What is Amazon Redshift & Spectrum in AWS?

CLOUD COMPUTING DATA ENGINEERING INTERMEDIATE RESOURCE

Introduction

Amazon Redshift is a data warehouse service in the cloud. While working on Redshift, we need to understand various aspects of Redshift such as cluster architecture, table design, data loading, and query performance tips, etc. We have extensive documentation already available to cover these aspects. However, in this article, we have tried to summarize these aspects in one place so that individuals can develop an overall understanding before starting to deep dive into each aspect individually.

This article was published as a part of the Data Science Blogathon

Table of contents

What is Amazon Redshift?

Amazon Redshift is a fully managed, petabyte-scale data warehousing service provided by AWS. It analyzes large datasets and delivers fast query performance for business intelligence, data analytics, and reporting. Redshift uses columnar storage and parallel query execution to efficiently process and analyze data, making it well-suited for data warehousing and ad hoc querying. It offers scalability, high availability, and integration with other AWS services, making it a popular choice for organizations handling large volumes of data.

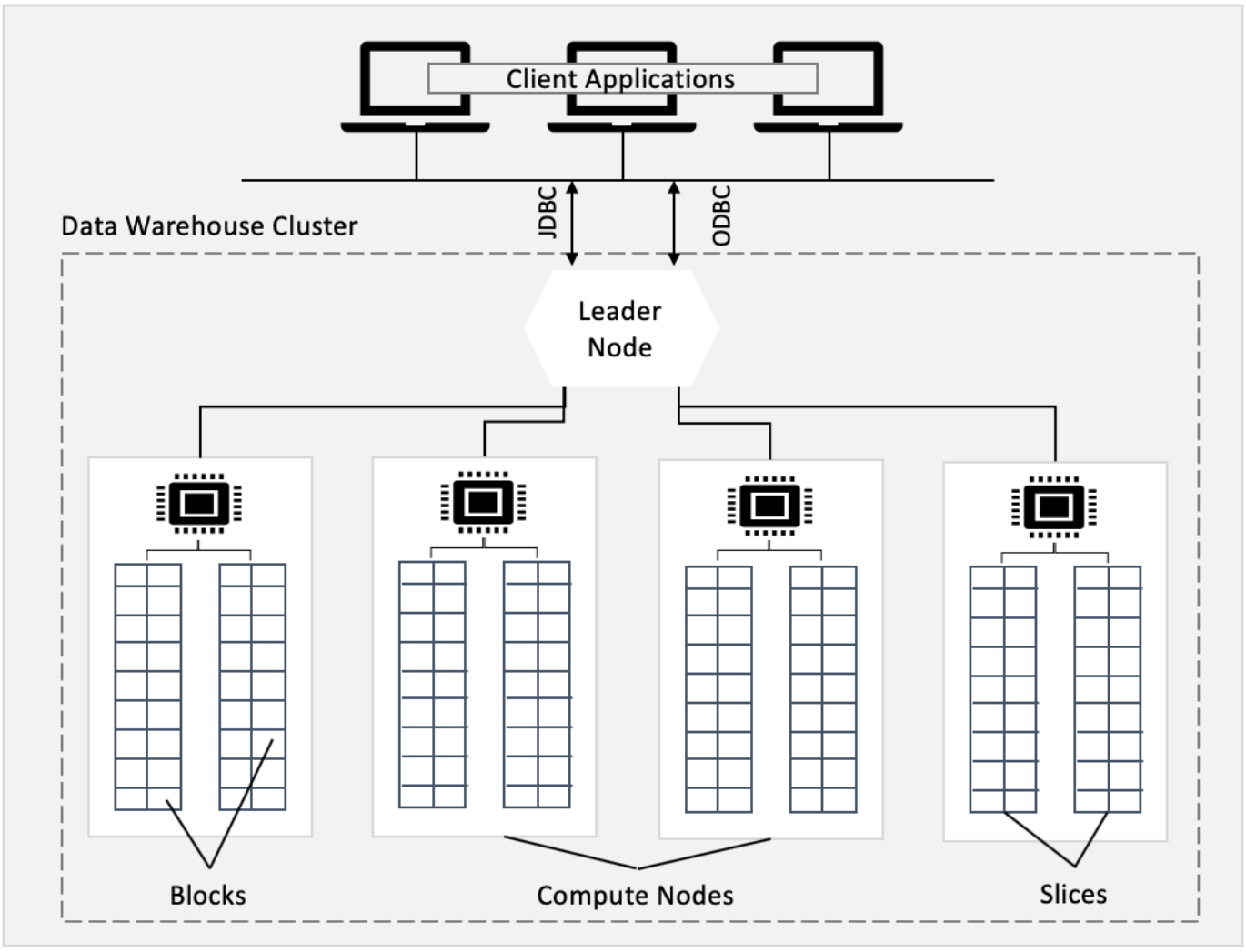
Understanding Cluster

Redshift cluster is a collection of computing resources called nodes. Each cluster runs an Amazon Redshift engine and contains one or more databases.

Leader and Compute Nodes in the Cluster

The cluster nodes are divided into leader nodes and compute nodes. The compute node is further partitioned into slices. Each slice is allocated a portion of compute node's memory and disk space. When a table is loaded with data, the rows are distributed to node slices.

In the case of more than one compute resource, a leader node is also created for coordination among compute nodes and external communications. The compute node is transparent to end-users.



AWS Redshift Cluster Example

Connecting to a Cluster for Query Execution

To execute a query on the cluster, client applications using industry-standard JDBC and ODBC drivers for PostgreSQL (Redshift is based on changes made to an older version of PostgreSQL 8.0.2).

The leader node is responsible for client communication. If the submitted query only references catalog tables (tables with a PG prefix, such as PG_TABLE_DEF) or does not references any tables, the query is

executed on leader nodes. Also, there are leader node-only functions that are only executed on the leader node.

However, if the submitted query references user-created tables or system tables (tables with an STL or STV prefix) and system views (views with an SVL or SVV prefix), the leader node distributes the query to compute nodes.

