AWS Redshift:

Amazon Redshift is a **cloud-based data warehousing service** offered by AWS (Amazon Web Services). It is designed to allow businesses to efficiently store and analyze large volumes of data. It is optimized for running complex queries and analytical workloads across petabytes of structured and semi-structured data.

An **Amazon Redshift cluster** consists of several key components that work together to deliver scalable and high-performance data warehousing and analytics capabilities. Below are the main components of a Redshift cluster:

**1. Leader Node**

* **Role**:  
  The leader node acts as the **central management point** for the cluster. It coordinates the execution of queries, compiles query plans, and distributes tasks to the compute nodes.
* **Responsibilities**:
  + Handles communication with client applications.
  + Parses SQL queries and generates execution plans.
  + Aggregates and returns the final query results to the client.
* **Note**: The leader node does not store user data.

**2. Compute Nodes**

* **Role**:  
  Compute nodes are responsible for **storing data** and **processing queries**.
* **Responsibilities**:
  + Execute the tasks assigned by the leader node.
  + Perform data storage, retrieval, and processing for analytical queries.
* **Details**:
  + Each compute node is divided into **slices**, which handle a portion of the node's workload.
  + Nodes communicate with each other to perform distributed processing.

**3. Node Slices**

* **Role**:  
  A slice is a **logical partition** of a compute node, used to distribute data and workload.
* **Responsibilities**:
  + Each slice is assigned a portion of the node's CPU, memory, and storage.
  + Ensures parallel execution of queries across slices for improved performance.
* **Details**:
  + The number of slices per node depends on the node type.

**4. Cluster Configuration**

* Defines the characteristics of the Redshift cluster.
  + **Node Types**:
    - **Dense Compute (DC)**: Optimized for high performance with SSD storage.
    - **RA3**: Allows separation of compute and storage for greater flexibility and cost efficiency.
  + **Node Count**:
    - Single-node cluster: Combines leader and compute functionality in one node (for smaller use cases).
    - Multi-node cluster: Includes one leader node and two or more compute nodes.

**5. Network and Security Components**

* **Virtual Private Cloud (VPC)**:  
  Controls the network environment of the cluster, including IP ranges and subnet settings.
* **Access Control**:
  + **AWS Identity and Access Management (IAM)** for user and service access control.
  + Security groups and ACLs control access to the cluster at the network level.
* **Encryption**:
  + Data can be encrypted using AWS Key Management Service (KMS) or customer-managed keys.

**6. Storage**

* **Columnar Storage**:  
  Data is stored in a columnar format to optimize analytical queries.
* **RA3 Nodes**:  
  Automatically manage cold data storage by offloading it to **Amazon S3** while keeping frequently accessed data in SSDs.
* **Data Distribution Styles**:  
  Determines how data is distributed across nodes and slices:
  + **Key Distribution**: Based on a specific column value.
  + **Even Distribution**: Evenly distributes rows across slices.
  + **All Distribution**: Replicates data to all nodes (useful for smaller datasets).

**7. Redshift Spectrum**

* **Role**:  
  Allows querying data directly in **Amazon S3** without loading it into the Redshift cluster.
* **Details**:
  + Extends the cluster by leveraging S3 as a data lake.
  + Spectrum workers process the queries separately from the main Redshift nodes.

**8. Maintenance and Monitoring**

* **Amazon CloudWatch**:  
  Monitors the health and performance of the cluster.
* **AWS Management Console/CLI**:  
  Provides tools to manage and scale the cluster.
* **Redshift Advisor**:  
  Offers recommendations to improve query performance and reduce costs.

**Summary of Core Components in a Redshift Cluster:**

| **Component** | **Role** |
| --- | --- |
| **Leader Node** | Manages query execution and communication with clients. |
| **Compute Nodes** | Store and process data. |
| **Node Slices** | Subdivisions of compute nodes for parallel processing. |
| **Storage** | Columnar storage optimized for analytical queries. |
| **Network/Security** | Ensures secure and controlled access to the cluster. |
| **Redshift Spectrum** | Extends querying capabilities to S3 data lakes. |
|  |  |

Node Slices:

