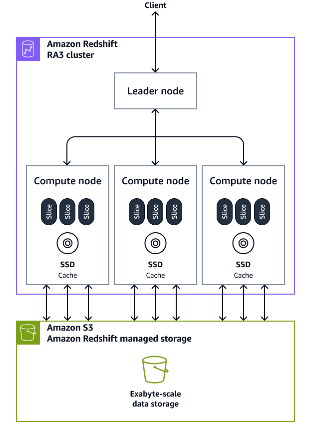
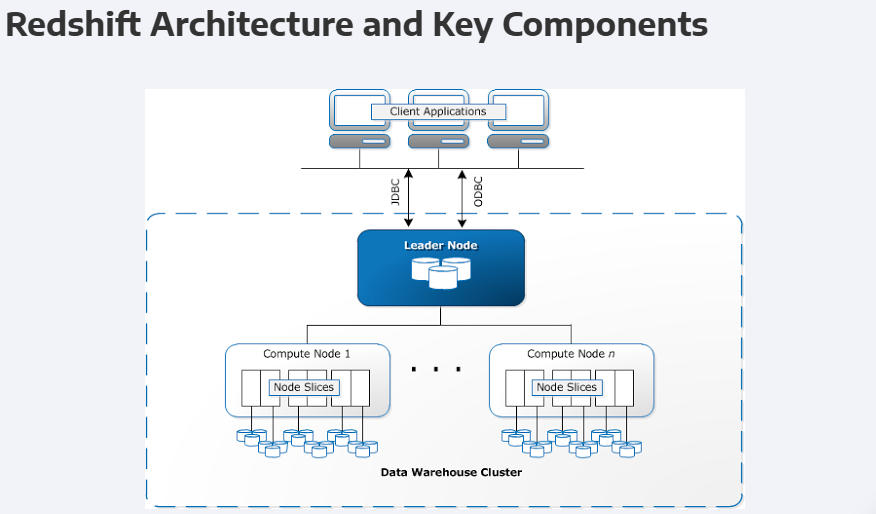
# Amazon Redshift: Architecture and Performance Optimization Guide

## Amazon Redshift data warehouse architecture components.

## I. Introduction to Amazon Redshift

**Amazon Redshift** is a fully managed, petabyte-scale **data warehouse** service in the cloud, designed for high-performance analytics on large datasets. It uses a **Massively Parallel Processing (MPP)** architecture to distribute data and query workloads across multiple nodes, enabling fast query execution for datasets ranging from gigabytes to petabytes. Redshift is optimized for complex analytical queries and integrates with AWS services like **Amazon S3** for scalable storage and **Redshift Spectrum** for querying external data.

## II. Redshift Architecture

Redshift’s architecture is built around a cluster model with distinct roles for processing and storage:

* **Leader Node**:
  + Manages client connections via **JDBC/ODBC**.
  + Parses queries, optimizes them, and creates **query execution plans**.
  + Coordinates work distribution to **Compute Nodes** and aggregates results.
  + Stores metadata, not user data.
* **Compute Nodes**:
  + Execute queries in parallel, processing data stored in **Redshift Managed Storage (RMS)**.
  + Divided into **Node Slices**, logical units with dedicated CPU, memory, and SSD storage.
  + Slice count varies by instance type (e.g., dc2.large has 2 slices, ra3.16xlarge has 16).
* **Redshift Managed Storage (RMS)**:
  + A scalable storage layer separate from compute.
  + Stores data in **columnar format** using 1 MB **data blocks**.
  + Caches **hot data** in SSDs and offloads **cold data** to **Amazon S3**.
  + Uses **Zone Maps** to track min/max values per block for efficient query pruning.
* **Amazon S3 Integration**:
  + Supports backups, data unloading, and external queries via **Redshift Spectrum**.
  + RMS moves cold data to S3 transparently for cost-effective scaling.

## III. Data Loading and Storage

### Data Loading Process

1. **Ingestion**:
   * Data is loaded from sources like **Amazon S3** or **DynamoDB** using the COPY command.
   * Example: COPY sales FROM 's3://mybucket/sales\_data' IAM\_ROLE 'arn:aws:iam::123:role/MyRole';
2. **Leader Node Coordination**:
   * Parses the command and assigns tasks to **Compute Nodes** based on the **Distribution Style**.
3. **Compute Node Processing**:
   * Applies the **Distribution Style** to assign data to **slices**.
   * Sorts data if a **Sort Key** is defined.
   * Stores data in **RMS** in **columnar format**.

