9
A Deeper Dive into Data Marts and Amazon Redshift

While a **data lake** enables a significant amount of analytics to happen inside it, there are several use cases where a data engineer may need to load data into an external **data warehouse**, or **data mart**, to enable a set of data consumers.

As we reviewed in *Chapter 2, Data Management Architectures for Analytics*, a data lake is a single source of truth across multiple lines of business, while a data mart generally contains a subset of data of interest to a particular group of users. A data mart could be a relational database, a data warehouse, or a different kind of datastore.

Data marts serve two primary purposes. First, they provide a database with a subset of the data in the data lake, optimized for specific types of queries (such as for a specific business function). In addition, they also provide a higher-performing, lower-latency query engine, which is often required for specific analytic use cases (such as for powering **Business Intelligence** (**BI**) applications).

In this chapter, we will focus on data warehouses and data marts and cover the following topics:

- Extending analytics with data warehouses/data marts
- What not to do – anti-patterns for a data warehouse
- Redshift architecture review and storage deep dive
- Designing a high-performance data warehouse
- Moving data between a data lake and Redshift

- Exploring advanced Redshift features
- Hands-on – deploying a Redshift Serverless cluster and running Redshift Spectrum queries

Technical requirements

For the hands-on exercises in this chapter, you will need permission to create a new IAM role, as well as permission to create a **Redshift** cluster.

You can find the code files of this chapter in the GitHub repository using the following link: `https://github.com/PacktPublishing/Data-Engineering-with-AWS-2nd-edition/tree/main/Chapter09`

Extending analytics with data warehouses/data marts

Tools such as **Amazon Athena** (which we will do a deeper dive into in *Chapter 11*, *Ad Hoc Queries with Amazon Athena*) allow us to run SQL queries directly on data in the data lake. While this enables us to query very large datasets that exist in an **Amazon S3** data lake, the performance of these queries is generally lower than the performance you get when running queries against data on a high-performance disk that is local to the compute engine.

However, not all queries require this kind of high performance, and we can categorize our queries and data into cold, warm, and hot tiers. Before diving into the topic of data marts and data warehouses, let's first take a look at the different tiers of queries/data storage that are common in data lake projects.

Cold and warm data

We've grouped the cold and warm data tiers into one section, as when building in AWS, both of these tiers generally use Amazon S3 storage.

As we have discussed elsewhere in this book, Amazon S3 is low-cost object storage that provides very high durability and an SLA of 99.9% availability, while also being massively scalable.

AWS offers different storage classes for the objects you store in Amazon S3, and some classes are better suited to cold data, while other classes are well suited to warm data. Let's first take a look at the difference between cold and warm data, and then we can do a deeper dive into the different storage classes.

Cold data

This is data that is not frequently accessed but it is mandatory to store for long periods for compliance and governance reasons, or historical data that is stored to enable future research and development (such as for training **Machine Learning** (**ML**) models).

An example of this is the access logs from a banking website. Unless there is a court-issued legal request to query the logs for some specific data, the chances are that after a few months, we will not need to access this data again. However, due to compliance reasons, we may need to store this data for a number of years.

Another example is detailed data from a range of sensors in a factory. This data may not be queried actively after 30 days, but we want to keep this data available in case there is a future ML project where it would be useful to train the ML model with rich, historical data.

Warm data

Warm data is data that is accessed relatively often but does not require extremely low latency for retrieval. This is data that needs to be queried on demand, such as data that is used in daily ETL jobs, or data used for ad hoc querying and data discovery.

An example of this kind of data is data that is ingested in our **raw** data lake zone daily, such as data from a transactional database system. This data will be processed by our ETL jobs daily, and data will be written out to the **transformed** zone.

Generally, data in the transformed zone will still be batch-processed for further business transforms, before being moved to the curated zone. All of these zones would likely fall into the category of warm data.

Amazon S3 storage classes

Broadly speaking, you can think of Amazon S3 storage classes as being part of one of three different groupings.

General purpose

The **Amazon S3 Standard storage class** is a general-purpose storage class that provides immediate (millisecond) access to data. This storage class is designed for data that is frequently accessed.

The Amazon S3 Standard storage class has a per-GB cost for data stored, and no per-GB cost to read or access the data. As such, it is ideal for storing current data lake data, such as newly ingested data, or data from the past month (or few months) that is queried regularly.

Infrequent Access

There are two different storage classes that are categorized as Infrequent Access – **S3 Standard-Infrequent Access** and **S3 One Zone-Infrequent Access**. Both of these storage classes provide immediate (millisecond) access to data, but the pricing model is different from the Standard storage class.

With the Infrequent Access storage classes, you pay a lower per-GB cost than S3 Standard for storing data, but there is a per-GB charge for

reading/accessing the data. Therefore, this storage class is ideal for data that you need to store for longer periods of time and access infrequently. For example, you may store the most recent 3 months of data in the Standard storage class, but store the data for the 24 months prior to that in Infrequent Access, as that data may be queried once a month for month-end reporting.

The difference between Standard-Infrequent Access and One Zone-Infrequent Access is that the One Zone data is only stored in a single availability zone. This reduces the durability of data stored in this storage class, while slightly reducing the data storage costs, and is only intended for data that can be easily recreated.

Note that data stored in the Infrequent Access storage classes is always charged for a minimum of 30 days, therefore it is not suited to data that will be deleted in less than 30 days. Infrequent Access is intended for longer-term data storage.

Archive storage

Amazon S3 has a number of **Glacier storage classes** that are intended for different types of archiving requirements.

The **Glacier Instant Retrieval storage class** offers archival storage for data that is not accessed regularly (for example, once per quarter) but you want to be able to access the data immediately. Objects in the Glacier Instant Retrieval storage class do not need to be restored before you access them, and therefore they can be queried by Athena. The Glacier Instant Retrieval storage class has a lower cost per GB of data stored than the Standard and Infrequent Access classes, but a higher cost to retrieve/access data than the other classes.

The Amazon S3 **Glacier Flexible Retrieval storage class** is intended for long-term storage where access to the data may be required a few times a year, and immediate access is not required. Data can be retrieved from S3 Glacier in minutes to hours (with different price

points for the retrieval, based on how quickly the data is required). Data in S3 Glacier cannot be directly queried with Amazon Athena or Glue jobs – it must be retrieved and stored in a regular storage class before it can be queried.

The Amazon S3 **Glacier Deep Archive storage class** is the lowest-cost storage for long-term data retention and is intended for data that may be retrieved at most once or twice a year. Data in this storage class can be retrieved within 12 hours.

Storage charges for objects in S3 Glacier storage classes are for a minimum of 90 days for Instant Retrieval and Flexible Retrieval and 180 days for Deep Archive, so they are not intended for data that is short-lived.

Data that you store in the Glacier Instant Retrieval storage class can still be a part of your data lake, as you are able to immediately access the data and query it with tools such as Amazon Athena. However, because of the data access charges, you need to ensure that data in this class is not queried on a regular basis as this can get very expensive. If data is stored in the Glacier Flexible Retrieval or Deep Archive storage classes, this data cannot be considered active data in the data lake, as the data cannot be queried without first restoring it from the archive.

Using Amazon S3 Lifecycle rules to automatically move data between storage classes

Amazon S3 enables you to configure **Amazon S3 Lifecycle rules** that are used to automatically move your data between the different storage classes. For example, you know that once your granular sales data per store is 3 months old, it is not queried often, so you could configure a lifecycle rule to automatically move data in the sales prefix of the bucket to either Infrequent Access or Glacier Instant Retrieval storage classes.

When creating a lifecycle rule, you can specify multiple transitions that apply at different ages (days after creation) of an object. You can also specify configuration options, including the following:

- You can either limit the scope of the rule to a full bucket or create a rule for a specific prefix in a bucket.
- You can select to exclude files under a specific size or over a specific size.
- You can move objects between storage classes or select to delete an object.

For example, you can create a configuration rule that will move objects to Standard-Infrequent Access after 30 days, and then to Glacier Instant Retrieval 90 days after the object was created. It will then move the object to Glacier Deep Archive 365 days after it was created and, finally, will permanently delete the object 1,095 days (3 years) after creation.

Enabling Amazon S3 to automatically optimize your storage costs

You can also use a special storage class called **Amazon S3 Intelligent Tiering** to automatically move data to the most cost-effective access tier, based on changing access patterns. When you place objects in the S3 Intelligent Tiering storage class, the object is moved between tiers based on when it was last accessed (which is different from how lifecycle rules work, as those are based on the number of days since the object was created).

The default tier is the Frequent Access tier, but if an object in this tier is not accessed for 30 days, it is automatically moved to the Infrequent Access tier. If an object is not accessed for 90 consecutive days, it is automatically moved to the Archive Instant Access tier (which, much like Glacier Instant Retrieval, makes the object available in milliseconds).

You can optionally also enable having the object moved to the Archive Access tier, after anywhere between 90 and 730 days without it being

accessed. Once data is moved to the Archive Access tier, it must be retrieved before it can be accessed, and retrieval ranges from minutes to 5 hours, although you can request expedited retrieval (at an additional charge), which restores objects in 1–5 minutes.

An additional option is to activate the Deep Archive Access tier for data that has not been accessed for between 180 and 730 days (configurable). The Deep Archive Access tier is similar to the Glacier Deep Archive storage class in terms of time to retrieve objects (9–12 hours).

When an object that has been moved into an Infrequent Access or Archive tier is accessed, it is moved back into the Frequent Access tier.

There are trade-offs between storage costs, data retrieval costs, and the amount of time required to access data that has been archived (for certain storage classes/tiers), and therefore you need to consider these before enabling either lifecycle rules or the optional archive components of the Intelligent Tiering storage class.

