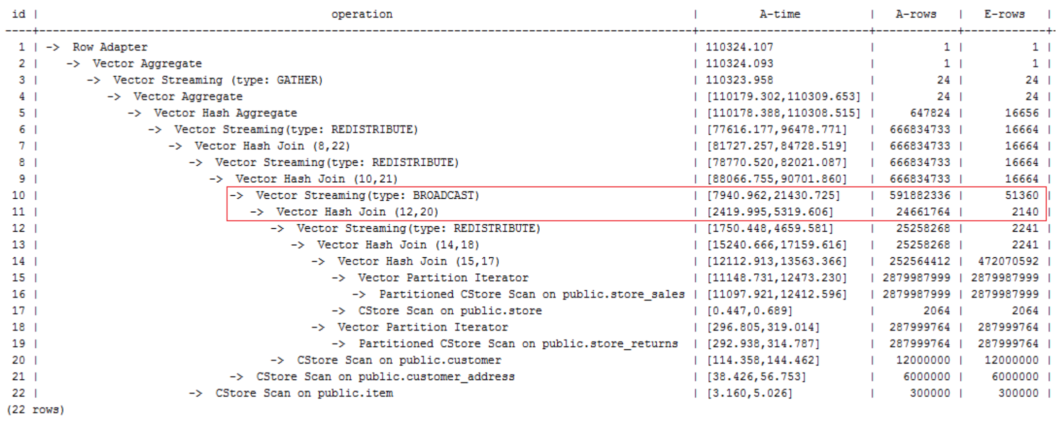
        - Skew information specified in hints is incorrect or incomplete, for example, no value is specified for join optimization.
        - Skew optimization is disabled by GUC parameters.
    - For unused guc hints, the possible causes are:
        - The configuration parameter does not exist.
        - The configuration parameter is not supported by GUC hints.
        - The configuration parameter value is invalid.
        - The statement-level GUC hint is not written in the top-level query.
        - The configuration parameter set by the GUC hint at the subquery level cannot be set at the subquery level.
        - The subquery where the GUC hint is located is pulled up.

4.6.12 Plan Hint Cases

This section takes the statements in TPC-DS (Q24) as an example to describe how to optimize an execution plan by using hints in 1000X+24DN environments. For example:

```
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. The original plan of this statement is as follows and the statement execution takes 110s:

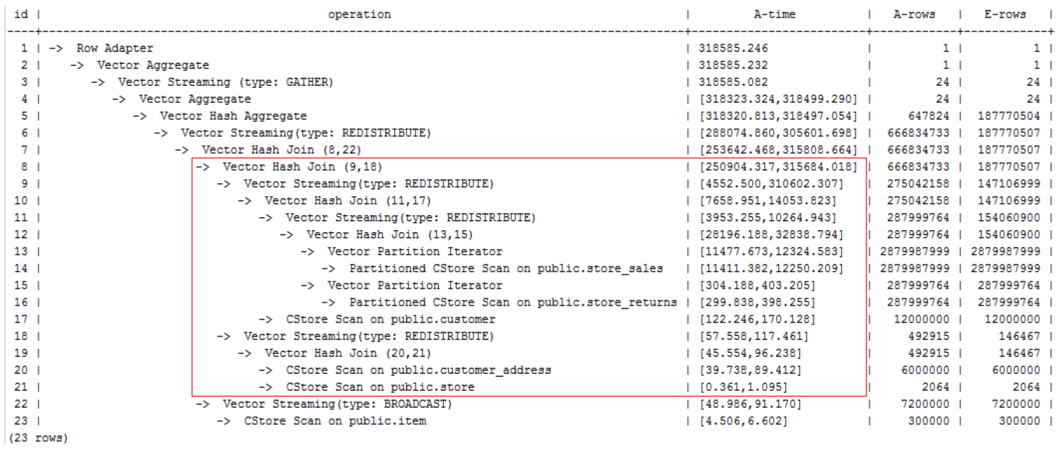


```
 id |                                     operation                                      |         A-time          |   A-rows   |   E-rows   |
----+------------------------------------------------------------------------------------+-------------------------+------------+------------+
  1 | -> Row 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)
```

In this plan, the performance of the layer-10 **broadcast** is poor because the estimation result generated at layer 11 is 2140 rows, which is much less than the actual number of rows. The inaccurate estimation is mainly caused by the underestimated number of rows in layer-13 hash join. In this layer, **store_sales** and **store_returns** are joined (based on the **ss_ticket_number** and **ss_item_sk** columns in **store_sales** and the **sr_ticket_number** and **sr_item_sk** columns in **store_returns**) but the multi-column correlation is not considered.

2. After the **rows** hint is used for optimization, the plan is as follows and the statement execution takes 318s:

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

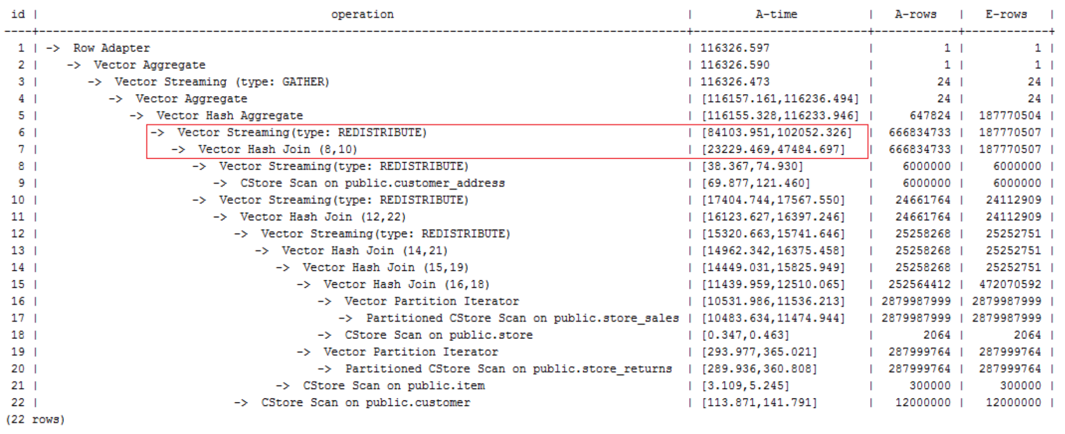


```
 id |                                     operation                                      |         A-time          |   A-rows   |   E-rows   |
----+------------------------------------------------------------------------------------+-------------------------+------------+------------+
  1 | -> Row 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)
```

The execution takes a longer time because layer-9 **redistribute** is slow. Considering that data skew does not occur at layer-9 **redistribute**, the slow redistribution is caused by the slow layer-8 **hashjoin** due to data skew at layer-18 **redistribute**.

3. Data skew occurs at layer-18 **redistribute** because **customer_address** has a few different values in its two join keys. Therefore, plan **customer_address** as the last one to be joined. After the hint is used for optimization, the plan is as follows and the statement execution takes 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 ...
```

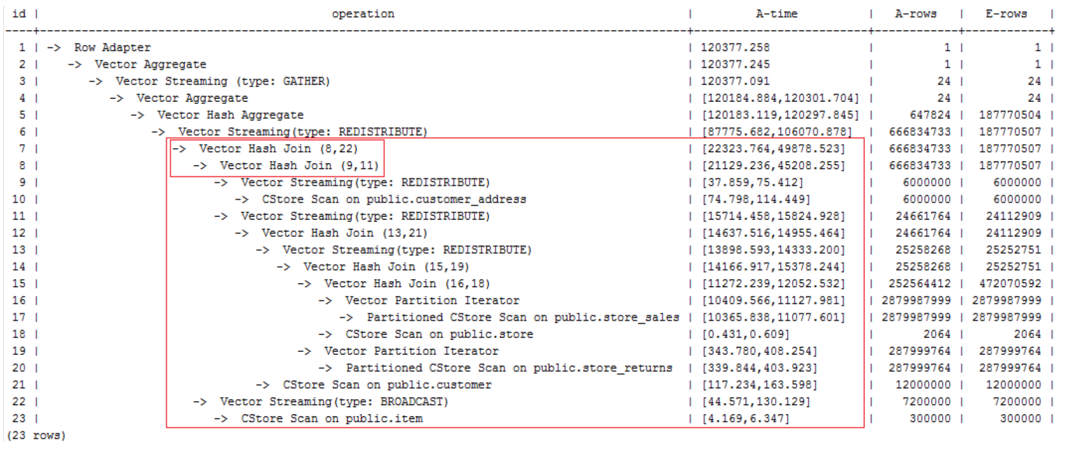


```
 id |                                operation                                 |         A-time          |   A-rows   |   E-rows   |
----+--------------------------------------------------------------------------+-------------------------+------------+------------+
  1 | ->  Row Adapter                                                          | 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)
```

Most of the time is spent on layer-6 **redistribute**. The plan needs to be further optimized.

4. Most of the time is spent on layer-6 **redistribute** because of data skew. To avoid the data skew, plan the **item** table as the last one to be joined because the number of rows is not reduced after **item** is joined. After the hint is used for optimization, the plan is as follows and the statement execution takes 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 ...
```

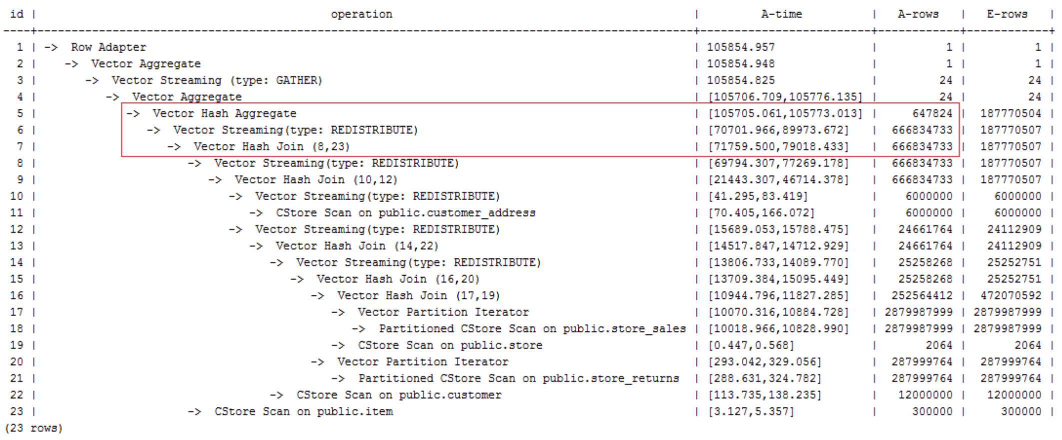


```
 id |                                operation                                 |         A-time          |   A-rows   |   E-rows   |
----+--------------------------------------------------------------------------+-------------------------+------------+------------+
  1 | ->  Row Adapter                                                          | 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)
```

Data skew occurs after the join of **item** and **customer_address** because **item** is broadcasted at layer-22. As a result, layer-6 **redistribute** is still slow.

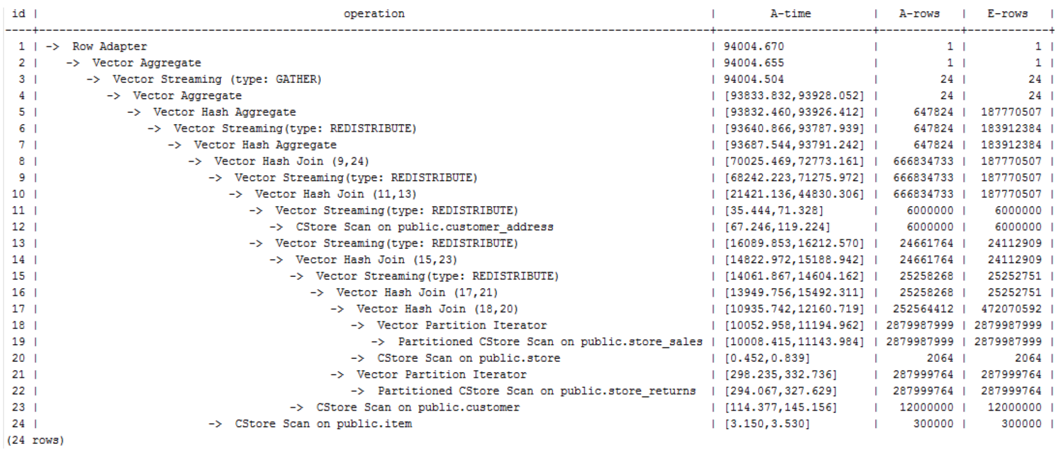
5. Add a hint to disable **broadcast** for **item** or add a **redistribute** hint for the join result of **item** and **customer_address**. After the hint is used for optimization, the plan is as follows and the statement execution takes 105s:

```
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)
```

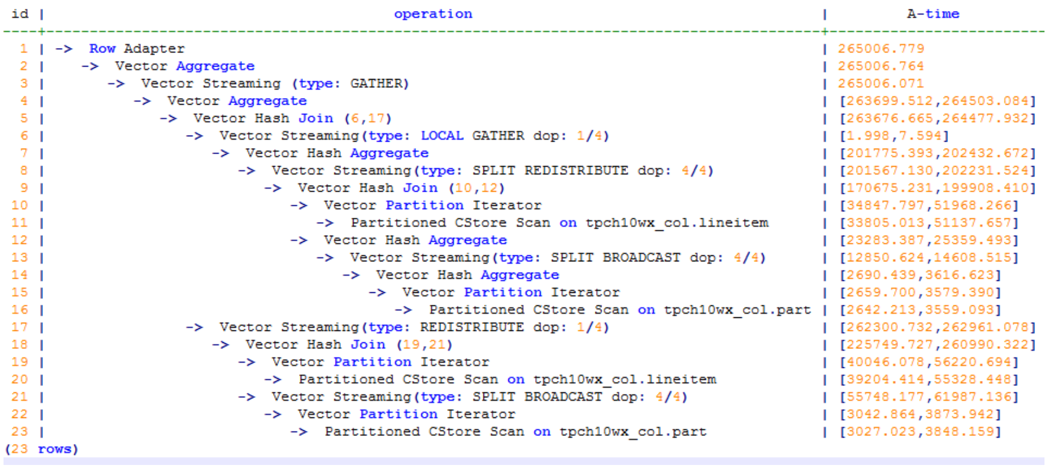
6. The last layer uses single-layer **Agg** and the number of rows is greatly reduced. Set **best_agg_plan** to **3** and change the single-layer **Agg** to a double-layer **Agg**. The plan is as follows and the statement execution takes 94s. The optimization ends.



