7
Transforming Data to Optimize for Analytics

In previous chapters, we covered how to architect a data pipeline and common ways of ingesting data into a data lake. We now turn to the process of transforming raw data in order to optimize the data for analytics, enabling an organization to efficiently gain new insights into their data.

Transforming data to optimize for analytics and create value for an organization is one of the key tasks for a data engineer, and there are many different types of transformations. Some transformations are common and can be generically applied to a dataset, such as converting raw files to Parquet format and partitioning the dataset. Other transformations use business logic in the transformations and vary based on the contents of the data and the specific business requirements.

In this chapter, we review some of the engines that are available in **AWS** for performing data transformations and also discuss some of the more common data transformations. However, this book focuses on the broad range of tasks that a data engineer is likely to work on, so it is not intended as a deep dive into **Apache Spark**, nor is it intended as a guide to writing **PySpark** or **Scala** code. Even so, there are many other great books and online resources focused purely on teaching Apache Spark, and you are encouraged to investigate these, as knowing how to code and optimize Apache Spark is a common requirement for data engineers.

The topics we cover in this chapter include the following:

- Overview of how transformations can create value
- Types of data transformation tools
- Common data preparation transformations
- Common business use case transformations
- Working with **change data capture** (**CDC**) data
- Hands-on: Building transformations with **AWS Glue Studio** and **Apache Spark**

Technical requirements

For the hands-on tasks in this chapter, you need access to the AWS Glue service, including AWS Glue Studio. You also need to be able to create a new S3 bucket and new IAM policies.

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/Chapter07`

Overview of how transformations can create value

As we have discussed in various places throughout this book, data can be one of the most valuable assets that an organization owns. However, raw, siloed data has limited value on its own, and we unlock the real value of an organization's data when we combine various raw datasets and transform that data through an analytics pipeline.

Cooking, baking, and data transformations

Look at the following list of food items and consider whether you enjoy eating them:

- Sugar
- Butter
- Eggs
- Milk

For many people, these are pretty standard food items, and some (like the eggs and milk) may be consumed on their own, while others (like the sugar and the butter) are generally consumed with something else, such as adding sugar to your coffee or tea or spreading butter on bread.

But, if you take those items and add a few more (like flour and baking powder) and combine all the items in just the right way, you could bake yourself a delicious cake, which would not resemble the raw ingredients at all. In the same way, our individual datasets have value on their own to the part of the organization that they come from, but if we combine these datasets in just the right way, we can create something totally new and different.

Now, if you happen to be having a party to celebrate something, your guests will appreciate the cake far more than they would appreciate just having the raw ingredients laid out! But if your goal was to provide breakfast for your friends, you may instead choose to fry the eggs, make some toast and spread the butter on the toast, and offer the milk and sugar to your guests for them to add to their coffee.

In both cases, you're using some common raw ingredients, then adding some additional items, and finally using different utilities to prepare the food (an oven for the cake and a stovetop for the fried eggs). How you combine the raw ingredients, and what you combine them with, depends on whether you're inviting friends over for breakfast, or whether you're throwing a party and want to celebrate with a cake.

In the same way, data engineers can use the same raw datasets, combine them with additional datasets, process them with different analytics engines, and create totally new and different datasets. How they combine the datasets, and which analytics engine they use, depends on what they're trying to create, which, of course, ultimately depends on what the business purpose is.

For example, your marketing team may want to combine a dataset that lists sales of each product, for each day over the past year, with weather

data for the past year (min and max temperature, and whether it rained or not). This would enable them to analyze which products sell best on hot days, cold days, and rainy days, so that they can tailor their marketing campaigns around this.

Another example is having the marketing team aggregate their campaign data by region, and combine this with sales data aggregated by region and by product category, to measure the effectiveness of their marketing campaigns in different regions and across different product lines. This enables the team to optimize their marketing campaigns based on past performance.

Transformations as part of a pipeline

In *Chapter 5*, *Architecting Data Engineering Pipelines*, we developed a high-level design for our data pipeline. We first looked at how we could work with various business users to understand what their requirements were (to keep our analogy going, whether they wanted a cake or breakfast). After that, we looked at three broad areas on which we gathered initial information, namely the following:

1. **Data consumers**: Who was going to be consuming the data we created and what tools would they use for data gathering (our guests)?
2. **Data sources**: Which data sources did we have access to that we could use to create our new dataset (our raw ingredients)?
3. **Data transformations**: We reviewed, at a high level, the types of transformations that may be required in our pipeline in order to prepare and join our datasets (the recipe for making a cake or for fried eggs).

We now need to develop a low-level design for our pipeline transformations, which will include determining the types of transformations we need to perform, as well as which data transformation tools we will use. In the next section, we begin by looking at the types of transformation engines that are available.

Types of data transformation tools

As we covered in *Chapter 3*, *The AWS Data Engineer's Toolkit*, there are a number of AWS services that can be used for data transformation. We reviewed a number of these services in that chapter, so make sure to review it again, but in this section, we will look more broadly at the different types of data transformation engines.

Apache Spark

Apache Spark is an in-memory engine for working with large datasets, providing a mechanism to split a dataset among multiple nodes in a cluster for efficient processing. Spark is an extremely popular engine to use for processing and transforming big datasets, and there are multiple ways to run Spark jobs within AWS.

With Apache Spark, you can either process data in batches (such as on a daily basis or every few hours) or process near real-time streaming data using **Spark Streaming**. In addition, you can use **Spark SQL** to process data using standard SQL, and **Spark ML** for applying machine learning techniques to your data. With **Spark GraphX**, you can work with highly interconnected points of data to analyze complex relationships, such as for social networking applications.

Within AWS, you can run Spark jobs using multiple AWS services. **AWS Glue** provides a serverless way to run Spark, and **Amazon EMR** provides both a managed service for deploying a cluster for running Spark as well as a serverless option. In addition, you can use AWS container services (**ECS** or **EKS**) to run a Spark engine in a containerized environment or use a managed service from an AWS partner, such as **Databricks**.

Hadoop and MapReduce

Apache Hadoop is a framework consisting of multiple open-source software packages for working with large datasets and can scale from run-

ning on a single server to running on thousands of nodes.

Before Apache Spark, tools within the Hadoop framework – such as **Hive** and **MapReduce** – were the most popular way to transform and process large datasets.

Apache Hive provides a SQL-type interface for working with large datasets, while MapReduce provides a code-based approach to processing large datasets. Hadoop MapReduce is used in a similar way to Apache Spark, with the biggest difference being that Apache Spark does all processing in memory. Hadoop MapReduce on the other hand, makes extensive use of traditional disk-based reads and writes to interim storage during processing.

For use cases with massive datasets that cannot be economically processed in memory, Hadoop MapReduce may be better suited. However, for most use cases, Apache Spark provides significant performance benefits, as well as the ability to handle streaming data, access to machine learning libraries, and an API for graph computation with GraphX. While Apache Spark has become the leading big data processing solution in recent years, there are many legacy Hadoop systems still being used to process data on a daily basis.

There are also components of Hadoop that are still commonly used for Spark processing. The Apache Hive metadata store (Data Catalog) is used by Spark to map databases and tables to physical files in storage. For example, the AWS Glue catalog (which we have discussed previously) is an Apache Hive-compatible metastore.

Within AWS, you can run a number of Hadoop tools using the managed Amazon EMR service. Amazon EMR simplifies the process of deploying Hadoop-based infrastructure and supports multiple Hadoop tools, including **Hive**, **HBase**, **Yarn**, **Tez**, **Pig**, and many others.

SQL

Structured Query Language (**SQL**) is another common method used for data transformation. The advantage of SQL is that SQL knowledge and experience are widely available, making it an accessible form of performing transformations for many organizations. However, a code-based approach to transformations (such as using Apache Spark) can be a more powerful and versatile way of performing transformations.

When deciding on a transformation engine, a data engineer needs to understand the skill sets available in the organization, as well as the toolsets and ultimate target for the data. If you are operating in an environment that has a heavy focus on SQL, with SQL skill sets being widely available and Spark and other skill sets being limited, then using SQL for transformation may make sense (although GUI-based tools can also be considered).

However, if you are operating in an environment that has complex data processing requirements, and where latency and throughput requirements are high, it may be worthwhile to invest in skilling up to use modern data processing approaches, such as Spark.

While we mostly focus on data lakes as the target for our data in this book, there are times where the target for our data transformations may be a data warehousing system, such as **Amazon Redshift** or **Snowflake**. In these cases, an **Extract, Load, Transform** (**ELT**) approach may be used, where raw data is loaded into the data warehouse (the *extract and load* portion of ELT), and then the transformation of data is performed within the data warehouse using SQL.