The intermediate results returned by compute nodes are finally aggregated by the leader node before returning to client applications.

Cluster Nodes

Now, as we understand Redshift cluster basics, let's understand node type, node size, and node count?

Supported Node Types

Amazon Redshift offers different node types based on different workload requirements such as performance, data size, future data growth, etc. Node types are divided into previous and current generation node types.

It is recommended to select node types from current generation node types. The current generation node type is grouped into DS2, DC2, and RA3. Each of these node types offers 2 node sizes These node sizes are based on the size of CPU, memory, and storage capacity. DC2 offers large and 8xlarge node types.

DS2 offers large and 8xlarge node types whereas RA3 offers 4xlarge and 16xlarge.

Deciding on a Node Type

DC2 is recommended for applications with low latency and high throughput requirements and with relatively fewer amounts of data. DS2 usage HDD for storing data which makes it less performant than DC2 and RA3.

DS2 is recommended for workloads with a relatively high amount of known data size, not expected to grow rapidly, and where performance is not the primary objective. RA3 is recommended if the data is expected to grow rapidly or if the flexibility to scale and pay for computing separate from storage is required.

Once node type is selected based on application requirements, it is required to select between node sizes. Unless there is a requirement for more resources, start with a low node size as a high node size for each node type is more expensive. Mixing of a node type and node size in a cluster is not supported.

The number of nodes in a cluster is determined by data volume as well as computing resource requirements. When any query is submitted to the leader node, it is passed to each slice for execution on the data present in that slice. There are benefits to distributing data across many slices. Once a cluster is created, the node count may be altered by cluster resize.

Resizing an Existing Cluster

Redshift supports two resize approaches, classic and elastic. Both approaches support node type changes, number of nodes changes, or both. However, elastic resize has certain limitations.

These limitations include limitations on the number of nodes after resizing. For example, dc2.large or ds2.xlarge node types, only double the size or half the size of the number of nodes of the original cluster are supported. Whereas classic resize does not have limitations on the number of nodes after resizing.

