

Data Engineering with **AWS**

Learn how to design and build cloud-based data transformation pipelines using AWS



Data Engineering with AWS

Learn how to design and build cloud-based data transformation pipelines using AWS

Gareth Eagar



BIRMINGHAM—MUMBAI

Data Engineering with AWS

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author(s), nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Publishing Product Manager: Reshma Raman

Senior Editor: Mohammed Yusuf Imaratwale

Content Development Editor: Sean Lobo

Technical Editor: Rahul Limbachiya

Copy Editor: Safis Editing

Project Coordinator: Aparna Ravikumar Nair

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Production Designer: Alishon Mendonca

First published: December 2021

Production reference: 1251121

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN: 978-1-80056-041-3

www.packt.com

Contributors

About the author

Gareth Eagar has worked in the IT industry for over 25 years, starting in South Africa, then working in the United Kingdom, and now based in the United States. In 2017, he started working at **Amazon Web Services (AWS)** as a solution architect, working with enterprise customers in the NYC metro area. Gareth has become a recognized subject matter expert for building data lakes on AWS, and in 2019 he launched the Data Lake Day educational event at the AWS Lofts in NYC and San Francisco. He has also delivered a number of public talks and webinars on topics relating to big data, and in 2020 Gareth transitioned to the AWS Professional Services organization as a senior data architect, helping customers architect and build complex data pipelines.

Additional contributors

Disha Umarwani is a Data and ML Engineer at **Amazon Web Services (AWS)**. She works with AWS health care and life science customers to design, architect and build analytics and ML solutions on the AWS cloud. Disha specializes in services like AWS Glue, Amazon EMR, and AWS Step functions.

Praful Kava is a Senior Specialist Solutions **Architect at Amazon Web Services (AWS)**. He guides customers in designing and engineering cloud-scale analytics pipelines on AWS. Outside work, Praful enjoys travelling with his family and exploring new hiking trails.

Natalie Rabinovich is a Senior Solutions Architect at **Amazon Web Services (AWS)**. She has extensive experience in data center infrastructure, data storage, and big data and analytics. Natalie helps organizations design reliable and cost-effective cloud solutions.

About the reviewers

Praveen Gupta is currently a data engineering manager and has over 17 years of experience in the IT industry. Praveen started his career as an ETL/reporting developer working on traditional RDBMSes and reporting tools. Since 2014, he has been working on the AWS cloud on projects related to data science/machine learning and building complex data engineering pipelines on AWS. He specializes in data ingestion, big data processing, reporting, and building massive data warehouses at the petabyte scale for his customers, helping his customers make data-driven decisions. Praveen has an undergraduate degree in computer science and a master's degree in computer science from UIUC, USA. Praveen lives in Portland, USA with his wife and 8-year-old daughter.

Mradul Saraf is a data engineer at an American multinational conglomerate that focuses on e-commerce, cloud computing, digital streaming, and artificial intelligence. It is one of the Big Five companies in the US. He has over four years of experience in data engineering, big data, and cloud computing. He holds a Bachelor of Technology degree in computer science from Maulana Azad National Institute of Technology, Bhopal. He has experience of architecture, analysis, design, development, implementation, maintenance, and support, along with experience of developing strategic methods for deploying big data technologies to efficiently meet big data processing requirements across multiple domains.

Table of Contents

Preface

Section 1: AWS Data Engineering Concepts and Trends

1

An Introduction to Data Engineering

Technical requirements	4	Understanding other common data-related roles	9
The rise of big data as a corporate asset	4	The benefits of the cloud when building big data analytic solutions	10
The challenges of ever-growing datasets	5	Hands-on – creating and accessing your AWS account	11
Data engineers – the big data enablers	7	Creating a new AWS account	12
Understanding the role of the data engineer	8	Accessing your AWS account	15
Understanding the role of the data scientist	8	Summary	19
Understanding the role of the data analyst	9		

2

Data Management Architectures for Analytics

Technical requirements	22	Dealing with big, unstructured data	24
The evolution of data management for analytics	22	A lake on the cloud and a house on that lake	25
Databases and data warehouses	23		

Understanding data warehouses and data marts – fountains of truth	27	Data lake logical architecture	42
Distributed storage and massively parallel processing	29	Bringing together the best of both worlds with the lake house architecture	45
Columnar data storage and efficient data compression	30	Data lakehouse implementations	46
Dimensional modeling in data warehouses	32	Building a data lakehouse on AWS	47
Understanding the role of data marts	36	Hands-on – configuring the AWS Command Line Interface tool and creating an S3 bucket	48
Feeding data into the warehouse – ETL and ELT pipelines	37	Installing and configuring the AWS CLI	49
Building data lakes to tame the variety and volume of big data	40	Creating a new Amazon S3 bucket	50
		Summary	50

3

The AWS Data Engineer's Toolkit

Technical requirements	52	AWS services for transforming data	64
AWS services for ingesting data	52	Overview of AWS Lambda for light transformations	64
Overview of Amazon Database Migration Service (DMS)	52	Overview of AWS Glue for serverless Spark processing	65
Overview of Amazon Kinesis for streaming data ingestion	54	Overview of Amazon EMR for Hadoop ecosystem processing	69
Overview of Amazon MSK for streaming data ingestion	59	AWS services for orchestrating big data pipelines	71
Overview of Amazon AppFlow for ingesting data from SaaS services	60	Overview of AWS Glue workflows for orchestrating Glue components	71
Overview of Amazon Transfer Family for ingestion using FTP/SFTP protocols	61	Overview of AWS Step Functions for complex workflows	73
Overview of Amazon DataSync for ingesting from on-premises storage	62	Overview of Amazon managed workflows for Apache Airflow	75
Overview of the AWS Snow family of devices for large data transfers	63		

AWS services for consuming data	77	Hands-on – triggering an AWS Lambda function when a new file arrives in an S3 bucket	83
Overview of Amazon Athena for SQL queries in the data lake	77	Creating a Lambda layer containing the AWS Data Wrangler library	83
Overview of Amazon Redshift and Redshift Spectrum for data warehousing and data lakehouse architectures	78	Creating new Amazon S3 buckets	85
Overview of Amazon QuickSight for visualizing data	81	Creating an IAM policy and role for your Lambda function	86
		Creating a Lambda function	88
		Configuring our Lambda function to be triggered by an S3 upload	93
		Summary	96

4

Data Cataloging, Security, and Governance

Technical requirements	98	AWS services for data encryption and security monitoring	110
Getting data security and governance right	98	AWS Key Management Service (KMS)	111
Common data regulatory requirements	99	Amazon Macie	112
Core data protection concepts	100	Amazon GuardDuty	112
Personal data	101	AWS services for managing identity and permissions	113
Encryption	101	AWS Identity and Access Management (IAM) service	113
Anonymized data	102	Using AWS Lake Formation to manage data lake access	116
Pseudonymized data/tokenization	102	Hands-on – configuring Lake Formation permissions	118
Authentication	103	Creating a new user with IAM permissions	119
Authorization	104	Transitioning to managing fine-grained permissions with AWS Lake Formation	123
Putting these concepts together	104	Summary	129
Cataloging your data to avoid the data swamp	105		
How to avoid the data swamp	106		
The AWS Glue/Lake Formation data catalog	108		

Section 2: Architecting and Implementing Data Lakes and Data Lake Houses

5

Architecting Data Engineering Pipelines

Technical requirements	134	Data standardization	143
Approaching the data pipeline architecture	134	Data quality checks	143
Architecting houses and architecting pipelines	135	Data partitioning	143
Whiteboarding as an information-gathering tool	136	Data denormalization	143
Conducting a whiteboarding session	137	Data cataloging	144
Identifying data consumers and understanding their requirements	138	Whiteboarding data transformation	144
Identifying data sources and ingesting data	140	Loading data into data marts	146
Identifying data transformations and optimizations	142	Wrapping up the whiteboarding session	147
File format optimizations	142	Hands-on – architecting a sample pipeline	149
		Detailed notes from the project "Bright Light" whiteboarding meeting of GP Widgets, Inc	150
		Summary	156

6

Ingesting Batch and Streaming Data

Technical requirements	158	Ingesting data from a relational database	165
Understanding data sources	158	AWS Database Migration Service (DMS)	166
Data variety	159	AWS Glue	166
Data volume	163	Other ways to ingest data from a database	167
Data velocity	163	Deciding on the best approach for ingesting from a database	169
Data veracity	164		
Data value	164	Ingesting streaming data	171
Questions to ask	165		

Amazon Kinesis versus Amazon Managed Streaming for Kafka (MSK)	171	Configuring DMS settings and performing a full load from MySQL to S3	181
Hands-on – ingesting data with AWS DMS	174	Querying data with Amazon Athena	184
Creating a new MySQL database instance	175	Hands-on – ingesting streaming data	186
Loading the demo data using an Amazon EC2 instance	177	Configuring Kinesis Data Firehose for streaming delivery to Amazon S3	186
Creating an IAM policy and role for DMS	179	Configuring Amazon Kinesis Data Generator (KDG)	187
		Adding newly ingested data to the Glue Data Catalog	190
		Querying the data with Amazon Athena	191
		Summary	191

7

Transforming Data to Optimize for Analytics

Technical requirements	194	Optimizing with data partitioning	202
Transformations – making raw data more valuable	194	Data cleansing	203
Cooking, baking, and data transformations	195	Business use case transforms	205
Transformations as part of a pipeline	196	Data denormalization	205
Types of data transformation tools	196	Enriching data	207
Apache Spark	196	Pre-aggregating data	207
Hadoop and MapReduce	197	Extracting metadata from unstructured data	208
SQL	198	Working with change data capture (CDC) data	209
GUI-based tools	199	Traditional approaches – data upserts and SQL views	210
Data preparation transformations	200	Modern approaches – the transactional data lake	211
Protecting PII data	200	Hands-on – joining datasets with AWS Glue Studio	214
Optimizing the file format	201		

Creating a new data lake zone – the curated zone	214	Finalizing the denormalization transform job to write to S3	222
Creating a new IAM role for the Glue job	215	Create a transform job to join streaming and film data using AWS Glue Studio	224
Configuring a denormalization transform using AWS Glue Studio	217	Summary	227

8

Identifying and Enabling Data Consumers

Technical requirements	230	Meeting the needs of data scientists and ML models	239
Understanding the impact of data democratization	230	AWS tools used by data scientists to work with data	239
A growing variety of data consumers	231	Hands-on – creating data transformations with AWS Glue DataBrew	242
Meeting the needs of business users with data visualization	232	Configuring new datasets for AWS Glue DataBrew	242
AWS tools for business users	233	Creating a new Glue DataBrew project	243
Meeting the needs of data analysts with structured reporting	235	Building your Glue DataBrew recipe	245
AWS tools for data analysts	236	Creating a Glue DataBrew job	248
		Summary	250

9

Loading Data into a Data Mart

Technical requirements	252	Using a data warehouse as a transactional datastore	256
Extending analytics with data warehouses/data marts	252	Using a data warehouse as a data lake	256
Cold data	252	Using data warehouses for real-time, record-level use cases	257
Warm data	253	Storing unstructured data	257
Hot data	255	Redshift architecture review and storage deep dive	258
What not to do – anti-patterns for a data warehouse	256	Data distribution across slices	258
		Redshift Zone Maps and sorting data	261

Designing a high-performance data warehouse	262	Hands-on – loading data into an Amazon Redshift cluster and running queries	275
Selecting the optimal Redshift node type	262	Uploading our sample data to Amazon S3	276
Selecting the optimal table distribution style and sort key	263	IAM roles for Redshift	277
Selecting the right data type for columns	263	Creating a Redshift cluster	280
Selecting the optimal table type	268	Creating external tables for querying data in S3	282
Moving data between a data lake and Redshift	272	Creating a schema for a local Redshift table	287
Optimizing data ingestion in Redshift	272	Running complex SQL queries against our data	288
Exporting data from Redshift to the data lake	274	Summary	292

10

Orchestrating the Data Pipeline

Technical requirements	294	AWS Step Function for a serverless orchestration solution	306
Understanding the core concepts for pipeline orchestration	294	Pros and cons of using AWS Step Function	308
What is a data pipeline, and how do you orchestrate it?	295	Deciding on which data pipeline orchestration tool to use	309
How do you trigger a data pipeline to run?	297	Hands-on – orchestrating a data pipeline using AWS Step Function	311
How do you handle the failures of a step in your pipeline?	298	Creating new Lambda functions	311
Examining the options for orchestrating pipelines in AWS	299	Creating an SNS topic and subscribing to an email address	313
AWS Data Pipeline for managing ETL between data sources	300	Creating a new Step Function state machine	314
AWS Glue Workflows to orchestrate Glue resources	301	Configuring AWS CloudTrail and Amazon EventBridge	319
Apache Airflow as an open source orchestration solution	303	Summary	324
Pros and cons of using MWAA	305		

Section 3: The Bigger Picture: Data Analytics, Data Visualization, and Machine Learning

11

Ad Hoc Queries with Amazon Athena

Technical requirements	328	Managing governance and costs with Amazon Athena Workgroups	341
Amazon Athena – in-place SQL analytics for the data lake	329	Athena Workgroups overview	341
Tips and tricks to optimize Amazon Athena queries	330	Enforcing settings for groups of users	342
Common file format and layout optimizations	330	Enforcing data usage controls	343
Writing optimized SQL queries	334	Hands-on – creating an Amazon Athena workgroup and configuring Athena settings	344
Federating the queries of external data sources with Amazon Athena Query Federation	337	Hands-on – switching Workgroups and running queries	347
Querying external data sources using Athena Federated Query	338	Summary	352

12

Visualizing Data with Amazon QuickSight

Technical requirements	354	Ingesting and preparing data from a variety of sources	364
Representing data visually for maximum impact	355	Preparing datasets in QuickSight versus performing ETL outside of QuickSight	365
Benefits of data visualization	356	Creating and sharing visuals with QuickSight analyses and dashboards	367
Popular uses of data visualizations	356	Visual types in Amazon QuickSight	368
Understanding Amazon QuickSight's core concepts	361		
Standard versus enterprise edition	361		
SPICE – the in-memory storage and computation engine for QuickSight	362		

Understanding QuickSight's advanced features – ML Insights and embedded dashboards	372	Hands-on – creating a simple QuickSight visualization	376
Amazon QuickSight ML Insights	372	Setting up a new QuickSight account and loading a dataset	376
Amazon QuickSight embedded dashboards	375	Creating a new analysis	379
		Summary	385

13

Enabling Artificial Intelligence and Machine Learning

Technical requirements	388	Hands-on – reviewing reviews with Amazon Comprehend	407
Understanding the value of ML and AI for organizations	389	Setting up a new Amazon SQS message queue	407
Specialized ML projects	389	Creating a Lambda function for calling Amazon Comprehend	408
Everyday use cases for ML and AI	391	Adding Comprehend permissions for our IAM role	411
Exploring AWS services for ML	392	Adding a Lambda function as a trigger for our SQS message queue	412
AWS ML services	393	Testing the solution with Amazon Comprehend	413
Exploring AWS services for AI	398	Summary	415
AI for unstructured speech and text	399	Further reading	416
AI for extracting metadata from images and video	403		
AI for ML-powered forecasts	405		
AI for fraud detection and personalization	406		

14

Wrapping Up the First Part of Your Learning Journey

Technical requirements	418	A decade of data wrapped up for Spotify users	423
Looking at the data analytics big picture	418	Ingesting and processing streaming files at Netflix scale	424
Managing complex data environments with DataOps	420	Imagining the future – a look at emerging trends	428
Examining examples of real-world data pipelines	422	ACID transactions directly on data lake data	429

More data and more streaming ingestion	429	Implementations of the data mesh architecture	433
Multi-cloud	430	Hands-on – cleaning up your AWS account	434
Decentralized data engineering teams, data platforms, and a data mesh architecture	430	Reviewing AWS Billing to identify the resources being charged for	435
Data and product thinking convergence	432	Closing your AWS account	437
Data and self-serve platform design convergence	432	Summary	439

Other Books You May Enjoy

Index

Preface

We live in a world where the amount of data being generated is constantly increasing. While a few decades ago, an organization may have had a single database that could store everything they needed to track, today most organizations have tens, hundreds, or even thousands of databases, along with data warehouses, and perhaps a data lake. And these data stores are being fed from an increasing number of data sources (transaction data, web server log files, IoT and other sensors, and social media, to name just a few).

It is no surprise that we hear more and more companies talk about being data-driven in their decision making. But in order for an organization to be truly data-driven, they need to be masters of managing and drawing insights from these ever-increasing quantities and types of data. And to enable this, organizations need to employ people with specialized data skills.

Doing a search on LinkedIn for jobs related to data returns over 1.5 million results (and that is just for the United States!). The job titles include roles such as data engineers (with 185,000 results), data scientists (120,000 results), and data architects (75,000 results).

While this book will not magically make you a data engineer, it has been designed to accelerate your journey toward data engineering on AWS. By the end of this book, you will not only have learned some of the core concepts around data engineering, but you will also have a good understanding of the wide variety of tools available in AWS for working with data. You will also have been through numerous hands-on exercises, gaining practical experience with things such as ingesting streaming data, transforming and optimizing data, building visualizations, and even drawing insights from data using AI.

Who this book is for

This book has been designed for two groups of people; firstly, those people looking to get started with a career in data engineering, and who want to learn core data engineering concepts. This book introduces many different aspects of data engineering, providing a comprehensive high-level understanding of, and practical hands-on experience with, different focus areas of data engineering.

Secondly, this book is for those people who may already have an established career focused on data, but who are new to the cloud, and to AWS in particular. For these people, this book provides a clear understanding of many of the different AWS services for working with data and gets them hands-on experience with a variety of these AWS services.

What this book covers

Each of the chapters in this book takes the approach of introducing important concepts and key AWS services and then providing a hands-on exercise related to the topic of the chapter:

Chapter 1, An Introduction to Data Engineering, reviews the challenges of ever-increasing datasets, and the role of the data engineer in working with data in the cloud.

Chapter 2, Data Management Architectures for Analytics, introduces foundational concepts and technologies related to big data processing.

Chapter 3, The AWS Data Engineer's Toolkit, provides an introduction to a wide range of AWS services that are used for ingesting, processing, and consuming data.

Chapter 4, Data Cataloging, Security, and Governance, covers the all-important topics of keeping data secure, ensuring good data governance, and the importance of cataloging your data.

Chapter 5, Architecting Data Engineering Pipelines, provides an approach for whiteboarding the high-level design of a data engineering pipeline.

Chapter 6, Ingesting Batch and Streaming Data, looks at the variety of data sources that we may need to ingest from and examines AWS services for ingesting both batch and streaming data.

Chapter 7, Transforming Data to Optimize for Analytics, covers common transformations for optimizing datasets and for applying business logic.

Chapter 8, Identifying and Enabling Data Consumers, is about better understanding the different types of data consumers that a data engineer may work to prepare data for.

Chapter 9, Loading Data into a Data Mart, focuses on the use of data warehouses as a data mart and looks at moving data between a data lake and data warehouse. This chapter also does a deep dive into Amazon Redshift, a cloud-based data warehouse.

Chapter 10, Orchestrating the Data Pipeline, looks at how various data engineering tasks and transformations can be put together in a data pipeline, and how these can be run and managed with pipeline orchestration tools such as AWS Step Functions.

Chapter 11, Ad Hoc Queries with Amazon Athena, does a deeper dive into the Amazon Athena service, which can be used for running SQL queries directly on data in the data lake, and beyond.

Chapter 12, Visualizing Data with Amazon QuickSight, discusses the importance of being able to craft visualizations of data, and how the Amazon QuickSight service enables this.

Chapter 13, Enabling Artificial Intelligence and Machine Learning, reviews how AI and ML are increasingly important for gaining new value from data, and introduces some of the AWS services for both ML and AI.

Chapter 14, Wrapping Up the First Part of Your Learning Journey, concludes the book by looking at the bigger picture of data analytics, including real-world examples of data pipelines and a review of emerging trends in the industry.

To get the most out of this book

Basic knowledge of computer systems and concepts, and how these are used within large organizations, is helpful prerequisite knowledge for this book. However, no data engineering-specific skills or knowledge is required. Also, a familiarity with cloud computing fundamentals and core AWS systems will make it easier to follow along, especially with the hands-on exercises, but detailed step-by-step instructions are included for each task.

Software/hardware covered in the book	Operating system requirements
Amazon Web Services (AWS)	A stable internet connection and a recent version of a modern web browser are required to access the AWS Management Console. For more information, refer to the following link: https://aws.amazon.com/premiumsupport/knowledge-center/browsers-management-console/ .

All hands-on exercises make use of cloud-based services, so beyond using a supported web browser with a stable internet connection, there are no additional hardware or software requirements.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Things change fast in the computing industry, and this is clearly seen within the cloud industry. AWS is constantly rolling out new services, as well as improvements for existing services, and some of these improvements lead to changes in the user interface provided via the AWS console.

As a result, some of the screenshots included in this book may not look identical to what you are seeing in the AWS console when completing hands-on exercises. Or, you may find that a specific screen has additional options beyond what is shown in the screenshot in this book. It is unlikely that these changes will prevent you from following along with the step-by-step instructions in this book, but anything that may significantly impact a hands-on exercise will be addressed with a note for that chapter in this book's GitHub repository. Therefore, please refer to the GitHub repository as you complete the hands-on exercises for each chapter. In addition to notes about any significant console changes, the GitHub repository also includes copies of code contained in this book and other useful resources.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Data-Engineering-with-AWS>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781800560413_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in the text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system."

A block of code is set as follows:

```
import boto3
import awswrangler as wr
from urllib.parse import unquote_plus
```

Any command-line input or output is written as follows:

```
$ aws s3 cp test.csv s3://dataeng-landing-zone-initials/
testdb/csvparquet/test.csv
```

Bold: Indicates a new term, an important word, or words that you see on screen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Select **System info** from the **Administration** panel."

Tips or Important Notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customer-care@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Data Engineering with AWS*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Section 1: AWS Data Engineering Concepts and Trends

To start with, we examine why data is so important to organizations today, and introduce foundational concepts of data engineering, including coverage of governance and security topics. We also learn about the AWS services that form part of the data engineer's toolkit, and get hands-on with creating an AWS account and using services such as Amazon S3, AWS Lambda, and AWS **Identity and Access Management (IAM)**.

This section comprises the following chapters:

- *Chapter 1, An Introduction to Data Engineering*
- *Chapter 2, Data Management Architectures for Analytics*
- *Chapter 3, The AWS Data Engineer's Toolkit*
- *Chapter 4, Data Cataloging, Security, and Governance*

1

An Introduction to Data Engineering

Data engineering is a fast-growing career path, and a role in high demand, as data becomes ever more critical to organizations of all sizes. For those that enjoy the challenge of putting together the "puzzle pieces" that build out complex data pipelines to ingest raw data, and to then transform and optimize that data for various data consumers, it can be a really rewarding career.

In this chapter, we look at the many ways that data has become an important and valuable corporate asset. We also review some of the challenges that organizations face as they deal with increasing volumes of data, and how data engineers can use cloud-based services to help overcome these challenges. We then set the foundations for the rest of the hands-on activities in this book by providing step-by-step details on creating a new **Amazon Web Services (AWS)** account.

Throughout this book, we are going to cover a number of topics that teach the fundamentals of developing data engineering pipelines on AWS, but we'll get started in this chapter with these topics:

- The rise of big data as a corporate asset
- The challenges of ever-growing datasets
- The role of the data engineer as a big data enabler
- The benefits of the cloud when building big data analytic solutions
- Hands-on - create or access an AWS account for following along with the hands-on activities in this book

Technical requirements

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/tree/main/Chapter01>

The rise of big data as a corporate asset

You don't need to look too far or too hard these days to hear about how big data and data analytics are transforming organizations and having an impact on society as a whole. We hear about how companies such as **TikTok** analyze large quantities of data to make personalized recommendations about which clip to show a user next. Also, we know how Amazon recommends products a customer may be interested in based on their purchase history. We read headlines about how big data could revolutionize the healthcare industry, or how stock pickers turn to big data to find the next breakout stock performer when the markets are down.

The most valuable companies in the US today are companies that are masters of managing huge data assets effectively, with the top five most valuable companies in Q4 2021 being the following:

- **Microsoft**
- **Apple**
- **Alphabet (Google)**
- **Amazon**
- **Tesla**

For a long time, it was companies that managed natural gas and oil resources, such as **ExxonMobil**, that were high on the list of the most valuable companies on the US stock exchange. Today, ExxonMobil will often not even make the list of the top 30 companies. It is no wonder that the number of job listings for people with skillsets related to big data is on the rise.