In our case we have 3 node cluster, which means will have 3 compute nodes like this. Each node have 2 vCPU, 15 GB RAM and 160 GB storage. And also when we actually review the details about dc2.large, it says there will be two slices on each of the node. These are nothing but this slices. A slice is nothing but combination of CPU, memory and storage. As we will be having two slices, it will get one vCPU close to seven and half GB memory and 80 GB storage. That's how a slice will be created on each node. In our case, we have dc2.large, which means we'll be having two such slices. You can also review the material related to node slices by scrolling down here. You can see here, a compute node is partitioned into slices. In our case it is two on dc2.large. Each slice is allocated a portion of nodes memory and disk space where it processes a portion of the workload assigned to the node. In our case, we have 15 GB memory on each node and 160 GB storage, which means it will be approximately getting seven and half GB memory and 80 GB storage for each slice. So that will be used when we actually create tables as part of Redshift clusters. You can also read through the details here. When you create a table, you can optionally specify one column as a distribution key. When the table is loaded with data, the rows are distributed on the node slices according to the distribution key that is defined for a table. Choosing a good distribution key enables Amazon Redshift to use parallel processing to load the data. We'll go through the details about distribution keys and distribution styles at a later point in time, and then you will understand how the data will be distributed among node slices that are part of each and every node in our Redshift cluster. As we understood some of the key concepts related to node slices and as we understood that data will be distributed as part of the node slices in each node in the cluster. Now let's go through the details related to distribution styles and dist keys as we explore all the distribution styles and dist keys, I will also go back to this diagram and try to explain how the data will be distributed among node slices on multiple nodes in the cluster.

Redshift queries have flavour of POSTGRES, it is built on top of POSTGRES query engine

**Why Doesn't Redshift Enforce Constraints?**

Redshift is designed as a **distributed data warehouse** optimized for analytical queries and performance at scale. Enforcing constraints (like primary keys or foreign keys) would introduce significant overhead due to:

* Distributed architecture and parallel processing.
* Massive data volumes, making integrity checks costly.

By avoiding enforcement, Redshift prioritizes performance for analytical workloads.

However not null constraint is enforced

It doesn’t use any indexes

Copy Data from S3 to Redshift:

We use access key and secret key when running the command from outside AWS environment

When command is called from aws services we can use IAM roles

create table orders(

order\_id int PRIMARY key,

order\_date DATETIME,

order\_customer\_id int,

order\_status varchar(30)

)

COPY orders

FROM 's3://atul.data/retail\_db/orders/part-00000'

CREDENTIALS 'aws\_access\_key\_id=AKIA34AMCWUYLVKXRBGB;aws\_secret\_access\_key=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'

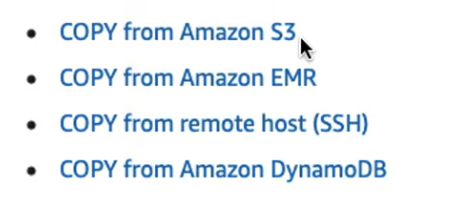
REGION 'ap-south-1'

FORMAT AS csv;

A screenshot of a computer program

Description automatically generated

Copy from Other Datasources

* 

SELECT \* from users limit 10;

create database retail\_db;

use database retail\_db;

drop table orders;

create table orders(

order\_id int PRIMARY key,

order\_date DATETIME,

order\_customer\_id int,

order\_status varchar(30)

)

COPY orders

FROM 's3://atul.data/retail\_db/orders/part-00000'

CREDENTIALS 'aws\_access\_key\_id=AKIA34AMCWUYLVKXRBGB;aws\_secret\_access\_key=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'

REGION 'ap-south-1'

FORMAT AS csv;

create table orders(

order\_id int PRIMARY key,

order\_date DATETIME,

order\_customer\_id int,

order\_status varchar(30)

)

COPY order\_items

FROM 's3://atul.data/retail\_db/order\_items/part-00000'

IAM\_ROLE 'arn:aws:iam::816069129520:role/redshift-s3-full-access'

REGION 'ap-south-1'

FORMAT AS CSV

DELIMITER ',';

SELECT \* from orders limit 10;

CREATE TABLE order\_items (

order\_item\_id INT NOT NULL,

order\_item\_order\_id INT NOT NULL,

order\_item\_product\_id INT NOT NULL,

order\_item\_quantity INT NOT NULL,

order\_item\_subtotal FLOAT NOT NULL,

order\_item\_product\_price FLOAT NOT NULL,

PRIMARY KEY (order\_item\_id)

);

SELECT \* from order\_items limit 10;

The keys in the JSON objects must exactly match the column names in the Redshift table. If there is a mismatch (e.g., missing or additional keys), errors may occur unless you specify other options like MAXERROR.

Connect to Redshift from External World:

We can make Redshift Public by attaching elastic IP to the cluster

**1. Key Distribution (KEY)**

Key distribution distributes the data based on a column (distribution key) that you choose. This helps optimize performance for joins, especially when joining tables on the distribution key column.