Alternatively, toolsets such as Apache Spark may be used with SQL, through **Spark SQL**. This provides a way to use SQL for transformations while using a modern data processing engine to perform the transformations, rather than using a data warehouse. This allows the data warehouse to be focused on responding to end-user queries, while data transformation jobs are offloaded to an Apache Spark cluster. In this scenario, we use an ETL approach, where data is **extracted** to intermediatory storage, Apache Spark is used to **transform** the data, and data is

then **loaded** into a different zone of the data lake, or into a data warehouse.

Tools such as **AWS Glue Studio** provide a visual interface that can be used to design ETL jobs, including jobs that use SQL statements to perform complex transformations. This helps users who do not have Spark coding skills to run SQL-based transforms using the power of the Apache Spark engine.

GUI-based tools

Another popular method of performing data transformation is through the use of GUI-based tools that significantly simplify the process of creating transformation jobs. There are a number of cloud and commercial products that are designed to provide a drag-and-drop-type approach to creating complex transformation pipelines, and these are widely used.

These tools may not provide the versatility and performance that you can get from designing transformations with code, but they do make the design of ETL-type transformations accessible to those without advanced coding skills. Some of these tools can also be used to automatically generate transformation code (such as Apache Spark code), providing a good starting point for a user to further develop the code, reducing ETL job development time.

Within AWS, the **Glue DataBrew** service is designed as a visual data preparation tool, enabling you to easily apply transformations to a set of files. With Glue DataBrew, a user can select from a library of over 250 common transformations and apply relevant transformations to incoming raw files.

With this service, a user can clean and normalize data to prepare it for analytics or machine learning model development through an easy-to-use visual designer, without needing to write any code.

Another AWS service that provides a visual approach to ETL design is **AWS Glue Studio**, a service that provides a visual interface for developing Apache Spark transformations. This can be used by people who do not have any current experience with Spark and can also be used by those who do know Spark, as a starting point for developing their own custom transforms. With AWS Glue Studio, you can create complex ETL jobs that join and transform multiple datasets, and then review the generated code and further refine it if you have the appropriate coding skills.

Outside of AWS, there are also many commercial products that provide a visual approach to ETL design. Popular products include tools from **Informatica**, **Matillion**, **Stitch**, **Talend**, **Panoply**, **Fivetran**, and many others.

As we have covered in this section, there are multiple approaches and engines that can be used for performing data transformation. However, whichever engine or interface is used, there are certain data transformations that are commonly used to prepare and optimize raw datasets, and we'll look at some of these in the next section.

Common data preparation transformations

The first set of transformations that we look at are those that help prepare the data for further transformations later in the pipeline. These transformations are designed to apply relatively generic optimizations to individual datasets that we are ingesting into the data lake. For these optimizations, you may need some understanding of the source data system and context, but, generally, you do not need to understand the ultimate business use case for the dataset.

Protecting PII data

Often, datasets that we ingest may contain **personally identifiable information (PII)** data, and there may be governance restrictions on

which PII data can be stored in the data lake. As a result, we need to have a process that protects the PII data as soon as possible after it is ingested.

There are a number of common approaches that can be used here (such as tokenization or hashing), each with its own advantages and disadvantages, as we discussed in more detail in *Chapter 4*, *Data Governance, Security, and Cataloging*. But whichever strategy is used, the purpose is to remove the PII data from the raw data and replace it with a value, or token, in a way that enables us to still use the data for analytics.

This type of transformation is generally the first transformation performed for data containing PII, and in many cases, it is done in a different zone of the data lake, designed specifically for handling PII data. This zone will have strict controls to restrict access for general data lake users, and the best practice would be to have the anonymizing process run in a totally separate AWS account. Once the transformation has anonymized the PII data, the anonymized files will be copied into the general data lake raw zone in the main processing account.

Depending on the method you want to use to transform PII data for anonymization, there may be multiple different toolsets that can be used. For example, in AWS both **AWS Glue Studio** and **AWS Glue DataBrew** can be used to detect and obfuscate PII data.

AWS Glue DataBrew provides more options for obfuscating the data. For example, with Glue DataBrew, you can redact data (replace PII data with a string of ######), replace/swap data (jumble the values in a column so that every value is moved to a different, random row), encrypt, or hash a PII value. With AWS Glue Studio, you can either redact or hash PII data.

Alternatively, for more complex use cases, you can use purpose-built managed services from commercial vendors that run in AWS, such as **PK Privacy from the company PKWARE**.

Optimizing the file format

Within modern data lake environments, there are a number of file formats that can be used that are optimized for data analytics. From an analytics perspective, the most popular file format currently is **Apache Parquet**.

Parquet files are column-based, meaning that the contents of the file are physically stored to have data grouped by columns, rather than grouped by rows as with most file formats (CSV files, for example, are physically stored to be grouped by rows). As a result, queries that select a set of specific columns (rather than the entire row) do not need to read through all the data in the Parquet file to return a result, leading to performance improvements.

Parquet files also contain metadata about the data they store. This includes schema information (the data type for each column), as well as statistics such as the minimum and maximum value for a column contained in the file, the number of rows in the file, and so on.

A further benefit of Parquet files is that they are optimized for compression. A 1 TB dataset in CSV format could potentially be stored as 130 GB in Parquet format once compressed. Parquet supports multiple compression algorithms, although **Snappy** is the most widely used compression algorithm.

These optimizations result in significant savings in terms of storage space used, and increased performance when running queries.

For example, the cost of an Amazon Athena query is based on the amount of compressed data scanned (at the time of writing, this cost was $5 per TB of scanned data). If only certain columns are queried of a Parquet file, then between the compression and only needing to read the data chunks for the specific columns, significantly less data needs to be scanned to resolve the query.

In a scenario where your data table is stored across perhaps hundreds of Parquet files in a data lake, the analytics engine is able to get further performance advantages by reading the metadata of the files. For example, if your query is just to count all the rows in a table, this information is stored in the Parquet file metadata, so the query doesn't need to actually scan any of the data. For this type of query, you will see that Athena indicates that 0 KB of data was scanned, therefore there is no cost for the query.

Or, if your query is for where the sales amount is above a specific value, the analytics engine can read the metadata for a column to determine the minimum and maximum values stored in the specific data chunk. If the value you are searching for is higher than the maximum value recorded in the metadata, then the analytics engine knows that it does not need to scan that specific column's data chunk. This results in both cost savings and increased performance for queries.

Because of these performance improvements and cost savings, a very common transformation is to convert incoming files from their original format (such as CSV, JSON, XML, and so on) into the analytics-optimized Parquet format.

Optimizing with data partitioning

Another common approach for optimizing datasets for analytics is to **partition** the data, which relates to how the data files are organized in the storage system for a data lake.

Hive partitioning splits the data from a table to be grouped together in different folders, based on one or more of the columns in the dataset. While you can partition the data based on any column, a common partitioning strategy that works for many datasets is to partition based on date.

For example, suppose you had sales data for the past four years from around the country, and you had columns in the dataset for **Day**,

Month, and **Year**. In this scenario, you could select to partition the data based on the **Year** column.

When the data was written to storage, all the data for each of the past few years would be grouped together with the following structure:

```
datalake_bucket/year=2023/file1.parquet
datalake_bucket/year=2022/file1.parquet
datalake_bucket/year=2021/file1.parquet
datalake_bucket/year=2020/file1.parquet
```

If you then run a SQL query and include a `WHERE Year = 2020` clause, for example, the analytics engine only needs to open up the single file in the `datalake_bucket/year=2020` folder. Because less data needs to be scanned by the query, it costs less and completes quicker. Note that for most datasets there will be multiple Parquet files per partition, but each file would only contain data related to the partitioned year.

Deciding on which column to partition by requires that you have a good understanding of how the dataset will be used. If you partition your dataset by year but a majority of your queries are by the **business unit (BU)** column across all years, then the partitioning strategy would not be effective.

Queries you run that do not use the partitioned columns may also end up causing those queries to run slower if you have a large number of partitions. The reason for this is that the analytics engine needs to read data in all partitions, and there is some overhead in working between all the different folders. If there is no clear common query pattern, it may be better to not even partition your data. But if a majority of your queries use a common pattern, then partitioning can provide significant performance and cost benefits.