There is also no doubt that data, when harnessed correctly and optimized for maximum analytic value, can be a game-changer for an organization. At the same time, those companies that are unable to effectively utilize their data assets risk losing a competitive advantage to others that do have a comprehensive data strategy and effective analytic and machine learning programs.

Organizations today tend to be in one of the following three states:

- They have an effective data analytics and machine learning program that differentiates them from their competitors.
- They are conducting **proof of concept** projects to evaluate how analytic and machine learning programs can help them achieve a competitive advantage.
- Their leaders are having sleepless nights worrying about how their competitors are using analytics and machine learning programs to achieve a competitive advantage over them.

No matter where an organization currently is in their data journey, if they have been in existence for a while, they have likely faced a number of common data-related challenges. Let's look at how organizations have typically handled the challenge of ever-growing datasets.

The challenges of ever-growing datasets

Organizations have many assets, such as physical assets, intellectual property, the knowledge of their employees, and trade secrets. But for too long, organizations did not fully recognize that they had another extremely valuable asset, and they failed to maximize the use of it—the vast quantities of data that they had gathered over time.

That is not to say that organizations ignored these data assets, but rather, due to the expense and complex nature of storing and managing this data, organizations tended to only keep a subset of data.

Initially, data may have been stored in a single database, but as organizations, and their data requirements, grew, the number of databases exponentially increased. Today, with the modern application development approach of microservices, companies commonly have hundreds, or even thousands, of databases. Faced with many data silos, organizations invested in data warehousing systems that would enable them to ingest data from multiple siloed databases into a central location for analytics. But due to the expense of these systems, there were limitations on how much data could be stored, and some datasets would either be excluded or only aggregate data would be loaded into the data warehouse. Data would also only be kept for a limited period of time as data storage for these systems was expensive, and therefore it was not economical to keep historical data for long periods. There was also a lack of widely available tools and compute power to enable the analysis of extremely large, comprehensive datasets.

As an organization continued to grow, multiple data warehouses and data marts would be implemented for different business units or groups, and organizations still lacked a centralized, single-source-of-truth repository for their data. Organizations were also faced with new types of data, such as semi-structured or even unstructured data, and analyzing these datasets with traditional tooling was a challenge.

As a result, new technologies were invented that were able to better work with very large datasets and different data types. Hadoop was a technology created in the early 2000s at **Yahoo** as part of a search engine project that wanted to index 1 billion web pages. Over the next few years, **Hadoop**, and the underlying **MapReduce** technology, became a popular way for all types of companies to store and process much larger datasets. However, running a Hadoop cluster was a complex and expensive operation requiring specialized skills.

The next evolution for big data processing was the development of **Spark** (later taken on as an **Apache** project and now known as **Apache Spark**), a new processing framework for working with big data. Spark showed significant increases in performance when working with large datasets due to the fact that it did most processing in memory, significantly reducing the amount of reading and writing to and from disks. Today, Apache Spark is often regarded as the gold standard for processing large datasets and is used by a wide array of companies, although there are still a lot of Hadoop MapReduce clusters in production in many companies.

In parallel with the rise of Apache Spark as a popular big data processing tool was the rise of the concept of data lakes—an approach that uses low-cost object storage as a physical storage layer for a variety of data types, provides a central catalog of all the datasets, and makes that data available for processing with a wide variety of tools, including Apache Spark. AWS uses the following definition when talking about data lakes:

A data lake is a centralized repository that allows you to store all your structured and unstructured data at any scale. You can store your data as-is, without having to first structure the data, and run different types of analytics—from dashboards and visualizations to big data processing, real-time analytics, and machine learning to guide better decisions.

You can find this definition here: <https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/>.

Having looked at how data analytics became an essential tool in organizations, let's now look at the roles that enable maximizing the value of data for a modern organization.

Data engineers – the big data enablers

Amid the increasing recognition of data as a valuable corporate asset and the introduction of new technologies to store and process vast amounts of data, there has been an increase in the opportunities and roles available for data-related careers.

Let's look at a sample use case, where a sales manager for a consumer goods organization wants to better understand which alternative products a customer considers before purchasing their product. In addition, they also want to have a better way of predicting product demand by category based on external factors, such as the expected weather.

Achieving the desired outcomes as specified by the sales manager will require bringing in data from multiple internal and external sources. Datasets that could be relevant to this scenario may include the following:

- Customer, product, and order relational databases
- Web server logs from the consumer-facing storefront
- Third-party sales data from online marketplaces where relevant products are sold (such as Amazon.com)
- Other relevant third-party datasets that may influence sales (for example, weather-related data)

Multiple teams would need to be involved in the project, with the following three roles playing a primary part in implementing the required solution.

Understanding the role of the data engineer

The role of a data engineer is to do the following:

- Design, implement, and maintain the pipelines that enable the ingestion of raw data into a storage platform.
- Transform that data to be optimized for analytics.
- Make that data available for various data consumers using their tool of choice.

In our scenario, the data engineer will first need to design the pipelines that ingest data from the various internal and external sources. To achieve this, they will use a variety of tools (more on that in future chapters), depending on the source system and whether it will be scheduled batch ingestion or real-time streaming ingestion.

The data engineer is also responsible for transforming the raw input datasets to optimize them for analytics, using various techniques (as discussed later in this book). The data engineer must also create processes to verify the quality of data, add metadata about the data to a data catalog, and manage the life cycle of code related to data transformation.

Finally, the data engineer may need to assist in integrating various data consumption tools with the transformed data, enabling data analysts and data scientists to use their preferred tools to draw insights from the data.

The data engineer uses tools such as **Apache Spark**, **Apache Kafka**, and **Presto**, as well as other commercially available products, to build the data pipeline and optimize data for analytics.

The data engineer is much like a civil engineer for a new residential development. The civil engineer is responsible for designing and building the roads, bridges, train stations, and so on to enable commuters to easily commute in and out of the development, while the data engineer is responsible for designing and building the infrastructure required to bring data into a central source and for optimizing the data for use by various data consumers.

Understanding the role of the data scientist

The role of a data scientist is to draw complex insights and make predictions based on various datasets, using machine learning and artificial intelligence. The data scientist will combine a number of skills, including computer science, statistics, analytics, and math, in order to help an organization answer complex questions and make informed decisions using data.

Data scientists need to understand the raw data and know how to use that data to develop and train complex machine learning models that will help recognize patterns in the data and predict future trends. In our scenario, the data scientist may build a machine learning model that uses past sales data, correlated with weather information for each day in the reporting period. They can then design and train this model to help business users get predictions on the likely top-selling categories for future dates based on the expected weather forecast.

Where the data engineer is like a civil engineer building infrastructure for a new development, the data scientist is developing cars, airplanes, and other forms of transport used to move in and out of the development. Data scientists create machine learning models that enable data consumers and business analysts to draw new insights and predictions from data.

Understanding the role of the data analyst

The role of a data analyst is to examine and combine multiple datasets in order to help a business understand trends in the data and to make more informed business decisions. While a data scientist develops models that make future predictions or identifies non-obvious patterns in data, the data analyst works with well-structured and modeled data to understand current conditions and to highlight recent patterns from the data.

A data analyst may answer questions such as which menu item sold best in different geographic regions over the past month, or which medical procedure had the best outcome for patients of different ages. These insights help an organization make better decisions for the future.

In our scenario, the data analyst may run complex queries against the different datasets that are available (such as an orders database or web server logs), joining together subsets of data from each source to gain new insights. For example, the data analyst may create a report highlighting which alternate products are most often browsed by a customer before a specific product is purchased. The data analyst may also make use of advanced machine learning models developed by the data scientists to gain further valuable insights.

Where the data engineer is like a civil engineer building infrastructure, and the data scientist is developing means of transportation, the data analyst is like a skilled pilot, using their expertise to get users to their end destination.

Understanding other common data-related roles

Organizations may have other role titles and job descriptions for data-related positions, but generally, these will be a subset of the roles described in the preceding sections.

For example, a **big data architect** could be a subset of the **data engineer** role, focused on designing the architecture for big data pipelines, but not building the actual pipelines. Or, a **data visualization developer** may be focused on building out visualizations using business intelligence tools, but this is effectively a subset of the **data analyst** role.

Larger organizations tend to have more focused job roles, while in a smaller organization a single person may take on the role of data engineer, data scientist, and data analyst.

In this book, we will focus on the role of the data engineer, and dive deep into how a data engineer is able to build complex data pipelines using the power of cloud computing services. Let's now look at how cloud computing has simplified how organizations are able to build and scale out big data processing solutions.

The benefits of the cloud when building big data analytic solutions

For a long time, organizations relied on complex systems that they would run in their own data centers to help them capture, store, and process large amounts of data. But over the last decade, there has been a trend of an increasing amount of data that organizations want to store and analyze, and on-premises systems have struggled to scale to keep up with demand. Scaling up these traditional tools for managing ever-increasing datasets has been expensive, complex, and time-consuming, and organizations have been seeking alternative solutions to cope with the increasing data volumes.

Ever since Amazon launched AWS in 2006, organizations have been realizing the benefits of running their workloads in the cloud. Cloud computing enables scalability, cost efficiency, security, and automation, which most companies find impossible to achieve within their own data centers, and this applies to the area of data analytics as well. One of the first AWS services was **Amazon Simple Storage Service (Amazon S3)**, a cloud-based object store that offers essentially unlimited scalability at low cost, and yet provides durability and availability that most data center managers could only dream of achieving. Today, Amazon S3 has become the physical storage layer for thousands of data lake projects, and a wide ecosystem of analytic tools has been created to work with the service.

Successful data engineers need to understand the tools available in the cloud for building out complex data analytic projects and understand which set of tools is best to achieve the outcome needed for their project. In this book, you will learn more about AWS tools for working with big data, and you will gain hands-on experience in developing a data engineering pipeline in AWS.

To get started, you will either need an existing AWS account or you will need to create a new AWS account so that you can follow along with the practical examples. Follow along with the next section as we provide step-by-step instructions for creating a new AWS account.

Hands-on – creating and accessing your AWS account

The projects in this book require you to access an AWS account with administrator privileges. If you already have administrator privileges for an AWS account and know how to access the AWS Management Console, you can skip this section and move on to *Chapter 2, Data Marts, Data Lakes, and the Data Lakehouse*.

If you are making use of a corporate AWS account, you will want to check with your AWS cloud operations team to ensure that your account has administrative privileges. Even if your daily-use account does not allow full administrative privileges, your cloud operations team may be able to create a **sandbox** account for you.

What is a sandbox account?

A sandbox account is an account isolated from your corporate production systems with relevant guardrails and governance in place, and is used by many organizations to provide a safe space for teams or individual developers to experiment with cloud services.

If you cannot get administrative access to a corporate account, you will need to create a personal AWS account or work with your cloud operations team to request specific permissions needed to complete each section. Where possible, we will provide links to AWS documentation that will list the required permissions, but the full details of the required permissions will not be covered directly in this book.

Important note about the costs associated with the hands-on tasks in this book

If you are creating a new personal account or using an existing personal account, you will incur and be responsible for AWS costs as you follow along in this book. While some services may fall under AWS free-tier usage, some of the services covered in this book will not. We strongly encourage you to set up budget alerts within your account and to regularly check your billing console.

See the AWS documentation on monitoring your usage and costs at <https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/monitoring-costs.html>.

Creating a new AWS account

To create a new AWS account, you will need the following things:

- An email address (or alias) that has not been used before to register an AWS account
- A phone number that can be used for important account verification purposes
- Your credit or debit card, which will be charged for AWS usage outside of the Free Tier

Tip regarding the phone number you use when registering

It is important that you keep your contact details up to date for your AWS account, as if you lose access to your account, you will need access to the email address and phone number registered for the account. If you expect that your contact number may change in the future, consider registering a virtual number that you will always be able to access and that you can forward to your primary number. One such service that enables this is **Google Voice** (<http://voice.google.com>).

The following steps will guide you through creating a new AWS account:

1. Navigate to the AWS landing page at <http://aws.amazon.com>.
2. Click on the **Create an AWS Account** link.
3. Provide an email address, specify a secure password (one that you have not used elsewhere), and provide a name for your account.

Tip about reusing an existing email address

Some email systems support adding a + sign followed by a few characters to the end of the username portion of your email address in order to create a unique email address that still goes to your same mailbox. For example, `atest.emailaddress@gmail.com` and `atest.emailaddress+dataengineering@gmail.com` will both go to the primary email address inbox. If you have used your primary email address previously to register an AWS account, you can use this tip to provide a unique email address during registration, but still have emails delivered to your primary account.

4. Select **Professional** or **Personal** for the account type (note that the functionality and tools available are the same no matter which one you pick).

Free Tier offers

All AWS accounts can explore 3 different types of free offers, depending on the product used.



Always free
Never expires



12 months free
Start from initial sign-up date



Trials
Start from service activation date

Sign up for AWS

Contact Information

How do you plan to use AWS?

- Business - for your work, school, or organization
- Personal - for your own projects

Who should we contact about this account?

Full Name

Gareth Eagar

Phone Number

Enter your country code and your phone number.

Country or Region

United States ▼

Address

_____ Road

Apartment, suite, unit, building, floor, etc.

City

State, Province, or Region

Postal Code

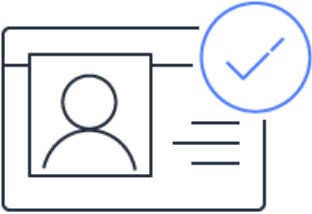

I have read and agree to the terms of the [AWS Customer Agreement](#).

Continue (step 2 of 5)

Figure 1.1 – Contact information during AWS account sign-up

5. Provide the requested personal information and then after reviewing the terms of the AWS Customer Agreement, click the checkbox if you agree to the terms, and then click on **Create Account and Continue**.
6. Provide a credit or debit card for payment information and select **Verify and Add**.

7. Provide a phone number for a verification text or call, enter the characters shown for the security check, and complete the verification.



Sign up for AWS

Confirm your identity

Before you can use your AWS account, you must verify your phone number. When you continue, the AWS automated system will contact you with a verification code.

How should we send you the verification code?

Text message (SMS)


Voice call

Country or region code

United States (+1) ▼

Mobile phone number

Security check



Type the characters as shown above

Send SMS (step 4 of 5)

Figure 1.2 – Confirming your identity during AWS account sign-up

8. Select a support plan.
9. You will receive a notification that your account is being activated. This usually completes in a few minutes, but it can take up to 24 hours. Check your email to confirm account activation.

What to do if you don't receive a confirmation email within 24 hours

If you do not receive an email confirmation within 24 hours confirming that your account has been activated, follow the troubleshooting steps provided by AWS Premium Support at <https://aws.amazon.com/premiumsupport/knowledge-center/create-and-activate-aws-account/>.

Accessing your AWS account

Once you have received the confirmation email confirming that your account has been activated, follow these steps to access your account and to create a new admin user:

1. Access the AWS console login page at <http://console.aws.amazon.com>.
2. Make sure **Root user** is selected, and then enter the email address that you used when creating the account.
3. Enter the password that you set when creating the account.

Best practices for securing your account

When you log in using the email address you specified when registering the account, you are logging in as the account's **root user**. It is a recommended best practice that you do not use this login for your day-to-day activities, but rather only use this when performing activities that require the root account, such as creating your first **Identity and Access Management (IAM)** user, deleting the account, or changing your account settings. For more information, see https://docs.aws.amazon.com/IAM/latest/UserGuide/id_root-user.html.

It is also strongly recommended that you enable **Multi-Factor Authentication (MFA)** on this and other administrative accounts. To enable this, see https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_mfa_enable_virtual.html.

In the following steps, we are going to create a new IAM administrative user account:

1. In the AWS Management Console, confirm which Region you are currently in. You can select any region, such as the Region closest to you geographically.

Important note about pricing differences in AWS Regions

Note that pricing for AWS services differs from Region to Region, so take this into account when selecting a Region to use for the exercises in this book and make sure you are always in the same Region when working through the exercises.

In the following screenshot, the user is in the **Ohio** Region (also known as us-east-2):

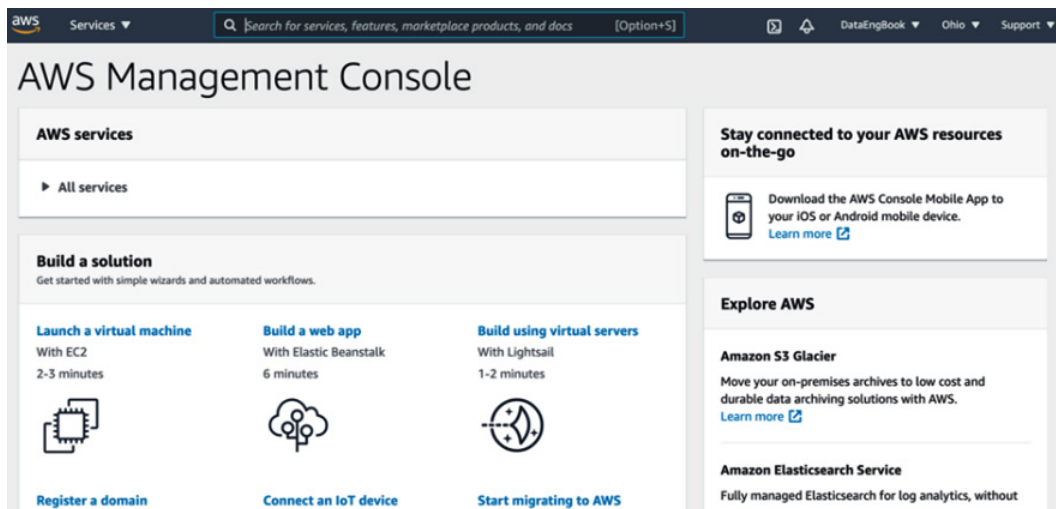


Figure 1.3 – AWS Management Console

2. In the search bar in the top middle of the screen, type in IAM and press *Enter*. This brings up the console for IAM.
3. On the left-hand side menu, click **Users** and then **Add user**.
4. Provide a username and select both **Programmatic access** as well as **AWS Management Console access**.

- Set a password for the console, and select whether to force a password change on the next login, then click **Next: Permissions**.

Add user

1 2 3 4 5

Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name*

[+ Add another user](#)

Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)

Access type* **Programmatic access**
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.

AWS Management Console access
Enables a **password** that allows users to sign-in to the AWS Management Console.

Console password* Autogenerated password
 Custom password

Show password

Require password reset User must create a new password at next sign-in
Users automatically get the [IAMUserChangePassword](#) policy to allow them to change their own password.

* Required [Cancel](#) [Next: Permissions](#)

Figure 1.4 – Creating a new user in the AWS Management Console

- For production accounts, it is best practice to grant permissions with a policy of least privilege, giving each user only the permissions they specifically require to perform their role. However, AWS managed policies can be used to cover common use cases in test accounts, and so to simplify the setup of our test account, we will use the **AdministratorAccess** managed policy. This policy gives full access to all AWS resources in the account.

On the **Set permissions** screen, select **Attach existing policies directly**. From the list of policies, select **AdministratorAccess**. Then, click **Next: Tags**.

7. Optionally, specify tags (key-value pairs), then click **Next: Review**.
8. Review the settings, and then click **Create user**.
9. Take note of the URL to sign in to your account.
10. Take note of the access key ID and secret access key as you will need these later. This is the only opportunity you will have to record the secret access key so it is important to safely record this information now:

Add user

1 2 3 4 5

✔ **Success**

You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.

Users with AWS Management Console access can sign-in at: [https://51\[redacted\].aws.amazon.com/console](https://51[redacted].aws.amazon.com/console)

[Download .csv](#)

	User	Access key ID	Secret access key	Email login instructions
▶	✔ g[redacted] in	AK[redacted]7H	***** Show	Send email ↗

Figure 1.5 – Successful creation of new IAM user

Important note about protecting your account

Make sure you protect this information as anyone who has access to your access key ID and secret access key is able to perform full administrative functions in your account, including deploying resources that you will be responsible for paying for.

For the remainder of the tutorials in this book, you should log in using the URL provided and the username and password you set for your IAM user. You should also strongly consider enabling MFA for this account, a recommended best practice for all accounts with administrator permissions.

Summary

In this chapter, we reviewed how data is becoming ever more important for organizations looking to gain new insights and competitive advantage, and introduced some of the core big data processing technologies. We also looked at the key roles related to managing, processing, and analyzing large datasets, and highlighted how cloud technologies enable organizations to better deal with the increasing volume, variety, and velocity of data.

In our first hands-on exercise, we provided step-by-step instructions for creating a new AWS account that can be used throughout the remainder of this book as we develop our own data engineering pipeline.

In the next chapter, we dig deeper into current approaches, tools, and frameworks that are commonly used to manage and analyze large datasets, including data warehouses, data marts, data lakes, and a relatively new concept, the data lake house. We also get hands-on with AWS again, this time installing and configuring the AWS **Command-Line Interface (CLI)** tool and creating an Amazon S3 bucket.

2

Data Management Architectures for Analytics

In *Chapter 1, An Introduction to Data Engineering*, we looked at the challenges introduced by ever-growing datasets, and how the cloud can help solve these analytical challenges. However, there are many different cloud services, open source frameworks, and architectures that can be used in analytical projects, depending on business requirements and objectives.

In this chapter, we will discuss how analytical technologies have evolved and introduce the key technologies and concepts that are foundational for building modern analytical architectures, irrespective of whether you build them on AWS or elsewhere.

If you have experience as a data engineer and have worked with enterprise data warehouses before, you may want to skim through the sections of this chapter, and then do the hands-on exercise at the end of this chapter. However, if you are new to data engineering and do not have experience with big data analytics, then the content in this chapter is important as it will provide an understanding of concepts that we will build on in the rest of this book.

In this chapter, we will cover the following topics:

- The evolution of data management for analytics
- An introduction to data warehouses, data marts, and ETL/ELT pipelines
- An overview of data lake architecture and concepts
- A deeper dive into an emerging architecture, the data lakehouse
- Hands-on – configuring the AWS Command Line Interface tool and creating an S3 bucket

Technical requirements

To complete the hands-on exercises included in this chapter, you will need access to an AWS account where you have administrator privileges (as covered in *Chapter 1, An Introduction to Data Engineering*).

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/tree/main/Chapter02>

The evolution of data management for analytics

Innovations in data management and processing over the last several decades have laid the foundations of modern-day analytic systems. When you look at the analytics landscape of a typical mature organization, you will find the footprints of many of these innovations in their data analytics platforms. As a data engineer, you may come across analytic pipelines that were built using technologies from different generations, and you may be expected to understand them. Therefore, it is important to be familiar with some of the key developments in analytics over time, as well as to understand the foundational concepts of analytical data storage, data management, and data pipelines.

Databases and data warehouses

Data processing and analytic systems have evolved over several decades. In the 1980s, the focus was on batch processing, where data would be processed in nightly runs on large mainframe computers.

In the 1990s, the use of databases exploded, and organizations found themselves with tens, or even hundreds, of databases supporting different business processes. Generally, these databases were for transactional processing, and the ability to perform analytics across systems was very limited.

As a result, in the 1990s, data warehouses become a popular tool where data could be ingested from multiple databases systems into a central repository, and the data warehouse could focus on running analytic reports.

The data warehouse was designed to store **well-integrated, highly structured, highly curated, and highly trusted** data. Data would be ingested on a regular basis from other highly structured sources, but before entering the data warehouse, the data would go through a significant amount of preprocessing, validation, and transformations. Any changes to the data warehouse schema, or the need to ingest new data sources, would require a significant effort to plan the schema and related processes.

Over the last few decades, businesses and consumers have rapidly adopted web and mobile technologies, and this has resulted in rapid growth in data sources, data volumes, and options to analyze an increasing amount of data. In parallel, organizations have realized the business value of insights they can gain by combining data from their internal systems with external data from their partners, the web, social media, and third-party data providers. To process increasingly larger data volumes and increased demands to support new consumers, data warehouses have evolved through multiple generations of technology and architectural innovations.

Early data warehouses were custom-built using common relational databases on powerful servers, but they required IT teams to manage host servers, storage, software, and integrations with data sources. These were difficult to manage, and so in the mid-2000s, there was an emergence of purpose-built hardware appliances designed as *modular data warehouse appliances built for terabyte- and petabyte-scale big data processing*. These appliances contained new hardware and software innovations and were delivered as easy to install and manage units from popular vendors such as **Oracle, Teradata, IBM Netezza, Pivotal Greenplum**, and others.