* **How it works**:
  + The values in the distribution key column are hashed, and the data with the same hash values is stored on the same slice of the compute node.
  + This is useful when you frequently join tables on this distribution key column because it minimizes data shuffling between nodes.
* **Use case**:
  + Ideal for tables that are frequently joined together on a particular column.
  + For example, if you have a **sales** table and a **customers** table, and you often join these on the customer\_id column, it would be a good idea to set customer\_id as the distribution key.
* **Example**:

sql

Copy code

CREATE TABLE sales (

sale\_id INT,

customer\_id INT,

amount DECIMAL(10, 2),

sale\_date DATE

)

DISTSTYLE KEY

DISTKEY (customer\_id);

**2. Even Distribution (EVEN)**

Even distribution distributes the data across all the slices of your cluster in a round-robin manner, meaning Redshift distributes each row to the next slice, and so on.

* **How it works**:
  + Each slice gets approximately the same number of rows, regardless of the content of the rows.
  + No specific column is used for this distribution, making it ideal when there is no column that is frequently used for joins or queries.
* **Use case**:
  + Good for tables with no obvious join keys or tables that aren't frequently involved in joins with other tables.
  + Ideal for smaller tables or tables with evenly distributed data that doesn't benefit from key-based optimization.
* **Example**:

sql

Copy code

CREATE TABLE products (

product\_id INT,

product\_name VARCHAR(255),

category\_id INT

)

DISTSTYLE EVEN;

**3. All Distribution (ALL)**

All distribution makes a full copy of the entire table on each slice of the cluster. This means that each compute node has a complete copy of the table's data.

* **How it works**:
  + Every slice has a copy of the entire table, which eliminates the need for any data shuffling during queries that involve this table.
  + This is ideal for small tables that are often used in joins.
* **Use case**:
  + Best for **small lookup tables** that are frequently joined with other large tables.
  + For example, dimension tables in a star schema (like **product**, **customer**, or **region** tables) that are referenced by many fact tables.
* **Example**:

sql

Copy code

CREATE TABLE categories (

category\_id INT,

category\_name VARCHAR(255)

)

DISTSTYLE ALL;

**Key Considerations When Choosing a Distribution Style:**

1. **Data Size**:
   * For **small tables** (e.g., dimension tables), use DISTSTYLE ALL.
   * For **large tables**, DISTSTYLE KEY is often ideal, especially if they are frequently joined on a specific column.
   * If no obvious distribution key exists, use DISTSTYLE EVEN.
2. **Joins**:
   * **KEY distribution** helps minimize the need to shuffle data during **joins** by placing rows with the same join key on the same node.
   * If your tables are **joined on different keys** or not joined often, EVEN may be a good choice.
3. **Query Performance**:
   * Using DISTSTYLE KEY for large tables that are frequently joined on a specific column can significantly improve performance.
   * For smaller dimension tables, using DISTSTYLE ALL ensures that the entire table is local to each compute node and avoids network traffic.
4. **Table Schema Design**:
   * It’s common to use DISTSTYLE KEY for fact tables (large, frequently joined tables) and DISTSTYLE ALL for dimension tables (small lookup tables).
   * For tables that are neither frequently joined nor used in joins, DISTSTYLE EVEN may be appropriate.

**Summary of Distribution Styles:**

| **Distribution Style** | **Description** | **Use Case** |
| --- | --- | --- |
| **KEY** | Distributes data based on a specific column (distribution key). | Best for tables that are frequently joined on a specific column. |
| **EVEN** | Distributes data in a round-robin manner, evenly across slices. | Best for tables with no clear distribution key or small tables not frequently joined. |
| **ALL** | Distributes a full copy of the table to every slice. | Best for small dimension or lookup tables. |

**Performance Considerations:**

* **Joins**: Tables with the same distribution key will minimize data shuffling during joins.
* **Data Shuffling**: When tables have different distribution styles or keys, Redshift may need to shuffle data between nodes during query execution, which can affect performance. This is why choosing the correct distribution style is important for optimization.
* **Skew**: If the data is not evenly distributed across slices, it can lead to performance degradation due to skewed processing. The distribution key helps mitigate skew if chosen carefully.