However, in most cases, it is highly recommended that you consider using the S3 Intelligent Tiering storage class (without optional archiving) as your default storage class for data in your data lake, as this can lead to significant cost optimization with very little effort or management overhead.

Each of these storage classes has different pricing plans. S3 Standard's cost is based on storage and API calls (`put` , `copy` , `get` , and more), while S3 Standard-Infrequent Access and Glacier Archive classes also have a cost per GB of data retrieved. S3 Intelligent Tiering does not have a cost per GB for data retrieved, but it does have a small monitoring and automation cost per object. For more details on pricing, see https://aws.amazon.com/s3/pricing/.

Hot data

Hot data is data that is highly critical for day-to-day analytics enablement in an organization. This is data that is likely accessed multiple times per day, and low-latency, high-performance access to the data is critical.

An example of this kind of data would be data used by a BI application (such as **Amazon QuickSight** or **Tableau**). This could be data that is used to show manufacturing and sales of products at different sites, for example. This is often the kind of data that is used by end user data consumers in the organization, as well as by business analysts who need to run complex data queries. This data may also be used in constantly refreshing dashboards that provide critical business metrics and KPIs used by senior executives in the organization.

In AWS, several services can be used to provide high-performance, low-latency access to data. These include the **RDS database** engines, the **NoSQL DynamoDB** database, as well as **OpenSearch** (for searching full-text data). However, from an analytic perspective, the most common targets for hot data are **Amazon Redshift** or **Amazon QuickSight SPICE** (which stands for **Super-fast， Parallel， In-memory Calculation Engine**):

- **Amazon Redshift** is a super-fast cloud-native data warehousing solution that provides high-performance, low-latency access to data stored in the data warehouse.
- **Amazon QuickSight** is a BI tool from Amazon for creating dashboards. With Amazon QuickSight, you have the option of reading data from sources, such as Amazon Redshift, or loading data directly into the QuickSight in-memory database engine (SPICE) for optimal high-performance, low-latency access.

As we mentioned previously, AWS offers purpose-built storage engines for different data types/temperatures. The decision of which engine to use is generally based on a cost versus performance trade-off.

In many cases, data is time-sensitive. There may be a business application that needs to report on historical statistics, current trends, and a zoomed-in view of the previous few months of data. Some of this data may also need to be refreshed frequently. This requires a data engineer to process the data, clean and transform it, and then load a subset of the data to a high-performing engine, such as Amazon Redshift.

In this chapter, we are going to focus on using Amazon Redshift as a high-performance data mart for hot data access. Data lakes are a great option from a cost and scalability perspective for storing large amounts of data and being the ultimate source of truth.

However, data warehouses provide an application-specific approach to querying large-scale structured and semi-structured data with the best performance and lowest latency.

What not to do – anti-patterns for a data warehouse

While there are many good ways to use a data warehouse for analytics, there are some approaches that at first may seem to be a good fit for a data warehouse but are generally not recommended.

Let's take a look at some of the ways of using a data warehouse that should be avoided.

Using a data warehouse as a transactional datastore

Data warehouses are designed to be optimized for **Online Analytical Processing** (**OLAP**) queries, so they should not be used for **Online Transaction Processing** (**OLTP**) queries and use cases.

While there are mechanisms to update or delete data from a data warehouse (such as the `merge` statement in Redshift), a data ware-

house is primarily designed for mostly append-only, or insert, queries. There are also other features of transactional databases (such as **MySQL** or **PostgreSQL**) that are available in Redshift – such as the concept of primary and foreign keys – but these are used for performance optimization and query planning and are **not enforced** by Redshift.

Using a data warehouse as a data lake

Data warehouses offer increased performance by having high-performance storage directly attached to the compute engine. A data warehouse is also able to scale to store vast amounts of data, and while primarily designed to support structured data, they are also able to offer some support for semi-structured data.

However, data warehouses, by design, require upfront thought about schema and table structure. They are also not designed to store unstructured data (such as images and audio), and they generally use SQL as the primary method for data querying and transformation. As data warehouses include a compute engine, their cost is also higher than storing data in low-cost object storage.

In contrast, with data lakes, you can store all the data in low-cost object storage and can ingest data without needing to design an appropriate schema structure first. You can also analyze the dataset directly (using tools such as Amazon Athena) and transform the data with a wide range of tools (**SQL** and **Spark**, for example), and then bring just the required data into the data warehouse.

While it is possible to load raw data in a data warehouse and then transform the data in the warehouse, that is the **Extract, Load, Transform** (**ELT**) process, this often ends up costing significantly more than storing the raw data in an object store such as S3, transforming the data directly in the data lake, and then loading a subset of data into a data warehouse.

The goal is to avoid storing unnecessary data in a data warehouse. Data warehouses are supposed to store curated datasets with well-defined schemas and should only store hot data that is needed for high-performance, low-latency queries.

Storing unstructured data

While some data warehouses (such as Amazon Redshift and Snowflake) can store semi-structured data (such as JSON data), data warehouses generally cannot be used to store unstructured data such as images, videos, and other media content.

You should always consider which data engine may be best for a specific data type before just defaulting to storing the data in a data warehouse. For example, Health Care FHIR data has a heavily nested JSON structure. While it is possible to store and query this in Amazon Redshift, or another data warehouse solution, you may want to consider using a solution designed for that specific data type, such as **Amazon HealthLake**.

Now that we have reviewed some of the ways that a data warehouse should not be used, let's dig deeper into the Redshift architecture.

Redshift architecture review and storage deep dive

In this section, we will take a deeper dive into the architecture of Redshift clusters, as well as into how data in tables is stored across Redshift nodes. This in-depth look will help you understand and fine-tune Redshift's performance, though we will also cover how many of the design decisions affecting table layout can be automated by Redshift.

In *Chapter 2*, *Data Management Architectures for Analytics*, we briefly discussed how the Redshift architecture uses leader and compute

nodes. Each compute node contains a certain amount of compute power (CPUs and memory), as well as a certain amount of local storage. When configuring your Redshift cluster, you can add multiple compute nodes, depending on your compute and storage requirements. Note that to provide fault tolerance and improved durability, the compute nodes have 2.5–3x the stated node storage capacity (for example, if the addressable storage capacity is listed as 2.56 TB, the actual underlying storage may be closer to 7.5 TB).

Every compute node is split into either 2, 4, or 16 slices, depending on the cluster type and size. Each slice is allocated a portion of the node's memory and storage and works as an independent worker, but in parallel with the other slices.

The slices store different columns of data for large tables, as distributed by the leader node. The data for each column is persisted as 1 MB immutable blocks, and each column can grow independently.

When a user runs a query against Redshift, the leader node creates a query plan and allocates work for each slice, and then the slices execute the work in parallel. When each slice completes its work, it passes the results back to the leader node for final aggregation or sorting and merging. However, this means that a query is only as good as its slowest partition (or slice).

Data distribution across slices

Let's have a look at how data is distributed across slices in Redshift:

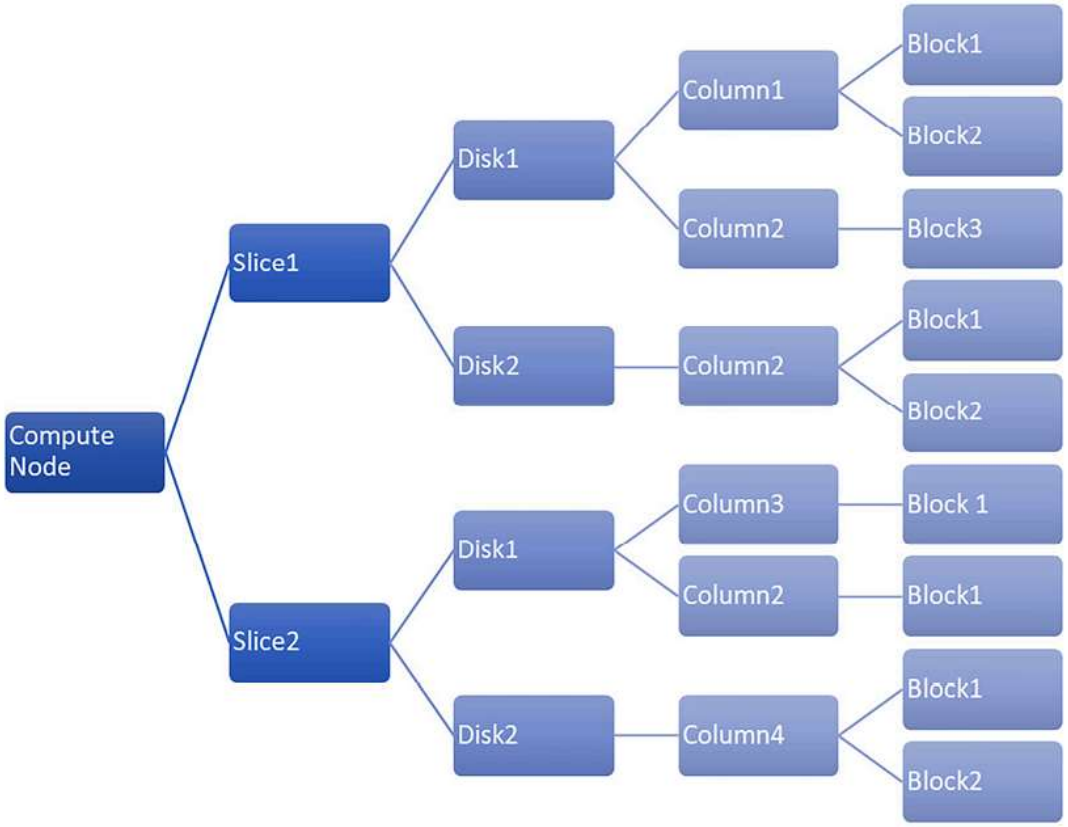

Figure 9.1: Data distribution across slices on a compute node

In the preceding diagram, we can see that `Column2` is distributed across `Slice1-Disk1`, `Slice1-Disk2`, and `Slice2-Disk1`. To increase data throughput and query performance, data should be spread evenly across slices to avoid I/O bottlenecks. If most of the data for a specific table were on one node, that node would end up doing all the heavy lifting and diminish the point of parallelism. Redshift supports multiple distribution styles, including `EVEN`, `KEY`, and `ALL` (and can automatically select the best distribution style, as we will discuss later in this chapter).