One more important difference between these two approaches is the number of slices after resizing. If the resizing is to be performed for performance improvement, classic resize is suggested as the total no of slices on all nodes in the cluster remains the same after elastic resizing.

Understanding Table Design

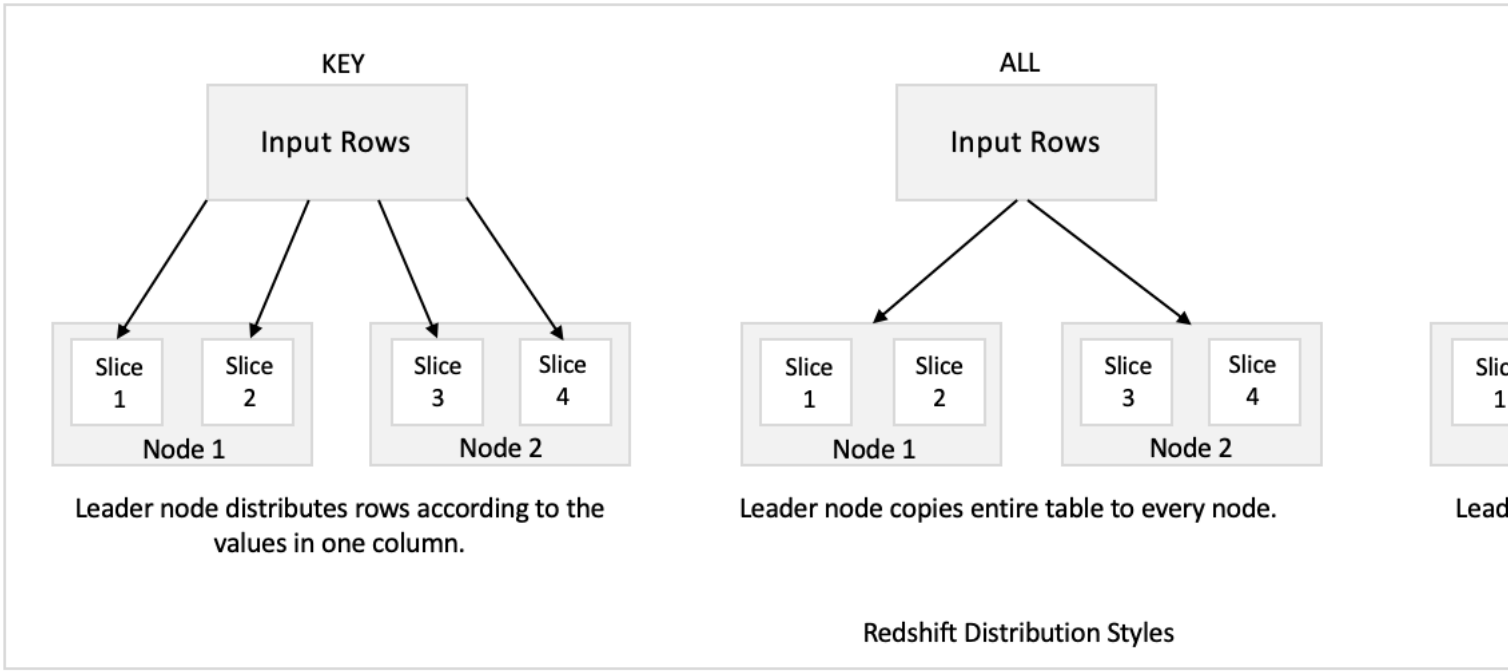
Now, we move to table design in Redshift. We will discuss three important topics here, distribution keys, column encodings, and Sort key.

Distribution Keys

When data is loaded into table, Redshift usage distribution style to determine the node slice where any row is assigned to. The four distribution styles supported by Redshift are AUTO, EVEN, KEY, and ALL. For brevity, we will not explore all four styles here.

However, for performance, the suggested one is the KEY distribution style where a column is defined as a distribution key. While loading the data, Redshift distributes rows to node slices based on values in one column. Rows with the same column value are placed on the same slice.