### Storage in RMS

* **Columnar Storage**: Stores data by column, reducing I/O for analytical queries.
* **Data Blocks**: Each column is split into 1 MB blocks with **Zone Maps** and compression metadata.
* **Hot vs. Cold Data**: RMS caches hot data in SSDs and offloads cold data to S3.

## IV. Distribution Styles

**Distribution Styles** determine how data is spread across **Compute Nodes** and **slices**, affecting query performance and data movement.

**Types of Distribution Styles**

| **Style** | **Description** | **Best Use Case** | **Data Duplication** |
| --- | --- | --- | --- |
| **EVEN** | Distributes rows randomly across slices | General-purpose tables, no specific joins | ❌ No |
| **KEY** | Distributes rows based on a **Distribution Key** | Large tables joined on a specific column | ❌ No |
| **ALL** | Duplicates the table across all nodes | Small dimension tables for joins | ✅ Yes |

* **Default**: **EVEN** for provisioned clusters; **AUTO** for serverless/RA3 clusters (Redshift chooses based on workload).

**Choosing a Distribution Style**

* **EVEN**: Balances workloads but may require data movement during joins.
* **KEY**: Optimizes joins by colocating rows; risks **data skew** if the key is unevenly distributed.
* **ALL**: Eliminates data movement for small tables but increases storage.

## V. Sort Keys

**Sort Keys** define the physical order of data within slices, improving query performance via **Zone Maps**.

**Types of Sort Keys**

| **Type** | **Description** | **Best Use Case** |
| --- | --- | --- |
| **Compound** | Sorts by the first column, then others | Queries filtering on a primary column |
| **Interleaved** | Equal weight to all columns | Queries filtering on multiple columns |

**Impact of No Sort Key**

* Data is stored in insertion order, reducing **Zone Map** efficiency and slowing queries.

## VI. Query Execution and Optimization

### Query Execution Flow

1. **Client Submission**: Query sent via **JDBC/ODBC**.
2. **Leader Node**: Parses, optimizes, and distributes tasks to **Compute Nodes**.
3. **Compute Nodes**: Fetch and process data blocks in parallel from **RMS**.
4. **Result Aggregation**: **Leader Node** aggregates and returns results.

### Optimization Techniques

* **Zone Maps**: Enable **block pruning** to skip irrelevant data.
* **Sort Keys**: Order data for faster filtering.
* **Distribution Styles**: Reduce data movement during joins.

## VII. Data Movement in Joins

Joins may trigger **data movement** if tables aren’t optimally distributed:

| **Join Type** | **Description** | **Data Movement** | **Performance** |
| --- | --- | --- | --- |
| **Co-located** | Same **Distribution Key** on both tables | ❌ None | 🚀 Fastest |
| **Broadcast** | Small table copied to all nodes | ✅ Small table | 👍 Good |
| **Redistribution** | Data shuffled across nodes | ✅ Large movement | 🐢 Slowest |

**Minimizing Data Movement**

* Use **KEY Distribution** with a common join key.
* Apply **ALL Distribution** for small dimension tables.

## VIII. Metadata and System Tables

System tables provide insights into data distribution:

| **Table** | **Purpose** | **Example Query** |
| --- | --- | --- |
| **STV\_BLOCKLIST** | Tracks block distribution across slices | SELECT node, slice, COUNT(\*) FROM STV\_BLOCKLIST GROUP BY node, slice; |
| **SVV\_TABLE\_INFO** | Shows distribution and sort key details | SELECT "table", diststyle, sortkey1 FROM SVV\_TABLE\_INFO; |