```
 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)
```

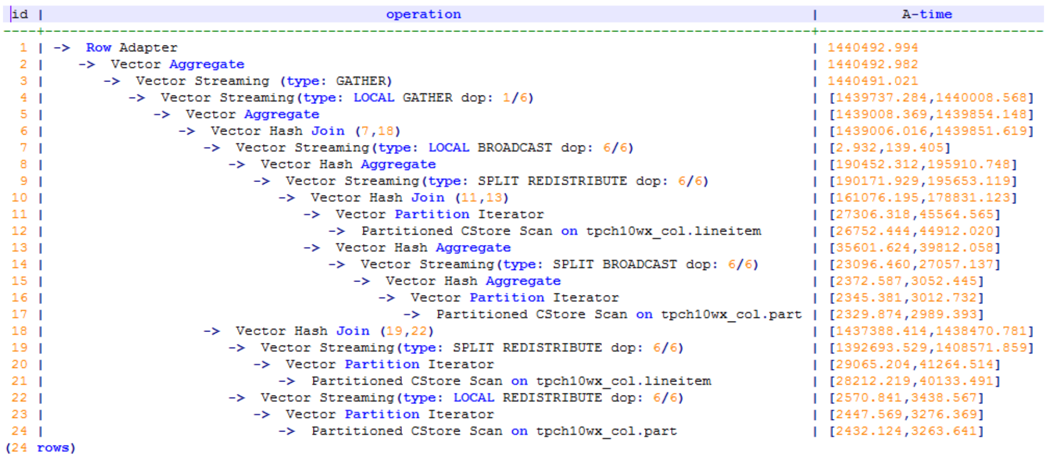
If the query performance deteriorates due to statistics changes, you can use hints to optimize the query plan. Take TPCH-Q17 as an example. The query performance deteriorates after the value of **default_statistics_target** is changed from the default one to **–2** for statistics collection.

1. If **default_statistics_target** is set to the default value **100**, the plan is as follows:



```
 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. If **default_statistics_target** is set to **–2**, the plan is as follows:



```
 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. After the analysis, the cause is that the stream type is changed from **BroadCast** to **Redistribute** during the join of the **lineitem** and **part** tables. You can use a hint to change the stream type back to **BroadCast**. For example:

```
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 Routinely Maintaining Tables

To ensure proper database running, after INSERT and DELETE operations, you need to routinely do **VACUUM FULL** and **ANALYZE** as appropriate for customer scenarios and update statistics to obtain better performance.

Related Concepts

You need to routinely run **VACUUM**, **VACUUM FULL**, and **ANALYZE** to maintain tables, because:

- **VACUUM FULL** reclaims disk space occupied by updated or deleted data and combines small-size data files.

- **VACUUM** maintains a visualized mapping to track pages that contain arrays visible to other active transactions. A common index scan uses the mapping to obtain the corresponding array and check whether pages are visible to the current transaction. If the array cannot be obtained, the visibility is checked by fetching stack arrays. Therefore, updating the visible mapping of a table can accelerate unique index scans.
- **VACUUM** can avoid old data loss caused by duplicate transaction IDs when the number of executed transactions exceeds the database threshold.
- **ANALYZE** collects statistics on tables in databases. The statistics are stored in the PG_STATISTIC system catalog. Then, the query optimizer uses the statistics to work out the most efficient execution plan.

Procedure

Step 1 Run the **VACUUM** or **VACUUM FULL** command to reclaim disk space.

- **VACUUM**:

  Do **VACUUM** to the table:
  ```
  VACUUM customer;
  VACUUM
  ```
  This command can be concurrently executed with database operation commands, including **SELECT**, **INSERT**, **UPDATE**, and **DELETE**; excluding **ALTER TABLE**.

  Do **VACUUM** to the partitioned table:
  ```
  VACUUM customer_par PARTITION ( P1 );
  VACUUM
  ```
- VACUUM FULL:
  ```
  VACUUM FULL customer;
  VACUUM
  ```
  VACUUM FULL needs to add exclusive locks on tables it operates on and requires that all other database operations be suspended.

  When reclaiming disk space, you can query for the session corresponding to the earliest transactions in the cluster, and then end the earliest long transactions as needed to make full use of the disk space.

  a. Run the following command to query for oldestxmin on the GTM:
  ```
  select * from pgxc_gtm_snapshot_status();
  ```
  b. Run the following command to query for the PID of the corresponding session on the CN. *xmin* is the oldestxmin obtained in the previous step.
  ```
  select * from pgxc_running_xacts() where xmin=1400202010;
  ```

Step 2 Do **ANALYZE** to update statistical information.
```
ANALYZE customer;
ANALYZE
```
Do **ANALYZE VERBOSE** to update statistics and display table information.
```
ANALYZE VERBOSE customer;
ANALYZE
```
You can use **VACUUM ANALYZE** at the same time to optimize the query.
```
VACUUM ANALYZE customer;
VACUUM
```

NOTE

VACUUM and **ANALYZE** cause a substantial increase in I/O traffic, which may cause poor performance of other active sessions. Therefore, you are advised to set by specifying the **vacuum_cost_delay** parameter.

Step 3 Delete a table

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

If the following output is displayed, the index has been deleted.

```
DROP TABLE
```

----End

Maintenance Suggestion

- Routinely do **VACUUM FULL** to large tables. If the database performance deteriorates, do **VACUUM FULL** to the entire database. If the database performance is stable, you are advised to monthly do **VACUUM FULL**.
- Routinely do **VACUUM FULL** to system catalogs, mainly **PG_ATTRIBUTE**.
- The automatic vacuum process (**AUTOVACUUM**) in the system automatically runs the **VACUUM** and **ANALYZE** statements to reclaim the record space marked as the deleted state and to update statistics related to the table.

4.8 Routinely Recreating an Index

Context

When data deletion is repeatedly performed in the database, index keys will be deleted from the index page, resulting in index distention. Recreating an index routinely improves query efficiency.

The database supports B-tree, GIN, and psort indexes.

- Recreating a B-tree index helps improve query efficiency.
  - If massive data is deleted, index keys on the index page will be deleted. As a result, the number of index pages reduces and index bloat occurs. Recreating an index helps reclaim wasted space.
  - In the created index, pages adjacent in its logical structure are adjacent in its physical structure. Therefore, a created index achieves higher access speed than an index that has been updated for multiple times.
- You are advised not to recreate a non-B-tree index.

Rebuilding an Index

Use either of the following two methods to recreate an index:

- Run the **DROP INDEX** statement to delete an index and run the **CREATE INDEX** statement to create an index.

  When you delete an index, a temporary exclusive lock is added in the parent table to block related read/write operations. When you create an index, the write operation is locked but the read operation is not. The data is read and scanned by order.

- Run the **REINDEX** statement to recreate an index:
  - When you run the **REINDEX TABLE** statement to recreate an index, an exclusive lock is added to block related read/write operations.
  - When you run the **REINDEX INTERNAL TABLE** statement to recreate an index for a **desc** table (), an exclusive lock is added to block read/write operations on the table.

Procedure

Assume the ordinary index areaS_idx exists in the **area_id** column of the imported table **areaS**. Use either of the following two methods to recreate an index:

- Run the **DROP INDEX** statement to delete the index and run the **CREATE INDEX** statement to create an index.
  a. Delete an index.
  ```
  DROP INDEX areaS_idx;
  DROP INDEX
  ```
  b. Create an index.
  ```
  CREATE INDEX areaS_idx ON areaS (area_id);
  CREATE INDEX
  ```
- Run the **REINDEX** statement to recreate an index.
  - Run the **REINDEX TABLE** statement to recreate an index.
  ```
  REINDEX TABLE areaS;
  REINDEX
  ```
  - Run the **REINDEX INTERNAL TABLE** statement to recreate an index for a **desc** table ().
  ```
  REINDEX INTERNAL TABLE areaS;
  REINDEX
  ```

4.9 Adjusting Key Parameters During SQL Tuning

This section describes key CN parameters that affect GaussDB(DWS) SQL optimization performance. For details about how to configure these parameters, see section **Configuring GUC Parameters** in the .

Table 4-2 CN parameters

Parameter/ Reference Value	Description
enable_nestloop=on	Specifies how the optimizer uses **Nest Loop Join**. If this parameter is set to **on**, the optimizer preferentially uses **Nest Loop Join**. If it is set to **off**, the optimizer preferentially uses other methods, if any. **NOTE** To temporarily change the value of this parameter in the current database connection (that is, the current session), run the following SQL statement: `SET enable_nestloop to off;` By default, this parameter is set to **on**. Change the value as required. Generally, nested loop join has the poorest performance among the three **JOIN** methods (nested loop join, merge join, and hash join). You are advised to set this parameter to **off**.
enable_bitmapscan=on	Specifies whether the optimizer uses bitmap scanning. If the value is **on**, bitmap scanning is used. If the value is **off**, it is not used. **NOTE** If you only want to temporarily change the value of this parameter during the current database connection (that is, the current session), run the following SQL statements: `SET enable_bitmapscan to off;` The bitmap scanning applies only in the query condition where **a > 1 and b > 1** and indexes are created on columns **a** and **b**. During performance tuning, if the query performance is poor and bitmapscan operators are in the execution plan, set this parameter to **off** and check whether the performance is improved.
enable_fast_query_shipping=on	Specifies whether the optimizer uses a distribution framework. If the value is **on**, the execution plan is generated on both CNs and DNs. If the value is **off**, the distribution framework is used, that is, the execution plan is generated on the CNs and then sent to DNs for execution. **NOTE** To temporarily change the value of this parameter in the current database connection (that is, the current session), run the following SQL statement: `SET enable_fast_query_shipping to off;`
enable_hashagg=on	Specifies whether to enable the optimizer's use of Hash-aggregation plan types.
enable_hashjoin=on	Specifies whether to enable the optimizer's use of Hash-join plan types.
enable_mergejoin=on	Specifies whether to enable the optimizer's use of Hash-merge plan types.

Parameter/ Reference Value	Description
enable_indexscan=on	Specifies whether to enable the optimizer's use of index-scan plan types.
enable_indexonlyscan=on	Specifies whether to enable the optimizer's use of index-only-scan plan types.
enable_seqscan=on	Specifies whether the optimizer uses bitmap scanning. It is impossible to suppress sequential scans entirely, but setting this variable to **off** allows the optimizer to preferentially choose other methods if available.
enable_sort=on	Specifies the optimizer sorts. It is impossible to fully suppress explicit sorts, but setting this variable to **off** allows the optimizer to preferentially choose other methods if available.
enable_broadcast=on	Specifies whether enable the optimizer's use of data broadcast. In data broadcast, a large amount of data is transferred on the network. When the number of transmission nodes (stream) is large and the estimation is inaccurate, set this parameter to **off** and check whether the performance is improved.
enable_redistribute=on This parameter is supported only by clusters of version 8.2.1.300 or later.	Controls the query optimizer's use of data transmission in **local redistribute** and **split redistribute** redistribution modes. This parameter corresponds to **enable_broadcast**. The optimizer may overestimate the cost of **local broadcast** and **split broadcast**. As a result, the optimizer selects the **local redistribute** or **split redistribute** redistribution plan. This may cause performance deterioration. Therefore, when the actual data volume of the network transmission node (stream) is small, you can set this parameter to **off** so that the optimizer preferentially selects the **broadcast** mode. Then you can check whether the performance is improved.
rewrite_rule	Specifies whether the optimizer enables a specific rewriting rule.

4.10 Configuring SMP

4.10.1 Application Scenarios and Restrictions

Context

The SMP feature improves the performance through operator parallelism and occupies more system resources, including CPU, memory, network, and I/O. Actually, SMP is a method consuming resources to save time. It improves system performance in appropriate scenarios and when resources are sufficient, but may deteriorate performance otherwise. In addition, compared with the serial processing, SMP generates more candidate plans, which is more time-consuming and may deteriorate performance.

Applicable Scenarios

- Operators supporting parallel processing are used.

  The execution plan contains the following operators:

  a. Scan: Row Storage common table and a line memory partition table sequential scanning, column-oriented storage ordinary table and column-oriented storage partition table sequential scanning, HDFS internal and external table sequence scanning. Surface scanning GDS data can be imported at the same time. All of the above does not support replication tables.
  b. Join: HashJoin, NestLoop
  c. Agg: HashAgg, SortAgg, PlainAgg, and WindowAgg, which supports only **partition by**, and does not support **order by**.
  d. Stream: Redistribute, Broadcast
  e. Other: Result, Subqueryscan, Unique, Material, Setop, Append, VectoRow, RowToVec

- SMP-unique operators

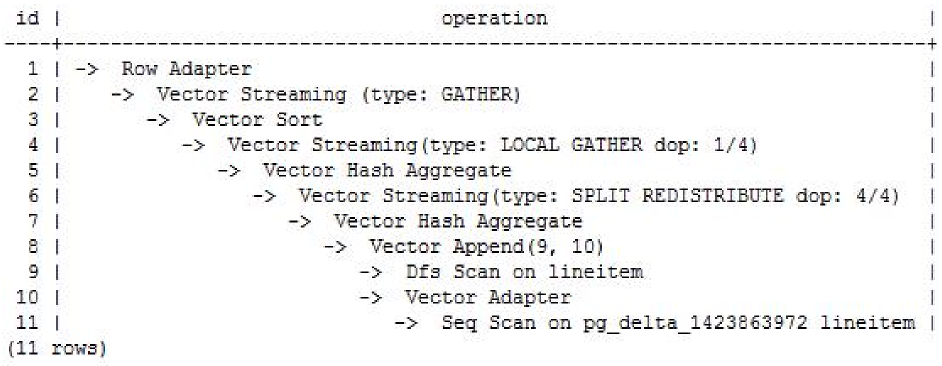
  To execute queries in parallel, Stream operators are added for data exchange of the SMP feature. These new operators can be considered as the subtypes of Stream operators.

  a. Local Gather aggregates data of parallel threads within a DN
  b. Local Redistribute redistributes data based on the distributed key across threads within a DN
  c. Local Broadcast broadcasts data to each thread within a DN.
  d. Local RoundRobin distributes data in polling mode across threads within a DN.
  e. Split Redistribute redistributes data across parallel threads on different DNs.
  f. Split Broadcast broadcasts data to all parallel DN threads in the cluster.

  Among these operators, Local operators exchange data between parallel threads within a DN, and non-Local operators exchange data across DNs.

- Example

  The TPCH Q1 parallel plan is used as an example.



```
 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)
```

In this plan, implement the Hdfs Scan and HashAgg operator parallel, and adds the Local Gather and Split Redistribute data exchange operator.

In this example, the sixth operator is Split Redistribute, and **dop: 4/4** next to the operator indicates that the degree of parallelism of the sender and receiver is 4. 4 No operator is Local Gather, marked dop: 1/4 above, this operator sender thread parallel degree is 4, while the receiving end thread parallelism degree to 1, that is, lower-layer 5 number Hash Aggregate operators according to the 4 parallel degree, while the working mode of the port on the upper-layer 1 to 3 number operator according to the executed one by one, 4 number operator is used to achieve intra-DN concurrent threads data aggregation.

You can view the parallelism situation of each operator in the dop information.

Non-applicable Scenarios

1. Short query operations are performed, where the plan generation is time-consuming.
2. Operators are processed on CNs.
3. Statements that cannot be pushed down are executed.
4. The **subplan** of a query and operators containing a subquery are executed.

