## Vertica Knowledge Base Article

# System Tables for Query Performance

Document Release Date: 8/20/2018

MICRO FOCUS | VERTICA

# Legal Notices


### Warranty

The only warranties for Micro Focus International plc products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. MF shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

### Restricted Rights Legend

Confidential computer software. Valid license from MF required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

### Copyright Notice

© Copyright 2015-2017 Micro Focus International plc

### Trademark Notices

Adobe™ is a trademark of Adobe Systems Incorporated.

Microsoft® and Windows® are U.S. registered trademarks of Microsoft Corporation.

UNIX® is a registered trademark of The Open Group.

This product includes an interface of the 'zlib' general purpose compression library, which is Copyright © 1995-2002 Jean-loup Gailly and Mark Adler.

# Contents

System Tables for Query Performance 4
About This Document 4
Profiling Queries in the Management Console 4
Using vsql Variables for Query Examples 4
Identifying Long-Running Queries 5
QUERY PROFILES System Table 5
Example Queries 5
Analyzing the Results 6
Analyzing Query Execution for Performance Tuning 7
Profiling Query Execution 7
Identifying Query Events 8
Determining Query Phase Execution Time 12
Identifying Top Operator Usage 14
Reviewing SIPs Performance 26
Reviewing Query Plans 28
Reviewing Resource Allocation 31
Projections Used by Query 33
Reviewing Table Partitioning 34
Reviewing Projections 35
Recommendations in this Document 41
For More Information 41

# System Tables for Query Performance

When analyzing options for query tuning, the first step is to analyze the query plan, as described in Reading Query Plans. The query plan explains how Vertica plans to process the query. The query plan describes the plan that the Optimizer calculates as the least-cost plan based on what projections the query accesses and on the table statistics. But the query plan does not provide any information about how the query actually executes.

You can analyze Vertica query performance in a number of different ways. This document describes Verticasystem tables that contain useful information that might shed light on your query performance.

When you execute a query, Vertica stores information in certain system tables. This information tells you what happens during query execution—what tasks the query is performing, what resources the query is using, and if there are any bottlenecks.

This document explains which Vertica system tables provide information about query execution and how to identify ways that you can use that information to analyze your queries' performance.

Performance tuning options for each query are different, so adapt the recommendations in this document for your own specific queries.

## About This Document

### Profiling Queries in the Management Console

In addition to system tables, the Vertica Management Console (MC) provides a user interface that facilitates query performance analysis. This document does not discuss the MC. This document uses SQL queries in vsql to retrieve query performance information.

For details about the Management Console, see Managing Queries in MC in the Vertica documentation.

### Using vsql Variables for Query Examples

This document includes many examples of SQL statements that query Vertica system tables. These tables help you identify performance bottlenecks. Transaction_id and statement_id are the fields that identify profile information in system tables.

Once you identify the transaction id and statement id for your slow queries, assign those values to variables and plug them into the queries in the examples in this document. This way, you don't have to type these values in each query predicate. Assign those values to a variable, as in this example:

```
\set t_id 45035996274879950
```

```
\set s_id 1
```

# Identifying Long-Running Queries

When trying to optimize query performance in your Vertica database, there are two strategies:

• Optimize the slowest query in your system.

• Try to optimize a particular query.

This section explains how to identify the query you are interested in. Once you get the transaction_id and statement_id for that question, subsequent sections of this document explain how to use those IDs to get more detailed information that can help you with tuning query performance.

## QUERY PROFILES System Table

To identify the longest running query, query the QUERY_PROFILES system table.

> Note For more information, see QUERY_PROFILES in the Vertica documentation.

## Example Queries

Take these steps to identify slow queries:

1. To retrieve the slowest query on the store_sales_fact table during a three-hour time period, run a query like the following on the QUERY_PROFILES table:

```
=> SELECT
    DATE_TRUNC('second',query_start::TIMESTAMP),
    session_id,
    transaction_id,
    statement_id,
    node_name,
    LEFT(query,100),
    ROUND((query_duration_us/1000000)::NUMERIC(10,3),3) duration_sec
FROM
    query_profiles
WHERE
    query ILIKE '% store_sales_fact %'
AND query_start BETWEEN '2016-07-24 19:00:00' AND '2016-07-24 22:00:00'
ORDER BY
    duration_sec DESC;
```

2. If you have the session_id, narrow down these results. Query the QUERY_PROFILES table to return all the queries that executed during that session and identify how long they took to execute:

```
=> SELECT
```

```
    DATE_TRUNC('second',query_start::TIMESTAMP),
    session_id,
    transaction_id,
    statement_id,
    node_name,
    LEFT(query,100),
    ROUND((query_duration_us/1000000)::NUMERIC(10,3),3) duration_sec
FROM
    query_profiles
WHERE
    session_id = :s_id
ORDER BY
    duration_sec DESC;
```

3. To identify the slowest queries during a specified period of time, try this query:

```
=> SELECT
    (query_duration_us/1000000)::NUMERIC(10,3) duration_sec,
    session_id,
    transaction_id,
    statement_id,
    node_name,
    LEFT(query,100),
FROM
    query_profiles
WHERE
    query_start BETWEEN '2016-07-24 19:00:0' AND '2016-07-24 22:00:00'
ORDER BY
    duration_sec DESC;
```

### Analyzing the Results

Once you have identified which query you want to analyze, use that query's transaction_id and statement_id to extract the full query statement so that you can profile the query:

```
=> SELECT query FROM query_profiles
   WHERE transaction_id = :t_id and statement_id = :s_id;
 query
-----------------------
 SELECT * FROM online_sales_fact
```

Once you have identified the query, execute the query with the PROFILE keyword. The PROFILE statement saves all the query execution information in the EXECUTION_ENGINE_PROFILES system table:

```
=> PROFILE SELECT * FROM online_sales.online_sales_fact;
NOTICE 4788:  Statement is being profiled
HINT:  Select * from execution_engine_profiles
```

```
WHERE transaction_id=:t_id and statement_id=s_id;
NOTICE 3557:  Initiator memory for
query: [on pool general: 69620 KB, minimum: 69620 KB]
-[ RECORD 1 ]--------------+---------
sale_date_key              | 1730
ship_date_key              | 1735
product_key                | 7285
product_version            | 3
customer_key               | 23378
call_center_key            | 61
online_page_key            | 447
shipping_key               | 66
warehouse_key              | 66
promotion_key              | 629
pos_transaction_number     | 4730816
sales_quantity             | 5
sales_dollar_amount        | 292
...
```

# Analyzing Query Execution for Performance Tuning

Now that you've identified the slow-running queries, you need to get more information about those queries to determine what is causing the performance issues. The rest of this document uses an example SQL statement that queries the VMart schema. The following queries show you how to query Vertica system tables to get specific information that provide useful information for performance tuning.

## Profiling Query Execution

With real-time profiling, you can monitor long-running queries while they execute. Real-time profiling counters are available for all statements while they execute. You can enable profiling using the PROFILE keyword on a specific SQL statement, or you can profile the database and/or the current session. If you have not enabled profiles, then profiling counters are unavailable after the statement completes

### EXECUTION_ENGINE_PROFILES System Table

The EXECUTION_ENGINE_PROFILES system table contains profile information about query execution.

#### Example Queries

First, execute your query with the PROFILE keyword:

- `\timing` -- to return the timing
- `\o /dev/null` -- to discard the output and not present it when executing
- `profile` -- keyword to force storing the information in execution_engine_profiles.

```
=> SELECT DISTINCT s.product_key, p.product_description
  FROM store.store_sales_fact s, public.product_dimension p
```

```
WHERE s.product_key = p.product_key
AND s.product_version = p.product_version
AND s.store_key IN   (
SELECT store_k ey
FROM store.store_dimension
WHERE store_state = 'MA')
ORDER BY s.product_key;
\o
```

**Analyzing the Results**

The preceding query returns the following information and stores the profiling information in the EXECUTION_ENGINE_PROFILES system table.

```
NOTICE 4788:  Statement is being profiled
HINT:  Select * from v_monitor.execution_engine_profiles where transaction_
id=45035996274879950 and statement_id=1;
NOTICE 3557:  Initiator memory for query: [on pool general: 1548488 KB, minimum:
1209481 KB] &lt;== The query executed in the GENERAL resource pool
NOTICE 5077:  Total memory required by query: [1548488 KB] &lt;== Memory required for
the query, which is limited by the pool where the query is executing
Time: First fetch (1000 rows): 718.507 ms. All rows formatted: 832.033 ms &lt;= How
long it took to execute the query
```

The profile information tells you whether the memory provided by the resource pool is sufficient to execute the query. If the memory was not sufficient, use a different pool for the profile analysis. To request a different pool, you need to have the proper permissions to use that resource pool.

```
=> SET SESSION RESOURCE_POOL = <desired_pool>
```

## Identifying Query Events

Two types of events occur for each executed query:

- Optimizer events occur during the query plan preparation by the Optimizer.
- Execution events occur during the query execution.

