



Blendo

AMAZON REDSHIFT GUIDE FOR DATA ANALYSTS

BECOME AN AMAZON REDSHIFT SUPERHERO

TABLE OF CONTENTS

1. Introduction
2. Chapter One - Cluster Management
3. Cluster Management: Introduction
4. Setup your Redshift Cluster
5. Configure Networking and Security Groups
6. User Management
7. Overview of System Tables and Views
8. Workload Management and Query Queues
9. Chapter Two - Import and Export Data
10. Import and Export Data: Introduction
11. Load Data Into Amazon Redshift
12. Export Data from Amazon Redshift
13. Tools to connect to your Amazon Redshift Cluster
14. Chapter Three - Data Modeling and Table Design

15. Data Modeling and Table Design: Introduction

16. Table Distribution Styles

17. Understanding and Selecting Sort Keys

18. Table Views in Amazon Redshift

19. Modeling Time Series data

20. Chapter Four - Maintenance

21. Maintenance: Introduction

22. Why to Vacuum Amazon Redshift?

23. Column Compression Settings

24. Monitoring Amazon Redshift Query Performance

25. Outro

INTRODUCTION

At Blendo we have tried various data warehouses for our use and as a destination of our **ETL as a service platform**. Today you will find many alternative cloud solutions from **Amazon, Google, Microsoft Azure** and more.

Amazon Redshift is part of the Amazon Web Services (AWS) ecosystem and one of the most popular data warehousing solutions. It is a petabyte scale, fully managed data warehouse as a service solution that runs on the cloud. It is SQL based, and you can communicate with it as you would do with PostgreSQL.

As Blendo, we support **Amazon Redshift as a data warehouse destination**, and as we have worked a lot with it, we gathered a lot of knowledge and experience on its ins and outs. We chose to share this knowledge as we hope it will make us exchange views and learn together.

Having in mind the Data Analyst of a company we collected all this knowledge inside the Amazon Redshift Guide for Data Analysts.

So let's get started. Shall we?

CHAPTER ONE - CLUSTER MANAGEMENT

Let's start off with the basics. Our first step is to have an Amazon Redshift cluster set up. This chapter will show you exactly how to do that.

CLUSTER MANAGEMENT: INTRODUCTION

Let's start off with the basics. Our first step is to have an Amazon Redshift cluster set up. This chapter will show you exactly how to do that.

How to setup the most important parts of your cluster, permissions and how to work with system tables and schemas.

Though, you need first to start with a decision. There are two types of Redshift instances you can set up in AWS, **Dense Compute (DC)** and **Dense Storage (DS)**.

Node Types in Amazon Redshift

When you launch a cluster, you need to specify the node type, that determines CPU, RAM, and storage for each node. A high-level difference is that:

The **Dense Compute** cluster has less storage, but better performance and speed. The more data you are querying, the more compute you need to keep queries fast. The Dense Compute type of instances is good to use if you need a high-performance data warehouse.

The **Dense Storage** cluster is designed for big data warehouses. So, if you have too much data to fit.

There is also a difference in the pricing, so go here to check some more information from AWS and its [pricing](#).

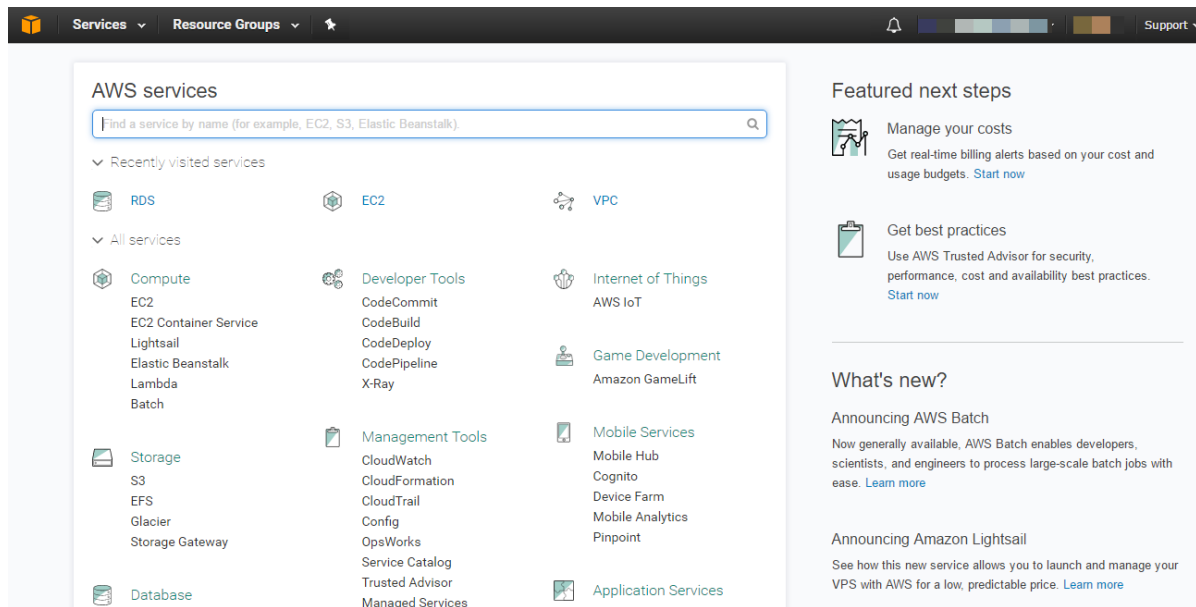
Having said that let's see with more details how to create an Amazon Redshift cluster in the next section.

SETUP YOUR REDSHIFT CLUSTER

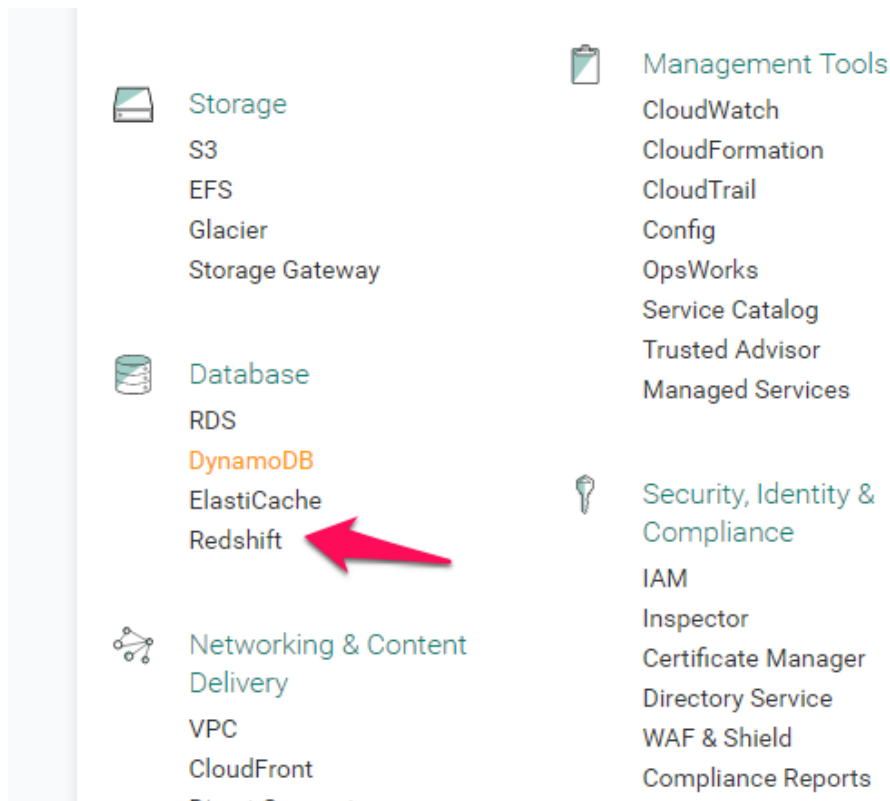
In this section, we will see how to setup a new Amazon Redshift cluster. Apparently, there are many things you may set up, but the main idea is as follows.

Your Amazon AWS dashboard

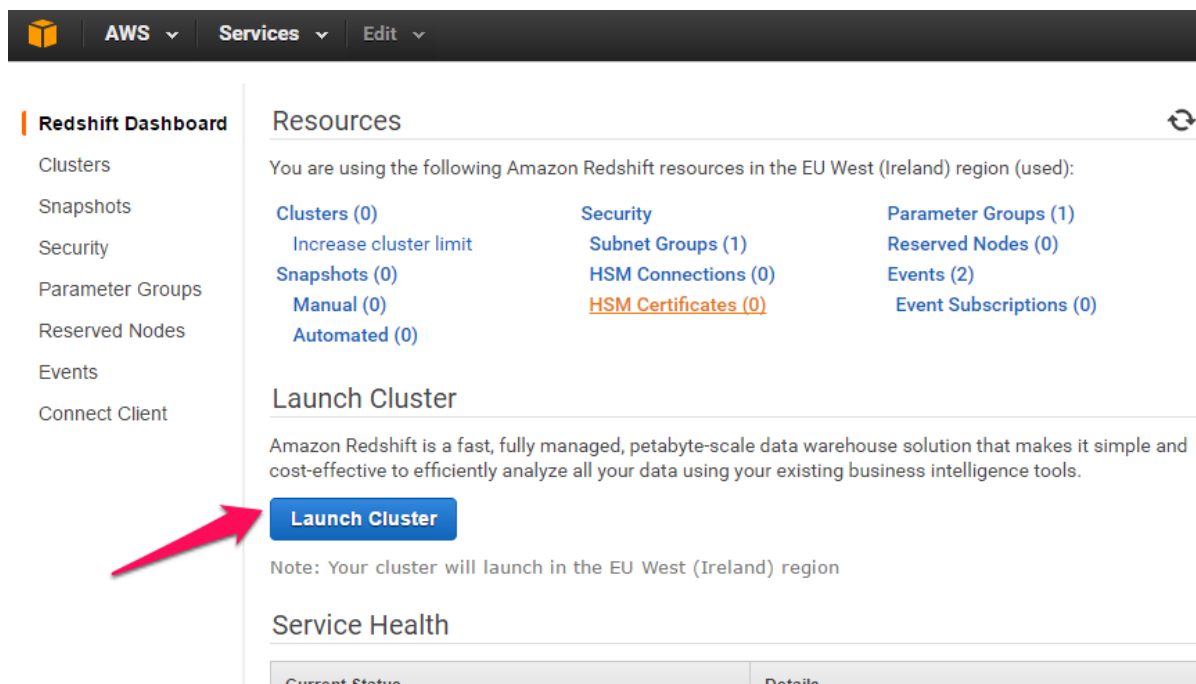
If you have an account in AWS, you can log in [here](#) (You may set up a new one too).



After your login to AWS, navigate to Databases and click on **Redshift**.



Click on **Launch Cluster**.



Let's setup our cluster.

- Cluster Identifier: Give it a name
- Database Name: This optional. You can give one that you want.
- Master User Name: Provide a Username of master user of the cluster.
- Master User Password: the Password :)

If you are ready, click on **Continue**.

CLUSTER DETAILS NODE CONFIGURATION ADDITIONAL CONFIGURATION REVIEW

Provide the details of your cluster. Fields marked with * are required.

Cluster identifier*	<input type="text" value="tyrion-lannister"/>	This is the unique key that identifies a cluster. This parameter is stored as a lowercase string. (e.g. my-dw-instance)
<hr/>		
Database name	<input type="text" value="lannisters"/>	Optional. A default database named dev is created for the cluster. Optionally, specify a custom database name (e.g. mydb) to create an additional database.
Database port*	<input type="text" value="5439"/>	Port number on which the database accepts connections.
Master user name*	<input type="text" value="tyrion"/>	Name of master user for your cluster. (e.g. awsuser)
Master user password*	<input type="password" value="....."/>	Password must contain 8 to 64 printable ASCII characters excluding: /, ", ', \, and @. It must contain 1 uppercase letter, 1 lowercase letter, and 1 number.
Confirm password*	<input type="password" value="....."/>	Confirm master user password

Selecting Node Type

Now we are in the Node Configuration part of our setup. At this point, we are going to choose between **Dense Compute (DC)** and **Dense Storage (DS)**. As a reminder:

The **Dense Compute** cluster provides less storage, but with better performance and speed. The more data you are querying, the more compute you need to keep queries fast. Dense Compute is the type of instance to use if you need a high-performance data warehouse.

The **Dense Storage** cluster is designed for big data warehouses. So, if you have too much data to fit.

Services Edit Kon

CLUSTER DETAILS NODE CONFIGURATION ADDITIONAL CONFIGURATION REVIEW

Choose a number of nodes and Node Type below. Number of Compute Nodes is required for multi-node clusters.

Node Type dc1.large

CPU 7 EC2 Compute Units (2 virtual cores) per node

Memory 15 GiB per node

Storage 160GB SSD storage per node

I/O Performance Moderate

Specifies the compute, memory, storage, and I/O capacity of the cluster's nodes.

Cluster Type Single Node

Number of Compute Nodes* 1

Maximum 1

Minimum 1

Single Node clusters consist of a single node which performs both leader and compute functions.

Cancel Previous Continue

For our example, we will leave the Node Type as is.

Selecting Number of Nodes

You will need to choose the number of nodes that your cluster will work. That number depends on the size of your dataset and your desired query performance.

For example ds2.xlarge Dense Compute nodes have 2TB HDD storage per node. For 12 TB of data, you need 6 ds2.xlarge nodes or 1 ds2.8xlarge nodes. At the same time choosing dc1.8xlarge Compute Node will give you 2.56TB SSD storage per node.

Node type dc1.large

CPU 7 EC2 Compute Units (2 virtual cores) per node

Memory 15 GiB per node

Storage 160GB SSD storage per node

I/O performance Moderate

Specifies the compute, memory, storage, and I/O capacity of the cluster's nodes.

Cluster type Multi Node

Number of compute nodes* 2

Maximum 32

Minimum 2

Compute nodes are used to execute your queries. One node acts as a leader node and is not charged. 1 node is charged for ODBC/JDBC connections and executed queries.

For our case, we chose: 2 x dc1.large Compute Nodes.

Click on **Continue**.

Additional Configuration

In this setup, you configure some additional setup and the networking options.

Parameter groups:

A parameter group is a group of parameters that apply to all of the databases that you create in the cluster. You associate a parameter group with each Amazon Redshift cluster you create. **Read more about Parameter Groups** in AWS's documentation [here](#).

Database Encryption:

You may also enable database encryption for your Amazon Redshift cluster to protect your data. When you enable encryption for a cluster, the data blocks and system metadata are encrypted for the cluster and its snapshots. **Read more about Amazon Redshift Database Encryption and the relevant best practices** in AWS's documentation [here](#).

The screenshot shows the 'Additional Configuration' step in the Amazon Redshift console. At the top, there is a progress bar with four steps: 'CLUSTER DETAILS', 'NODE CONFIGURATION', 'ADDITIONAL CONFIGURATION' (which is the current step), and 'REVIEW'. Below the progress bar, the text 'Provide the optional additional configuration details below.' is displayed. There are two main configuration options, each highlighted with a red rectangular box. The first box contains the 'Cluster parameter group' label and a dropdown menu currently showing 'default.redshift-1.0'. To the right of this box is the text 'Parameter group to associate with this cluster.' The second box contains the 'Encrypt database' label and three radio button options: 'None' (which is selected), 'KMS', and 'HSM'. To the right of this box is a blue link that says 'Learn more about database encryption'. Below these boxes, the text 'Configure networking options:' is visible.

For our example, we leave the default settings*.

**It is advised to read and review Amazon AWS's documentation for both above cases.*

Network Setup

Now let's configure our networking options.

Virtual Private Cloud (VPC)

Amazon Virtual Private Cloud (Amazon VPC) resembles a traditional virtual network. You may define that virtual network, the VPC, and launch the AWS services you need to run into it. For more details about VPC check AWS's documentation [here](#).

So as a first step if you have a VPC, then you need to provide it. If you do not have a VPC, a default is created.

Configure networking options:

Choose a VPC Default VPC (vpc-a6e912c3) ▼ The identifier of the VPC in which you want to create your cluster

Cluster subnet group default ▼ Selected Cluster Subnet Group may limit the choice of Availability Zones

Publicly accessible ☒ Yes ☐ No Select Yes if you want the cluster to be accessible from the public internet. Select No if you want it to be accessible only from within your private VPC network

For our example, we leave the default*.

**It is advised to read and review Amazon AWS's documentation for the above cases.*

Cluster subnet group

If you are going to provision a Redshift cluster in a VPC, then you need to create a cluster subnet group. You can have multiple subnets that will help you organize your AWS resources. For more details on Amazon Redshift Cluster Subnet Groups check AWS's documentation [here](#).

Configure networking options:

Choose a VPC Default VPC (vpc-a6e912c3) ▼ The identifier of the VPC in which you want to create your cluster

Cluster subnet group default ▼ Selected Cluster Subnet Group may limit the choice of Availability Zones

Publicly accessible ☒ Yes ☐ No Select Yes if you want the cluster to be accessible from the public internet. Select No if you want it to be accessible only from within your private VPC network

Configure the Publicly Accessible option

Configuring this is optional, but if you want to access your Redshift cluster from outside of AWS, then you need to add a public IP by setting Publicly Accessible to Yes. If you want your cluster to be accessible only from within your private VPC network, then choose No.

If you select Yes, then you have the option to select an Elastic IP address (EIP) to use for the external IP address.

An EIP is a static IP address that is associated with your AWS account. You can use an EIP to connect to your cluster from outside the VPC. Read more about EIP in AWS's documentation [here](#).

Publicly accessible ☒ Yes ☐ No Select Yes if you want the cluster to be accessible from the public internet. Select No if you want it to be accessible only from within your private VPC network.

Choose a public IP address ☒ Yes ☐ No Select Yes if you want the cluster to have a public IP address that can be accessed from the public Internet, select No if you want the cluster to have a private IP address that can only be accessed from within the VPC.

Elastic IP Select an EIP to use to connect to the cluster from outside of the VPC.

The choice of using a public IP or not is up to you.

Amazon Redshift Enhanced VPC Routing

If you select Yes, then Amazon Redshift forces all COPY and UNLOAD traffic between your cluster and your data repositories through your Amazon VPC. That is important as this routing affects the traffic between your services as it travels through the Internet (including traffic to other services within the AWS network).

Enhanced VPC Routing needs extra care, and you probably need to review AWS's documentation [here](#).

Enhanced VPC Routing ☐ Yes ☒ No Select Yes if you want to enable Enhanced VPC Routing. [Learn more](#)

Availability zone The EC2 Availability Zone that the cluster will be created in.

Availability Zone

Each region in AWS has multiple Availability Zones that are isolated locations known as Availability Zones. Read more about Regions and Availability Zones in AWS's documentation [here](#).