Dealing with big, unstructured data

While data warehouses have steadily evolved over the last 25+ years to support increasing volumes of highly structured data, there has been exponential growth in semi-structured and unstructured data produced by modern digital platforms (such as mobile and web applications, sensors, IoT devices, social media, and audio and video media platforms). These platforms produce data at a high velocity, and in much larger volumes than data produced by traditional structured sources. To gain a competitive advantage by transforming customer experience and business operations, it has become essential for organizations to gain deeper insights from these new data sources. Data warehouses are good at storing and managing flat, structured data from traditional sources as a set of tables, organized as a relational schema. However, they are not well suited to handling the huge volumes of high velocity semi-structured and unstructured data, which are becoming increasingly popular.

As a result, in the early 2010s, new technologies for big data processing became popular. **Hadoop**, an open source framework for processing large datasets on clusters of computers, became the leading way to work with big data. These clusters contain tens of hundreds of machines with attached disk volumes that can hold tens of thousands of terabytes of data managed under a single distributed filesystem known as the **Hadoop Distributed File System (HDFS)**.

Many organizations deployed **Hadoop distributions** from providers such as **Cloudera**, **Hortonworks**, **MapR**, and **IBM** to large clusters of computers in their data centers. These Hadoop packages include cluster management utilities, as well as pre-configured and pre-integrated open source distributed data processing frameworks such as **map-reduce**, **Hive**, **Spark**, and **Presto**. Distributed data processing frameworks such as **Apache Spark** have also been built to process a wide variety of structured, semi-structured, and unstructured data, and can provide very high throughput at hundreds of terabytes by distributing processing across thousands of machines in a cluster.

However, building and scaling on-premises Hadoop and Spark clusters typically requires a large upfront capital investment in machines and storage. The ongoing management of the cluster and big data processing pipelines requires a team of specialists that includes the following:

- Hadoop administrators specialized in cluster hardware and software
- Data engineers specialized in processing frameworks such as Spark, Hive, and Presto

As the volume of data grows, new machines and storage continually need to be added to the cluster.

Big data teams managing on-premises clusters typically spend a significant percentage of their time managing and upgrading the cluster's hardware and software, as well as optimizing workloads.

A lake on the cloud and a house on that lake

Over the last decade, organizations have been increasingly adopting public cloud infrastructure to reap the benefits of the following:

- On-demand capacity
- Limitless and elastic scaling
- Global footprint
- Usage-based cost models
- Freedom from managing hardware

Through both infrastructure and software as service models, cloud providers such as **Amazon Web Services (AWS)**, **Google Compute Cloud (GCP)**, and **Microsoft Azure** are enabling organizations to build new applications, as well as migrate their on-premises workloads to the public cloud.

Since AWS made **Amazon Redshift** available in 2013, leading cloud providers have started providing data warehouses as a cloud-native service. Over time, cloud data warehouses have rapidly expanded their feature sets to exceed those of high-performance, on-premises data warehousing appliances. Besides no upfront investment, petabyte scale, and high performance, cloud data warehouse services provide elastic capacity scaling, variable cost, and freedom from infrastructure management.

Since the arrival of cloud data warehouse services, the number of organizations building their data warehouses in the cloud has been accelerating. In the last 5 years alone, thousands of organizations have either built new warehouses or migrated their existing data warehouses and analytics workloads from on-premises vendor products and appliances to cloud data warehousing services (such as **Amazon Redshift**, **Snowflake**, **Google BigQuery**, and **Azure Synapse**).

Many organizations have deployed both data warehouses and big data clusters in their data centers to manage and analyze structured and semi-structured data, respectively. They have also had to build and manage data movement pipelines to support the use cases that require integrating data from both data warehouses and big data clusters. Often, silos and data movement have made it difficult to have a single source of truth, which has led to delays in getting the required insights.

Another trend in recent years has been the adoption of highly durable, inexpensive, and virtually limitless cloud object stores. Cloud object stores, such as Amazon S3, can store hundreds of petabytes of data at a fraction of the cost of on-premises storage, and they support storing data regardless of its source, format, or structure. They also provide native integrations with hundreds of cloud-native and third-party data processing and analytics tools.

These new cloud object stores have enabled organizations to build a new, more integrated analytics data management approach, called the *data lake architecture*. A data lake architecture makes it possible to create a single source of truth by bringing together a variety of data of all sizes and types (structured, semi-structured, unstructured) in one place: a central, highly scalable repository built using inexpensive cloud storage. In the last few years, thousands of organizations have built a data lake using cloud technologies, and that trend is accelerating.

Instead of lifting and shifting existing data warehouses and Hadoop clusters to the cloud, many organizations are refactoring their previously on-premises workloads to build an integrated cloud data lake. In this approach, all the data is ingested and processed in the data lake to build a single source of truth, and then a subset of the *hot* data is loaded into the dimensional schemas of a cloud data warehouse to support lower latency access.

A new trend we have seen over the last few years is cloud providers adding capabilities to their analytics services that support the emergence of a newer data analytics architecture, called *lake house architecture*, or *data lakehouse*. The lake house architecture approach is geared to natively integrate the best capabilities of data lakes and data warehousing, including the following:

- Ability to quickly ingest any type of data
- Storing and processing petabytes of unstructured, semi-structured, and structured data
- Support for **ACID** transactions (the ability to concurrently read, insert, update, and delete records in a dataset managed by the data lakehouse)
- Low latency data access
- Ability to consume data with a variety of tools, including SQL, Spark, machine learning frameworks, and business intelligence tools

Several competing data lakehouse offerings are currently being developed by various companies, but notable among them are Redshift Spectrum and Lake Formation on AWS, Synapse on Microsoft Azure cloud, and Databricks Delta Lake.

We will cover the data lakehouse architecture in more detail in the *Bringing together the best of both worlds with a data lakehouse* section, but first, we dive deeper into data warehousing concepts.

Understanding data warehouses and data marts – fountains of truth

An **Enterprise Data Warehouse (EDW)** is the central data repository that contains structured, curated, consistent, and trusted **data assets** that are organized into a well-modeled schema. The *data assets* in an EDW are made up of all the relevant information about key business domains and are built by integrating data sourced from the following places:

- Run-the-business applications (ERPs, CRMs, Line of Business applications) that support all the key business domains across the enterprise.
- External data sources such as data from partners and third parties.

An enterprise data warehouse provides business users and decision-makers with an easy-to-use, central platform that helps them find and analyze a well-modeled, well-integrated, single version of truth about various business subject areas such as customer, product, sales, marketing, supply chain, and more. Business users analyze data in the warehouse to measure business performance, find current and historical business trends, find business opportunities, and understand customer behavior.

In the remainder of this section, we will review the foundational concepts of a data warehouse by discussing a typical data management architecture with an EDW at the center, as depicted in the following diagram. Typically, a data warehouse-centric architecture includes the following:

- A data warehouse (and optionally multiple subject-focused data marts)
- Data warehouse integrations with various data sources, across business domains
- Data warehouse integrations with end user analytics tools and systems consuming the data stored in the warehouse

The following diagram shows how an enterprise data warehouse fits into an analytics architecture:

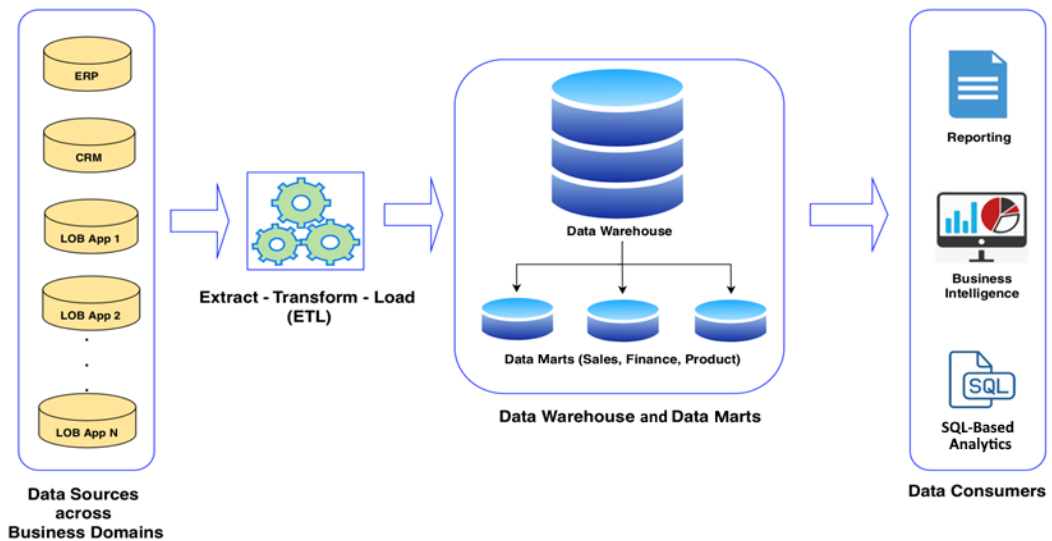


Figure 2.1 – Enterprise data warehousing architecture

At the center of our architecture is the enterprise data warehouse, which hosts a set of *data assets* that contain current and historical data about key business subject areas. On the left-hand side, we have our source systems and an ETL pipeline to load the data into the warehouse. On the right-hand side, we can see several systems/applications that consume data from the data warehouse.

In the next couple of sections, we'll look at how modern cloud-native data warehouses, such as Amazon Redshift, leverage parallel processing and columnar storage to store and process petabytes of data. Amazon Redshift provides very high query throughput while processing high data volumes.

Distributed storage and massively parallel processing

In the following diagram, we can see the underlying architecture of an Amazon Redshift cluster:

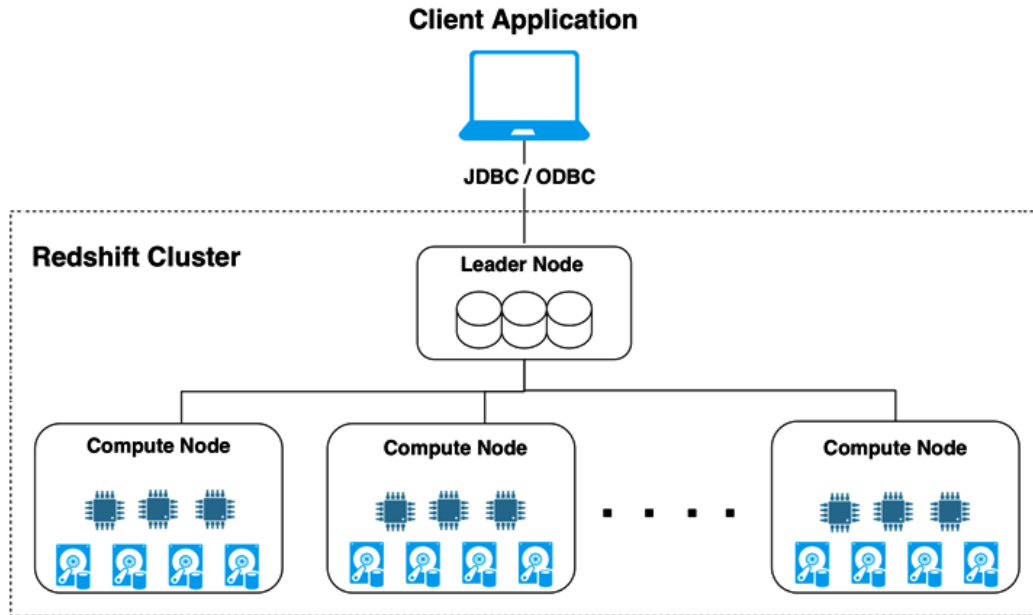


Figure 2.2 – MPP architecture of an Amazon Redshift cluster

As we can see, an Amazon Redshift cluster contains several compute resources, along with their associated disk storage. There are two types of nodes in a Redshift cluster:

- One leader node, which interfaces with client applications, receives and parses queries, and coordinates query execution on compute nodes
- Multiple compute nodes, which store warehouse data and run query execution steps in parallel.

Each compute node has its own independent processors, memory, and storage volumes that are isolated from other compute nodes in the cluster (this is called a **shared-nothing architecture**). Data is stored in a distributed fashion across compute nodes. The cluster can easily be scaled up to store and process petabytes of data by simply adding more compute nodes to the cluster (**horizontal scaling**).

Cloud data warehouses implement a distributed query processing architecture called **Massively Parallel Processing (MPP)** to accelerate queries on massive volumes of data. In this approach, the cluster leader node compiles the incoming client query into a distributed execution plan. It then coordinates the execution of segments of compiled query code on multiple compute nodes of the data warehouse cluster, in parallel. Each compute node executes assigned query segments on its respective portion (stored locally on the node) of the distributed dataset. To optimize MPP throughput, datasets may be evenly distributed across the nodes to ensure participation of the maximum number of cluster nodes in parallel query processing. To accelerate distributed MPP joins, most commonly joined datasets are distributed across cluster nodes by common join keys, so that matching slices of tables being joined end up on the same compute nodes.

Columnar data storage and efficient data compression

In addition to providing massive storage and cluster computing, modern data warehouses also boost query performance through **column-oriented** storage and **data compression**. In this section, we'll examine how this works, but first, let's understand how traditional **Online Transaction Processing (OLTP)** databases store their data.

OLTP applications typically work with entire rows that include all columns of the table (for example, read/write a sales record or look up a catalog record). To serve OLTP applications, backend databases need to efficiently read and write full rows to the disk. To speed up full row lookups and updates, OLTP databases use a row-oriented layout to store table rows on the disk. In a **row-oriented** physical data layout, all the column values of a given row are co-located, as depicted in the following diagram:

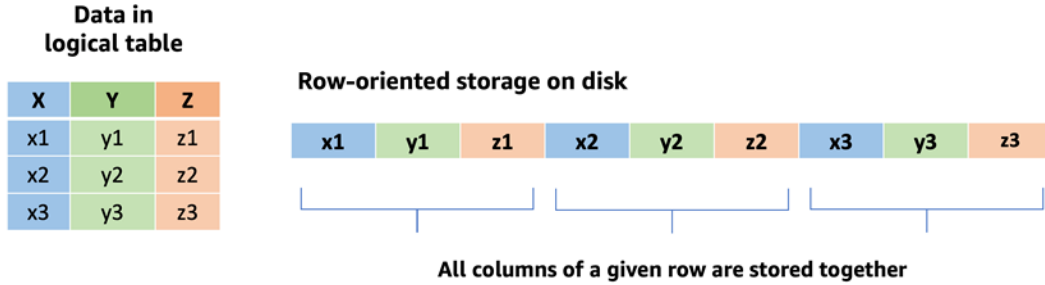


Figure 2.3 – Row-oriented storage layout

Most analytics queries that business users run against a data warehouse are written to answer a specific question and typically include grouping and the aggregations (such as sum, average, mean) of a narrow set of columns from fact and dimension tables (these typically contain many more columns than the narrow set of columns included in the query). Analytics queries typically need to scan through a large number of rows but need data from only a narrow set of columns that the query cares about. A row-oriented physical data layout forces analytics queries to scan a large number of full rows (all columns), even though they need only a subset of the columns from these rows. Analytics queries on a row-oriented database can thus require a much higher number of disk I/O operations than necessary.

Modern data warehouses store data on disks using a *column-oriented* physical layout. This is more suitable for analytical query processing, which only requires a subset of columns per query. While storing a table's data in a column-oriented physical layout, a data warehouse breaks a table into groups of rows, called row chunks/groups. It then takes a row chunk at a time and lays out data from that row chunk, one column at a time, so that all the values for a column (that is, for that row chunk) are physically co-located on the disk, as depicted in the following diagram:

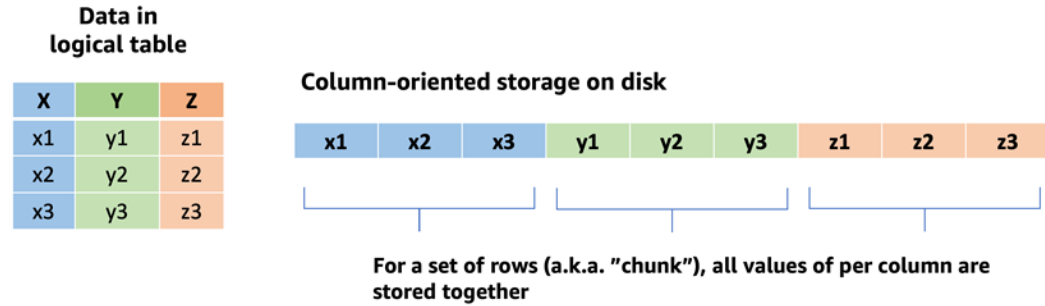


Figure 2.4 – Column-oriented storage layout

Data warehouses repeat this for all the row chunks of the table. In addition to storing tables as row chunks and using a column-oriented physical layout on disks, data warehouses also maintain in-memory maps of locations of these chunks. Modern data warehouses use these in-memory maps to pin-point column locations on the disk and read the physically co-located values of the column. This enables the query engine to retrieve data for only the narrow set of columns needed for a given analytics query. By doing this, disk I/O is significantly reduced compared to what would be required to run the same query on a row-oriented database.

Most modern data warehouses compress the data before storing it on the disk. In addition to saving storage space, compressed data requires much lower disk I/O to read and write data to the disk. Compression algorithms provide much better compression ratios when all the values being compressed have the same data type and have a larger percentage of duplicates. Since column-oriented databases lay out values of the same column (hence the same data type, such as strings or integers) together, data warehouses achieve good compression ratios, resulting in faster read/writes and smaller on-disk footprints.

Dimensional modeling in data warehouses

Data assets in the warehouse are typically stored as relational tables that are organized into widely used dimensional models, such as a star or snowflake schema. Storing data in a warehouse using a dimensional model makes it easier to retrieve and filter relevant data, and it also makes analytic query processing flexible, simple, and performant.

Let's dive deeper into two widely used data warehouse modeling techniques and see how we can organize sales domain entities as an example. Note that this example is just a subsection of a much larger data warehouse schema.

The following diagram illustrates how data in a sales subject area can be organized using a star schema:

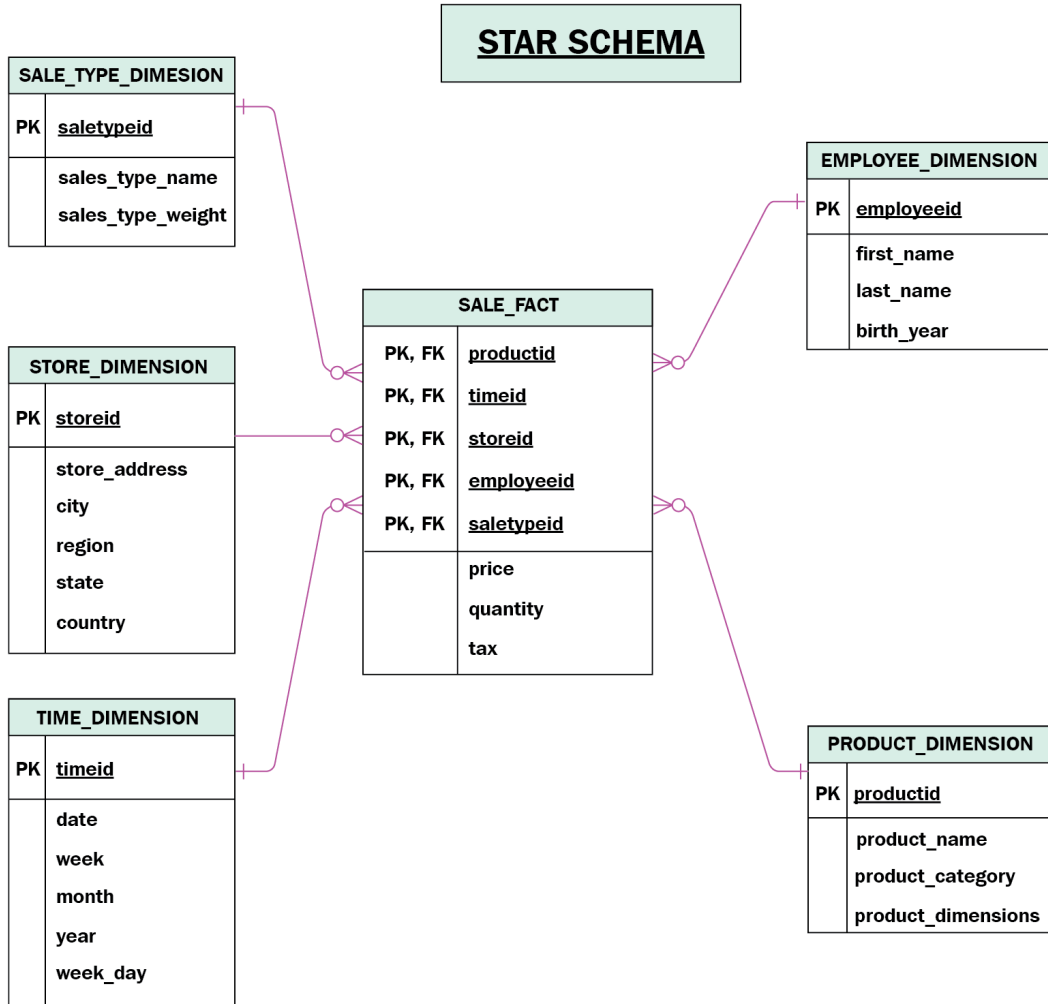


Figure 2.5 – Sales data entities organized as a star schema

In the preceding diagram, the data entities are organized like a star, with the `Sales fact` table forming the middle of the star and the `dimension` tables forming the corners. A `fact` table stores the granular numeric measurements/metrics (such as price or quantity, in our example) of a business subject area such as `Sales`. The `dimension` tables, surrounding the `fact` tables, store the context under which fact measurements were captured. Each dimension table essentially provides the attributes of the contextual entity such as who (employee, customer), what (product), where (store, city, state), and when (sales date/time, quarter, year, weekday).

In a **star schema**, `fact` tables contain `foreign key` columns to store pointers to the related rows in the `dimension` tables. Dimensional attributes are key to finding and aggregating measurements stored in the `fact` tables in a data warehouse. Business analysts typically slice, dice, and aggregate facts from different dimensional perspectives to generate business insights about the subject area represented by the star schema. They can find out answers to questions such as, what is the total volume of a given product sold over a given period? What is the total revenue in a given product category? Which store sells the greatest number of products in a given category?

In a star schema, while data for a subject area is normalized by splitting measurements and context information into separate `fact` and `dimension` tables, individual `dimension` tables are typically kept denormalized so that all the related attributes of a dimensional topic can be found in a single table. This makes it easier to find all the related attributes of a dimensional topic in a single table (fewer joins, simpler to understand model), but for larger `dimension` tables, a denormalized approach can lead to data duplication and inconsistencies within the `dimension` table. Large denormalized `dimension` tables can also be slow to update.

One approach you can follow to work around these issues is a slightly modified type of schema, the **snowflake schema**, as shown in the following diagram:

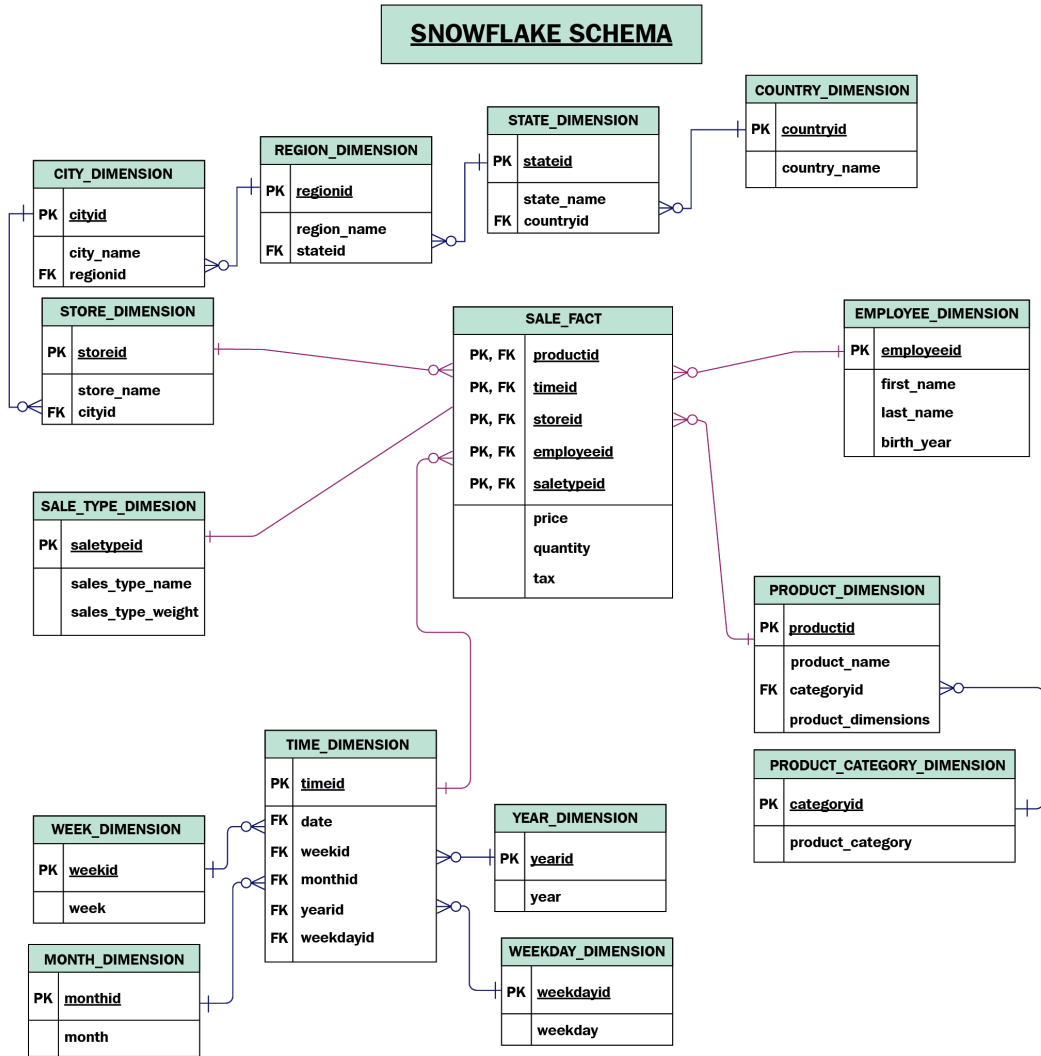


Figure 2.6 – Sales data entities organized as a snowflake schema

The challenges of inconsistencies and duplication in a star schema can be addressed by **snowflaking** (basically normalizing) each dimension table into multiple related dimension tables (normalizing the original product dimension into product and product category dimensions, for example). This continues until each dimension table contains only attributes with direct correlation to the table's primary key. The highly normalized model resulting from this snowflaking is called a snowflake schema.