Query events can be positive or negative events. Some examples of query events are listed in the following tables.

**Negative Optimizer Events**

| Negative Optimizer Event | Description |
| --- | --- |
| DELETE_WITH_NON_ OPTIMIZED_PROJECTION | The Optimizer had to use a slower path to delete rows in a projection. |
| MEMORY_LIMIT_HIT | The Optimizer used all of its allocated memory while |

| Negative Optimizer Event | Description |
|---|---|
| | planning. |
| NO_HISTOGRAM | The Optimizer encountered a predicate on a column for which it does not have a histogram. |

**Positive Optimizer Events**

| Positive Optimizer Event | Description |
|---|---|
| GROUPBY_ PUSHDOWN | The Optimizer pushed a GroupBy past a join. |
| NO_GROUPBY_ PUSHDOWN | The Optimizer couldn't push a GroupBy past a join. |
| NODE_PRUNING | The Optimizer pruned a certain number of Vertica nodes from projection access. |
| TRANSITIVE_ PREDICATE | The Optimizer created a transitive predicate due to a Join condition. |

**Negative Execution Engine Events**

| Negative Execution Engine Event | Description |
|---|---|
| GROUP_BY_SPILLED | GROUP BY key set did not fit in memory; using external sort grouping. |
| JOIN_SPILLED | Inner join did not fit in memory; using external sort merge join. |
| RESEGMENTED_MANY_ ROWS | Many rows were resegmented during plan execution. |
| WOS_SPILL | WOS is full. Data is spilling to a new ROS container. |

**Positive Execution Engine Events**

| Positive Execution Engine Event | Description |
|---|---|
| GROUP_BY_ | In-memory prepass was disabled due to ineffectiveness. |

| Positive Execution Engine Event | Description |
| --- | --- |
| PREPASS_FALLBACK | |
| MERGE_ CONVERTED_TO_ UNION | MERGE converted to UNION, followed by SORT. |
| PARTITIONS_ ELIMINATED | Some storage containers will not be processed because they contain no relevant data. |
| RLE_OVERRIDDEN | Compressed execution will not be used on some columns, because the average run counts are not large enough. |
| SEQUENCE CACHE REFILLED | Session cache for a sequence has been used up. Taking a GLOBAL CATALOG X Lock to replenish the cache. |
| SIP_FALLBACK | Sidewise Information Passing (SIPs) filter was disabled due to ineffectiveness. |
| SMALL_MERGE_ REPLACED | Small StorageMerge was replaced with StorageUnion for efficiency. |

**QUERY_EVENTS System Table**

The QUERY_EVENTS system table tells you what events occurred during query execution.

**Example Queries**

Use the following query to view all query events for a specific transaction and statement:

```
=> SELECT event_type, event_category,
   COUNT(DISTINCT node_name), COUNT(*) FROM query_events
   WHERE transaction_id = :t_id AND statement_id = :s_id
   GROUP BY 1,2 ORDER BY 2;
       event_type        | event_category | count | count
-------------------------+----------------+-------+-------
MERGE_CONVERTED_TO_UNION | EXECUTION      |     3 |     3
SMALL_MERGE_REPLACED     | EXECUTION      |     3 |     9
(2 rows)
```

Use the following query to review the path details at the time the events occurred.

```
=> SELECT node_name, event_type, event_description,
   operator_name, path_id, event_details, suggested_action
   FROM query_events WHERE transaction_id =
   :t_id AND statement_id = :s_id ORDER BY node_name;
```

```
-[ RECORD 1 ]-----+-----------------------------------------------------------
node_name         | v_vmart_node0001
event_type        | MERGE_CONVERTED_TO_UNION
event_description | Merge converted to union, followed by sort.
operator_name     | Sort
path_id           | -1
event_details     | Projection: public.result_table_b0
suggested_action  |
-[ RECORD 2 ]-----+-----------------------------------------------------------
node_name         | v_vmart_node0001
event_type        | SMALL_MERGE_REPLACED
event_description | Small StorageMerge replaced with StorageUnion for efficiency
operator_name     | StorageMerge
path_id           | 4
event_details     | Projection: public.new_addresses_b0
suggested_action  |
-[ RECORD 3 ]-----+-----------------------------------------------------------
node_name         | v_vmart_node0001
event_type        | GROUP_BY_SPILLED
event_description | GROUP BY key set did not fit in memory, using external sort
grouping.
operator_name     | GroupByHash
path_id           | 4
event_details     |
suggested_action  | Consider a sorted projection. Increase memory available to the
plan.
-[ RECORD 4 ]-----+----------------------------------------------------------
node_name         | v_vmart_node0001
event_type        | RESEGMENTED_MANY_ROWS
event_description | Many rows were resegmented during plan execution.
operator_name     | NetworkSend
path_id           | 2
event_details     |
suggested_action  | Consider different projection segmentation.
-[ RECORD 5 ]----+-------------------------------------------------------
...
-[ RECORD 11 ]----+-----------------------------------------------------------
node_name         | v_vmart_node0003
event_type        | SMALL_MERGE_REPLACED
event_description | Small StorageMerge replaced with StorageUnion for efficiency
operator_name     | StorageMerge
path_id           | 4
event_details     | Projection: public.new_addresses_b0
suggested_action  |
...
```

**Analyzing the Results**

For all negative events, review the suggested_action field. Implement that suggestion to see if it improves the performance of that query.

## Determining Query Phase Execution Time

In a healthy system, most query execution takes place during the ExecutePlan phase. But sometimes this is not true. Large amounts of time spent in other phases of query execution may indicate a system issue that needs further analysis by Vertica Support.

### DC_QUERY_EXECUTIONS Table

To see the duration of each query phase, query the DC_QUERY_EXECUTIONS table. Execution time on the initiator node is different than on non-initiator nodes because the initiator node has to perform extra tasks.

### Example Queries

In the following query, v_vmartdb_node0001 (the local node) is the initiator node. As expected, the ExecutePlan phase is much longer than the other phases.

```
=> SELECT
    dc_query_executions.node_name,
    dc_query_executions.transaction_id,
    dc_query_executions.statement_id,
    dc_query_executions.execution_step,
    ((dc_query_executions.completion_time - dc_query_executions."time")) AS duration
FROM
    v_internal.dc_query_executions
WHERE
    transaction_id = :t_id
AND statement_id = :s_id
AND node_name =
    (
        SELECT
            local_node_name())
ORDER BY
    dc_query_executions."time";
      node_name    |  transaction_id   | statement_id |        execution_step
|    duration
-------------------+-------------------+--------------+------------------------------
+-----------------
v_vmartdb_node0001 | 45035996274879950 |            1 | Plan
| 00:00:00.020269
v_vmartdb_node0001 | 45035996274879950 |            1 | InitPlan
| 00:00:00.001941
v_vmartdb_node0001 | 45035996274879950 |            1 | SerializePlan
| 00:00:00.001025
v_vmartdb_node0001 | 45035996274879950 |            1 | PreparePlan
| 00:00:00.01296
v_vmartdb_node0001 | 45035996274879950 |            1 | PreparePlan:TakeTableLocks
| 00:00:00.000005
v_vmartdb_node0001 | 45035996274879950 |            1 | PreparePlan:DistPlanner
| 00:00:00.000565
v_vmartdb_node0001 | 45035996274879950 |            1 | PreparePlan:LocalPlan
| 00:00:00.001329
v_vmartdb_node0001 | 45035996274879950 |            1 | PreparePlan:EEcompile
```

```
| 00:00:00.00162
  vmartdb_node0001 | 45035996274879950 |            1 | CompilePlan
| 00:00:00.009477
v_vmartdb_node0001 | 45035996274879950 |            1 | CompilePlan:ReserveResources
| 00:00:00.000046
v_vmartdb_node0001 | 45035996274879950 |            1 | CompilePlan:EEpreexecute
| 00:00:00.005646
v_vmartdb_node0001 | 45035996274879950 |            1 | ExecutePlan
| 00:00:00.756467
v_vmartdb_node0001 | 45035996274879950 |            1 | AbandonPlan
| 00:00:00.004101
(13 rows)
```

A similar query shows that the executor node v_test_db_node0002 spent time in the following phases.

```
v_test_db_node0002 | 45035996273718437 |            8 | PreparePlan:DeserializePlan
| 00:00:00.009016
v_test_db_node0002 | 45035996273718437 |            8 | PreparePlan:TakeTableLocks
| 00:00:00.000009
v_test_db_node0002 | 45035996273718437 |            8 | PreparePlan:DistPlanner
| 00:00:00.000168
v_test_db_node0002 | 45035996273718437 |            8 | PreparePlan:LocalPlan
| 00:00:00.002264
v_test_db_node0002 | 45035996273718437 |            8 | PreparePlan:EEcompile
| 00:00:00.009847
v_test_db_node0002 | 45035996273718437 |            8 | CompilePlan:ReserveResources
| 00:00:00.000116
v_test_db_node0002 | 45035996273718437 |            8 | CompilePlan:EEpreexecute
| 00:00:00.019912
```