Select **No Preference** to have AWS select the availability zone that your Redshift cluster will be created. Otherwise, select a specific availability zone.

VPC Security Groups

Last but not least, Security Groups. A Security Group is a set of rules that control access to your Redshift cluster, for example, a range of IP addresses that allow a third party tool to connect to your Redshift. You can select this Security Group here, but you can also assign it later in your cluster configuration. For more details on Security Groups read AWS documentation [here](#).

Optionally, associate your cluster with one or more security groups.

VPC security groups



List of VPC security groups to associate with this cluster.

Optionally, create a basic alarm for this cluster.

Click on **Continue**.

Launch your Cluster



In the next page review your settings and click **Launch Cluster**.

Next, click **close** and in the next screen wait for your cluster to become **Available** and **Healthy**.

Launch Cluster

Manage Tags

Manage IAM roles

<input type="checkbox"/>		Cluster	Cluster Status	DB Health	In Maintenance	Recent Actions
<input type="checkbox"/>	 	tyrion-lannister	creating	unknown	unknown	0

In the next section, we will see how to define networking and security settings for your Redshift instance.

This section contains parts of our knowledge base article "[Setup Amazon Redshift](#)"

CONFIGURE NETWORKING AND SECURITY GROUPS

There are two types of Redshift cluster subnets:

- EC2 Classic subnet
- VPC subnet

Let's see what our cluster supports as it depends on the region you are. To see this, you will have to check your cluster's **Configuration** tab. You will either see a field called **Cluster Security Groups** or **VPC Security Groups**.

The screenshot shows the Amazon Redshift console interface. On the left is a navigation menu with 'Clusters' highlighted. The main panel shows the 'Configuration' tab for cluster 'tyrion-lannister'. The 'Cluster Subnet Group' is 'default'. The 'VPC security groups' field is highlighted with a red box and shows '(active)'. Other fields include 'Cluster Name', 'Cluster Type', 'Node Type', 'Nodes', 'Zone', 'Created Time', 'Cluster Version', 'VPC ID', 'Cluster Parameter Group', and 'Enhanced VPC Routing'.

Cluster Properties		Cluster Status	
Cluster Name	tyrion-lannister	Cluster Status	available
Cluster Type	Multi Node	Database Health	healthy
Node Type	dc1.large	In Maintenance Mode	no
Nodes	2	Parameter Group Apply Status	in-sync
Zone	eu-west-1a	Pending Modified Values	None
Created Time	March 22, 2017 at 3:57:30 PM UTC+2		
Cluster Version	1.0.1232		
VPC ID	View VPCs		
Cluster Subnet Group	default		
VPC security groups	(active)		
Cluster Parameter Group	sync		
Enhanced VPC Routing	No		


You can find more details about these two subnet types you may check from Amazon the [Managing Clusters in an Amazon Virtual Private Cloud \(VPC\) guide](#).


Configure Security Groups

Amazon Redshift permissions: EC2-Classic

Click on the name next to **Cluster Security Groups**.



ard

Cluster:  Configuration Status Performance Queries Loads




Cluster: 

Cluster Database Backup

S

Endpoint:  s.com:5439 (No Authorizations) 

Cluster Properties

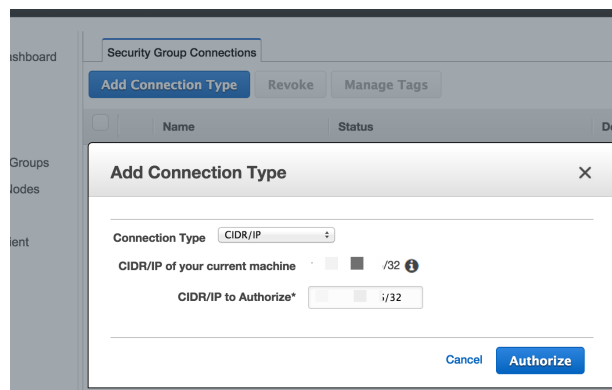
Cluster Name	
Cluster Type	Single Node
Node Type	dc1.large
Nodes	1
Zone	us-west-2a
Created Time	
Cluster Version	
Cluster Security Groups	default (active)
Cluster Parameter Group	default.redshift-1.0 (in-sync)

Cluster Database Properties

Port	5439
------	------

Then you will see a new page called **Security**. Click on the name of your **Security group**. At next page click on **Add Connection Type**.

In the pop-up choose **Connection Type**: CIDR/IP and the default IP is your current IP address.
































































































Click **Authorize**.

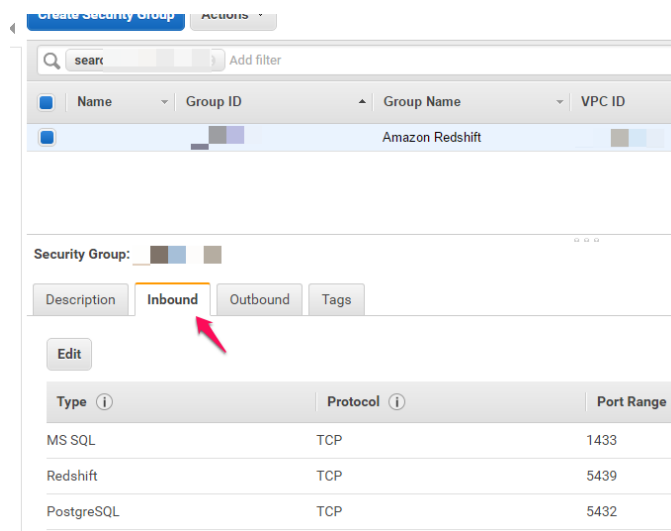
Amazon Redshift permissions: EC2-VPC

For our example case, click on the name next to **VPC Security Groups**.

Cluster Properties

Cluster Name	tyrion-lannister
Cluster Type	Multi Node
Node Type	dc1.large
Nodes	2
Zone	eu-west-1a
Created Time	March 22, 2017 at 3:57:30 PM UTC+2
Cluster Version	1.0.1232
VPC ID	                               View VPCs
Cluster Subnet Group	default
VPC security groups	                               (active)
Cluster Parameter Group	                               -sync)
Enhanced VPC Routing	No

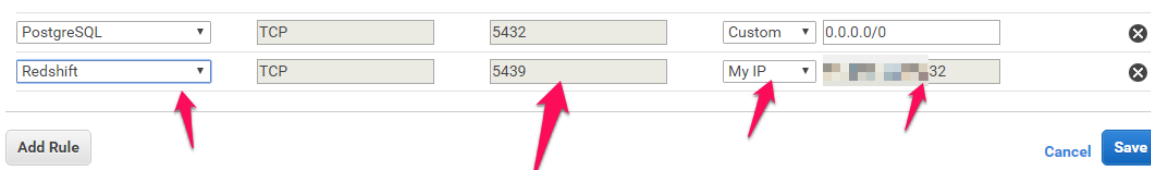
Click the **Inbound** tab and then click on **Edit**.




















Type	Protocol	Port Range
MS SQL	TCP	1433
Redshift	TCP	5439
PostgreSQL	TCP	5432

We need to add one rule.

In the **Type** drop-down select **Redshift**. Then leave the defaults. At **Source**, in the drop-down select **My IP**. Leave the default.



PostgreSQL	TCP	5432	Custom	0.0.0.0/0
Redshift	TCP	5439	My IP	                 32

Add Rule **Cancel** **Save**

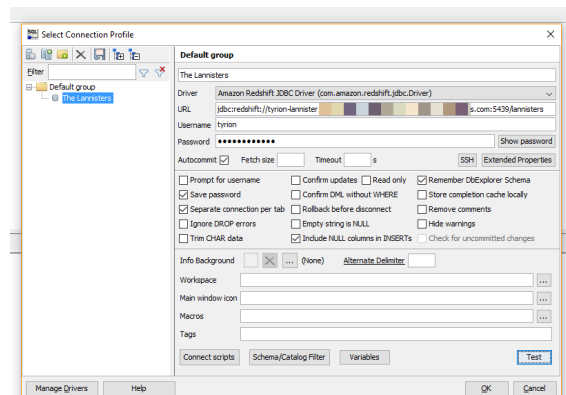
Click **Save**

Now we are ready to connect to our cluster. There are various [tools to connect to your Amazon Redshift](#) cluster, and we will see them later in more detail. For now, we will try a connection with SQL Workbench/J.

Connecting with SQL Workbench/J

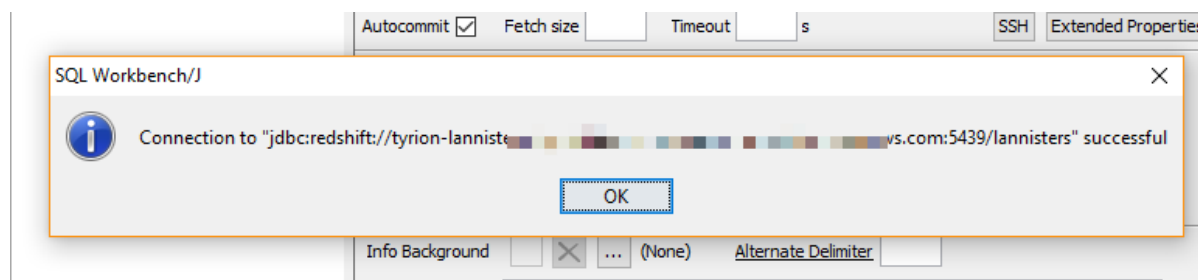
Let's see how to connect with SQL Workbench/J*.

1. Download SQL Workbench/J from [here](#).
2. Read the Installing and starting SQL Workbench/J page from [here](#).
3. Download the latest Amazon Redshift JDBC drivers from Amazon [here](#).
4. Open SQL Workbench/J.
5. If the Select Connection Profile does not open, click File, and then Connect.
6. Type a name for the profile. For example, *The Lannisters*.
7. Click Manage Drivers.
8. In the new window find and select the JAR with the Amazon Redshift JDBC Driver.
9. Click OK
10. For the rest of settings check the screenshot below.



*For more detailed instructions examine the documentation from AWS [here](#).

Click on **Test**, and if you see the bellow message, you are in.



USER MANAGEMENT

In the prior sections, we covered [how to setup an Amazon Redshift cluster](#) and [how to configure networking and security groups](#). In this section, we go over some matters about **User Management**. As this is a relatively complicated issue, we have included various links from Amazon's documentation for reference.

We mentioned earlier, when you create an instance you create the **Master User** and its login credentials. However, Redshift allows you to control further and configure your permission settings or add new users.

Access to Amazon Redshift resources has 3 levels of control:

1. Cluster Connectivity Permissions
2. Cluster Management Permissions
3. Access to Database Permissions

Let's see an overview, but first, what is **IAM**?

IAM is **AWS Identity and Access Management** and helps you control access to AWS resources. IAM controls who can use your AWS resources, which of these resources and how.

Cluster Connectivity Permissions

In the previous section, we discussed the cluster connectivity permissions which are actually over networking access and security groups. So you are welcome to refer to [Configure Networking and Security Groups](#).

Cluster Management Permissions

Cluster Management permissions are provisioned to AWS users, and their access is managed by **IAM policies** and **roles**. Amazon separates such access to Authentication and Access Control.

Authentication

AWS account root user – That is your AWS account when you first signed up. It provides complete access to all of your AWS resources.

We always need to be cautious with User Management and Amazon recommends using your root credentials only to create an administrator user. An administrator user is an IAM user with full permissions to your AWS account. Then, use this administrator user to create other IAM users and roles with limited permissions.

IAM user – Is an identity with specific permissions to your AWS resources. You can have an IAM user name and password and also generate access keys for accessing AWS services programmatically either through one of the several SDKs or by using the **AWS Command Line Interface (CLI)**.

IAM role – The IAM role is similar to an IAM user but does not associate with a specific person. An IAM role enables you to do specific things in AWS. It has no username, password or keys associated with it but instead, if a user is assigned to an IAM Role, access keys are created dynamically and provided to that user. Everything you need to know about IAM Roles is [here](#).

Access Control

Access control is the permission you have to perform operations like creating an Amazon Redshift cluster, IP addresses, Security Groups, Snapshots and more. Amazon Redshift supports identity-based policies (IAM Policies) which are policies attached to an IAM identity. For example, you may attach permissions policy to a user to allow him to create an Amazon Cluster.

For detailed information and best practices about using IAM policies see the **Access Management** section in the IAM User Guide [here](#).

You may also find a list with example policies for administering AWS resources in [here](#).

Notes & Resources:

- The must read **IAM User Guide**.
- Overview of AWS IAM Permissions [here](#).
- Overview of IAM Policies [here](#).
- A significant part of AWS IAM User Guide are the **IAM Best Practices**.

Access to Database Permissions

Access to the database is the ability to have control over a database's objects like tables and views. You must be a **superuser** to create an Amazon Redshift user. The Master User is a superuser.

A database superuser bypasses all permission checks. Be very careful when using a superuser role. AWS recommends that you do most of your work as a role that is not a superuser. Superusers retain all privileges regardless of GRANT and REVOKE commands.

A **superuser** can create other **superusers** and **users**. These users can be owners of databases, tables, views, grant privileges for specific objects and resources. **Superusers have database ownership privileges to all databases.**

Below we are going to see some Amazon Redshift queries which can be helpful in your user management along with links with more details on each command.

Create New User

```
CREATE USER <user_name>;  
  
CREATE USER <user_name> WITH PASSWORD '<a_password>';
```

More info on the [CREATE USER](#) command.

Create User in a Group

```
CREATE USER <user_name> WITH PASSWORD '<a_password>' IN GROUP  
<group_name>;
```

More info on the [CREATE USER](#) command.

Drop a User

```
DROP USER IF EXISTS <user_name>;
```

Note: You cannot drop a user if the user owns any database object, such as a schema, database, table, or view, or if the user has any privileges on a table, database, or group.

More info on the [DROP USER](#) command

Alter a User

```
ALTER USER <user_name> CREATEDB;
```

That command allows the *USER* <user_name> to create new databases. Here is more info on the **ALTER USER** command.

View all Users

```
SELECT * FROM pg_user;
```

To view the list of users, we query the `pg_user` catalog table.

Create New Group

```
CREATE GROUP <group_name>;
```

More info on the **CREATE GROUP** command.

View all Groups

```
SELECT * FROM pg_group;
```

To view the list of groups, we query the `pg_group` catalog table.

View all Schemas

```
SELECT * FROM pg_namespace;
```

To view a list of all schemas, we query the `pg_namespace` catalog table.

View Tables that belong to a Schema

```
SELECT distinct(<table_name>) FROM pg_table_def  
WHERE <schema_name> = 'pg_catalog';
```

The above will query the `pg_table_def` system catalog table and will return a list of tables in the `pg_catalog` schema.

Grant USAGE on a schema to a user

```
GRANT USAGE ON SCHEMA <schema_name> TO <user_name>;
```

The *USER* will now have *USAGE* rights on a <schema_name> *SCHEMA*.

Grant SELECT privileges to a user to all tables

```
GRANT SELECT ON ALL TABLES IN SCHEMA <schema_name> TO <user_name>;
```

The *USER* will now have *SELECT* permissions on all the tables in a <schema_name> *SCHEMA*.

Grant SELECT privileges to a user to a table

```
GRANT SELECT ON TABLE <schema_name>.<table_name> TO <user_name>;
```

The *USER* <user_name> will now have *SELECT* rights on *TABLE* <table_name> in the <schema_name> *SCHEMA*.

Grant ALL privileges to a user to all tables

```
GRANT ALL ON ALL TABLES IN SCHEMA <schema_name> TO <user_name>;
```

The *USER* will now have *ALL* rights on *ALL TABLES* in the <schema_name> *SCHEMA*.

Create a Read-only User

Create a New User with:

```
CREATE USER <user_name> WITH PASSWORD '<password>';
```

Then we can grant *USAGE* rights on the <schema_name> *SCHEMA* to the <user_name> with:

```
GRANT USAGE ON SCHEMA <schema_name> TO <user_name>;
```

Last grant *SELECT* on the <table_name> *TABLE* with:

```
GRANT SELECT ON TABLE <schema_name>.<table_name> TO <user_name>;
```

Grant USAGE on the schema to a group

```
GRANT USAGE ON SCHEMA <schema_name> TO <group_name>;
```

The *GROUP* <group_name> will now have *USAGE* rights on the <schema_name> *SCHEMA*.

Grant SELECT privileges to a group, to all tables

```
GRANT SELECT ON ALL TABLES IN SCHEMA <schema_name> TO  
<group_name>;
```

The *GROUP* <group_name> will now have *SELECT* rights on *ALL TABLES* in the <schema_name> *SCHEMA*.

Grant SELECT privileges to a group, to a table

```
GRANT SELECT ON TABLE <schema_name>.<table_name> TO <group_name>;
```

The *GROUP* <group_name> will now have *SELECT* rights on <table_name> in the

`<schema_name> SCHEMA.`

Notes & Resources:

- More details on **superusers**, [here](#).
- More details on **Users**, [here](#).
- More info on Creating, Altering, and Deleting **Groups** [here](#).
- More info on Creating, Altering, and Deleting **Schemas** [here](#).
- Information on [Managing Database Security](#).

OVERVIEW OF SYSTEM TABLES AND VIEWS

A Redshift cluster has many system tables and views you can query to understand how your system behaves. However, before we start, here are some naming conventions. There are two types of system tables: STL and STV tables. There are also some Views. Let's see:

STL tables: (Tables with *stl_* prefix) These system tables have logs, provide a history of the system, and they are persisted.

STV tables: (Tables with *stv_* prefix) They are virtual tables with snapshots of the current system state data.

SVL views: (Views with *svl_* prefix) They are system views that contain references to STL tables and logs for more detailed information.

SVV views: (Views with *svv_* prefix) They are system views that contain references to STV tables and snapshots for more detailed information.

STL System Tables for Logging

We said earlier that these tables have logs and provide a history of the system. These tables reside on every node in the data warehouse cluster and take the information from the logs and format them into usable tables for system administrators.

STL log tables retain two to five days of log history, depending on log usage and available disk space. For more, you may periodically unload it into Amazon S3.

Let's see below some important ones for an Analyst and reference:

STL_ALERT_EVENT_LOG

Records alert when the query optimizer identifies conditions that might indicate performance issues. You may use the STL_ALERT_EVENT_LOG table to identify **opportunities to improve query performance**.

STL_CONNECTION_LOG

This table logs all connections, disconnections, and authentication attempts.

STL_DDLTEXT

The table holds DDL statements that were run on the system. These DDL statements include:

- CREATE SCHEMA, TABLE, VIEW
- DROP SCHEMA, TABLE, VIEW
- ALTER SCHEMA, TABLE

STL_EXPLAIN

The table contains the EXPLAIN plan for a query that was submitted for execution.

STL_WLM_ Tables

The system tables with the *STL_WLM_* prefix will help you understand better how your workload management strategy works. Read more in the [Workload Management \(WLM\)](#) section of our guide.