The snowflake schema can be designed by extending the star schema or can be built from the ground up by ensuring that each dimension is highly normalized and connected to the related dimension tables, thus forming a hierarchy. A snowflake schema can reduce redundancy and minimize disk space compared to a star schema, which often contains duplicate records. However, on the other hand, the snowflake schema may necessitate complex joins to answer business queries and may slow down query performance.

Understanding the role of data marts

Data warehouses contain data from all relevant business domains and have a comprehensive but complex schema. Data warehouses are designed for the cross-domain analysis that's required to inform strategic business decisions. However, organizations often also have a narrower set of users who want to focus on a particular line of business, department, or business subject area. These users prefer to work with a repository that has a simple-to-learn schema, and only the subset of data that focuses on the area they are interested in. Organizations typically build data marts to serve these users.

A data mart is focused on a single business subject repository (for example, marketing, sales, or finance) and is typically created to serve a narrower group of business users, such as a single department. A data mart often has a set of denormalized fact tables organized in a much simpler schema compared to that of an enterprise data warehouse. Simpler schemas and a reduced amount of data volume make data marts faster to build, simpler to understand, and easier to use for end users. A data mart can be created in one of the following formats:

- Top down: Data is taken from an existing data warehouse and focuses on a slice of business subject data.
- Bottom up: Data is sourced directly from run-the-business applications related to a business domain of interest.

Both data warehouses and data marts provide an integrated view of data from multiple sources, but they differ in the scope of data they store. Data warehouses provide a central store of data for the entire business and cover all business domains. Data marts serve a specific division, or business function, by providing an integrated view of a subject area relevant to that business function.

So far, we have discussed various aspects of data warehouses and data marts, including the central data repositories of our *Enterprise data warehousing architecture* from *Figure 2.1*. Now, let's look at the components in our architecture that feed data to central data repositories.

Feeding data into the warehouse – ETL and ELT pipelines

To bring data into the warehouse (and optionally, data marts), organizations typically build data pipelines that do the following:

- Extract data from source systems.
- Transform source data by validating, cleaning, standardizing, and curating it.
- Load the transformed source data into the enterprise data warehouse schema, and optionally a data mart as well.

In these pipelines, the first step is to extract data from source systems, but the next two steps can either take on a **Transform-Load** or **Load-Transform** sequence.

The data warehouses of a modern organization typically need to be fed data from a diverse set of sources, such as ERP and CRM application databases, files stored on **Network Attached Storage (NAS)** arrays, **SaaS** applications, and external partner applications. The components that are used to implement the **Extract** step of both ETL and ELT pipelines typically need to connect to these sources and handle diverse data formats (including relational tables, flat files, and continuous streams of records).

The decision as to whether to build an **Extract-Transform-Load (ETL)** or **Extract-Load-Transform (ELT)** data pipeline is based on the following:

- The complexity of the required data transformations.
- The speed at which source data needs to be made available for analysis in the data warehouse after it's produced in the source system.

The following diagram shows a typical ETL pipeline for loading data into a data warehouse:

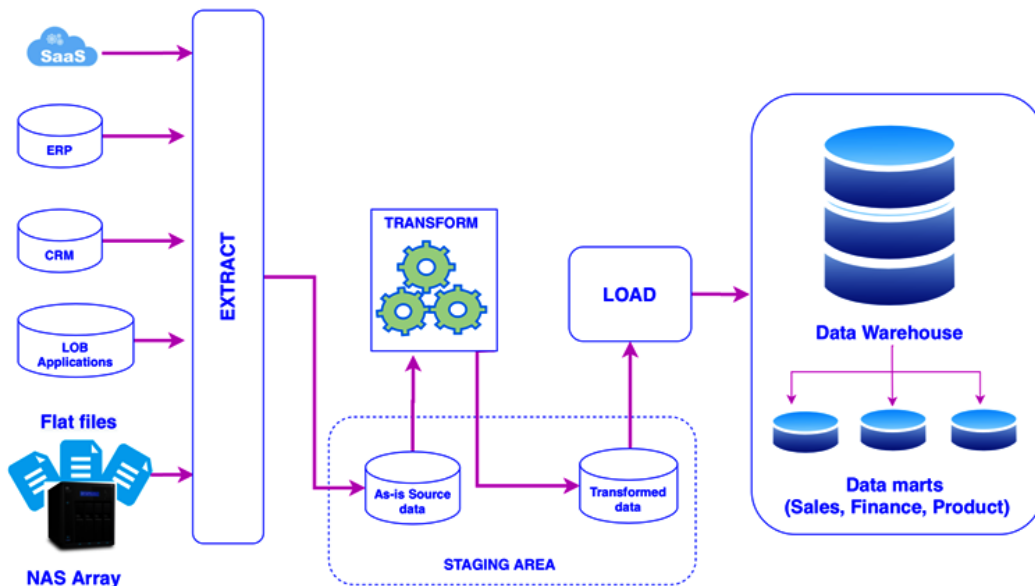


Figure 2.7 – ETL pipeline

An **ETL** pipeline extracts data from various sources and stores it in a staging area first (a system outside the warehouse). Transformation operations are then performed on staged data to validate it, clean it, standardize it, structure it, and convert it into a form suitable so that it can be loaded into the data warehouse (and optionally, data marts). The transformed data from the staging area is then loaded into the warehouse's dimensional schema. An ETL approach to building a data pipeline is typically used when the following are true:

- Source database technologies and formats are different from those of the data warehouse
- Data volumes are small to moderate
- Data transformations are complex and compute-intensive

In an ETL pipeline, transformations are performed outside the data warehouse using custom scripts, a cloud-native ETL service such as **AWS Glue**, or a specialized ETL tool from a commercial vendor such as **Informatica**, **Talend**, **DataStage**, **Microsoft**, or **Pentaho**.

On the other hand, an **ELT** pipeline extracts data (typically, highly structured data) from various sources and loads it *as-is* (matching the sources systems' data structures) into a staging area within the data warehouse. The database engine powering the data warehouse is then used to perform transformation operations on the staged data to make it ready for consumption.

The following diagram shows a typical ELT pipeline:

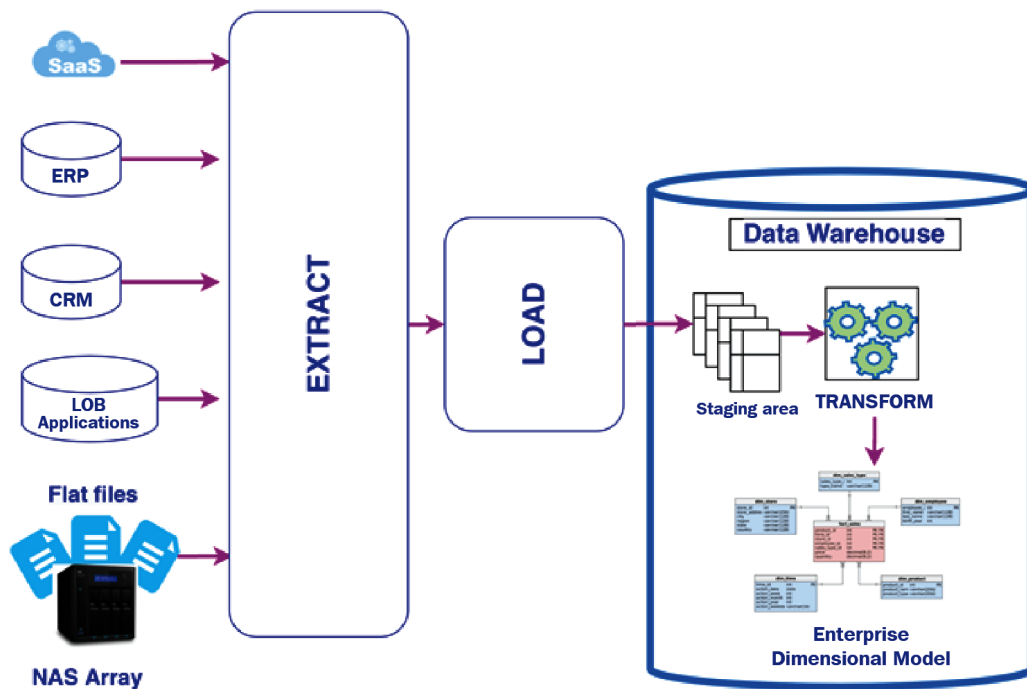


Figure 2.8 – ELT pipeline

The ELT approach allows for rapidly loading large amounts of source data into the warehouse. Furthermore, the MPP architecture of modern data warehouses can significantly accelerate the transform steps in ELT pipelines. The ELT approach is typically leveraged when the following are true:

- Data sources and the warehouse have similar database technologies, making it easier to directly load source data into the staging tables in the warehouse.
- A large volume of data needs to be quickly loaded into the warehouse.
- All the required transformation steps can be executed using the native SQL capabilities of the warehouse's database engine.

In this section, we learned how data warehouses can store and process petabytes of structured data. Modern data warehouses provide high-performance processing using compute parallelism, columnar physical data layout, and dimensional data models such as star or snowflake schemas.

The data management architectures in modern organizations need capabilities that can store and analyze exploding volumes of semi-structured and unstructured data, in addition to handling structured data from traditional sources. In the next section, we'll learn about a new architecture, called data lakes, that today's leading organizations typically implement to store, process, and analyze structured, semi-structured, and unstructured data.

Building data lakes to tame the variety and volume of big data

Along with the rise of new data types and increasing data volumes, we have seen an increase in the ways that organizations look to draw insights from data. Machine learning in particular has become a popular tool for analytics, enabling organizations to automatically extract metadata from unstructured data sources, which can then be analyzed with traditional analytic tools:

- Creating automated transcripts of call center audio recordings
- Using **natural language processing (NLP)** algorithms to extract sentiment data from text
- Identifying objects, people, and expressions in an image

As we saw in the previous section, enterprise data warehouses have been the go-to repositories for storing highly structured tabular data sourced from traditional run-the-business transactional applications. But the lack of a well-defined tabular structure makes unstructured and semi-structured data unsuitable for storing in typical data warehouses. Also, while they are good for use cases that need SQL-based processing, data warehouses are limited to processing data using only SQL, and SQL is not the right tool for all data processing requirements. For example, extracting metadata from unstructured data, such as audio files or images, is best suited for specialized machine learning tools.

Most traditional data warehouses also tightly couple compute and storage. If you need additional compute power, you need to add a processing node that also includes storage. If you need additional storage, you add a processing node that also includes additional compute power. With these systems, you always add compute and storage together, even if you only need one of those additional resources.

On the other hand, a cloud data lake is a central, highly scalable repository in the cloud where an organization can manage exabytes of various types of data, such as the following:

- Structured data (row-column-based tables)
- Semi-structured data (such as JSON and XML files, log records, and sensor data streams)
- Unstructured data (such as audio, video streams, Word/PDF documents, and emails)

Data from any of these sources can be quickly loaded into the data lake as-is (keeping the original source format and structure). Unlike with data warehouses, data does not need to be converted into a standard structure.

A cloud data lake also natively integrates with cloud analytic services that are decoupled from data lake storage and enables diverse analytic tools, including SQL, code-based tools (such as Apache Spark), specialized machine learning tools, and business intelligence visualization tools.

In the next section, we will dive deeper into the architecture of a typical data lake.

Data lake logical architecture

Let's take a closer look at the architecture of a cloud-native data lake by looking at its logical architecture, as shown in the following diagram:

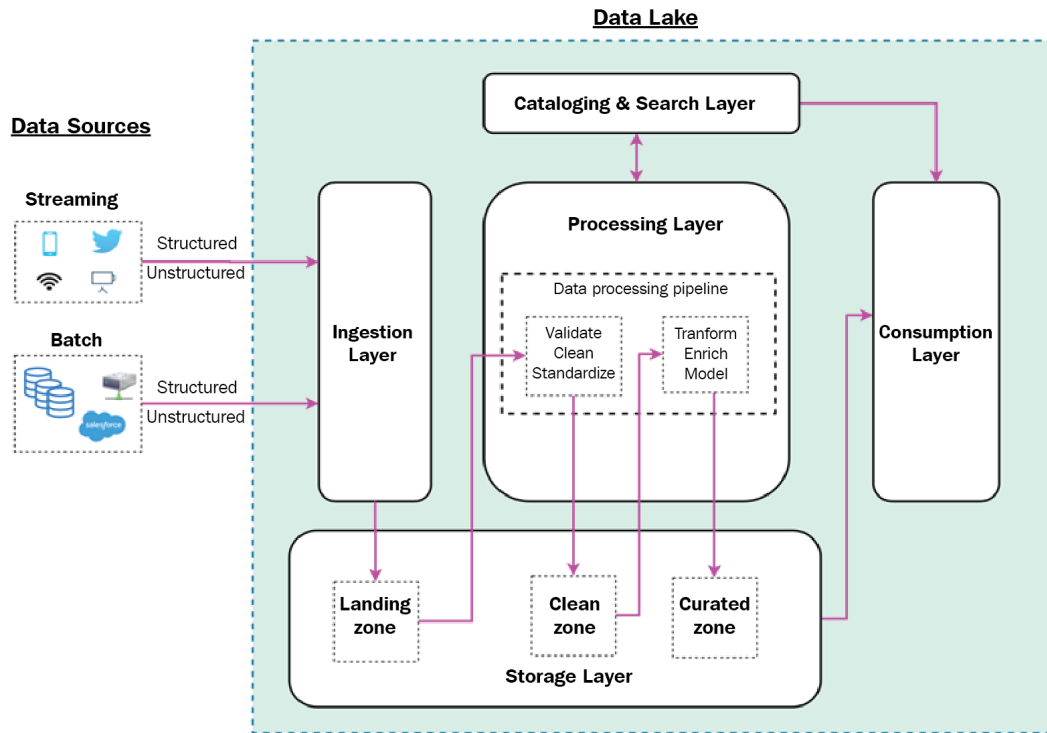


Figure 2.9 – Data lake logical layered architecture

We can visualize a data lake architecture as a set of independent components organized into five logical layers. A layered, component-oriented data lake architecture can evolve to incorporate innovations in data management and analytics methods, as well as to make use of new tools. This keeps the data lake responsive to new data sources and changing requirements. In the following sections, we will dive deeper into these layers.

The storage layer and storage zones

At the center of the data lake architecture is the **storage layer**, which provides virtually unlimited, low-cost storage that can store a variety of datasets, irrespective of their structure or format.

The storage layer is organized into different zones, with each zone having a specific purpose. Data moves through the various zones of the data lake, with new, modified copies of the data in each zone as the data goes through various transformations. There are no hard rules about how many zones there should be, or the names of zones, but the following zones are commonly found in a typical data lake:

- **Landing/raw zone:** This is the zone where the ingestion layer writes data, as-is, from the source systems. The landing/raw zone permanently stores the raw data from source.
- **Clean/transform zone:** The initial data processing of data in the landing/raw zone, such as validating, cleaning, and optimizing datasets, writes data into the clean/transform zone. The data here is often stored in optimized formats such as Parquet, and it is often partitioned to accelerate query execution and downstream processing. Data in this zone may also have had PII information removed, masked, or replaced with tokens.
- **Curated/enriched zone:** The data in the clean/transformed zone may be further refined and enriched with business-specific logic and transformations, and this data is written to the curated/enriched zone. This data is in its most consumable state and meets all organization standards (cleanliness, file formats, schema). Data here is typically partitioned, cataloged, and optimized for the consumption layer.

Depending on the business requirements, some data lakes may include more or fewer zones than the three highlighted here. For example, a very simple data lake may just have two zones (the raw and curated zones), while some data lakes may have five or more zones to handle intermediate stages or specific requirements.

Cataloging and search layer

A data lake typically hosts a large number of datasets (potentially thousands) from a variety of internal and external sources. These datasets are used by several users across the organization, and these users need the ability to search for available datasets and review the schema and other metadata of those datasets. The **cataloging and search layer** provides this metadata (schema, partitioning information, categorization, ownership, and more) about the datasets hosted in the storage layer. The cataloging layer can also track changes that have been made to the schemas of the datasets in the lake. This layer should also provide a search capability to simplify the task of finding a required dataset among the many datasets held in the lake.

Ingestion layer

The **ingestion layer** is responsible for connecting to diverse types of data sources and bringing their data into the storage layer of the data lake. This layer contains several independent components, each purpose-built to connect to a data source with a distinct profile in terms of the following:

- Data structure (structured, semi-structured, unstructured)
- Data delivery type (table rows, data stream, data file)
- Data production cadence (batch, streaming)

This component-oriented composition of the ingestion layer provides the flexibility to simply add new components to match a new data source's distinct profile.

A typical ingestion layer may contain several components (tools) for connecting to and ingesting from the various data sources. Examples include **Amazon Database Migration Services (DMS)** for ingesting from various databases, and **Amazon Kinesis Firehose** to ingest streaming data into the data lake. An overview of these tools, and many others, will be covered in *Chapter 3, The AWS Data Engineers Toolkit*, and we will dive deep into the ingestion layer in *Chapter 6, Ingesting Batch and Streaming Data*.

Processing layer

Once the ingestion layer brings data from a source system into the landing zone, it is the **processing layer** that makes it ready for consumption by data consumers. The processing layer transforms the data in the lake through various stages of data cleanup, standardization, and enrichment. Along the way, the processing layer stores transformed data in the different zones – writing it into the clean zone and then the curated zone, and then ensuring that the data catalog gets updated.

The components in the ingestion and processing layers are used to create ELT pipelines. In these pipelines, the ingestion layer components extract data from the source and load it into the data lake, and then the processing layer components transform it to make it suitable for consumption by components in the consumption layer.

Consumption layer

Once data has been ingested and processed to make it consumption-ready, it can be analyzed using several techniques, such as interactive query processing, business intelligence dashboarding, and machine learning. To perform analytics on data in the lake, the **consumption layer** provides purpose-built tools. The tools in the consumption layer can natively access data from the storage layer, and the schema can be accessed from the catalog layer (to apply schema-on-read to the lake-hosted data).

Data lake architecture summary

In this section, we learned about data lake architectures, and how they can enable organizations to manage and analyze vast amounts of structured, unstructured, and semi-structured data.

Analytics platforms at a typical organization need to serve warehousing style structured data analytics use cases (such as complex queries and BI dashboarding), as well as use cases that require managing and analyzing vast amounts of unstructured data. Organizations typically end up building both a data warehouse and a data lake.

In the next section, we will look at an emerging data management and analytics architecture that's often referred to as the lake house architecture, or the data lakehouse. This architecture natively integrates data warehouses and data lakes and unlocks the best of both worlds.

Bringing together the best of both worlds with the lake house architecture

In today's highly digitized world, data about customers, products, operations, and the supply chain can come from many sources and can have a diverse set of structures. To gain deeper and more complete data-driven insights about a business topic (such as the customer journey, customer retention, product performance, and more), organizations need to analyze all the relevant topic data of all the structures from all the sources, together.

Organizations collect and analyze structured data in data warehouses, and they build data lakes to manage and analyze unstructured data. Historically, organizations have built data warehouse and data lake solutions in isolation from each other, with each having its own separate data ingestion, storage, processing, and governance layers. Often, these disjointed efforts to build separate data warehouse and data lake ecosystems have ended up creating data and processing silos, data integration complexity, excessive data movement, and data consistency issues. These, in turn, have led to delays and increased costs in gaining deeper insights that only come when you combine and analyze all the relevant data.

In this section, we will look at a recently emerging cloud-native architecture, called *lake house architecture*. This new architecture approach enables organizations to collect, manage, process, and analyze all of their structured and unstructured data in a simple and integrated fashion. The data lakehouse architecture combines the best features of both the data warehouse and data lake worlds, and also provides a unified interface to enable access to all topic relevant data, of all structures, in an integrated way.

Data lakehouse implementations

Over the last 2 to 3 years, various cloud providers, software providers, and open source organizations have been building new products to enable this move toward a data lakehouse architecture. The implementation approaches to a data lakehouse (also sometimes referred to as data lakehouse) vary across different platform providers:

- **Databricks** has introduced an offering called Databricks Delta Lake. Delta Lake provides a storage layer that enables ACID transactions directly in the data lake. With this functionality, records can be inserted, updated, and deleted for tables in the data lake, something that was previously not easily available.
- **Apache Hudi** is a relatively new open source project that enables users to perform insert, update, and delete operations on data in the data lake, without needing to build their own custom solutions.
- **Microsoft Azure** has added a capability called Polybase to their warehouse service, known as Azure Synapse Analytics. Polybase allows Azure Synapse Analytics users to include data stored in Azure Blob storage, Azure Data Lake Store, and Hadoop to help process their T-SQL queries.
- **Amazon Web Services (AWS)** has added several new capabilities, including new features in Redshift Spectrum and Lake Formation, to enable building a data lakehouse architecture on AWS. This includes the ability to read data lake tables in S3 from Amazon Redshift, as well as to perform inserts, updates, and deletes on data lake tables using Lake Formation governed tables.

In the rest of this section, we'll look at an implementation architecture of a data lakehouse on AWS.

Building a data lakehouse on AWS

Two components that provide the key foundations for building a lake house architecture in AWS are Amazon Redshift Spectrum and AWS Lake Formation. The following diagram shows a data lake architecture with these two components:

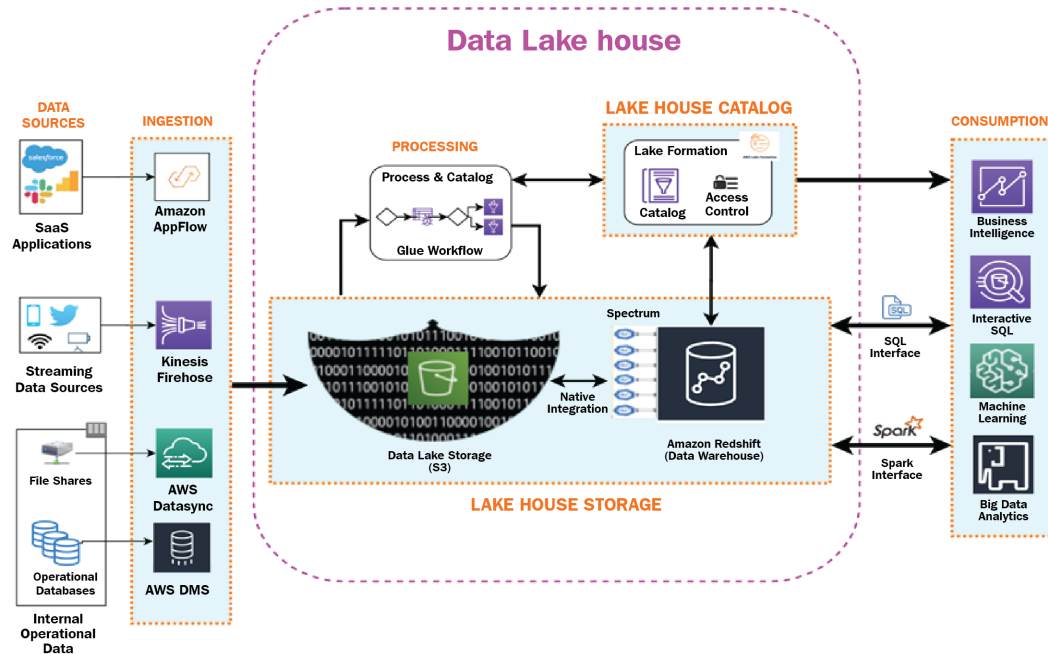


Figure 2.10 – Data lakehouse on AWS

Redshift Spectrum is a feature of the Amazon Redshift data warehouse service that enables Redshift to read data stored in S3. Redshift Spectrum is essentially a query processing layer that uses Amazon-managed compute nodes to natively query structured and semi-structured data hosted in data lake storage (S3). Spectrum enables an Amazon Redshift data warehouse to present a single unified interface, where users can run SQL statements that combine data from both Redshift (data warehouse) and S3 (data lake). Spectrum thus enables users to query all the data in the lake house using a single SQL interface.