4.10.2 Resource Impact on SMP Performance

The SMP architecture uses abundant resources to obtain time. After the plan parallelism is executed, the resource consumption is added, including the CPU, memory, I/O, and network bandwidth resources. As the parallelism degree is expanded, the resource consumption increases. If these resources become a bottleneck, the SMP cannot improve the performance and the overall cluster performance may be deteriorated. Adaptive SMP is provided to dynamically select the optimal parallel degree for each query based on the resource usage and query requirements. The following information describes the situations that the SMP affects theses resources:

- **CPU resources**

  In a general customer scenario, the system CPU usage rate is not high. Using the SMP parallelism architecture will fully use the CPU resource to improve the system performance. If the number of CPU kernels of the database server is too small and the CPU usage is already high, enabling the SMP parallelism may deteriorate the system performance due to resource compete between multiple threads.

- **Memory resources**

  The query parallel causes memory usage growth, but the memory upper limit used by each operator is still restricted by **work_mem**. Assume that **work_mem** is 4 GB, and the degree of parallelism is 2, then the memory upper limit of each concurrent thread is 2 GB. When **work_mem** is small or the system memory is sufficient, running SMP parallelism may push data down to disks. As a result, the query performance deteriorates.

- **Network bandwidth resources**

  To execute a query in parallel, data exchange operators are added. Local Stream operators exchange data between threads within a DN. Data is exchanged in memory and network performance is not affected. Non-Local operators exchange data over the network and increase network load. If the capacity of a network resource becomes a bottleneck, parallelism may also increase the network load.

- **I/O resources**

  A parallel scan increases I/O resource consumption. It can improve performance only when I/O resources are sufficient.

4.10.3 Other Factors Affecting SMP Performance

Besides resource factors, there are other factors that impact the SMP parallelism performance, such as unevenly data distributed in a partitioned table and system parallelism degree.

- **Impact of data skew on SMP performance**

  Serious data skew deteriorates parallel execution performance. For example, if the data volume of a value in the join column is much more than that of other values, the data volume of a parallel thread will be much more than that of others after Hash-based data redistribution, resulting in the long-tail issue and poor parallelism performance.

- **Impact on the SMP performance due to system parallelism degree**

  The SMP feature uses more resources, and unused resources are decreasing in a high concurrency scenario. Therefore, enabling the SMP parallelism will result in serious resource compete among queries. Once resource competes occur, no matter the CPU, I/O, memory, or network resources, all of them will result in entire performance deterioration. In the high concurrency scenario, enabling the SMP will not improve the performance effect and even may cause performance deterioration.

4.10.4 Suggestions for SMP Parameter Settings

To enable the SMP adaptation function, set **query_dop** to **0** and adjust the following parameters to obtain an optimal DOP selection:

- comm_usable_memory

  If the system memory is large, the value of **max_process_memory** is large. In this case, you are advised to set the value of this parameter to 5% of **max_process_memory**, that is, 4 GB by default.

- comm_max_stream

  The recommended value for this parameter is calculated as follows: comm_max_stream = Min(dop_limit x dop_limit x 20 x 2, max_process_memory (bytes) x 0.025/Number of DNs/260). The value must be within the value range of **comm_max_stream**.

- max_connections

  The recommended value for this parameter is calculated as follows: max_connections = dop_limit x 20 x 6 + 24. The value must be within the value range of **max_connections**.

CAUTION

In the preceding formulas, **dop_limit** indicates the number of CPUs corresponding to each DN in the cluster. It is calculated as follows: **dop_limit** = Number of logical CPU cores of a single server/Number of DNs of a single server.

4.10.5 SMP Manual Optimization Suggestions

To manually optimize SMP, you need to be familiar with **Suggestions for SMP Parameter Settings**. This section describes how to optimize SMP.

Constraints

The CPU, memory, I/O, and network bandwidth resources are sufficient. The SMP architecture uses abundant resources to save time. After the plan parallelism is executed, resource consumption increases. When these resources become a bottleneck, SMP may deteriorate, rather than improve performance. In addition, it takes a longer time to generate SMP plans than serial plans. Therefore, in TP services that mainly involve short queries or in case resources are insufficient, you are advised to disable SMP by setting **query_dop** to **1**.

Procedure

1. Observe the current system load situation. If the resource is sufficient (the resource usage ratio is smaller than 50%), perform step 2. Otherwise, exit this system.
2. Set **query_dop** to **1** (default value). Use **explain** to generate an execution plan and check whether the plan can be used in scenarios in **Application Scenarios and Restrictions**. If the plan can be used, go to the next step.
3. Set **query_dop=-***value*. The value range of the parallelism degree is [1, *value*].
4. Set **query_dop=***value*. The parallelism degree is 1 or *value*.
5. Before the query statement is executed, set **query_dop** to an appropriate value. After the statement is executed, set **query_dop** to **off**. For example:

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

NOTE

- If resources are enough, the higher the parallelism degree is, the better the performance improvement effect is.
- The SMP parallelism degree supports a session level setting and you are advised to enable the SMP before executing the query that meets the requirements. After the execution is complete, disable the SMP. Otherwise, SMP may affect services in peak hours.
- SMP adaptation (**query_dop** ≤ 0) depends on resource management. If resource management is disabled (use_workload_manager is **off**), plans with parallelism degree of only 1 or 2 are generated.

4.11 Querying SQL Statements That Affect Performance Most

This section describes how to query SQL statements whose execution takes a long time, leading to poor system performance.

Procedure

Step 1 Query the statements that are run for a long time in the database.

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

After the query, query statements are returned as a list, ranked by execution time in descending order. The first result is the query statement that has the longest execution time in the system. The returned result contains the SQL statement invoked by the system and the SQL statement run by users. Find the statements that were run by users and took a long time.

Alternatively, you can set **current_timestamp - query_start** to be greater than a threshold to identify query statements that are executed for a duration longer than this threshold.

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

Step 2 Set the parameter **track_activities** to **on**.

```
SET track_activities = on;
```

The database collects the running information about active queries only if the parameter is set to **on**.

Step 3 View the running query statements.

Viewing **pg_stat_activity** is used as an example here.

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

If the **state** column is idle, the connection is idle and requires a user to enter a command.

To identify only active query statements, run the following command:

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

Step 4 Analyze the status of the query statements that were run for a long time.

- If the query statement is normal, wait until the execution is complete.
- If a query statement is blocked, run the following command to view this query statement:

```
SELECT datname, usename, state, query FROM pg_stat_activity WHERE waiting = true;
```

The command output lists a query statement in the block state. The lock resource requested by this query statement is occupied by another session, so this query statement is waiting for the session to release the lock resource.

NOTE

Only when the query is blocked by internal lock resources, the **waiting** field is **true**. In most cases, block happens when query statements are waiting for lock resources to be released. However, query statements may be blocked because they are waiting to write in files or for timers. Such blocked queries are not displayed in the **pg_stat_activity** view.

----End

5 Optimization Cases

5.1 Case: Selecting an Appropriate Distribution Column

Distribution columns are used to distribute data to different nodes. A proper distribution key can avoid data skew.

When performing join query, you are advised to select the join condition in the query as the distribution key. When a join condition is used as a distribution key, related data is distributed locally on DNs, reducing the cost of data flow between DNs and improving the query speed.

Before optimization

Use **a** as the distribution column of **t1** and **t2**. The table definition is as follows:

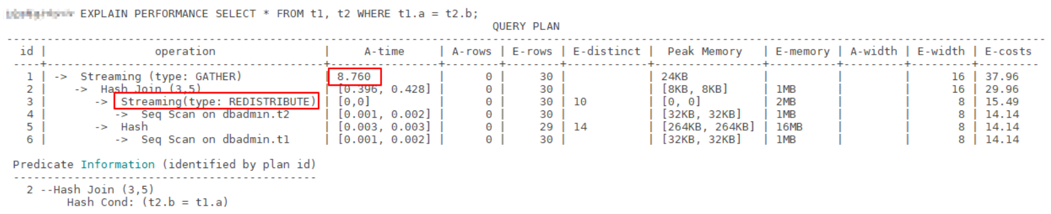
```
CREATE TABLE t1 (a int, b int) DISTRIBUTE BY HASH (a);
CREATE TABLE t2 (a int, b int) DISTRIBUTE BY HASH (a);
```

The following query is executed:

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

In this case, the execution plan contains **Streaming(type: REDISTRIBUTE)**, that is, the DN redistributes data to all DNs based on the selected column. This will cause a large amount of data to be transmitted between DNs, as shown in **Figure 5-1**.

Figure 5-1 Selecting an appropriate distribution column (1)



```
[redacted] 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)
```

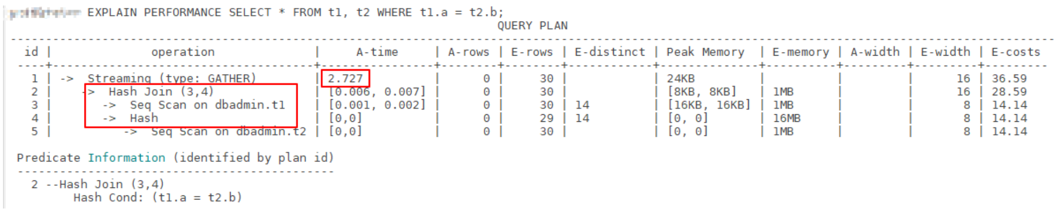
After optimization

Use the join condition in the query as the distribution key and run the following statement to changethe distribution key of **t2** as **b**:

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

After the distribution column of table **t2** is changed to column **b**, the execution plan does not contain **Streaming(type: REDISTRIBUTE)**. This reduces the amount of communication data between DNs and reduces the execution time from 8.7 ms to 2.7 ms, improving query performance, as shown in **Figure 5-2**.

Figure 5-2 Selecting an appropriate distribution column (2)



```
[redacted] 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)
```

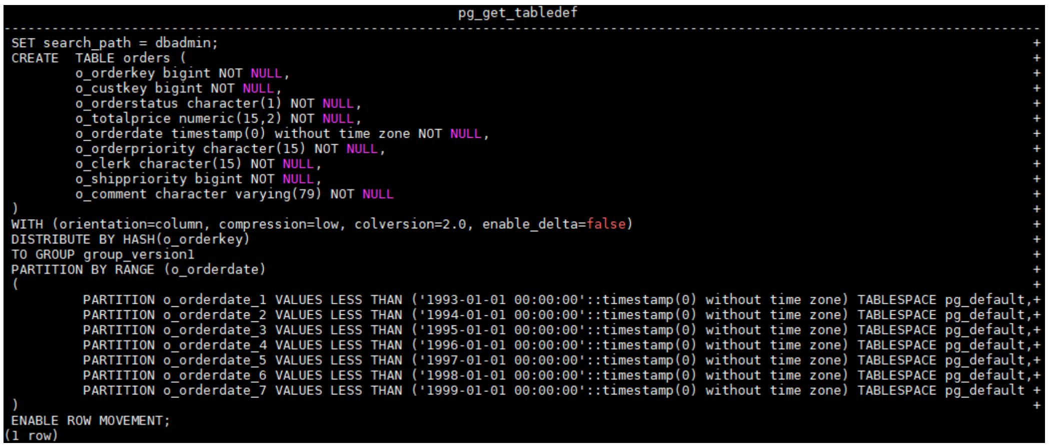
5.2 Case: Creating an Appropriate Index

Creating a proper index can accelerate the retrieval of data rows in a table. Indexes occupy disk space and reduce the speed of adding, deleting, and updating rows. If data needs to be updated very frequently or disk space is limited, you need to limit the number of indexes. Create indexes for large tables. Because the more data in the table, the more effective the index is. You are advised to create indexes on:

- Columns that need to be queried frequently
- Joined columns. For a query on joined columns, you are advised to create a composite index on the joined columns. For example, if the join condition is **select * from t1 join t2 on t1.a=t2.a and t1.b=t2.b**. You can create a composite index on the **a** and **b** columns of table **t1**.
- Columns having filter criteria (especially scope criteria) of a **where** clause
- Columns that appear after **order by**, **group by**, and **distinct**

Before optimization

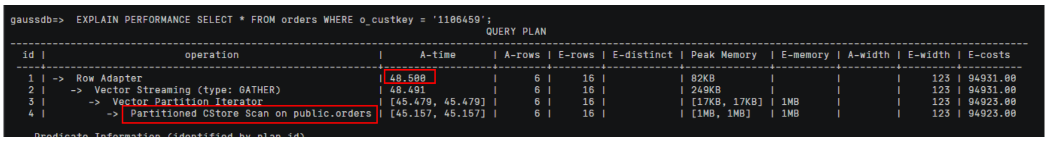
The column-store partitioned table **orders** is defined as follows:



```
                                                          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)
```

Run the SQL statement to query the execution plan when no index is created. It is found that the execution time is 48 milliseconds.

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



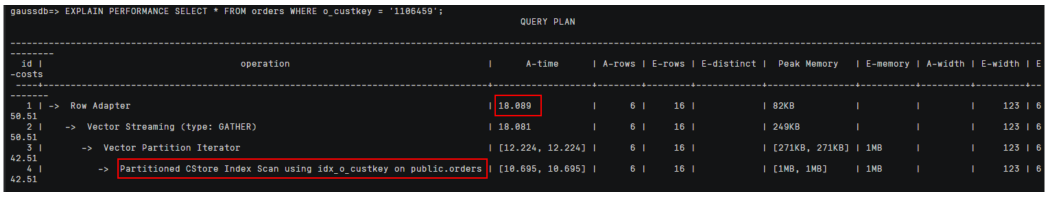
```
gaussdb=>  EXPLAIN PERFORMANCE SELECT * FROM orders WHERE o_custkey = '1106459';
                                                                     QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 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

    Predicate Information (identified by plan_id)
```

After optimization

The filtering condition column of the **where** clause is **o_custkey**. Add an index to the **o_custkey** column.