Data distribution has two primary goals, uniform workload distribution by minimizing data skewness and minimize data movement during query execution by collocating joining rows from tables. Both goals add to minimizing query time.



Redshift Distribution Styles

Column Encodings

Redshift is a columnar data store which means the same column from multiple rows is packed together to create a block. For Redshift, the block size is 1 MB. Each slice contains multiple blocks. Columnar storage enables specifying different compression encodings suitable for each column data type separately.

Compression encodings help in saving space for storing data on a disk. This further reduces the I/O required for loading data in tables as well as for other operations that are part of query execution. This helps in minimizing query time.

SSN	Name	City	State
56782332	Nor Norman	Phoenix	Arizona
72659645	Franke Jesse	Chicago	Illinois
12738852	Tia Zed	Los Angeles	California

893246759	557865423	867789032	56782332	72659645	12738852	332348798	165423451	232144334	767788909

Columnar - Block 1

Sort Keys

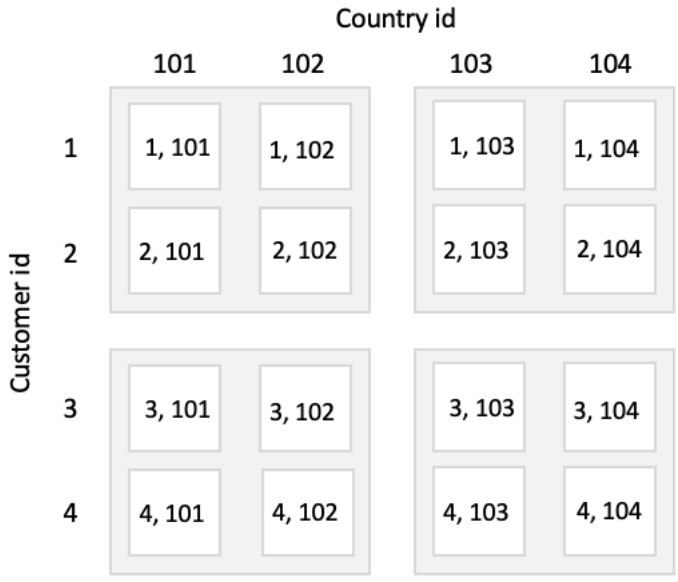
Redshift maintains in-memory metadata, called zone maps, to maintain minimum and maximum values for each 1 MB block. Zone maps are used during query processing to find relevant blocks before disk scan. If the column is not sorted on disk, the min-max range can overlap in the zone map for different blocks.

This overlap can reduce the impact of zone map effectiveness. Redshift usage sort keys to store data on disk in sorted order. Supported sort key types are compound and interleaved. Both types support specifying multiple columns as the sort key.

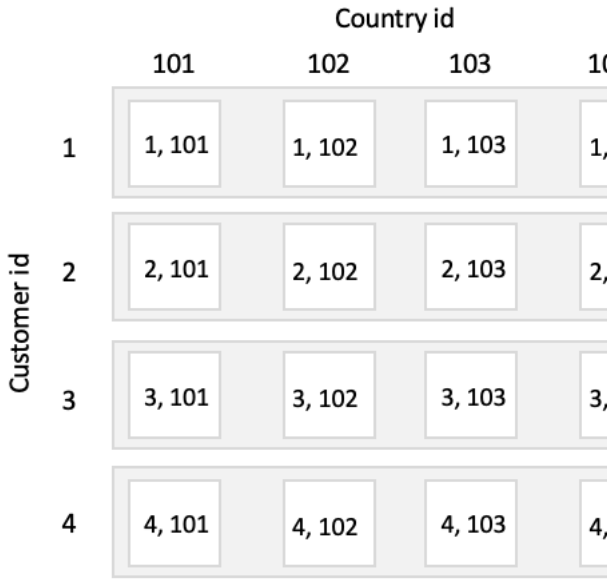
The difference lies in the way each sort type assigns weight to columns during the sort. Redshift sorts rows in the order columns are listed in the compound sort key.