**Analyzing the Results**

To understand these results, you need to understand what happens in each query execution phase. The following table describes each phase and what might occur during that phase to impact query performance.

| Phase(s) | Description | Likely causes of performance issues |
|---|---|---|
| Plan, InitPlan, Serialize Plan, AbandonPlan | These phases occur only on the initiator node. | Any slowness during these phases means that the Optimizer took longer than expected to plan the query. Slowness during these phases is commonly related to concurrency. The Optimizer needs to take a catalog lock to plan a query.<br><br>Alternatively, slowness during these phases could indicate UDP issues when Vertica uses spread to send the plan to execute to other |

| Phase(s) | Description | Likely causes of performance issues |
| --- | --- | --- |
| | | nodes and or to send an AbandonPlan message to other nodes. |
| ExecutePlan | This phase is the actual execution of the query. Vertica stores the details about the query execution in EXECUTION_ENGINE_ PROFILES. | If the slowness of the query occurs during the ExecutePlan phase, use the next few example queries on the EXECUTION_ENGINE_PROFILES table to identify the root cause of the slowness. |
| CompilePlan | This phase executes on all the nodes and has two parts as follows:<br>• The first part, the ReserveResources phase, shows the time that Vertica process took to reserve the resources.<br>• The second part is the EEpreexecute phase. The EEpreexecute phase prepares the system to execute a particular operator. | The length of the EEPrexecute time depends on the operator and can include tasks, such as allocating memory, starting threads, and opening network connections. |

## Identifying Top Operator Usage

Operators are Execution Engine components that work with the data and then move data to the next operator.

**EXECUTION_ENGINE_PROFILES System Table**

The EXECUTION_ENGINE_PROFILES tables stores the time spent by each operator can be seen in the EXECUTION_ENGINE_PROFILES table in the counter initialization time (us) value for each operator.

If your query is slow in the ExecutePlan phase, you can find more information about that phase in the EXECUTION_ENGINE_PROFILES system table. The EXECUTION_ENGINE_PROFILES table can contain thousands of rows, so it's a good idea to aggregate the data to explore this table's contents.

For the best results, aggregate the rows by operator, query plan path ID, and node_name.

The following operators might appear in a query plan.

| Operators | Description | Considerations |
|---|---|---|
| Copy | During load, makes a copy of the data for buddy projections. | |
| DataTarget | During load, writers data to the WOS or ROS. | |
| ExprEval | Evaluations expressions, for example C1+C2. | Select only the columns that you need to evaluate the expression. |
| Filter | Filters tuples to the next operator. | |
| GroupByHash | Aggregates tuples in a hash in memory. | This operator uses all available memory.<br>If the operator doesn't have enough memory, data spills to disk. This operator must complete before the next operator starts. |
| GroupByPipe | Aggregates tuples that are sorted in order to stream data to the next operator. | Uses less memort than a hash join. |
| JoinMerge | Joins pre-sorted tuples. | Uses less memory than a hash join. |
| JoinHash | Joins tuples that are not pre-sorted by loading the inner side of a join in memory. | If the inner join is large and does not fit in memory, the query fails.<br>If the inner join is small, JoinHash can be faster than JoinMerge. |
| Load | Loads data from disk and parses the input. | |
| Merge | Merges data streams in one sorted stream. | |
| NetworkRecv<br>NetworkSend | Amount of data that is sent to or received from other notes. | Try to design your projections so that this operator doesn't occur in the middle of your query plan.<br>Data is streamed in a single thread. |

| Operators | Description | Considerations |
| --- | --- | --- |
| | | For each pair of network operators (send/receive), more memory is required so more memory is reserved for the query.<br><br>Memory requests for data buffers increase proportionally with the number of nodes. |
| ParallelMerge | Combines sorted data streams. | |
| ParallelUnion | Combines data streams, not necessarily sorted. | |
| Root | The first operator. | |
| Scan | Reads data from disk and applies filter. | |
| Sort | Sorts a data stream. | |
| StorageMerge | Combines storage, retaining the sort order. | |
| StorageUnion | Combines storage without retaining the sort order. | |
| TopK | Analytic function that returns the top N tuples. | |
| Val | Evaluates expressions in joins such as tableA.C1=tableB.C2. | |

**Example Query: Identifying the Slowest Path and Operator**

The following query helps identify the slowest path and the Execution Engine operator that was executing in that path. For slow queries with many paths and operators, focus your analysis on paths and operators that seem likely to have performance issues.

> Note The aggregation in this query can be misleading because it calculates a total for each operator without considering how many nodes or threads were executing in parallel.

```
=> SELECT
```

```
    operator_name,
    path_id,
    SUM(counter_value)
FROM
    execution_engine_profiles
WHERE
    transaction_id = :t_id
AND statement_id = :s_id
AND counter_name ILIKE 'execution%'
GROUP BY
    operator_name,
    path_id
ORDER BY
    3 DESC LIMIT 20;
 operator_name | path_id |   sum
---------------+---------+---------
 Scan          |       5 | 1501914
 Join          |       4 |  659055
 GroupByHash   |       2 |  307442
 StorageUnion  |       2 |  120715
 Join          |       3 |  111232
 Root          |      -1 |   34964
 ExprEval      |       3 |   31785
 NetworkSend   |       2 |   27947
 ParallelUnion |       2 |   13525
 GroupByPipe   |       2 |    7594
 NetworkRecv   |       2 |    6775
 Scan          |       6 |    6292
 NetworkSend   |       6 |    5328
 NetworkRecv   |       6 |    3683
 GroupByPipe   |       8 |    1865
 StorageUnion  |       6 |    1312
 NetworkRecv   |       8 |    1118
 NewEENode     |      -1 |     955
 Scan          |       8 |     879
 ParallelMerge |       2 |     715
(20 rows)
```

**Example Query: Identifying the Slowest Path and Operator by Node**

To see the slowest path and operator by node, aggregate by node_name, as in the next query. If one node shows a longer execution time than the other nodes, the data may be skewed or the slowest node may be slow for some other reason.

For example, if a SCAN operator is slow and one node is slower than the others, that node may have a slower disk than the others. If that is the case, use vioperf to see the I/O throughput of the nodes. If a NetworkSend operator is slow, there may be network issues. Use `netstat` to see if there are any TPC package issues.

```
=> SELECT
    node_name,
```

```
    operator_name,
    path_id,
    SUM(counter_value)          sum_time,
    COUNT(DISTINCT operator_id) num_operators
FROM
    dc_execution_engine_profiles
WHERE
    transaction_id = :t_id
AND statement_id = :s_id
AND counter_name ILIKE 'execution%'
GROUP BY
    node_name,
    operator_name,
    path_id
ORDER BY
    4 DESC LIMIT 20;
       node_name    | operator_name | path_id | sum_time | num_operators
--------------------+---------------+---------+----------+---------------
 v_vmartdb_node0001 | Scan          |       5 |   544160 |             2
 v_vmartdb_node0003 | Scan          |       5 |   498164 |             2
 v_vmartdb_node0002 | Scan          |       5 |   459590 |             2
 v_vmartdb_node0001 | Join          |       4 |   225466 |             2
 v_vmartdb_node0002 | Join          |       4 |   223134 |             2
 v_vmartdb_node0003 | Join          |       4 |   210455 |             2
 v_vmartdb_node0002 | GroupByHash   |       2 |   106011 |             4
 v_vmartdb_node0001 | GroupByHash   |       2 |   105288 |             4
 v_vmartdb_node0003 | GroupByHash   |       2 |    96143 |             4
 v_vmartdb_node0001 | StorageUnion  |       2 |    40551 |             2
 v_vmartdb_node0002 | StorageUnion  |       2 |    40251 |             2
 v_vmartdb_node0003 | StorageUnion  |       2 |    39913 |             2
 v_vmartdb_node0001 | Join          |       3 |    37540 |             2
 v_vmartdb_node0003 | Join          |       3 |    37466 |             2
 v_vmartdb_node0002 | Join          |       3 |    36226 |             2
 v_vmartdb_node0001 | Root          |      -1 |    34964 |             1
 v_vmartdb_node0001 | ExprEval      |       3 |    10858 |             2
 v_vmartdb_node0001 | NetworkSend   |       2 |    10780 |             2
 v_vmartdb_node0003 | ExprEval      |       3 |    10543 |             2
 v_vmartdb_node0002 | ExprEval      |       3 |    10384 |             2
(20 rows)
```

The num_operators column shows how many operators ran in parallel to compute the query results. This count is usually managed by the EXECUTIONPARALLELISM resource pool. However, depending on other conditions such as the number of ROS containers, the number of concurrent operators can be less than the resource pool configuration specifies.