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

Run the SQL statement to query the execution plan after the index is created. It is found that the execution time is 18 milliseconds.



```
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 Case: Adding NOT NULL for JOIN Columns

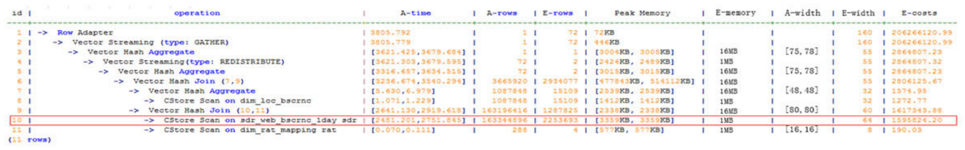
If there are many **NULL** values in the **JOIN** columns, you can add the filter criterion **IS NOT NULL** to filter data in advance to improve the **JOIN** efficiency.

Before optimization

```
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 ) ) ;
```

Figure 5-3 shows the execution plan.

Figure 5-3 Adding NOT NULL for JOIN columns (1)



```
 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)
```

After optimization

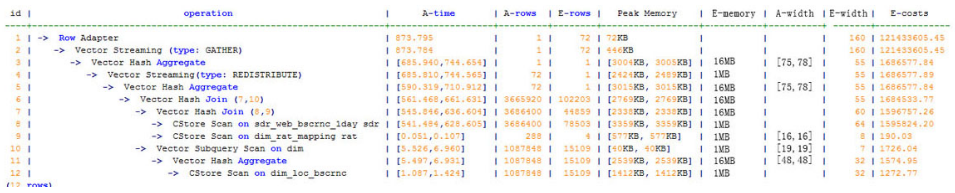
1. As shown in **Figure 5-3**, the sequential scan phase is time consuming.
2. The JOIN performance is poor because a large number of null values exist in the JOIN column **BSCRNC_ID** of the PS.SDR_WEB_BSCRNC_1DAY table.

   Therefore, you are advised to manually add **NOT NULL** for **JOIN** columns in the statement, as shown below:

```
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;
```

   Figure 5-4 shows the execution plan.

Figure 5-4 Adding NOT NULL for JOIN columns (2)



```
 id |                        operation                        |       A-time        |  A-rows  | E-rows  |     Peak Memory      | E-memory | A-width  | E-width |   E-costs
----+---------------------------------------------------------+---------------------+----------+---------+----------------------+----------+----------+---------+--------------
  1 | ->  Row Adapter                                         | 873.795             |        1 |      72 | 72KB                 |          |          |     160 | 121433605.45
  2 |   ->  Vector Streaming (type: GATHER)                   | 873.784             |        1 |      72 | 446KB                |          |          |     160 | 121433605.45
  3 |     ->  Vector Hash Aggregate                           | [685.940,744.654]   |        1 |       1 | [3004KB, 3005KB]     | 16MB     | [75,78]  |      55 | 1686577.84
  4 |       ->  Vector Streaming(type: REDISTRIBUTE)          | [685.810,744.565]   |       72 |       1 | [2424KB, 2489KB]     | 1MB      |          |      55 | 1686577.89
  5 |         ->  Vector Hash Aggregate                       | [590.319,710.912]   |       72 |       1 | [3015KB, 3015KB]     | 16MB     | [75,78]  |      55 | 1686577.84
  6 |           ->  Vector Hash Join (7,10)                   | [561.468,661.631]   |  3665920 |  102203 | [2769KB, 2769KB]     | 16MB     |          |      55 | 1684533.77
  7 |             ->  Vector Hash Join (8,9)                  | [545.846,636.604]   |  3686400 |   44859 | [2338KB, 2338KB]     | 16MB     |          |      60 | 1596757.26
  8 |               ->  CStore Scan on sdr_web_bscrnc_1day sdr | [541.484,628.605]  |  3686400 |   78503 | [3359KB, 3359KB]     | 1MB      |          |      64 | 1595824.20
  9 |               ->  CStore Scan on dim_rat_mapping rat    | [0.051,0.107]       |      288 |       4 | [577KB, 577KB]       | 1MB      | [16,16]  |       8 | 190.03
 10 |             ->  Vector Subquery Scan on dim             | [5.526,6.960]       |  1087848 |   15109 | [40KB, 40KB]         | 1MB      | [19,19]  |       7 | 1726.04
 11 |               ->  Vector Hash Aggregate                 | [5.497,6.931]       |  1087848 |   15109 | [2539KB, 2539KB]     | 16MB     | [48,48]  |      32 | 1574.95
 12 |                 ->  CStore Scan on dim_loc_bscrnc       | [1.087,1.424]       |  1087848 |   15109 | [1412KB, 1412KB]     | 1MB      |          |      32 | 1272.77
(12 rows)
```

5.4 Case: Pushing Down Sort Operations to DNs

In an execution plan, more than 95% of the execution time is spent on **window agg** performed on the CN. In this case, **sum** is performed for the two columns separately, and then another **sum** is performed for the separate sum results of the two columns. After this, trunc and sorting are performed in sequence. You can try to rewrite the statement into a subquery to push down the sorting operations.

Before optimization

The table structure is as follows:

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

The query statements are as follows:

```
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;
```

The execution plan is as follows:

```
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)
```

As we can see, both **window agg** and **sort** are performed on the CN, which is time consuming.

After optimization

Modify the statement to a subquery statement, as shown below:

```
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);
```

Perform **sum** on the **trunc** results of the two columns, take it as a subquery, and then perform **window agg** for the subquery to push down the sorting operation to DNs, as shown below:

```
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)
```

The optimized SQL statement greatly improves the performance by reducing the execution time from 120s to 7s.

5.5 Case: Configuring cost_param for Better Query Performance

The cost_param parameter is used to control use of different estimation methods in specific customer scenarios, allowing estimated values to be close to onsite values. This parameter can control various methods simultaneously by performing AND (&) operations on the bit for each method. A method is selected if its value is not **0**.

Scenario 1: Before Optimization

If **bit0** of **cost_param** is set to **1**, an improved mechanism is used for estimating the selection rate of non-equi-joins. This method is more accurate for estimating the selection rate of joins between two identical tables. The following example describes the optimization scenario when **bit0** of **cost_param** is set to **1**. In V300R002C00 and later, **cost_param & 1=0** is not used. That is, an optimized formula is selected for calculation.

NOTE

The selection rate indicates the percentage for which the number of rows meeting the join conditions account of the **JOIN** results when the **JOIN** relationship is established between two tables.

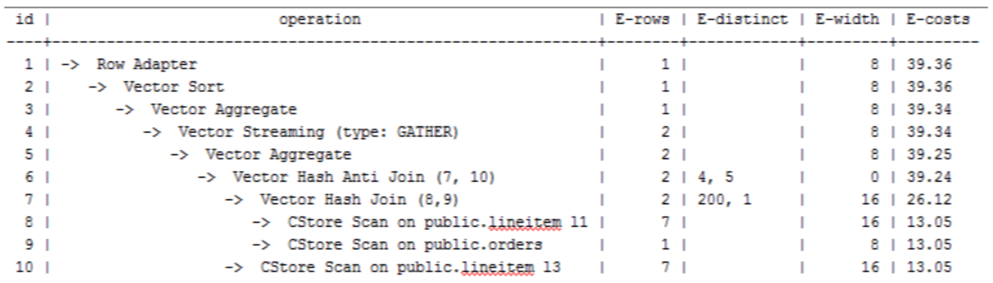
The table structure is as follows:

```
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);
```

The query statements are as follows:

```
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;
```

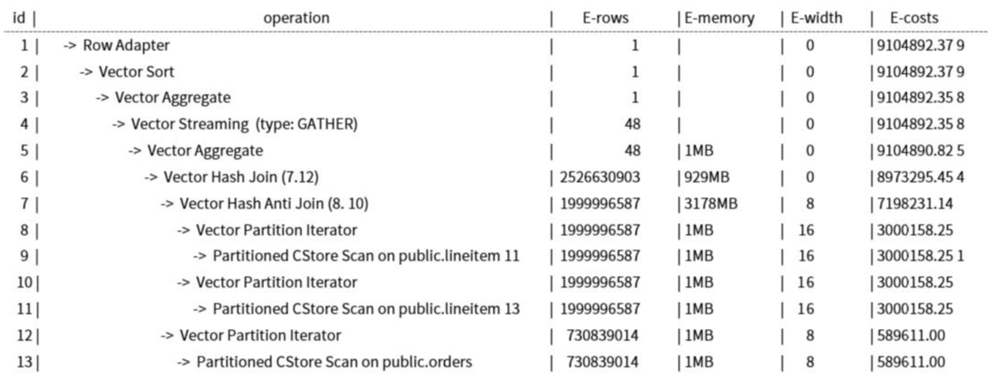
The following figure shows the execution plan. (When **verbose** is used, **distinct** is added for column selection which is controlled by **cost off/on**. The hash join rows show the estimated number of distinct values and the other rows do not.)



```
 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 l1 |      7 |            |      16 | 13.05
  9 |                      ->  CStore Scan on public.orders      |      1 |            |       8 | 13.05
 10 |                   ->  CStore Scan on public.lineitem l3    |      7 |            |      16 | 13.05
```

Scenario 1: After Optimization

These queries are from Anti Join connected in the **lineitem** table. When **cost_param & bit0** is **0**, the estimated number of Anti Join rows greatly differs from that of the actual number of rows, compromising the query performance. You can estimate the number of Anti Join rows more accurately by setting **cost_param & bit0** to **1** to improve the query performance. The optimized execution plan is as follows:



```
 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
```

Scenario 2: Before Optimization

If **bit1** is set to **1** (**set cost_param=2**), the selection rate is estimated based on multiple filter criteria. The lowest selection rate among all filter criteria, but not the product of the selection rates for two tables under a specific filter criterion, is used as the total selection rate. This method is more accurate when a close correlation exists between the columns to be filtered. The following example describes the optimization scenario when **bit1** of **cost_param** is set to **1**.

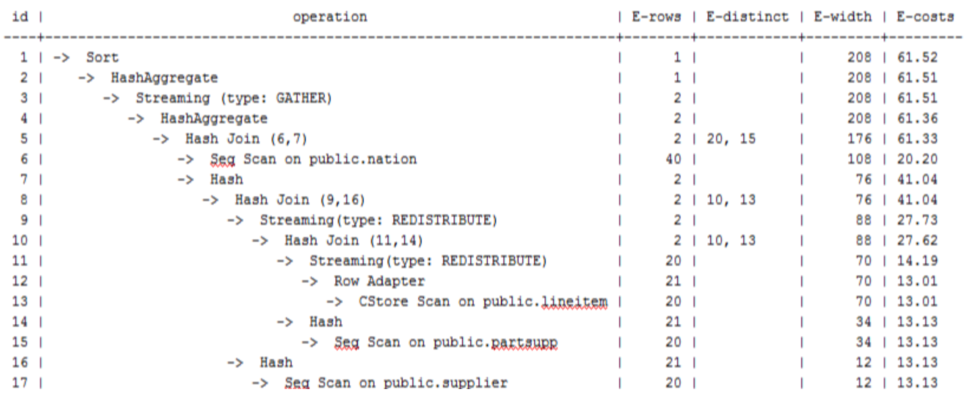
The table structure is as follows:

```
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);
```

The query statements are as follows:

```
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;
```

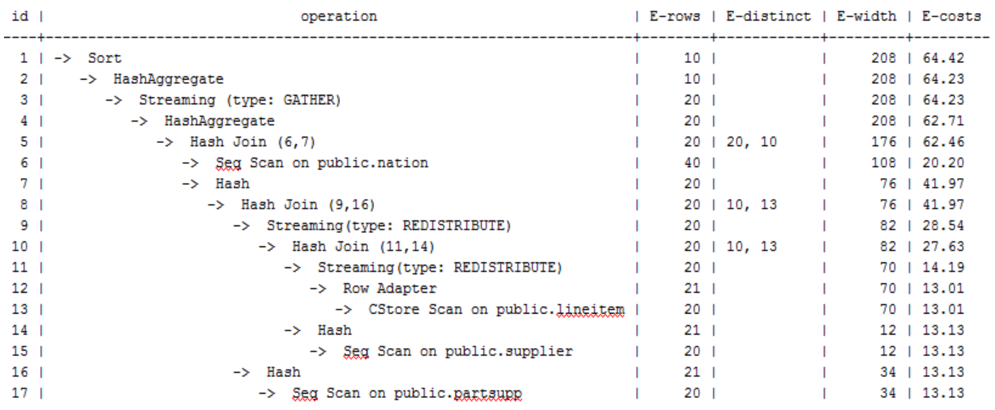
When **bit1** of **cost_param** is **0**, the execution plan is shown as follows:



```
 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
```

Scenario 2: After Optimization

In the preceding queries, the hash join criteria of the supplier, lineitem, and partsupp tables are setting **lineitem.l_suppkey** to **supplier.s_suppkey** and **lineitem.l_partkey** to **partsupp.ps_partkey**. Two filter criteria exist in the hash join conditions. **lineitem.l_suppkey** in the first filter criteria and **lineitem.l_partkey** in the second filter criteria are two columns with strong relationship of the lineitem table. In this situation, when you estimate the rate of the hash join conditions, if **cost_param & bit1** is **0**, the selection rate is estimated based on multiple filter criteria. The lowest selection rate among all filter criteria, but not the product of the selection rates for two tables under a specific filter criterion, is used as the total selection rate. This method is more accurate when a close correlation exists between the columns to be filtered. The plan after optimization is shown as follows:



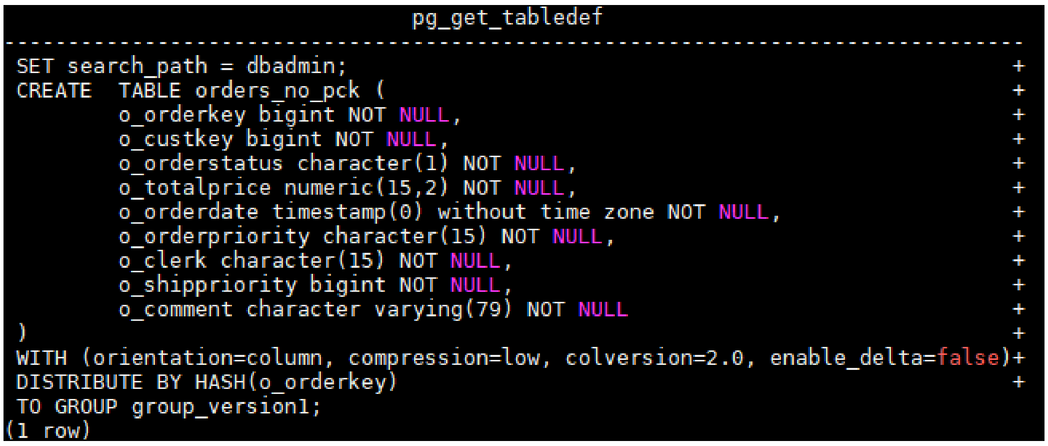
```
 id |                          operation                                  | E-rows | E-distinct | E-width | E-costs
----+---------------------------------------------------------------------+--------+------------+---------+---------
  1 | ->  Sort                                                            |     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 Case: Adjusting the Partial Clustering Key

Partial Cluster Key (PCK) is an index technology that uses min/max indexes to quickly scan base tables in column storage. Partial cluster key can specify multiple columns, but you are advised to specify no more than two columns. It can be used to accelerated queries on large column-store tables.