Lake Formation provides the central lake house catalog where users and analytics services can search, discover, and retrieve metadata for a dataset. The central lake house catalog is automatically kept up to date with metadata from all the lake house datasets by using the catalog automation capability provided by AWS Glue. Glue can be configured to periodically *crawl* through the lake house storage and discover datasets, extract their metadata (such as location, schema, format, partition information, and more), and then store this metadata as a table in the central Lake Formation catalog. The metadata of a table in the catalog can be used by AWS analytics services, such as **Amazon Athena**, to locate a corresponding dataset in the lake house storage and apply schema-on-read to that dataset during query execution.

In addition to adding Redshift Spectrum and Lake Formation capabilities, AWS has also enabled various cloud services in the processing and consumption layers to be able to access all lake house data using either Redshift SQL or an Apache Spark interface. For example, AWS Glue (which provides a serverless Apache Spark environment) and Amazon EMR (a managed Spark environment) include native Spark plugins that can access Redshift tables, in addition to objects in S3 buckets, all in the same job. Amazon Athena supports query federation, which enables Athena to query data in the data lake, as well as data stored in other engines such as Amazon Redshift or an Amazon RDS database.

AWS has also enhanced the AWS Lake Formation service to support governed tables. With governed tables, users can run transactional queries against data stored in the table, including inserts, updates, and deletes. In addition to this, with governed tables, users can *time travel*, which means they can query a table and specify a specific time, and the results that are returned will represent the data as it was at the specified point in time.

In future chapters, the hands-on exercises will cover various tasks related to ingesting, transforming, and querying data in the data lake, but in this chapter, we are still setting up some of the foundational tasks. In the next section, we will work through the process of installing and configuring the AWS **Command Line Interface (CLI)**, and we will create an Amazon S3 bucket.

Hands-on – configuring the AWS Command Line Interface tool and creating an S3 bucket

In *Chapter 1, An Introduction to Data Engineering*, you created an AWS account and an AWS administrative user, and then ensured you could access your account. As part of the process of creating the administrative user, you took note of the Access Key ID and Secret Access Key, both of which are needed for authenticating programmatic access to your account.

In this chapter, we will use those keys to configure the AWS CLI. We will also use the CLI to create an Amazon S3 bucket (a storage container in the Amazon S3 service).

Installing and configuring the AWS CLI

To configure the AWS CLI, you will need an Access Key ID and Secret Access Key for an IAM administrative user.

The following steps will install the AWS CLI and configure it for use in the hands-on sections throughout the remainder of this book:

1. Download the appropriate AWS CLI installer for your platform (Mac, Windows, or Linux) from <https://aws.amazon.com/cli/>.
2. Run the installer to complete the installation of the AWS CLI.
3. To configure the CLI, run `aws configure` at the Command Prompt and provide the AWS Access Key ID and AWS Secret Access Key for your IAM Administrative user. Also, provide a default region – in the examples in this book, we will use `us-east-2` (Ohio), but you can use a different region if it supports all the services and features covered in this book. For `Default output format`, press *Enter* to leave it as the default, as shown in the following command block:

```
$ aws configure
AWS Access Key ID [None]: AKIAX9LFIEPF3KKQUI
AWS Secret Access Key [None]:
neKLcXPXlabP9C90a0qeBkWZAbnbM4ihesP9N1u3
Default region name [None]: us-east-2
Default output format [None]: ENTER
```

Creating a new profile

If you already have the AWS CLI configured and associated with a different IAM user account, you have the option of configuring multiple profiles, each one associated with a different IAM user. To do this, run the `configure` command with the `profile` argument, specifying a name for the profile. For example, you could run `aws configure --profile dataengbook`, and then provide the details for the IAM administrative user we created in *Chapter 1, An Introduction to Data Engineering*. Then, when running through the tutorials in this book, make sure you always specify the profile created here. For example, to list the S3 buckets in your account using the `dataengbook` profile you just created, you would run `aws s3 ls --profile dataengbook`.

Thus, we have installed and configured the AWS CLI. Next, we will see how to create a new Amazon S3 bucket.

Creating a new Amazon S3 bucket

To confirm that we have configured the CLI correctly, we will create a new S3 bucket using the AWS CLI.

Amazon **Simple Storage Service (S3)** is an object storage service that offers near unlimited capacity with high levels of durability and availability. To store data in S3, you need to create a bucket. Once created, the bucket can store any number of objects.

Each S3 bucket needs to have a globally unique name, and it is recommended that the name be DNS compliant. For more information on rules for bucket names, see <https://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html>.

To create an S3 bucket using the AWS CLI, run the following command at the Command Prompt:

```
$ aws s3 mb s3://<bucket-name>
```

Remember that the bucket name you specify here must be globally unique. If you attempt to create a bucket using a name that another AWS account has used, you will see an error similar to the following:

```
$ aws s3 mb s3://test-bucket
```

```
make_bucket failed: s3://test-bucket An error occurred  
(BucketAlreadyExists) when calling the CreateBucket operation:  
The requested bucket name is not available. The bucket  
namespace is shared by all users of the system. Please select a  
different name and try again.
```

If your `aws s3 mb` command returned a message similar to the following, then congratulations! Your AWS CLI has been successfully configured:

```
make_bucket: <bucket-name>
```

Summary

In this chapter, we learned about the foundational architectural concepts that are typically applied when designing real-life analytics data management and processing solutions. We also discussed three analytics data management architectures that you would find most commonly used across organizations today: data warehouse, data lake, and data lakehouse.

In the next chapter, we will provide an overview of several AWS services that are used in the creation of these architectures – from services for ingesting data, to services that help perform data transformation, to services that are designed for querying and analyzing data.

3

The AWS Data Engineer's Toolkit

Back in 2006, Amazon launched **Amazon Web Services (AWS)** to offer on-demand delivery of IT resources over the internet, essentially creating the cloud computing industry. Ever since then, AWS has been innovating at an incredible pace, continually launching new services and features to offer broad and deep functionality across a wide range of IT services.

Traditionally organizations built their own big data processing systems in their data centers, implementing commercial or open source solutions designed to help them make sense of ever-increasing quantities of data. However, these systems were often complex to install, requiring a team of people to maintain, optimize, and update, and scaling these systems was a challenge, requiring large infrastructure spend and significant delays while waiting for hardware vendors to install new compute and storage systems.

Cloud computing has enabled the removal of many of these challenges, including the ability to launch fully configured software solutions at the push of a button and having these systems automatically updated and maintained by the cloud vendor. Organizations also benefit from the ability to scale out by adding resources in minutes, all the while only paying for what was used, rather than having to make large upfront capital investments.

Today, AWS offers around 200 different services, including a number of analytics services that can be used by data engineers to build complex data analytic pipelines. There are often multiple AWS services that could be used to achieve a specific outcome, and the challenge for data architects and engineers is to balance the pros and cons of a specific service, evaluating it from multiple perspectives, before determining the best fit for the specific requirements.

In this chapter, we introduce a number of these AWS managed services commonly used for building big data solutions on AWS, and in later chapters, we will look at how you can architect complex data engineering pipelines using these services. As you go through this chapter, you will learn about the following topics:

- AWS services for ingesting data
- AWS services for transforming data
- AWS services for orchestrating big data pipelines
- AWS services for consuming data
- Hands-on - an AWS Lambda function when a new file arrives in an S3 bucket

Technical requirements

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/tree/main/Chapter03>

AWS services for ingesting data

The first step in building big data analytic solutions is to **ingest data** from a variety of sources into AWS. In this section, we introduce some of the core AWS services designed to help with this; however, this should not be considered a comprehensive review of every possible way to ingest data into AWS.

Don't feel overwhelmed by the number of services we cover in this section! We will explore approaches for deciding on the right service for your specific use case in later chapters, but it is important to have a good understanding of the available tools upfront.

Overview of Amazon Database Migration Service (DMS)

One of the most common ingestion use cases is to sync data from a traditional database system into an analytic pipeline, either landing the data in an Amazon S3-based data lake, or in a data warehousing system such as **Amazon Redshift**.

Amazon DMS is a versatile tool that can be used to migrate existing database systems to a new database engine, such as migrating an existing **Oracle** database to an **Amazon Aurora with PostgreSQL compatibility** database. But from an analytics perspective, Amazon DMS can also be used to run continuous replication from a number of common database engines into an **Amazon S3 data lake**.

As discussed previously, data lakes are often used as a means of bringing in data from multiple different data sources into a centralized location to enable an organization to get the big picture across different business units and functions. As a result, there is often a requirement to perform continuous replication of a number of production databases into Amazon S3.

For our use case, we want to *sync* our production customer, products, and order databases into the data lake. Using DMS, we can do an initial load of data from the databases into S3, specifying the format that we want the file written out in (CSV or Parquet), and the specific ingestion location in S3.

At the same time, we can also set up a DMS task to do ongoing replication from the source databases into S3 once the full load completes. With transactional databases, the rows in a table are regularly updated, such as if a customer changes their address or telephone number. When querying the database using SQL, we can see the updated information, but in most cases, there is no practical method to track changes to the database using only SQL. Because of this, DMS uses the database transaction log files from the database to track updates to rows in the database and writes out the target file in S3 with an extra column added (Op) that indicates which operation is reflected in the row – an insert, update, or deletion. The process of tracking and recording these changes is commonly referred to as **Change Data Capture (CDC)**.

Picture a situation where you have a source table with a schema of `cust id`, `lastname`, `firstname`, `address`, and `phone`, and the following sequence of events happens:

- A new customer is added with all fields completed.
- The phone number was entered incorrectly, so the record has the phone number updated.
- The customer record is then deleted from the database.

We would see the following in the CDC file that was written out by DMS:

```
I, 9335, Smith, John, "1 Skyline Drive, NY, NY", 201-555-9012
U, 9335, Smith, John, "1 Skyline Drive, NY, NY", 201-555-9034
D, 9335, Smith, John, "1 Skyline Drive, NY, NY", 201-555-9034
```

The first row in the file shows us that a new record was *inserted* into the table (represented by the I in the first column). The second row shows us that a record was *updated* (represented by the U in the first column). And finally, the third entry in the file indicates that this record was *deleted* from the table (represented by the D in the first column).

We would then have a separate update process that would run to read the updates and apply those updates to the full load, creating a new point-in-time snapshot of our source database. The update process would be scheduled to run regularly, and each time it runs it would apply the latest updates as recorded by DMS to the previous snapshot, creating a new point-in-time snapshot. We will review this kind of update job and approach in more detail in *Chapter 7, Transforming Data to Optimize for Analytics*.

When to use: Amazon DMS simplifies migrating from one database engine to a different database engine, or syncing data from an existing database to Amazon S3 on an ongoing basis.

When not to use: If you're looking to sync an on-premises database to the same engine in AWS, it is often better to use native tools from that database engine. DMS is primarily designed for heterogeneous migrations (that is, from one database engine to a different database engine).

Overview of Amazon Kinesis for streaming data ingestion

Amazon Kinesis is a managed service that simplifies the process of ingesting and processing streaming data in real time, or near real time. There are a number of different use cases that Kinesis can be used for, including ingestion of streaming data (such as log files, website clickstreams, or IoT data), as well as video and audio streams.

Depending on the specific use case, there are a number of different services that you can select from that form part of the overall Kinesis service. Before we go into more detail about these services, review the following summary of the various Amazon Kinesis services:

- **Kinesis Data Firehose:** Ingests streaming data, buffers for a configurable period, then writes out to a limited set of targets (**S3**, **Redshift**, **Elasticsearch**, **Splunk**, and others)
- **Kinesis Data Streams:** Ingests real-time data streams, processing the incoming data with a custom application and low latency

- **Kinesis Data Analytics:** Reads data from a streaming source and uses SQL statements or **Apache Flink** code to perform analytics on the stream
- **Kinesis Video Streams:** Processes streaming video or audio streams, as well as other time-serialized data such as thermal imagery and RADAR data

Amazon Kinesis Agent

In addition to the AWS Kinesis services, AWS also provides an agent to easily consume data from a file and write that data out in a stream to either Kinesis Data Streams or Kinesis Data Firehose.

The **Amazon Kinesis Agent** is available on GitHub as a Java application under the Amazon Software License (<https://github.com/aws-labs/amazon-kinesis-agent>), as well as in a version for Windows (Amazon Kinesis Agent for Microsoft Windows).

The agent can be configured to monitor a set of files, and as new data is written to the file, the agent buffers the data (configurable for a duration of between 1 second and 15 minutes) and then writes the data to Kinesis. The agent handles retry on failure, as well as file rotation and checkpointing.

An example of a typical use case is a scenario where you want to analyze events happening on your website in near real time. The Kinesis Agent can be configured to monitor the Apache web server log files on your web server, convert each record from the Apache access log format to JSON format, and then write records out reflecting all website activity every 30 seconds to Kinesis, where Kinesis Data Analytics can be used to analyze events and generate custom metrics based on a tumbling 5-minute window.

When to use: The Amazon Kinesis Agent is ideal for when you want to stream data to Kinesis that is being written to a file in a separate process (such as log files).

When not to use: If you have a custom application where you want to emit streaming events (such as a mobile application, or IoT device) you may want to consider using the Amazon **Kinesis Producer Library (KPL)**, or the **AWS SDK**, to integrate sending streaming data directly with your application.

Amazon Kinesis Firehose

Amazon Kinesis Firehose is designed to enable you to easily ingest data in near real time from streaming sources and write out that data to common targets, including Amazon S3, Amazon Redshift, Amazon Elasticsearch, as well as third-party services (such as **Splunk**, **Datadog**, and **New Relic**).

With Kinesis Firehose, you can easily ingest data from streaming sources, process or transform the incoming data, and deliver that data to a target such as Amazon S3 (among others). A common use case for data engineering purposes is to ingest website clickstream data from the Apache web logs on a web server and write that data out to an S3 data lake (or a Redshift data warehouse).

In this example, you could install the Kinesis Agent on the web server and configure it to monitor the Apache web server log files. Based on the configuration of the agent, on a regular schedule the agent would write records from the log files to the Kinesis Firehose endpoint.

The Kinesis Firehose endpoint would buffer the incoming records, and either after a certain time (1-15 minutes) or based on the size of incoming records (1 MB–128 MB) it would write out a file to the specified target. Kinesis Firehose requires you to specify both a size and a time, and whichever is reached first will trigger the writing out of the file.

When writing files to Amazon S3, you also have the option of transforming the incoming data to **Parquet** or **ORC** format, or to perform custom transforms of the incoming data stream using an Amazon Lambda function.

When to use: Amazon Kinesis Firehose is the ideal choice for when you want to receive streaming data, buffer that data for a period, and then write the data to one of the targets supported by Kinesis Firehose (such as Amazon S3, Amazon Redshift, Amazon Elasticsearch, or a supported third-party service).

When not to use: If your use case requires very low latency processing of incoming streaming data (that is, immediate reading of received records), or you want to use a custom application to process your incoming records or deliver records to a service not supported by Amazon Kinesis Firehose, then you may want to consider using **Amazon Kinesis Data Streams** or Amazon **Managed Streaming for Apache Kafka (MSK)** instead.

Amazon Kinesis Data Streams

While Kinesis Firehose buffers incoming data before writing it to one of its supported targets, Kinesis Data Streams provides increased flexibility for how data is consumed and makes the incoming data available to your streaming applications with very low latency (AWS indicates data is available to consuming applications within 70 milliseconds of the data being written to Kinesis).

Companies such as **Netflix** use Kinesis Data Streams to ingest terabytes of log data every day, enriching their networking flow logs by adding in additional metadata, and then writing the data to an open source application for performing near real-time analytics on the health of their network.

You can write to Kinesis Data Streams using the Kinesis Agent, or you can develop your own custom applications using the AWS SDK or the KPL, a library that simplifies writing data records with high throughput to a Kinesis data stream.

The Kinesis Agent is the simplest way to send data to Kinesis Data Streams if your data can be supported by the agent (such as when writing out log files), while the AWS SDK provides the lowest latency, and the Amazon KPL provides the best performance and simplifies tasks such as monitoring and integration with the **Kinesis Client Library (KCL)**.

There are also multiple options available for creating applications to read from your Kinesis data stream, including the following:

- Using other Kinesis services (such as Kinesis Firehose or Kinesis Data Analytics).
- Running custom code using the AWS Lambda service (a serverless environment for running code without provisioning or managing servers).
- Setting up a cluster of Amazon **EC2** servers to process your streams. If using a cluster of Amazon EC2 servers to process your stream, you can use the KCL to handle many of the complex tasks associated with using multiple servers to process a stream, such as load balancing, responding to instance failures, checkpointing records that have been processed, and reacting to resharding (increasing or decreasing the number of shards used to process streaming data).

When to use: Amazon Kinesis Data Streams is ideal for use cases where you want to process incoming data as it is received, or you want to create a high-availability cluster of servers to process incoming data with a custom application.

When not to use: If you have a simple use case that requires you to write data to specific services in near real time, you should consider Kinesis Firehose if it supports your target destination. If you are looking to migrate an existing Apache Kafka cluster to AWS, then you may want to consider migrating to Amazon MSK. If Apache Kafka supports third-party integration that would be useful to you, you may want to consider Amazon MSK.

Amazon Kinesis Data Analytics

Amazon Kinesis Data Analytics simplifies the process of processing streaming data, using either standard SQL queries or an Apache Flink application.

An example of a use case for Kinesis Data Analytics is to analyze incoming clickstream data from an e-commerce website to get near real-time insight into the sales of a product. In this use case, an organization may want to know how the promotion of a specific product is impacting sales to see whether the promotion is being effective, and Kinesis Data Analytics can enable this using relatively simple SQL queries to process records being sent from their web server clickstream logs. This enables the business to quickly get answers to questions such as "how many sales of product x have there been in each 5-minute period since our promotion went live?"

When to use: If you want to use SQL expressions to analyze data or extract key metrics over a rolling time period, Kinesis Data Analytics significantly simplifies this task. If you have an existing Apache Flink application that you want to migrate to the cloud, consider running the application using Kinesis Data Analytics.

Amazon Kinesis Video Streams

Amazon Kinesis Video Streams can be used to process time-bound streams of unstructured data such as video, audio, and RADAR data.

Kinesis Video Streams takes care of provisioning and scaling the compute infrastructure that is required to ingest streaming video (or other types of media files) from potentially millions of sources. Kinesis Video Streams enables playback of video for live and on-demand viewing and can be integrated with other Amazon API services to enable applications such as computer vision and video analytics.

Appliances such as video doorbell systems, home security cameras, and baby monitors can stream video through Kinesis Video Analytics, simplifying the task of creating full-featured applications to support these appliances.

When to use: When creating applications that use a supported source, Kinesis Video Streams significantly simplifies the process of ingesting streaming media data and enabling live or on-demand playback.

Note about AWS service reliability

AWS services are known to be extremely reliable, and generally significantly exceed the uptime and reliability of what most organizations can achieve in their own data centers. However, as Werner Vogels (Amazon's CTO) has been known to say, *"Everything fails all the time."*

In November 2020, the Amazon Kinesis service running out of data centers in the Northern Virginia region (us-east-1) experienced a period of a number of hours where there were *increased error rates* for users of the service. During this time, many companies reported having their services affected, including Roomba vacuum cleaners, Ring doorbells, The Washington Post newspaper, Roku, and others.

This is a clear reminder that while AWS services generally offer extremely high levels of availability, if you require absolutely minimal downtime you need to design the ability to fail-over to a different AWS Region in your architecture.

Overview of Amazon MSK for streaming data ingestion

Apache Kafka is a popular open source distributed event streaming platform that enables an organization to create high-performance streaming data pipelines and applications, and Amazon **MSK (Managed Streaming for Apache Kafka)** is a managed version of Apache Kafka available from AWS.

While Apache Kafka is a popular choice for organizations, it can be a challenge to install, scale, update, and manage in an on-premises environment, often requiring specialized skills. To simplify these tasks, AWS offers Amazon MSK, which enables an organization to deploy an Apache Kafka cluster with a few clicks in the console, and reduces the management overhead by automatically monitoring cluster health and replacing failed components.

When to use: Amazon MSK is an ideal choice if your use case is a replacement for an existing Apache Kafka cluster, or if you want to take advantage of the many third-party integrations from the open source Apache Kafka ecosystem.

When not to use: Amazon Kinesis may be a better streaming solution if you are creating a new solution from scratch, as Kinesis is serverless and you only pay for data throughput (whereas with Amazon MSK you pay for the cluster, whether you are sending data through it or not).

Overview of Amazon AppFlow for ingesting data from SaaS services

Amazon AppFlow can be used to ingest data from popular SaaS services, and to transform and write the data out to common analytic targets, such as Amazon S3, Amazon Redshift, and Snowflake (a popular cloud data warehousing solution), as well as being able to write to some SaaS services.

For example, AppFlow can be used to ingest lead data from **Marketo**, a developer of marketing automation solutions, where your organization may capture details about a new lead. Using AppFlow, you can create a flow that will automatically create a new **Salesforce** contact record whenever a new Marketo lead is created.

From a data engineering perspective, you can create flows that will automatically write out new opportunity records created in Salesforce into your S3 data lake or Redshift data warehouse, enabling you to join those opportunity records with other datasets to perform advanced analytics.

AppFlow can be configured to run on a schedule, or in response to specific events, and can filter data, mask data, validate data, and perform calculations from data fields in the source.

While it is expected that new integrations will be added over time, as of the time of publication of this book the following integrations were supported by Amazon AppFlow:

- AWS services:
 - **Amazon EventBridge** (a serverless event bus that ingests data and routes it to targets)
 - **Amazon Redshift** (a cloud-based data warehousing service)
 - **Amazon S3** (an object storage service, often used as the storage layer for analytic data lakes)
 - **Amazon Honeycode** (a managed service for building mobile and web applications with no programming required)
 - **Amazon Lookout for Metrics** (a machine learning service for identifying outliers in business and operational metrics and determining their root cause)

- Third-party services:
 - **Amplitude** (a product analytics toolset)
 - **Datadog** (an application monitoring service)
 - **Dynatrace** (an applications and infrastructure monitoring service)
 - **Google Analytics** (a service for monitoring and tracking website traffic)
 - **Infor Nexus** (an on-demand global supply chain management platform)
 - **Marketo** (marketing automation software to help engage customers and prospects)
 - **Salesforce** (customer relationship management and related services)
 - **ServiceNow** (a platform for managing digital workflows)
 - **Singular** (a marketing analytics and ETL solution)
 - **Slack** (a channel-based messaging platform)
 - **Snowflake** (a cloud-based data warehouse solution)
 - **Trend Micro** (a workload security solution)
 - **Upsolver** (a service for turning event streams into analytics-ready data)
 - **Veeva** (a cloud computing service focused on pharmaceutical and life sciences companies)
 - **Zendesk** (a customer service and helpdesk platform)

When to use: Amazon AppFlow is an ideal choice for ingesting data into AWS if one of your data sources is a supported SaaS.

Overview of Amazon Transfer Family for ingestion using FTP/SFTP protocols

The **Amazon Transfer Family** provides a fully managed service that enables file transfers directly into and out of Amazon S3 using common file transfer protocols, including **FTP** and **SFTP**.

Many organizations today still make use of these protocols to exchange data with other organizations. For example, a real-estate company may receive the latest **MLS (Multi-Listing Service)** files from an MLS provider via SFTP. In this case, the real-estate company would have configured a server running SFTP, and created an SFTP user account that the MLS provider can use to connect to the server and transfer the files.