You can also partition across multiple columns. For example, if you regularly process data at the day level, then you could implement the following partition strategy:

```
datalake_bucket/year=2021/month=6/day=1/file1.parquet
```

This significantly reduces the amount of data to be scanned when queries are run at the daily level and also works for queries at the month or year level. However, another warning regarding partitioning is that you want to ensure that you don't end up with a large number of small files. The optimal size of each Parquet file in a data lake is between 128 MB and 1 GB. The Parquet file format can be split, which means that multiple nodes in a cluster can process data from a file in parallel. However, having lots of small files requires a lot of overhead for opening files, reading metadata, scanning data, and closing each file, and can significantly impact performance. Therefore, it is better to have fewer partitions with larger files than to have hundreds, or thousands, of partitions but each partition only has a single file that is a few MB in size.

Partitioning is an important data optimization strategy and is based on how the data is expected to be used, either for the next transformation stage or for the final analytics stage. Determining the best partitioning strategy requires that you understand how the data will be used next.

Data cleansing

Optimizing the data format and partitioning data are transformation tasks that work on the format and structure of the data but do not directly transform the data. Data cleansing, however, is a transformation that alters parts of the data.

Data cleansing is often one of the first tasks to be performed after ingesting data and helps ensure that the data is valid, accurate, consistent, complete, and uniform. Source datasets may be missing values in some rows, have duplicate records, have inconsistent column names, use different formats, and so on. The data cleansing process works to resolve these issues on newly ingested raw data to better prepare the data for analytics. While some data sources may be nearly completely clean on

ingestion (such as data from a relational database), other datasets are more likely to contain data needing cleansing, such as data from web forms, surveys, manually entered data, or **Internet of Things** (**IoT**) data from sensors.

Some common data transformation tasks for data cleansing include the following:

1. **Ensuring consistent column names**: When ingesting data from multiple datasets, you may find that the same data in different datasets has different column names. For example, one dataset may have a column called `date_of_birth`, while another dataset has a column called `birthdate`. In this case, a cleansing task may be to rename the `date_of_birth` column heading to `birthdate`.
2. **Changing column data types**: It is important to ensure that a column has a consistent data type for analytics. For example, a certain column may be intended to contain integers, but due to a data entry error, one record in the column may contain a string. When running data analytics on this dataset, having a string in the column may cause the query to fail. In this case, your data cleansing task needs to replace all string values in a column that should contain integers with a null value, which will enable the query to complete successfully.
3. **Ensuring a standard column format**: Different data sources may contain data in a different format. A common example of this is for dates, where one system may format the date as `MM-DD-YYYY`, while another system contains the data as `DD-MM-YYYY`. In this case, the data cleansing task will convert all columns in `MM-DD-YYYY` into the format `DD-MM-YYYY`, or whatever your corporate standard is for analytics.
4. **Removing duplicate records**: With some data sources, you may receive duplicate records (such as when ingesting streaming data, where only-once delivery is not always guaranteed). A data cleansing task may be required to identify and either remove or flag duplicate records.

5. **Providing missing values**: Some data sources may contain missing values in some records, and there are a number of strategies to clean this data. The transformation may replace missing values with a valid value, which could be the average, or median, or the values for that column, or potentially just an empty string or a null. Alternatively, the task may remove any rows that have missing values for a specific column. How to handle missing values depends on the specific dataset and the ultimate analytics use case.

There are many other common tasks that may be performed as part of data cleansing. Within AWS, the *Glue DataBrew* service has been designed to provide an easy way to cleanse and normalize data using a visual design tool and includes over 250 common data cleansing transformations.

Once we have our raw datasets optimized for analytics, we can move on to looking at transforming our datasets to meet business objectives.

Common business use case transformations

In a data lake environment, you generally ingest data from many different source systems into a landing, or raw, zone. You then optimize the file format and partition the dataset, as well as applying cleansing rules to the data, potentially now storing the data in a different zone, often referred to as the clean zone. At this point, you may also apply updates to the dataset with CDC-type data and create the latest view of the data, which we examine in the next section.

The initial transforms we covered in the previous section could be completed without needing to understand too much about how the data is going to ultimately be used by the business. At that point, we were still working on individual datasets that will be used by downstream transformation pipelines to ultimately prepare the data for business analytics.

But at some point, you, or another data engineer working for a line of business, are going to need to use a variety of these ingested data sources to deliver value to the business for a specific use case. After all, the whole point of the data lake is to bring varied data sources from across the business into a central location, to enable new insights to be drawn from across these datasets.

The transformations that we discuss in this section work across multiple datasets to enrich, denormalize, and aggregate the data based on the specific business use case requirements.

Data denormalization

Source data systems, especially those from relational database systems, are mostly going to be highly normalized. This means that the source tables have been designed to contain information about a specific individual entity or topic. Each table will then link to other topics with related information through the use of foreign keys.

For example, you would have one table for customers and a separate table for salespeople. A record for a customer will include an identifier for the salesperson that works with that customer (such as `sales_person_id`). If you want to get the name of the salesperson that supports a specific customer, you could run a SQL query that joins the two tables. During the join, the system queries the customer table for the specific customer record and determines the `sales_person_id` value that is part of the record for that customer. The system then queries the `sales_person` table, finding the record with that `sales_person_id`, and can then access the name of the salesperson from there.

Our normalized customer table may look as follows:

Customer_ID	Last_Name	First_Name	Address_Street	Address_City	Address_State	Phone_Number	Sales_Person_ID
1	Smith	Jonathan	123 Main Street	Springville	MA	555-943-1987	2
2	Mendez	Bruno	5449 South West Street	Jersey	PA	555-615-1609	3
3	Sachdeva	Viyoma	94 Midland Avenue	Oxford	NJ	555-664-0464	1

Figure 7.1: Normalized customer table

And our normalized `sales_person` table may look as follows:

Sales_Person_ID	Last_Name	First_Name	Territory_Code
1	Taylor	Chris	95
2	Williams	Carmen	42
3	Kelly	Michael	23

Figure 7.2: Normalized sales_person table

Structuring tables this way has write-performance advantages for **Online Transaction Processing** (**OLTP**) systems and also helps to ensure the referential integrity of the database. Normalized tables also consume less disk space, since data is not repeated across multiple tables. This was a bigger benefit in the early days of databases when storage was limited and expensive, but it is not a significant benefit today with low-cost object storage systems such as Amazon S3.

When it comes to running **Online Analytical Processing** (**OLAP**) queries, having to join data across multiple tables does incur a performance hit. Therefore, data is often denormalized for analytics purposes.

If we had a use case that required us to regularly query customers with their salesperson details, we may want to create a new table that is a denormalized version of our `customer` and `sales_person` tables.

The denormalized customer table may look as follows:

Customer_ID	Last_Name	First_Name	Address_Street	Address_City	Address_State	Phone_Number	Sales_Person_Last	Sales_Person_First
1	Smith	Jonathan	123 Main Street	Springville	MA	555-943-1987	Williams	Carmen
2	Mendez	Bruno	5449 South West Street	Jersey	PA	555-615-1609	Kelly	Michael
3	Sachdeva	Viyoma	94 Midland Avenue	Oxford	NJ	555-664-0464	Taylor	Chris

Figure 7.3: Denormalized customer_sales_person table

With this table, we can now make a single query that does not require any joins in order to determine the details for a salesperson for a specific customer.

While this was a simple example of a **denormalization** use case, an analytics project may have tens, or even hundreds, of similar denormalization transforms. A denormalization transform may also join data from multiple source tables and may end up creating very wide tables.

It is important to spend time understanding the use case requirements and how the data will be used, and then determine the right table structure and required joins.

Performing these kinds of denormalization transforms can be done with Apache Spark, GUI-based tools, or SQL. AWS Glue Studio can also be used to design these kinds of table joins using a visual interface.

Enriching data

Similar to the way we joined two tables in the previous example for denormalization purposes, another common transformation is to **join tables** for the purpose of enriching the original dataset.

Data that is owned by an organization is already valuable but can often be made even more valuable by combining data the organization owns with data from third parties, or with data from other parts of the business. For example, a company that wants to market credit cards to consumers may purchase a database of consumer credit scores to match against their customer database, or a company that knows that its sales are impacted by weather conditions may purchase historical and future weather forecast data to help them analyze and forecast sales information.

AWS provides a data marketplace with the **AWS Data Exchange** service, a catalog of datasets available via paid subscription, as well as a number of free datasets. AWS Data Exchange currently contains over 1,000

datasets that can be easily subscribed to. Once you subscribe to a dataset, the Data Exchange API can be used to load data directly into your Amazon S3 landing zone.

In these scenarios, you would ingest the third-party dataset to the landing zone of your data lake, and then run a transformation to join the third-party dataset with company-owned data.