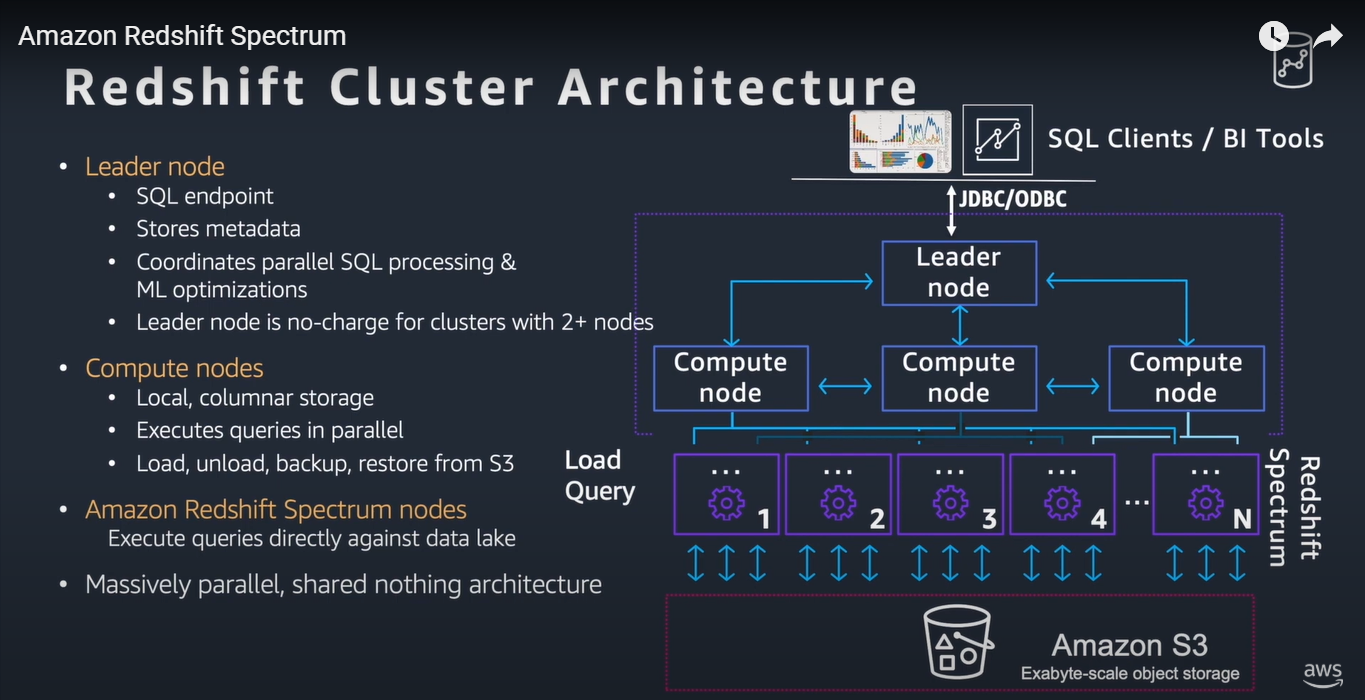
## IX. Best Practices for Performance Optimization

* **Distribution Styles**: Use **KEY** for joins, **ALL** for small tables, **EVEN** for balance.
* **Sort Keys**: Apply **Compound** for primary filters, **Interleaved** for multi-column queries.
* **Query Optimization**: Leverage **Zone Maps** and minimize data movement.

## Confusion Summary: Key Areas of Misunderstanding

1. **Data Storage and Compute Node Roles**
   * **Confusion**: Compute nodes store data persistently.
   * **Clarification**: Data resides in **RMS**; compute nodes use SSDs for temporary processing.
2. **Distribution Styles**
   * **Confusion**: Distribution styles only affect queries, not storage.
   * **Clarification**: They are applied at load time, impacting storage and performance.
3. **Sort Keys**
   * **Confusion**: Sort keys are applied during queries.
   * **Clarification**: They are applied at load time, ordering data in slices.
4. **Data Movement in Joins**
   * **Confusion**: Unclear when data movement happens.
   * **Clarification**: Minimized with **co-located joins**; otherwise, **broadcast** or **redistribution** occurs.
5. **Slices and Data Assignment**
   * **Confusion**: How slices and data assignment work.
   * **Clarification**: **Leader Node** assigns slices at load time; compute nodes use metadata to fetch data.

## Redshift Spectrum



* **Redshift Spectrum** is the engine that handles queries for **external data in S3**.
* **Leader Node** creates the **query execution plan**, and **Redshift Spectrum** is responsible for scanning the external data in S3.
* **Distribution Styles** and **Sort Keys** only affect **Redshift's internal data** (in RMS). For external data, **Redshift Spectrum** handles data retrieval from S3, but **distribution styles** and **sort keys** are **not applied** to external data directly.
* If your query involves both **internal** and **external data**, **distribution styles** and **sort keys** apply to the **internal Redshift data** to improve query performance and minimize data shuffling.