With Amazon Transfer for SFTP, the real-estate company could easily migrate to the managed AWS service, replicating the account setup that exists for their MLS provider on their on-premises server with an account in their Amazon Transfer service. With little to no change on the side of the provider, when future transfers are made via the managed AWS service, these would be written directly into Amazon S3, making the data immediately accessible to data transformation pipelines created for the Amazon S3-based data lake.

When to use: If an organization currently receives data via FTP, SFTP, or FTPS, they should consider migrating to the managed version of this service offered by Amazon Transfer.

Overview of Amazon DataSync for ingesting from on-premises storage

There is often a requirement to ingest data from existing on-premises storage systems, and **Amazon DataSync** simplifies this process while offering high performance and stability for the data transfers.

Network File System (NFS) and **Server Message Block (SMB)** are two common protocols that are used to allow computer systems to access files stored on a different system. With DataSync, you can easily ingest and replicate data from file servers that use either of these protocols. DataSync also supports ingesting data from on-premises object-based storage systems that are compatible with core **AWS S3 API** calls.

DataSync can write to multiple targets within AWS, including Amazon S3, making it an ideal way to sync data from on-premises storage into your AWS S3 data lake. For example, if you have a solution running in your data center that writes out end-of-day transactions to a file share, DataSync can ensure that the data is synced to your S3 data lake. Another common use case is to transfer large amounts of historical data from an on-premises system into your S3 data lake.

When to use: Amazon DataSync is a good choice when you're looking to ingest current or historical data from compatible on-premises storage systems to AWS over a network connection.

When not to use: For very large historical datasets where sending the data over a network connection is not practical, you should consider using the Amazon Snow family of devices. If you want to perform preprocessing of data, such as converting Apache web server log files to JSON, consider using Amazon Kinesis Agent to preprocess the data and then send data to Amazon S3 via Amazon Kinesis Firehose.

Overview of the AWS Snow family of devices for large data transfers

For use cases where there are very large datasets that need to be ingested into AWS, and either a lack of a good network connection or just the sheer size of the dataset makes it impractical to transfer via a network connection, the **AWS Snow family of devices** can be used.

The AWS Snow family of devices are ruggedized devices that can be shipped to a location and attached to a network connection in the local data center. Data can be transferred over the local network, and the device is then shipped back to AWS where the data will be transferred to Amazon S3. All the devices offer encryption of data at rest, and most of the devices also offer compute ability, enabling edge computing use cases.

There are multiple devices available for different use cases, as summarized here:

- **AWS Snowcone:** Lightweight (4.5 lb/2.1 kg) device with 8 TB of usable storage
- **AWS Snowball Edge Optimized (for Data Transfer):** Mediumweight (49.7 lb/22.5 kg) device with 80 TB of usable storage
- **AWS Snowmobile:** Large 45-foot ruggedized shipping container pulled by a semi-trailer truck. Capacity of up to 100 PB

AWS services for transforming data

Once your data is ingested into an appropriate AWS service, such as Amazon S3, the next stage of the pipeline needs to **transform** the data to optimize it for analytics and to make it available to your data consumers.

Some of the tools we discussed in the previous section for ingesting data into AWS can perform light transformations as part of the ingestion process. For example, Amazon DMS can write out data in Parquet format (a format optimized for analytics), as can Kinesis Firehose. However, heavier transformations are often required to fully optimize your data for a differing set of analytic tasks and diverse data consumers, and in this section, we will examine some of the core AWS services that can be used for this.

Overview of AWS Lambda for light transformations

AWS Lambda provides a serverless environment for executing code and is one of AWS's most popular services. You can trigger your Lambda function to execute your code in multiple ways, including through integration with over 140 other AWS services, and you only pay for the duration that your code executes, billed in 1-millisecond increments, and based on the amount of memory that you allocate for your function.

In the data engineering world, a common use case for Lambda is for performing validation or light processing and transformation of incoming data. For example, if you have incoming CSV files being sent by one of your partners throughout the day, you can trigger a Lambda function to run each time a new file is received, have your code validate that the file is a valid CSV file, perform some computation on one of the columns and update a database with the result, and then move the file into a different bucket where a batch process will later process all files received for the day.

With the ability to run for up to 15 minutes, and with a maximum memory configuration of 10 GB, it is possible to do more advanced processing as well. For example, you may receive a ZIP file containing hundreds of XML files, and in your Lambda function you want to unzip the file, and then for each file you want to validate that it is valid XML, perform calculations on fields in the file to update various other systems, concatenate the contents of all the files, and write that out in Parquet format in a different zone of your data lake.

Lambda is also massively parallel, meaning that it can easily scale for highly concurrent workloads. In the preceding example of processing CSV files as they arrive in an S3 bucket, if hundreds of files were all delivered within a period of just a few seconds, a separate Lambda instance would be spun up for each file, and AWS would automatically handle the scaling of the Lambda functions to enable this. By default, you can have 1,000 concurrent Lambda executions within an AWS Region for your account, but you can work with AWS support to increase this limit into the hundreds of thousands.

AWS Lambda supports many different languages, including **Python**, which has become one of the most popular languages for data engineering-related tasks.

Overview of AWS Glue for serverless Spark processing

AWS Glue has multiple components that could have been split into multiple separate services, but these components can all work together, and so AWS has grouped them together into the AWS Glue family. In this section, we look at the core Glue components related to data processing.

Serverless ETL processing

At the heart of AWS Glue is a serverless environment providing either a Python engine (known as **Glue Python shell**) or an Apache Spark engine for performing data transformations and processing. Python can be used for performing transformations on small to medium datasets, while Apache Spark enables optimal processing for very large datasets:

- Apache Spark is an open source engine for distributed processing of large datasets across a cluster of compute nodes, which makes it ideal for taking a large dataset, splitting the processing work among the nodes in the cluster, and then returning a result. As Spark does all processing in memory, it is highly efficient and performant and has become the tool of choice for many organizations looking for a solution for processing large datasets.
- Python, which runs on a single node, has become an extremely popular language for performing data engineering-related tasks in scenarios where the power of a multi-node cluster is not required.

The following diagram depicts the two different Glue engines – a single-node Glue Python shell on the left, and a multi-node Glue Apache Spark cluster on the right:

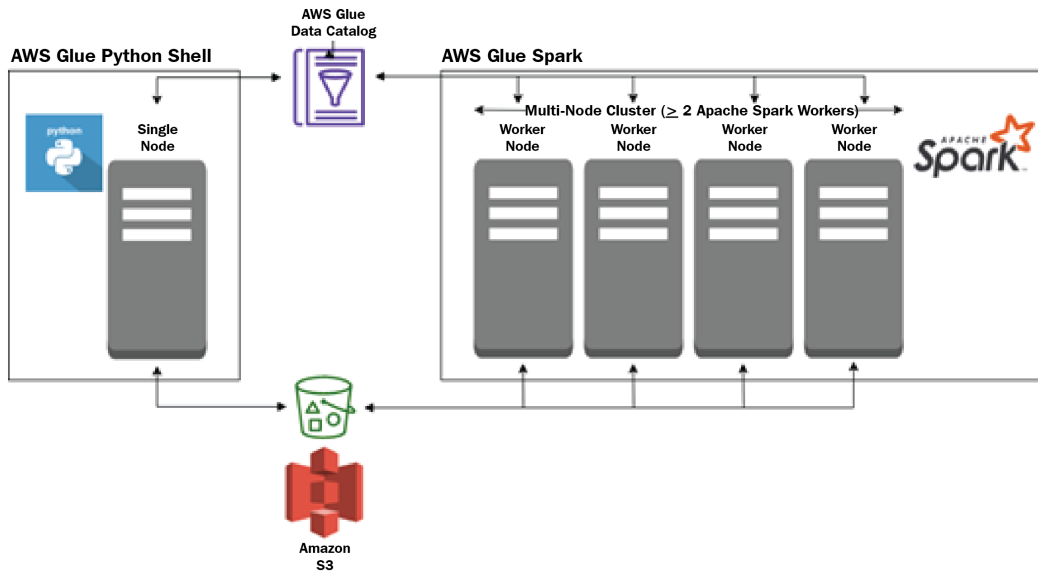


Figure 3.1 – Glue Python shell and Glue Spark engines

Both engines can work with data that resides in Amazon S3, and with the **AWS Glue Data Catalog**. Both engines are serverless from the perspective of a user, meaning a user does not need to deploy or manage servers, a user just needs to specify the number of **Data Processing Units (DPUs)** that they want to power their job. Glue ETL jobs are charged based on the number of DPUs configured, as well as the amount of time that the underlying code executes for in the environment.

While AWS Glue does provide additional Spark libraries and functionality to simplify some common ETL tasks, their use is optional, and existing Spark code can be easily run as is with AWS Glue.

AWS Glue also supports **Spark Streaming**, an extension of the core Spark API designed to process live data streams.

AWS Glue Data Catalog

To complement the ETL processing functionality described previously, AWS Glue also includes a *data catalog* that can be used to provide a logical view of data stored physically on a disk, and objects in the catalog can then be directly referenced from your ETL code.

In a scenario where you use DMS to replicate your **Human Resources (HR)** database to S3, you will end up with a prefix (directory) in S3 for each table from the source database. In this directory, there will generally be multiple files containing the data from the source table – for example, 20 CSV files containing the rows from the source employee table.

The Glue catalog can provide a logical view of this dataset, and capture additional metadata about the dataset, in the data catalog. For example, the data catalog consists of a number of databases at the top level (such as the HR database), and each database contains one or more tables (such as the Employee table), and each table contains metadata, such as the column headings and data types for each column (such as `employee_id`, `lastname`, `firstname`, `address`, and `dept`), as well as references to the S3 location for the data that makes up that table.

In the following screenshot, we see a bucket that contains objects under the prefix `hr/employees` and a number of CSV files that contain data imported from the employee database:

The screenshot displays the Amazon S3 console interface for a bucket named 'employee/'. The breadcrumb trail shows the path: Amazon S3 > dataeng-[redacted] hr/ > employee/. A 'Copy S3 URI' button is visible in the top right corner. Below the breadcrumb, there are two tabs: 'Objects' (selected) and 'Properties'. The main content area shows 'Objects (20)' and a description: 'Objects are the fundamental entities stored in Amazon S3. For others to access your objects, you'll need to explicitly grant them permissions. [Learn more](#)'. Below this, there are buttons for 'Refresh', 'Delete', 'Actions', 'Create folder', and 'Upload'. A search bar with the placeholder 'Find objects by prefix' is present. The object list table has columns for 'Name', 'Type', and 'Storage class'. The objects listed are CSV files with names ranging from 'LOAD00000001.csv' to 'LOAD00000011.csv', all with a 'Standard' storage class.

<input type="checkbox"/>	Name	Type	Storage class
<input type="checkbox"/>	LOAD00000001.csv	csv	Standard
<input type="checkbox"/>	LOAD00000002.csv	csv	Standard
<input type="checkbox"/>	LOAD00000003.csv	csv	Standard
<input type="checkbox"/>	LOAD00000004.csv	csv	Standard
<input type="checkbox"/>	LOAD00000005.csv	csv	Standard
<input type="checkbox"/>	LOAD00000006.csv	csv	Standard
<input type="checkbox"/>	LOAD00000007.csv	csv	Standard
<input type="checkbox"/>	LOAD00000008.csv	csv	Standard
<input type="checkbox"/>	LOAD00000009.csv	csv	Standard
<input type="checkbox"/>	LOAD00000010.csv	csv	Standard
<input type="checkbox"/>	LOAD00000011.csv	csv	Standard

Figure 3.2 – Amazon S3 bucket with CSV files making up the Employee table

The screenshot of the following AWS Glue Data Catalog shows us the logical view of this data. We can see that this is the employee table, and it references the S3 location shown in the preceding screenshot. In this logical view, we can see that the employee table is in the HR database, and we can see the columns and data types that are contained in the CSV files:

Tables > employee Last updated 9 Dec 2020 09:46 PM Table Version (Current version) ▾

[Edit table](#) [Delete table](#) [View properties](#) [Compare versions](#) [Edit schema](#)

Name employee

Description hr

Database hr

Classification csv

Location s3://data-warehouse-us-east-1-us-east-1-123456789012-us-east-1/hr/employee/

Connection No

Deprecated No

Last updated Wed Dec 09 21:46:50 GMT-500 2020

Input format org.apache.hadoop.mapred.TextInputFormat

Output format org.apache.hadoop.hive.q1.io.HiveIgnoreKeyTextOutputFormat

Serde serialization lib org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe

Serde parameters field.delim ,

skip.header.line.count 1 sizeKey 6300 objectCount 20 UPDATED_BY_CRAWLER hr-employee-crawler

Table properties CrawlerSchemaSerializerVersion 1.0 recordCount 40 averageRecordSize 156 CrawlerSchemaDeserializerVersion 1.0

compressionType none columnsOrdered true areColumnsQuoted false delimiter , typeOfData file

Schema Showing: 1 - 18 of 18

	Column name	Data type	Partition key	Comment
1	emp_id	bigint		
2	last_name	string		
3	first_name	string		
4	hire_date	bigint		
5	street_address	string		
6	street_address_2	string		
7	city	string		
8	state	string		

Figure 3.3 – AWS Glue Data Catalog showing a logical view of the Employee table

The AWS Glue Data Catalog is a **Hive metastore**-compatible catalog, and all you really need to know about that statement is that it means that the AWS Glue catalog works with a variety of other services and third-party products that can integrate with Hive metastore-compatible catalogs.

Within the AWS ecosystem, a number of services can use the AWS Glue Data Catalog. For example, Amazon Athena uses the AWS Glue Data Catalog to enable users to run SQL queries directly on data in Amazon S3, and Amazon EMR and the AWS Glue ETL engine use the catalog to enable users to reference catalog objects (such as databases and tables) directly in their ETL code.

AWS Glue crawlers

AWS Glue crawlers are processes that can examine a data source (such as a path in an S3 bucket) and automatically infer the schema and other information about that data source, so that the AWS Glue Data Catalog can be automatically populated with relevant information.

For example, we could point an AWS Glue Crawler at the S3 location where DMS replicated the Employee table of our HR database. When the Glue Crawler runs, it examines a portion of each of the files in that location, identifies the file type (CSV, Parquet), uses a classifier to infer the schema of the file (column headings and types), and then adds that information into a database in the Glue catalog.

Note that you can also add databases and tables to the Glue catalog using the Glue API, or via SQL statements in Athena, so using Glue crawlers to automatically populate the catalog is optional.

Overview of Amazon EMR for Hadoop ecosystem processing

Amazon EMR provides a managed platform for running popular open source big data processing tools, such as Apache Spark, Apache Hive, Apache Hudi, **Apache HBase**, **Presto**, **Pig**, and others. **Amazon EMR** takes care of the complexities of deploying these tools and managing the underlying clustered Amazon EC2 compute resources.

You may have noticed in the previous paragraph that Amazon EMR can be used to run Apache Spark, and you might be wondering why AWS has two services that effectively offer the same big data processing engine. While either service can be used to perform big data processing using the Apache Spark engine, there are important differences.

For a start, AWS Glue offers a serverless environment for running Apache Spark, whereas with Amazon EMR you need to specify the detailed configuration of the cluster you want to run Apache Spark. And, ultimately, this is probably one of the biggest differentiators between the services.

If your use case would benefit from being able to more finely tune the environment where Apache Spark runs, then Amazon EMR would be a better fit as it provides more options for specifying the configuration of the compute cluster and Spark settings than AWS Glue allows. Also, with AWS Glue you pay a slightly higher cost for an equivalent sized server than you would with Amazon EMR, but AWS Glue requires far less understanding or experience with regard to running an Apache Spark environment, and as a result, Glue requires much less configuration to get your Apache Spark code running.

There may also be workloads where you want a permanently running Apache Spark environment, which you can get with Amazon EMR at a lower cost. However, one of the benefits of using the cloud for your big data processing requirements is that you can run transient clusters that are spun up to run a specific job, and then shut down – and this is possible with both solutions.

In summary, if you have a team that has experience of running Apache Spark environments, and your use case requires clusters that are fine-tuned as far as compute power and Apache Spark settings go, then Amazon EMR may be the way to go. But if you have a simpler use case and just want to be able to take your Apache Spark code and get it running to process your data with minimal configuration, then AWS Glue may be best suited.

The other important differentiator is that Amazon EMR offers many additional frameworks and tools for big data processing. So, if you're migrating an environment that uses Apache Hive, Presto, or other toolsets supported in EMR, then Amazon EMR would be a great fit.

The following diagram shows an EMR cluster, including some of the open source projects that can be run on the cluster:

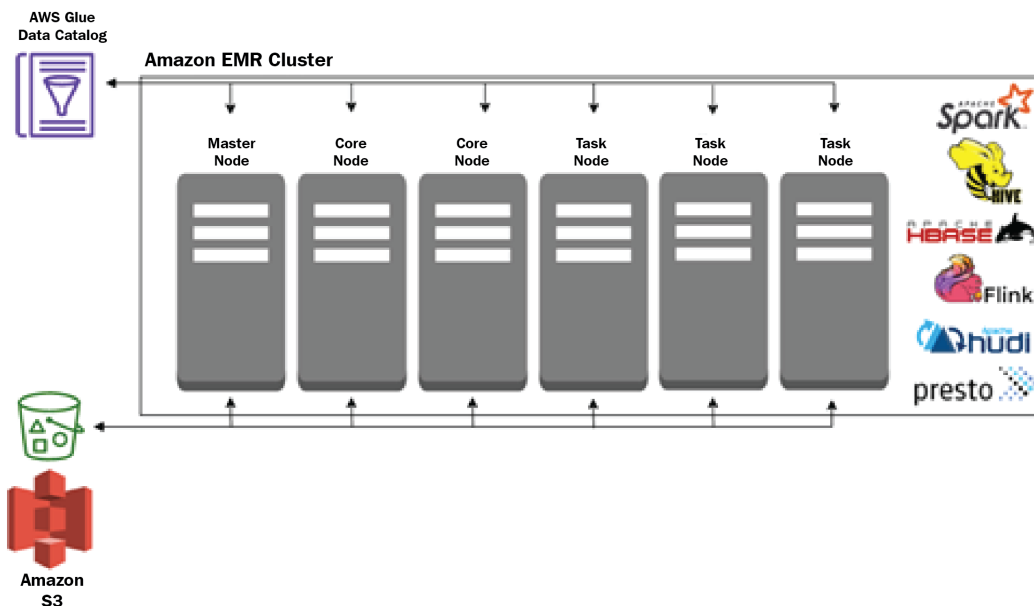


Figure 3.4 – High-level overview of an EMR cluster

Each EMR cluster requires a master node, and at least one **core node** (a worker node that includes local storage), and then optionally a number of **task nodes** (worker nodes that do not have any local storage).

AWS services for orchestrating big data pipelines

As discussed in *Chapter 2, Data Management Architectures for Analytics*, a data pipeline can be built to bring in data from source systems, and then transform that data, often moving the data through multiple stages, further transforming or enriching the data as it moves through each stage.

An organization will often have tens or hundreds of pipelines that work independently or in conjunction with each other on different datasets and perform different types of transformations. Each pipeline may use multiple services to achieve the goals of the pipeline and orchestrating all the varying services and pipelines can be complex. In this section, we look at a number of AWS services that help with this **orchestration** task.

Overview of AWS Glue workflows for orchestrating Glue components

In the *AWS services for transforming data* section, we covered AWS Glue, a service that includes a number of components. As a reminder, they are as follows:

- A serverless Apache Spark or Python shell environment for performing ETL transformations
- The Glue data catalog, which provides a centralized logical representation (database and tables) of the data in an Amazon S3 data lake
- Glue crawlers, which can be configured to examine files in a specific location, automatically infer the schema of the file, and add the file into the AWS Glue data catalog

AWS Glue workflows are a functionality within the AWS Glue service and have been designed to help orchestrate the various AWS Glue components. A workflow consists of an ordered sequence of steps that can run Glue crawlers and Glue ETL jobs (Spark or Python shell).

The following diagram shows a visual representation of a simple Glue workflow that can be built in the AWS Glue console:

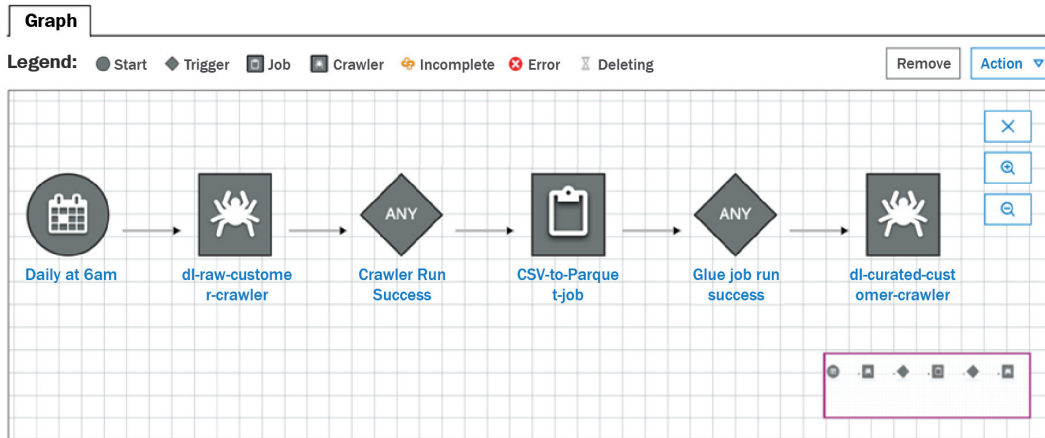


Figure 3.5 – AWS Glue workflow

This workflow orchestrates the following tasks:

- It runs a *Glue Crawler* to add newly ingested data from the raw zone of the data lake into the Glue data catalog.
- Once the Glue Crawler completes, it triggers a *Glue ETL job* to convert the raw CSV data into Parquet format, and writes to the curated zone of the data lake.
- When the Glue job is complete, it triggers a Glue Crawler to add the newly transformed data in the curated zone, into the Glue data catalog.

Each step of the workflow can retrieve and update the state information about the workflow. This enables one step of a workflow to provide state information that can be used by a subsequent step in the workflow. For example, a workflow may run multiple ETL jobs, and each ETL job can update state information, such as the location of files that it outputted, that will be available to be used by subsequent workflow steps.

The preceding diagram is an example of a relatively simple workflow, but AWS Glue workflows are capable of orchestrating much more complex workflows. However, it is important to note that Glue workflows can only be used to orchestrate Glue components, which are ETL jobs and Glue crawlers.

If you only use AWS Glue components in your pipeline, then AWS Glue workflows are well suited to orchestrate your data transformation pipelines. But if you have a use case that needs to incorporate other AWS services in your pipeline (such as AWS Lambda), then keep reading to learn about other available options.

Overview of AWS Step Functions for complex workflows

Another option for orchestrating your data transformation pipelines is **AWS Step Functions**, a service that enables you to create complex workflows that can be integrated with many AWS services.

Step Functions is serverless, meaning that you do not need to deploy or manage any infrastructure, and you pay for the service based on your usage, not on fixed infrastructure costs.

With Step Functions, you use JSON to define a state machine using a structured language known as the **Amazon States Language**. Alternatively, you can use Step Functions Workflow Studio to create a workflow using a visual interface that supports drag and drop. The resulting workflow can run multiple tasks, can run different branches based on a choice, can enter a wait state where you specify a delay before the next step is run, can loop back to previous steps, as well as various other things that can be done to control the workflow.

When you start a state machine, you include JSON data as input text that will be passed to the first state in the workflow. The first state in the workflow uses the input data, performs the function it is configured to do (such as running a Lambda function using the input passed into the state machine), modifies the JSON data, and then passes the modified JSON data to the next state in the workflow.

You can trigger a step function using **Amazon EventBridge** (such as on a schedule or in response to something else triggering an EventBridge event) or can trigger the step function on-demand by calling the Step Functions API.