STL_QUERY & STL_QUERYTEXT

STL_QUERY returns execution information about a database query. STL_QUERYTEXT returns the query text for SQL commands. These commands are:

- SELECT, SELECT INTO
- INSERT, UPDATE, DELETE
- COPY
- VACUUM, ANALYZE
- CREATE TABLE AS (CTAS)

STL_VACUUM

The table displays raw and block statistics for tables we vacuumed. Read more on it in our [Vacuum Command in Amazon Redshift](#) section.

Of course there are even more tables. So [here](#) is a full list of all the STL tables in Amazon Redshift.

STV System Tables for Snapshot Data

STV are tables with snapshots of the current system state data.

Let's see below some important ones for an Analyst and reference:

STV_EXEC_STATE

Use the STV_EXEC_STATE table to find out information about queries and query steps that are actively running on Amazon Redshift.

STV_LOCKS

Use the STV_LOCKS table to view any current updates on tables in the database.

STV_PARTITIONS

To monitor your current Disk Space Usage, you have to query the STV_PARTITIONS table. For more details check the [Monitoring Query Performance](#) section of our guide.

STV_WLM_ Tables

The system tables with the *STV_WLM_* prefix will help you understand better how your workload management strategy works. Read more in the [Workload Management \(WLM\)](#) section of our Amazon Redshift guide.

Of course there are even more tables. So [here](#) is a full list of all the STV tables in Amazon Redshift.

System Views

The System Views provide quicker and easier access to commonly queried data found in STV and STL tables.

SVV_DISKUSAGE

The SVV_DISKUSAGE view contains information about data allocation for the tables in a database.

SVL_QUERY_SUMMARY

Always keep an eye on the SVL_QUERY_SUMMARY view. If you see queries with the *is_diskbased* field set to true, you might have to revise your Workload strategy and assign more memory to it. Read more in our [Workload Management \(WLM\)](#) section of the guide.

SVV_TABLE_INFO

This is an important system table that holds information related to the performance of all queries and your cluster. SVV_TABLE_INFO contains summary information about your tables. Read more in the [Monitoring Query Performance](#) section of our Amazon Redshift guide.

SVV_VACUUM_PROGRESS

The system view SVV_VACUUM_PROGRESS returns an estimate of remaining time for a vacuuming process that is currently running. Read more on it in our [Vacuum Command in Amazon Redshift](#) section.

Of course there are even more views. So [here](#) is a full list of all the System Views in Amazon Redshift.

System Catalog Tables

System catalog tables have a *PG_* prefix. The system catalogs store schema metadata, such as information about tables and columns. Like PostgreSQL, Redshift has the standard PostgreSQL catalog tables like *pg_namespace* or *pg_group*.

PG_DEFAULT_ACL

The table contains information about default access privileges. For more details go [here](#).

PG_LIBRARY

Stores information about user-defined libraries. For more details go [here](#).

PG_STATISTIC_INDICATOR

Stores information about the number of rows inserted or deleted since the last ANALYZE. The PG_STATISTIC_INDICATOR table is updated frequently following DML operations, so statistics are approximate. For more details go [here](#).

PG_TABLE_DEF

Stores information about table columns. For more details go [here](#).

WORKLOAD MANAGEMENT AND QUERY QUEUES

No matter how big your *Amazon Redshift* cluster is, it will always have a certain amount of resources available. As you start digging further and further into your data, more queries will have to be executed.

Queries that in most cases will have to be scheduled to run at specific time intervals to generate new reports and their execution time, together with the resources needed will vary greatly. As soon as you start using your *Amazon Redshift* cluster in a production environment, you will experience a situation where an ad-hoc query that takes too long to complete might block some important queries that have to generate a report.

So how do you deal with these situations? Fortunately, *Amazon Redshift* is equipped with a mechanism that allows us to manage the resources of a *cluster* by creating priorities queues with specific resources attached to them and then attach queries to these queues. In this way, it is possible to prioritize the execution of our queries taking into consideration their importance against the available resources that we have.

Introducing Amazon Redshift WorkLoad Management

Which is the mechanism that *Redshift* offers for managing the queries that you have to execute on your cluster against the available resources? A few notable things about *Redshift Workload Management*,

- The principal mechanism that allows the management of resources is the definition of priorities queues for the execution of queries.
- By default, *Amazon Redshift* defines two queues
 - *One Superuser queue*, which is reserved only for the superuser role and it **cannot** be configured.
 - *One default user queue*, which by default is configured to run up to five queries concurrently. Keep in mind that as you add new queues, the *default queue* must be the last queue in the *WLM configuration*.

- The maximum concurrency level across **all** your queues is 50 queries, so you cannot run more than 50 queries at the same time.
- You can create up to 8 queues that can run up to 50 concurrent queries on your cluster.

Using Amazon Redshift Workload Management

Every time a user executes a query on an Amazon Redshift cluster, the query is assigned to a query queue. In the default situation, the query will be assigned to the *default user queue* if it is executed by any user without the superuser role. Each queue contains some slots, and it has allocated a portion of the clusters' memory which is divided into the available slots. So what things you can configure on a custom defined queue,

1. *Concurrency level*, which is the number of queries that can run at the same time on a particular queue.
2. *User Groups*, you can specify specific user groups to specific queues, in this way the queries of these users will always be routed to a specific queue.
3. *Query groups*, this is the way that *Redshift* has to route specific queries to specific queues. When you define your query, you attach a specific *Query Group* which is nothing more than a label, and this query will end up in a specific queue for execution.
4. *Memory Percent to use*, Probably one of the most important parameters for a queue. That is the main mechanism that allows us to control how the resources of our cluster are consumed by the queries we execute. With it, we can declare to Redshift the amount of memory that will be available to our queries that run in a specific queue.
5. *Timeout*, inevitably some queries will keep running for a long time, this might block other important queries from running. For this reason, it is important to set up a specific time threshold after which the queries will be canceled.

Working with the Amazon Redshift Workload Management Configuration

All the above-mentioned parameters can be altered by the user. The best way to do it is by:

- Using the Amazon Redshift management console
- Alternatively, by using the Amazon Redshift CLI

Before you start experimenting with the Workload Management configuration, you

should understand how *Redshift* assigns queries to the available queues. The way that **queries get assigned to queues is very well explained on this flowchart** . Make sure that you understand well the rules as it will help you troubleshoot your workload as you start working with a large number of queries on your cluster.

The assignment of queries to specific queues happens either by setting a user group or a query group to your query. The assignment is done by using the **SET** command, and you can see **some examples of queries here** .

The Workload Management Configuration properties are divided into two types, the *static* and the *dynamic*. The main difference is that the dynamic properties do not require from your to reboot your cluster while the static ones do.

- *Static Properties*
 - User Groups
 - User Group wildcard
 - Query Groups
 - Query group wildcard
- *Dynamic Properties*
 - Concurrency
 - Percent of memory used
 - Timeout

Based on the above, it is advised that you define your query and user groups at the beginning and make sure that you do not change them often. Restarting your cluster will add downtime to your services, and it is a time-consuming process.

Useful Notes

Avoid using the superuser queue to execute regular queries; this queue has the highest priority compared to the rest of your queues, so it is better to use it mainly for debugging.

You might have a query that requires more memory than what is shared to the available queue slots. **It is possible to assign more than one slots to a query to overcome this situation**. That is something that can also happen temporarily, for example when you would like to execute a *VACUUM* command on your cluster.

Always keep an eye on the **SVL_QUERY_SUMMARY** view. If you see queries with the *is_diskbased* field set to true, you might have to revise your Workload strategy and assign more memory to it.

Some system tables will help you understand better how your workload management

strategy works.

- **STL_WLM_ERROR**. Contains a log of WLM-related error events.
- **STL_WLM_QUERY**. Lists queries that are being tracked by WLM.
- **STV_WLM_CLASSIFICATION_CONFIG**. It shows the current classification rules for WLM.
- **STV_WLM_QUERY_QUEUE_STATE**. It records the current state of the query queues.
- **STV_WLM_QUERY_STATE**. Provides a snapshot of the present state of queries that are being tracked by WLM.
- **STV_WLM_QUERY_TASK_STATE**. It contains the current state of query tasks.
- **STV_WLM_SERVICE_CLASS_CONFIG**. It records the service class configurations for WLM.
- **STV_WLM_SERVICE_CLASS_STATE**. It contains the current state of the service classes.

CHAPTER TWO - IMPORT AND EXPORT DATA

In this section, we find out how we can:

1. Load data into Amazon Redshift, the different strategies, and mechanisms that the system supports.
2. Work with the data, which might also require exporting the data from *Amazon Redshift* to perform an analysis.

IMPORT AND EXPORT DATA: INTRODUCTION

Data Warehouses, as the name also suggests, are built to store data. For this reason, the process of importing and exporting data is important. Additionally, because of their nature, data warehouses like *Amazon Redshift* are usually populated with information that comes from other heterogeneous systems. A typical scenario might include:

- **Logs**, after they are pre-processed and structured accordingly to the database schema.
- **Transactional Databases**, like a *MySQL* or *PostgreSQL* where transactional data from an application update often.
- **Static Files or Datasets**, like authority files in *CSV* format that your company might maintain.
- **Third Party Cloud Services**, like a *CRM* or *Marketing* platform that your company might use.

Traditionally, *Data Warehouse* systems load data in batches through periodical ETL jobs. Although this is still the most common use case, lately there's an increasing demand for almost near time loading of data into systems like *Amazon Redshift*.

Traditionally, Data Warehouse systems load data in batches through periodical ETL jobs. Although this is still the most common use case, lately there's an increasing demand for almost near time loading of data.

In this section, we find out how we can:

Load data into Amazon Redshift, the different strategies, and mechanisms that the system supports.

Work with the data, which might also require **exporting** the data from *Amazon Redshift* to perform an **analysis**.

LOAD DATA INTO AMAZON REDSHIFT

To carry out the most typical scenario for loading data into *Amazon Redshift*, we use another *Amazon AWS* resource as a staging entity for the process. E.g., you load data into your cluster through **Amazon S3** or **Amazon DynamoDB**. To do that an essential prerequisite is to set up the right permission and roles to make these resources accessible to Amazon Redshift.

Access Rights and Credentials

To grant access to an Amazon Redshift instance to access and manipulate other resources, you need to authenticate it. There are two options available:

1. **Role Based Access.** Your cluster temporarily assumes an AWS Identity and Access Management (IAM) role on your behalf.
2. **Key Based Access.** You provide the Access Key ID and Secret Access Key for an IAM user that is authorized to access the AWS resources that contain the data.

Role Based Access

Role Based Access is the recommended by *Amazon* access option. It is more secure and offers fine-grained control of access to resources and sensitive data. To use a *Role Based Access Control*, you first have to create an IAM Role using the Amazon Redshift Service Role type and attach that role to a cluster. The minimum number of permissions the role must have is the following:

- *COPY*
- *UNLOAD*
- *CREATE LIBRARY*

For a step by step guide on how to create an IAM Role and attach it to a cluster, please see [this guide from Amazon](#).

After the creation of an IAM Role, an ARN for this particular role returns. The *COPY* command that loads data into the cluster uses this ARN as a parameter.

Key Based Access

To gain this type of access, you need to provide the Access Key ID together with the Secret Access Key for an already existing IAM User. That user must have the authorization to access a particular resource, from which data load into the *Amazon Redshift cluster*.

It is strongly suggested to use *Role Based Access* instead of a *Key Base* one, for security reasons. If for any reason you prefer to use the combination of a key and a secret, then make sure you do not use keys that associate with your root account. Instead, create a new IAM User [following this guide](#).

The newly created *IAM User* should have at least the following permissions:

- *COPY*
- *UNLOAD*
- *CREATE LIBRARY*

Importing Data

Amazon Redshift supports loading data from different sources, and the methods to use are mainly two:

1. Use of the *COPY* command
2. Using *DML* * (*INSERT*, *UPDATE*, and *DELETE*) Commands

(* *DML* = *Data manipulation language* - [Wikipedia](#))

As *Amazon Redshift* is built on top of a PostgreSQL clone, you can connect to a cluster using the available JDBC and ODBC drivers and perform queries using *DML* commands. This technique should be avoided, even if it feels familiar to a data analyst.

It is not suggested to use this method of importing data, executing bulk *INSERT* commands over a *JDBC* or *ODBC*, does not perform well with large amounts of data.

On the contrary, using the *COPY* command is the suggested way to load large volumes of data on an *Amazon Redshift* cluster. In this way, we utilize the full

potential of the MPP (Massive Parallel Processing) of Redshift, and data loads more efficiently and faster.

To ensure that the cluster is loading data to its full capacity you should:

1. Split your data into multiple files, e.g. if you are using a source like Amazon S3 or multiple hosts
2. Define the best **Table Distribution Style** for your data

Sources to Load your Data

The *COPY* command supports a wide number of different sources to load data.

Amazon S3

The first and most common source is **Amazon S3**. There you can load data in CSV or JSON serialization.

Amazon EMR Cluster

Another potential source where *COPY* can load data is an **Amazon EMR cluster**. Data on an Amazon EMR Cluster get stored on an *HDFS* file system from which you can load it in parallel into an *Amazon Redshift Instance*.

Remote Hosts

It is also possible to load data (in parallel again) using the *COPY* command, from an arbitrary number of remote hosts. These hosts might be EC2 instances or any other computer connected to the Internet. The remote hosts should be able to allow access through SSH for *COPY* to be able to connect and pull the data.

*For more information on **how to load data from remote hosts, you can check here**.*

DynamoDB

Finally, *COPY* supports **DynamoDB** as an input source. So, it is possible to replicate tables from DynamoDB to tables on *Amazon Redshift*.

*For **a guide on how to sync the two systems using the COPY command, you can check here**.*

COPY command parameters

The *COPY* command takes some parameters you should be aware. The following three are the required parameters:

1. **Data source**, the place where we pull the data
2. **Table Name**, the destination of where we store the data
3. **Authorization**, credentials to access the data on the data source

There are also some optional parameters which might be helpful for optimization reasons and for making your ETL processes more resilient.

1. **Data Conversion Parameters** are necessary for making your data loading process more resilient to errors. *COPY* attempts to implicitly convert the strings in the source data to the data type of the target column. *Using these parameters, you can explicitly define how to perform the conversions.*
2. **Column Mapping**. The default behavior of the *COPY* command is to insert the data into the table, in the same order found in the input source. If you change this behavior, you should provide an optional *column mapping*.
3. **Data Format Parameters**. There are some different options, ranging from CSV to **AVRO files**, that *COPY* understands. Using these parameters, you can provide to *COPY* a different type of data serialization.
4. **Data Load Parameters** define the behavior of the loading process. You can change the way errors are handled and reported or the sample size for the selection of a compression method.

The above parameters make the *COPY* command a very flexible and versatile tool for pushing data into *Amazon Redshift*. Combined with knowledge about the nature and provenance of the data can lead to an optimal method for consistently loading data into an *Amazon Redshift* cluster.

Loading data from other sources

Many other sources are not supported by *COPY* directly. In this case, the recommended approach is to figure out a way to place the data on *Amazon S3* and then use the *COPY* command.

Some notable examples are databases like **PostgreSQL** and **MySQL** that might be running on **AWS RDS** or any other provider. In this case, you need to figure out a way to dump the data from the database and then load it into *S3*.

MySQL

For MySQL you can check the following resources:

You can use *mysqldump* to dump your data into CSV format. [Here you can find out how to convert the output of *mysqldump* into a CSV file.](#)

Here you can also find a [small utility written in *Python*](#) to do the same thing.

PostgreSQL

In a similar fashion, you can use the `psql` command to dump tables into CSV format, using the `/copy` command parameter.

In this post here you can find the appropriate syntax to do it .

The output of the above commands can then be loaded into *S3* and then use the *Amazon Redshift COPY* command to load the data into the cluster.

Cloud Applications

Pulling data out of cloud applications is much more complicated. You need to develop a custom connector for each application to pull data out of it. Then, turn it into an appropriate CSV or JSON file and store it into *S3*.

Alternatively, you can use a service like **Blendo** to automatically sync the data from cloud applications like CRM and marketing platforms into Amazon Redshift.

Streaming data into Amazon Redshift

So far, all the different methods for loading data into *Amazon Redshift* are for updating your cluster in batches. That is a natural choice because traditionally, data warehouses were intended to be used to analyze large amounts of historical data.

Currently, there's an increasing trend towards more real-time analytic applications where data is analyzed in almost real time. To address this new requirement *Amazon Redshift*, currently, supports the ingestion of data in real time using **Amazon Kinesis**.

The way to ingest streaming data into *Redshift* is by using the *Amazon Kinesis Firehose* which automatically batches and compresses streaming data before it loads it into a *Redshift* cluster. In this case, batched data are delivered automatically to *S3* and then into *Redshift* with a *COPY* command that is provided by the user during configuration.

An interesting and useful feature, supported by *Amazon Kinesis*, is the online transformation of data. It is possible to define custom transformations as **Amazon Lambda** functions that transform your data before it loaded on a *Redshift cluster*.

For more information about ingesting streaming into *Redshift*, you can check [here](#) and [here](#).

Useful Notes

Loading data into *Amazon Redshift* is probably one of the most fundamental operations that we perform frequently. It's not uncommon to deploy complex pipelines which load data from many different sources into *Redshift*. Many things can go wrong in such a complex procedure, so here are a few things that might help you figure out what is going wrong.

The data to be loaded should be in UTF-8 encoding. No other encoding is supported, and if you try to load a different one, you might end up with errors. If for any reason you end up having invalid characters in the data you are going to load; then you might find helpful to add the **ACCEPTINVCHARS** parameter to the *COPY* command.

Make sure CHARS and VARCHARS conform to the max size of the responsive columns. The size of strings is measured in bytes and not characters.

If you are using any special characters in your CSV files, make sure to define an escape character as a parameter to your *COPY* command.

If your CSV has headers, use the **IGNOREHEADER** parameter for the *COPY* command.

All errors that occur during the data loading phase are stored into the *STL_LOAD_ERRORS* system table. Consult this table whenever you get an error as it contains useful error messages.

EXPORT DATA FROM AMAZON REDSHIFT

Equally important to **loading data into a data warehouse like Amazon Redshift**, is the process of exporting or unloading data from it. There are a couple of different reasons for this.

First, whatever action we perform to the data stored in Amazon Redshift, new data is generated. This data to be useful and actionable should be exported and consumed by a different system. Data is exported in various forms, from dashboards to raw data that is then consumed by different applications.

Second, you might need to unload data to analyze it using statistical methodologies or to build predictive models. This kind of applications requires from the data analyst to go beyond the SQL capabilities of the data warehouse.

In this chapter, we see how data is unloaded from *Amazon Redshift* and how someone can directly export data from it using frameworks and libraries that are common among analysts and data scientists.