**Example Queries: Viewing Counter Details**

After you have identified the slowest path, get more details from each operator’s counters in that path.

Each operator has different counters. The following table describes the counters that might give some insight about query performance.

| Counter | Description |
| --- | --- |
| execution time (us) | CPU time spent by thread, excluding wait time. |
| clock time (us) | Time intervals of an operator, including wait time. |
| initialization time (us) | Time spent initializing an operator. This time can include tasks such as allocating memory, starting threads, and opening network connections. Each operator performs different tasks. |
| start time/end time (us) | Start/stop time of a single operator. |
| rows processed | Data rows processed by the operator. |
| input queue wait (us) | Time spent by the Execution Engine waiting for upstream operators. |
| memory reserved (bytes) | Memory requested by the Optimizer to the Resource Manager. The Optimizer uses statistics to estimate how much memory will be needed. |
| memory allocated (bytes) | Memory allocated by the Execution Engine operators when executing the query. When the query startsm, the memory is reserved, but the memory is not allocated until the operators need it. |
| file handles | The number of files that need to be opened. This number depends on the number of columns and ROS containers that need to be opened to read the needed information. |
| bytes received/bytes sent | Number of bytes received ot sent by the query. |
| rows received/sent | Number of data rows received and sent by the query. |
| RLE rows produces | Number of tuples produced by the operator that are still in RLE format as stored on disk. Shows that Vertica has not materialized the column ter and the operator was able to work with compressed data. |
| rows produced | Number of logical rows produced by the operator. |

| Counter | Description |
| --- | --- |
| consumer stall (us) | Amount of time that the operator is waiting to get data from the previous operator. |
| size of raw temp data (bytes) | The size of the data that spilled to disk. |

The following query returns the details of all counters in the local path on the local node and the average value.

If the execution time on all the nodes is similar, filter the data on just on the local node so that your analysis queries execute faster and use fewer resources. To filter the analysis to just the local node, use the function LOCAL_NODE_NAME. If one node is slower than the other nodes, perform the analysis on the slowest node.

The count column indicates how many instances of each operator were executed in parallel. The avg column represents the average value per counter across all nodes.

```
=> \set path_id 5
=> SELECT
    operator_name,
    counter_name,
    path_id,
    COUNT(DISTINCT operator_id),
    AVG(counter_value)
FROM
    execution_engine_profiles
WHERE
    transaction_id = :t_id
AND statement_id = :s_id
AND path_id = :path_id
AND node_name =
    (
        SELECT
            LOCAL_NODE_NAME())
GROUP BY
    1,2,3
HAVING
    SUM(counter_value) > 0 ORDER BY 1;
 operator_name |                 counter_name                 | path_id | count |
   avg
---------------+----------------------------------------------+---------+-------+---
---------------
 Scan          | blocks analyzed by SIPs expression           |       5 |     2 |
       40636
 Scan          | bytes read from cache                        |       5 |     2 |
    23560280
 Scan          | bytes read from disk                         |       5 |     2 |
```

```
      7970589
 Scan          | clock time (us)                              |       5 |     2 |
       647620
 Scan          | current memory padding (bytes)               |       5 |     2 |
      2592620
 Scan          | current unbalanced memory allocations (count) |       5 |     2 |
          330
 Scan          | current unbalanced memory capacity (bytes)   |       5 |     2 |
      4210688
 Scan          | current unbalanced memory overhead (bytes)   |       5 |     2 |
         6720
 Scan          | current unbalanced memory padding (bytes)    |       5 |     2 |
          996
 Scan          | current unbalanced memory requested (bytes)  |       5 |     2 |
      2026748
 Scan          | end time                                     |       5 |     2 |
2914941832314339
 Scan          | estimated rows produced                      |       5 |     2 |
     60000000
 Scan          | execution time (us)                          |       5 |     2 |
       544160
 Scan          | initialization time (us)                     |       5 |     2 |
          430
 Scan          | memory allocated (bytes)                     |       5 |     2 |
      4488072
 Scan          | number of cancel requests received           |       5 |     2 |
           12
 Scan          | peak file handles                            |       5 |     2 |
           12
 Scan          | peak memory allocations (count)              |       5 |     2 |
           54
 Scan          | peak memory padding (bytes)                  |       5 |     2 |
      2592620
 Scan          | peak memory requested (bytes)                |       5 |     2 |
      1693456
 Scan          | peak unbalanced memory allocations (count)   |       5 |     2 |
          332
 Scan          | peak unbalanced memory capacity (bytes)      |       5 |     2 |
      4210688
 Scan          | peak unbalanced memory overhead (bytes)      |       5 |     2 |
         6720
 Scan          | peak unbalanced memory padding (bytes)       |       5 |     2 |
          996
 Scan          | peak unbalanced memory requested (bytes)     |       5 |     2 |
      2092284
 Scan          | rle rows produced                            |       5 |     2 |
       404250
 Scan          | rows filtered by SIPs expression             |       5 |     2 |
      9595008
 Scan          | rows processed                               |       5 |     2 |
      9999258
 Scan          | rows processed by SIPs expression            |       5 |     2 |
     10060698
```

```
 Scan          | rows produced                                     |       5 |      2 |
      404250
 Scan          | start time                                        |       5 |      2 |
291494183123736
```

The following query identifies specific counter values as pivoted values so you can easily compare paths and operators. Your specific use case dictates which counters you're interested in.

```
=> SELECT path_id, operator_name, COUNT(DISTINCT operator_id) num_operators, COUNT
(DISTINCT node_name) num_nodes
       , SUM(DECODE(counter_name, 'bytes received', counter_value, NULL)) AS 'bytes_
received'
       , SUM(DECODE(counter_name, 'bytes sent', counter_value, NULL)) AS 'bytes_sent'
       , SUM(DECODE(counter_name, 'execution time (us)', counter_value, NULL)) AS
'execution_time_us'
       , SUM(DECODE(counter_name, 'rows received', counter_value, NULL)) AS 'rows_received'
       , SUM(DECODE(counter_name, 'rle rows produced', counter_value, NULL)) AS 'rle_rows_
produced'
       , SUM(DECODE(counter_name, 'rows produced', counter_value, NULL)) AS 'rows_prod'
       , SUM(DECODE(counter_name, 'consumer stall (us)', counter_value, NULL)) AS 'cons_
stall'
       , SUM(DECODE(counter_name, 'producer stall (us)', counter_value, NULL)) AS 'prod_
stall'
       , SUM(DECODE(counter_name, 'cumulative size of raw temp data (bytes)', counter_
value, NULL))
       AS ' temp_data' FROM execution_engine_profiles
       WHERE transaction_id= :t_id AND statement_id= :s_id
       AND node_name = (SELECT LOCAL_NODE_NAME())GROUP BY 1, 2  ORDER BY 1,2 ;
 path_id | operator_name | num_operators | num_nodes | bytes_received | bytes_sent |
execution_time_us | rows_received | rle_rows_produced | rows_produced | consumer_
stall | producer_stall | cumulative size of raw temp data
---------+---------------+---------------+-----------+----------------+------------+-
------------------+---------------+-------------------+---------------+--------------
--+----------------+---------------------------------
      -1 | NewEENode     |             1 |         1 |                |            |
            907 |               |             55619 |         55619 |
|                |
      -1 | Root          |             1 |         1 |                |            |
          34964 |               |                   |         55619 |
|                |
       2 | GroupByHash   |             4 |         1 |                |            |
         105288 |               |             56086 |         56086 |
|                |                            0
       2 | GroupByPipe   |             2 |         1 |                |            |
           2356 |               |            404250 |        404250 |
|                |
       2 | NetworkRecv   |             2 |         1 |        1659961 |            |
           4353 |         93335 |             93335 |         93335 |
|                |
       2 | NetworkSend   |             2 |         1 |                |     673086 |
```

```
          10780 |               |             31144 |          31144 |         2539735
|         67437 |
      2 | ParallelMerge |             1 |           1 |                |             |
            239 |               |             18621 |          18621 |
|               |
      2 | ParallelUnion |             3 |           1 |                |             |
           4138 |               |             75181 |          75181 |
|               |
      2 | StorageUnion  |             2 |           1 |                |             |
          40551 |               |            404250 |         404250 |
|               |
      3 | ExprEval      |             2 |           1 |                |             |
          10858 |               |            404250 |         404250 |
|               |
      3 | Join          |             2 |           1 |                |             |
          37540 |               |            404250 |         404250 |
|               |                                 0
      4 | Join          |             2 |           1 |                |             |
         225466 |               |            404250 |         404250 |
|               |                                 0
      5 | Scan          |             2 |           1 |                |             |
         544160 |               |            404250 |         404250 |
|               |
      6 | NetworkRecv   |             1 |           1 |        1073992 |             |
            635 |         60000 |             60000 |          60000 |
|               |
      6 | NetworkSend   |             1 |           1 |                |     1075688 |
           2887 |               |             20045 |          20045 |           34962
|             0 |
      6 | Scan          |             1 |           1 |                |             |
           2550 |               |             20045 |          20045 |
|               |
      6 | StorageUnion  |             1 |           1 |                |             |
            428 |               |             20045 |          20045 |
|               |
      8 | GroupByHash   |             1 |           1 |                |             |
             21 |               |                 4 |              4 |
|               |                                 0
      8 | GroupByPipe   |             1 |           1 |                |             |
            612 |               |                 8 |              8 |
|               |
      8 | NetworkRecv   |             1 |           1 |            100 |             |
            417 |            10 |                10 |             10 |
|               |
      8 | NetworkSend   |             1 |           1 |                |         102 |
             11 |               |                 4 |              4 |           15432
|             0 |
      8 | Scan          |             1 |           1 |                |             |
            354 |               |                 4 |              8 |
|               |
      8 | StorageUnion  |             1 |           1 |                |             |
            165 |               |                 8 |              8 |
|               |
```

```
(23 rows)
```

The following query identifies the specific counter values as pivoted values so you can easily compare counter values in the different paths and operators. Your specific use case dictates the counters that you need to evaluate.