Now when it comes to distribution styles, we have AUTO, then EVEN, then KEY, then ALL, these are the different distribution styles we have. The default distribution style is nothing but AUTO, You will understand what AUTO means in a moment, to understand more about AUTO, you need to understand all about EVEN , KEY and ALL. Let me go through the details about all these distribution styles. First, I'll be focusing on ALL distribution, then I'll be talking about EVEN, then I'll be talking about KEY, then finally I will talk about AUTO. When it comes to ALL, a copy of the entire table is distributed to every node, which means if I create a table with dist style as ALL, the copy will be there on each and every node in the cluster. If you review this architecture diagram, you can see that we have multiple compute nodes. Let's take the example of cluster with three nodes, when we create table with distribution style as ALL , the table will be there on each and every node in the cluster. If the size of the table is 100 MB, the overall storage for the table will be 300 MB because there is a copy of the table on all the nodes in the cluster. Now the next distribution style is nothing but EVEN, the major difference between ALL and EVEN is, as part of ALL, the 100 MB table will be copied on all the three nodes in the cluster. The same 100 MB table will be distributed evenly across all the three nodes in the cluster. As the table get data using statements like insert or copy automatically data will be distributed on all the nodes in the cluster and the distribution will be even. In our case, if you have 100 MB table, each node will get close to 33 MB data and you will see data distributed on all the three nodes in the cluster. Let's go back to the architecture diagram here and you can visualize it from this. Now, when it comes to KEY, it will actually distribute the data based upon the KEY. If the KEY is skewed, then the data will be skewed across all the nodes in the cluster, skew means not distributing the data evenly. If the column is not skewed, then the data will be distributed on all the nodes in the cluster very close to the even. When it comes to KEY distribution, along with dist style key, we also need to specify one of the columns in the table as dist key. You will understand the syntax at a later point in time, but that is the major difference between EVEN and KEY. When it comes to EVEN, data will be automatically distributed in round robin fashion as it is populated into the table. Whereas when it comes to KEY distribution, the data will be distributed on all the nodes in the cluster based upon certain key. Now let's talk about AUTO. AUTO is nothing but start with ALL and change the style depending upon the data that is being loaded into the table, it might change it to EVEN or KEY depending upon how the data is being loaded into the table. **So AUTO is nothing but start with ALL and change the style to whatever is optimal for us. It will be technical automatically for us,**  we don't need to worry too much about getting in there and changing it. You can actually go through the material and understand how it works. So these are the four different distribution styles which we can leverage as we understand the concepts related to distribution style. Now let's go through the details about syntax and semantics of how to specify the distribution style and also how the tables can be created and copy the data into those tables. We will also try to review some of the metadata tables to understand how data is distributed on multiple nodes in the cluster.

A screenshot of a computer

Description automatically generated

A **federated query** in **AWS Redshift** allows you to query and combine data from your Amazon Redshift cluster with data stored in external databases, such as **Amazon RDS**, **Amazon Aurora**, or other supported PostgreSQL and MySQL databases. This enables seamless integration of data across different systems without requiring the data to be ingested into Redshift first.

**Key Features**

1. **Query External Data Sources**:
   * Access data from PostgreSQL or MySQL databases directly without loading it into Redshift.
2. **Combine Data**:
   * Combine data from Redshift tables with data from external sources in a single query.
3. **Performance Optimization**:
   * Redshift uses the pushdown mechanism to delegate as much query processing as possible to the external database, reducing data movement and improving performance.
4. **Secure Access**:
   * Federated queries leverage **AWS IAM roles** for secure authentication and permissions when accessing external data sources.

**How Federated Queries Work**

1. **Create an External Schema**:
   * Use the CREATE EXTERNAL SCHEMA command to define a schema that maps to the external data source.
2. **Query the External Data**:
   * Use SQL to query the external database just like querying a Redshift table.
3. **Data Integration**:
   * Combine results from Redshift and external tables using JOIN, UNION, or similar SQL operations.

**Example Use Case**

Suppose you have:

* **Sales data** stored in Redshift.
* **Customer data** stored in an Amazon Aurora PostgreSQL database.

You can use a federated query to combine the sales and customer data for analytics without moving the customer data to Redshift.

CREATE EXTERNAL SCHEMA apg

FROM POSTGRES

DATABASE 'database-1' SCHEMA 'myschema'

URI 'endpoint to aurora hostname'

IAM\_ROLE 'arn:aws:iam::123456789012:role/Redshift-SecretsManager-RO'

SECRET\_ARN 'arn:aws:secretsmanager:us-west-2:123456789012:secret:federation/test/dataplane-apg-creds-YbVKQw';

**What is Amazon Redshift Spectrum?**

**Amazon Redshift Spectrum is a feature of Amazon Redshift that allows you to directly query data stored in Amazon S3 without having to load it into your Redshift cluster. This enables serverless querying of vast amounts of data in S3, seamlessly combining it with the data stored in your Redshift cluster for integrated analytics.**