Pre-aggregating data

One of the benefits of data lakes is that they provide a low-cost environment for storing large datasets, without needing to pre-process the data or determine the data schema up front. You can ingest data from a wide variety of data sources and store the detailed granular raw data for a long period inexpensively. Then, over time, as you find you have new questions you want to ask of the data, you have all the raw data available to work with and can run ad-hoc queries against the data.

However, as the business develops specific questions they want to regularly ask of the data, the answers to these questions may not be easy to obtain through ad-hoc SQL queries. As a result, you may create transform jobs that run on a scheduled basis to perform the heavy computation that may be required to gain the required information from the data, making it easier for business users to gain the insights they need.

For example, you may create a transform job that creates a denormalized version of your sales data that includes, among others, columns for the store number, city, and state for each transaction. You may then have a **pre-aggregation transform** that runs daily to read this denormalized sales data (which may contain tens of millions of rows per day and tens or hundreds of columns) and compute sales, by category, at the store, city, and state level, and write these out to new tables. You may have hundreds of store managers who need access to store-level data at the category level via a BI visualization tool, but because we have pre-aggregated the data into new tables, the computation does not need to be run every time a report is run.

Pre-aggregating data reduces time to insights for business users and provides significant performance improvements, and therefore you should look to understand the frequent queries that are run by your data consumers, to determine where pre-aggregating data could bring business benefits.

Extracting metadata from unstructured data

As we have discussed previously, a data lake may also contain **unstructured data**, such as audio or image files. While these files cannot be queried directly with traditional analytical tools, we can create a pipeline that uses **Machine Learning** (**ML**) and **Artificial Intelligence** (**AI**) services to extract metadata from these unstructured files.

For example, a company that employs real-estate agents (realtors) may capture images of all houses for sale. One of their data engineers could create a pipeline that uses an AI service such as **Amazon Rekognition** to automatically identify objects in the image and to identify the type of room (kitchen, bedroom, and so on). This captured metadata could then be used in traditional analytics reporting.

Another example is a company that stores audio recordings of customer service phone calls. A pipeline could be built that uses an AI tool such as **Amazon Transcribe** to create transcripts of the calls, and then a tool such as **Amazon Comprehend** could perform sentiment analysis on the transcript. This would create an output that indicates whether the customer sentiment was positive, negative, or neutral for each call. This data could be joined with other data sources to develop a target list of customers to send specific marketing communication.

While unstructured data such as audio and image files may at first appear to have no benefit in an analytics environment, with modern AI tools, valuable metadata can be extracted from many of these sources. This metadata in turn becomes a valuable dataset that can be combined with other organizational data, in order to gather new insights through innovative analytics projects.

While we have only highlighted a few common transforms, there are literally hundreds of different transforms that may be used in an analytics project. Each business is unique and has unique requirements, and it is up to an organization's data teams to understand which data sources are available, and how these can be cleaned, optimized, combined, enriched, and otherwise transformed to help answer complex business questions.

Another aspect of data transformation is the process of applying updates to an existing dataset in a data lake, and we examine strategies for doing this in the next section.

Working with Change Data Capture (CDC) data

One of the most challenging aspects of working within a data lake environment is the processing of updates to existing data, such as with **Change Data Capture** (**CDC**) data. We have discussed CDC data previously, but as a reminder, this is data that contains updates to an existing dataset.

A good example of this is data that comes from a relational database system. After the initial loading of data to the data lake is complete, a system (such as **Amazon DMS**) can read the database transaction logs and write all future database updates to Amazon S3. For each row written to Amazon S3, the first column of the CDC file would contain one of the following characters (see the section on Amazon DMS in *Chapter 3*, *The AWS Data Engineer's Toolkit*, for an example of a CDC file generated by Amazon DMS):

1. **I – Insert**: This indicates that this row contains data that was newly inserted into the table
2. **U – Update**: This indicates that this row contains data that updates an existing record in the table
3. **D – Delete**: This indicates that this row contains data for a record that was deleted from the table

Traditionally, though, it has not been possible to execute updates or deletes of individual records within a data lake. Remember that Amazon S3 is an object storage service, so you can delete and replace a file but you cannot edit or just replace a portion of a file.

If you just append the new records to the existing data, you will end up with multiple copies of the same record, with each record reflecting the state of that record at a specific point in time. This can be useful to keep the history of how a record has changed over time, and so sometimes a transform job will be created to append the newly received data to the relevant table in the data lake for this purpose (potentially adding in a timestamp column that reflects the CDC data-ingestion time for each row). At the same time, we want our end users to be able to work with a dataset that only contains the current state of each data record.

There are two common approaches to handling updates to data in a data lake, as we explore next.

Traditional approaches – data upserts and SQL views

One of the traditional approaches to dealing with CDC data is to run a transform job, on a schedule, that effectively merges the new CDC data with the existing dataset, keeping only the latest records. This is commonly referred to as performing an **upsert** (a combination of update and insert).

One way to do this is to create a transform in Spark that reads existing data into one DataFrame, reads the new data into a different DataFrame, and then merges the DataFrames using custom logic, based on the specific dataset. The transform can then overwrite the existing data or write data to a new date-based partition, creating a new snapshot of the source system. A certain number of snapshots can be kept, enabling data consumers to query data from different points in time.

These transforms can end up being complex, and it is challenging to create a transform that is generic across all source datasets. Also, when overwriting the existing dataset with the updated dataset, there can be disruptions for data consumers who are trying to read from the dataset while the update is running. And as the dataset grows, the length of time and compute resources required to read in the full dataset in order to update it can become a major challenge. There are various strategies for dealing with these challenges, but they are complex, and for a long time, each organization facing these challenges had to implement its own complex solutions.

In order to create a solution that could be used across multiple different datasets, one common approach is to create a configuration table that captures details about source tables. This config table contains information such as a column that should be considered the primary key and a list of columns on which to partition the output. When the transform job runs, it reads the configuration table in order to integrate that table's specific settings with the logic in the transform job.

AWS has a blog post that provides a solution for using **AWS DMS** to capture CDC data from source databases and then run an **AWS Glue** job to apply the latest updates to the existing dataset. This blog post also creates a **DynamoDB** table to store configuration data on the source tables, and the solution can be deployed into an existing account using the provided **AWS CloudFormation** template. For more information, see the AWS blog post titled *Load ongoing data lake changes with AWS DMS and AWS Glue*.

An alternative approach is to use **Athena views** to create a virtualized table that shows the latest state of the data. An Athena view is a query that runs whenever the virtual table is queried, using a `SELECT` query that is defined in the view. The view definition will join the source (the current table) and the table with the new **CDC data**, and return a result that reflects the latest state of the data.

Creating a view that combines the existing data and the new CDC data enables consumers to query the latest state of the data, without needing to wait for a daily transform job to run to consolidate the datasets. However, performance will degrade over time as the CDC data table grows, so it is advisable to also have a daily job that will run to consolidate the new CDC data into the existing dataset. Creating and maintaining these views can be fairly complex, especially when combined with a need to also have a daily transform to consolidate the datasets.

For many years, organizations have faced the challenge of building and maintaining custom solutions like these to deal with CDC data and other data lake updates. However, in recent years, a number of new offerings have been created to address these requirements more generically, as we see in the next section.

Modern approaches – Open Table Formats (OTFs)

Over the past few years, a number of new **Open Table Formats (OTFs)** have been developed to support the idea of a more *transactional data lake*. When we refer to a transactional data lake, we are referencing the ability of a data lake to contain properties that were previously only available in a traditional database, such as the ability to update and delete individual records. In addition, many of these new solutions also provide support for **schema evolution** and **time travel** (the ability to query data as it was at a previous point in time).

Technically, these new OTFs bring **ACID** semantics to the data lake:

1. **Atomicity**: An expectation that data written will either be written as a full transaction or will not be written at all, and the dataset will be returned to its state prior to the transaction on failure
2. **Consistency**: The expectation that even if a failure occurs, the dataset will stay consistent
3. **Isolation**: The expectation that one transaction on the dataset will not be affected by another transaction that is requested at the same time

4. **Durability**: The expectation that once a successful transaction has been completed, this transaction will be durable (it will be permanent, even if there is a later system failure)

Now, this does not mean that these modern data lake solutions can replace existing OLTP-based databases. You are not going to suddenly see retailers dump their PostgreSQL, MySQL, or SQL Server databases that run their **Customer Relationship Management** (**CRM**) systems and instead use a data lake for everything.

Rather, data lakes are still intended as an analytical platform, but these new solutions significantly simplify the ability to apply changes to existing records, as well as the ability to delete records from a large dataset. These solutions also help to ensure data consistency as multiple teams work on the same datasets. There is still latency involved with these types of transactions, but much of the complexity involved with consolidating new and updated rows in a dataset, and providing a consistent, up-to-date view of data with lower latency, is handled by these solutions.