How to Export Data from Redshift

The ***COPY*** command is the most common and recommended way for **loading data into Amazon Redshift**. Similarly, *Amazon Redshift* has the ***UNLOAD*** command, which can be used to unload the result of a query to one or more files on *Amazon S3*.

The data is *unloaded* in CSV format, and there's a number of parameters that control how this happens.

- **Manifest.** This parameter indicates to *Amazon Redshift* to generate a *Manifest* file in JSON format, listing all the files that will be produced by the ***UNLOAD*** command.
- **Delimiter.** Specifies the delimiter to use in the CSV file.
- **Encrypted.** Specifies that the generated on *S3* files will be encrypted using the *AMAZON S3* server side encryption.
- **BZIP2 or GZIP.** Indicates that the unloaded files will be compressed using one of

the two compression methods.

- **NULL.** *NULL* indicates which character to be used to represent *NULL* values. It is important if you perform further analysis on the data.

After a successful invocation of the *UNLOAD* command, the data will be available on *S3* in CSV which is a format friendly for analysis but to interact with the data someone has to access it on *S3*.

How to Read Data from Amazon S3

The *UNLOAD* command gets your data into *Amazon S3* so that you can work with it after its extraction from *Amazon Redshift*. Now you need somehow to interact with *S3* and access your files.

Amazon AWS SDKs

The most common way to do that is by using the *Amazon AWS SDKs*. For a data analyst, the most useful one of the SDKs is probably *Boto3* which is the official *Python* SDK for the AWS services.

Boto3 is a generic AWS SDK with support for all the different APIs that Amazon has, including *S3* which is the one we are interested. Download a file using *Boto3* is a very straightforward process.

```
import boto3
s3 = boto3.resource('s3')

with open('filename', 'wb') as data:
    s3.download_fileobj('mybucket', 'mykey', data)
```

Of course, it is possible to read a file directly into memory and use it with all the popular *Python* libraries for statistical analysis and modeling. It is advised, though, that you cache your data locally by saving into files on your local file system. Otherwise, every run of your program will require downloading the data from *S3* over the network, something that adds additional latency and cost to your operations.

As we see, all it takes to download a file from *S3* is 4 lines of code, and it requires to know the bucket where your files exist, the name of the file you want to download and the key to use as credentials.

R

R is another popular choice for Analysts and Data Scientists when it comes to a language for scientific and statistical computing. Unfortunately, there's no official *AWS SDK* for *R*, but there are a few options out there to help you interact directly with your data stored on *S3*.

A common way for *R* to interact with the *AWS APIs* is by invoking commands directly to the *AWS CLI* using the *R System()* call. To do that, you need to install the *CLI* and then invoke the commands you want, e.g. *get an object from S3*.

Another option is to use the *Cloudyr package* for *S3*. With it, download and working with files on *S3* is just a one line command inside your *R* code. Just make sure that you have configured your credentials correctly for accessing your *Amazon S3* account.

Reading Data directly from Amazon Redshift

So far we have seen how we can unload data from *Amazon Redshift* and interact with it through *Amazon S3*. This method is preferable when working with large amounts of data and you have concluded to the shape of the data that you would like to work.

There are cases where interacting directly with *Amazon Redshift* might be more desirable. For example during the process of exploring your data and deciding on the features that you would like to create out of it.

In this case, waiting to go through *S3* every time you change something to the queries you work with adds much delay to your analysis.

To access your data directly on *Amazon Redshift*, you can use the drivers for *PostgreSQL* that your language of choice has.

Python

For *Python*, you can use *Psycopg* which is the library recommended by *PostgreSQL*. The same can also be used to access your *Amazon Redshift* cluster and execute queries directly from within your *Python* code. After you have established a connection with your *Amazon Redshift*, you can work with the data using either *NumPy* or *Pandas*.

R

In a similar way to *Python* you can also interact with your *Redshift cluster* from within *R*. All it takes is to include the "RPostgreSQL" package to your code, and then you can execute queries directly to your data warehouse and pull data out of it and use it within your code.

Useful Notes

You can find more information here on [how to access your data in Amazon Redshift with Python and R](#).

A useful tutorial for working with PostgreSQL from *Python*.

[Getting started with PostgreSQL in R](#) has some great material that might be helpful also with *Amazon Redshift*.

TOOLS TO CONNECT TO YOUR AMAZON REDSHIFT CLUSTER

Now that we saw how we export data from Amazon Redshift let's see an application of such data. As we said in the previous section, two are the main reasons we need our data.

First, every action we perform to the data stored in Amazon Redshift creates new data. This data can be used raw or by a Business Intelligence tool.

Second, you might need to unload data to analyze it using statistical methodologies or to build predictive models. That means the data analyst should go beyond the SQL capabilities of the data warehouse.

Data management and **ETL as a service tools** like **Blendo** gives you the power to consolidate all your data in Amazon Redshift. The next step is up to your use case, and the tools below can get you covered.

Which SQL IDE is the best to connect to AWS Redshift?

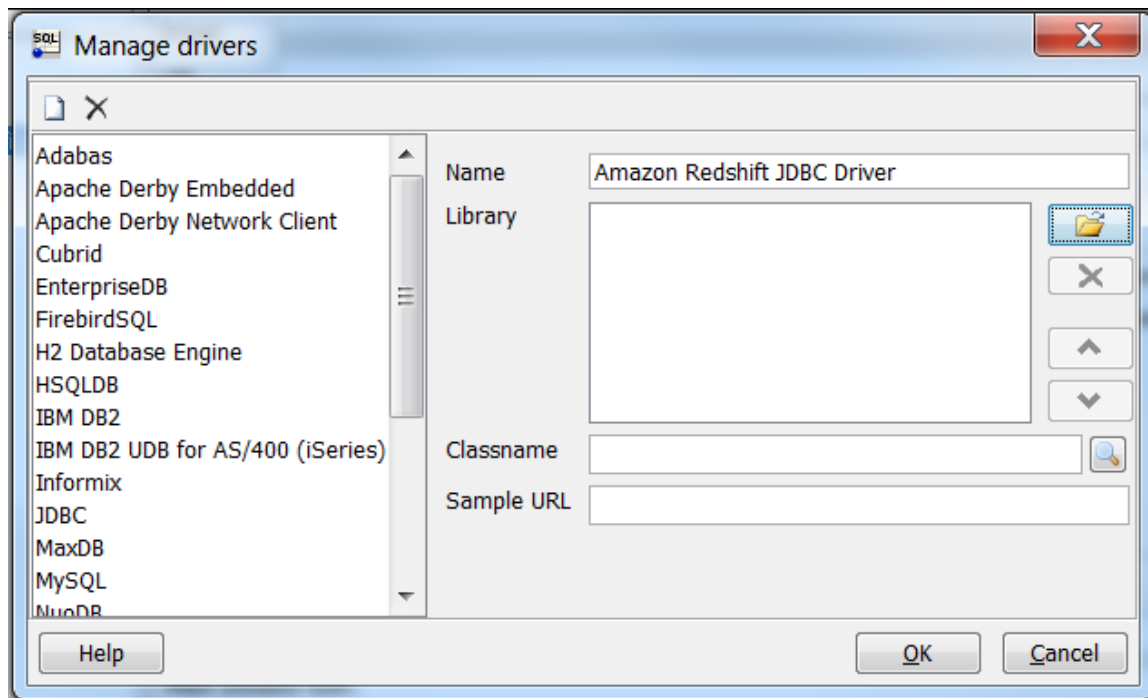
Amazon does not provide any SQL client tools, but they recommend SQL Workbench/J. Though you can connect to your cluster using psql or any SQL IDE that supports PostgreSQL JDBC or ODBC drivers.

SQL Workbench/J

Amazon in their guide uses SQL Workbench/J. It is simple, and it just works!

1. Download SQL Workbench/J from [here](#).
2. Read the Installing and starting SQL Workbench/J page from [here](#).
3. Follow these instructions on how to connect to your Amazon Redshift cluster over

a JDBC Connection in SQL Workbench/J from Amazon [here](#).



psql command line tool

You may also connect with psql to an Amazon Redshift cluster. Psql is a terminal-based front end from PostgreSQL, and it is pretty straightforward to use. You will need your Amazon Redshift cluster endpoint, database, and port to connect.

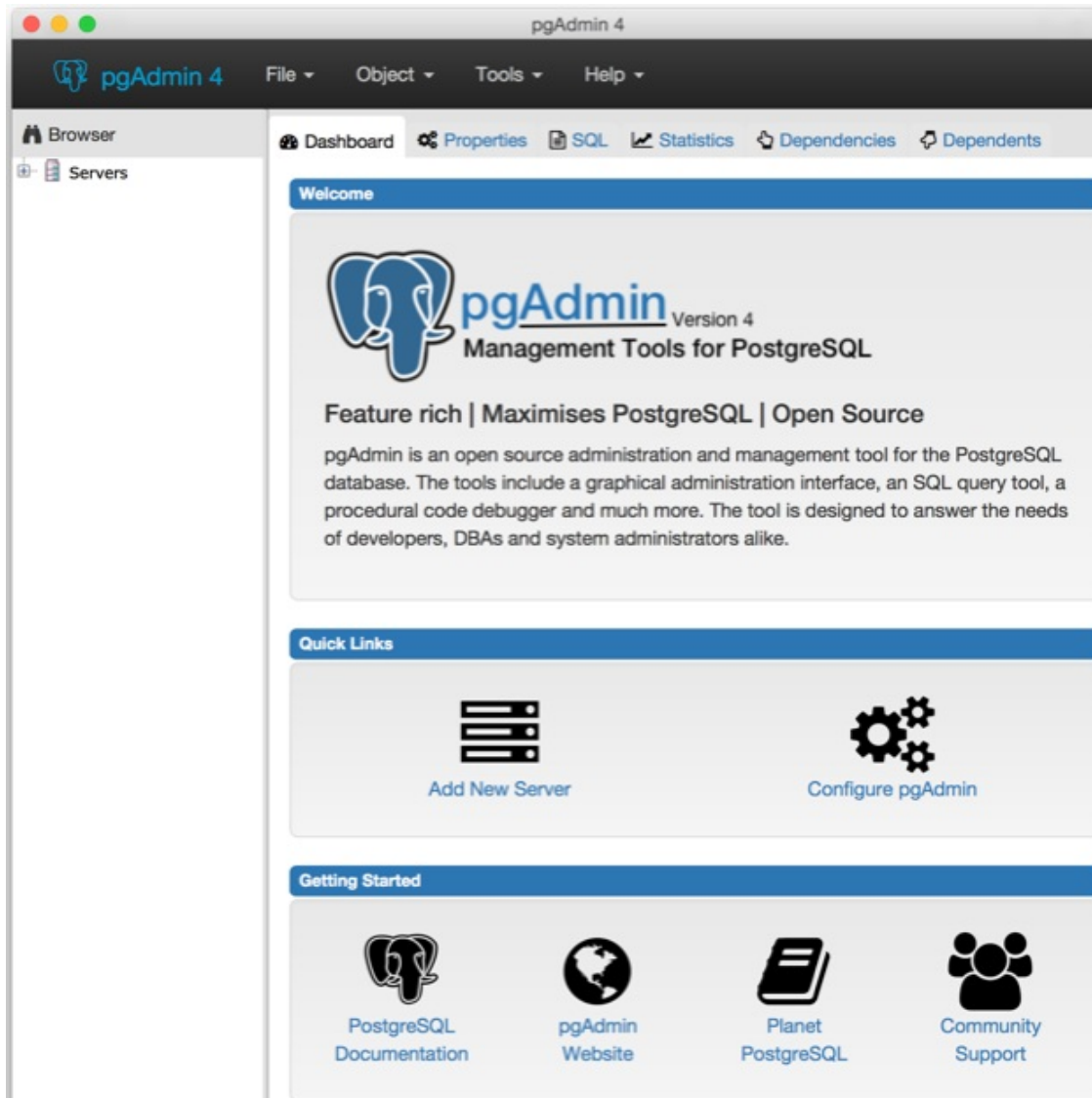
1. Download psql Tool from [here](#).
2. Read the how to use psql from [here](#).
3. Follow these instructions on how to connect to your Amazon Redshift cluster with the psql tool with [this guide](#) from Amazon.

Properties	SSL	SSH Tunnel	Advanced
Name	<input type="text"/>		
Host	<input type="text"/>		
Port	<input type="text" value="5432"/>		
Service	<input type="text"/>		
Maintenance DB	<input type="text" value="postgres"/>		
Username	<input type="text"/>		
Password	<input type="password"/>		
Store password	<input checked="" type="checkbox"/>		
Colour	<input type="text"/>		

pgAdmin

You may also connect with pgAdmin. It is all time classic in the PostgreSQL community and its latest version (as of today) pgAdmin 4, is a complete rewrite of pgAdmin, built using Python and Javascript/jQuery.

1. You can find it [here](#).
2. Documentation is [here](#).



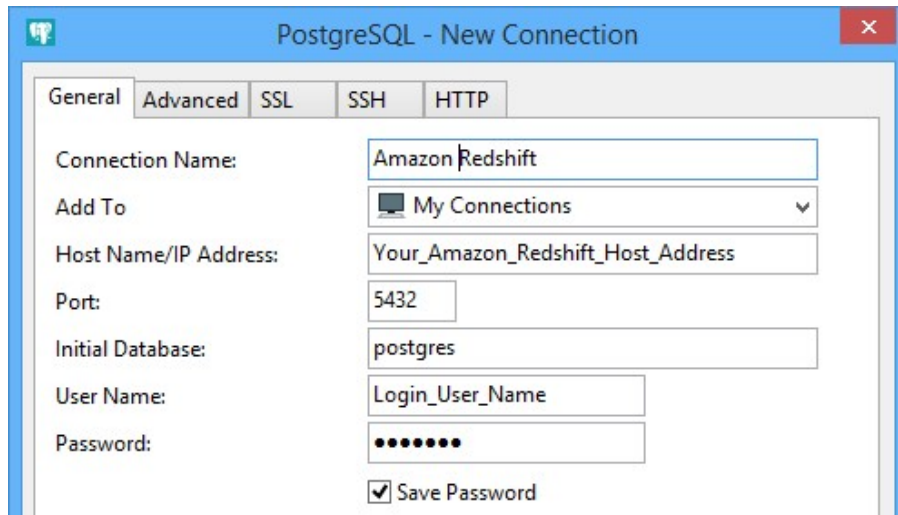
JetBrains DataGrip

1. An excellent SQL IDE from JetBrains. Get it [here](#).
2. Read their [blog](#) for new updates.

Navicat Essentials

Navicat offers many flavors; this is a compact version of Navicat.

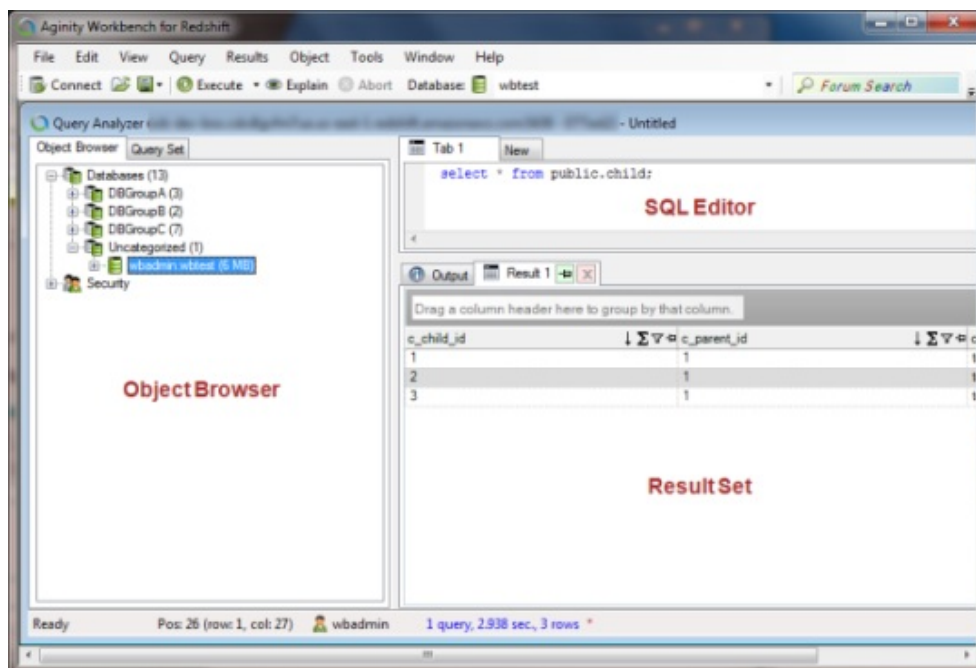
1. Get Navicat from [here](#).
2. Read how to connect to Amazon Redshift [here](#).



Aginity Workbench

A Windows based, Amazon Redshift specific IDE.

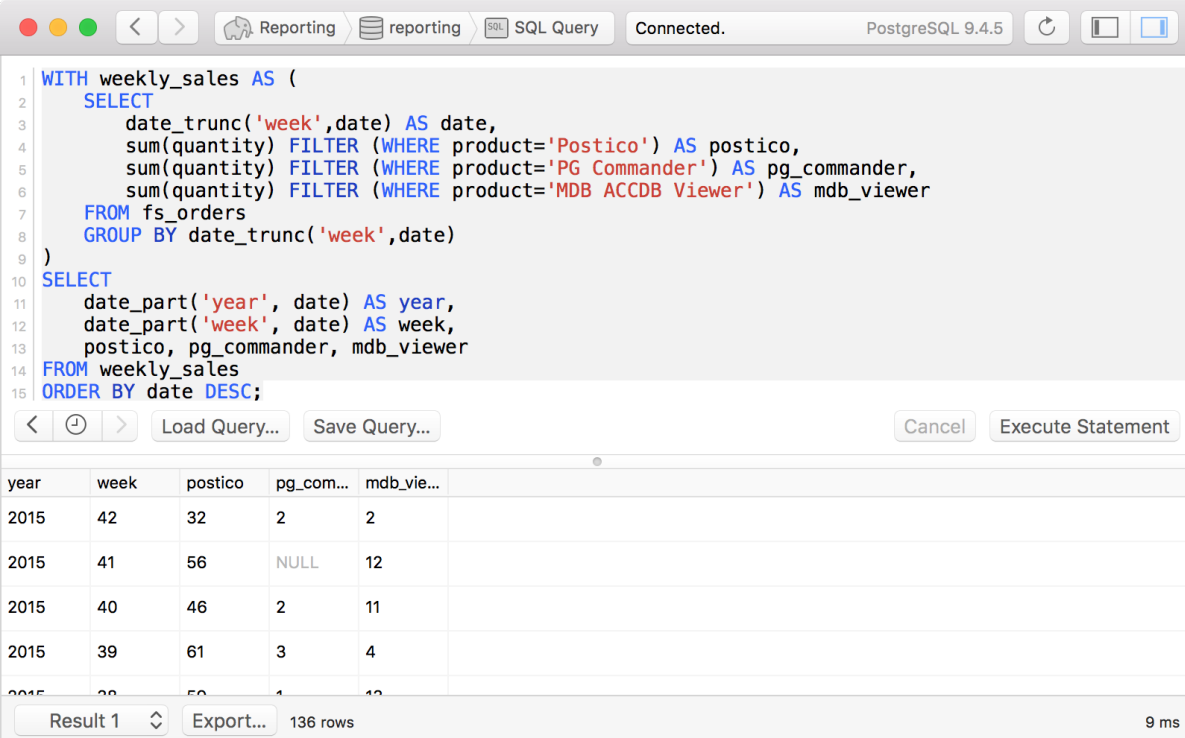
1. Get Aginity Workbench from [here](#).
2. Read how to connect to Amazon Redshift from [here](#).