The distribution style that's selected for a specific table determines which slice a row in a column will be stored on.

One of the most common operations when performing analytics is the `JOIN` operation. Let's look at an example where we have two tables, one of which is a small dimension table (2–3 million rows) and the

other is a very large fact table (potentially with hundreds of millions of rows). For a reminder about data warehouse schema design with fact and dimension tables, refer back to the section *Dimensional modeling in data warehouses* in *Chapter 2*, *Data Management Architectures for Analytics*.

The small dimension table can easily fit into the storage of a single node, while the large fact table needs to be spread across multiple nodes. One of the biggest impacts on performance regarding a `JOIN` query is when data needs to be shuffled (copied) between nodes. To avoid this, and to optimize `JOIN` performance, the smaller dimension table can be stored on all the slices of the cluster by specifying an **ALL distribution style**. For the larger table, data can be equally distributed across all the slices in a round-robin fashion by specifying an **EVEN distribution style**. By doing this, every slice will have a full copy of the small dimension table and it can directly join that with the subset of data it holds for the large fact table, without needing to shuffle the dimension data from other slices.

While this can be ideal for query performance, the `ALL` distribution style does have some overhead with regard to the amount of storage space used by the cluster, as well as a negative performance impact for data loads.

An alternative approach that can be used to optimize joins, especially if both tables being joined are large, is to ensure that the same slice stores the rows for both tables that will need to be joined. A way to achieve this is by using the `KEY` distribution style, where a hash value of one of the columns will determine which row of each table will be stored on which slice.

For example, let's say that we have a table that stores details about all of the products we sell, and that this table contains a `product_id` column. Let's also say we have a different table that contains details of all sales, and that it also contains a column called `product_id`.

In our queries, we often need to join these tables on the `product_id` column. By distributing the data for both tables based on the value of the `product_id` column, we can help ensure that all the rows that need to be joined are on the same slice. Redshift would determine the hash value of, for example, `product_id "DLX5992445"` and, based on that hash value, determine which slice the data should be stored on. With this approach, all the rows from both tables that contain that `product_id` would be stored on the same slice.

For grouping and aggregation queries, you also want to reduce data shuffling (copying data from one node to another to run a specific query) to save network I/O. This can also be achieved by using the `KEY` distribution style to keep records with the same key on the same slice. In this scenario, you would specify the column used in the `GROUP BY` clause as the key to distribute the data on.

However, if we queried one of these tables with a `WHERE` filter on the `product_id` column, then this distribution would create a bottleneck, as all the data that needed to be returned from the query would be on one slice. As such, you should avoid specifying a `KEY` distribution on a column that is commonly used in a `WHERE` clause. Finally, the column that's used for `KEY` distribution should always be one with high cardinality and normal distribution of data to avoid hot partitions and data skew.

While this can be very complex, Redshift can automatically optimize configuration items such as distribution styles, as we will discuss later in this chapter in the *Designing a high-performance data warehouse* section.

Redshift Zone Maps and sorting data

The time it takes a query to return results is also impacted by hardware factors – specifically, the amount of disk seek and disk access time:

- **Disk seek** is the time it takes a hard drive to move the read head from one block to another (as such, it does not apply to nodes that use SSD drives).
- **Disk access** is the latency in reading and writing stored data on disk blocks and transferring the requested data back to the client.

To reduce data access latency, Redshift stores in-memory metadata about each disk block on the leader node in what is called **Zone Maps**. For example, Zone Maps store the minimum and maximum values for the data of each column that is stored within a specific 1 MB data block. Based on these Zone Maps, Redshift knows which blocks contain data relevant to a query, so it can skip reading blocks that do not contain data needed for the query. This helps optimize query performance by magnitudes by reducing the number of reads.

Zone Maps are most effective when the data on blocks is sorted. When defining a table, you can optionally define one or more sort keys, which determines how data is sorted within a block. When choosing multiple sort keys, you can either have a priority order of keys using a **compound sort key** or give equal priority to each sort key using an **interleaved sort key**. The default sort key type is a compound sort key, and this is recommended for most scenarios.

Sort keys should be on columns that are frequently used with range filters or columns where you regularly compute aggregations. While sort keys can help significantly increase query performance by improving the effectiveness of Zone Maps, they can harm the performance of ingest tasks. In the next section, we will look at how Redshift simplifies some of these difficult design decisions by being able to automatically optimize a table's sort key. In most cases, it makes sense to have Redshift automatically optimize the sort key, at least initially, and then over time, investigate manually setting the sort key if you feel further optimization is required.

Designing a high-performance data warehouse

When you're looking to design a high-performing data warehouse, multiple factors need to be considered. These include items such as cluster type and sizing, compression types, distribution keys, sort keys, data types, and table constraints.

As part of the design process, you will need to consider several trade-offs, such as cost versus performance. Business requirements and the available budget will often drive these decisions.

Beyond decisions about infrastructure and storage, the logical schema design also plays a big part in optimizing the performance of the data warehouse. Often, this will be an iterative process, where you start with an initial schema design that you refine over time to optimize for increased performance.

Provisioned versus Redshift Serverless clusters

When creating an Amazon Redshift cluster, you can select to either use a serverless Redshift configuration or provision specific resources. With **Redshift Serverless** clusters, you only pay for compute costs while the cluster is active, and the cluster automatically shuts down during periods of inactivity. With a provisioned cluster, you pay for the compute resources for as long as the cluster is up (although you can manually choose or schedule times to pause and resume the cluster, and when paused, you only pay for storage).

Both provisioned and serverless clusters support the same features, and both can handle complex workloads and advanced queries. With provisioned clusters, you need to determine how many nodes to configure in the cluster, as well as what types of nodes, in order to get the performance you require. With Redshift Serverless, you only need to specify a base number for **Redshift Processing Units** (**RPUs**), and

Redshift Serverless will automatically scale in order to handle the load placed on the cluster. As a result, Redshift Serverless is able to more dynamically respond to changes in processing load.

Redshift Serverless clusters also do not require you to specify maintenance windows or to plan for software upgrades. As a serverless service, updated software versions are automatically applied and there is no interruption to existing connections or queries when Redshift changes the underlying software version.

Overall, Redshift Serverless is simpler to deploy and manage than working with a provisioned cluster; however, the costs for Redshift Serverless are harder to predict and manage. To help manage costs, you can set limits on the maximum RPUs consumed over a period of time (daily, weekly, or monthly), and when the limit is reached, either generate a notification or prevent any further queries from being run.

If you have unpredictable workloads, or your cluster is only used at certain times (such as a development or test cluster), then there are many advantages to using Redshift Serverless. However, if you have well-defined and relatively constant workloads, then using a provisioned cluster can provide lower costs (especially when you commit to usage by purchasing a 1-year or 3-year Reserved Instance for Redshift).

Selecting the optimal Redshift node type for provisioned clusters

With provisioned clusters, there are different types of nodes available, each with different combinations of CPU, memory, storage capacity, and storage type. The following are the three families of node types:

1. **RA3 nodes**: These nodes use managed storage, which decouples compute and storage since you pay a per-hour compute fee, and a separate fee based on how much managed storage you use over the

month. Storage is a combination of local SSD storage and data stored in S3 and is fully managed by Redshift.
2. **DC2 nodes**: These are designed for compute-intensive workloads and feature a fixed amount of local SSD storage per node. With DC2 nodes, compute and storage are coupled (meaning that to increase either compute or storage, you need to add a new node containing both compute and storage).
3. **DS2 nodes (legacy)**: These are legacy nodes that offer compute with attached large hard disk drives. With DS2 nodes, compute and storage are also coupled.

AWS recommends that for small provisioned clusters (under 1 TB compressed in size), you use DC2 nodes, while larger data warehouses make use of the RA3 nodes with managed storage.

Note that many advanced features of Redshift, such as data sharing, are only available with RA3 nodes. The DS2 node type is a legacy node type that is not generally recommended for use when creating a new Redshift cluster.

When creating a new provisioned Redshift cluster in the console, you have the option of entering information about your data's size, type of data, and data retention, and Redshift will provide a recommended node type and the number of nodes for your workload.

Selecting the optimal table distribution style and sort key

In the early days of Redshift, users had to specifically select the distribution style and sort key that they wanted to use for each table. When a Redshift cluster was not performing as well as expected, it would often turn out that the underlying issue was having a non-optimal distribution style and/or sort key.

As a result, Amazon introduced new functionality that enabled Redshift to use advanced Artificial Intelligence methods to monitor queries being run on the cluster, and to automatically apply the optimal distribution style and/or sort key. Optimizations can be applied to tables within a few hours of a minimum number of queries being run.

If you create a new table and do not specify a specific distribution style or sort key, Redshift sets both of those settings to `AUTO`. Smaller tables will initially be set to have an `ALL` distribution style, while larger tables will have an `EVEN` distribution style.

If a table starts small but grows over time, Redshift automatically adjusts the distribution style to `EVEN`. Over time, as Redshift analyzes the queries being run on the cluster, it may further adjust the table distribution style to be `KEY`-based.