The following is an example of a Step Functions state machine:

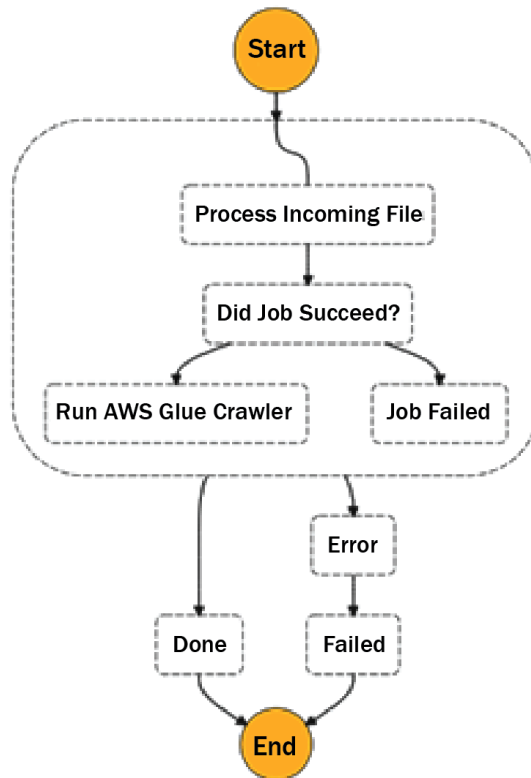


Figure 3.6 – AWS Step Functions state machine

This state machine performs the following steps:

1. A CloudWatch event is triggered whenever a file is uploaded to a particular Amazon S3 bucket, and the CloudWatch event starts our state machine, passing in a JSON object that includes the location of the newly uploaded file.
2. The first step, **Process Incoming File**, runs a Glue Python shell job that takes the location of the uploaded file as input and processes the incoming file (for example, converting from CSV to Parquet format). The output of the Python function indicates whether the file processing succeeded or failed, and if succeeded it also includes the S3 path where the Parquet file was written. This information is included in the JSON passed to the next step.
3. The **Did Job Succeed?** step is of type Choice. It examines the JSON data passed to the step, and if the `jobStatus` field is set to `succeeded`, it branches to **Run AWS Glue Crawler**. If the `jobStatus` field is set to `failed`, it branches to **Job Failed**.

4. In the **Run AWS Glue Crawler** step, a Lambda function is triggered, which in turn triggers an AWS Glue Crawler to run against the location where the previous Lambda function had written the Parquet file. Note that Step Functions can directly trigger the running of a Glue job but currently does not support directly running a Glue Crawler, which is why we use a Lambda function to trigger the Crawler.
5. The **Job Failed** step stops the execution of the state machine and marks the execution as a failure.
6. Outside of the dotted line box in *Figure 3.6* we can perform error handling. We have a catch statement in our state machine that detects whether the state machine execution is in an error state, and if it is, it runs the **Error** step.
7. In the **Error** step, a Lambda function is triggered that sends out a notification to the data engineering team to indicate that file processing failed.

Overview of Amazon managed workflows for Apache Airflow

Apache Airflow is a popular open source solution for orchestrating complex data engineering workflows. It was created by **Airbnb** in 2014 to help their internal teams manage their increasingly complex workflows and became a top-level Apache project in 2019.

Airflow enables users to create processing pipelines programmatically (using the Python programming language) and provides a user interface to monitor the execution of the workflows. Complex workflows can be created, and Airflow includes support for a wide variety of integrations, including integrations with services from AWS, Microsoft Azure, Google Cloud Platform, and others.

However, installing and configuring Apache Airflow in a way that can support the resilience and scaling required for large production environments is complex, and maintaining and updating the environment can be challenging. As a result, AWS created **Managed Workflows for Apache Airflow (MWAA)**, which enables users to easily deploy a managed version of Apache Airflow that can automatically scale out additional workers as demand on the environment increases, and scale in the number of workers as demand decreases.

An MWAA environment consists of the following components:

- **Scheduler:** The scheduler runs a multithreaded Python process that controls what tasks need to be run, and where and when to run those tasks.
- **Worker/executor:** The worker/s execute/s tasks. Each MWAA environment contains at least one worker, but when configuring the environment, you can specify the maximum number of additional workers that should be made available. MWAA automatically scales out the number of workers up to that maximum, but will also automatically reduce the number of workers as tasks are completed and if no new tasks need to run. The workers are linked to your VPC (the private network in your AWS account).
- **Meta-database:** This runs in the MWAA service account and is used to track the status of tasks.
- **Web server:** The web server also runs in the MWAA service account and provides a web-based interface that users can use to monitor and execute tasks.

Note that even though the meta-database and web server run in the MWAA service account, there are separate instances of these for every MWAA environment, and there are no components of the architecture that are shared between different MWAA environments.

When migrating from an on-premises environment where you already run Apache Airflow, or if your team already has Apache Airflow skills, then the MWAA service should be considered for managing your data processing pipelines and workflows in AWS. However, it is important to note that while this is a managed service (meaning that AWS deploys the environment for you and upgrades the Apache Airflow software), it is not a serverless environment.

With MWAA, you select a core environment size (small, medium, or large), and are charged based on the environment size, plus a charge for the amount of storage used by the meta-database and for any additional workers you make use of. Whether you run one 5-minute job per day, or run multiple simultaneous jobs 24 hours a day, 7 days a week, the charge for your core environment will remain the same. With serverless environments, such as Amazon Step Functions, billing is based on a consumption model, so there is no underlying monthly charge.

AWS services for consuming data

Once the data has been transformed and optimized for analytics, the various data consumers in an organization need easy access to the data via a number of different types of interfaces. Data scientists may want to use standard SQL queries to query the data, while data analysts may want to both query the data in place using SQL and also load subsets of the data into a high-performance data warehouse for low-latency, high-concurrency queries and scheduled reporting. Business users may prefer accessing data via a visualization tool that enables them to view data represented as graphs, charts, and other types of visuals.

In this section, we introduce a number of AWS services that enable different types of data consumers to work with our optimized datasets. We don't cover all services that can be used to consume data in this section, but instead highlight the primary services relevant to the data engineering role.

Overview of Amazon Athena for SQL queries in the data lake

Amazon Athena is a serverless solution for using standard SQL queries to query data that exists in a data lake, or in other data sources. As soon as a dataset has been written to Amazon S3 and cataloged in the AWS Glue Data Catalog, users can run complex SQL queries against the data without needing to set up or manage any infrastructure.

What is SQL?

Structured Query Language (SQL) is a standard language used to query relational datasets. A person proficient in SQL can draw information out of very large relational datasets easily and quickly, combining different tables, filtering results, and performing aggregations.

Data scientists and data analysts frequently use SQL to explore and better understand datasets that may be useful to them. Enabling these data consumers to query the data in an Amazon S3 data lake, without needing to first load the data into a traditional database system, increases productivity and flexibility for these data consumers.

Many tools are designed to interface with data via SQL, and these tools often connect to the SQL data source using either a **JDBC** or **ODBC** database connection. Amazon Athena enables a data consumer to query datasets in the data lake (or other connected data sources) through the AWS Management Console interface, or through a JDBC or ODBC driver.

Graphical SQL query tools, such as **SQL Workbench**, can connect to Amazon Athena via the JDBC driver, and you can programmatically connect to Amazon Athena and run SQL queries in your code through the ODBC driver.

Athena Federated Query, a feature of Athena, enables you to build connectors so that Athena can query other data sources, beyond just the data in an S3 data lake. Amazon provides a number of pre-built open source connectors for Athena, enabling you to connect Athena to sources such as **Amazon DynamoDB** (a NoSQL database), as well as other Amazon-managed relational database engines, and even Amazon CloudWatch Logs, a centralized logging service. Using this functionality, a data consumer can run a query using Athena that gets active orders from Amazon DynamoDB, references that data against the customer database running on PostgreSQL, and then brings in historical order data for that customer from the S3 data lake – all in a single SQL statement.

Overview of Amazon Redshift and Redshift Spectrum for data warehousing and data lakehouse architectures

Data warehousing is not a new concept or technology (as we discussed in *Chapter 2, Data Management Architectures for Analytics*), but Amazon Redshift was the first cloud-based data warehouse to be created. Launched in 2012, it was AWS's fastest-growing service by 2015, and today there are tens of thousands of customers that use it.

A Redshift data warehouse is designed for reporting and analytic workloads, commonly referred to as **Online Analytical Processing (OLAP)** workloads. Redshift provides a clustered environment that enables all the compute nodes in the cluster to work with portions of the data involved in a SQL query, helping to provide the best performance for scenarios where you are working with data that has been stored in a highly structured manner, and you need to do complex joins across multiple large tables on a regular basis. As a result, Redshift is an ideal query engine for reporting and visualization services that need to work with large datasets.

A typical SQL query that runs against a Redshift cluster would be likely to retrieve data from hundreds or thousands, or even millions, of rows in the database, often performing complex joins between different tables, and likely doing calculations such as aggregating, or averaging, certain columns of data. The queries run against the data warehouse will often be used to answer questions such as "What was the average sale amount for sales in our stores last month, broken down by each ZIP code of the USA?", or "Which products, across all of our stores, have seen a 20% increase in sales between Q4 last year and Q1 of this year?".

In a modern analytic environment, a common use case for a data warehouse would be to load a subset of data from the data lake into the warehouse, based on which data needs to be queried most frequently and which data needs to be used for queries requiring the best possible performance.

In this scenario, a data engineer may create a pipeline to load customer, product, sales, and inventory data into the data warehouse on a daily basis. Knowing that 80% of the reporting and queries will be on the last 12 months of sales data, the data engineer may also design a process to remove all data that's more than 12 months old from the data warehouse.

But what about the 20% of queries that need to include historical data that's more than 12 months old? That's where Redshift Spectrum comes in, a feature of Amazon Redshift that enables a user to write a single query that queries data that has been loaded into the data warehouse, as well as data that exists outside the data warehouse, in the data lake. To enable this, the data engineer can configure the Redshift cluster to connect with the AWS Glue Data Catalog, where all the databases and tables for our data lake are defined. Once that has been configured, a user can reference both internal Redshift tables and tables registered in the Glue data catalog.

The following diagram shows the Redshift and Redshift Spectrum architecture:

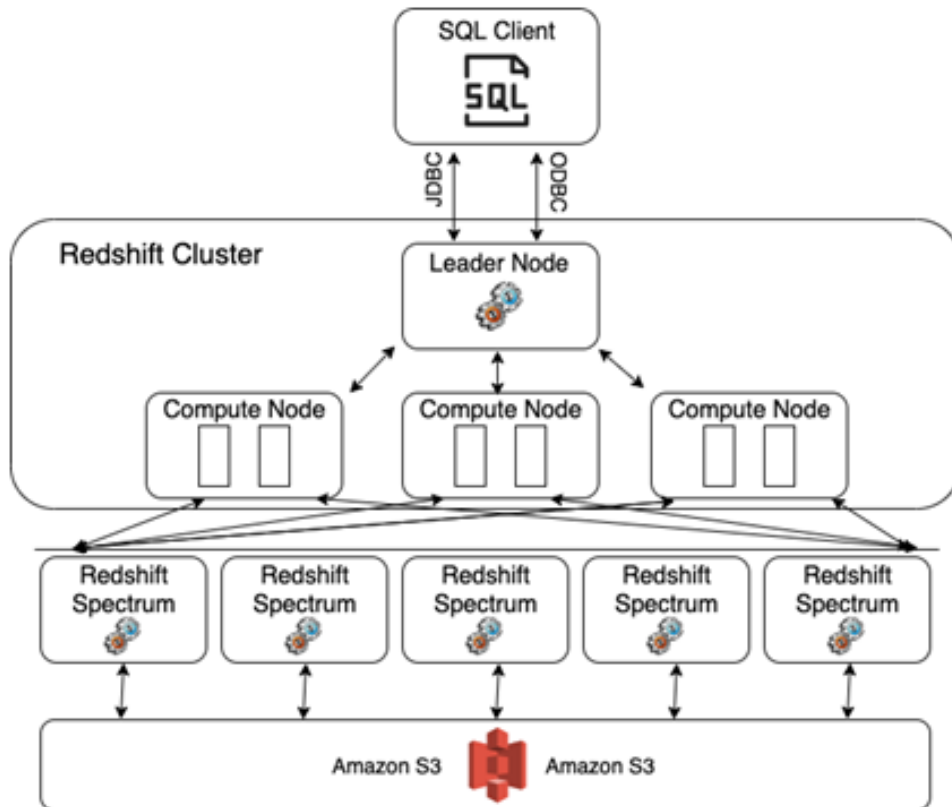


Figure 3.7 – Redshift architecture

In the preceding diagram, we can see that a user connects to the Redshift leader node (via JDBC or ODBC). This node does not query data directly but is effectively the *central* brain behind all the queries that do run on the cluster. In a scenario where a user is running a query that needs to query both current (last 12 months of) sales data, as well as historical sales data, the process works as follows:

1. Using a SQL client, the user makes a connection and authenticates with the Redshift leader node, and sends through a SQL statement that queries both the `current_sales` table (a table in which the data exists within the Redshift cluster and contains the past 12 months of sales data) and the `historical_sales` table (a table that is registered in the Glue data catalog, and where the data files are located in the Amazon S3 data lake, which contains historical sales data going back 10 years).
2. The leader node analyzes and optimizes the query, compiles a query plan, and pushes individual query execution plans to the compute nodes in the cluster.

3. The compute nodes query data they have locally (for the `current_sales` table) and query the AWS Glue Data Catalog to gather information on the external `historical_sales` table. Using the information they gathered, they can optimize queries for the external data and push those queries out to the Redshift Spectrum layer.
4. Redshift Spectrum is outside of a customer's Redshift cluster and is made up of thousands of worker nodes (Amazon EC2 compute instances) in each AWS Region. These worker nodes are able to scan, filter, and aggregate data from the files in Amazon S3, and then stream results back to the Amazon Redshift cluster.
5. The Redshift cluster performs final operations to join and merge data, and then returns the results to the user's SQL client.

Overview of Amazon QuickSight for visualizing data

"A picture is worth a thousand words" is a common saying, and is a sentiment that most business users would strongly agree with. Imagine for a moment that you are a busy sales manager, and it's Monday morning and you need to quickly determine how your various sales territories performed last quarter before your 9 a.m. call.

Your one option is to receive a detailed spreadsheet showing the specific sales figures broken down by territory and segment, as per *Figure 3.8*:

Sales Data by Territory and Segment			
Territory	SMB	Midmarket	Enterprise
East Q3	\$ 168,778	\$ 210,696	\$ 423,875
East Q4	\$ 196,254	\$ 244,995	\$ 492,878
South Q3	\$ 99,361	\$ 168,572	\$ 263,119
South Q4	\$ 116,895	\$ 198,320	\$ 309,552
Central Q3	\$ 132,882	\$ 203,082	\$ 296,332
Central Q4	\$ 156,332	\$ 238,920	\$ 245,000
Mountain Q3	\$ 127,699	\$ 213,247	\$ 271,440
Mountain Q4	\$ 146,780	\$ 245,112	\$ 312,000
West Q3	\$ 156,147	\$ 210,558	\$ 396,885
West Q4	\$ 185,889	\$ 250,664	\$ 526,995

Figure 3.8 – SALES Table showing sales data by territory and segment

The other option you have is to receive a graphical representation of the data in the form of a bar graph, as shown in *Figure 3.9*. Within the interface, you can filter data by territory and market segment, and also drill down to get more detailed information:

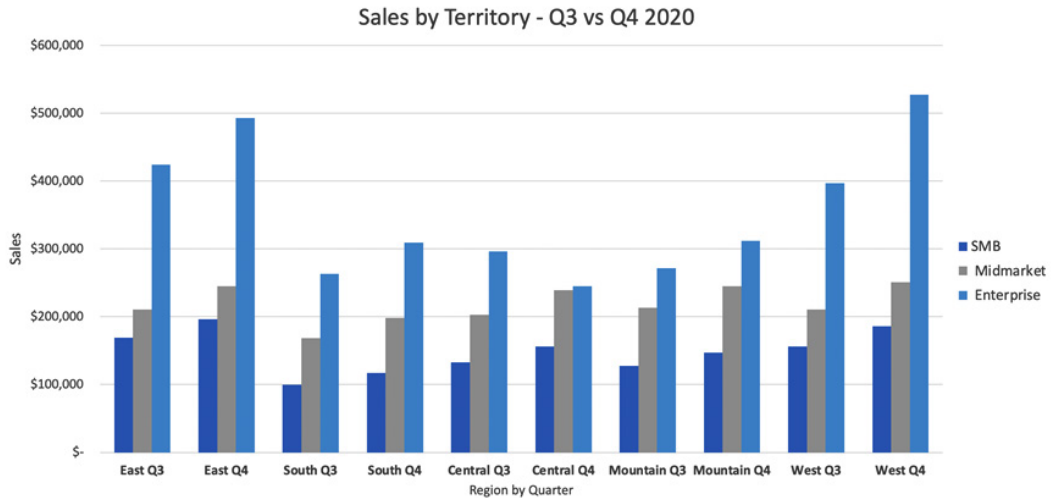


Figure 3.9 – Sample graph showing sales data by territory and segment

Most people would prefer the graphical representation of the data, as they can easily visually compare sales between quarters, segments, and territories, or identify the top sales territory and segment with just a glance. As a result, the use of business intelligence tools, which provide visual representations of complex data, is extremely popular in the business world.

Amazon QuickSight is a service from AWS that enables the creation of these types of complex visualizations, but beyond just providing static visuals, the charts created by QuickSight enable users to filter data and drill down to get further details. For example, our sales manager could filter the visual to just see the numbers from Q4, or to just see the enterprise segment. The user could also drill down into the Q4 data for the enterprise segment in the West territory to see sales by month, for example.

Amazon QuickSight is serverless, which means there are no servers for the organization to set up or manage, and there is a simple monthly fee based on the user type (either an author, who can create new visuals, or a reader, who can view visuals created by authors).

A data engineer can configure QuickSight to access data from a multitude of sources, including accessing data in an Amazon S3-based data lake via integration with Amazon Athena.

In the next section, we wrap up the chapter by getting hands-on with building a simple transformation that converts a CSV file into Parquet format, using Lambda to perform the transformation.

Hands-on – triggering an AWS Lambda function when a new file arrives in an S3 bucket

In the hands-on portion for this chapter, we're going to configure an S3 bucket to automatically trigger a Lambda function whenever a new file is written to the bucket. In the Lambda function, we're going to make use of an open source Python library called **AWS Data Wrangler**, created by AWS Professional Services to simplify common ETL tasks when working in an AWS environment. We'll use the AWS Data Wrangler library to convert a CSV file into Parquet format, and then update the AWS Glue Data Catalog.

Creating a Lambda layer containing the AWS Data Wrangler library

Lambda layers allow your Lambda function to bring in additional code, packaged as a .zip file. In our use case, the Lambda layer is going to contain the AWS Data Wrangler Python library, which we can then attach to any Lambda function where we want to use the library.

To create a Lambda layer, do the following:

1. Access the 2.10.0 version of the AWS Data Wrangler library in GitHub at <https://github.com/aws-labs/aws-data-wrangler/releases>. Under **Assets**, download the `awsdatawrangler-layer-2.10.0-py3.8.zip` file to your local drive.
2. Log in to the AWS Management Console as the administrative user you created in *Chapter 1, An Introduction to Data Engineering* (<https://console.aws.amazon.com>).
3. Make sure that you are in the region that you have chosen for performing the hands-on sections in this book. The examples in this book use the `us-east-2` (Ohio) region.
4. In the top search bar of the AWS console, search for and select the **Lambda** service.
5. In the left-hand menu, under **Additional Resources**, select **Layers**, and then click on **Create layer**.

6. Provide a **name** for the layer (for example, `awsDataWrangler210_python38`), an optional description, and then upload the `.zip` file you downloaded from GitHub. For **Compatible runtimes – optional**, select **Python 3.8** and then click **Create**. The following screenshot shows the configuration for this step:

The screenshot shows the 'Create layer' configuration page in the AWS Lambda console. The page is titled 'Create layer' and has a 'Layer configuration' section. The 'Name' field contains 'awsDataWrangler210_python38'. The 'Description - optional' field contains 'AWS Data Wrangler, Version 2.10.0, for Python 3.8'. Under the 'Upload' section, the 'Upload a .zip file' radio button is selected. An 'Upload' button is next to the file name 'awswrangler-layer-2.10.0-py3.8.zip (45.1 MB)'. Below this, there is a note: 'For files larger than 10 MB, consider uploading using Amazon S3.' The 'Compatible architectures - optional' section has two checkboxes: 'x86_64' and 'arm64', both of which are unchecked. The 'Compatible runtimes - optional' section has a dropdown menu labeled 'Runtimes' with a downward arrow. Below the dropdown, a tag 'Python 3.8' with a close button 'X' is visible. The 'License - optional' section has an empty text area. At the bottom right, there are two buttons: 'Cancel' and 'Create'.

Figure 3.10 – Creating and configuring an AWS Lambda layer

By creating a Lambda layer for the AWS Data Wrangler library, we can use AWS Data Wrangler in any of our Lambda functions just by ensuring this Lambda layer is attached to the function.

Creating new Amazon S3 buckets

In this step, we create new Amazon S3 buckets. In a subsequent step, we will create a Lambda function that is automatically triggered whenever a new file is uploaded to the source bucket and writes out a transformed version of that file to the target bucket. As discussed in *Chapter 2, Data Management Architectures for Analytics*, it is common for data lakes to have multiple zones for the data to move through. In this section, we create a bucket to be our landing zone (for ingestion of raw files), and a clean zone (for files that have undergone initial processing and optimization).

To create the new Amazon S3 buckets, follow these steps:

1. Log in to the AWS Management Console as the administrative user you created in *Chapter 1, An Introduction to Data Engineering* (<https://console.aws.amazon.com>), and ensure you are in the region you have chosen for performing the hands-on sections in this book.
2. In the top search bar, search for and select the **S3** service, and then click on **Create bucket**.
3. Provide a name for your source bucket (for example, `dataeng-landing-zone-<initials>`). This is where we will upload a file later and have it trigger our Lambda function.

Remember that bucket names need to be globally unique (not just unique within your account), so if you receive an error when creating the bucket, modify your bucket name to ensure it is unique (such as adding additional letters or numbers to your initials).

4. Ensure that your bucket is being created in the region you have chosen to use for the exercises in this book.
5. Accept all other defaults and click **Create bucket**.

Repeat these steps, but this time we create an Amazon S3 bucket for writing out our newly transformed files, so provide a bucket name similar to the following: `dataeng-clean-zone-<initials>`.

Creating an IAM policy and role for your Lambda function

In this section, we are setting up a Lambda function to be triggered every time a new file is uploaded to a specific Amazon S3 bucket. For this to work, we need to ensure that our Lambda function has the following permissions:

- Read our source S3 bucket (for example, `dataeng-landing-zone-<initials>`)
- Write to our target S3 bucket (for example, `dataeng-clean-zone-<initials>`)
- Write logs to Amazon CloudWatch
- 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. From the **Services** dropdown, 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. Note that you can also copy and paste this policy by accessing the policy on this book's GitHub page. Note that if doing a copy and paste from the GitHub copy of this policy, you must replace `dataeng-landing-zone-<initials>` with the name of the source bucket you created in the previous step and replace `dataeng-clean-zone-<initials>` with the name of the target bucket you created in the previous step.