Postico

It has a simple interface that works nicely. It is a Mac native and supports database systems derived from PostgreSQL like Amazon Redshift.

1. Get Postico from [here](#).
2. Get started guide is [here](#).



The screenshot shows the Postico application interface. At the top, there's a toolbar with icons for Reporting, reporting, SQL Query, and a status bar indicating 'Connected.' and 'PostgreSQL 9.4.5'. The main area displays a SQL query:

```
1 WITH weekly_sales AS (  
2   SELECT  
3     date_trunc('week',date) AS date,  
4     sum(quantity) FILTER (WHERE product='Postico') AS postico,  
5     sum(quantity) FILTER (WHERE product='PG Commander') AS pg_commander,  
6     sum(quantity) FILTER (WHERE product='MDB ACCDB Viewer') AS mdb_viewer  
7   FROM fs_orders  
8   GROUP BY date_trunc('week',date)  
9 )  
10 SELECT  
11   date_part('year', date) AS year,  
12   date_part('week', date) AS week,  
13   postico, pg_commander, mdb_viewer  
14 FROM weekly_sales  
15 ORDER BY date DESC;
```

Below the query editor, there are buttons for 'Load Query...', 'Save Query...', 'Cancel', and 'Execute Statement'. The results are displayed in a table with columns: year, week, postico, pg_com..., and mdb_vie... The table shows data for the year 2015, with weeks 42, 41, 40, 39, and 38. The bottom status bar indicates 'Result 1', 'Export...', '136 rows', and '9 ms'.

year	week	postico	pg_com...	mdb_vie...
2015	42	32	2	2
2015	41	56	NULL	12
2015	40	46	2	11
2015	39	61	3	4
2015	38	50	1	10

Squirrel

It is Java based, Open Source SQL IDE.

You can download it [here](#).

JackDB

JackDB is an SQL IDE that works with PostgreSQL, Amazon Redshift and more.

1. You can find it [here](#).
2. Documentation is [here](#).

It is Open Source universal database tool.

1. You can get it [here](#).
2. Instructions / Wiki [here](#).

We are sure there are more in the vastness of the web. If you want to add one tool just mail us at team@blendo.co

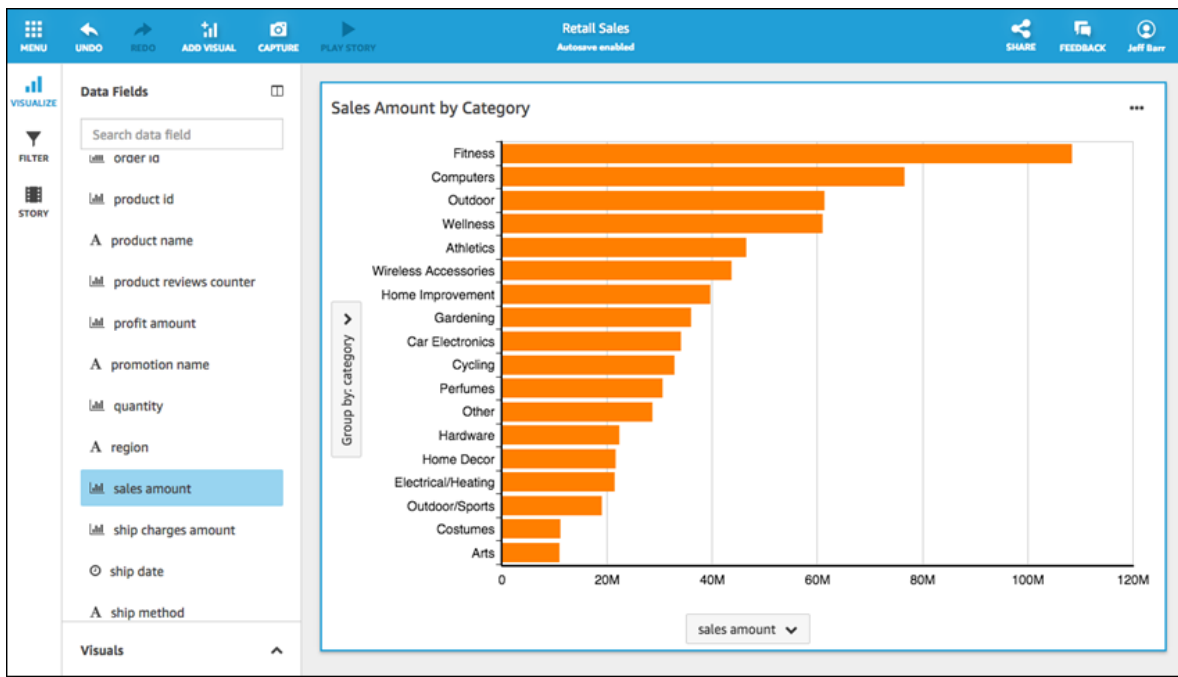
Which BI tools can I use to connect a BI tool to Amazon Redshift?

It is of most importance to get all your data in Redshift first. Then connecting a BI tool in an Amazon Redshift cluster is usually, straightforward. Some time ago we wrote a post with [the ultimate list of custom dashboards and BI tools](#) . The list gets updated with new tools, but for our Amazon Redshift guide, we stay to those that work with specifically with Amazon Redshift.

For this reason, we took as a benchmark the Business Intelligence Partners list in Amazon's Partner Network. Here are some of the most prominent.

Amazon QuickSight

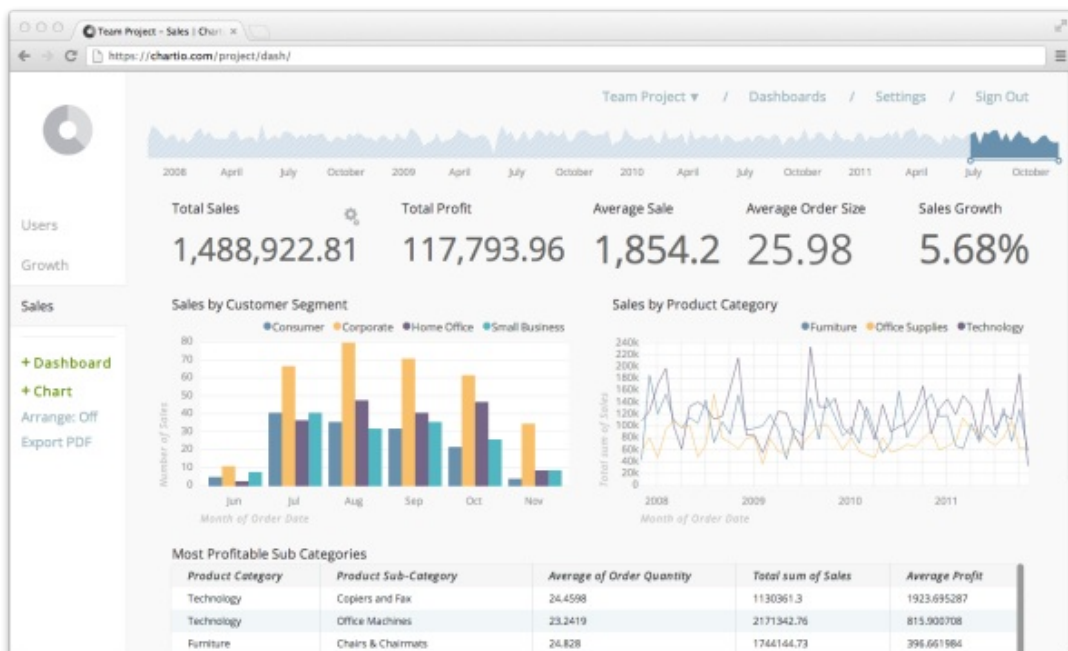
An Amazon product, fast and can connect to all of Amazon's products as data sources like Redshift. QuickSight can access data from many different sources, both on-premises and in the cloud. There's built-in support for Amazon Redshift, RDS, Amazon Aurora, EMR, Kinesis, PostgreSQL, and more.



Chartio

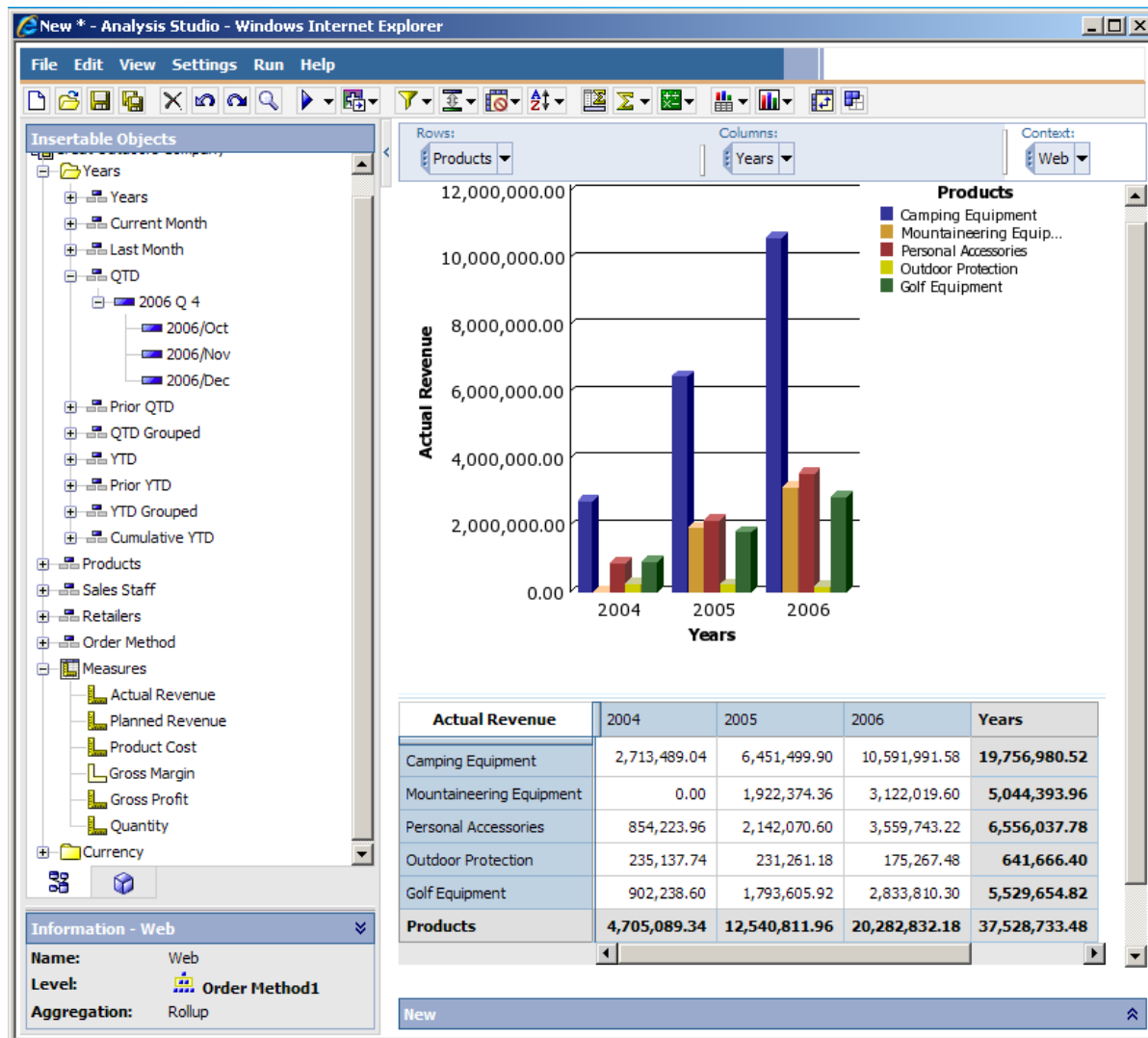
Great BI tool out there and Blendo partner. They provide a comfortable, light-weight tool for data pulls, so either you use SQL or their drag-and-drop Interactive Mode.

*Chartio partners with Blendo. So if you need all your data into Amazon Redshift and then a tool for your visualizations then **Blendo + Chartio** are one of your great choices.*



IBM Cognos Business Intelligence

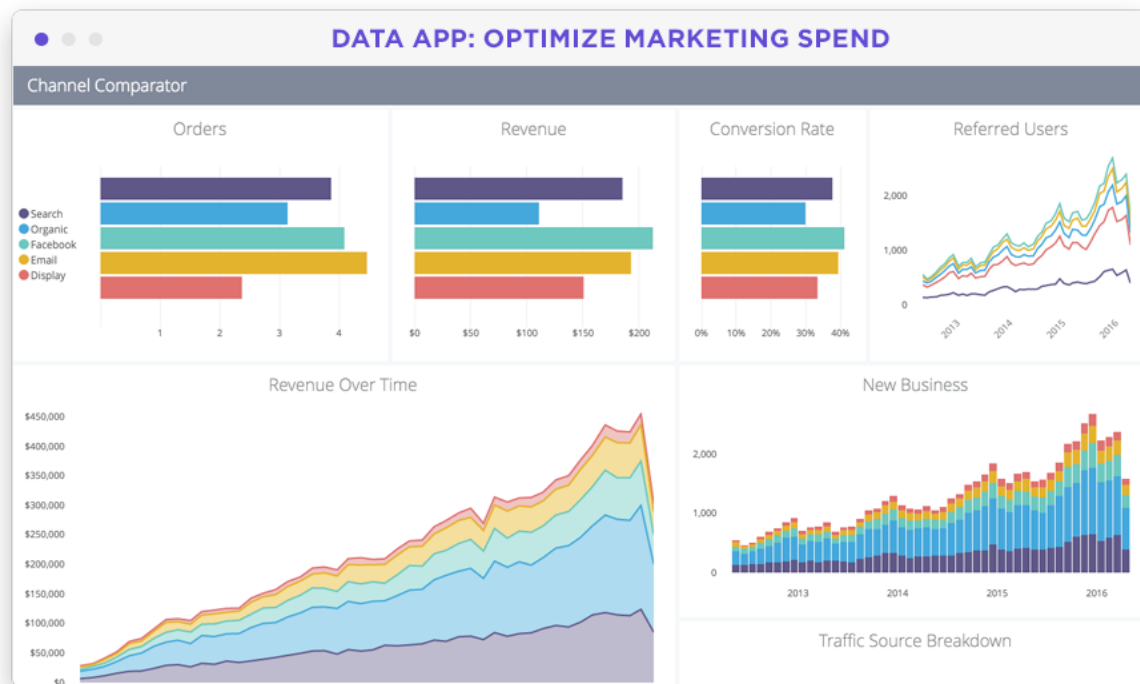
IBM Cognos Business Intelligence is a web-based, integrated business intelligence suite by IBM. It provides a toolset for reporting, analysis, scorecards, and monitoring of events and metrics.



Looker

A lightweight but powerful "data exploration" tool that allows you to "look" into your data without writing any code or SQL. Their LookML is awesome.

*Looker partners with Blendo. So if you need all your data into Amazon Redshift and then a tool for your visualizations then **Blendo + Looker** are one of your great choices.*



Mode

Mode is a data analysis platform that combines a powerful, web-based SQL editor with charting and sharing tools. Their product was built for SQL proficient users.

Mode interface showing a SQL query editor and a table view of the results.

Query:

```

1 SELECT DATE_TRUNC('month', z.occurred_at) AS "Month",
2
3 -- This finds the average user age of each login during the month
4 AVG(z.user_age) AS "Average user age in days",
5
6 -- This calculates how many users from each signup cohort logged in each month.
7 COUNT(DISTINCT CASE WHEN z.activation_month = '2013-07-01' THEN z.user_id ELSE NULL END) AS "Sign
8 COUNT(DISTINCT CASE WHEN z.activation_month = '2013-08-01' THEN z.user_id ELSE NULL END) AS "Sign
9 COUNT(DISTINCT CASE WHEN z.activation_month = '2013-09-01' THEN z.user_id ELSE NULL END) AS "Sign
10 COUNT(DISTINCT CASE WHEN z.activation_month = '2013-10-01' THEN z.user_id ELSE NULL END) AS "Sign
11 FROM (
12 -- This subquery adds the user's age and signup month to each event.
13 SELECT events.*,
14 DATE_TRUNC('month', users.activated_at) AS activation_month,
15 EXTRACT('day' FROM events.occurred_at - users.activated_at) AS user_age
16 FROM benn.fake_dimension_users users
17 JOIN benn.fake_fact_events events
18 ON events.user_id = users.user_id
19 AND events.event_name = 'login'
20 WHERE users.activated_at IS NOT NULL
21 AND events.occurred_at < '2014-06-01'
22 AND events.occurred_at >= '2013-07-01'
23 ) z
24 GROUP BY 1
25 ORDER BY 1

```

Table view (11 rows returned):

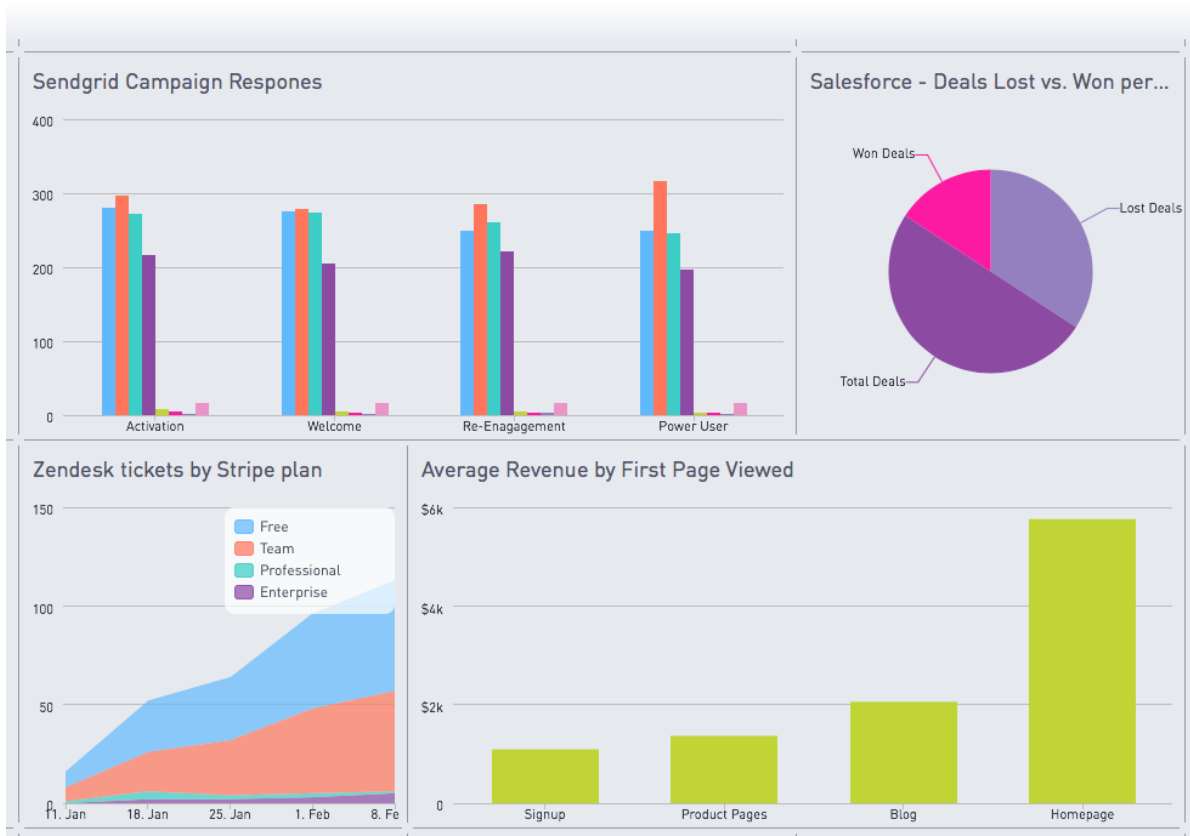
Month	Average user age in days	Signed up in July 13	Signed up in Aug 13	Signed up in Sept 13	Sign
2013-07-01 00:00:00	12.35	676	0	0	
2013-08-01 00:00:00	16.98	102	677	0	
2013-09-01 00:00:00	20.95	25	137	703	

Periscope

Periscope offers an engaging SQL environment for SQL-savvy users.