Similarly, Redshift analyzes queries being run to determine the optimal sort key for a table. The goal of this optimization is to optimize the data blocks that are read from the disk during a table scan.

It is strongly recommended that you allow Redshift to manage distribution and sort key optimizations for your table automatically, but you do have the power to manually configure these settings if you have a unique use case, or if you find that you need further performance improvements beyond what is achieved with automatic selection.

Selecting the right data type for columns

Every column in a Redshift table is associated with a specific data type, and this data type ensures that the column will comply with specific constraints. This helps enforce the types of operations that can be performed on the values in the column.

For example, an arithmetic operation such as `sum` can only be performed on **numeric** data types. If you needed to perform a `sum` operation on a column type that was defined as a **character** or **string** type, you would need to cast it to a numeric type first. This can have an impact on query performance, so it needs to be taken into consideration.

There are broadly six data types that Amazon Redshift currently supports. Let's do a deeper dive into each of these data types.

Character types

Character data types are equivalent to string data types in programming languages and relational databases, and are used to store text.

There are two primary character types:

1. `CHAR(n)`, `CHARACTER(n)`, and `NCHAR(n)`: These are fixed-length character strings that support single-byte characters only. Data is stored with trailing white spaces at the end to convert the string into a fixed length. If you defined a column as `CHAR(8)`, for example, data in this column would be stored as follows:

   ```
   CHAR(8)
   "ABC     "
   "DEF     "
   ```

   However, the trailing white space is ignored during queries. For example, if you were querying the length of one of the aforementioned records, it would return a result of `3`, not `8`. Also, if you were querying the table for records matching `"ABC"`, the trailing space would again be ignored and the record would be returned.

2. `VARCHAR(n)` and `NVARCHAR(n)`: These are variable-length character strings that support multi-byte characters. When creating this data type, to determine the correct length to specify, you should

multiply the number of bytes per character by the maximum number of characters you need to store.
A column with `VARCHAR(8)`, for example, can store up to eight single-byte characters, four 2-byte characters, or two 4-byte characters. To calculate the value of `n` for `VARCHAR`, multiply the number of bytes per character by the number of characters. As this data type is for variable-length strings, the data is not padded with trailing white space.

When deciding on the character type, if you need to store multi-byte characters, then you should always use the `VARCHAR` data type. For example, the Euro symbol (`€`) is represented by a 3-byte character, so this should not be stored in a `CHAR` column.

However, if your data can always be encoded with single-byte characters and is always a fixed length, then use the fixed-width `CHAR` data type. An example of this is columns that store phone numbers or IP addresses.

AWS recommends that you always use the smallest possible column size rather than providing a very large value, for convenience, as using an unnecessarily large length can have a performance impact for complex queries. However, there is a trade-off because if the value is too small, you will find that queries may fail if the data you attempt to insert is larger than the length specified. Therefore, consider what may be the largest potential value you need to store for a column and use that when defining the column.

Numeric types

Number data types in Redshift include integers, decimals, and floating-point numbers. Let's look at the primary numeric types.

Integer types

Integer types are used to store whole numbers, and there are a few options based on the size of the integer you need to store:

1. `SMALLINT` / `INT2` : These integers have a range of -32,768 to +32,767.
2. `INTEGER` / `INT` / `INT4` : These integers have a range of -2,147,483,648 to +2,147,483,647.
3. `BIGINT` / `INT8` : These integers have a range of -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807.

You should always use the smallest possible integer type that will be able to store all expected values. For example, if you're storing the age of a person, you should use `SMALLINT` , while if you're storing a count of product inventory where you expect to have hundreds of thousands of units to potentially a few million units on hand, you should use the `INTEGER` type.

Decimal type

The `DECIMAL` type allows you to specify the precision and scale you need to store. The `precision` indicates the total number of digits on both sides of the decimal point, while the `scale` indicates the number of digits on the right-hand side of the decimal point. You define the column by specifying `DECIMAL(precision, scale)` .

Creating a column and specifying a type as `DECIMAL(7,3)` would enable values in the range of -9,999.999 to +9,999.999.

The `DECIMAL` type is useful for storing the results of complex calculations where you want full control over the accuracy of the results.

Floating-point types

These numeric types are used to store values with variable precision. The floating-point types are known as inexact types, which means you may notice a slight discrepancy when storing and reading back a spe-

cific value, as some values are stored as approximations. If you need to ensure exact calculations, you should use the `DECIMAL` type instead.

The two floating-point types that are supported in Redshift are as follows:

1. `REAL` / `FLOAT4` : These support values of up to 6 digits of precision.
2. `DOUBLE PRECISION` / `FLOAT8` / `FLOAT` : These support values of up to 15 digits of precision.

This data type is used to avoid overflow errors for values that are mathematically within range, but the string length exceeds the range limit. When you insert values that exceed the precision for that type, the values are truncated. For a column of the `REAL` type (which supports up to 6 digits of precision), if you insert `7876.7876` , it would be stored as `7876.78` . Or, if you attempted to insert a value of `787678.7876` , it would be stored as `787678` .

Datetime types

These types are equivalent to simple date, time, or timestamp columns in programming languages. The following date/time types are supported in Redshift:

1. `DATE` : This column type supports storing a date without any associated time. Data should always be enclosed in double quotation marks.
2. `TIME` / `TIMEZ` : This column type supports storing a time of day without any associated date. `TIMEZ` is used to specify the time of day with the time zone, with the default time zone being **Coordinated Universal Time (UTC)**. `TIME` is stored with up to 6-digit precision for fractional seconds.
3. `TIMESTAMP` / `TIMESTAMPZ` : This column type is a combination of `DATE` followed by `TIME` / `TIMEZ` . If you insert a date without a time value, or only a partial time value, into this column type, any miss-

ing values will be stored as `00`. For example, a `TIMESTAMP` of `2021-05-23` will be stored as `20121-05-23 00:00:00`.

Boolean type

The **Boolean** type is used to store single-byte literals with a `True` or `False` state or `UNKNOWN`. When inserting data into a Boolean type field, the valid set of specifiers for `True` is `{TRUE, 't', 'true', 'y', 'yes', '1'}`. The valid set of specifiers for `False` is `{FALSE 'f' 'false' 'n' 'no' '0'}`. And if a column has a `NULL` value, it is considered `UNKNOWN`.

Regardless of what literal string was used to insert a column of the Boolean type, the data is always stored and displayed as `t` for true and `f` for false.

HLLSKETCH type

The `HLLSKETCH` type is a complex data type that stores the results of what is known as the **HyperLogLog algorithm**. This algorithm can be used to estimate the cardinality (number of unique values) in a large multiset very efficiently. Estimating the number of unique values is a useful analytic function that can be used to map trends over time.

For example, if you run a large social media website with hundreds of millions of people visiting every day, to track trends, you may want to calculate how many unique visitors you have each day, each week, or each month. Using traditional SQL to perform this calculation would be impractical as the query would take too long and would require an extremely large amount of memory.

This is where algorithms such as the HyperLogLog algorithm come in. Again, there is a trade-off, as you do give up some level of accuracy in exchange for a much more efficient way of getting a good estimate of cardinality (generally, the error range is expected to be between 0.01 and 0.6%). Using this algorithm means you can now work with ex-

tremely large datasets and calculate the estimated unique values with minimal memory usage and within a reasonable time.

Redshift stores the result of the HyperLogLog algorithm in a data type called `HLLSKETCH`. You could have a daily query that runs to calculate the approximate unique visitors to your website each day and store that in an `HLLSKETCH` data type. Then, each week, you could use Redshift's built-in aggregate and scalar functions on the `HLLSKETCH` values to combine multiple `HLLSKETCH` values to calculate weekly totals.

SUPER type

To support semi-structured data (such as arrays and JSON data) more efficiently in Redshift, Amazon provides the `SUPER` data type. You can load up to 1 MB of data into a column that is of the `SUPER` type, and then easily query the data without needing to impose a schema first.

For example, if you're loading `JSON` data into a `SUPER` data type column, you don't need to specify the data types of the attributes in the JSON document. When you query the data, dynamic typing is used to determine the data type for values in the JSON document.

The `SUPER` data type offers significantly increased performance for querying semi-structured data versus unnesting the full JSON document and storing it in columns. If the JSON document contains hundreds of attributes, the increase in performance can be significant.

Amazon did announce support for up to 16 MB of data in the `SUPER` type; however, at the time of writing, this functionality is still in preview. Also, this functionality currently has a number of known limitations, such as not supporting data sharing, elastic resize, or query federation if a table has a `SUPER` type that holds objects larger than 1 MB.

Selecting the optimal table type

Redshift supports several different types of tables. Making use of a variety of table types for different purposes can help significantly increase query performance. Here, we will look at the different types of tables and discuss how each type can affect performance.

Local Redshift tables

The most common and default table type in Redshift is a table that is permanently stored on Redshift-managed storage. For RA3 node types, this includes local SSD drives as well as Amazon S3, while with DC2 node types, this would be local SSD storage.

One of the biggest advantages of a lake house architecture is the performance enhancement of placing hot data on high-performance local drives, along with high-network bandwidth and a large high-speed cache, as available in Redshift. With RA3 nodes, Redshift uses Amazon S3-based storage, while automatically loading frequently queried data to local storage, in order to optimize performance (as we cover in more detail later in this section).

Redshift stores data in a columnar data format, which is optimized for analytics and uses compression algorithms to reduce disk lookup time when a query is run. By using ML-based automatic optimizations related to table maintenance tasks such as vacuum, table sort, selection of distribution, and sort keys, as well as workload management, Redshift can turbo-charge query performance.