**Key Features of Redshift Spectrum**

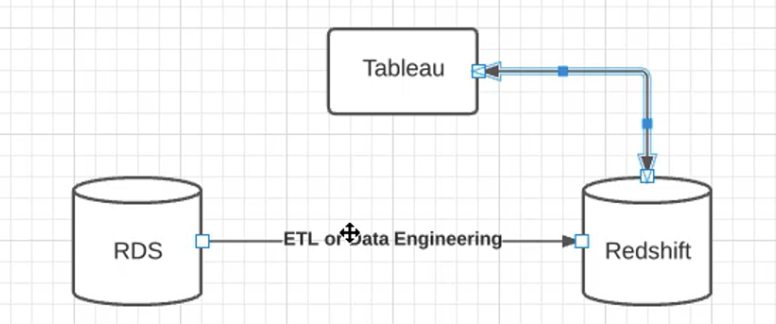
1. **Query S3 Data Directly:**
   * **Redshift Spectrum enables querying open file formats (e.g., Parquet, ORC, Avro, JSON, CSV, etc.) stored in S3.**
2. **Combine S3 and Redshift Data:**
   * **You can join tables in Redshift with external tables in S3 to perform complex analytics across structured and semi-structured data.**
3. **Scalable Processing:**
   * **Redshift Spectrum scales out the query processing to thousands of nodes, allowing you to query exabytes of data efficiently.**
4. **Pay Per Query:**
   * **You are charged only for the amount of data scanned in S3, making it cost-effective.**
5. **Leverages Glue Data Catalog:**
   * **Redshift Spectrum integrates with AWS Glue to manage metadata, allowing you to use the Glue Data Catalog as a central metadata repository.**

**How Redshift Spectrum Works**

1. **Create an External Table:**
   * **Use the CREATE EXTERNAL TABLE command to define a table that maps to data stored in S3.**
2. **Query the External Table:**
   * **Use standard SQL to query the external table as if it were a regular table in Redshift.**
3. **Processing:**
   * **Redshift Spectrum processes queries using a fleet of Spectrum nodes that are separate from your Redshift cluster. Only the results are sent back to the Redshift cluster, reducing the load on your cluster.**

**Use Cases**

1. **Data Lake Analytics:**
   * **Query massive datasets stored in S3 without moving or transforming the data.**
2. **Data Archival and Query:**
   * **Archive historical data in S3 while keeping recent, frequently accessed data in Redshift. Use Spectrum to query archived data when needed.**
3. **Combine Structured and Semi-Structured Data:**
   * **Analyze structured data in Redshift alongside semi-structured data (e.g., JSON or Parquet) stored in S3.**