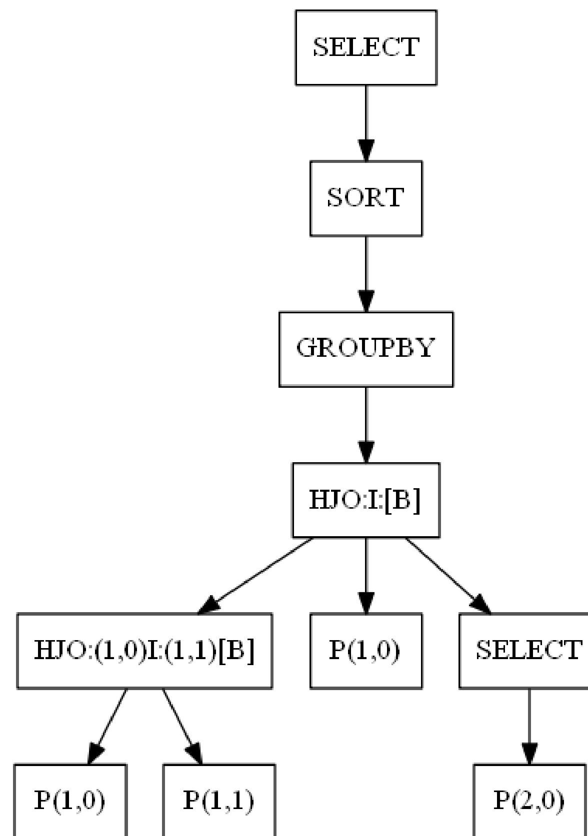
```
=> SELECT path_id, operator_name, count(distinct operator_id) num_operators, count
(distinct node_name) num_nodes
, sum(DECODE(counter_name, 'bytes received', counter_value, NULL)) AS 'bytes_
received'
, sum(DECODE(counter_name, 'bytes sent', counter_value, NULL)) AS 'bytes_sent'
, sum(DECODE(counter_name, 'execution time (us)', counter_value, NULL)) AS
'execution_time_us'
, sum(DECODE(counter_name, 'rows received', counter_value, NULL)) AS 'rows_received'
, sum(DECODE(counter_name, 'rle rows produced', counter_value, NULL)) AS 'rle_rows_
produced'
, sum(DECODE(counter_name, 'rows produced', counter_value, NULL)) AS 'rows_prod'
, sum(DECODE(counter_name, 'consumer stall (us)', counter_value, NULL)) AS 'cons_
stall'
, sum(DECODE(counter_name, 'producer stall (us)', counter_value, NULL)) AS 'prod_
stall'
, sum(DECODE(counter_name, 'cumulative size of raw temp data (bytes)', counter_value,
NULL)) AS ' temp_data'
from dc_execution_engine_profiles where transaction_id= :t_id and statement_id= :s_id
and node_name = (select local_node_name()) group by 1, 2  order by 1,2 ;
 path_id | operator_name | num_operators | num_nodes | bytes_received | bytes_sent |
execution_time_us | rows_received | rle_rows_produced | rows_prod | cons_stall |
prod_stall | temp_data
---------+--------------+--------------+----------+----------------+------------+-
------------------+--------------+------------------+-----------+-----------+-----
-------+----------
      -1 | NewEENode     |             1 |         1 |                |            |
            1907 |               |             55619 |     55619 |            |
     |
      -1 | Root          |             1 |         1 |                |            |
           84804 |               |                   |     55619 |            |
     |
       2 | GroupByHash   |             8 |         1 |                |            |
          115847 |               |             45751 |     45751 |            |
     |         0
       2 | GroupByPipe   |             2 |         1 |                |            |
           31552 |               |             50581 |     50581 |            |
     |
       2 | NetworkRecv   |             2 |         1 |        1752275 |            |
           42063 |         87732 |             87732 |     87732 |            |
     |
       2 | NetworkSend   |             2 |         1 |                |     640179 |
            4509 |               |             22017 |     22017 |    3439817 |
   0 |
       2 | ParallelMerge |             1 |         1 |                |            |
           15525 |               |             14024 |     14024 |            |
```

```
     |
       2 | ParallelUnion |            5 |         1 |                |            |
            4998 |               |             63840 |     63840 |            |
     |
       2 | StorageUnion  |            4 |         1 |                |            |
           10491 |               |             50581 |     50581 |            |
     |
       3 | ExprEval      |            2 |         1 |                |            |
            6482 |               |             50583 |     50583 |            |
     |
       3 | Join          |            2 |         1 |                |            |
           21632 |               |             50583 |     50583 |            |
     |         0
       4 | Join          |            2 |         1 |                |            |
          278566 |               |             50583 |     50583 |            |
     |         0
       5 | Scan          |            2 |         1 |                |            |
          293481 |               |             50583 |     50583 |            |
     |
       6 | NetworkRecv   |            1 |         1 |        1208577 |            |
            8309 |         60000 |             60000 |     60000 |            |
     |
       6 | NetworkSend   |            1 |         1 |                |    1211754 |
            2046 |               |             15052 |     15052 |     346612 |
   0 |
       6 | Scan          |            1 |         1 |                |            |
           80978 |               |             15052 |     15052 |            |
     |
       6 | StorageUnion  |            1 |         1 |                |            |
            2632 |               |             15052 |     15052 |            |
     |
       8 | GroupByHash   |            1 |         1 |                |            |
            2332 |               |                 4 |         4 |            |
     |         0
       8 | GroupByPipe   |            1 |         1 |                |            |
           12961 |               |                 4 |         4 |            |
     |
       8 | NetworkRecv   |            1 |         1 |             99 |            |
            6988 |            10 |                10 |        10 |            |
     |
       8 | NetworkSend   |            1 |         1 |                |        153 |
              67 |               |                 4 |         4 |     356037 |
   0 |
       8 | Scan          |            1 |         1 |                |            |
           42909 |               |                 4 |         4 |            |
     |
       8 | StorageUnion  |            1 |         1 |                |            |
            1237 |               |                 4 |         4 |            |
     |
(23 rows)
```

Having the data in a pivot format, you can easily to see the transition of counter values between operators and identify possible issues with the flow of data while executing the query. Remember that a query executes from bottom up, but its paths may happen in parallel.

For example, in the previous query, the path 3 join needs to wait for the earlier path to complete before it can execute; that is, it must evaluate the expression created in the earlier path. Paths 4 and 8 can execute in parallel.

The query graph, as explained in Reading Query Plans, helps you see how operators work in parallel or sequentially. For an example of a query graph, see the following Simplified Join graph as follows. The simplified plan does not have the path name, but this graph gives a clear picture of what operators run in parallel and what operators need data from previous operator to start work:




## Reviewing SIPs Performance

When SIPs (Sidewise Information Passing), Vertica filters tuples from the outer table of a join before going through the join operator. The records to be filtered are based on the predicates on the inner join table of the join; those predicates are on columns that are not part of the join key.

Filtering the tuples before the join can improve performance because Vertica materializes only the records of columns that meet the conditions of the joins.

Vertica 7.2.x includes three enhancements to the SIPs capabilities:

- Vertica applies the SIPs optimization to both merge joins and hash joins.
- The EXECUTION_ENGINE_PROFILES table now has the counter_tag column. counter_tag is a string that uniquely identifies the counter for operators that might need to distinguish between different instances.
- New SIPs-related counters.
  - SIPsProcessedRows: Number of rows processed by SIPs expression
  - SIPSPrunedRows: Number of rows filtered by SIPs expression

Looking at counters in the SIPs operator helps you see that the extra filter in the outer join reduces the number of tuples that the query needs to process. Sometimes, the number of tuples that are reduced is too small to justify the extra filter. In those cases, disabling SIPs for that particular query may improve its performance.