The effectiveness introduced by using compound sort keys reduces with column order. Therefore, the most frequently used column is suggested to use as the first one followed by the second most frequent and so on.

For interleaved sort keys, each listed key is assigned an equal value. The compound key is suggested if the query includes JOINS, PARTITION BY, GROUP BY and ORDER BY. The interleaved key is suggested for highly selective queries which include one or more sort key columns in WHERE clause, for example, *select e_name from employee where e_state = 'CA'*.



Interleaved sort key on customer id and country id



Compound sort key on customer id and country id

Usage by Client Applications

Let’s now discuss data loading and query in the Redshift cluster.

Available Data Load Options

Redshift allows loading data in multiple ways. Let’s start with the COPY command which is an efficient way to load a large amount of data in parallel from Amazon S3, Amazon EMR, Amazon DynamoDB, and multiple other data sources on the remote host.

The single COPY command can load data in parallel from multiple files. For optimal loading in parallel, the suggested file size is between 1 MB and 125 MB after compression. While loading data, smaller file sizes help in the better division of workload among the nodes in the cluster.

Other ways to load data in a table includes single inserts, multi-row inserts, bulk insert and using staging table to perform a merge. If the COPY command cannot be used, using a multi-row insert is preferred over multiple single inserts. Bulk insert is used with SELECT clause along with to INSERT or CREATE TABLE AS command to move data or a subset of data from one table to another table.

Redshift does not support merge or upsert as a single command. However, a merge operation can be performed in a single transaction by using a staging table.

Using Spectrum Instead of Redshift

However, there can be scenarios where loading all data in Redshift may not be required. Data used rarely by client applications, we categorized as cold data. Application teams can save costs by maintaining them in s3 only instead of loading them in Redshift tables as Redshift is about 4x costly than s3.

Redshift provides a way to use these cold data by means of Spectrum. The structure of s3 files can be defined and registered as tables in an external data catalog, such as AWS Glue or Apache Hive metastore. External tables can be created either by DDL commands in Amazon Redshift or by using some tool that connects to an external data catalog.

Once the external tables are defined, applications can query external tables using JDBC or ODBC connections. The external tables can be joined with Redshift tables that reside on the cluster’s local disk. As the data is queried from the s3 location and moved to the cluster during query processing, queries on Spectrum tables do run slower than Redshift.

Querying Small and Large Volume Data

Once data is loaded in tables, applications can use JDBC or ODBC for connecting to cluster and query tables data. In the Redshift cluster, the leader node is responsible for returning query results to client applications.

However, for queries with large data volume, the leader node can become a bottleneck as it can hold the connection till the time all data is returned. This impacts query performance for other concurrent requests.

For these scenarios, it is suggested to unload data first to s3 before the application can use it. To unload data from a table or from a query output, UNLOAD command can be used. Data can be unloaded in delimited format or fixed-width format. The data can be unloaded serially or parallelly, in one file or in multiple files.

Solving for Query Slowness

In case of any query slowness, application teams can query system tables (tables with an STL or STV prefix) and system views (views with an SVL or SVV prefix) to understand information about system functioning. Application teams can explore different queries' runtime behavior using these tables and categorize them into different workload groups.

To separate the impact of different workloads on each other, query queues can be created in workload management (WLM). Each queue is allocated a portion of the cluster's available memory. If the cluster is running multiple short queries as well as long-running queries, the short queries may have to wait longer due to long-running queries. A separate queue can be created for such short queries. Or to automatically prioritize shorter queries over longer queries by WLM without a separate queue, short-query acceleration can be enabled.

Conclusion

There are numerous optimization tips that are available for query level optimization which we are not discussing here as this may extend the scope of this article.

However, it is advised to understand the optimization tips for Redshift as well as Spectrum queries before using them in production.

The media shown in this article on Sign Language Recognition are not owned by Analytics Vidhya and are used at the Author's discretion.

Article Url - https://www.analyticsvidhya.com/blog/2021/06/understand-all-about-amazon-redshift/

Mayank Rijhwani