While the best performance is gained by coupling compute and storage, it can result in an unnecessary increase in cost when you need to scale out either just compute or storage. To solve this, Amazon introduced RA3 nodes with **Redshift Managed Storage** (**RMS**), which provides the best of both worlds. RA3 nodes offer tightly coupled compute with high-performance SSD storage, as well as additional S3-based storage that can be scaled separately. No changes need to be made to workflows to use these nodes, as Redshift automatically manages the

movement of data between the local storage and S3-managed storage based on data access patterns.

With RMS, data is initially stored on the local SSD drives, but if data on a node exceeds the space available on the SSD storage, Redshift automatically moves colder data to Amazon S3. Redshift uses an advanced algorithm to determine what data should be stored on the local SSD drives and what data is moved to Amazon S3, and automatically moves data between the two storage tiers as required. Some of the factors that the algorithm takes into account include data block temperature, data block age, and workload patterns.

External tables for querying data in Amazon S3 with Redshift Spectrum

To take advantage of our data lake (which we consider to be our single source of truth), Redshift supports the concept of external tables. These tables are effectively schema objects in Redshift that point to database objects in the **AWS Glue Data Catalog** (or optionally an **Amazon EMR Hive metastore**).

Once we have created the external schema in Redshift that points to a specific database in the Glue Data Catalog, we can then query any of the tables that belong to that database, and **Redshift Spectrum** will access the data from the underlying Amazon S3 files. Note that while Redshift Spectrum does offer impressive performance for reading large datasets from Amazon S3, it will generally not be quite as fast as reading that same dataset if it were stored within RMS.

By accessing the data directly from our S3 data lake, we avoid replicating multiple copies of the data across our data warehouse clusters. However, we still get to take advantage of the **Massively Parallel Processing** (**MPP**) query engine in Redshift to query the data. With Redshift Spectrum, we can still get impressive performance while di-

rectly accessing our single-source-of-truth data lake data, without needing to constantly load and refresh data lake datasets into Redshift.

When running queries in Redshift, we are free to run complex joins on data between local and external tables. We can also query data (or a subset of data) from an external S3 table, and then write that data out to a local Redshift table when we want to make a specific dataset, or portion of a dataset, available locally in Redshift for optimal query performance.

A common use case for Redshift Spectrum is where a company knows that 80% of their queries access data generated in the past 12 months, but that 20% of their queries also rely on accessing historical data from the past 5 years. In this scenario, the past 12 months of data can be loaded into Redshift on a rolling basis and queried with optimal performance. However, the smaller portion of queries that need historical data can read that data from the data lake using Redshift Spectrum, with the understanding that reading historical data may not be quite as fast as reading data from the past 12 months.

Redshift Spectrum also supports reading data in Amazon S3 that uses **Open Table Formats**, such as Delta Lake and Apache Hudi. Support for reading Apache Iceberg data with Redshift Spectrum is available in preview at the time of writing (and we will cover Open Table Formats in more detail in *Chapter 14*, *Building Transactional Data Lakes*).

Another common use case for external tables is to enable Redshift to read data from file formats that are not natively supported in Redshift, such as **Amazon Ion**, **Grok**, **RCFile**, and **Sequence** files.

An important point to keep in mind when planning your use of external tables is that the cost of Redshift Spectrum, when run from provisioned clusters, is based on the amount of data that's scanned by a Spectrum query, in addition to the fixed costs of the cluster. With Redshift Serverless, cluster cost is based on the RPUs consumed by

queries you run on the cluster, whether the queries are on local tables, or whether they use Redshift Spectrum to query data in the data lake.

Also, while query performance with Redshift Spectrum is often impressive, it still may not match the performance when querying data stored locally in RMS. Therefore, you should consider loading frequently queried data directly into RMS, rather than only relying on external tables. This is especially true for datasets that are used for tasks such as constantly refreshing dashboards, datasets that are frequently queried by a large group of users, or where you need to do a large number of joins across tables.

In the hands-on section of this chapter, we will configure a Redshift Spectrum external table and query data from that table using both Redshift and Athena.

Temporary staging tables for loading data into Redshift

Redshift, like many other data warehousing systems, supports the concept of a **temporary table**. Temporary tables are session-specific, meaning that they are automatically dropped at the end of a session and are unrecoverable.

However, temporary tables can significantly improve the performance of some operations as temporary tables are not replicated in the same way permanent tables are, and inserting data into temporary tables does not trigger automatic cluster incremental backup operations. One of the common uses of temporary tables (also sometimes referred to as staging tables) is for updating and inserting data into existing tables.

Traditional transactional databases support an operation called an `UPSERT`, which is useful for **Change Data Capture** (**CDC**). An `UPSERT` transaction reads new data and checks if there is an existing matching

record based on the primary key. If there is an existing record, the record is updated with the new data, and if there is no existing record, a new record is created.

While Redshift does support the concept of primary keys, this is for informational purposes and is only used by the query optimizer. Redshift does not enforce unique primary keys or foreign key constraints. As a result, the `UPSERT` SQL clause is not supported natively in Redshift.

However, in April 2023, Amazon Redshift announced support for the `MERGE` command, which enables applying source data changes into a Redshift table using a simple SQL statement. A common approach is to load the latest snapshot of data from a source system into a temporary table in Redshift, and then use the `MERGE` command to merge changes in the dataset into a target table.

Data caching using Redshift materialized views

Data warehouses are often used as the backend query engine for BI solutions. A visualization tool such as Amazon QuickSight (which we will discuss in more detail in *Chapter 12*, *Visualizing Data with Amazon QuickSight*) can be used to build dashboards based on data stored in Amazon Redshift (and other data sources).

The dashboards are accessed by different business users to visualize, filter, and drill down into different datasets. Often, the queries that are needed to create a specific visualization will need to reference and join data from multiple Redshift tables, and potentially perform aggregations and other calculations on the data.

Instead of having to rerun the same query over and over as different users access the dashboards, you can effectively cache the query results by creating what is called a **materialized view**.

Materialized views increase query performance by orders of magnitude by precomputing expensive operations such as join results, arithmetic calculations, and aggregations, and then storing the results of the query in a view. The BI tool can then be configured to query the view, rather than querying the tables directly. From the perspective of the BI tool, accessing the materialized view is the same as accessing a table.

You can choose to either manually refresh a materialized view using the `REFRESH MATERIALIZED VIEW` statement, or you can configure a materialized view to be automatically refreshed when the underlying base table data changes. Note, however, that Redshift prioritizes other running workloads over auto-refresh, and therefore might not immediately refresh a materialized view if the system is under load.

A common use case for materialized views would be to store the results of the advanced queries and calculations needed to aggregate sales by store daily. Each night, the day's sales can be loaded into Redshift from the data lake, and on completion of the data ingest, a materialized view can be created or refreshed. In this way, the complex calculations and joins required to determine sales by store are run just once, and when users query the data via their BI tool, they access the results of the query through the materialized view.

Now that we've looked at the types of tables that are supported in Redshift, let's look at the best practices involved in ingesting data into Redshift.

Moving data between a data lake and Redshift

Moving data between a data lake and a data warehouse, such as Amazon Redshift, is a common requirement for many use cases. Data may be cleansed and processed with Glue ETL jobs in the data lake, for

example, and then hot data can be loaded into Redshift so that it can be queried via BI tools with optimal performance.

In the same way, there are certain use cases where data may be further processed in the data warehouse, and this newly processed data then needs to be exported back to the data lake so that other users and processes can consume this data.

In this section, we will examine some best practices and recommendations for both ingesting data from the data lake and exporting data back to the data lake.

Optimizing data ingestion in Redshift

While there are various ways that you can insert data into Redshift, the recommended way is to bulk ingest data using the Redshift `COPY` command. The `COPY` command enables optimized data to be ingested from the following sources:

- **Amazon S3**
- **Amazon DynamoDB**
- **Amazon Elastic MapReduce (EMR)**
- **Remote SSH hosts**

When running the `COPY` command, you need to specify an IAM role, or the access key and secret access key of an IAM user, that has relevant permissions to read the source (such as Amazon S3), as well as the required Redshift permissions. AWS recommends creating and using an IAM role with the `COPY` command.

When reading data from Amazon S3, Amazon EMR, or from a remote host via SSH, the `COPY` command supports various formats, including CSV, Parquet, Avro, JSON, ORC, and many others.

To take advantage of the multiple compute nodes in a cluster when ingesting files into a Redshift-provisioned cluster, you should aim to

match the number of ingest files with the number of slices in the cluster. Each slice of the cluster can ingest data in parallel with all the other slices in the cluster, so matching the number of files to the number of slices results in the maximum performance for the ingest operation, as shown in the following diagram:

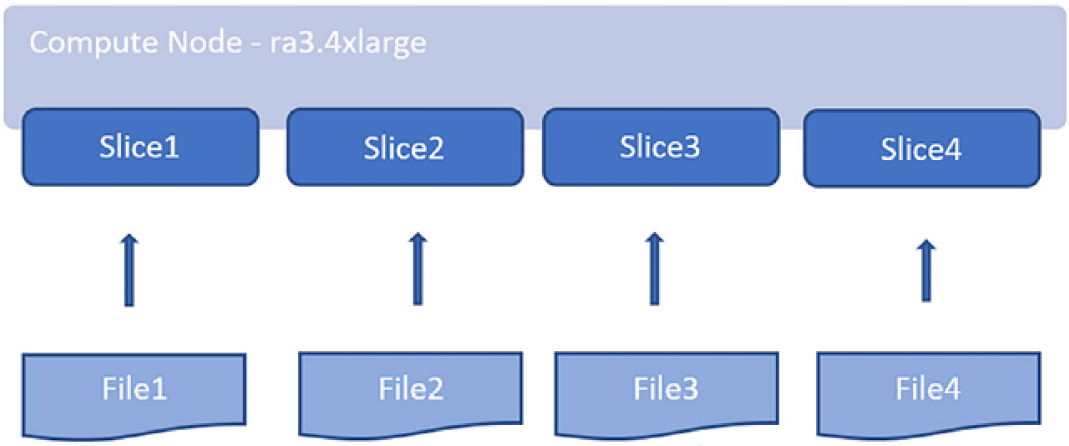

Figure 9.2: Slices in a Redshift compute node

If you have one large ingest file, it should be split into multiple files, with each file having a size between 1 MB and 1 GB (after compression). To determine how many slices you have in your cluster, refer to the AWS documentation on Redshift cluster configuration.