In *Chapter 14*, *Building Transactional Data Lakes*, we will do a deeper dive into OTFs, but let's have a quick look at some of the most common offerings for these new transactional data lakes.

Apache Iceberg

Apache Iceberg was created by engineers at **Netflix** and is designed as an OTF for very large datasets. The code was donated to the Apache Software Foundation and became a top-level project in May 2020. Since then, it has become a very popular choice for creating a transactional data lake.

Iceberg supports schema evolution, time travel for querying at a point in time, atomic table changes (to ensure that data consumers do not see partial or uncommitted changes), and support for multiple simultaneous writers.

In August 2021, a new start-up, **Tabular**, was formed by the creators of Iceberg to build a cloud-native data platform powered by Apache Iceberg. Also, most of the large cloud providers (as well as other vendors) have added support for Apache Iceberg into many of their products (for example, AWS Glue provides native support for the Apache Iceberg table format).

Apache Hudi

Apache Hudi started out as a project within **Uber** (the ride-sharing company) to provide a framework for developing low-latency and high-efficiency data pipelines for their large data lake environment. They subsequently donated the project to the Apache Software Foundation, which in turn made it a top-level project in 2020. Today, Apache Hudi is a popular option for building out transactional data lakes that support the ability to efficiently upsert new/changed data into a data lake, as well as to easily query tables and get the latest updates returned. AWS supports running Apache Hudi within the Amazon EMR managed service, as well as with AWS Glue.

Databricks Delta Lake

Databricks, a company formed by the original creators of Apache Spark, have developed their own approach to providing a transactional data lake, which has become popular over the past few years. This solution, called **Delta Lake**, is an open table format for streaming and batch operations that provides ACID transactions for inserts, updates, and deletes. In addition, Delta Lake supports time travel, which enables a query to retrieve data as it was at any point in time. Databricks have open-sourced this solution and made it available on GitHub at `https://github.com/delta-io/delta`.

In addition to the open-source version of Delta Lake, Databricks also offers a fully supported commercial version of Delta Lake that is popular with large enterprises. For more information on the commercial version of Delta Lake, see `https://databricks.com/product/delta-lake-on-databricks`.

Handling updates to existing data in a data lake has been a challenge for as long as data lakes have been in existence. Over the years, some common approaches have emerged to handle these challenges, but each organization has had to effectively *reinvent the wheel* to implement their own solution.

Now, with a number of companies recently creating solutions to provide a more transactional-type data lake that simplifies the process of inserting, updating, and deleting data, it makes sense to explore these open table formats, as outlined in this section, and covered more fully in *Chapter 14*, *Building Transactional Data Lakes*.

So far in this chapter, we have covered data preparation transformations, business use case transforms, and how to handle CDC-type updates for a data lake. Now we get hands-on with data transformation using AWS Glue Studio.

Hands-on – joining datasets with AWS Glue Studio

For our hands-on exercise in this chapter, we are going to use *AWS Glue Studio* to create an Apache Spark job that joins streaming data with data we migrated from our MySQL database in the previous chapter.

Creating a new data lake zone – the curated zone

As discussed in *Chapter 2*, *Data Management Architecture for Analytics*, it is common to have multiple zones in a data lake, containing different copies of our data as it gets transformed. So far, we have ingested raw data into the landing zone and then converted some of those datasets into Parquet format, and written the files out in the clean zone. In this chapter, we will be joining multiple datasets together and will write out the new dataset to the curated zone of our data lake. The curated zone is intended to store data that has been transformed and is ready for consumption by data consumers. We created an Amazon S3 bucket for the

curated zone in a previous chapter, so now we can create a new AWS Glue database for this zone of our data lake:

1. Log in to the **AWS Management Console** (https://console.aws.amazon.com).
2. In the **top search bar**, search for and select **Glue** to access the Glue console.
3. On the left-hand side, select **Databases**, and then click **Add database**.
4. For **Database name**, type `curatedzonedb`, and then click **Create database**.

We have now created a new curated zone database for our data lake, and in the next step, we will create a new IAM role to provide the permissions needed for our Glue transformation job to run.

Creating a new IAM role for the Glue job

When we configure the Glue job using Glue Studio, we will need to specify an IAM role that has the following permissions:

1. Read our source S3 bucket (for example, `dataeng-landing-zone-gse23 and dataeng-clean-zone-gse23`)
2. Write to our target S3 bucket (for example, `dataeng-curated-zone-gse23`)
3. Access to Glue temporary directories
4. Write logs to **Amazon CloudWatch**
5. Access to all Glue API actions (to enable the creation of new databases and tables)

To create a new AWS IAM role with these permissions, follow these steps:

1. In the top search bar of the **AWS Management Console**, search for and select the **IAM** service, and in the left-hand menu, select **Policies**, and then click on **Create policy**.

2. By default, the **Visual editor** tab is selected, so click on **JSON** to change to the **JSON** tab.
3. Provide the JSON code from the following code blocks, replacing the boilerplate code. This policy provides the required S3 permissions, and we will provide Glue and CloudWatch permissions via a managed policy in a later step.
   Note that you can also copy and paste this policy by accessing the policy on this book's GitHub page. If doing a copy and paste from the GitHub copy of this policy, you must replace `<initials>` in bucket names with the unique identifier you used when creating the buckets.
   The first block of the policy configures the policy document and provides permissions to get objects from Amazon S3 that are in the Amazon S3 buckets specified in the resource section. Make sure you replace `<initials>` with the unique identifier you have used in your bucket names:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::dataeng-landing-zone-<initials>/*",
                "arn:aws:s3:::dataeng-clean-zone-<initials>/*"
                ]
        },
```

4. This next block of the policy provides permissions for all Amazon S3 actions (`get` , `put` , and so on) that are in the Amazon S3 bucket specified in the resource section (in this case, our curated zone bucket). Make sure you replace `<initials>` with the unique identifier you have used in your bucket names:

```
        {
            "Effect": "Allow",
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::dataeng-curated-zone-<initials>/*"
        }
    ]
}
```

5. Click on **Next: tags** and then click on **Next: Review**.
6. Provide a name for the policy, such as `DataEngGlueCWS3CuratedZoneWrite`, and then click **Create policy**.
7. In the left-hand menu, click on **Roles** and then **Create role**.
8. For **Trusted entity**, ensure **AWS service** is selected, and for **Use case** search for and select **Glue**, and then click **Next**. Listing Glue as a trusted entity for this role enables the AWS Glue service to assume this role to run transformations.
9. Under the **Attach permissions** policies, select the policy we just created (for example, `DataEngGlueCWS3CuratedZoneWrite`) by searching and then clicking in the checkbox.
10. Also, search for `AWSGlueServiceRole` and click on the checkbox to select this role (make sure that it is the managed `AWSGlueServiceRole` policy, and not a policy previously created, such as `AWSGlueServiceRole-streaming-crawler-Fefdx-s3Policy`). This managed policy provides access to temporary directories used by Glue, as well as **CloudWatch** logs and Glue resources.
11. Then, click **Next**.
12. Provide a role name, such as `DataEngGlueCWS3CuratedZoneRole`, and click **Create role**.

We have now configured the permissions required for our Glue job to be able to access the required resources, so we can now move on to building our transformation using Glue Studio.

Configuring a denormalization transform using AWS Glue Studio

We are now ready to create an **Apache Spark** job to denormalize the film data that we migrated from our MySQL database. The dataset we migrated is normalized currently (as expected for data coming from a relational database), so we want to denormalize some of the data to use in future transforms.

Ultimately, we want to be able to analyze various data points of our new streaming library of classic movies. One of the data points we want to understand is which categories of movies are the most popular, but to find the name of a category associated with a specific movie, we need to query three different tables in our source dataset. The tables are as follows:

1. `film` : This table contains details of each film in our classic movie library, including `film_id` , `title` , `description` , `release_year` , and `rating` . However, this table does not contain any information about the category that the film is in.
2. `category` : This table contains the name of each category of film (such as action, comedy, drama, and so on), as well as `category_id` . However, this table does not contain any information that links a category with a film.
3. `film_category` : This table is designed to provide a link between a specific film and a specific category. Each row contains a `film_id` value and associated `category_id` .

When analyzing the incoming streaming data about viewers streaming our movies, we don't want to have to do joins on each of the above tables to determine the category of movie that was streamed. So, in this first transform job that we are going to create, we denormalize this group of tables so that we end up with a single table that includes the category for each film in our film library.

To build the denormalization job using AWS Glue Studio, follow these steps:

1. In the AWS Management Console, use the top search bar to search for and select the **Glue** service.
2. In the left-hand menu, click on **ETL Jobs**.
3. Click on the **Visual ETL** option
4. On the left-hand side, under the **Visual** tab, click on **Amazon S3 (source)** as a new node.
5. On the right-hand side, under **Data source properties – S3**, ensure **Data Catalog table** is selected for **S3 source type**, and from the dropdown, select the `sakila` database.
6. For the **Table** dropdown, select `film_category`.
7. Set **Name** for this node to be `S3 - Film-Category`.

At this point, the Glue Studio screen should look as follows:

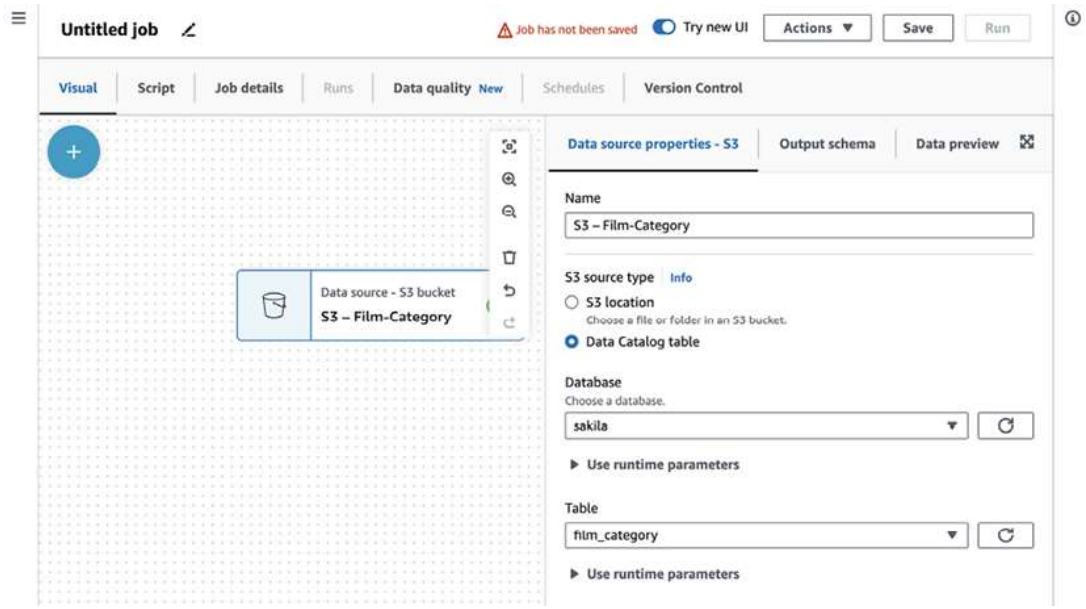

Figure 7.4: Glue Studio with first S3 data source

8. Click the **plus sign (+)** in the top left of the visual editor, and repeat *steps 4-7*, adding another S3 source for the `film` table, but set **Name** to `S3 - Film`. Once done, your Glue studio screen should look as follows:

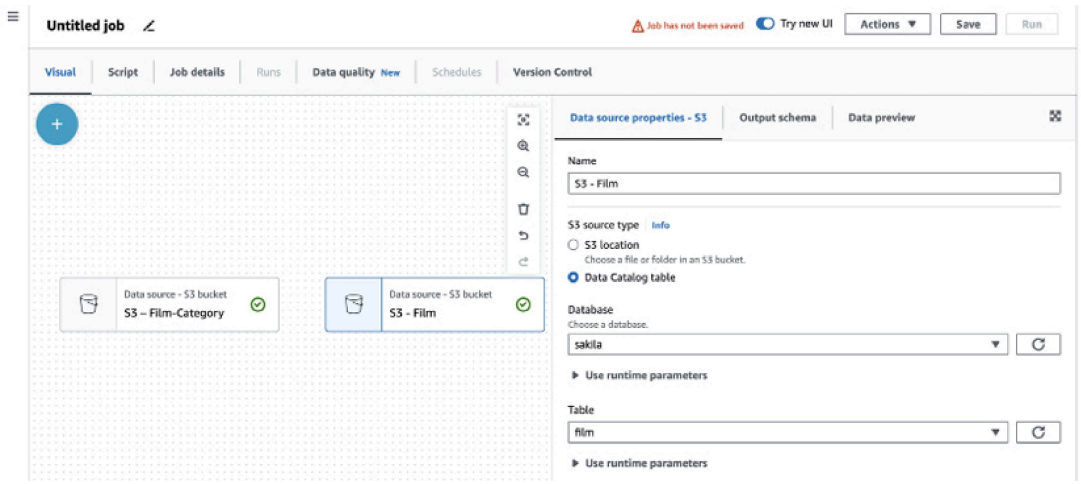

Figure 7.5: Glue Studio with two S3 data sources

9. In the **Designer** window, click on the **plus sign (+)** again, and from the **Transform** tab, select the **Join** transform.
10. The **Join** transform requires two "parent" nodes – the two tables that we want to join. To set the parent nodes, use the **Node parents** drop-down to select the **S3 – Film** and **S3 – Film-Category** tables.
11. You will see a red checkmark on the **Transform** tab, indicating an issue that needs to be resolved. Click on the **Transform** tab, and you will see a warning about both tables having a column with the same name. Glue Studio offers to automatically resolve the issue by adding a custom prefix to the columns in the right-hand table. Click on **Resolve it** to have Glue Studio automatically add a new transform that renames the columns in the right-hand table.
12. There are a number of different join types that Glue Studio supports. Review the **Join type** drop-down list to understand the differences. For our use case, we want to join all the rows from our left-hand table (`film`) with matching rows from the right-hand table (`film_category`). The resulting table will have rows for every film, and each row will also include information from the `film_category` table – in this case, the `category_id` value for each film. For **Join type**, select **Left join**, and then click **Add condition**. We want to match the `film_id` field from the film table with the `film_id` field from the `film_category` table. Remember, though, that we had Glue

Studio automatically rename the fields in the `film_category` table, so for the `film_category` table, select the **(right)** `film_id` field. Once done, your Glue Studio screen should look as follows:

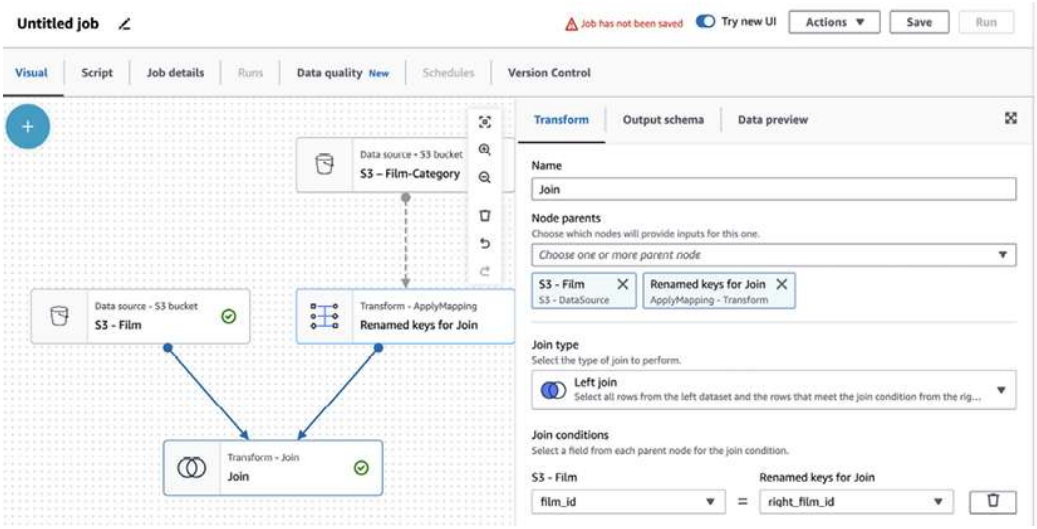

Figure 7.6: Glue Studio after first table join