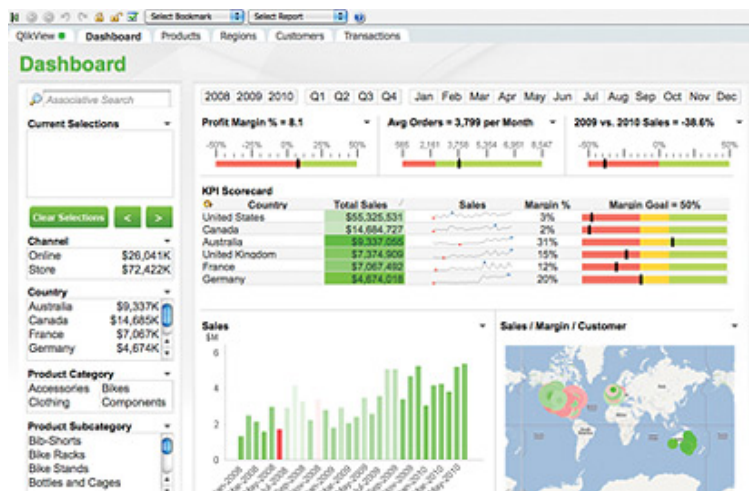


Multi Source Overview



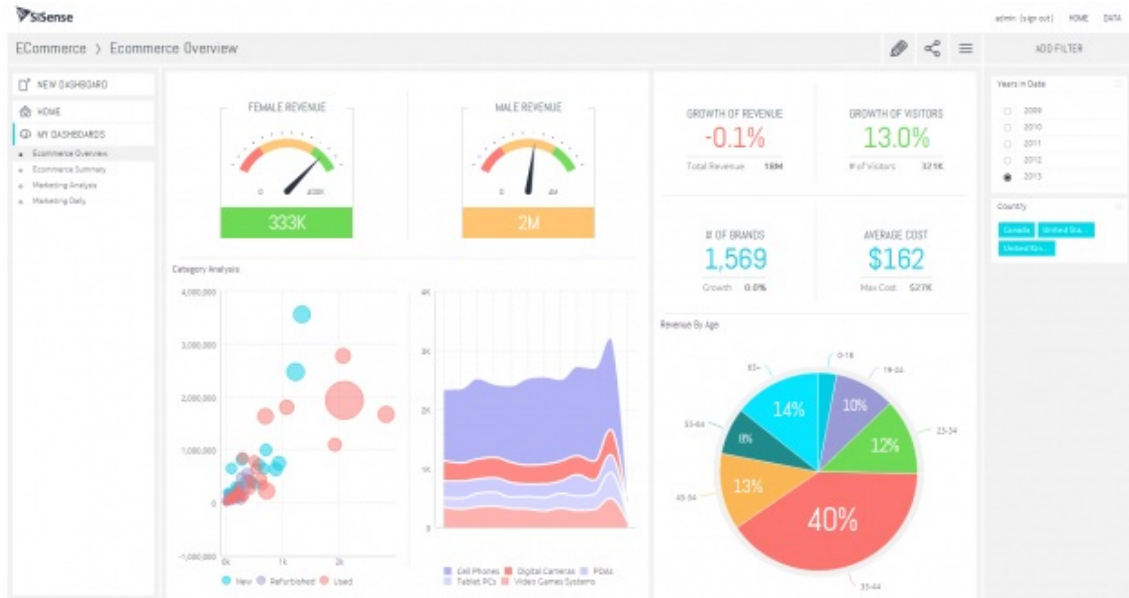
Qlik

QlikView is a great product for Guided Analytics and more data-savvy users.



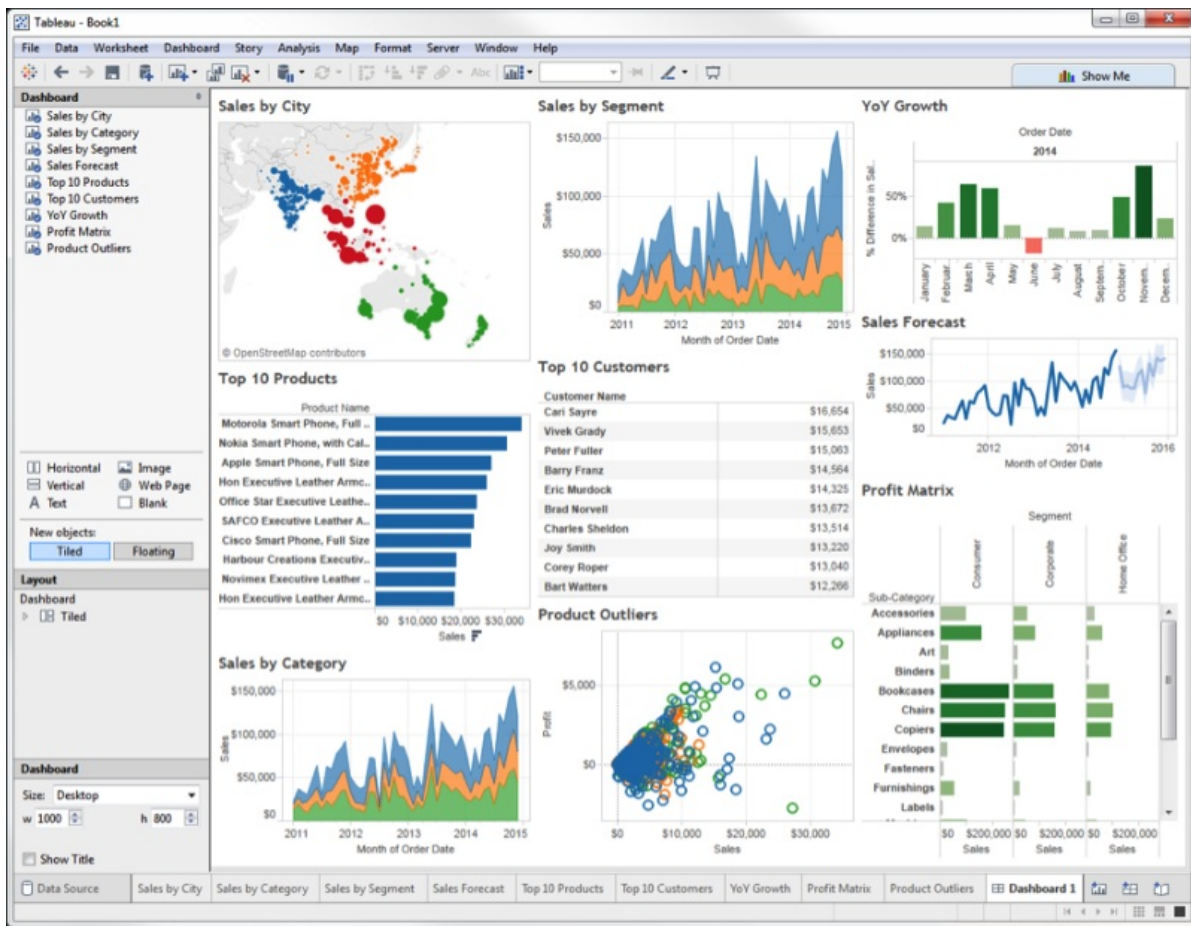
SiSense

SiSense can handle large databases and datasets quite easily, easy to design and customize.



Tableau

Popular and powerful desktop data visualization tool.



Yellowfin

Yellowfin's platform includes notable discovery features and ease of use.



You may find the full **Business Intelligence Partners** list in [Amazon's Partner Network \(APN\)](#).

How can I connect as a Data Analyst / Data Scientist to Redshift?

Amazon Machine Learning

Amazon Machine Learning is a service that makes it easy for developers of all skill levels to use machine learning technology. You can get it from [here](#).

Jupyter

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and explanatory text. More information [here](#).

Python

Python is widely used in Data Science projects. More info on Python [here](#) and a tutorial on [how to connect to Amazon Redshift and PostgreSQL with Python](#) .

R

R along with Python are the most commonly used programming languages in Data Science today. [Here](#) is the official site of R. Here is a tutorial on [how to connect to Amazon Redshift and PostgreSQL with R](#).

Julia

Julia is a high-level, high-performance dynamic programming language for numerical computing. It provides a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library. More information [here](#).

Matlab

A platform used by millions of engineers and scientists worldwide to analyze and design systems and products. Get it [here](#).

CHAPTER THREE - DATA MODELING AND TABLE DESIGN

In this section, we will see the main differences that an *OLAP* system like *Amazon Redshift* has, compared to *OLTP* systems and most importantly how these differences affect the way we design a database.

We go through the important concept of Distribution Keys, Sort Keys and Materialized Views.

DATA MODELING AND TABLE DESIGN: INTRODUCTION

Amazon Redshift Data Warehouse is a **relational** database system. It is based on *PostgreSQL* from which it inherits a number of design characteristics.

As a relational database system, Redshift organizes the information in tables. However, at the same time, as a high performance, distributed data warehouse system, *Amazon Redshift* is optimized to execute complex queries over huge amounts of data. For this reason, there are important differences between *Amazon Redshift* and other common database systems like *PostgreSQL*.

In this section, we will see the main differences that an **OLAP** system like *Amazon Redshift* has, compared to **OLTP** systems and most importantly how these differences affect the way we design a database.

We go through the important concept of **Distribution Keys**, which is essential for designing a performant database. Both the nature of the data and the queries that the analyst plans to execute, influence significantly the choice of Distribution Keys.

Sort Keys is the second important design concept that affects the tables of a database significantly, and it contributes to the performance of the queries to perform. In this section, we will see how to use the knowledge of a data analyst to influence the selection of proper *Sort Keys* for your tables.

At the end of this section, we see *Amazon Redshift's* support for **Materialized Views** and how these views can improve query performance.

There's also another parameter of a table on *Amazon Redshift* used as part of table design. The parameter is **Column Compression Type**, but we decided to cover this in the context of the **Maintenance Section** in our guide.

*There's also another parameter of a table on Amazon Redshift used as part of table design. The parameter is **Column Compression Type**, but we decided to cover this in the context of the **Maintenance Section** in our guide.*

Amazon Redshift and Table Constraints

Anyone familiar with SQL will be aware of the following constraints:

- Uniqueness
- Primary Key
- Foreign Key
- Not Null Constraint

The above are commonly used in database design and also exist on *Amazon Redshift*. The main difference is that **except the *Not Null Constraint***, the rest are not enforced by *Amazon Redshift*.

They exist as metadata for informational purposes and help the query planner to optimize the query execution. So you should define them but keep in mind that the system will not enforce these constraints. That means you have to rely on your *ETL* infrastructure for their enforcement. **Blendo** as an **ETL as a service platform**, for example, can take care of these constraints when it loads data into an *Amazon Redshift* cluster.

It is **crucial** to remember that if you use any of these constraints, you have to be certain your data do not violate them. The query planner assumes that the constraints are respected, and you might end up with wrong results to your queries.

*It is **crucial** to remember that if you use any of these constraints, you have to be certain your data do not violate them. The query planner assumes that the constraints are respected, and you might end up with wrong results to your queries.*

TABLE DISTRIBUTION STYLES

Amazon Redshift is a distributed relational database system capable of performing queries efficiently over petabytes of data. That is achieved by the combination of highly parallel processing, columnar design and targeted data compression encoding schemes.

In this section, we see how tables can be optimized to leverage the highly parallel nature of *Amazon Redshift* by defining **Distribution Keys** and selecting the appropriate **Table Distribution Style**.

Parallelization on Amazon Redshift

Distribution and parallelization on *Amazon Redshift* happen on two levels. What follows is a brief description of the high-level architecture of *Amazon Redshift*. For more information check this [Section](#) and also the [Amazon documentation](#).

Amazon Redshift consists of a number of nodes where each one is an actual computing node on *Amazon AWS*, with dedicated resources like memory, CPU and disk space. These nodes create an *Amazon Redshift Cluster* over high-speed network connections.

One node is denoted the *Leader* and the rest act as *compute* nodes. The leader is responsible for distributing the data across the different *compute nodes* to achieve the desired performance.

Each *compute node* has its disk storage divided into *slices*. Although the number of slices varies depending on the type and size of a node, there are at least two of them on each *compute node*. Data distributed on slices is queries in parallel on each *compute node*.

When a table is created, and data is loaded into it, *Redshift* distributes its data across the different *compute nodes* and *slices* of a cluster. This distribution of data happens according to a *Distribution Plan* that can be selected by the user. When we perform a query, the Query Optimizer figures out if data has to be moved from one

node or slice to another for the query to execute successfully.

This redistribution of data across the cluster might account for the poor performance of your queries if the distribution plan that was selected distributed the data in a way that is not optimal for the query execution.

This redistribution of data across the cluster might account for the poor performance of your queries if the distribution plan that was selected distributed the data in a way that is not optimal for the query execution.

Keep in mind that redistribution might include the relocation of a small number of rows, up to the copying of entire tables across all the nodes.

When you decide which distribution style to use for each table, keep the following in your mind:

- When data is distributed unevenly across your computing nodes, some of them have to work more than the other. That is not desirable; you do not want underutilized *compute nodes*. So you should aim for an as much as a uniform distribution of your data.
- Think of the analysis and the queries you plan to execute. A data analyst can help tremendously in the design of the database here. If tables have to move around your cluster, then you should probably reconsider your *distribution style*.
- Distribution is per table. So you can select a different distribution style for each of the tables you are going to have in your database.

Distribution Styles

Amazon Redshift supports three distinct table distribution styles.

- **Even** Distribution. This is the **default** distribution style of a table.
- **Key** Distribution.
- **All** Distribution.

Even Distribution

In *Even Distribution* the *Leader* node distributes the data of a table evenly across all slices, using a round robin approach.

Key Distribution

In *Key Distribution*, data is distributed across slices by the *Leader node* matching the

values of a specific column. So all rows with the same value in the selected column will end up at the same slice.

All Distribution

With *All Distribution*, Leader node distributes the table to all the *computing nodes*. A copy of the table is maintained on all nodes which means that the storage needed for the table is multiplied by the number of nodes. As such, each operation of this particular table takes longer as it has to repeat on all nodes.

To view the *distribution style* of a table, you need to query the *PG_CLASS* system catalog table. The *RELDISTSTYLE* column contains the information of the table distribution style.

1. 0 for *EVEN* distribution style
2. 1 for *KEY* distribution style
3. 8 for *ALL* distribution style

Use the following query to find the *distribution style* of your tables.

```
select relname, reldiststyle from pg_class
where relname like 'YOUR_TABLE_NAME';
```

A table's *Distribution Style* cannot change after its creation. If you decide at some point that a different style is preferable, you should perform a *DEEP COPY* of the entire table.

A table's Distribution Style cannot change after its creation. If you decide at some point that a different style is preferable, you should perform a DEEP COPY of the entire table.

Choosing Distribution Styles

Choosing the right *Distribution Style* for each of your tables might be a tricky process. At this point, we discuss some general rules that might help you to decide which *distribution style* to choose for your tables.

Before you start thinking about the table *distribution style*, you can decide upon the primary and foreign keys. Even if *Redshift* does not enforce these constraints, it helps the query planner to optimize the queries, and also it helps you understand better what queries you are going to use.

This knowledge can also help you choose the right *distribution style*. Of course, as the selection of a *distribution style* is highly sensitive to the queries that will run on the cluster, the input of a data analyst is valuable, let's see why.

If a table does not participate in any *JOIN*, it is highly denormalized, and in general, you cannot think of a specific *distribution style* for it, chose the *EVEN* style.

If you have small tables, that do not often change, for example, authority tables, then you should choose *ALL* style. Think for example a table with country names and country codes; you might be joining this table to get the country name out of the code. Such tables are best to copy over the whole cluster.

If a table participates in *JOINS*, it might be beneficial to select a *KEY* distribution. Depending on how the table participates in the *JOIN*, you should distribute it either on the primary or the foreign key of the table. Keep in mind that joins happens in pairs, and so you should also consider the other joining table and its *distribution style*.

In general, you should strive to select a distribution style that evenly distributes the data across all worker nodes. An *EVEN* style is not always an option as *JOINS* are involved, but your data analyst is your best friend here. She should have a good idea beforehand on the nature of the data that each candidate for a *Distribution Key*.

You are looking for columns that contribute to a uniform distribution, so columns with "heavy hitters" are to be avoided.

Using the Query Planner for Optimizing the Distribution Style

The above rules of thumb might be helpful when you work with a small database and simple queries, but as the queries get more complex and there are multiple *JOINS* involved, it becomes more and more difficult to figure out the right *distribution style* based on the queries performed.

