HEVO

A COMPLETE GUIDE TO

REDSHIFT QUERY OPTIMIZATION

Table of Contents

Redshift - A New Era of Data Warehousing

Amazon Redshift is a fully managed, cloud-based, petabyte-scale data warehouse service by Amazon Web Services (AWS). It is an efficient solution to collect and store all your data and enables you to analyze it using various business intelligence tools to acquire new insights for your business and customers.

Redshift has emerged as the undisputed leader in the Data warehousing segment, making it an inevitable component of any robust Data Analytics stack. Redshift's massively parallel columnar data store that can be brought to life in minutes, it's ease of use and speed of computing have been the core reasons for its vast adoption.

Whether you are evaluating using Redshift, or have already set it up in your system, this ebook can be your guiding light discover levers that enable you to optimize the Redshift queries. Designed by our internal Redshift experts, this ebook will come handy in tuning your Redshift queries for performance and efficiency, in turn delivering the best results for your spend.

Since its inception, Hevo has helped businesses automate their data migration processes from any source to Redshift. This ebook is a compilation of our practical tips, real-world examples, and best practices for optimizing Redshift Queries.

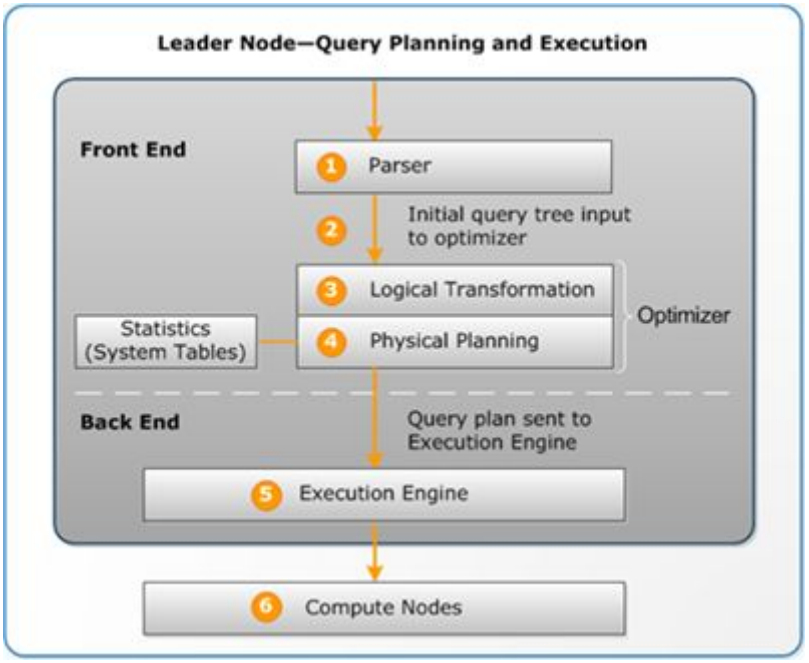
The ebook is divided into 5 broad sections.

- Introduction to Query Planning and Execution
- Query Plan
- EXPLAIN Operators
- Data Redistribution
- Factors Affecting Query Performance
- Query Warm-Up

Each section further deep dives into individual features and elements that you can tune to optimize queries on Redshift. Relevant code snippets are also included for your benefit.

Introduction to Query Planning and Execution

To troubleshoot queries and optimise them, it is important to understand how Redshift plans Query Execution and how it actually execute them.



As mentioned in the above diagram the Query Planning and Execution can be divided into multiple steps. Let's look at what each of these steps do:

Parser

The Parser receives the SQL query and parses it to create a logical tree representative of the initial original query.

Logical Transformation

Logical Transformation is carried out by the Optimizer. Optimizer evaluates the query and rewrites it if necessary to improve efficiency. Optimizer may also create multiple queries out of the original query.

Physical Planning

During this step the Optimizer generates a query execution plan (or multiple plans if the previous step produced multiple queries) to execute query in the most efficient manner. We will discuss Query plans in a greater detail later in this ebook.

Execution Engine

The execution engine take the execution plan and generates query steps and segments. It also generates compiled code to be run by individual compute node slices for each of the query step and segment. The compiled code is then broadcasted to the compute nodes for execution.

Compute Nodes

Compute Nodes execute the compiled code in parallel to generate results. During this process, intermediate query results from various steps are transferred across nodes to be used by the next steps.

Query Plan

The query plan is the most important aspect of Query execution process. Therefore, most of the Query Optimisation activity revolves around evaluating and optimising query plans. A query plan gives you the information on the individual operations executed to execute the query.

To generate a Query Plan you can use **EXPLAIN** command. EXPLAIN command doesn't run the query, it only shows the plan the Redshift would use if you run the query. If you change the structure of the tables involved in a query or run VACUUM or run ANALYSE on the tables the Query Plan may change.

Below is an example of the EXPLAIN output for a simple group by query on event table.

```
explain select eventname, count(*) from event group by eventname;

QUERY PLAN
------------------------------------------------------------------
XN HashAggregate  (cost=131.97..133.41 rows=576 width=17)
  ->  XN Seq Scan on event  (cost=0.00..87.98 rows=8798 width=17)
```

EXPLAIN displays the following information about each operation:

Cost

Cost of the relative value for comparing each operation within a plan. Cost consists of two values separated by two periods. In the above example `131.97..133.41` the first value 131.97 represents the cost of generating the first result of the operation and value 133.41 represents the cost of completing the operation. The costs are cumulative as you read up the plan. Hence, the cost of operation `HashAggregate` includes the cost of operation `Seq Scan on event`.