For example, if you had a cluster with `4 x ra3.4xlarge` nodes, you would have 16 slices (there are four slices per `ra3.4xlarge` node). If your ingest file were 64 GB in size, you would split the file into 64 x 1 GB files, and each of the slices in the cluster would then ingest a total of four files.

Note that when using the `COPY` command to ingest data, the `COPY` operation is treated as a single transaction across all files. If one of our 64 files failed to be copied, the entire copy would be aborted and the transaction would be rolled back.

While it is possible to use `INSERT` statements to add rows to a table, adding single rows, or just a few rows, using `INSERT` statements is not recommended. Adding data to a table using `INSERT` statements is significantly slower than using the `COPY` command to ingest data. If you do need to add data using `INSERT` statements, you can insert multiple rows with a single statement using multi-row insert, by specifying multiple comma-separated rows. You should add as many rows as possible with a single `INSERT` statement to improve performance and maximize how data blocks are stored.

When loading data from an Amazon EMR cluster, you can use the `COPY` command in Redshift and specify the EMR cluster ID and the HDFS path that the data should be loaded from. However, before doing this, you need to configure the nodes in the EMR cluster to accept SSH requests from your Redshift cluster, and you need to ensure the appropriate security groups have been configured to allow connections between Redshift and the EMR nodes.

Alternatively, you can directly load data into Redshift from a Spark application using the AWS-optimized Spark-Redshift JDBC driver, available in EMR 6.9, EMR Serverless, and Glue 4.0 and later. In the background, the Spark DataFrame you are loading is written to a temporary S3 bucket, and then a `COPY` command is executed to load the data into Redshift. You can also read data from Redshift into a Spark DataFrame by using the Spark-Redshift JDBC driver.

Automating data loads from Amazon S3 into Redshift

In November 2022, AWS announced the preview of new functionality in Redshift to perform an auto-copy of data from an S3 bucket into an Amazon Redshift cluster. Using this functionality, you can specify that a `COPY` command should be created as a job that will monitor an S3 location for new files, and as new files become available, these will automatically be loaded into the Redshift table that you specify.

As of the time of writing, this functionality is still in preview and has some limitations, such as not supporting the loading of Parquet and ORC files. Please refer to the latest Amazon Redshift documentation for the latest supported features for `COPY` jobs.

Exporting data from Redshift to the data lake

Similar to how the `COPY` command can be used to ingest data to Redshift, you can use the `UNLOAD` command to copy data from a Redshift cluster to Amazon S3.

To maximize the performance of `UNLOAD`, Redshift uses multiple slices in the cluster to write out data to multiple files simultaneously. Each file that is written can be a maximum size of 6.2 GB, although there is an option to specify a smaller maximum file size (and this also gives some control over the number of files that are written out). Depending on the size of the dataset you are unloading, it would generally be recommended to specify a `MAXFILESIZE` option of 1 GB.

When running the `UNLOAD` command, you specify a `SELECT` query to determine what data will be unloaded. To unload a full single table, you would specify `SELECT * from TABLENAME` in your `UNLOAD` statement. However, you could use more advanced queries in the `UNLOAD` statement, such as a query that joins multiple tables, or a query that uses a `WHERE` clause to unload only a subset of the data in a table. It is recommended that you specify an `ORDER BY` clause in the query, especially if you plan to load the data back into Redshift.

By default, data is unloaded in a pipe-delimited text format, but unloading data in Parquet format is also supported. For most use cases where you're exporting data to a data lake, it is recommended to specify the Parquet format for the unloaded data. The Parquet format is optimized for analytics, is compressed (so it uses less storage space in S3), and the `UNLOAD` performance can be up to twice as fast when unloading in Parquet format versus unloading in text format.

If you're performing an `UNLOAD` on a specific dataset regularly, you can use the `ALLOWOVERWRITE` option to allow Redshift to overwrite any existing files in the specified path. Alternatively, you can use the `CLEANPATH` option to remove any existing files in the specified path before writing data out.

Another best practice recommendation for unloading large datasets to a data lake is to specify the `PARTITION` option and to provide one or more columns that the data should be partitioned by. When writing out partitioned data, Redshift will use the standard Hive partitioning format. For example, if you partition your data by the `year` and `month` columns, the data will be written out as follows:

```
s3://unload_bucket_name/prefix/year=2021/month=July/000.parquet
```

When using the `PARTITION` option with the `CLEANPATH` option, Redshift will only delete files for the specific partitions that it writes out to.

Let's now look at some of the other advanced features available in Redshift, before moving on to the hands-on part of this chapter.

Exploring advanced Redshift features

While Redshift was first launched a long while back (2013), AWS is continually adding new functionality and features to Redshift. In this section, we are going to look at some of the advanced capabilities launched in the past few years that can help you get the most from your Redshift cluster. You should also regularly review the AWS *What's New* page for Redshift (**https://aws.amazon.com/redshift/whats-new/**), as well as AWS blog posts tagged with Redshift (**https://aws.amazon.com/blogs/big-data/tag/amazon-redshift/**), to ensure you keep up to date with new features.

Data sharing between Redshift clusters

There are a number of use cases where you may want to share data from one Redshift cluster with data in another (or multiple other) Redshift cluster. For example, if you implement a data mesh architecture, you may want to make data available from one part of the business easily accessible for other parts of the business without needing to create a duplicate copy of the data. For this use case, a data producer in one part of the business can share data from their Redshift RA3 cluster with the Redshift cluster of data consumers in another part of the business that have been authorized to access the data.

Another example of a use case for sharing data between clusters is where you may want to have the compute resources of a Redshift cluster focused just on data ingestion, transforming data, creating materialized views, etc. without having an impact on consumers who want to query the same data. In this case, you can have a dedicated ELT cluster that does the ingestion and transformation, and then launch a separate cluster that your data consumers will use to query the data. With this scenario, you can share the data from the *ELT cluster* with the *consumer cluster*, enabling each use case to have its own dedicated Redshift compute resources. If you ingest and transform data only once a day, you can even pause the *ELT cluster* once the ingestion and transformation are complete, and the *consumer cluster* will still be able to read the data from the paused *ELT cluster*.

In a similar way, you may have multiple different teams wanting to run queries on the same dataset. Traditionally these teams would end up competing for cluster resources, and it would not be easy to proportionally allocate costs to the different teams based on their usage of the cluster. However, with Redshift data sharing, each team can have its own Redshift cluster, with dedicated resources, and yet they can all access the same dataset. This also means that it is much simpler to allocate data warehousing costs for each team, as they each have their own cluster.

With Amazon Redshift data sharing, you can share live data from a source RA3 Redshift cluster with a target RA3 Redshift cluster in the same, or different, AWS account, as well as across regions (for example, a Redshift cluster in the N. Virginia us-east-1 Region can share data with a Redshift cluster in the Ohio us-east-2 Region). As of the time of writing, data that is shared with another cluster is available in the target cluster as read-only data (i.e., the cluster accessing shared data cannot write or update any of the shared tables). Data that is shared is shared "live," meaning that the target cluster sees the data in its current, most updated form, as it is updated in the source cluster.

You can select to either manage data sharing in Redshift directly, or you can use the AWS Lake Formation service to define and enforce Redshift data shares. This includes the ability to define column and row-level access permissions for shared tables. You can learn more about managing Redshift data shares with Lake Formation in the AWS documentation at `https://docs.aws.amazon.com/redshift/latest/dg/lake-formation-datashare.html`.

Machine learning capabilities in Amazon Redshift

Amazon Redshift includes functionality to integrate with the **Amazon SageMaker** ML service. This functionality, called Redshift ML, enables you to create and train new ML models on data in Redshift, and to perform inference (getting a prediction from the model) as part of a Redshift SQL query.

When you use the Redshift ML functionality to create a new model based on data in Redshift, the data will be automatically exported to Amazon S3, and SageMaker AutoML will then be used to train a new model and make that available in Redshift. Note that using this functionality incurs S3 storage costs for the data exported, as well as Amazon SageMaker costs for the model training.

An example of how you can use this functionality is to use a table in Redshift that contains customer information such as their ZIP/postal code, how long they have had an account with your company, the total revenue you have billed them for, as well as the number of customer service calls they have made. In addition, to train the new model, your dataset should include a field that indicates whether they are still an active customer or not. Based on this data, you can create a model that will predict whether a customer (or group of customers) is likely to cancel their service (often referred to as customer churn).

With the above dataset, you can use the Redshift `CREATE MODEL` statement in a SQL query, and Redshift will export the dataset to S3, and then use the Amazon SageMaker Autopilot feature to train the model. Once trained and validated, Amazon SageMaker will deploy the model and prediction function to Amazon Redshift.

Once complete, you can use SQL statements to provide similar information used to train the model (ZIP code, account length, spend, and customer service calls) to receive a prediction on whether a customer (or group of customers) is likely to cancel their service.

For more information on Redshift ML, see the Amazon Redshift documentation at https://docs.aws.amazon.com/redshift/latest/dg/machine_learning.html.