In this case, the *PLAN* of a query is reported by using the *EXPLAIN* command.

EXPLAIN command

When you execute a query together with the *EXPLAIN* command, you get back a plan on how the query is going to be executed. Inside this plan, there's useful information that can be used to decide if there's space for improvement by selecting different *distribution styles*. Each step of the plan comes with a label that can help identifying

issues related to the distribution style.

DS_DIST_NONE and **DS_DIST_ALL_NONE** labels indicate that no redistribution is required for the execution of the step. This is good as no re-evaluation of the distribution style is suggested.

DS_DIST_INNER label is an indication that the redistribution cost might be high as the as the inner table has to be redistributed.

DS_DIST_ALL_INNER indicates that you should probably use a distribution key or even style for the outer table of a join.

DS_BCAST_INNER and **DS_DIST_BOTH** are not good and indicate that you should consider changing the distribution key of the outer table or use an *ALL* style for the inner table.

Useful Resources

Amazon provides an excellent plan on how to test your *Distribution Styles* using samples of your actual data.

It also hosts an extensive documentation [reference for Distribution Styles](#).

UNDERSTANDING AND SELECTING SORT KEYS

"Any inaccuracies in this index may be explained by the fact that it has been sorted with the help of a computer"

Donald Knuth *Sorting and Searching*, (Addison-Wesley, 1973)

The above is a small in-joke you may find in the Index of the famous work of Donald Knuth (these jokes were replaced in later publications). Jokes aside, sorting is necessary, as it helps to find the information we are looking for fast. Machines are exquisite in sorting data, but it is the responsibility of a human to select what we should sort and for what reason.

In this section, we discuss why *sorting* is necessary on *Amazon Redshift*, how it affects query performance and how an analyst can leverage *Sort Keys* to optimize the performance of an *Amazon Redshift cluster*.

What are Sort Keys?

When you create a table, you can optionally define one or more columns as *sort keys*. These columns are being used as data is loaded into the table to sort it accordingly. During this process some metadata is also generated, e.g. the *min* and *max* values of each block are stored and can be accessed directly without iterating the data every time a query executes.

These metadata pass to the query planner which in turn exploits this information to generate execution plans that are more performant.

Based on the above it becomes obvious that *Sort Keys* is an important performance tuning parameter of our tables that,

- It can improve *query performance* and
- Its tuning depends heavily on the *queries* we plan to execute and thus to go through the analysis to be performed by the *analyst* is important in finding the

most efficient *Sort Keys*.

Sort Key Types

Amazon Redshift supports two different types of *Sort Keys*, **Compound Sort Keys**, and **Interleaved Sort Keys**. Selecting the right kind requires knowledge of the queries that you plan to execute.

Compound Sort Keys

Compound Sort Keys, are made up of all the columns that are listed in the sort key definition during the creation of the table, in the order that they are listed. The order is important, as the performance decreases when queries depend on the secondary sort columns.

When you define a Compound Sort Key, make sure to put as first in the list, the most frequently used column in your queries.

When you define a Compound Sort Key, make sure to put as first in the list, the most frequently used column in your queries.

Compound Sort Keys work best in situations where the **query's filter applies conditions**, which use a prefix of the *sort keys*. Thus, they can improve the performance of queries with the following operators.

- Joins
- GROUP BY
- ORDER BY
- Window functions with
 - PARTITION BY
 - ORDER BY

Interleaved Sort Keys

Contrary to *Compound Sort Keys*, *Interleaved Sort Keys* put an equal weight to each of the included columns in the sort key. If there's no *dominant* column in your queries, then you might get improved query performance by creating an *Interleaved Sort Key*.

Notably, in the case where a query uses restrictive predicates on secondary sort columns, *Interleaved Sort Keys* might significantly improve query performance.

As a case of a restrictive predicate, consider a *WHERE* clause you filter your data using an equality operator. E.g.

```
SELECT email from users WHERE name = 'John'
```

Interleaved Sort Keys are *more efficient* with large tables. To find out if a table is a good candidate for using them, you can query the **STV_BLOCKLIST** system table. What you are looking for, is tables with a high number of 1MB blocks per slice and distributed over all slices if possible.

Another example where you might want to consider an *Interleaved Sort Key* is when you plan to sort over only one column. In this situation, it might give better query performance if the column values have a long common prefix. Again, the nature of the data is important here, so the knowledge of a *data analyst* might help to figure out if it makes sense to use one or the other *Sort Key Type*.

How to select Sort Key Types

To summarize all the above information, when you choose *Sort Keys* for your tables, keep the following in mind:

1. Do you plan to use **more than one columns** as *Sort Keys*?
 - NO. Then *Interleaved Sort Keys* might work better.
 - YES. Is there a dominant column appearing in your queries?
 - NO. Then again you should consider *Interleaved Sort Keys*.
 - YES. Then use *Compound Keys* and make sure that dominant column is **first** in the column list.
2. Do you have **highly selective restrictive predicates** in your queries?
 - YES. Consider *Interleaved Sort Keys*
 - NO. If your queries include JOINS, GROUP BY, ORDER BY and window functions with PARTITION BY or ORDER BY. Then consider *Compound Keys*
3. Do you work with **large tables** (make sure to check table statistics)?
 - YES. Consider using *Interleaved Sort Keys*
 - NO. *Compound Sort Keys* might work better

Finally, it is important to know that as you load more data on your sorted tables, performance deteriorates over time. To fix this problem, you read here *how to VACUUM your tables*.

When you select your *Sort Keys*, you need to understand that

- *VACUUMING your tables is unavoidable, and you have to consider the performance hit that this has to your operations.*
- The *Sort Key Type* affects the performance of your *VACUUMING* process.

In general, *Interleaved Sort Keys* are more sensitive to *VACUUMING*, and usually, it takes longer to perform it on tables that have this kind of *sort keys* defined. As it is preferred to use this type with large tables, the result might be long *VACUUMING* times.

So, plan accordingly and make sure you have a good understanding of your data as an analyst and consult the *SVV_INTERLEAVED_COLUMNS* table for vital statistics on your tables that help you figure out the best possible *Vacuuming* strategy.

Useful Resources

As always, it helps to *start with the documentation* that is provided by Amazon.

It also always helps to run tests with your actual data before you come up with a good scheme of *Sort Keys*. *Here you can find an example of how to do this* .

Sort Keys are just one key ingredient of performance tuning on *Amazon Redshift*. Make sure you read about *Distribution Keys*.

Also, you should periodically re-assess the validity of your choices regarding *Sort Keys*. Check the section on *Maintenance* about *Monitoring Query Performance* to see what else you should be taking care of on a periodical basis.

TABLE VIEWS IN AMAZON REDSHIFT

Creating views on your tables is a great way of organizing your analysis. Especially in OLAP systems like *Amazon Redshift*, the analysis usually performed is quite complex and using views is a great way of organizing long queries into smaller parts.

In this chapter, we explore the mechanism for table views of *Amazon Redshift*, its limitations and possible workarounds to obtain the benefits of *materialized views*.

Create Table Views on Amazon Redshift

Creating a view on *Amazon Redshift* is a straightforward process. You just need to use the **CREATE VIEW** command. The command takes as a parameter the query that you wish to use for the view and some other options:

- A **Name** which is the name of the view/table it is going to be created. If you prefix the name with a #, the view is temporary and exists only for the current session.
- **column_name**, you can optionally rename the columns to be created. By default, the names derive from the original table.
- **OR REPLACE** which tells *Redshift* what to do if a view with the same name already exists. It is replaced only if the query is different.

Limitations of Redshift Table Views

Views are coming with some restrictions on *Amazon Redshift* with the most notable being the following:

You cannot DELETE or UPDATE a *Table View*. It looks like a table from a SELECT query, but you cannot affect its data.

The *Table View* is not physically materialized, which means that there's no actual table created in the database. Instead, the query executes every time you request access to the *Table View*.

Table Views reference the internal names of the tables and columns and not what is visible to you. That might lead to fragmentary views and queries.

The query planner is having issues optimizing queries over views.

Overcoming the limitations of Table Views on Amazon Redshift with Materialized Views

There is a way to overcome the above limitations of *Amazon Redshift* and its *Table Views*. The way to do it is by emulating *Materialized Views* on your cluster. To do that, you create actual tables using the queries that you would use for your views. That is possible on *Amazon Redshift* as you can invoke the **CREATE TABLE** with a query as a parameter which defines the table using other tables.

In this way, you are manually creating *Materialized Views* on your system, and you can overcome the limitations we mentioned earlier. Now it is possible to UPDATE and DELETE your views, and the query planner optimizes your queries.

The main issue with this approach is that you have to maintain the *Materialized Views* on yourself. So whenever a change occurs on one of your tables, you need to update your *Materialized Views* by dropping and Recreating the tables.

In an *OLAP* system like *Amazon Redshift*, data does not change that often and thus maintaining these tables as *Materialized Views* should not be a big problem.

Nevertheless, as your tables increase in numbers and your queries get more complex, it would be best to rely on your ETL solution to handle these views.

As your tables increase in numbers and your queries get more complex, it would be best to rely on your ETL solution to handle these views.

ETL as a service Platforms like **Blendo** make it easier to manage a large number of *Materialized Views* as part of your ETL pipeline.

MODELING TIME SERIES DATA

Working with time series data is an important part of the job of a data analyst. Especially as someone delves into behavioral data, the temporal dimension cannot be ignored.

Data like the interactions of a customer with a product over time, the behavior of mail recipients to campaigns and the behavior of buyers on an e-commerce site, can be perceived as time series.

Usually, time-series data are characterized by their volume, e.g. sensor data or log files. When someone is looking for a solution to work with large amounts of time series data, then Redshift always comes up. The reason is the parallel nature of *Amazon Redshift* and its ability to handle a large volume of data.

At this section, we see how time series data can be efficiently stored and queried on *Amazon Redshift*

Storing Time Series Data on Amazon Redshift

Traditionally, storing time series data in a relational database involved the **partitioning** of the database. Partitioning is the process of logically splitting a larger table into smaller physical ones.

There are many reasons, mainly performance related that someone would partition their tables. However, it also makes much sense to do it when you have to deal with time series data. In this case, you are partitioning your data based on time, e.g. on a monthly basis or even a daily basis, depending on what kind of analysis you intend to perform and the amount of data you have to deal.

For an interesting article on how to work with *partitions* on PostgreSQL, you can check [here](#).

Partitions are also used to implement retention policies, so, for example, it is possible to retain only the latest 30 days of data. Retention is necessary with *time series* data,

as their volume might be enormous and after some time not relevant to the kind of analysis a data analyst wants to perform.

The problem here is that *Amazon Redshift* does not support table partitioning in the traditional way that other RDBMS systems do. So, using table partitioning to work with time series data is not possible. Instead, you will need to implement the partitioning logic into your *ETL* process.

Retention period

As *Amazon* also recommends, if you plan to work with *time series* data and you have a fixed retention period then you should organize your data around tables that correspond to that retention period. For instance, if you plan to retain your data on a monthly basis, then you should have one table per month. If you intend to have a yearly retention, then you should have one table per year, and so forth.

There are some pros to this approach.

First, it is **easy** to **delete** old data, as you only have to drop your tables. Keep in mind that deletion is not an easy task on *Amazon Redshift* and if you do it regularly, then *Vacuuming* will be required at some point.

Then, you can use a *UNION ALL* view and have a unified view of all your time series data without having to deal with many different tables and joins in your queries.

Finally, if you have a *timestamp* column in your data and you use it as a *sort key*, you effectively store your data in sort key order, which eliminates the need for *vacuuming*, which is always a good thing.

The above table design emulates what the *partitioning* techniques do on a traditional RDBMS where each table on *Amazon Redshift* corresponds to a partition and the *UNION ALL* view to the original table. The difference here is that how the data will be partitioned on different tables is something that has to be implemented by you.

It is important to remember that whenever you drop old data, you need to renew your *UNION ALL View* to ensure that it reflects your current data, as views do not automatically update by *Amazon Redshift*.

It is important to remember that whenever you drop old data, you need to renew your UNION ALL View to ensure that it reflects your current data, as views do not automatically update by Amazon Redshift.

In case that you do not wish to maintain multiple tables, you can also have only one and enforce a retention policy, e.g. assume that you want to have a weekly retention policy.

Then you have only one table and when the week ends, you should:

- Rename your table
- Create a new one using the original name
- Insert any data from the old table that is inserted after the end of the retention period.
- Drop the old table

Be careful and do not forget to do all the above steps into a *TRANSACTION*.

Be careful and do not forget to do all the above steps into a TRANSACTION.

Sort Keys and Distribution Keys for Time Series data

Previous chapters have explained why *selecting the appropriate sort keys*, and *distribution keys* is an important process for maximizing the value you can get from your *Amazon Redshift* cluster. That is equally important for time series data.

Selecting a distribution key

Selecting an appropriate *distribution key* for time series data is an easy task. Whenever you have a timestamp column, and you should have one as you are dealing with *time series* data, you should use that as your *distribution key*. Timestamps, ensure the uniformity of such a key which is a desirable attribute for *Amazon Redshift*.

In this way, you can be sure that your data will not be skewed and you will end up with balanced processing on your cluster.

Selecting a *sort key*

Sort Keys are a bit more complicated, but again you should exploit the temporal

nature of your data. First, if you have a timestamp column, you should consider using it as a *Sort Key*. By doing this, you also get the benefit of not having to perform *Vacuuming* as we mentioned earlier.

Additionally, having the timestamp as a *sort key* will also benefit the performance of your queries. It is most likely that if you plan to work with time series, you have to use time time-related columns in your queries.

The type of the *Sort Key* is not affected by the temporal nature of your data but you should consider the type of analysis that you plan to do. Check also the **section on *Sort Keys*** for guidelines on [selecting the type of *Sort Key*](#) for your time series tables.

Examples of Time Series Data

A typical example of time series data that you might want to load on your *Amazon Redshift* cluster, is data coming from [Mixpanel](#). This data correspond to events of users from your product and is an excellent example of a time series. You can also read a more elaborate analysis of [how to work with this time series data on Redshift](#) .

Data coming from *Mixpanel* has a quite simple format, what we are interested in, is the following two attributes.

- **properties_time**: The time the event was triggered
- **event**: the event that was triggered at that time

The data that is coming from *Mixpanel* can be stored into different tables using the retention strategy that we want, e.g. monthly or weekly, by following the guidelines of this section.

Additionally, the attribute **properties_time** can be used as a distribution key which will ensure the uniform distribution of data on your cluster. It should also be used as a *Sort Key*, something that will make your table to have data orderly stored and thus avoid having to *vacuum* it.

The attribute **event** is most likely to be used in your queries in conjunction with **properties_time**. For this reason, it would be good to create *Compound Sort Keys* on your *Mixpanel* tables, using **properties_time** as the first key and the event as the second one.

Useful Resources

Here you can find a post describing [how to export data from *Mixpanel* to *Redshift*](#) to

CHAPTER FOUR - MAINTENANCE

In this section, we will see why it is important to involve the data analyst in the maintenance process and how the knowledge of the data and the queries to be performed, can help not only keep the cluster healthy but also maximize its performance.

MAINTENANCE: INTRODUCTION

Keeping our dashboards and **analytic pipelines healthy**, require a performant cluster. For this reason, we need to periodically **perform maintenance tasks** on our Amazon Redshift Instance.

We will see why it is important to involve the data analyst in the maintenance process and how the knowledge of the data and the queries to be performed, can help not only keep the cluster healthy but also maximize its performance.

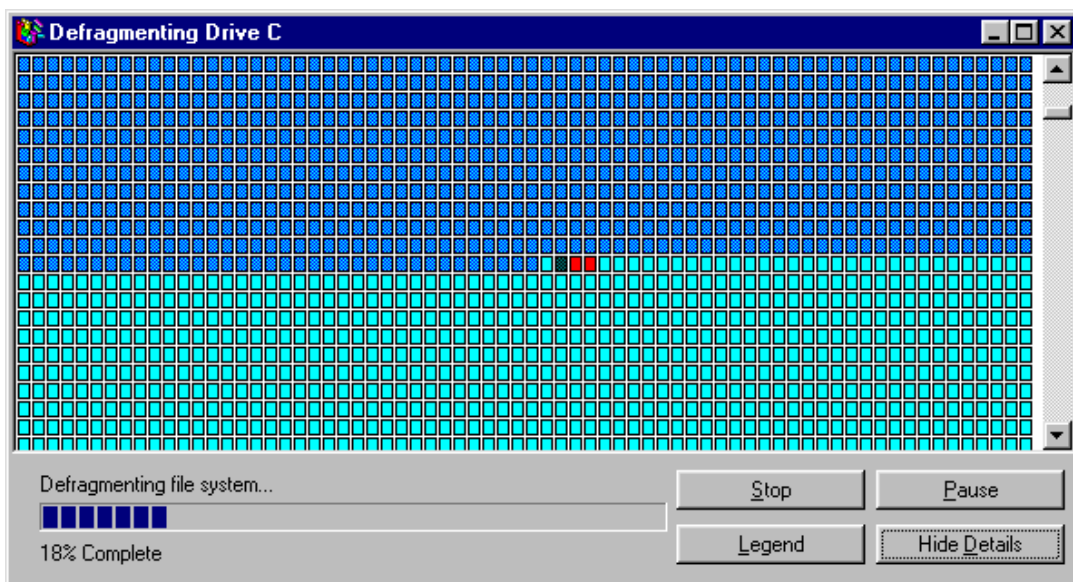
First, we will see what the Vacuum process is and how we can perform it. Why we need it and why it is important to know the nature of your data for laying out a good Vacuuming strategy.

Then investigate what Column Compression Encoding is. How it affects the utilization of our cluster and the speed of our queries. How the deep knowledge of the data that the analyst has, can help decide which encoding is optimal.

Finally, we will go through some different sources of information that we can use to monitor the performance of the executed queries. Last but not least provide some hints on how to use this information to identify what causes the queries to slow down and what maintenance technique to perform to fix them.

WHY TO VACUUM AMAZON REDSHIFT?

Most people who have worked with a Windows PC, especially in the past where SSD disks were not that common, are aware of a utility tool called, *Disk Defragmenter*.



Microsoft included Disk Defragmenter with all their OS distribution. Now, you might wonder why we mention a Windows OS utility tool in a guide about Amazon Redshift, but there's a similar process related to Amazon Redshift. That process is Vacuuming, and it also happens to be quite important for the health and maintenance of our Redshift cluster.

The Vacuuming process, is quite important for the health and maintenance of your AWS Redshift cluster.

The *purpose of the Disk Defragmenter was to rearrange the files on the disk to increase access speed*. The Vacuuming process of tables on an Amazon Redshift cluster is of course not about rearranging files, but instead about reclaiming freed space under some specific conditions.