Before Optimization

Create a column-store table **orders_no_pck** without partial clustering (PCK). The table is defined as follows:

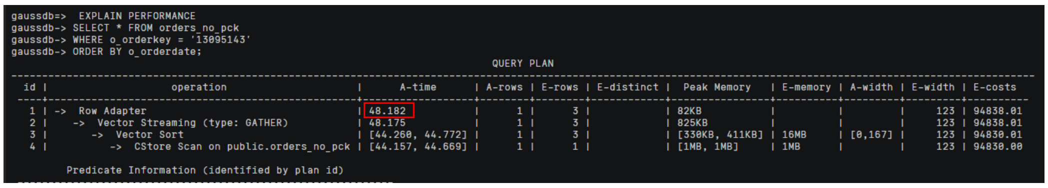


```
                                   pg_get_tabledef
--------------------------------------------------------------------------------
 SET search_path = dbadmin;                                                    +
 CREATE  TABLE orders_no_pck (                                                 +
         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)
```

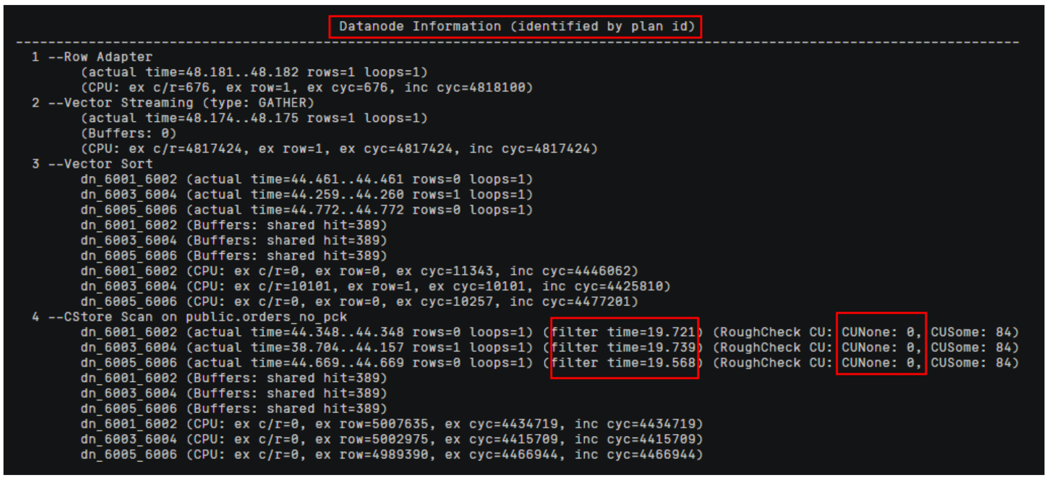
Run the following SQL statement to query the execution plan of a point query:

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

As shown in the following figure, the execution time is 48 ms. Check **Datanode Information**. It is found that the filter time is 19 ms and the CUNone ratio is 0.



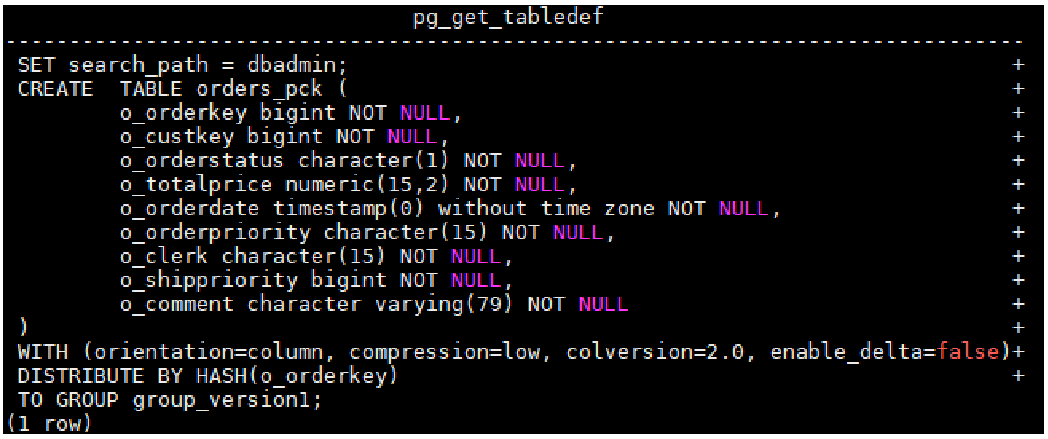
```
gaussdb=>  EXPLAIN PERFORMANCE
gaussdb-> SELECT * FROM orders_no_pck
gaussdb-> WHERE o_orderkey = '13095143'
gaussdb-> ORDER BY o_orderdate;
                                                                     QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------------------------
 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)
```

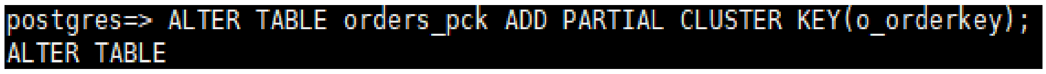
After Optimization

The created column-store table **orders_pck** is defined as follows:



```
                              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)
```

Use **ALTER TABLE** to set the **o_orderkey** field to **PCK**:

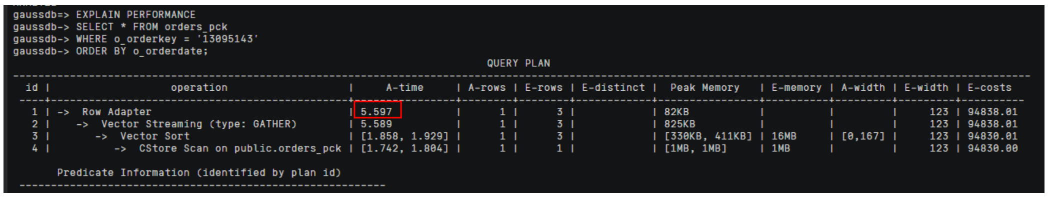


```
postgres=> ALTER TABLE orders_pck ADD PARTIAL CLUSTER KEY(o_orderkey);
ALTER TABLE
```

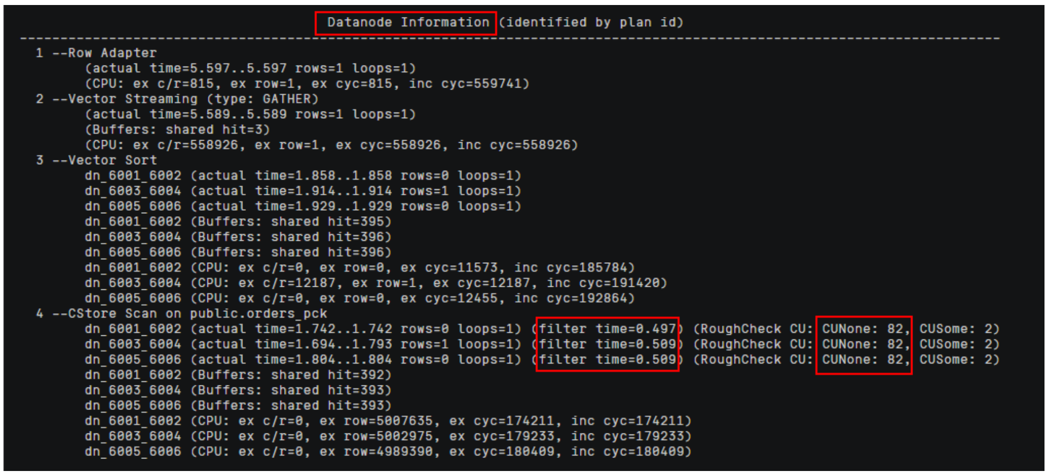
Run the following SQL statement to query the execution plan of the same point query SQL statement again:

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

As shown in the following figure, the execution time is 5 ms. Check **Datanode Information**. It is found that the filter time is 0.5 ms and the CUNone ratio is 82. The higher the CUNone ratio, the higher performance that the PCK will bring.



```
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)
 -----------------------------------------------------------
```



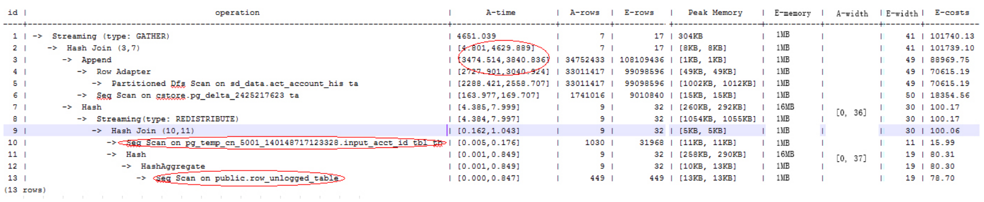
```
                               Datanode Information (identified by plan id)
------------------------------------------------------------------------------------------------------
  1 --Row Adapter
        (actual time=5.597..5.597 rows=1 loops=1)
        (CPU: ex c/r=815, ex row=1, ex cyc=815, inc 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 Case: Adjusting the Table Storage Mode in a Medium Table

In GaussDB(DWS), row-store tables use the row execution engine, and column-store tables use the column execution engine. If both row-store table and column-store tables exist in a SQL statement, the system will automatically select the row execution engine. The performance of a column execution engine (except for the indexscan related operators) is much better than that of a row execution engine. Therefore, a column-store table is recommended. This is important for some medium result set dumping tables, and you need to select a proper table storage type.

Before Optimization

During the test at a site, if the following execution plan is performed, the customer expects that the performance can be improved and the result can be returned within 3s.

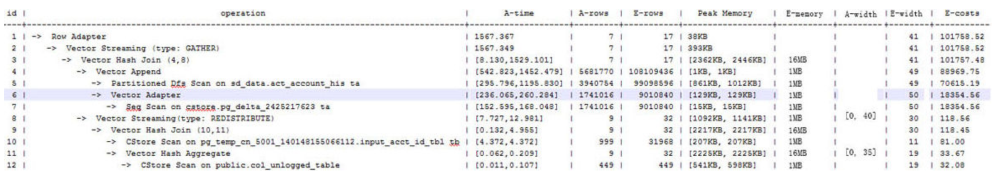


```
 id |                              operation                                      |       A-time        |  A-rows  |  E-rows   |   Peak Memory      | E-memory |  A-width |E-width|  E-costs
----+-----------------------------------------------------------------------------+---------------------+----------+-----------+--------------------+----------+----------+-------+-----------
  1 | ->  Streaming (type: GATHER)                                                | 4651.039            |        7 |        17 | 304KB              | 1MB      |          |    41 | 101740.13
  2 |    ->  Hash Join (3,7)                                                      | [4.801,4629.889]    |        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          | [2288.421,2558.707] | 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 | 15.99
 11 |                ->  Hash                                                     | [0.001,0.849]       |        9 |        32 | [258KB, 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)
```

After Optimization

It is found that the row engine is used after analysis, because both the temporary plan table input_acct_id_tbl and the medium result dumping table row_unlogged_table use a row-store table.

After the two tables are changed into column-store tables, the system performance is improved and the result is returned by 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 | 38KB               |          |         |     41 | 101758.52
 2 |   ->  Vector Streaming (type: GATHER)                                         | 1567.349               |       7 |        17 | 393KB              |          |         |     41 | 101758.52
 3 |     ->  Vector Hash Join (4,8)                                                | [8.130,1529.101]       |       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)                                          | [0.132,4.955]          |       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
```

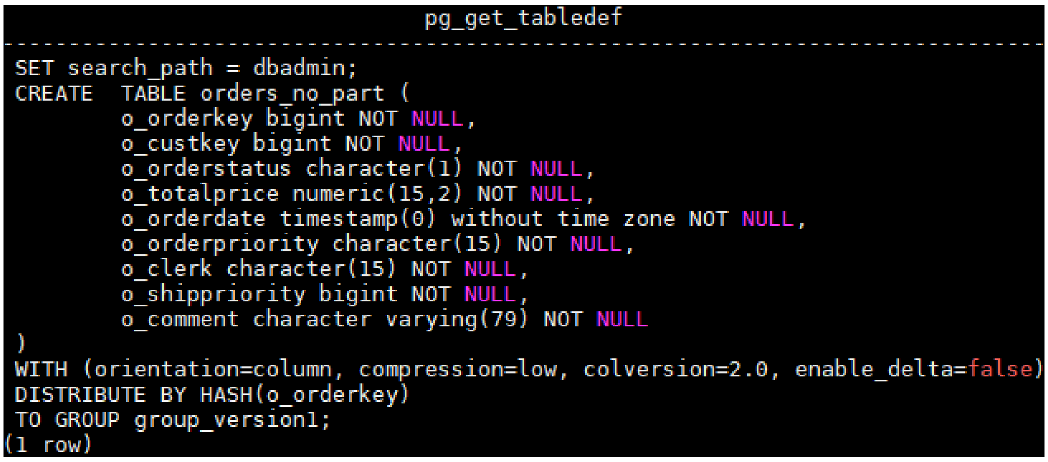
5.8 Case: Reconstructing Partition Tables

Partitioning refers to splitting what is logically one large table into smaller physical pieces based on specific schemes. The table based on the logic is called a partitioned table, and a physical piece is called a partition. Generally, partitioning is applied to tables that have obvious ranges. Partitions on such tables allow scanning on a small part of data, improving the query performance.

During query, partition pruning is used to minimize bottom-layer data scanning to narrow down the overall scope of scanning in a table. Partition pruning means that the optimizer can automatically extract partitions to be scanned based on the partition key specified in the **FROM** and **WHERE** statements. This avoids full table scanning, reduces the number of data blocks to be scanned, and improves performance.

Before Optimization

Create a non-partition table **orders_no_part**. The table definition is as follows:

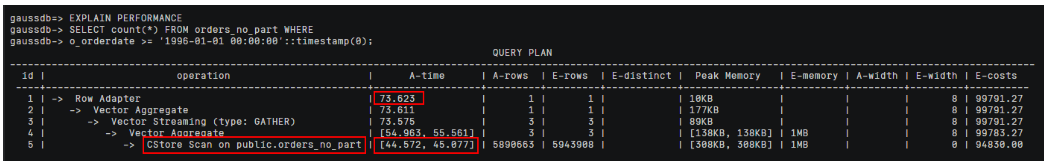


```
                                pg_get_tabledef
--------------------------------------------------------------------------------
 SET search_path = dbadmin;
 CREATE  TABLE orders_no_part (
         o_orderkey bigint NOT NULL,
         o_custkey bigint NOT NULL,
         o_orderstatus character(1) 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)
```

Run the following SQL statement to query the execution plan of the non-partition table:

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

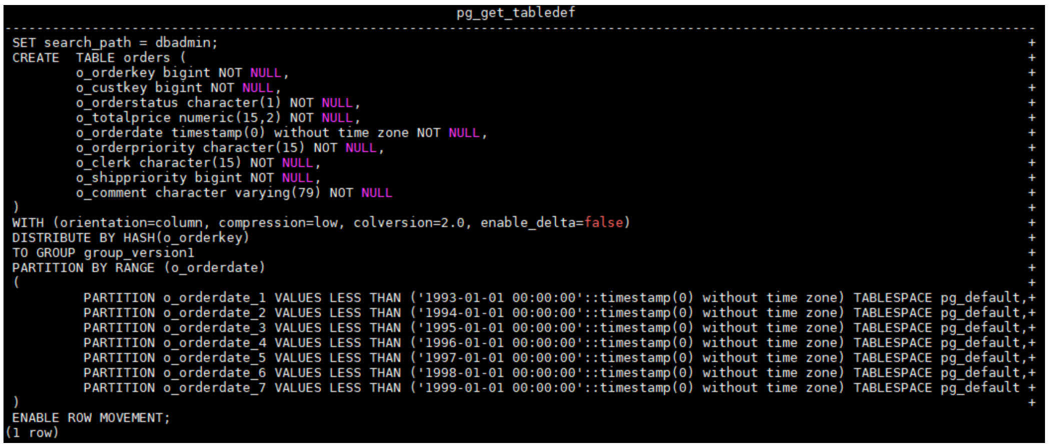
As shown in the following figure, the execution time is 73 milliseconds, and the full table scanning time is 44 to 45 milliseconds.



```
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
```

After Optimization

Create a partitioned table **orders**. The table is defined as follows:

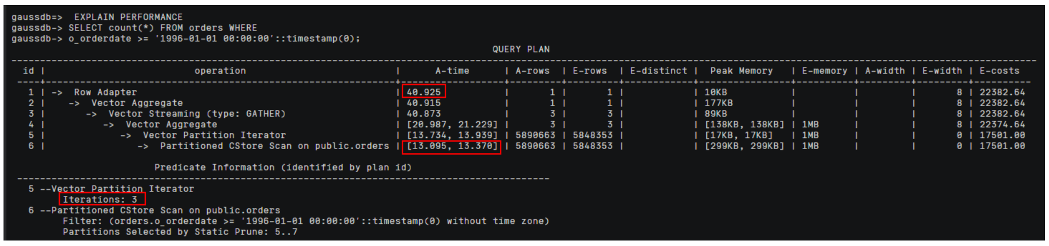


```
                                                      pg_get_tabledef
------------------------------------------------------------------------------------------------------------------------------------
 SET search_path = dbadmin;                                                                                                         +
 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)
```

Run the SQL statement again to query the execution plan of the partitioned table. The execution time is 40 ms, in which the table scanning time is only 13 ms. The smaller the value of **Iterations**, the better the partition pruning effect.