To disable SIPs for a particular query, add the following hint to your query:

```
/*+add_vertica_options(BASIC,DISABLE_SIPS) */
```

### DC_SIPS_STATISTICS System Table

To complete the analysis of SIPs information, Vertica added a new Data Collector table in 7.2.x: DC_SIPS_STATISTICS. In this table, you can see the number of rows that the SIPs process pruned. The number of rows pruned may not offset the amount of time it takes for SIPs to prune the rows. When this happens, the performance improvement won't be significant, and you should disable SIPs.

### Example Query

```
=> SELECT
    node_name,
    TIME,
    sip_expr_id,
    sip_entries,
    rows_processed,
    rows_pruned,
    blocks_processed,
    blocks_pruned,
    blocks_pruned_BY_valuelist
FROM
    dc_sips_statistics
WHERE
    transaction_id = :t_id
AND statement_id = :s_id
AND node_name =
    (
```

```
        SELECT
            LOCAL_NODE_NAME());
-[ RECORD 1 ]--------------+------------------------------
node_name                  | v_vmartdb_node0001
time                       | 2016-03-08 21:01:56.275266+00
sip_expr_id                | 2
sip_entries                | 20154
rows_processed             | 20480
rows_pruned                | 0
blocks_processed           | 10159
blocks_pruned              | 0
blocks_pruned_by_valuelist | 0
-[ RECORD 2 ]--------------+------------------------------
node_name                  | v_vmartdb_node0001
time                       | 2016-03-08 21:01:56.275297+00
sip_expr_id                | 3
sip_entries                | 5
rows_processed             | 20480
rows_pruned                | 0
blocks_processed           | 10159
blocks_pruned              | 0
blocks_pruned_by_valuelist | 0
-[ RECORD 3 ]--------------+------------------------------
node_name                  | v_vmartdb_node0001
time                       | 2016-03-08 21:01:56.275355+00
sip_expr_id                | 2
sip_entries                | 4
rows_processed             | 60000
rows_pruned                | 20480
blocks_processed           | 0
blocks_pruned              | 10159
blocks_pruned_by_valuelist | 0
```

#### Analyzing the Results

If the SIPs process pruned a large number of rows, then the overhead associated with SIPs is worth it. If the SIPs process pruned no rows, or a small number of rows, the SIPs process is not providing any benefit.

## Reviewing Query Plans

You should analyze query execution in parallel with reviewing the query plan so that you can understand the data flow.

### QUERY_PLAN_PROFILES System Table

When you execute a query, Vertica saves the query plan in the QUERY_PLAN_PROFILES system table.

#### Example Queries

The following query shows the query plan plus additional information such as running time and memory allocated on the specific path. In this query, the path description is truncated to 120 characters for presentation purposes.

```
=> SELECT
    path_id,
    path_line_index pos,
    running_time,
    memory_allocated_bytes AS mem_alloc,
    read_from_disk_bytes read_from_disk,
    LEFT(path_line, 70) FROM query_plan_profiles
WHERE
    transaction_id = :t_id
AND
    statement_id = :s_id
ORDER BY path_id,path_line_index;
 path_id | pos |  running_time   | mem_alloc | read_from_disk |
           left
---------+-----+-----------------+-----------+----------------+---------------------
--------------------------------------------------
       2 |   1 | 00:00:00.765351 | 746200704 |                | +-GROUPBY HASH (SORT
OUTPUT) (GLOBAL RESEGMENT GROUPS) (LOCAL RESEGMEN
       2 |   2 |                 |           |                | |  Group By:
s.product_key, p.product_description
       2 |   3 |                 |           |                | |  Execute on: All
Nodes
       3 |   1 | 00:00:00.732285 |  28023488 |                | | +---> JOIN HASH
[Semi] [Cost: 21K, Rows: 2M] (PATH ID: 3) Inner (BRO
       3 |   2 |                 |           |                | | |      Join Cond:
(s.store_key = VAL(3))
       3 |   3 |                 |           |                | | |      Materialize
at Input: s.store_key
       3 |   4 |                 |           |                | | |      Materialize
at Output: s.product_key
       3 |   5 |                 |           |                | | |      Execute on:
All Nodes
       4 |   1 | 00:00:00.72732  |  37829248 |                | | | +-- Outer -> JOIN
HASH [Cost: 14K, Rows: 5M] (PATH ID: 4) Inner (B
       4 |   2 |                 |           |                | | | |      Join Cond:
(s.product_key = p.product_key) AND (s.product_v
       4 |   3 |                 |           |                | | | |      Execute
on: All Nodes
       5 |   1 | 00:00:00.591365 |  15884032 |              0 | | | | +-- Outer ->
STORAGE ACCESS for s [Cost: 6K, Rows: 5M] (PATH ID:
       5 |   2 |                 |           |                | | | | |
Projection: store.store_sales_fact_b0
       5 |   3 |                 |           |                | | | | |
Materialize: s.product_key, s.product_version
       5 |   4 |                 |           |                | | | | |      Execute
on: All Nodes
       5 |   5 |                 |           |                | | | | |      Runtime
Filters: (SIP2(HashJoin): s.product_key), (SIP3(H
       6 |   1 | 00:00:00.613828 |  28816384 |              0 | | | | +-- Inner ->
STORAGE ACCESS for p [Cost: 266, Rows: 60K] (PATH I
       6 |   2 |                 |           |                | | | | |
Projection: public.product_dimension_b0
       6 |   3 |                 |           |                | | | | |
```

```
Materialize: p.product_key, p.product_version, p.product_
       6 |   4 |                 |           |                | | | | |      Execute
on: All Nodes
       8 |   1 | 00:00:00.51101  |  32135168 |              0 | | | | | +---> STORAGE
ACCESS for store_dimension [Cost: 34, Rows: 16] (P
       8 |   2 |                 |           |                | | | | |
Projection: store.store_dimension_b0
       8 |   3 |                 |           |                | | | | |
Materialize: store_dimension.store_key
       8 |   4 |                 |           |                | | | | |      Filter:
(store_dimension.store_state = 'MA')
       8 |   5 |                 |           |                | | | | |      Execute
on: All Nodes
(25 rows)
```

If the query plan is too large to analyze, focus on the slowest path and the paths adjacent to that particular path.

```
=> \set path_id 4
=> SELECT
    path_id,
    path_line_index pos,
    running_time,
    memory_allocated_bytes AS mem_alloc,
    read_from_disk_bytes      read_from_disk,
    LEFT(path_line, 70)
FROM
    query_plan_profiles
WHERE
    transaction_id = :t_id
AND statement_id = :s_id
AND path_id BETWEEN :path_id -2 AND :path_id +2
ORDER BY
    path_id,
    path_line_index;
 path_id | pos |  running_time   | mem_alloc | read_from_disk |
           left
---------+-----+-----------------+-----------+----------------+---------------------
--------------------------------------------------
       2 |   1 | 00:00:00.765351 | 746200704 |                | +-GROUPBY HASH (SORT
OUTPUT) (GLOBAL RESEGMENT GROUPS) (LOCAL RESEGMEN
       2 |   2 |                 |           |                | |  Group By:
s.product_key, p.product_description
       2 |   3 |                 |           |                | |  Execute on: All
Nodes
       3 |   1 | 00:00:00.732285 |  28023488 |                | | +---> JOIN HASH
[Semi] [Cost: 21K, Rows: 2M] (PATH ID: 3) Inner (BRO
       3 |   2 |                 |           |                | | |      Join Cond:
(s.store_key = VAL(3))
       3 |   3 |                 |           |                | | |      Materialize
at Input: s.store_key
       3 |   4 |                 |           |                | | |      Materialize
```

```
at Output: s.product_key
       3 |   5 |                  |           |                 | | |      Execute on:
All Nodes
       4 |   1 | 00:00:00.72732  |  37829248 |                 | | | +-- Outer -> JOIN
HASH [Cost: 14K, Rows: 5M] (PATH ID: 4) Inner (B
       4 |   2 |                  |           |                 | | | |      Join Cond:
(s.product_key = p.product_key) AND (s.product_v
       4 |   3 |                  |           |                 | | | |      Execute
on: All Nodes
       5 |   1 | 00:00:00.591365 |  15884032 |               0 | | | | +-- Outer ->
STORAGE ACCESS for s [Cost: 6K, Rows: 5M] (PATH ID:
       5 |   2 |                  |           |                 | | | | |
Projection: store.store_sales_fact_b0
       5 |   3 |                  |           |                 | | | | |
Materialize: s.product_key, s.product_version
       5 |   4 |                  |           |                 | | | | |      Execute
on: All Nodes
       5 |   5 |                  |           |                 | | | | |      Runtime
Filters: (SIP2(HashJoin): s.product_key), (SIP3(H
       6 |   1 | 00:00:00.613828 |  28816384 |               0 | | | | +-- Inner ->
STORAGE ACCESS for p [Cost: 266, Rows: 60K] (PATH I
       6 |   2 |                  |           |                 | | | | |
Projection: public.product_dimension_b0
       6 |   3 |                  |           |                 | | | | |
Materialize: p.product_key, p.product_version, p.product_
       6 |   4 |                  |           |                 | | | | |      Execute
on: All Nodes
(20 rows)
```

### Analyzing the Results

Analyze query execution in parallel with reviewing the query plan so that you can understand the data flow while the query was executing. Pay close attention to the query execution time, the amount of memory used, and the data read from disk to see if anything unusual might be impacting query performance.