Running Redshift clusters across multiple Availability Zones

For some users, the risk of a Redshift cluster being unavailable for a period of time in the event of an issue with an AWS Availability Zone (AZ) is not a significant risk. In many cases, data warehouse-based queries and reports are not considered critical dependencies for the business, and a business may be willing to take the risk of a certain period of downtime of their Redshift cluster.

With Redshift Serverless, you always configure the serverless cluster with subnets in 3 different availability zones, and this helps ensure the continued availability of the serverless endpoint in the event of an issue impacting an availability zone.

With RA3-provisioned nodes, a feature called cluster relocation is enabled by default, and this allows Redshift to automatically relocate a cluster to a different availability zone in the event of issues impacting cluster operations in the current availability zone. While this can take place automatically, there is a period of downtime while the cluster is relocated. Note however that after relocation, the cluster can still be accessed using the same endpoint (therefore applications do not need to be reconfigured to point to a new endpoint after relocation).

However, there may be other use cases where a business depends heavily on the availability of its data warehouse for running queries and reports and is willing to accept increased costs in exchange for increased availability.

In November 2022, AWS announced the preview availability of a new feature that enables Redshift RA3 clusters to be deployed across multiple AZs. When configuring an RA3 Redshift cluster in multi-AZ mode, you specify the number of nodes you want in a single AZ, and Redshift will deploy those number of nodes in each of the two AZs. All nodes across the AZs can perform read and write workload processing during normal operation.

With multi-AZ, data is stored in RMS, which uses Amazon S3 and makes the data available simultaneously in both AZs where the compute nodes are running. If multiple queries are running against the multi-AZ Redshift cluster, the queries will run on compute nodes in both AZs; however, each individual query will use compute resources in a single AZ only.

Therefore, you need to ensure that you have enough compute nodes in each AZ to be able to handle your most complex queries.

Learn more about running Redshift across multi-AZs by reading the documentation at https://docs.aws.amazon.com/redshift/latest/mgmt/managing-cluster-multi-az.html.

Redshift Dynamic Data Masking

In addition to Redshift data access controls that enable row-level security and column-level security, you can also define **Dynamic Data Masking** (**DDM**) policies to further protect sensitive data stored in Redshift. Using this functionality, you can define policies that mask data values at the time that data is returned to a user. This enables you to protect sensitive data returned in queries, without needing to transform and store the data in Redshift in a protected format.

For example, if you store confidential PII data (such as a national identity number or social security number), you can store the full data in Redshift but can ensure that when users query the data, they are not able to access the PII data directly. The policies you define can completely or partially redact the data, or can apply a hashing function to return a hash instead of the actual data. For example, using this functionality, you could configure a policy that returns just the last 4 digits of a social security number in a query, instead of displaying the full social security number. And you can apply the policy to different user roles, so that some users may be able to access the full number, while other users only see the last 4 digits.

Read more about DDM in the Redshift documentation at https://docs.aws.amazon.com/redshift/latest/dg/t_ddm.html.

Zero-ETL between Amazon Aurora and Amazon Redshift

In November 2022, AWS announced the preview support of **zero-ETL** integration between Amazon Aurora and Amazon Redshift. With this functionality, data written to Amazon Aurora can be made available in Amazon Redshift within seconds.

This functionality enables you to analyze data from multiple Amazon Aurora database clusters in an Amazon Redshift cluster, in near real time. Previously, this would have required more complex solutions using ETL pipelines and services such as AWS **Database Migration Service (DMS)** to move data from an Aurora database into Redshift. Using this functionality, you can use Amazon Redshift's advanced analytics and ML capabilities to gain new insights from transactional data.

In the preview announcement, AWS announced zero-ETL support for Amazon Aurora MySQL 3 (with MySQL 8.0 compatibility). At the time of writing, support for Amazon Aurora with PostgreSQL compatibility was not available.

Resizing a Redshift cluster

There may be times when you need to resize your Redshift cluster, either permanently (due to sustained increased load on the cluster) or temporarily (to handle month-end data processing, for example). Redshift enables this through a process known as **elastic resize**.

With elastic resize, you can add or remove nodes from a cluster, or change the node type (for example, from DC2 nodes to RA3 nodes). Elastic resize usually completes within around 10–15 minutes, and while the resize operation takes place, the cluster is in read-only mode. During the process, queries may be temporarily paused, and it is possible that some queries may be dropped. Also, if the cluster has been configured to share data with other clusters, those other clusters may not be able to connect and access data during a portion of the resize.

Elastic resize is recommended when you need to resize a cluster, but there are times when an elastic resize may not be supported for a specific cluster configuration. In those cases, you can perform a classic resize. This process takes longer, but it does support scenarios where elastic resize does not work, such as where you have a cluster that does not use KMS encryption, and you want to now encrypt your cluster.

Now that you have a good understanding of the Redshift architecture, important considerations for optimizing the performance of your cluster, and an understanding of some of the advanced features in Redshift, it is time to get hands-on with deploying a Redshift cluster.

Hands-on – deploying a Redshift Serverless cluster and running Redshift Spectrum queries

In our Redshift hands-on exercise, we're going to create a new **Redshift Serverless** cluster and configure **Redshift Spectrum** so that we can query data in external tables on Amazon S3. We'll then use both Redshift Spectrum and Amazon Athena to query data in S3.

Uploading our sample data to Amazon S3

For this exercise, we are going to use some data generated with a service called Mockaroo (`https://www.mockaroo.com/`). This service enables us to generate fake data with a wide variety of field types and is useful for demos and testing.

We will upload this dataset, containing a list of users, to Amazon S3 and then query it using Redshift Spectrum. Note that all data in this file is fake data, generated with the tool mentioned above. Therefore, the names, email addresses, street addresses, phone numbers, etc. in this dataset are not real.

Let's get started:

1. Download the fake list of users from the GitHub site for this book using the following link: https://github.com/PacktPublishing/Data-Engineering-with-AWS-2nd-edition/blob/main/Chapter09/user_details.csv.
2. Log in to the **AWS Console** and access the **Amazon S3** service. Navigate to your landing zone bucket (for example, `dataeng-landing-zone-initials`) and create a new prefix called `users`.
3. Upload the `user_details.csv` file into the data lake's landing zone bucket (which you created in *Chapter 3, The AWS Data Engineers Toolkit*) under the `users` prefix. For example:

```
s3://dataeng-landing-zone-gse23/users/user_details.csv
```

4. To verify that the files have been uploaded correctly, we can use **S3 Select** to directly query uploaded files. In the **Amazon S3 console**, navigate to the `users` prefix in the landing zone bucket, and select the `user_details.csv` file. From the **Actions** menu, click on **Query with S3 Select**. Leave all the options as their defaults and click **Run SQL query**. This will display a few records from the CSV file.

Having uploaded our users file to the data lake, we now need to create the IAM roles that our Redshift cluster will use, and then create the cluster.

IAM roles for Redshift

Amazon Redshift Spectrum enables our cluster to read data that is in our Amazon S3-based data lake directly, without needing to load the data into the cluster. Redshift Spectrum uses the AWS Glue Data Catalog, so it requires AWS Glue permissions in addition to Amazon S3 permissions. If you are operating in an AWS Region where AWS Glue

is not supported, then Redshift Spectrum uses the Amazon Athena catalog, so you would require Amazon Athena permissions.

To create the IAM role that grants the required Redshift Spectrum permissions, follow these steps:

1. Navigate to the **AWS IAM Management console**, click on **Roles** on the left-hand side, and click on **Create role**.
2. Ensure that **AWS service** is selected for **Select type of trusted entity**, and then search for **Redshift** in the drop-down list. For **Use cases for other AWS services**, select **Redshift – Customizable**. Click on **Next**.
3. Attach the following four policies to the role:
    1. `AmazonS3FullAccess`
    2. `AWSGlueConsoleFullAccess`
    3. `AmazonAthenaFullAccess`
    4. `AmazonRedshiftAllCommandsFullAccess`

IMPORTANT NOTE ABOUT PERMISSIONS

The preceding policies provide broad access to various AWS services, including full access to all S3 files in your account. If you are using an account created specifically for the hands-on exercises in this book, or you are using a limited sandbox account provided by your organization, then these permissions may be safe. However, in an AWS account that is shared with others, such as a corporate production account, you should not use these policies. Instead, you should create new policies that, for example, limit access to only the S3 buckets that are used in the hands-on exercises. *Using full access policies, as we have here, is not a good security practice for shared or production accounts*.

---

4. Then, click on **Next** and provide a **Role name**, such as `AmazonRedshiftSpectrumRole`. Make sure that the three policies listed in *Step 3* are included and that **Trusted entities** includes `redshift.amazonaws.com`. Once confirmed, click **Create role**.
5. Search for the role you just created and click on the role's name. On the **Summary** screen, take note of **Role ARN** as this will be needed later.

Now that we have created an IAM role that provides the permissions needed for Redshift Spectrum to access the required resources, we can move on to creating our cluster.

Creating a Redshift cluster

We are now ready to create our **Redshift Serverless** cluster and attach the IAM policy for Redshift Spectrum to the cluster. Let's get started:

IMPORTANT NOTE ABOUT REDSHIFT COSTS

At the time of writing, AWS offers a free trial for new Redshift Serverless customers, enabling you to create and test out a new Redshift Serverless cluster with $300 of credit that can be used over 90 days. However, this is only available if your organization has not previously created a Redshift Serverless cluster. If you've previously created an Amazon Redshift Serverless cluster, you are not eligible for the free trial and your usage of Redshift Serverless will be billed for. If you are eligible for the free trial but you leave your Redshift cluster running beyond the free trial time limit, you will be charged for usage of the cluster. For more information, see https://aws.amazon.com/redshift/free-trial/.