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

As shown in the following figure, the execution time is 40 milliseconds, and the table scanning time is only 13 milliseconds. A smaller **Iterations** value indicates a better partition pruning effect.



```
gaussdb=>  EXPLAIN PERFORMANCE
gaussdb-> SELECT count(*) FROM orders 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                                         | 40.925             |       1 |       1 |            | 10KB             |          |         |       8 | 22382.64
  2 |    ->  Vector Aggregate                                 | 40.915             |       1 |       1 |            | 177KB            |          |         |       8 | 22382.64
  3 |       ->  Vector Streaming (type: GATHER)               | 40.873             |       3 |       3 |            | 89KB             |          |         |       8 | 22382.64
  4 |          ->  Vector Aggregate                           | [20.987, 21.229]   |       3 |       3 |            | [138KB, 138KB]   | 1MB      |         |       8 | 22374.64
  5 |             ->  Vector Partition Iterator               | [13.734, 13.939]   | 5890663 | 5848353 |            | [17KB, 17KB]     | 1MB      |         |       0 | 17501.00
  6 |                ->  Partitioned CStore Scan on public.orders | [13.095, 13.370] | 5890663 | 5848353 |            | [299KB, 299KB]   | 1MB      |         |       0 | 17501.00

                         Predicate Information (identified by plan id)
 ------------------------------------------------------------------------------------------------------
   5 --Vector Partition Iterator
         Iterations: 3
   6 --Partitioned CStore Scan on public.orders
         Filter: (orders.o_orderdate >= '1996-01-01 00:00:00'::timestamp(0) without time zone)
         Partitions Selected by Static Prune: 5..7
```

5.9 Case: Adjusting the GUC Parameter best_agg_plan

Symptom

The t1 table is defined as follows:

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

Assume that the distribution column of the result set provided by the agg lower-layer operator is setA, and the group by column of the agg operation is setB, the agg operations can be performed in two scenarios in the stream framework.

Scenario 1: setA is a subset of setB.

In this scenario, the aggregation result of the lower-layer result set is the correct result, which can be directly used by the upper-layer operator. For details, see the following figure:

```
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)
```

Scenario 2: setA is not a subset of setB.

In this scenario, the Stream execution framework is classified into the following three plans:

hashagg+gather(redistribute)+hashagg

redistribute+hashagg(+gather)

hashagg+redistribute+hashagg(+gather)

GaussDB(DWS) provides the guc parameter **best_agg_plan** to intervene the execution plan, and forces the plan to generate the corresponding execution plan. This parameter can be set to **0**, **1**, **2**, and **3**.

- When the value is set to **1**, the first plan is forcibly generated.
- When the value is set to **2** and if the **group by** column can be redistributed, the second plan is forcibly generated. Otherwise, the first plan is generated.
- When the value is set to **3** and if the **group by** column can be redistributed, the third plan is generated. Otherwise, the first plan is generated.
- When the value is set to **0**, the query optimizer chooses the most optimal plan by the three preceding plans' evaluation cost.

Possible impacts are as follows:

```
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)
```

Summary

Generally, the optimizer chooses an optimal execution plan, but the cost estimation, especially that of the intermediate result set, has large deviations, which may result in large deviations in agg calculation. In this case, you need to use best_agg_plan to adjust the agg calculation model.

When the aggregation convergence ratio is very small, that is, the number of result sets does not become small obviously after the agg operation (5 times is a critical point), you can select the redistribute+hashagg or hashagg+redistribute +hashagg execution mode.

5.10 Case: Rewriting SQL Statements and Eliminating Prune Interference

A filter criterion that contains the expression of partition key cannot be used for pruning. As a result, the query statement scans almost all data in the partitioned table.

Before Optimization

t_ddw_f10_op_cust_asset_mon indicates the partitioned table. **year_mth** indicates the partition key. This field is an integer consisting of the **year** and **mth** values.

The following figure shows the tested SQL statements.

```
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 );
```

The test result shows that the table scan of the SQL statement takes 10 seconds. The execution plan of the SQL statement is as follows.

```
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
```

After Optimization

After analyzing the execution plan of the statement and checking the SQL self-diagnosis information in the execution plan, the following diagnosis information is found:

```
                                 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
```

The filter criterion contains the expression **(year_mth + 1) > 202008**. A filter criterion that contains the expression of partition key cannot be used for pruning. As a result, the query statement scans almost all data in the partitioned table.

Compared with the original SQL statement, the expression **(year_mth + 1) > 202008** is derived from the expression **b1.year_mth + 1 > substr('20200822',1 ,6)**. Based on the diagnosis information, the SQL statement is modified as follows.

```
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;
```

After the modification, the SQL statement execution information is as follows. The alarm indicating that the pruning is not performed is cleared. After the pruning, the score of the partition to be scanned is 1, and the execution time is shortened from 10 seconds to 3 seconds.

```
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
```

5.11 Case: Rewriting SQL Statements and Deleting in-clause

Before Optimization

in-clause/any-clause is a common SQL statement constraint. Sometimes, the clause following **in** or **any** is a constant. For example:

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

or

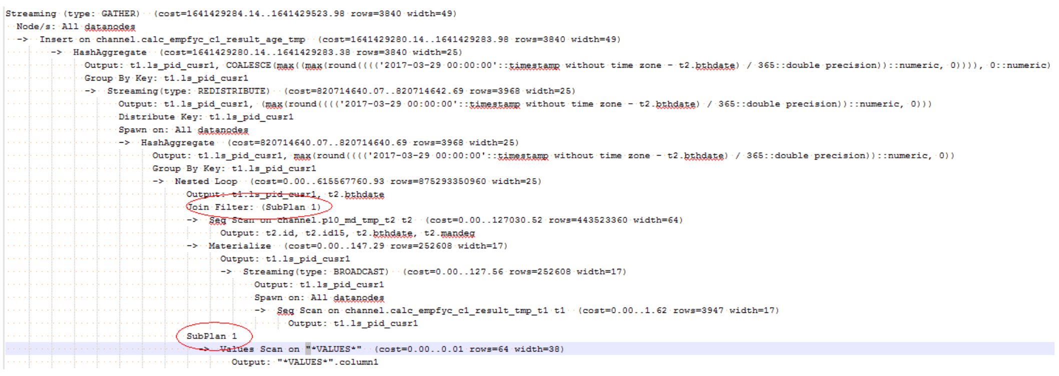
```
select
count(1)
from calc_empfyc_c1_result_tmp_t1
where ls_pid_cusr1 in any('20120405', '20130405');
```

Some special usages are as follows:

```
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;
```

Where **id** and **id15** are columns of p10_md_tmp_t2. ls_pid_cusr1 = any(values(id),(id15)) equals t1. ls_pid_cusr1 = id or t1. ls_pid_cusr1 = id15.

Therefore, join-condition is essentially an inequality, and nestloop must be used for this join operation. The execution plan is as follows:



```
Streaming (type: GATHER)  (cost=1641429284.14..1641429523.98 rows=3840 width=49)
  Node/s: All datanodes
  ->  Insert on channel.calc_empfyc_c1_result_age_tmp  (cost=1641429280.14..1641429283.98 rows=3840 width=49)
        ->  HashAggregate  (cost=1641429280.14..1641429283.38 rows=3840 width=25)
              Output: t1.ls_pid_cusr1, COALESCE(max((max(round(((('2017-03-29 00:00:00'::timestamp without time zone - t2.bthdate) / 365::double precision))::numeric, 0)))), 0::numeric)
              Group By Key: t1.ls_pid_cusr1
              ->  Streaming(type: REDISTRIBUTE)  (cost=820714640.07..820714642.69 rows=3968 width=25)
                    Output: t1.ls_pid_cusr1, (max(round(((('2017-03-29 00:00:00'::timestamp without time zone - t2.bthdate) / 365::double precision))::numeric, 0)))
                    Distribute Key: t1.ls_pid_cusr1
                    Spawn on: All datanodes
                    ->  HashAggregate  (cost=820714640.07..820714640.69 rows=3968 width=25)
                          Output: t1.ls_pid_cusr1, max(round(((('2017-03-29 00:00:00'::timestamp without time zone - t2.bthdate) / 365::double precision))::numeric, 0))
                          Group By Key: t1.ls_pid_cusr1
                          ->  Nested Loop  (cost=0.00..615567760.93 rows=875293350960 width=25)
                                Output: t1.ls_pid_cusr1, t2.bthdate
                                Join Filter: (SubPlan 1)
                                ->  Seq Scan on channel.p10_md_tmp_t2 t2  (cost=0.00..127030.52 rows=443523360 width=64)
                                      Output: t2.id, t2.id15, t2.bthdate, t2.mandeg
                                ->  Materialize  (cost=0.00..147.29 rows=252608 width=17)
                                      Output: t1.ls_pid_cusr1
                                      ->  Streaming(type: BROADCAST)  (cost=0.00..127.56 rows=252608 width=17)
                                            Output: t1.ls_pid_cusr1
                                            Spawn on: All datanodes
                                            ->  Seq Scan on channel.calc_empfyc_c1_result_tmp_t1 t1  (cost=0.00..1.62 rows=3947 width=17)
                                                  Output: t1.ls_pid_cusr1
                                SubPlan 1
                                  ->  Values Scan on "*VALUES*"  (cost=0.00..0.01 rows=64 width=38)
                                        Output: "*VALUES*".column1
```

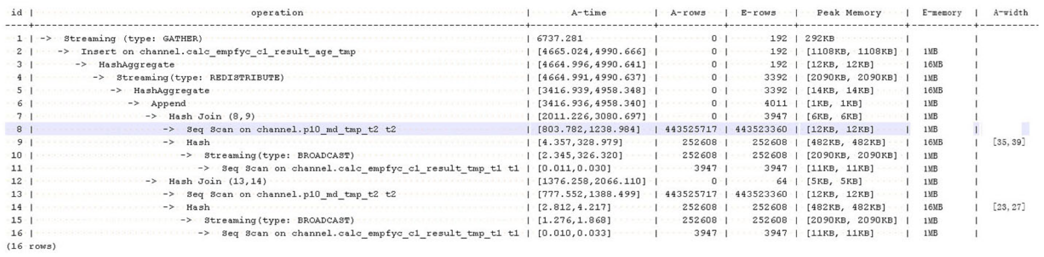
After Optimization

The test result shows that both result sets are too large. As a result, nestloop is time-consuming with more than one hour to return results. Therefore, the key to performance optimization is to eliminate nestloop, using more efficient hashjoin. From the perspective of semantic equivalence, the SQL statements can be written as follows:

```
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;
```

Note: Use **UNION ALL** instead of **UNION** if possible. **UNION** eliminates duplicate rows while merging two result sets but **UNION ALL** merges the two result sets without deduplication. Therefore, replace **UNION** with **UNION ALL** if you are sure that the two result sets do not contain duplicate rows based on the service logic.

The optimized SQL queries consist of two equivalent join subqueries, and each subquery can be used for hashjoin in this scenario. The optimized execution plan is as follows:



```
 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)
```

Before the optimization, no result is returned for more than 1 hour. After the optimization, the result is returned within 7s.

5.12 Case: Setting Partial Cluster Keys

You can add **PARTIAL CLUSTER KEY**(*column_name*[,...]) to the definition of a column-store table to set one or more columns of this table as partial cluster keys. In this way, each 70 CUs (4.2 million rows) will be sorted based on the cluster keys by default during data import and the value range is narrowed down for each of the new 70 CUs. If the **where** condition in the query statement contains these columns, the filtering performance will be improved.

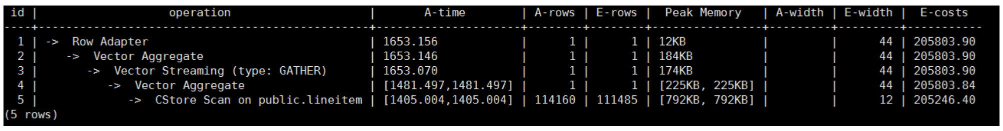
Before Optimization

The partial cluster key is not used. The table is defined as follows:

```
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;
```

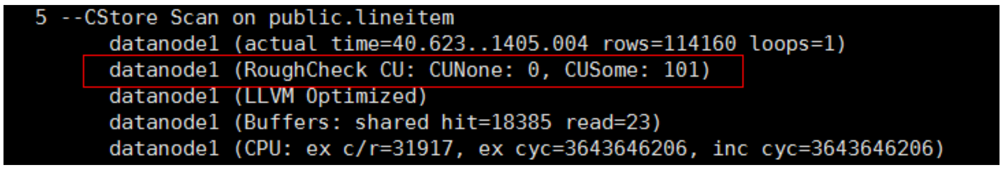
After the data is imported, perform the query and check the execution time.

Figure 5-5 Partial cluster keys not used



```
 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)
```

Figure 5-6 CU loading without partial cluster keys



```
  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)
```

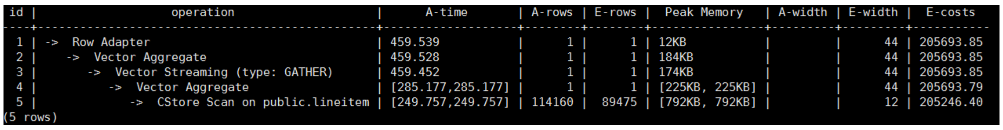
After Optimization

In the **where** condition, both the **l_shipdate** and **l_quantity** columns have a few distinct values, and their values can be used for min/max filtering. Therefore, modify the table definition as follows:

```
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);
```

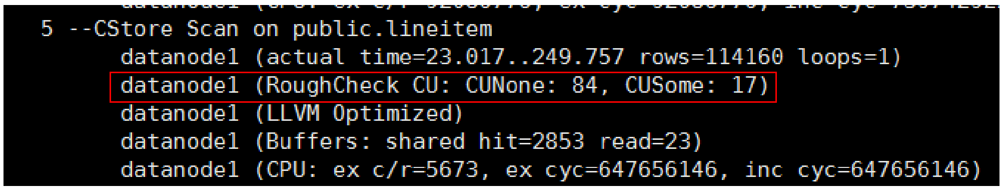
Import the data again, perform the query, and check the execution time.