****

At this time. We are talking about some of the key concepts related to Redshift. They are nothing but federated queries and spectrum. For now, we are primarily focused on federated queries. A federated query is nothing but running query from Redshift by connecting to traditional RDBMS's which are running as part of RDS such as Postgres, MySQL, AWS server, etc,. So when it comes to Federated query, it will look like this. You will have a redshift database or redshift cluster. Then you will have traditional RDBMS's as part of RDS. It can be Postgres, it can be MySQL, or it can be AWS server itself. Then instead of having additional tools, what you can do, you can actually run queries from here to here, by creating external schemas, by providing required information to connect to this. And then we should be able to run queries against the tables that are part of RDS, which is nothing but postgres or MySQL and you should be able to process data using your redshift capacity. Now, before getting into those details, let's understand why we need to do this. So when it comes to RDS, it is nothing but RDBMS type of database. RDBMS stands for Relational Database Management System. RDBMS databases are primarily used for transactional based systems. For example, let's say you want to run E commerce platform where you want to sell some products or subscriptions on regular basis. There might be hundreds or thousands of customers and you want to persist all the transactions that are happening with your customers into some database. That's where RDBMS comes into picture. If yours is a AWS native shop, you will be using RDS for that. As part of RDS. You can pick either Aurora or Postgres or MySQL and you can take it further. All your transactions will be persisted in this. RDBMS databases. Tables will be typically normalized in this case, and it is primarily used for persisting the transactions that are happening with your customers. Now you might want to understand how the business is going on. For that, you might want to run queries against your tables. You can definitely connect your BA tool. For example, let's say this is your BA tool. I am taking Tableau as example. You should be able to connect tableau to this RDS. Let me actually give the bidirectional one. You can connect tableau to RDS or RDBMS and you should be able to generate reports. However, the joins in some cases can be quite expensive. It is not recommended to use BA tools such as Tableau directly to connect to transactional based systems and run reports off of it. Instead of running reports off of RDS or RDBMS instances. We typically build data warehouse. We process data from RDS or any source database into RDS. It is called as ETL or Data Engineering. Whatever data that is being generated here as per your requirements, you will process it and store it in data warehouse's such as redshift. Now, when it comes to BA tools, you connect BA tools to these data warehouse and then you generate reports off of it. Now the database tables that are there as part of the data warehouses will be fine tuned for our reporting requirements, and hence the reports will start performing lot better. Even if the reports are slow, it will not impact the source RDBMS database. It is isolated from these side effects of heavy weight processing that is relevant for these BI type of requirements. That is why we typically have data warehouse as part of redshift and run reports off of Redshift using BA tools such as Tableau. Now either you can develop ETL frameworks or data engineering applications using traditional ETL tools such as Informatica, Abinitio etc. Or you can even build custom tools using Python as programming language. If it is very huge data. Then you can even leverage Spark and you can process the data. If you are e-commerce platform is not very big to build data lake and build solutions off of data lake. What you can do is you can directly integrate the redshift with the traditional RDBMS's such as Postgres, MySQL, etc. and then you should be able to get the data from the RDBMS tables and store it into redshift tables, using the federated query concept. All you need to do is, you need to expose the databases that are there as part of your RDS as external schemas and databases as part of your redshift cluster. And when you run queries against those schemas, the queries will actually go into the RDS and the data will be fetched from RDS to redshift. The way we typically leverage that feature is we might have a nightly job as part of that nightly job. We might kick off a Redshift PL SQL related job, which will actually consume the data from RDS using Federated Query and populate the tables in Redshift. That's where the Federated queries concepts come into picture. We can build solutions using Redshift PL SQL, or Python based approach, and we should be able to process the data in a very effective manner. That being said, now let's go through the details about how we can actually configure Federated queries using RDS databases and schemas as part of redshift clusters. We'll also see how to run queries and we'll make sure that everything is validated at the end. Let's go step by step to understand what it takes to actually run federated queries from redshift cluster by connecting to traditional RDS based database. In our case will be using Postgres as an example.



To run federated query both postgres n Redshift should be in same vpc or we need to do vpc peering

create external schema spectrum\_schema from data catalog

database 'parquet\_db'

iam\_role 'arn:aws:iam::816069129520:role/redshift-s3-full-access';

create external database if not exists;

CREATE EXTERNAL SCHEMA apg

FROM POSTGRES

DATABASE 'postgres' SCHEMA 'public'

URI 'atuldb.cnoyumcu25us.ap-south-1.rds.amazonaws.com'

IAM\_ROLE 'arn:aws:iam::816069129520:role/redshift-s3-full-access'

SECRET\_ARN 'arn:aws:secretsmanager:ap-south-1:816069129520:secret:res-AQUn6H';

When you run a **federated query** in Amazon **Redshift** that involves a join between **Redshift tables** and **RDS tables** (or **Amazon Aurora** tables), the query execution is split between **Amazon Redshift** and the external **RDS/Aurora** instance. Here’s a detailed breakdown of how the process works:

**1. Where Does the Query Run?**

The query runs in **Amazon Redshift**, but part of the query (the part involving RDS or Aurora) is **delegated** to the external **RDS** or **Aurora** instance.

* **Redshift Processing**: Redshift processes the parts of the query that involve **Redshift tables**.
* **RDS/Aurora Processing**: For the external RDS/Aurora table, the relevant parts of the query are sent to the **RDS/Aurora database** for processing, and only the result set is returned to Redshift.

**2. How Does the Federated Query Work?**

Here’s the step-by-step breakdown of what happens when you run a federated query:

1. **Query Parsing**:
   * When you submit a federated query in Redshift (e.g., joining a Redshift table with an RDS table), Redshift will parse the query and identify which parts involve **Redshift tables** and which parts involve **external data sources** (such as RDS or Aurora).
2. **Delegating to RDS/Aurora**:
   * Redshift will delegate the parts of the query that involve **RDS/Aurora** tables to the external database server (the RDS/Aurora instance). This is done via **JDBC** or **ODBC** connections.
   * Redshift does not load RDS data into its internal storage. Instead, it sends the SQL commands or query logic related to the RDS/Aurora table to the **RDS instance** for execution.
3. **Processing at RDS**:
   * The **RDS** or **Aurora** database processes the query locally on its own database server, just as if you were running the query directly against that database.
   * For example, if you are querying an **RDS MySQL** table and joining it with a Redshift table, the **RDS MySQL** server will handle the data retrieval, filtering, and processing for the RDS-related part of the query.
4. **Results Sent Back to Redshift**:
   * The results of the **RDS/Aurora** query are sent back to **Redshift**.
   * The data returned by RDS is then used in the final query execution, which may involve further processing or joining with Redshift tables.
5. **Final Execution in Redshift**:
   * Once Redshift receives the results from the external RDS/Aurora instance, it will perform any remaining operations like joins, aggregations, or filtering using the Redshift tables.
   * The final results are returned to you as the output of the federated query.