---

1. Navigate to the **Amazon Redshift** console at https://console.aws.amazon.com/redshiftv2/. If you have not created a cluster before, there should be a button marked **Try Redshift Serverless free trial** that you can click in order to quickly configure a new serverless cluster. The option for **Use default settings** should already be checked, which enables us to create a cluster with default settings, so leave this option selected. If **Try Redshift Serverless free trial** is not displayed, then click on **Create workgroup** and provide a workgroup and namespace name, and set the base capacity to `8` RPUs.
2. Scroll down to **Associated IAM roles** and click on **Associate IAM role**. Search for the role you created previously (such as `AmazonRedshiftSpectrumRole`) and select the role, then click **Associate IAM roles**.
3. Select the role you just associated, and then click **Set default** and **Make default**. Click on **Confirm** when prompted.
4. Scroll down to the bottom of the page and click on **Save configuration**.
5. This will now create your Amazon Redshift Serverless cluster and should complete in under 5 minutes. Once complete, click on **Continue**.

You should be redirected to the **Amazon Redshift Serverless** dashboard, which includes information on the default namespace and workgroup you just created (both called **default**), and also includes information on how much of the $300 in credits is still available, if you qualify for the free trial.

Querying data in the sample database

When a new Redshift Serverless cluster is created, it includes easy access to sample datasets that you can query as you explore the Redshift query editor and interface. The following steps show you how to access the sample database:

1. On the **Redshift Serverless dashboard**, click on the **Query data** button to launch the query editor.
2. The query editor will launch in a new window, and a list of work-groups will be displayed on the left. The **Serverless: default** work-group should be listed. Click to expand the workgroup name, which will display a popup for you to select an authentication method (since this is the first time you are connecting to the workgroup). Select the **Federated user** option, which uses your IAM credentials to generate temporary credentials for accessing the database. There is a default `dev` database that is created for new clusters, so leave `dev` set as the database to connect to, and click on **Create connection**.
3. Expand the `sample_data_dev` schema, and then click on the folder icon next to the `tickit` table. This loads the sample data into Redshift, but you need a database to store this data. Therefore, when you click on the folder icon (**Open sample notebooks**), you are prompted to allow Redshift to create a sample database. Click on **Create**. It may take a few minutes for the sample data to be loaded.
   The Redshift query editor includes notebook functionality, where you can write up a number of SQL statements, as well as blocks, where you can include comments using Markdown syntax. You can either click on **Run all** to run all the statements in the notebook, or you can click on **Run** in an individual block to run just a single statement.

4. Click on **Run all** to run all the statements in this notebook.
5. In the first block (**Sales per event**), there are two SQL statements, and as a result, there are two **Result** tabs below. The first statement just set the path to use the **tickit** schema, and therefore there is not a lot to show in the result box. Click on **Result 2** to view the results of the second statement, which queries the `sales` table and shows the `total_price` for each event in the database.

Up to this point we have loaded and queried some sample data using a query editor notebook, but let's now move on to defining an external schema where we can point to the data we uploaded at the start of the hands-on section. We can then query that data using **Redshift Spectrum**.

Using Redshift Spectrum to directly query data in the data lake

To query data in Amazon S3 using Redshift Spectrum, we need to define a database, schema, and table.

Note that Amazon Redshift and AWS Glue use the term *database* differently. In Amazon Redshift, a database is a top-level container that contains one or more schemas, and each schema can contain one or more tables. When you use the Redshift query editor, you specify the name of the database that you want to connect to, and any objects you create are created in that database. When you query a table, you specify the schema name along with the table name.

However, in AWS Glue, there is no concept of a schema, just a database, and tables are created in the database.

With the command shown in the following steps, we can create a new Redshift schema, defined as an external schema (meaning objects created in the schema will be defined in the AWS Glue Data catalog), and we specify that we want to create a new database in the Glue Data catalog called `users`. For Redshift to be able to write to the Glue Data Catalog and access objects in S3, we need to specify the ARN for the Redshift Spectrum role that we previously created, as this has access to all data in S3:

1. In the Redshift query editor v2 interface, click on the **PLUS (+)** sign, and then click on **Editor** to create a new tab that provides a simple query interface.

2. In the **database** dropdown, change the database from `sample_data_dev` to `dev`, so that our new schema is created in the database called `dev`. Then run the following command in the new tab to create a new external schema called `spectrum_schema`, and to also create a new database in the Glue Data catalog called `users`. Make sure to replace the `iam_role` ARN with the ARN you recorded previously when you created an IAM role for Redshift Spectrum:

```
create external schema spectrum_schema
from data catalog
database 'users'
iam_role 'arn:aws:iam::1234567890:role/AmazonRedshiftSpectrumRole'
create external database if not exists;
```

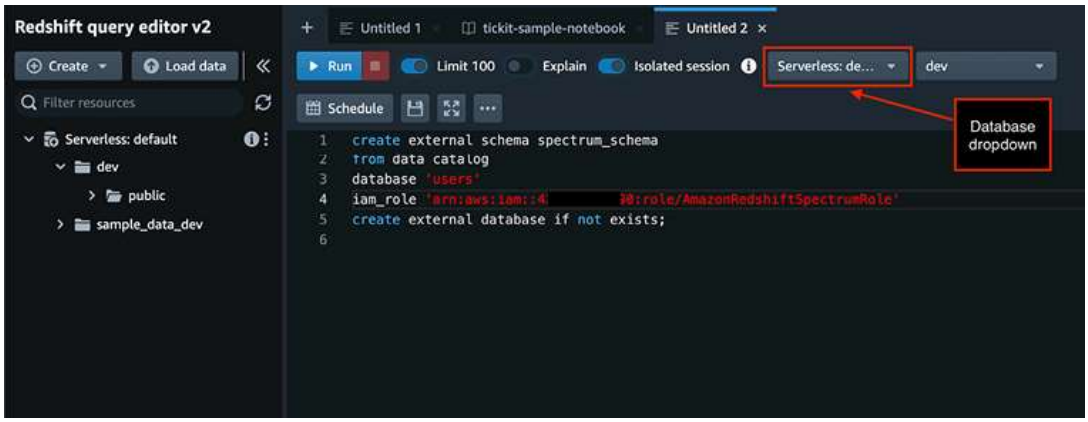

Figure 9.3: Creating a Redshift external schema

3. We can now define an **external table** that will be registered in the Glue Data Catalog under our `users` database in Glue and in the `spectrum_schema` in Redshift. When defining the table, we specify the columns that exist, the format of the files (text or comma delimited), and the location in S3 where the text files were uploaded. We also include a `table properties` attribute that indicates that the sample data has a header row, which we want to ignore. Replace the previous SQL statement with the following statement, and

make sure to replace the bucket name of the S3 location with the name of your data lake landing zone bucket:

```
CREATE EXTERNAL TABLE spectrum_schema.user_details(
  id INTEGER,
  first_name VARCHAR(40),
  last_name VARCHAR(40),
  email VARCHAR(60),
  gender VARCHAR(15),
  address_1 VARCHAR(80),
  address_2 VARCHAR(80),
  city VARCHAR(40),
  state VARCHAR(25),
  zip VARCHAR(5),
  phone VARCHAR(12)
)
row format delimited
fields terminated by ','
stored as textfile
location 's3://dataeng-landing-zone-initials/users/'
table properties ('skip.header.line.count'='1');
```

4. We can now run a query against the table to ensure that it was created successfully. In the left-hand navigation pane of the **Redshift query editor v2**, click on the **REFRESH** icon to update the interface to show the newly created objects. Under `dev / spectrum_schema / tables`, you should see the `user_details` table. Clicking on this table will show the table schema as we defined it. We can also query a sample of the data from the newly defined table by running the following query in the query editor (making sure that we have the `dev` database selected in the **database** dropdown of the query editor):

```
select * from spectrum_schema.user_details limit 10;
```

5. We can also confirm that the new external table was created successfully by checking the Glue Data Catalog. Search for and open

the **Glue** service in the AWS Console.

6. Click on **Databases**, and then click on the **users** database. In the list of tables, click on the **user_details** table.
7. To view a sample of the data in the table, click on **Actions**, and then click **View data**. If you receive a popup indicating that you will be using the Athena service to query the data and that there may be separate charges, click on **Proceed**. When the Athena console opens (in a new browser tab), you should see 10 records from our `user_details` table.

With the above tasks, we queried data in our S3 data lake using both Amazon Athena and Amazon Redshift (using the Redshift Spectrum functionality).

DELETE THE SERVERLESS CLUSTER

If you do not plan to explore additional Redshift functionality, you should delete the Redshift Serverless workgroup first, and then the namespace, in order to prevent unnecessary charges from being incurred over time.

Summary

In this chapter, we learned how a cloud data warehouse can be used to store hot data to optimize performance and manage costs (such as for dashboarding or other BI use cases). We reviewed some common "anti-patterns" for data warehouse usage before diving deep into the Redshift architecture to learn more about how Redshift optimizes data storage across nodes.

We then reviewed some of the important design decisions that need to be made when creating a Redshift cluster optimized for performance, before reviewing how to ingest data into Redshift and unload data from Redshift.

Finally, we reviewed some of the advanced features of Redshift (such as data sharing, DDM, and cluster resizing) before moving on to doing some hands-on exercises.

In the hands-on exercise portion of this chapter, we created a new Redshift Serverless cluster, explored some sample data, and configured Redshift Spectrum to query data from Amazon S3.

In the next chapter, we will discuss how to orchestrate various components of our data engineering pipelines.

Learn more on Discord

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:

https://discord.gg/9s5mHNyECd