Figure 5-7 Partial cluster keys used



```
 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)
```

Figure 5-8 CU loading with partial cluster keys



```
  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)
```

After partial cluster keys are used, the execution time of **5-- CStore Scan on public.lineitem** decreases by 1.2s because 84 CUs are filtered out.

Optimization

- Select partial cluster keys.
  - The following data types support cluster keys: character varying(n), varchar(n), character(n), char(n), text, nvarchar2, timestamp with time zone, timestamp without time zone, date, time without time zone, and time with time zone.
  - Smaller number of distinct values in a partial cluster key generates higher filtering performance.
  - Columns that can filter out larger amount of data is preferentially selected as partial cluster keys.
  - If multiple columns are selected as partial cluster keys, the columns are used in sequence to sort data. You are advised to select a maximum of three columns.
- Modify parameters to reduce the impact of partial cluster keys on the import performance.

  After partial cluster keys are used, data will be sorted when they are imported, affecting the import performance. If all the data can be sorted in the memory, the keys have little impact on import. If some data cannot be sorted in the memory and is written into a temporary file for sorting, the import performance will be greatly affected.

The memory used for sorting is specified by the **psort_work_mem** parameter. You can set it to a larger value so that the sorting has less impact on the import performance.

The volume of data to be sorted is specified by the **PARTIAL_CLUSTER_ROWS** parameter of the table. Decreasing the value of this parameter reduces the amount of data to be sorted at a time. **PARTIAL_CLUSTER_ROWS** is usually used along with the **MAX_BATCHROW** parameter. The value of **PARTIAL_CLUSTER_ROWS** must be an integer multiple of the **MAX_BATCHROW** value. **MAX_BATCHROW** specifies the maximum number of rows in a CU.

5.13 Case: Converting from NOT IN to NOT EXISTS

nestloop anti join must be used to implement **NOT IN**, while you can use **Hash anti join** to implement **NOT EXISTS**. If no **NULL** value exists in the **JOIN** column, **NOT IN** is equivalent to **NOT EXISTS**. Therefore, if you are sure that no **NULL** value exists, you can convert **NOT IN** to **NOT EXISTS** to generate **hash joins** and to improve the query performance.

Before Optimization

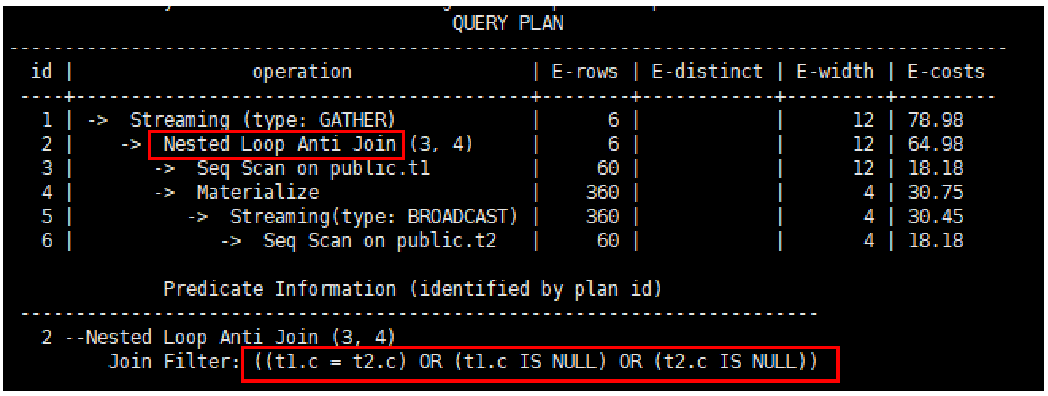
Create two base tables **t1** and **t2**.

```
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);
```

Run the following SQL statement to query the **NOT IN** execution plan:

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

The following figure shows the statement output.



```
                                  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))
```

According to the returned result, nest loops are used. As the OR operation result of NULL and any value is NULL,

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

the preceding condition expression is equivalent to:

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

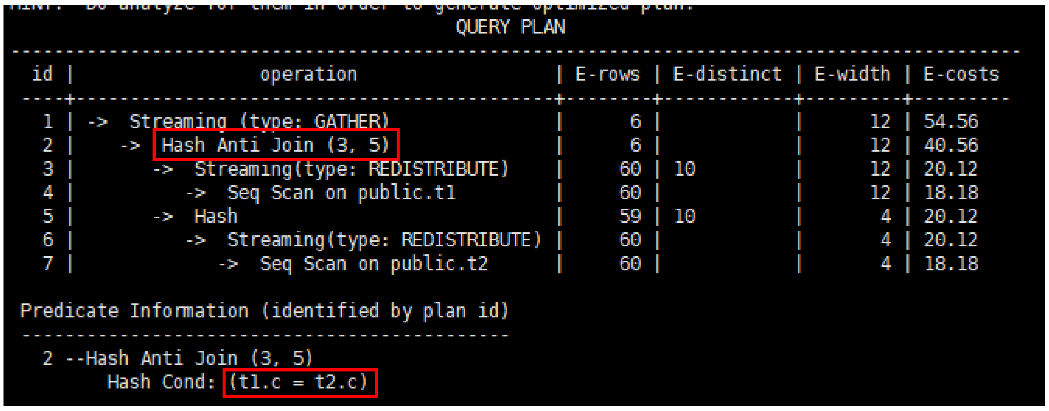
After Optimization

The query can be modified as follows:

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

Run the following statement to query the execution plan of **NOT EXISTS**:

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



```
HINT:  Do analyze for them in order to generate optimized plan.
                                           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 Execution Troubleshooting

6.1 Low Query Efficiency

A query task that used to take a few milliseconds to complete is now requiring several seconds, and that used to take several seconds is now requiring even half an hour. This section describes how to analyze and rectify such low efficiency issues.

Procedure

Perform the following procedure to locate the cause of this fault.

Step 1 Run the **analyze** command to analyze the database.

The **analyze** command updates data statistics information, such as data sizes and attributes in all tables. This is a lightweight command and can be executed frequently. If the query efficiency is improved or restored after the command execution, the autovacuum process does not function well and requires further analysis.

Step 2 Check whether the query statement returns unnecessary information.

For example, if we only need the first 10 records in a table but the query statement searches all records in the table, the query efficiency is fine for a table containing only 50 records but very low for a table containing 50,000 records.

If an application requires only a part of data information but the query statement returns all information, add a LIMIT clause to the query statement to restrict the number of returned records. In this way, the database optimizer can optimize space and improve query efficiency.

Step 3 Check whether the query statement still has a low response even when it is solely executed.

Run the query statement when there are no or only a few other query requests in the database, and observe the query efficiency. If the efficiency is high, the previous issue is possibly caused by a heavily loaded host in the database system or an inefficient execution plan.

Step 4 Check the same query statement repeatedly to check the query efficiency.

One major cause that will reduce query efficiency is that the required information is not cached in the memory or is replaced by other query requests because of insufficient memory resources.

Run the same query statement repeatedly. If the query efficiency increases gradually, the previous issue might be caused by this reason.

----End

6.2 Different Data Is Displayed for the Same Table Queried By Multiple Users

Problem

Two users log in to the same database human_resource and run the **select count(*) from areas** statement separately to query the areas table, but obtain different results.

Possible Causes

Check whether the two users really query the same table. In a relational database, a table is identified by three elements: **database**, **schema**, and **table**. In this issue, **database** is **human_resource** and **table** is **areas**. Then, check **schema**. Log in as users **dbadmin** and **user01** separately. It is found that **search_path** is **public** for **dbadmin** and *$user* for **user01**. By default, a schema having the same name as user **dbadmin**, the cluster administrator, is not created. That is, all tables will be created in **public** if no schema is specified. However, when a common user, such as **user01**, is created, the same-name schema (**user01**) is created by default. That is, all tables are created in **user01** if the schema is not specified. In conclusion, both the two users are operating the table, causing that the same-name table is not really the same table.

Troubleshooting Method

Use *schema***.table** to determine a table for query.

6.3 An Error Occurs During the Integer Conversion

Problem

The following error is reported during the integer conversion:

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

Possible Causes

Some data types cannot be converted to the target data type.

Troubleshooting

Gradually narrow down the range of SQL statements to locate the fault.

6.4 Automatic Retry upon SQL Statement Execution Errors

With automatic retry (referred to as CN retry), GaussDB(DWS) retries an SQL statement when the execution of a statement fails. If an SQL statement sent from the **gsql** client, JDBC driver, or ODBC driver fails to be executed, the CN can automatically identify the error reported during execution and re-deliver the task to retry.

The restrictions of this function are as follows:

- Functionality restrictions:
  - CN retry increases execution success rate but does not guarantee success.
  - CN retry is enabled by default. In this case, the system records logs about temporary tables. If it is disabled, the system will not record the logs. Therefore, do not repeatedly enable and disable CN retry when temporary tables are used. Otherwise, data inconsistency may occur after a CN retry following a primary/standby switchover.
  - CN retry is enabled by default. In this case, the **unlogged** keyword is ignored in the statement for creating unlogged tables and thereby ordinary tables will be created by using this statement. If CN retry is disabled, the system records logs about unlogged tables. Therefore, do not repeatedly enable and disable CN retry when unlogged tables are used. Otherwise, data inconsistency may occur after a CN retry following a primary/standby switchover.
  - When GDS is used to export data, CN retry is supported. The existing mechanism checks for duplicate files and deletes duplicate files during data export. Therefore, you are advised not to repeatedly export data for the same foreign table unless you are sure that files with the same name in the data directory need to be deleted.
- Error type restrictions:

  Only the error types in **Table 6-1** are supported.
- Statement type restrictions:

  Support single-statement CN retry, stored procedures, functions, and anonymous blocks. Statements in transaction blocks are not supported.
- Statement restrictions of a stored procedure:
  - If an error occurs during the execution of a stored procedure containing **EXCEPTION** (including statement block execution and statement execution in EXCEPTION), the stored procedure can be retried. If an internal error occurs, the stored procedure will retry first, but if the error is captured by **EXCEPTION**, the stored procedure cannot be retried.
  - Packages that use global variables are not supported.
  - **DBMS_JO** is not supported.
  - **UTL_FILE** is not supported.

  - If the stored procedure has printed information (such as **dbms_output.put_line** or **raise info**), the printed information will be output repeatedly when retry occurs, and "Notice: Retry triggered, some message may be duplicated. " will be output before the repeated information.
- Cluster status restrictions:
  - Only DNs or GTMs are faulty.
  - The cluster can be recovered before the number of CN retries reaches the allowed maximum (controlled by **max_query_retry_times**). Otherwise, CN retry may fail.
  - CN retry is not supported during scale-out.
- Data import restrictions:
  - The **COPY FROM STDIN** statement is not supported.
  - The **gsql \copy from** metacommand is not supported.
  - **JDBC CopyManager copyIn** is not supported.

Table 6-1 lists the error types supported by CN retry and the corresponding error codes. You can use the GUC parameter **retry_ecode_list** to set the list of error types supported by CN retry. You are not advised to modify this parameter. To modify it, contact the technical support.

Table 6-1 Error types supported by CN retry

Error Type	Error Code	Remarks
CONNECTION_RESET_BY_PEER	YY001	TCP communication errors: Connection reset by peer (communication between the CN and DNs)
STREAM_CONNECTION_RESET_BY_PEER	YY002	TCP communication errors: Stream connection reset by peer (communication between DNs)
LOCK_WAIT_TIMEOUT	YY003	Lock wait timeout
CONNECTION_TIMED_OUT	YY004	TCP communication errors: Connection timed out
SET_QUERY_ERROR	YY005	Failed to deliver the **SET** command: Set query
OUT_OF_LOGICAL_MEMORY	YY006	Failed to apply for memory: Out of logical memory
SCTP_MEMORY_ALLOC	YY007	SCTP communication errors: Memory allocate error
SCTP_NO_DATA_IN_BUFFER	YY008	SCTP communication errors: SCTP no data in buffer

Error Type	Error Code	Remarks
SCTP_RELEASE_MEMORY_CLOSE	YY009	SCTP communication errors: Release memory close
SCTP_TCP_DISCONNECT	YY010	SCTP communication errors: TCP disconnect
SCTP_DISCONNECT	YY011	SCTP communication errors: SCTP disconnect
SCTP_REMOTE_CLOSE	YY012	SCTP communication errors: Stream closed by remote
SCTP_WAIT_POLL_UNKNOW	YY013	Waiting for an unknown poll: SCTP wait poll unknown
SNAPSHOT_INVALID	YY014	Snapshot invalid
ERRCODE_CONNECTION_RECEIVE_WRONG	YY015	Connection receive wrong
OUT_OF_MEMORY	53200	Out of memory
CONNECTION_FAILURE	08006	GTM errors: Connection failure
CONNECTION_EXCEPTION	08000	Failed to communicate with DNs due to connection errors: Connection exception
ADMIN_SHUTDOWN	57P01	System shutdown by administrators: Admin shutdown
STREAM_REMOTE_CLOSE_SOCKET	XX003	Remote socket disabled: Stream remote close socket
ERRCODE_STREAM_DUPLICATE_QUERY_ID	XX009	Duplicate query id
ERRCODE_STREAM_CONCURRENT_UPDATE	YY016	Stream concurrent update
ERRCODE_LLVM_BAD_ALLOC_ERROR	CG003	Memory allocation error: Allocate error
ERRCODE_LLVM_FATAL_ERROR	CG004	Fatal error
HashJoin temporary file reading error (ERRCODE_HASHJOIN_TEMP_FILE_ERROR).	F0011	File error

Error Type	Error Code	Remarks
Buffer file reading error (ERRCODE_BUFFER_FILE_ERROR)	F0012	File reading error.
Partition number error (ERRCODE_PARTITION_NUM_CHANGED).	45003	During scanning on a list partition table, it is found that the number of partitions is different from that in the optimization phase. This problem usually occurs when the queries and **ADD**/**DROP** partitions are concurrently executed. (This error is supported only by cluster 8.1.3 and later versions.)
Unmatched schema name (ERRCODE_UNMATCH_OBJECT_SCHEMA)	42P30	Unmatched schema name

To enable CN retry, set the following GUC parameters:

- Mandatory GUC parameters (required by both CNs and DNs)

  max_query_retry_times

  CAUTION

  If CN retry is enabled, temporary table data is logged. For data consistency, do not switch the enabled/disabled status for CN retry when the temporary tables are being used by sessions.

- Optional GUC parameters

  cn_send_buffer_size

  max_cn_temp_file_size

7 query_band Load Identification

Overview

GaussDB(DWS) implements load identification and intra-queue priority control based on query_band. It provides more flexible load identification methods and identifies load queues based on job types, application names, and script names. Users can flexibly configure query_band identification queues based on service scenarios. In addition, priority control of job delivery in the queue is implemented. In the future, priority control of resources in the queue will be gradually implemented.

Administrators can configure the queue associated with query_band and estimate the memory limit based on service scenarios and job types to implement more flexible load control and resource management and control. If query_band is not configured for the service or the user does not associate query_band with an action, the queue associated with the user and the priority in the queue is used by default.

Load Behaviors Supported by query_band

query_band is a session-level GUC parameter. It is a job identifier of the character data type. Its value can be any string. However, for easier differentiation and configuration, query_band only identifies key-value pairs. For example:

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

JobName=abc, **AppName=test**, and **UserName=user** are independent key-value pairs. Specifications of the query_band key-value pairs:

- query_band is set in key-value pair mode, that is, 'key=value'. Multiple query_band key-value pairs can be set in a session. Multiple key-value pairs are separated by semicolons (;). The maximum length of both the **query_band** key-value pair and parameter value is 1024 characters.
- The query_band key-value pair supports the following valid characters: digits 0 to 9, uppercase letters A to Z, lowercase letters a to z, '.', '-', '_', and '#'.

query_band is configured, and identifies load behaviors, using key-value pairs. The supported load behaviors are described in **Table 7-1**.

Table 7-1 Load behaviors supported by QUERY_BAND

Type	Behavior	Behavior Description
Workload management (workload)	Resource pool (respool)	query_band associated with a resource pool
Workload management (workload)	Priority	Priority in the queue
Order	Queue (respool) Currently, this field is invalid and is used for future extension.	**query_band** query order

The "Type" is used to classify load behaviors. Different load behaviors may belong to a same type. For example, both "Resource pool" and a "Priority" belong to "Workload management". The "Behavior" indicates a load behavior associated with a query_band key-value pair. The "Behavior description" describes a specific load behavior. The "Order" in the "Type" is used to indicate the priority of the query_band load behavior identification. When a session has multiple query_band key-value pairs, the query_band key-value pair with a smaller order value is preferentially used to identify a load behavior. Each query_band key-value pair can have multiple associated load behaviors, while one load behavior can only have one associated key-value pair. The query_band load behavior is described as follows:

- Resource pool: query_band can be associated with resource pools. During job execution, if a resource pool is associated with query_band, the resource pool is used in preference. Otherwise, the resource pool associated with the user is used.
  - When query_band is associated with a resource pool, an error is reported if the resource pool does not exist, and the association fails.
  - When query_band is associated with a resource pool, the dependency between query_band and the resource pool is recorded.
  - When a resource pool associated with query_band is deleted, a message is displayed indicating that the resource pool fails to be deleted because of the dependency between query_band and the resource pool.
- Intra-queue priority: query_band can be associated with job priorities, including high, medium, and low. Rush is provided as a special priority (green channel). The default priority is medium. In practice, most jobs use the medium priority, low-priority jobs use the low priority, and privileged jobs use the high priority. It is not recommended that a large number of jobs use the high priority. The rush priority is used only in special scenarios and is not recommended in normal cases.

  The intra-queue priority is used to implement the queuing priority.

  - In the static load management scenario, when the CN concurrency is insufficient, CN global queuing is triggered. The CN global queue is a priority queue.
  - In the dynamic load management scenario, if the DN memory is insufficient, CCN global queuing is triggered. The CCN global queue is a priority queue.
  - When the resource pool concurrency or memory is insufficient, resource pool queuing is triggered. The resource pool queue is a priority queue.

  The preceding priority queues comply with the following scheduling rules:

  - Jobs with a higher priority are scheduled first.
  - After all jobs with a high priority are scheduled, jobs with a low priority are scheduled.
  - In dynamic load management scenarios, the CN global queue does not support the query_band priority.

- Order: The identification order of query_bands can be configured. The default order value is **-1**. Except the default order value, there are no two query_bands with the same order value. The query_band order is verified when being configured. If there are query_bands with the same order value, the order values are recursively increased by 1 until there are no query_bands with the same order value.
  - If a session has multiple query_band key-value pairs, the query_band key-value pair with a smaller order value is used for load identification.
  - **0** is the smallest order value, and the default order value **-1** is the largest order value.
  - If the query_bands are all of the same order value, the anterior query_band is used for load identification.
  - For example, if in **set query_band='b=1;a=3;c=1'; b=1**, the order value of **b=1** is **-1**, **a=3** is **4**, **c=1** is **1**, **c=1** is used as the query_band for load identification. This design enables load administrators to adjust load scheduling.

Application and Configuration of query_band

- The **pg_workload_action** cross-database system catalog is used to store the query_band action and order. For details, see **PG_WORKLOAD_ACTION**.
- The default action and order are not stored in the **pg_workload_action** system catalog. If a non-default action is set for query_band, the default action is also displayed when actions are queried. The message <query_band information not found> is displayed when the action and order to be queried are the default query_band action.
- The **gs_wlm_set_queryband_action** function sets the query_band sequence. The maximum length of the first parameter, that is, the query_band key value pair, is 63 characters. For the second parameter, it is case insensitive and multiple actions are separated by semicolons (;). **order** is the default parameter and its default value is **-1**. For details, see the **gs_wlm_set_queryband_action** function in section .
- The **gs_wlm_set_queryband_order** function sets the query_band sequence. The maximum length of the first parameter, that is, a query_band key value pair, is 63 characters. The value of query_band must be greater than or equal

to **–1**. Except the default value **–1**, the value of query_band order must be unique. When setting the query_band order, if there are query_bands with the same order values, the original order value is increased by 1. For details, see the **gs_wlm_set_queryband_order** function in section .

- The **gs_wlm_get_queryband_action** function is used to query the query_band action. For details, see **gs_wlm_set_queryband_action** in section .
- **pg_queryband_action** provides the system view for querying all query_band actions. For details, see **PG_QUERYBAND_ACTION**.
- The query_band priority is displayed as an integer in the load management view (**PG_SESSION_WLMSTAT**). The mapping between numbers and priorities is as follows:
  – 0: not controlled by load management
  – 1: low
  – 2: medium
  – 4: high
  – 8: rush
- Permission control: Except initial users, other users have the permission to set and query query_band only when they are authorized.

NOTE

When all running jobs are canceled in batches or the maximum number of concurrent jobs in a queue is 1 and only one queue is running jobs, the CN may be triggered to automatically wake up jobs. As a result, jobs are not delivered by priority.

Examples

Step 1 Set the associated resource pool to **p1**, priority to **rush**, and order to **1** for query_band **JobName to abc**.

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

Step 2 Change the associated resource pool to **p2** for query_band **JobName=abc**.