## Reviewing Resource Allocation

Each query that executes uses system resources. Multiple queries executing concurrently can compete for system resources. Make sure to allocate resources appropriate to your database workload for good query performance.

### RESOURCE_ALLOCATION System Table

The RESOURCE_ACQUISITIONS system table contains the resources acquired by a given query.

### Example Queries

The following query returns information about the resource requests for a particular transaction_id and statement_Id.

```
=> SELECT
    node_name,
```

```
    request_type,
    pool_name,
    memory_kb,
    filehandles,
    threads,
    succeeded,
    result,
    failing_resource
FROM
    resource_acquisitions
WHERE
    transaction_id = :t_id
AND statement_id = :s_id
ORDER BY
    node_name;
      node_name    | request_type | pool_name | memory_kb | filehandles | threads |
succeeded | result  | failing_resource
-------------------+--------------+-----------+-----------+-------------+---------+--
---------+---------+------------------
v_vmartdb_node0001 | Acquire      | general   |    102400 |           0 |       0 | t
        | Granted |
v_vmartdb_node0001 | Reserve      | general   |   1548488 |          54 |      56 | t
        | Granted |
v_vmartdb_node0002 | Reserve      | general   |   1548488 |          51 |      53 | t
        | Granted |
v_vmartdb_node0003 | Reserve      | general   |   1548488 |          51 |      53 | t
        | Granted |(4 rows)
```

The Acquire request—a request to acquire a specific resource—occurs only in the initiator node. This request indicates the resources used by the Optimizer to plan the query. The default value is 100 MB.

However, if the MEMORY_LIMIT_HIT event appears in the QUERY_EVENTS system table, 100 MB is not enough to plan the query. Your query may be too complicated, or there may be too many projections on the query's tables for the Optimizer to create the best plan before using up the 100 MB.

To resolve this issue, simplify the query, drop any unused and unnecessary projections, or increase the memory available for the Optimizer to plan the query. To increase the available memory, change the MaxOptMemMB configuration parameter:

```
=> ALTER TABLE dbname SET MaxOptMemMB = 150;
```

In the executor nodes, ReserveRequest asks to reserve resources to execute the query. If the reserved memory is not sufficient to execute the query, the AcquireAdditional request type appears. Depending on the operator needs, the AcquireAdditional request may or may not succeed. The succeeded column indicates if Vertica granted the additional resources or not. If the request did not succeed, the results column indicates why, as in the following example:

```
v_vmartdb_node0001 | 45045996273723096 |           1 | AcquireAdditional | t
|                  | Granted
                                                       |    151.03
v_vmartdb_node0001 | 45035996273723096 |           1 | AcquireAdditional | t
|                  | Granted
                                                       |    228.03
v_vmartdb_node0001 | 45035996273723096 |           1 | AcquireAdditional | f
| Memory(KB)       | Request exceeds limits: Memory(KB) Exceeded: Requested =
161480705, Free = 7744915 (Limit = 246861296, Used = 239116381) |    382.03
v_vmartdb_node0001 | 45035996273723096 |           1 | AcquireAdditional | f
| Memory (KB)      | Request exceeds limits: Memory(KB) Exceeded: Requested =
161480705, Free = 7744915 (Limit = 246861296, Used = 239116381) |    382.03
```

RequestAdditional requests occur most often in hash joins or GROUP BY HASH.

#### Analyzing the Results

If the query requests more resources than Vertica has reserved for that query, perform further analysis to find out why the Optimizer did not plan the query properly. This problem may be an issue with table statistics. If the statistics are based on a 10% sample, run ANALYZE_ HISTOGRAM to increase the statistics sample and improve the accuracy of the statistics.

### Projections Used by Query

To understand better why the Optimizer created the plan it did, you need to understand the projections used and their characteristics.

#### PROJECTIONS_USAGE System Table

The PROJECTIONS_USAGE table contains information about projections that Vertica used for each executed query

#### Example Queries

The following query returns the projections used for a specific query. Usually, Vertica uses projections with the same offset (b0, b1). When a node is down, you may see a mixture of projections with different offsets.

```
=> SELECT
    node_name,
    projection_name,
    anchor_table_name
FROM
    projection_usage
WHERE
    transaction_id = :t_id
AND statement_id = :s_id;
       node_name    |   projection_name    | anchor_table_name
--------------------+----------------------+-------------------
 v_vmartdb_node0001 | product_dimension_b0 | product_dimension
 v_vmartdb_node0001 | store_dimension_b0   | store_dimension
 v_vmartdb_node0001 | store_sales_fact_b0  | store_sales_fact
(3 rows)
```

Using the preceding query, you can save the table and projection information in variables.

Defining variables with the projection name allows you to reuse the queries without having to modify and enter the same information many times.

```
\set t_proj '''store_sales_fact_b0'''
\set t_name '''store_sales_fact'''
\set t_schema '''store'''
```

#### Analyzing the Results

Normally, a Vertica query uses only the same offset projection, for example, product_dimension_b0. If you see a projection accessing different offset projections like product_dimension_b0 and product_dimension_b1, the table join may affect query performance.

### Reviewing Table Partitioning

Partitioning is a table property. If a table is partitioned, all the projections of that table are partitioned using the same expression. Defining a partition is part of the logical database design and must be defined by the user. The Vertica Database Designer (DBD) does not recommend partition expressions. Partitions can improve parallelism during query execution.

For best results, partition fact tables because:

- If there are predicates on the partition key, scan time is improved because Vertica can easily find the ROS containers that match the predicates. If the ROS containers of partitioned tables are not needed, Vertica can eliminate the containers from being processed during query execution. This process is called partition pruning. To eliminate ROS containers, Vertica compares query predicates to partition-related metadata.
- The data from different partitions are stored in separate files on disk, improving parallel execution.
- Having data partitioned helps avoid deletes during data loads. (For more information, see Best Practices for Deleting Data.)
- For fact tables, using partitions is the most efficient way to remove old data.

#### TABLES System Table

The TABLES system table contains information about all tables in the database, including if the table is partitioned, and if so, how it is partitioned.

#### Example Query

Query the TABLES system table to see if the table is partitioned and to identify the partition expression. The partition expression defines how to sort the partition.

```
=> SELECT
    table_schema,
    table_name,
    owner_name,
```

```
    partition_expression,
    create_time
FROM
    tables
WHERE
    table_name = :t_name
AND table_schema = :t_schema;
 table_schema |    table_name    | owner_name | partition_expression         |
create_time
--------------+------------------+------------+------------------------------+-------
---------------------------------------
 store        | store_sales_fact | dbadmin    | store_sales_fact.store_key   |  2016-
08-02 17:48:43.60541+00
(1 row)
```

#### Analyzing the Results

If your table is large or a fact table, partition it so that Vertica can organize the data on disk to the benefit of query performance. Vertica reads just the partition that it needs.

Because the data is distributed among separated files, partitioning can increase parallelism.

Partitioning can also help performance when removing data from the system.

For more details about partitioning, see Using Table Partitions in the Vertica documentation.

### Reviewing Projections

When evaluating query performance, it's important to review the projections that the query is accessing. Find out if the projection:

- Is up to date
- Has segments
- Is a pre-join projection
- Is a superprojection
- Segmentation expression
- Is segmented by fields that distribute data evenly and reduce segmentation in queries
- Is a live aggregate projection

#### PROJECTIONS System Table

The PROJECTIONS system table contains important information about your projections. The created_epoch column in the PROJECTIONS system table tells you in what epoch the projection was created.