However, as with Disk Defragmenter, the Vacuum command process is an important process which also affects the performance of our queries, among others.

Vacuum is an important process which also affects the performance of our queries, among others

Why does Vacuuming exist on Amazon Redshift?

Amazon Redshift does not reclaim free space automatically. Such available space is created whenever you **delete** or **update rows** on a table. This process is a design choice inherited from PostgreSQL and a routine maintenance process which we need to follow for our tables if we want to maximize the utilization of our Amazon Redshift cluster.

So by running a Vacuum command on one of our tables, we reclaim any free space that is the result of delete and update operations. At the same time, the data of the table get sorted.

This way, we end up with a compact and sorted table, which are useful for the performance of our cluster.

If you wonder why the *update* operations are also included together with *deletes*, this happens because behind the scenes an UPDATE command is the combination of a DELETE command, where the old row is first deleted, and then an INSERT command where the new row is inserted.

During a DELETE command, a row is marked as deleted but not removed. Additionally, the query processor has to scan all the rows, including those marked as deleted. So it is easy to understand that keeping deleted rows on a table costs additional process and thus slow down your queries.

In extreme situations, you might even end up with queries which may time-out due to the extra overhead the deleted but not reclaimed space might add.

How often should we run the Vacuum command in our Amazon Redshift tables?

The most common advice when it comes to vacuuming process is, **doing as often as you can**.

However, as in most cases, you need to consider a few factors to come up with the ideal vacuuming plan for your situation. Figuring out the vacuuming frequency might

sound like a task for a DBA, but it can be significantly improved if the data analyst or the data scientist also contributes to it.

There are two reasons for this:

1. The data analyst is the person who knows best the importance of the queries planned for execution. Thus it is possible to rank their execution differently based on importance.
2. The data analyst is aware of the structure, queries and the data involved and thus can help in determining the vacuuming type needed based on each query.

So, determining an efficient vacuuming plan requires knowledge related to the above two points.

Still, you need to consider the following factors too:

- *Vacuuming* process should happen during periods of inactivity or at least minimal activity on your cluster. So query planning is again essential here.
- The longer the time between consecutive vacuum commands for a table, *the longer* it takes for the vacuuming process to end.
- As vacuuming is about going through your data and reclaiming rows marked as deleted, it is an I/O intensive process. So, it affects any other queries or processes that you might be running concurrently, but the good thing is,
- Vacuuming can happen concurrently with other processes, so it may not block any ETL processes or queries you might be running.

Vacuum types

The Vacuuming process comes in different flavors, it is a configurable process and depending on what we want to achieve with our data and the type of queries we intend to perform on it, we have different options.

- **VACUUM FULL.** Vacuum Full is also the default configuration of a *vacuum* command, so if you do not provide any parameters to the command, this is performed on your data. With a *Full Vacuum type*, we both reclaim space, and we also sort the remaining data. These steps happen one after the other, so Amazon *Redshift* first recovers the space and then sorts the remaining data.
- **VACUUM DELETE ONLY.** If we select this option, then we only reclaim space and the remaining data is not sorted.
- **VACUUM SORT ONLY.** With this option, we do not reclaim any space, but we try to sort the existing data.
- **VACUUM REINDEX.** This command is probably the most resource intensive of all the *table vacuuming* options on *Amazon Redshift*. It is a *full vacuum type*

together with reindexing of interleaved data. It makes sense only for tables that use interleaved sort keys.

Selecting the most efficient Vacuum Type requires knowledge of all the queries involved with the tables to vacuum.

At this point, a data analyst can help to decide the most effective strategy for each table. The central questions that the analyst can answer and which can help us select the proper plan are:

1. Do any of your queries benefit from a sorted key on the tables used by these queries?
2. Does the data analyst have created any *interleaved sort keys*?

If the answer to (1) is no, then it might be better to proceed with a **VACUUM DELETE** strategy. If the answer to (2) is yes, then it might be beneficial to move forward with a **VACUUM REINDEX** strategy.

In any case, as the queries change, the need for indexes and sort keys change, the type and the way we schedule our vacuuming process might have to change as well. Thus it is important to always have the data analyst in the loop regarding the maintenance of an Amazon Redshift Cluster.

Interleaved Sort Keys and Vacuuming

Interleaved Sort Keys are a useful tool for improving the performance of specific queries. The case of *Interleaved Sort Keys* warrants a particular mention when it comes to the *vacuuming process* for the following reasons:

1. A data analyst will most likely use *Interleaved Sort Keys* with big tables
2. There's a particular type of *Vacuuming* when these keys exist in a table, which is computationally heavy.
3. The effect of UPDATE and DELETE operations to the deterioration of the query performance, is greater when *Interleaved Sort Keys* are involved.

So, it is important for an analyst to consider as part of the pros and cons of *Interleaved Sort Keys*, also the performance penalty of a *vacuuming process* and monitor closer the performance of queries that involve such sort keys.

The key metric we should watch to decide when a **VACUUM REINDEX** should happen is the **skew of values of columns** that have acted as an *Interleaved Sort Key* for the table.

Things to keep in mind

Table Vacuuming on *Amazon Redshift* is an important maintenance function that affects the work of a data analyst but how it is used, should also be influenced by her. A few things an analyst should keep in mind regarding the *Vacuuming process*:

1. *Vacuuming can help improve the query performance*
2. The *Vacuuming frequency* and the *Vacuuming type* should be affected by the type of queries we plan to perform.
3. The *Vacuuming process does not block the rest of the operations on a cluster but while it runs might affect the performance. So plan the query execution and Vacuum accordingly.*
4. *Interleaved Sort Keys* are significantly affected by unclaimed space on your cluster, so monitor these tables even closer.
5. As a data analyst, it is important to keep *Vacuum* in mind when you design your tables. **There is a certain limit on the number of columns** that a table have to perform a Vacuum, and this number is less than the maximum number of allowed columns for an *Amazon Redshift Table*. This number varies depending on your cluster configuration but nevertheless, try not to reach it. If for any reason you exceed the limit, Vacuum is not possible; we need to perform a DEEP COPY to reclaim free space. Unfortunately, *DEEP COPY* cannot happen while you operate on the table and you should avoid it.

Useful Resources

The system view ***SVV_VACUUM_PROGRESS*** returns an estimate of remaining time for a *vacuuming process* that is currently running.

The system table ***STL_VACUUM*** displays raw and block statistics for tables we vacuumed.

Finally, you can have a look to the ***Analyze & Vacuum Schema Utility*** provided and maintained by Amazon. This script can help you automate the *vacuuming process* for your *Amazon Redshift* cluster.

COLUMN COMPRESSION SETTINGS

Data cleaning and preparation are among the most time consuming parts of a Data Analyst or Data Scientist's job. During these tasks, the data analyst tries to further understand the data. What they can achieve with the data in analytic terms and how it should be organized to achieve their analytic goals.

In this chapter, we are going to see how to use this knowledge to optimize an Amazon Redshift Cluster and improve the query performance. Such improvement can happen by selecting an appropriate **Column Compression Encoding**. This choice is largely driven by the nature of the data that the column holds.

Why is Column Compression Important?

Choosing the appropriate *Column Compression Encoding* is usually perceived as a choice made during the process of design the tables of a database. However, data is something that changes with time and a decision that made sense a few months ago, might not be the optimal one anymore.

Data is something that changes with time and a decision that made sense a few months ago, might not be the optimal one anymore.

For this reason, we prefer to include *Column Compression Settings* as part of cluster maintenance, identifying again how the work of a data analyst can drive the related choices more efficiently.

Amazon Redshift is a **columnar database**, and the *compression* of columns can significantly affect the performance of queries. The reduced size of columns, result in a smaller amount of disk I/O operations and therefore it improves query performance.

The reduced size of columns, result in a smaller amount of disk I/O operations and therefore it improves query performance.

How Column Compression Works

By default, *Amazon Redshift*, stores data in its raw and uncompressed format. It is possible to define a Column Compression Encoding manually or ask *Amazon Redshift* to select an *Encoding* **automatically** during the execution of a COPY command.

The recommended way of applying *Column Compression Encodings* is by allowing Amazon Redshift to do it automatically but there are cases where manual selection might result in more optimized columns and tables.

Automatic Compression works by analyzing the data that are imported by the COPY command. Still, it has the following limitations:

1. You cannot perform *Automatic Column Compression* Encoding on a table that already has data.
2. It requires enough rows in the load data to decide an appropriate Column Compression Encoding successfully. The recommended amount of data is at least 100,000 rows.

Automatic compression works by taking a sample of the data to be loaded and selects the most appropriate *Column Compression Encoding* for each column of that table.

In the end, it recreates the table with the selected *Column Compression Encoding* for each column.

Amazon Redshift does not provide a mechanism for automatically detecting if a Compression Encoding of a column should change. For this reason, we can use the input of our data analyst to decide if the *encoding* should adjust and select the most appropriate one.

Amazon Redshift does not provide a mechanism for automatically detecting if a Compression Encoding of a column should change. For this reason, we can use the input of our data analyst to decide if the encoding should adjust and select the most appropriate one.

How the Compression Encoding of a column on an

existing table can change

Currently, *Amazon Redshift* does not provide a mechanism to modify the *Compression Encoding* of a column on a table that already has data.

The preferred way of performing such a task is by following the next process:

1. **Create** a new column with the desired *Compression Encoding*
2. **Copy** the data of the initial column to the new one
3. **Delete** the old column
4. **Rename** the new column to the name of the old one

```
alter table customers add column name_new varchar lzo;  
update customers set name_new = name;  
alter table customers drop column name;  
alter table customers rename column name_new to name;
```

Selecting the right Compression Encoding for your data

The time the above process takes is dependent on our table's size.

As we mentioned earlier, the right *Compression Encoding* depends on the nature of our stored data. *Amazon Redshift* tries to analyze the data and select the best possible *Encoding* while offering a broad range of different *Encodings* that can cover different scenarios.

The default selections by *Amazon Redshift* are the following:

- *Columns* defined as sort keys, are assigned a RAW compression, which means that they are not compressed. This is something to be considered by the analyst as when we set a sort key to improve the performance of a query, it is not possible to benefit from the compression of the column.
- *Columns* of a numerical type, like REAL and DOUBLE PRECISION together with BOOLEAN types are also assigned a RAW compression.
- Any other *Column* is assigned the LZ0 compression.

Amazon Redshift supports a **larger set of different** encodings that we can select manually based on the nature of the data we are going to work. More specifically:

If an analyst knows that a column takes values from a small controlled vocabulary, something that usually is evident during the performance of the preliminary analysis, a **Byte-Dictionary Encoding** can be more efficient. Think of possible cases for such an encoding columns that hold data like country names, ISO codes, currency codes or

HTTP response codes.

Among the most important data types that an analyst works with is the DATETIME. Some of the most valuable data is coming in the form of time series. For DATETIMES *Delta Encodings* can be more efficient.

LZO is automatically assigned by *Amazon Redshift* for the majority of the available data types as it offers a good trade-off between performance and compression ratio. It works extremely well with the CHAR and VARCHAR datatypes, so for variables or columns containing Strings, this encoding is a good choice.

Mostly Encoding is an interesting case of an encoding, mainly used with numerical data. We may use it when the datatype of a column is larger than the majority of data stored on it. By using this encoding, you can compress the majority of these values to smaller standard and keep the few larger outliers in their raw form.

This encoding can be used to optimize numerical columns, and the decision to use it can be driven by the analysis performed by an analyst before we load the data into the database. If for example, you have relatively small integers and a few outliers of huge numbers, you can select a data type that can hold the large numbers but compress the majority of the values using a smaller numerical data type.

Runlength Encoding can be used to compress data in a column that take values from a relatively small dictionary of values that are at least partially ordered. This encoding works because it substitutes repetitive values by a token pointing to the actual value, together with the number the value appears.

The use of this type of encoding requires an in-depth data understanding, and the insights of a data analyst might be of great use for selecting it.

Text255 and Text32K encodings like the *Runlength Encoding* are exploiting the higher occurrence of specific terms inside a text to compress the column data. If the analyst has identified that a column with string contains specific terms more often, she can select this encoding.

Finally, the *Zstandard* is a generic encoding that can be used with all the available data types of *Amazon Redshift* although it works best with CHAR and VARCHAR columns. If you work with text data and you cannot exploit any characteristics that would drive you to select among the previous encodings, this one might be a good generic choice.

Things to keep in mind when choosing Column Compression Encodings

Selecting the best *compression encodings* for your data **require a deep understanding of both the data that you are working with and the analysis** that you intent to perform. Both are subject of iterations, and as time goes on, things might change.

For this reason, it is important to go back and rethink your compression encodings whenever your analysts detect any substantial change. It might not be a frequent process like **Vacuuming**, but you should certainly do it. You should certainly leverage your data understanding as a data analyst to choose or update the correct encodings too.

Useful Resources

The SQL command ***ANALYZE COMPRESSION*** performs a compression analysis on your data and returns suggestions for the compression encoding to be used.

At this **repository**, you can find a tool written in python and maintained by *Amazon* that may help you automate the process of analyzing and selecting compression encodings for your tables.

If you are serious about the performance of your *Amazon Redshift Cluster* and you want to get the most out of it, you must validate your hypotheses in regards to the right *compression encodings* of your table columns. **Here you can find a good methodology for testing your assumptions.**

MONITORING AMAZON REDSHIFT QUERY PERFORMANCE

So far we have looked at how the knowledge of the data that a data analyst carries can help with the periodical **maintenance of an Amazon Redshift Cluster**. Knowing the nature of the data we work with, can help us to maximize the potential of our cluster by using tools like the **Column Compression Encoding** of a table and the **Vacuuming process** mechanism.

When we talk about maximize the potential of a cluster, we usually look at two main metrics. The first is its capacity, i.e. the amount of data we can load into it. The second is the time it takes for our *Amazon Redshift Cluster* to answer our queries.

So, no matter how many tools we have for optimizing our cluster, if we are not aware of its performance and more specifically the query execution time, we cannot use the knowledge of our data together with the provided tools for optimization. For this reason, **Monitoring the Query Performance** on our cluster should be an important part of our cluster maintenance routine.

No matter how many tools we have for optimizing our cluster, if we are not aware of its performance and more specifically the query execution time, we cannot use the knowledge of our data together with the provided tools for optimization.

In this chapter, we discuss how we can monitor the *Query Performance* on our *Amazon Redshift* instance.

Monitoring the Query Performance using the AWS Console

The easiest way to check how your queries perform is by using the **AWS Console**. From the cluster list, you can select the cluster for which you would like to see how

your queries perform. There, by clicking on the *Queries* tab, you get a list of all the queries executed on this specific cluster.

For each query, you can quickly check the time it takes for its completion and at which state it currently is. Also, you can monitor the CPU Utilization and the Network throughput during the execution of each query.

The *AWS Console* gives you access to a bird's eye view of your queries and their performance for a specific query, and it is good for pointing out problematic queries. *Amazon Redshift* also offers access to much more information, stored in some system tables, together with some special commands. All of these can help you debug, optimize and understand better the behavior and performance of queries.

STL_ALERT_EVENT_LOG

The *STL_ALERT_EVENT_LOG* table logs an alert every time the query optimizer identifies an issue with a query. You can use these alerts as indicators on how to optimize your queries.

When you get an alert on the table, the command *ANALYZE* can be used to update the statistics of a table and point out how to correct a problem, e.g. *vacuuming* might be required.

SVV_TABLE_INFO

The next important system table that holds information related to the performance of all queries and your cluster is *SVV_TABLE_INFO*. That table contains summary information about your tables.

After you have identified a query that is not performing as desired, using information from the AWS Console and the *STL_ALERT_EVENT_LOG*, you can consult this table for hints on how the tables that participate in a query might affect its performance. To be more precise, this is a *view* that utilizes data from multiple other tables to provide its information.

This view contains information that might help an analyst identify what is causing the deterioration of a query, as it contains information linked to *Compression Encoding*, *Distribution Keys*, *Sort Styles*, *Data Distribution Skew* and overall table statistics. For example,

- The **empty** column contains the number of blocks eligible to be freed by a *vacuum command*. *IMPORTANT* - this is a deprecated column to be removed in future releases.
- The **unsorted** column contains the percentage of rows that are currently unsorted. That* might indicate that a *full vacuum or a vacuum sort only* might help explain the current performance of a query.

- **Sortkey1_enc** contains information for the **compression encoding** of the first column in the sort key.
- **Skew_sortkey1** is the ratio of the largest non-sort key column's size to the size of the first column of the sort key. Have a look at this if you worry for the effectiveness of your sort keys.
- **Skew_rows** is usually used to evaluate the effectiveness of your current distribution strategy.
- **Max_varchar** contains the largest column's size with a datatype of **VARCHAR**. Columns of this data type, while compressing quite well, might not fit in memory and thus trigger the creation of a temporary table. That is something that can affect your queries' performance.

Amazon Redshift offers a wealth of information for monitoring the query performance. There are both visual tools and raw data that you may query on your *Redshift Instance*. A combined usage of all the different information sources related to the query performance can help you identify performance issues early. Figure out what causes them and together with the input from an analyst, improve them significantly.

Monitoring Disk Space

Another factor of a cluster that you should monitor closely, which affects the performance of your queries and you can manage it by both *VACUUMING* and the proper selection of *Compression Encodings* for your columns is the **cluster's free disk space**.

Temp tables are often created when you execute queries, and if your cluster is full then these tables cannot be created, so you might start noticing failing queries.

Temp tables are often created when you execute queries, and if your cluster is full then these tables cannot be created, so you might start noticing failing queries.

To monitor your current **Disk Space Usage**, you have to query the **STV_PARTITIONS** table. It contains information related to the disk speed performance and disk utilization. For example, the following query prints information about the capacity used for each of the cluster's disks, the percentage that currently used, at which host each disk is and who is the owner.

```
select owner, host, diskno, used, capacity,
(used-tossed)/capacity::numeric *100 as pctused
```

```
from stv_partitions order by owner;
```

If usage percentage is high, we can *Vacuum* our tables or delete some unnecessary tables that we might have. If utilization is uneven, then we might want to reconsider the distribution strategy that we follow. Examining the results can help us to quickly see if data is not evenly distributed across the disks of our cluster and their current usage.

Useful Resources

Your starting point regarding the *Monitoring* of your *Query Performance* should be the [AWS Console](#). It offers an excellent view of all your queries and some vital statistics that can help you quickly identify any issues.

Amazon also provides some auxiliary tools that use the information stored in the system tables of *Amazon Redshift* to offer more detailed monitoring. You can check [this monitoring solution](#) which is using [Amazon Cloudwatch](#) and [Amazon Lambda](#) to perform more detailed cluster monitoring.



You don't need to build your
pipelines from scratch.

We have you covered. Join a growing
number of companies that use Blendo for
their data infrastructure needs.

Contact us: team@blendo.co

[Learn more](#)