```
SELECT * FROM gs_wlm_set_queryband_action('JobName=abc','respool=p2');
gs_wlm_set_queryband_action
-----------------------------
 t
(1 row)
```

Step 3 Change the priority to **high** for query_band **JobName=abc**.

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

Step 4 Change the order to **3** for query_band **JobName=abc**.

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

Step 5 Query the load behaviors associated with query_band.

```
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)
```

----End

8 Common Performance Parameter Optimization Design

To improve the cluster performance, you can use multiple methods to optimize the database, including hardware configuration, software driver upgrade, and internal parameter adjustment of the database. This section describes some common parameters and recommended configurations.

1. query_dop

   Set the user-defined query parallelism degree.

   The SMP architecture uses abundant resources to obtain time. After the plan parallelism is executed, more resources are consumed, including the CPU, memory, I/O, and network bandwidth. As the DOP grows, the resource consumption increases.

   – When resources become a bottleneck, the SMP cannot improve the performance and may even deteriorate the performance. In the case of a resource bottleneck, you are advised to disable the SMP.
   – If resources are sufficient, the higher the DOP, the more the performance is improved.

   The SMP DOP can be configured at a session level and you are advised to enable the SMP before executing the query that meets the requirements. After the execution is complete, disable the SMP. Otherwise, SMP may affect services in peak hours.

   You can set **query_dop** to **10** to enable the SMP in a session.

2. enable_dynamic_workload

   Enable dynamic load management. Dynamic load management refers to the automatic queue control of complex queries based on user loads in a database. This fine-tunes system parameters without manual adjustment.

   This parameter is enabled by default. Notes:

   – A CN in the cluster is used as the Central Coordinator (CCN) for collecting and scheduling job execution. Its status will be displayed in **Central Coordinator State**. If there is no CCN, jobs will not be controlled by dynamic load management.
   – Simple query jobs (which are estimated to require less than 32 MB memory) and non-DML statements (statements other than **INSERT**,

UPDATE, **DELETE**, and **SELECT**) have no adaptive load restrictions. Control the upper memory limits for them on a single CN using **max_active_statements**.

- In adaptive load scenarios, the value cannot be increased. If you increase it, memory cannot be controlled for certain statements, such as statements that have not been analyzed.
- Reduce concurrency in the following scenarios, because high concurrency may lead to uncontrollable memory usage.
  - A single tuple occupies excessive memory, for example, a base table contains a column more than 1 MB wide.
  - A query is fully pushed down.
  - A statement occupies a large amount of memory on the CN, for example, a statement that cannot be pushed down or a cursor withholding statement.
  - An execution plan creates a hash table based on the hash join operator, and the table has many duplicate values and occupies a large amount of memory.
  - UDFs are used, which occupy a large amount of memory.

When configuring this parameter, you can set **query_dop** to **0** (adaptive). In this case, the system dynamically selects the optimal DOP between 1 and 8 for each query based on resource usage and plan characteristics. The **enable_dynamic_workload** parameter supports the dynamic memory allocation.

3. max_active_statements

   Specifies the maximum number of concurrent jobs. This parameter applies to all the jobs on one CN.

   Set the value of this parameter based on system resources, such as CPU, I/O, and memory resources, to ensure that the system resources can be fully utilized and the system will not be crashed due to excessive concurrent jobs.

   - If this parameter is set to **-1** or **0**, the number of global concurrent jobs is not limited.
   - In the point query scenario, you are advised to set this parameter to **100**.
   - In an analytical query scenario, set this parameter to the number of CPU cores divided by the number of DNs. Generally, its value ranges from 5 to 8.

4. **session_timeout**

   By default, if a client is in idle state after connecting to a database, the client automatically disconnects from the database after the duration specified by the parameter.

   Value range: an integer ranging from 0 to 86400. The minimum unit is second (s). **0** means to disable the timeout. Generally, you are advised not to set this parameter to **0**.

5. The five parameters that affect the database memory are as follows:

   max_process_memory, **shared_buffers**, **cstore_buffers**, **work_mem**, and **maintenance_work_mem**

– max_process_memory

  max_process_memory is a logical memory management parameter. It is used to control the maximum available memory on a single CN or DN.

  Non-secondary DNs are automatically adapted. The formula is as follows: (Physical memory size) x 0.8/(1 + Number of primary DNs). If the result is less than 2 GB, 2 GB is used by default. The default size of the secondary DN is 12 GB.

– shared_buffers

  Specifies the size of the shared memory used by GaussDB(DWS). If the value of this parameter is increased, GaussDB(DWS) requires more System V shared memory than the default system setting.

  You are advised to set **shared_buffers** to a value less than 40% of the memory. It is used to scan row-store tables. Formula: **shared_buffers** = (Memory of a single server/Number of DNs on a single server) x 0.4 x 0.25

– cstore_buffers

  Specifies the size of the shared buffer used by column-store tables and column-store tables (ORC, Parquet, and CarbonData) of OBS and HDFS foreign tables.

  Column-store tables use the shared buffer specified by **cstore_buffers** instead of that specified by **shared_buffers**. When column-store tables are mainly used, reduce the value of **shared_buffers** and increase that of **cstore_buffers**.

  Use **cstore_buffers** to specify the cache of ORC, Parquet, or CarbonData metadata and data for OBS or HDFS foreign tables. The metadata cache size should be 1/4 of **cstore_buffers** and not exceed 2 GB. The remaining cache is shared by column-store data and foreign table column-store data.

– **work_mem**

  Specifies the size of the memory used by internal sequential operations and the Hash table before data is written into temporary disk files.

  Sort operations are required for **ORDER BY**, **DISTINCT**, and merge joins. Hash tables are used in hash joins, hash-based aggregation, and hash-based processing of **IN** subqueries.

  In a complex query, several sort or hash operations may run in parallel. Each operation will be allowed to use as much memory as this parameter specifies. If the memory is insufficient, data will be written into temporary files. In addition, several running sessions may be performing such operations concurrently. Therefore, the total memory used may be many times the value of **work_mem**.

  The formulas are as follows:

  For non-concurrent complex serial queries, each query requires five to ten associated operations. Configure **work_mem** using the following formula: **work_mem** = 50% of the memory/10.

  For non-concurrent simple serial queries, each query requires two to five associated operations. Configure **work_mem** using the following formula: **work_mem** = 50% of the memory/5.

For concurrent queries, configure **work_mem** using the following formula: **work_mem** = **work_mem** for serial queries/Number of concurrent SQL statements.

  – **maintenance_work_mem**

    Specifies the maximum size of memory used for maintenance operations, involving **VACUUM**, **CREATE INDEX**, and **ALTER TABLE ADD FOREIGN KEY**.

    Setting suggestions:

    If you set this parameter to a value greater than that of **work_mem**, database dump files can be cleaned up and restored more efficiently. In a database session, only one maintenance operation can be performed at a time. Maintenance is usually performed when there are not many sessions.

    When the automatic cleanup process is running, up to **autovacuum_max_workers** times of the memory will be allocated. In this case, set **maintenance_work_mem** to a value greater than or equal to that of **work_mem**.

6. **bulk_write_ring_size**

   Specifies the size of a ring buffer used for parallel data import.

   This parameter affects the database import performance. You are advised to increase the value of this parameter on DNs when a large amount of data is to be imported.

7. The following parameters affect the database connection:

   max_connections and **max_prepared_transactions**

   – max_connections

     Specifies the maximum number of concurrent connections to the database. This parameter affects the concurrent processing capability of the cluster.

     Setting suggestions:

     Retain the default value of this parameter on CNs. Set this parameter on DNs to a value calculated using this formula: Number of CNs x Value of this parameter on a CN.

     If the value of this parameter is increased, GaussDB(DWS) may require more System V shared memory or semaphore, which may exceed the default maximum value of the OS. In this case, modify the value as needed.

   – max_prepared_transactions

     Specifies the maximum number of transactions that can stay in the **prepared** state simultaneously. If the value of this parameter is increased, GaussDB(DWS) requires more System V shared memory than the default system setting.

NOTICE

The value of **max_connections** is related to **max_prepared_transactions**. Before configuring **max_connections**, ensure that the value of **max_prepared_transactions** is greater than or equal to that of **max_connections**. In this way, each session has a prepared transaction in the waiting state.

8. checkpoint_completion_target

   Specifies the target for which the checkpoint is completed.

   Each checkpoint must be completed within 50% of the checkpoint interval.

   The default value is **0.5**. To improve the performance, you can change the value to 0.9.

9. data_replicate_buffer_size

   Specifies the memory used by queues when the sender sends data pages to the receiver. The value of this parameter affects the buffer size used for the replication from the primary server to the standby server.

   The default value is **128 MB**. If the server memory is 256 GB, you can increase the value to 512 MB.