**Redhsift Serverless**

Namespace and Workgroup:

This is the section which talks about Amazon, redshift, serverless workgroups and namespaces. In traditional RDBMS's or traditional redshift. When it comes to databases, we have objects. Objects are nothing but tables, indexes, etc. Also we have users. The users will have access to the databases and the database objects. Also, we run queries against these objects. Whenever we run queries, we need resources. When it comes to traditional RDBMS's or traditional redshift, both the database objects and users as well as compute resources are tightly coupled when it comes to Amazon redshift serverless. These are decoupled. Now let's go through the definition of both namespace and workgroup. When it comes to namespace, it is nothing but collection of database objects and users. It means tables, indexes, views, etc. all come under namespace. If you want to run queries against these tables, you need to have resources. The resources are termed as workgroup, so workgroup is nothing but collection of compute resources. Whereas namespace is nothing but collection of database objects and users. On top of RPU's, which are typically used to run queries. VPC Subnet groups and security groups also come under work group. Keep in mind that most of the costs associated with redshift serverless will be taken away by RPU's in workgroup. When it comes to namespace, we have databases because the redshift is a multitenant database and hence we can have multiple databases. On top of databases, schemas, tables, users, even key management service keys for encrypting data come under namespace. Also, when it comes to storage properties, it include the database name and password of the admin user permissions and encryption and security. Also, there are additional concepts such as datashares, recovery points and usage limits that comes under namespace. I'll not be going through all those details at this time. We'll just focus on how to get started with, as part of the section or module. As part of the previous lecture, we have gone through the details about initiating the creation of workgroup and namespace. In this lecture, we have gone through the concepts related to namespace and workgroup. On top of concepts such as namespace and workgroup it is also important for you to understand what RPU is. You'll be charged based on RPU's, associated with the workgroup, based on the queries you are running against the workgroup. RPU stands for Redshift Processing Unit. It is the unit based on which the costs are computed. As we have gone through the key concepts related to namespace and workgroup. In the next lecture, let's go through the interface related to the workgroup as well

**Keep in mind that the relationship between namespace and workgroup is nothing but one to many. Which means one namespace can have multiple workgroups or more than one workgroup can be associated with one namespace. On top of the namespaces or workgroups.**

**In Amazon Redshift Serverless, RPU (Redshift Processing Unit) is the unit of measure for compute capacity. RPUs represent the amount of compute resources allocated to process queries in a Redshift Serverless workgroup.**

**Key Aspects of RPUs**

1. **Scalable Compute Power:**
   * **Redshift Serverless automatically adjusts the number of RPUs based on the query workload, ensuring you have sufficient compute resources when needed while scaling down during idle periods.**
   * **RPUs combine CPU, memory, and networking resources to handle your query workload.**
2. **Billing and Costs:**
   * **Costs are based on the number of RPUs consumed. You are billed per second for the compute capacity used, with a minimum billing duration of 60 seconds per query.**
   * **Redshift Serverless bills you for:**
     + **The active compute time when queries are being executed.**
     + **Storage costs for data stored in the namespace.**
3. **Performance:**
   * **Higher RPU allocations result in faster query execution, especially for complex or large workloads.**
   * **When the workload increases, Redshift Serverless scales RPUs up dynamically to meet demand. Similarly, it scales down when demand decreases to save costs.**
4. **Baseline and Limits:**
   * **Base Capacity: You can configure a baseline RPU capacity for your workgroup (e.g., 32 RPUs) to ensure a minimum level of performance.**
   * **Maximum RPUs: You can define the maximum RPUs that can be provisioned to control costs and prevent over-scaling.**

**How RPUs Work in Practice**

* **Idle State:**
  + **Redshift Serverless does not allocate compute resources when there are no queries running, effectively setting RPUs to zero (you are only charged for storage).**
* **Query Execution:**
  + **When a query is submitted, Redshift Serverless allocates RPUs dynamically to execute the query based on its complexity and data size.**
* **Parallel Queries:**
  + **For multiple simultaneous queries, more RPUs are allocated to handle the increased workload.**

**8 RPU IS Min and it can scale upto** 5632 RPU

**Use Case Example**

**Suppose you configure a workgroup with a base capacity of 16 RPUs and allow scaling up to 128 RPUs. Here's what happens:**