13. Let's provide a name for the temporary table created as a result of the join. On the **Node properties** tab, change **Name** to `Join – Film-Category_ID` .
14. We don't need all the data that is in our temporary `Join – Film-Category_ID` table, so we can now use the **Glue Change Schema** transform to drop the columns we don't need, rename fields, and so on. Click on the **plus sign (+)** , and from the **Transforms** tab, select **Change Schema**.
15. Some of the fields that are related to our original data from when these movies were rented out from our DVD stores are not relevant to our new streaming business, so we can drop those now. At the same time, we can drop some of the fields from our `film_category` table, as the only column we need from that table is `category_id` . Select the **Drop** checkbox for the following columns:
    1. `rental_duration`
    2. `rental_rate`
    3. `replacement_cost`
    4. `last_update`

    5. `(right) film_id`
    6. `(right) last_update`
16. We can now add a transform, which will join the results of the **Change Schema** transform with our category table, adding the name of the category for each film. To add the **Category** table to our transform, click the **plus sign (+),** and from the **Data** tab, select **Amazon S3**. Ensure **Data Catalog Table** is selected, and for **Database**, select **sakila**; and for **Table**, select **Category**. To provide a descriptive name, change **Name** to `S3 – Category` .
17. We can now add our final transformation. Click the **plus sign (+)**, and from the **Transforms** tab, select **Join**.
18. We always need two tables for a join, so from the **Node properties** tab, use the **Node parents** dropdown to add the **Change Schema** transform as a parent of the join, and change **Name** to `Join – Film-Category` .
19. On the **Transform** tab, select **Left join** for **Join type**, and then click **Add condition**. From the **S3 – Category** table, select the `category_id` field, and from the **Change Schema** table, select the `(right) category_id` field.
20. Now we will add one last **Change Schema** transform to again remove unneeded fields, and rename fields where appropriate. Click the **plus sign (+)**, and from the **Transforms** tab, select **Change Schema**. Click the checkbox next to the following columns in order to drop them:
    1. `last_update`
    2. `(right) category_id`
21. Then, for the **Source key** value of **name**, change **Target key** to be `category_name` , as this is a more descriptive name for this field.

In this section, we configured our Glue job for the transform steps required to denormalize our film and category data. In the next section, we will complete the configuration of our Glue job by specifying where we want our new denormalized table to be written.

Finalizing the denormalization transform job to write to S3

To finalize the configuration of our transform job using Glue Studio, we now need to specify the target that we want to write our data to:

1. Add a target by clicking on the **plus sign (+)**, and from the **Targets** tab, select Amazon S3.
2. On the **Data target properties – S3** tab, ensure **Parquet** is selected for **Format** and **Snappy** for **Compression type**. Click on **Browse S3** for **S3 Target Location**, select the checkbox for the `dataeng-curated-zone-<initials>` bucket, and click **Choose**. In the **S3 Target Location** field, add a prefix after the bucket name of `/filmdb/film_category/`.
3. For **Data Catalog update options**, select **Create a table in the Data Catalog, and on subsequent runs, update the schema and add new partitions**.
4. For **Database**, select `curatedzonedb` from the drop-down list.
5. For **Table name**, type in `film_category`. Note that Spark requires lowercase table and column names, and that the only special character supported by Athena is the underscore character, which is why we use this rather than a hyphen.
   Our **Data target properties – S3** configuration should look as follows:

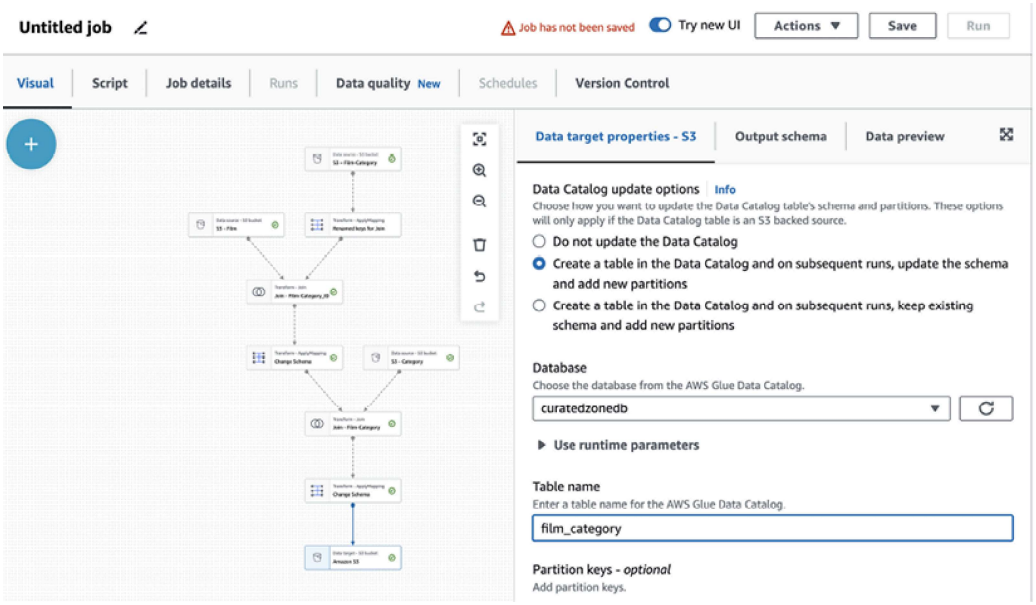

Figure 7.7: Data target properties – S3 configuration

A note about partition keys

Our sample dataset is very small (just 1,000 film records), but imagine for a moment that we were trying to create a similar table, including category information, for all the books ever published. According to an estimate from Google in 2010, there were nearly 130 million books that they planned to scan into a digital format. If our intention was to query all this book data to gather information on the books by category, then we would add a partition key, and specify `category_name` as a partition. When the data was written to S3, it would be grouped into different prefixes based on the category name, and this would significantly increase performance when we queried books by category.

---

6. We can now provide a name and permissions configuration for our job. In the top left, change from the **Visual** tab to the **Job details** tab.
7. Set the name of the job to be `Film Category Denormalization`.
8. For **IAM Role**, from the dropdown, select the role we created previously (`DataEngGlueCWS3CuratedZoneRole`).

9. For **Requested number of workers**, change this to `2`. This configuration item specifies the number of nodes configured to run our Glue job, and since our dataset is small, we can use the minimum number of nodes. Since we are using the minimum number of nodes, auto-scaling will not make a difference, but it is **strongly encouraged** to use the option for **Automatically scale the number of workers** when configuring your jobs. This feature ensures that Glue dynamically changes the number of workers to match what is needed by your job, saving you money by minimizing idle workers.
10. For **Job bookmark**, ensure this setting is set to **Disable**. A job bookmark is a feature of Glue that tracks which files have been previously processed so that a subsequent run of the job does not process the same files again. For our testing purposes, we may want to process our test data multiple times, so we disable the bookmark.
11. For **Number of retries**, ensure this is set to **0**. If our job fails to run, we don't want it to automatically repeat.