#### Example Query

Use the table schema and name to get information about all of that table's projections:

```
=> \x
```

```
=> SELECT
    projection_schema,
    projection_name,
    is_prejoin,
    is_up_to_date,
    has_statistics,
    is_segmented,
    segment_expression,
    is_super_projection,
    created_epoch
FROM
    projections
WHERE
    anchor_table_name = :t_name
AND projection_schema = :t_schema;
-[ RECORD 1 ]-------+-----------------------------------------------------------
---------------------projection_schema   | store
projection_name     | store_sales_fact_b0
is_prejoin          | f
is_up_to_date       | t
has_statistics      | t
is_segmented        | t
segment_expression  | hash(store_sales_fact.date_key, store_sales_fact.product_key,
store_sales_fact.product_version, store_sales_fact.store_key, store_sales_
fact.promotion_key, store_sales_fact.customer_key, store_sales_fact.employee_key,
store_sales_fact.pos_transaction_number, store_sales_fact.sales_quantity, store_
sales_fact.sales_dollar_amount, store_sales_fact.cost_dollar_amount, store_sales_
fact.gross_profit_dollar_amount, store_sales_fact.transaction_time, store_sales_
fact.tender_type, store_sales_fact.transaction_type)
is_super_projection | t
created_epoch       | 12
-[ RECORD 2 ]-------+--------------------------------------------------------
projection_schema   | store
projection_name     | store_sales_fact_b1
is_prejoin          | f
is_up_to_date       | t
has_statistics      | t
is_segmented        | t
segment_expression  | hash(store_sales_fact.date_key, store_sales_fact.product_key,
store_sales_fact.product_version, store_sales_fact.store_key, store_sales_
fact.promotion_key, store_sales_fact.customer_key, store_sales_fact.employee_key,
store_sales_fact.pos_transaction_number, store_sales_fact.sales_quantity, store_
sales_fact.sales_dollar_amount, store_sales_fact.cost_dollar_amount, store_sales_
fact.gross_profit_dollar_amount, store_sales_fact.transaction_time, store_sales_
fact.tender_type, store_sales_fact.transaction_type)
is_super_projection | t
created_epoch       | 12
```

**Analyzing the Results**

When reviewing this projection information, consider the following:

- By default, when creating auto-projections from COPY and INSERT INTO statements, Vertica automatically segments the first 32 columns of data. This task helps avoid data skew.

  However, having 32 segmented columns complicates the hash algorithm and can be CPU intensive, especially if several of the segmented columns are VARCHAR(1000). For these projections, review the default segmentation to see if performance might benefit from a better segmentation for the projection. For more information, see Designing for Segmentation in the Vertica documentation.
- Review the segmentation for the query-specific projections and superprojections to see if improving the segmentation might help performance.
- If a table has statistics, the Optimizer creates a low-cost query plan that chooses the best projections for the query to access. To ensure the best plan, create or update statistics for all tables.

**PROJECTION_COLUMNS System Table**

Query the PROJECTION_COLUMNS system table to check the projection ORDER BY clause. You can then see which columns are part of the ORDER BY clause and in what position. In addition, if an ORDER BY column has statistics, you can find out the types of statistics, what type of data is in the column, and what (if any) encoding type is used.

**Example Query**

The following query shows only the columns that are part of the ORDER BY clause. If you want to see all the table columns, remove the predicate sort_position >= 0:

```
=> SELECT
    projection_name,
    projection_column_name,
    column_position,
    sort_position,
    encoding_type,
    access_rank,
    statistics_type,
    statistics_updated_timestamp
FROM
    projection_columns
WHERE
    sort_position >= 0
AND projection_name = :t_proj
AND table_schema = :t_schema
ORDER BY
    projection_name,
    sort_position;
   projection_name    |    projection_column_name    | column_position | sort_position |
encoding_type | access_rank | statistics_type | statistics_updated_timestamp
---------------------+-----------------------------+-----------------+---------------
+---------------+-------------+-----------------+------------------------------
 store_sales_fact_b0 | employee_key                |               6 |             0 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
```

```
 store_sales_fact_b0 | customer_key              |               5 |              1 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | promotion_key             |               4 |              2 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | store_key                 |               3 |              3 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | product_key               |               1 |              4 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | product_version           |               2 |              5 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | date_key                  |               0 |              6 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | pos_transaction_number    |               7 |              7 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | sales_quantity            |               8 |              8 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | sales_dollar_amount       |               9 |              9 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | cost_dollar_amount        |              10 |             10 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | gross_profit_dollar_amount |             11 |             11 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | transaction_type          |              12 |             12 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | transaction_time          |              13 |             13 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
 store_sales_fact_b0 | tender_type               |              14 |             14 |
AUTO          |           0 | ROWCOUNT        | 2016-02-26 14:44:41.275664+00
(15 rows)
```

**Analyzing the Results**

Encoding reduces the footprint of data stored on disk so that during query execution, there are fewer bytes to read from disk. There are several encoding algorithms, described in Encoding-Type in the Vertica documentation.

Database Designer (DBD) uses a 1% sample of your data to define the best encoding for the columns of each projection. The best encoding is based on field type and cardinality. If no encoding is defined, Vertica specifies AUTO in the projection DDL. This setting specifies to use the best encoding for the data type, without considering the cardinality.

The following table lists the best and default encodings for Vertica data types.

| Data Type + ENCODING hint | INTEGER, NUMERIC (<=18), DATE, TIMESTAMP, etc. | NUMERIC (19+) | FLOAT | CHAR/VARCHAR | BOOLEAN | Notes |
|---|---|---|---|---|---|---|
| <default> aka AUTO | Delta Int Pack | LZO | LZO | String_LZO | LZO | Use if nothing else below applies. |
| NONE | Delta Int Pack | LZO | LZO | LZO | LZO | Don't use. |
| RLE | RLE+LZO | RLE+LZO | RLE+LZO | String RLE+LZO | RLE+LZO | Use when the column is sorted and the number repeated records exceeds 10 (on average). |
| BLOCK_DICT | Block Dict | Block Dict | Block Dict | Block Dict | Block Dict | Use when there are few distinct values per block. |
| BLOCKDICT_ COMP | Block Dict Comp | Block Dict Comp | Block Dict Comp | Block Dict Comp | Block Dict Comp | Use when there are few distinct values per block and skew is high. |
| DELTAVAL | Block Delta Val | LZO | LZO | LZO | LZO | Use when integers are in a narrow range. |
| GCDDELTA | Block GCD Delta Val | LZO | LZO | LZO | LZO | Use when integers are a multiple of a common factor. |
| COMMONDELTA_ COMP | Common Delta | LZO | Common Delta | LZO | LZO | Use if the number of deltas per block is less than the range of values in the block, and less than the number of distinct values in the block. |
| DELTARANGE_ COMP | Delta Range | LZO | Delta Range | LZO | LZO | Use for floating point/integer data that has many distinct values per block. |
| DELTARANGE_ COMP_SP | Delta Range | LZO | Delta Range | LZO | LZO | Use with single-precision floating point data. |

To encode and materialize columns, Vertica needs to apply the encoding algorithm, which uses CPU cycles. So if your queries or loads are CPU bound, removing the encoding may help performance.

It is important that you identify whether your query is CPU bound or I/O bound. If the query is CPU bound, you may be using incorrect encoding.

**PROJECTIONS_STORAGE System Table**

Query the PROJECTIONS_STORAGE system table to see how large the projections are on each node.

```
=> SELECT
    node_name,
    projection_schema,
    projection_name,
    SUM(row_count)                                    row_count,
    ROUND(SUM(used_bytes)/1024^2::NUMERIC(10,3),3) used_GB,
    COUNT(DISTINCT node_name)                         num_nodes,
    SUM(ros_count)                                    ros_count
FROM
    projection_storage
WHERE
    node_name IN
    (
        SELECT
            node_name
        FROM
            nodes
        WHERE
            is_ephemeral = 'f' )
AND projection_name = :t_proj
AND projection_schema = :t_schema
GROUP BY
    node_name,
    projection_schema,
    projection_name
ORDER BY
    2,3,1;
     node_name      | projection_schema |   projection_name   | row_count | used_GB |
num_nodes | ros_count
--------------------+-------------------+---------------------+-----------+---------
+-----------+-----------
 v_test_db_node0001 | store             | store_sales_fact_b0 |   1250032 |  29.586 |
       1 |         1
 v_test_db_node0002 | store             | store_sales_fact_b0 |   1250571 |  29.583 |
       1 |         1
 v_test_db_node0003 | store             | store_sales_fact_b0 |   1250344 |  29.594 |
       1 |         1
 v_test_db_node0004 | store             | store_sales_fact_b0 |   1249053 |  29.565 |
       1 |         1
(4 rows)
```

**Analyzing the Results**

If you see that one node has more data than other nodes, looking at the row_count column, that means your data is skewed. Check your projection segmentation.

If you see that one node has a much higher ros_count, that can indicate that the Tuple Mover is not working properly on that node.

Vertica is as fast as the slowest node in the cluster. If one node has an issue such as a larger amount of data, this table provides that information.

## Recommendations in this Document

Each query performance use case is different, so use this document as a reference. To get the most out of these examples, use variables as suggested in this document, and cut and paste your specific queries in the statements outlined here.

If you have other useful queries that help you with performance tuning, we'd love to hear about them. Add them to the Comments section of thtis document so that the Vertica user community can add to their performance-tuning experti se.

## For More Information

- Vertica Community Edition
- Vertica Documentation
- Vertica User Community
- Query Plans
- Query Plans in vsql

System Tables for Query Performance