* **During idle periods, no RPUs are allocated, and you are not billed for compute.**
* **When a query is submitted, the system allocates 16 RPUs for the workload.**
* **If the workload requires more compute (e.g., due to a large join or complex aggregation), the system automatically scales up to 64 or more RPUs.**
* **Once the workload decreases, the RPUs scale back down to save costs.**

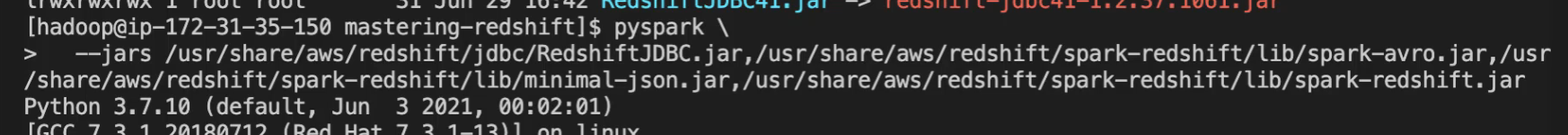
**Benefits of RPUs**

1. **Dynamic Scaling: Automatically adjusts compute resources to match workload requirements.**
2. **Cost-Efficiency: Pay only for what you use, avoiding over-provisioning of resources.**
3. **Simplicity: No need to manually manage clusters or nodes.**

**Run redshift querires using Spark Jobs from EMR**

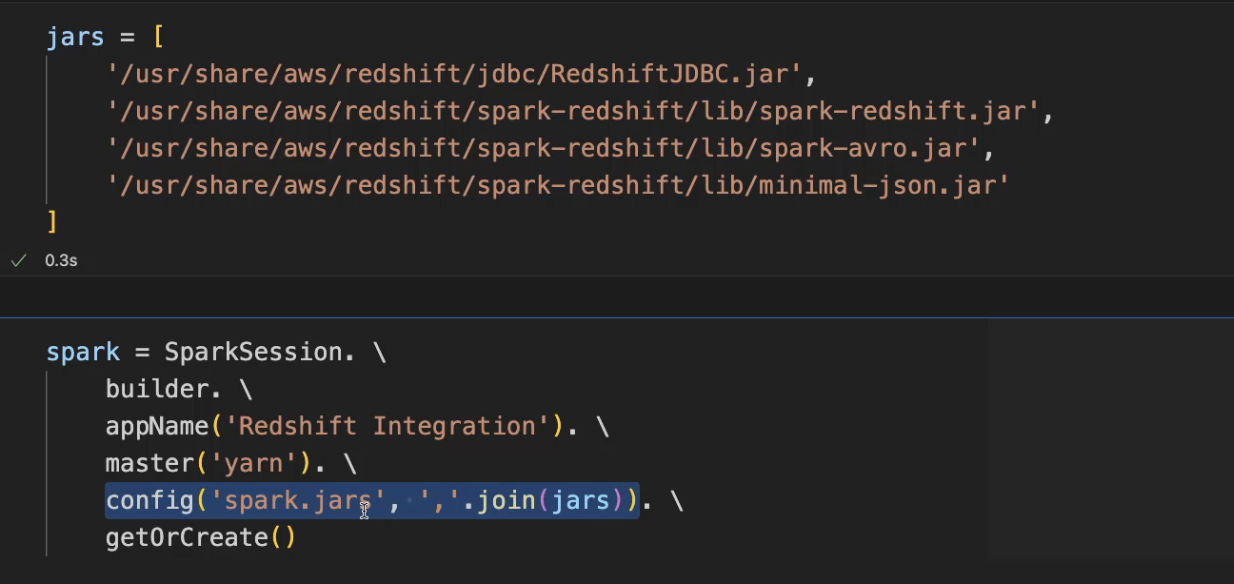
**A screen shot of a computer

Description automatically generated**

****

**A screen shot of a computer program

Description automatically generated**

****

**A screen shot of a computer code

Description automatically generated**

**A computer screen with text on it

Description automatically generated**

**A computer screen with text and images

Description automatically generated with medium confidence**

**A diagram of a computer system

Description automatically generated**

**git clone** [**https://github.com/itversity/mastering-emr.git**](https://github.com/itversity/mastering-emr.git)

**It is very important to set Set spark home path**

**export JAVA\_HOME=/etc/alternatives/jre**

**export SPARK\_HOME=/usr/lib/spark/**

**export PYTHONPATH=/usr/bin/python3**

**export HOME=/home/Hadoop**