12. Leave all other defaults, and at the top right, click on **Save**. Then, click on **Run** to run the transform job.
13. Click on the **Runs** tab in order to monitor the job run. You can also change to the **Script** tab if you want to view the Spark code that AWS Glue Studio generated.
14. When the job completes, navigate to the **Amazon S3** console and review the output location (such as `dataeng-curated-zone-gse23/filmdb/film_category`) to validate that the new Parquet files were created. Also, navigate to the **AWS Glue** console to confirm that the new table (`film_category`) was created in `curatedzonedb`.

In the preceding steps, we denormalized data related to our catalog of films and their categories, and we can now join data from this new table with our streaming data.

Create a transform job to join streaming and film data using AWS Glue Studio

In this section, we're going to use **AWS Glue Studio** to create another transform, this time to join the table containing all streams of our movies, with the denormalized data about our film catalog:

1. In the **AWS Management Console**, use the top search bar to search for and select the **Glue** service.
2. In the left-hand menu, click on **ETL jobs**.
3. Click on the **Visual ETL** option for **Create job**.
4. In the **Visual** tab, under **Add nodes**, select **Amazon S3 (source)**.
5. On the right-hand side, under the **Data source properties – S3 tab**, under **S3 source type**, ensure **Data Catalog table** is selected, and from the dropdown, select the `curatedzonedb` database.
6. For the **Table** dropdown, select `film_category`.
7. Set **Name** to `S3 – Film_Category`.
8. On the left-hand side, under the **Visual** tab, click on the PLUS (+) sign. Repeat *steps 4–7*, adding another S3 source for the **streaming** table from the `streamingdb` database, and setting the name to `S3 - Streaming`.
9. Click on the **plus sign (+)**, and from the **Transforms** tab, add a **Change Schema** transform for the **S3 – Streaming** data source (confirm that the **S3 – Streaming** source is selected by looking at **Node parents** on the **Node properties** tab).
10. Under the **Node properties** tab, change the **Name** of this transform element to `Change Schema – Streaming`.
11. On the **Transform** tab, under **Change Schema**, change the name of the `film_id` key to `film_id_streaming` (by changing the **Target key** value for `film_id`). Both of our S3 source tables have a `film_id` field, which is why we need to change the field name for one of the tables.
12. Click the **plus sign (+)** again, and under the **Transform** tab, add a **Join** transform and set **Join type** to **Left join**.
13. Under the **Transform** tab, ensure that the **Change Schema – Streaming** element and the `S3 – Film_Category` are listed as **Node parents.**

14. Under the **Transform** tab, for **Join conditions**, click on **Add condition**. Select `film_id_streaming` for the **Change Schema – Streaming** element, and `film_id` for the `S3 – Film_Category` element.
    Your Glue Studio visual designer should look as follows:

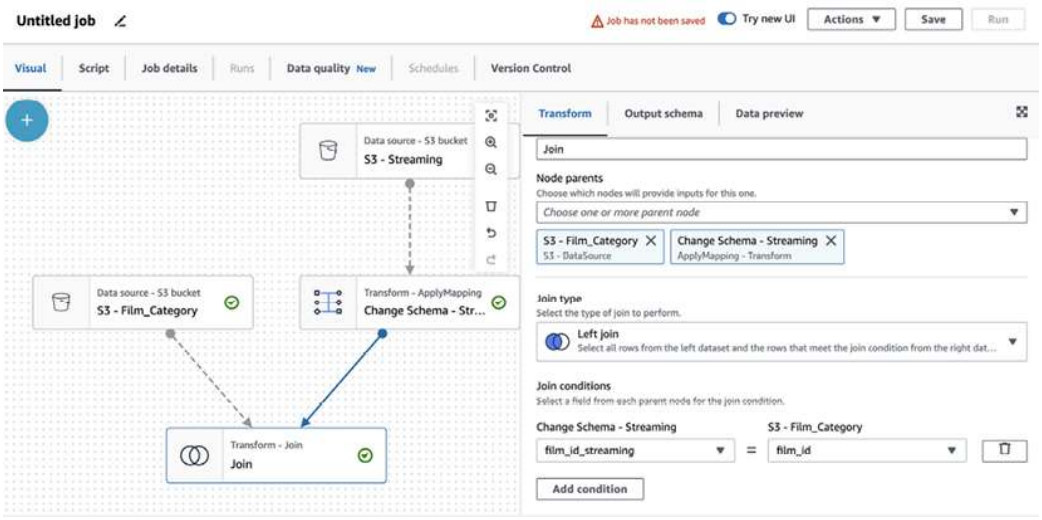

Figure 7.8: Glue Studio interface showing the first join

15. Add a target by clicking on the **plus sign (+)**, and under the **Targets** tab, click on Amazon S3.
16. For **Format**, select **Parquet** from the dropdown, and for **Compression type**, select **Snappy**.
17. For **S3 Target Location**, click **Browse S3**, click the selector for the `dataeng-curatedzone-<initials>` bucket, and click **Choose**. Add a prefix after the bucket of `/streaming/streaming-films/` .
18. For **Data Catalog update options**, select **Create a table in the Data Catalog, and on subsequent runs, update the schema and add new partitions**.
19. For **Database**, select `curatedzonedb` from the drop-down list.
20. For **Table name**, type in `streaming_films` .
    Our **Data Target Properties – S3** configuration should look as follows:

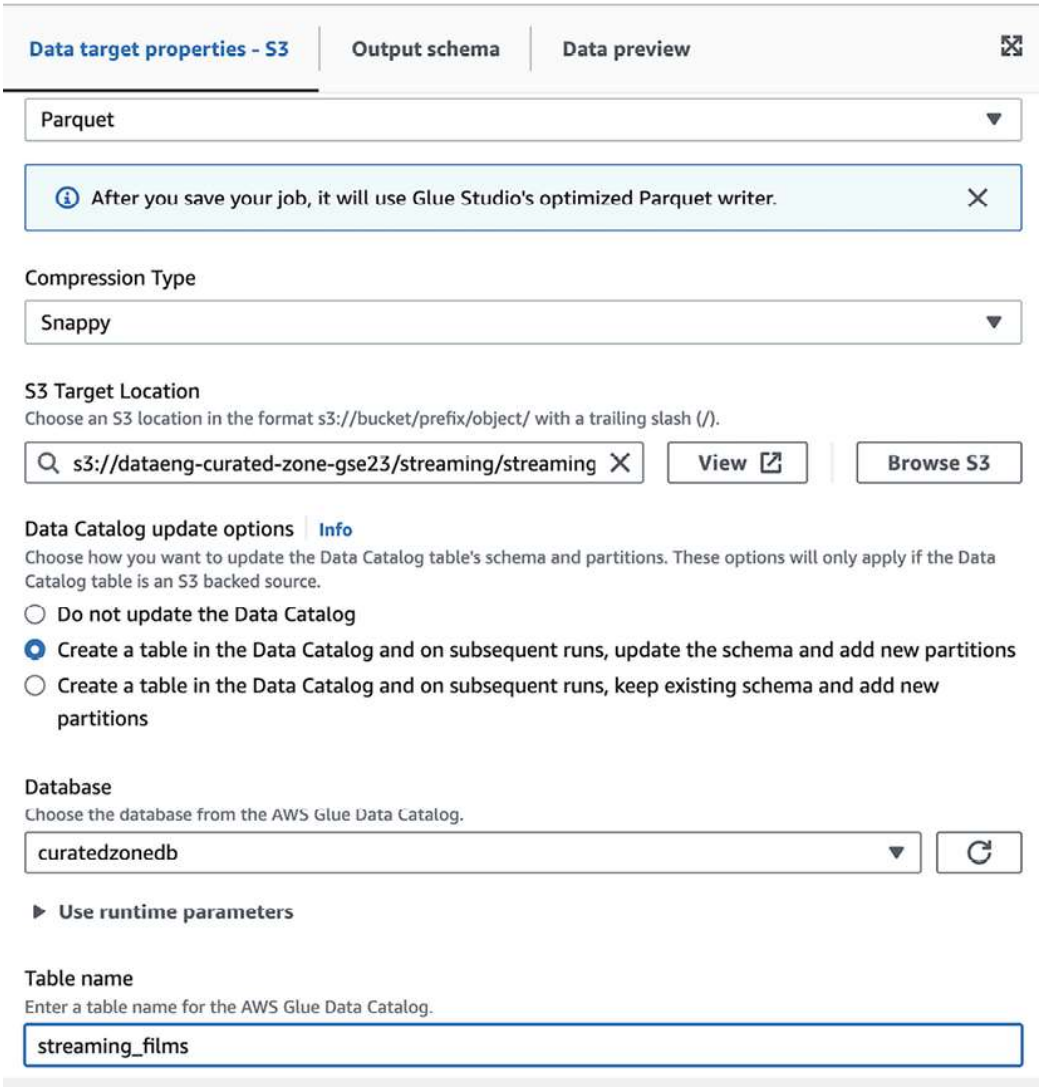

Figure 7.9: Glue Studio interface showing target configuration

21. We can now provide a name and permissions configuration for our job. Along the top list of tabs, change from the **Visual** tab to the **Job details** tab.
22. Set the name of the job to be `Streaming Data Film Enrichment`.
23. For **IAM Role**, from the dropdown, select the role we created previously (`DataEngGlueCWS3CuratedZoneRole`).
24. For **Number of workers**, change this to **2**.
25. For **Job bookmark**, make sure this is set to **Disable**.
26. For **Number of retries**, make sure this is set to **0**.
27. Leave all other defaults, and at the top right, click on **Save**. Then click on **Run** to run the transform job.
28. Click on the **Runs** tab in order to monitor the job run.

29. When the job status changes to **Succeeded**, click on **Databases** (under **Data Catalog)** to confirm that the new table (`streaming_films`) was created in the `curatedzonedb` database.
30. Navigate to the **Amazon S3** console and review the output location (`dataeng-curatedzone-<initials>/streaming/streaming-films/`) to validate that the files were created.

We have now created a single table that contains a record of all streams of our classic movies, along with details about each movie, including the category of the movie. This table can be efficiently queried to analyze streams of our classic movies to determine the most popular movie and movie category, and we can break this down by state and other dimensions.

Summary

In this chapter, we've reviewed a number of common transformations that can be applied to raw datasets, covering both generic transformations used to optimize data for analytics and business transforms to enrich and denormalize datasets.

This chapter built on previous chapters in this book. We started by looking at how to architect a data pipeline, then reviewed ways to ingest different data types into a data lake, and in this chapter, we reviewed common data transformations.

In the next chapter, we will look at common types of data consumers and learn more about how different data consumers want to access data in different ways, and with different tools.

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