Rows

Rows represent the estimated number of rows returned by an operation. This estimate is based on the table statistics generated by ANALYZE command and can be less reliable if ANALYSE hasn’t been run recently.

Width

Width represents the estimated size of an average row returned by an operation.

EXPLAIN Operators

This section explains various operators seen in the EXPLAIN output and their impact on the query.

Seq Scan

A Sequential Scan (Seq Scan) operator indicates a sequential scan on the table. Seq Scan scans each row of the table from the beginning till the end to evaluate constraints in the query. For very large tables, this can be an extremely heavy operation resulting in huge I/O cost.

Join

Redshift selects between various Join operators based on the structure of the table, the location of data to be joined and the requirements specific to the query. Different types of Join operators used by Redshift are:

- **Nested Loop -** It is the least optimal Join operator mainly used for cross joins and inequality joins.
- **Hash Join and Hash -** Hash Join and Hash operator are used for executing inner and left/right outer joins. This operators is used when joining tables don't have the joining columns as both distribution and sort keys. The hash operator created a hash of the inner table, the hash join operator reads the outer table, hashed the joining column and joins it with the hash generated for the inner table. This operator is more optimal than Nested Loop.
- **Merge Join -** Merge Join operator is used for inner and outer joins when joining tables have the joining columns as both distribution and sort keys and when unsorted rows in joining tables in less than 20%. Merge Join is the fastest Join operator.

Aggregate

Aggregate operators are used in queries using aggregate functions and Group By operations:

- **Aggregate -** Used for scalar aggregate functions like AVG, SUM, etc.
- **HashAggregate -** Used for unsorted grouped aggregate functions.
- **GroupAggregate -** Used for sorted grouped aggregate functions.

Sort

Sort operators are used when queries have sort or merge results:

- **Sort -** Operator for ORDER BY clauses and other sort operations required for joins, SELECT DISTINCT and window functions.
- **Merge -** Operator used when merging intermediate sort results produced by parallel execution steps.

Data Redistribution

The query plan generated by EXPLAIN command also specifies the way in which data will be moved around the cluster for joins queries. Data movement can be of two kinds:

- **Broadcast -** The values from one side of the join are copied to every compute node in the cluster.
- **Redistribution -** The data values are sent from one slice to another slice based to match the distribution key of other joining table.

The following attributes in the query plan determine how data will be moved in the cluster:

- **DS_BCAST_INNER -** A copy of entire inner table is broadcast to each compute node.
- **DS_DIST_ALL_NONE -** Joining tables have already been distributed to each node using DISTSTYLE ALL, hence redistribution is not required
- **DS_DIST_NONE -** No tables are redistributed as system is able to join corresponding slices without moving data.
- **DS_DIST_INNER -** Inner table is redistributed
- **DS_DIST_OUTER -** Outer table is redistributed
- **DS_DIST_ALL_INNER -** Outer table uses DISTSTYLE ALL, hence the entire inner table is copied to a single slice
- **DS_DIST_BOTH -** Both tables are redistributed

Factors Affecting Query Performance

Number of Nodes

More nodes provide more resources in terms of CPU and memory enabling your queries to execute faster. You need to have the right number of nodes given the query workload on the cluster and amount of data being queried.

Storage Type

Redshift Nodes come in two Storage Types - HDD and SSD. If you have a storage intensive cluster you may use HDD. Whereas, for a compute intensive cluster SSD will be better.

Data Distribution

Redshift distributes the data across node slices according to the DISTSTYLE of each table. The query execution engine redistributes the data across nodes for performing joins and aggregations. Choosing the right DISTSTYLE can reduce the amount of resources consumed in redistribution phase by locating data where it needs to be before performing a join.

Data Sort Order

Redshift sorts data in tables according to the sort key(s) assigned to each table. The query optimizer uses that information to reduce the number of blocks scanned for WHERE constraints in the query.

Size of DataSet

The data not queried in tables should be archived from Redshift. For example, if you are not querying data for transactions older than a year, then the data older than a year must be archived so that Redshift doesn't spend resources scanning and redistributing that data

Concurrent Queries

Redshift has a limited number of query slots used to execute queries. Having too many concurrent queries may reduce performance of the cluster. Implementing workflow management can improve performance for critical queries.

Query Structure

The manner in which query is written has a huge impact on the performance. You should try to write query to return the minimum data required for analysis.

Query Warm-Up

The first time a query runs on a cluster, the compiled code is generated for that query and stored in a LRU cache. Subsequent runs of the same query (even with different parameters) uses the earlier compiled code thereby, omitting the overhead of interpreting and parsing the query. Hence, it may be useful to have Warm-Up runs for critical queries.

Conclusion

Implementing the best practices described in this e-book will ensure a speedy and efficient Querying on Redshift Data Warehouse and in turn do justice to your analytics projects. This will also ensure that your business users do not have to wait to answer their most important business questions.

We, at **Hevo**, simplify the complex process of unifying data from multiple sources through our Data Integration platform. We can help you bring data from various sources to Redshift in real time in a hassle - free fashion. Reach out to us on hello@hevodata.com or sign up for a 14 day free trial here.