This first block of the policy configures the policy document and provides permissions for using CloudWatch log groups, log streams, and log events:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:PutLogEvents",
        "logs:CreateLogGroup",
        "logs:CreateLogStream"
```

```

    ],
    "Resource": "arn:aws:logs:*:*:*"
  },

```

This next block of the policy provides permissions for all Amazon S3 actions (get and put) that are in the Amazon S3 bucket specified in the resource section (in this case, our clean-zone and landing-zone buckets). Make sure you replace `dataeng-clean-zone-<initials>` and `dataeng-landing-zone-<initials>` with the name of the S3 buckets you created in a previous step:

```

    {
      "Effect": "Allow",
      "Action": [
        "s3:*"
      ],
      "Resource": [
        "arn:aws:s3:::dataeng-landing-zone-INITIALS/*",
        "arn:aws:s3:::dataeng-landing-zone-INITIALS",
        "arn:aws:s3:::dataeng-clean-zone-INITIALS/*",
        "arn:aws:s3:::dataeng-clean-zone-INITIALS"
      ]
    },

```

In the final statement of the policy, we provide permissions to use all AWS Glue actions (create job, start job, and delete job). Note that in a production environment, you should limit the scope specified in the resource section:

```

    {
      "Effect": "Allow",
      "Action": [
        "glue:*"
      ],
      "Resource": "*"
    }

```

```
]
}
```

4. Click on **Next Tags, and then Next: Review**.
5. Provide a name for the policy, such as `DataEngLambdaS3CWGluePolicy`, and then click **Create policy**.
6. In the left-hand menu, click on **Roles** and then **Create role**.
7. For trusted entity, ensure **AWS service** is selected, and for service, select **Lambda** and then click **Next: Permissions**. In *Step 4* of the next section (*Creating a Lambda function*), we will assign this role to our Lambda function.
8. Under **Attach permissions**, select the policy we just created (for example, `DataEngLambdaS3CWGluePolicy`) by searching and then clicking in the tick box. Then click **Next: Tags**.
9. Provide any tags you would like associated with this policy (optional) and click **Next: Review**.
10. Provide a role name, such as `DataEngLambdaS3CWGlueRole`, and click **Create role**.

Creating a Lambda function

We are now ready to create our Lambda function that will be triggered whenever a CSV file is uploaded to our source S3 bucket. The uploaded CSV file will be converted to Parquet, written out to the target bucket, and added to the Glue catalog using AWS Data Wrangler:

1. In the AWS console, from the **Services** dropdown, select the **Lambda** service, and in the left-hand menu select **Functions** and then click **Create function**.
2. Select **Author from scratch** and provide a **function name** (such as `CSVtoParquetLambda`).
3. For **Runtime**, select **Python 3.8** from the drop-down list.

4. Expand **Change default execution role** and select **Use an existing role**. From the drop-down list, select the role you created in the previous section (such as `DataEngLambdaS3CWGlueRole`):

The screenshot shows the AWS Lambda 'Create function' console. At the top, there are navigation links for 'Lambda', 'Functions', and 'Create function'. The main heading is 'Create function' with an 'Info' link. Below this, there is a prompt: 'Choose one of the following options to create your function.' Two options are presented in boxes: 'Author from scratch' (selected with a blue radio button) and 'Use a blueprint'. The 'Author from scratch' box contains the text 'Start with a simple Hello World example.' The 'Use a blueprint' box contains the text 'Build a Lambda application from sample code a configuration presets for common use cases.'

Below the options is the 'Basic information' section. It includes a 'Function name' field with the value 'CSVtoParquetLambda' and a note: 'Enter a name that describes the purpose of your function. Use only letters, numbers, hyphens, or underscores with no spaces.' The 'Runtime' field is set to 'Python 3.8' with a note: 'Choose the language to use to write your function.'

The 'Permissions' section has a note: 'By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can custo'. Below this is a section titled 'Change default execution role' with a dropdown arrow. It includes an 'Execution role' section with a note: 'Choose a role that defines the permissions of your function. To create a custom role, go to the IAM console.' Three radio buttons are shown: 'Create a new role with basic Lambda permissions', 'Use an existing role' (selected), and 'Create a new role from AWS policy templates'. Below this is an 'Existing role' section with a note: 'Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload lo'. A dropdown menu shows the selected role: 'DataEngLambdaS3CWGlueRole'.

Figure 3.11 – Creating and configuring a Lambda function

5. Do not change any of the **advanced settings** and click **Create function**.
6. Click on **Layers** in the first box, and then click **Add a layer** in the second box.
7. Select **Custom layers**, and from the dropdown, select the AWS Data Wrangler layer you create in a previous step (such as `aawsDataWrangler210_python38`). Select the latest version and then click **Add**:

The screenshot shows the AWS Lambda console interface for the function `CSVtoParquetLambda`. The **Layers** section is expanded, displaying a table of configured layers. The table has four columns: **Merge order**, **Name**, **Layer version**, and **Version ARN**. One layer is listed with a merge order of 1, name `awsDataWrangler200_python38`, version 1, and ARN `arn:aws:lambda:us-east-2:515154026536:layer:awsDataWrangler200_python38:1`. Buttons for **Edit** and **Add a layer** are visible to the right of the table.

Merge order	Name	Layer version	Version ARN
1	<code>awsDataWrangler200_python38</code>	1	<code>arn:aws:lambda:us-east-2:515154026536:layer:awsDataWrangler200_python38:1</code>

Figure 3.12 – Adding an AWS Lambda layer to an AWS Lambda function

8. Click on your function name (such as `CSVtoParquetLambda`) in the first block, and then scroll down to the **Code Source** section. The following code can be downloaded from this book's GitHub repository. Make sure to replace any existing code in `lambda_function` with this code.

In the first few lines of code, we import `boto3` (the AWS Python SDK), `aws wrangler` (which is part of the AWS Data Wrangler library that we added as a Lambda layer), and a function from the `urllib` library called `unquote_plus`:

```
import boto3
import aws wrangler as wr
from urllib.parse import unquote_plus
```

We then define our main function, `lambda_handler`, which is called when the Lambda function is executed. The event data contains information such as the S3 object that was uploaded and was the cause of the trigger that ran this function. From this event data, we get the S3 bucket name and the object key. We also set the Glue catalog `db_name` and `table_name` based on the path of the object that was uploaded.

```
def lambda_handler(event, context):
    # Get the source bucket and object name as passed to
    # the Lambda function
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        key = unquote_plus(record['s3']['object']['key'])

        # We will set the DB and table name based on the last
        # two elements of
        # the path prior to the file name. If key = 'dms/
        # sakila/film/LOAD01.csv',
        # then the following lines will set db to sakila and
        # table_name to 'film'
        key_list = key.split("/")
        print(f'key_list: {key_list}')
        db_name = key_list[len(key_list)-3]
        table_name = key_list[len(key_list)-2]
```


We now print out some debug type information that will be captured in our Lambda function logs. This includes information such as the Amazon S3 bucket and key that we are processing. We then set the `output_path` value here, which is where we are going to write the Parquet file that this function creates. Make sure to change the `output_path` value of this code to match the name of the target S3 bucket you created earlier:

```
print(f'Bucket: {bucket}')
```

```
print(f'Key: {key}')
```

```
print(f'DB Name: {db_name}')
```

```
print(f'Table Name: {table_name}')
```

```
input_path = f"s3://{bucket}/{key}"
```

```
print(f'Input_Path: {input_path}')
```

```
output_path = f"s3://dataeng-clean-zone-INITIALS/{db_
```

```
name}/{table_name}"
```

```
print(f'Output_Path: {output_path}')
```

We can then use the AWS Data Wrangler library (defined as `wr` in our function) to read the CSV file that we received. We read the contents of the CSV file into a pandas DataFrame we are calling `input_df`. We also get a list of current Glue databases, and if the database we want to use does not exist, we create it:

```
input_df = wr.s3.read_csv([input_path])
```

```
current_databases = wr.catalog.databases()
```

```
wr.catalog.databases()
```

```
if db_name not in current_databases.values:
```

```
    print(f'- Database {db_name} does not exist ...
```

```
    creating')
```

```
    wr.catalog.create_database(db_name)
```

```
else:
```

```
    print(f'- Database {db_name} already exists')
```

Finally, we can use the AWS Data Wrangler library to create a Parquet file containing the data we read from the CSV file. For the S3 to Parquet function, we specify the name of the dataframe (`input_df`) that contains the data we want to write out in Parquet format. We also specify the S3 output path, the Glue database, and the table name:

```
result = wr.s3.to_parquet(  
    df=input_df,  
    path=output_path,  
    dataset=True,  
    database=db_name,  
    table=table_name,  
    mode="append")  
  
print("RESULT: ")  
print(f'{result}')  
  
return result
```

9. Click on **Deploy**.
10. Click on the **Configuration** tab, and on the left-hand side click on **General configuration**. Click the **Edit** button and modify the **Timeout** to be 1 minute (the default timeout of 3 seconds is likely to be too low to convert some files from CSV to Parquet format).

Configuring our Lambda function to be triggered by an S3 upload


Our final task is to configure the Lambda function so that whenever a CSV file is uploaded to our landing zone bucket, the Lambda function runs and converts the file to Parquet format:

1. In the **Function Overview** box of our Lambda function, click on **Add trigger**.
2. For **Trigger configuration**, select the **Amazon S3** service from the drop-down list.
3. For **Bucket**, select your landing zone bucket (for example, `dataeng-landing-zone-<initials>`).

4. We want our rule to trigger whenever a new file is created in this bucket, no matter what method is used to create it (**Put, Post, or Copy**), so select **All object create events** from the list.
5. For **suffix**, enter `.csv`. This will configure the trigger to only run the Lambda function when a file with a `.csv` extension is uploaded to our landing-zone bucket.
6. Acknowledge the warning about **Recursive invocation** that can happen if you set up a trigger on a specific bucket to run a Lambda function, and then you get your Lambda function to create a new file in that same bucket and path. This is a good time to double-check and make sure that you are configuring this trigger on the `LANDING_ZONE` bucket (for example, `dataeng-landing-zone-<initials>`) and not the target `CLEAN_ZONE` bucket that our Lambda function will write to:

Add trigger

Trigger configuration

 **S3**
aws storage

Bucket
Please select the S3 bucket that serves as the event source. The bucket must be in the same region as the function.

dataeng-landing-zone-
▼
↻

Event type
Select the events that you want to have trigger the Lambda function. You can optionally set up a prefix or suffix for an event. However, for each bucket, individual events cannot have multiple configurations with overlapping prefixes or suffixes that could match the same object key.

All object create events
▼


Prefix - optional
Enter a single optional prefix to limit the notifications to objects with keys that start with matching characters.

e.g. images/

Suffix - optional
Enter a single optional suffix to limit the notifications to objects with keys that end with matching characters.

`.csv`

Lambda will add the necessary permissions for Amazon S3 to invoke your Lambda function from this trigger. [Learn more](#) about the Lambda permissions model.

 **Recursive invocation**
If your function writes objects to an S3 bucket, ensure that you are using different S3 buckets for input and output. Writing to the same bucket increases the risk of creating a recursive invocation, which can result in increased Lambda usage and increased costs. [Learn more](#)

I acknowledge that using the same S3 bucket for both input and output is not recommended and that this configuration can cause recursive invocations, increased Lambda usage, and increased costs.

Cancel
Add

Figure 3.13 – Configuring an S3-based trigger for an AWS Lambda function

7. Click **Add** to create the trigger.
8. Create a simple CSV file called `test.csv` that you can use to test the trigger. Ensure that the first line has column headings, as per the following example:

```
Name, favorite_num
Gareth, 23
Tracy, 28
Chris, 16
Emma, 14
```

Ensure you create the file with a standard text editor, and not Word processing software (such as Microsoft Word) or any other software that will add additional formatting to the file.

9. Upload your test file to your source S3 bucket by running the following on the command line. Make sure to replace `dataeng-landing-zone-initials` with the name of the source bucket you created previously:

```
aws s3 cp test.csv s3://dataeng-landing-zone-initials/
testdb/csvparquet/test.csv
```

10. If everything has been configured correctly, your Lambda function will have been triggered and will have written out a Parquet-formatted file to your target S3 bucket and created a Glue database and table. You can access the Glue service in the AWS Management Console to ensure that a new database and table have been created and can run the following command at the command prompt to ensure that a Parquet file has been written to your target bucket. Make sure to replace `dataeng-clean-zone-initials` with the name of your target S3 bucket:

```
aws s3 ls s3://dataeng-clean-zone-initials/testdb/
csvparquet/
```

The result of this command should display the Parquet file that was created by the Lambda function.

Summary

In this chapter, we covered a lot! We reviewed a range of AWS services at a high level, including services for ingesting data from a variety of sources, services for transforming data, and services for consuming and working with data.

We then got hands-on, building a solution in our AWS account that converted a file from CSV format to Parquet format and registered the data in the AWS Glue Data Catalog.

In the next chapter, we cover a really important topic that all data engineers need to have a good understanding of and that needs to be central to every project that a data engineer works on, and that is security and governance.

4

Data Cataloging, Security, and Governance

There is probably no more important topic to cover in a book that deals with data than data security and governance (and the related topic of data cataloging). Having the most efficient data pipelines, the fastest data transformations, and the best data consumption tools is not worth much if the data is not kept secure. Also, data storage must comply with local laws for how the data should be handled, and the data needs to be cataloged so that it is discoverable and useful to the organization.

Sadly, it is not uncommon to read about data breaches and poor data handling by organizations, and the consequences of this can include reputational damage to the organization, as well as potentially massive penalties imposed by the government.

In this chapter, we will do a deeper dive into the important considerations around best practices for handling data responsibly. We will cover the following topics:

- Getting data security and governance right
- Cataloging your data to avoid the data swamp
- The AWS Glue/Lake Formation data catalog

- AWS services for data encryption and security monitoring
- AWS services for managing identity and permissions
- Hands-on – configuring Lake Formation permissions

Technical requirements

To complete the hands-on exercises included in this chapter, you will need an AWS account where you have access to a user with administrator privileges (as covered in *Chapter 1, An Introduction to Data Engineering*).

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/tree/main/Chapter04>

Getting data security and governance right

Data security dictates how an organization should protect data to ensure that data is stored securely (such as in an encrypted state) and that access by unauthorized entities is prevented. For example, all the things an organization does to prevent falling victim to a ransomware attack, or having their data stolen and sold on the dark web, falls under data security.

Data governance, on the other hand, is related to ensuring that only people that need access to specific datasets have that access (such as ensuring that data is not just generally open to all users of a system without considering whether they need access to that data to perform their job). Governance also applies to ensuring that an organization only uses and processes data on individuals in approved ways and that organizations provide data disclosures as required by law.

Not providing adequate protection and security of an organization's data, or not complying with relevant governance laws, can end up being a very expensive mistake for an organization.

According to an article on CSO Online titled *The biggest data breach fines, penalties, and settlements so far* (<https://www.csoonline.com/article/3410278/the-biggest-data-breach-fines-penalties-and-settlements-so-far.html>), penalties and expenses related to data breaches have cost companies over \$1.3 billion.

For example, Equifax, the credit agency firm, had a data breach in 2017 that exposed the personal and financial information of nearly 150 million people. As a result, Equifax agreed to pay at least \$575 million in a settlement with several United States government agencies, and U.S. States.

But beyond financial penalties, a data breach can also do incalculable damage to an organization's reputation and brand. Once you lose the trust of your customers, it can be very difficult to earn that trust back.

Beyond data breaches where personal data is stolen from an organization's system, failure to comply with local regulations can also be costly. There are an increasing number of laws that define under what conditions a company may collect, store, and process personal information. Not complying with these laws can result in significant penalties for an organization, even in the absence of a data breach.

For example, Google was hit with a fine of more than \$50 million for failing to adequately comply with aspects of a European regulation known as the **General Data Protection Regulation (GDPR)**. Google appealed the decision, but in 2020, the decision was upheld by the courts, leaving the penalty on Google in place.

Common data regulatory requirements

No matter where you operate in the world, there are very likely several regulations concerning data privacy and protection that you need to be aware of, and plan for, as a data engineer. A small selection of these include the following:

- The **General Data Protection Regulation (GDPR)** in the European Union
- The existing **California Consumer Privacy Act (CCPA)** and the recently passed **California Privacy Rights Act (CPRA)** in California, USA
- The **Personal Data Protection Bill (PDP Bill)** in India
- The **Protection of Personal Information Act (POPIA)** in South Africa

These laws can be complex and cover many different areas, which is far beyond the scope of this book. However, generally, they involve individuals having the right to know what data a company holds about them; ensuring adequate protection of personal information that the organization holds; enforcing strict controls around data being processed; and in some cases, the right of an individual to request their data being deleted from an organization's system.

In the case of GDPR, an organization is subject to the regulations if they hold data on any resident of the European Union, even if the organization does not have a legal presence in the EU.

In addition to these broad data protection and privacy regulations, many regulations apply additional requirements to specific industries or functions. Let's take a look at some examples:

- The **Health Insurance Portability and Accountability Act (HIPAA)**, which applies to organizations that store an individual's healthcare and medical data
- The **Payment Card Industry Data Security Standard (PCI DSS)**, which applies to organizations that store and process credit card data

Understanding what these regulations require and how best to comply with them is often complex and time-consuming. While, in this chapter, we will look at general principles that can be applied to protect data used in analytic pipelines, this chapter is not intended as a guide on how to comply with any specific regulation.

GDPR specifies that in certain cases, an organization must appoint a **Data Protection Officer (DPO)**. The DPO is responsible for training staff involved in data processing and conducting regular audits, among other responsibilities.

If your organization has a DPO, ensure you set up a time to meet with the DPO to fully understand the regulations that may apply to your organization and how this may affect analytic data. Alternatively, work with your **Chief Information Security Officer (CISO)** to ensure your organization seeks legal advice on which data regulations may apply.

If you must participate in a compliance audit for an analytic workload running in AWS, review the *AWS Artifact* (<https://aws.amazon.com/artifact/>) service, a self-service portal for on-demand access to AWS's compliance reports.

Core data protection concepts

There are several concepts and terminology related to protecting data that are important for a data engineer to understand. In this section, we will briefly define some of these.

Personally identifiable information (PII)

Personally identifiable information (PII) is a term commonly used in North America to reference any information that can be used to identify an individual. This can refer to either the information on its own being able to identify an individual or where the information can be combined with other linkable information to identify an individual. It includes information such as full name, social security number, IP address, and photos or videos.

PII also covers data that provides information about a specific aspect of an individual (such as a medical condition, location, or political affiliation).

Personal data

Personal data is a term that is defined in GDPR and is considered to be similar to, but broader than, the definition of PII. Specifically, GDPR defines personal data as follows:

"Any information relating to an identified or identifiable natural person ("data subject"); an identifiable natural person is one who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person."

GDPR, Article 4, Definitions (<https://eur-lex.europa.eu/legal-content/EN/TXT/HTML/?uri=CELEX:32016R0679#d1e1374-1-1>).

Encryption

Encryption is a mathematical technique of encoding data using a key in such a way that the data becomes unrecognizable and unusable. An authorized user who has the key used to encrypt the data can use the key to decrypt the data and return it to its original plaintext form.

Encrypted data may be able to be decrypted by a hacker without the key through the use of advanced computational resources, skills, and time. However, a well-designed and secure encryption algorithm increases the difficulty of decrypting the data without the key, increasing the security of the encrypted data.

There are two important types of encryption and both should be used for all data and systems:

- **Encryption in transit:** This is the process of encrypting data as it moves between systems. For example, a system that migrates data from a database to a data lake should ensure that the data is encrypted before being transmitted, that the source and target endpoints are authenticated, and the data can then be decrypted at the target for processing. This helps ensure that if someone can intercept the data stream during transmission, that the data is encrypted and therefore unable to be read and used by the person who intercepted the data. A common way to achieve this is to use the **Transport Layer Security (TLS)** protocol for all communications between systems.
- **Encryption at rest:** This is the encryption of data that is written to a storage medium, such as a disk. After each phase of data processing, all the data that is persisted to disk should be encrypted.

Encryption (in transit and at rest) is a key tool for improving the security of your data, but other important tools should also be considered, as covered in the subsequent sections.

Anonymized data

Anonymized data is data that has been altered in such a way that personal data is irreversibly de-identified, rendering it impossible for any PII data to be identified. For example, this could involve replacing PII data with randomly generated data, in such a way that the randomization cannot be reversed to recreate the original data.

Another way anonymization can be applied is to remove most of the PII data so that only a few attributes that may be considered PII remains, but with enough PII data removed to make it difficult to identify an individual. However, this contains risk, as it is often still possible to identify an individual even with only minimal data. A well-known study (<https://dataprivacylab.org/projects/identifiability/paper1.pdf>) found that with just ZIP code, gender, and date of birth information, 87% of the population in the United States can be uniquely identified.

Pseudonymized data/tokenization

Pseudonymized data is data that has been altered in such a way that personal data is de-identified. While this is similar to the concept of anonymized data, the big difference is that with pseudonymized data, the original PII data can still be accessed.

Pseudonymized data is defined by GDPR as data that cannot be attributed to a specific data subject without the use of separately kept "*additional information*."

There are multiple techniques for creating pseudonymized data. For example, you can replace a full name with a randomly generated token, a different name (so that it looks real but is not), a hash representing the name, and more. However, whichever technique is used, it must be possible to still access the original data.

One of the most popular ways to do this is to have a tokenization system generate a unique, random token that replaces the PII data.

For example, when a raw dataset is ingested into the data lake, the first step may be to pass the data through the **tokenization** system. This system will replace all PII data in the dataset with an anonymous token, and will record each **real_data | token substitution** in a secure database. Once the data has been transformed, if a consumer requires access and is authorized to access the PII data, they can pass the dataset to the tokenization system to be detokenized (that is, have the tokens replaced with the original, real values).

The benefit of a tokenization system is that the generated token is random and does not contain any reference to the original value, and there is no way to determine the original value just from the token. If there is a data breach that can steal a dataset with tokenized data, there is no way to perform reverse engineering on the token to find the original value.

However, the tokenization system itself contains all the PII data, along with the associated tokens. If an entity can access the tokenized data and is also able to compromise the tokenization system, they will have access to all PII data. Therefore, it is important that the tokenization system is completely separate from the analytic systems containing the tokenized data, and that the tokenization system is protected properly.

On the other hand, hashing is generally considered the least secure method of de-identifying PII data, especially when it comes to data types with a limited set of values, such as social security numbers and names.

Hashing uses several popular hashing algorithms to create a hash of an original value. An original value, such as the name "*John Smith*," will always return the same hash value for a specific algorithm.

However, all possible social security numbers and most names have been passed through popular hashing algorithms and lookup tables have been created, known as rainbow tables. Using these rainbow tables, anyone can take a hashed name or social security number and quickly identify the original value.

For example, if you use the SHA-256 hashing algorithm, the original value of "*John Smith*" will always return "*ef61a579c907bbcd674c0dbcbcf7f7af8f851538eef7b8e58c5bee0b8cfdac4a*".

If you used the SHA-256 hashing algorithm to de-identify your PII data, it would be very easy for a malicious actor to determine that the preceding value referenced "*John Smith*" (just try Googling the preceding hash and see how quickly the name John Smith is revealed). While there are approaches to improving the security of a hash (such as salting the hash by adding a fixed string to the start of the value), it is still generally not recommended to use hashing for any data that has a well-known, limited set of values, or values that could be guessed.

Authentication

Authentication is the process of validating that a claimed identity is that identity.

A simple example is when you log in to a **Google Mail (Gmail)** account. You provide your identity (your Gmail email address) and then validate that it is you by providing something only you should know (your password), and possibly also a second factor of authentication (by entering the code that is texted to your cell phone).

Authentication does not specify what you can access but does attempt to validate that you are who you say you are. Of course, authentication systems are not foolproof. Your password may have been compromised on another website, and if you had the same password for your Gmail account, someone could use that to impersonate you. If you have **multi-factor authentication (MFA)** enabled, you receive a code on your phone or a physical MFA device that you need to enter when logging in, and that helps to further secure and validate your identity.

Federated identity is a concept related to authentication and means that responsibility for authenticating a user is done by another system. For example, when logging in to the AWS Management Console, your administrator could set up a federated identity so that you use your Active Directory credentials to log in via your organization's access portal, and the organization's Active Directory server authenticates you. Once authenticated, the Active Directory server confirms to the AWS Management Console that you have been successfully authenticated as a specific user. This means you do not need a separate username and password to log in to the AWS system, but that you can use your existing Active Directory credentials to be authenticated to an identity in AWS.

Authorization

Authorization is the process of authorizing access to a resource based on a validated identity. For example, when you log in to your Google account (where you are authenticated by your password, and perhaps a second factor such as a code that is texted to your phone), you may be authorized to access that identity's email, and perhaps also the Google Calendar and Google Search history for that identity.

For a data analytics system, once you validate your identity with authentication, you need to be authorized to access specific datasets. A data lake administrator can, for example, authorize you to access data that is in the Conformed Zone of the data lake, but not grant you access to data in the Raw Zone.

Putting these concepts together

Getting data protection and governance right does not happen by itself. It is important that you plan for and thoughtfully execute the process of protecting and governing your data. This will involve using some of the concepts introduced previously, such as the following:

- Making sure PII data is replaced with a token as the first processing step after ingestion (and ensuring that the tokenization system is secure).
- Encrypting all data at rest with a well-known and reliable encryption algorithm and ensuring that all connections use secure encrypted tunnels (such as by using the TLS protocol for all communications between systems).

- Implementing federated identities where user authorization for analytic systems is performed via a central corporate identity provider, such as Active Directory. This ensures that, for example, when a user leaves the company and their Active Directory account is terminated, their access to analytic systems in AWS is terminated as a result.
- Implementing least privilege access, where users are authorized for the minimum level of permissions that they need to perform their job.

This is also not something that a data engineer should do in isolation. You should work with your organization's security and governance teams to ensure you understand any legal requirements for how to process and secure your analytical data. You should also regularly review, or audit, the security policies in place for your analytic systems and data.

Cataloging your data to avoid the data swamp

Even if you do protect your data correctly and handle it as required by local regulations, if you do not make it easy for your users to find your analytic datasets and understand more about those datasets, your analytic data can become a liability.

You have probably heard about swamps, even if you have never actually been to one. Generally, swamps are known to be wet areas that smell pretty bad, and where some trees and other vegetation may grow, but the area is generally not fit to be used for most purposes (unless, of course, you're an ogre similar to Shrek, and you make your home in the swamp!).

In contrast to a swamp, when most people think about a lake, they picture beautiful scenery with clean water, a beautiful sunset, and perhaps a few ducks gently floating on the water. Most people would hate to find themselves in a swamp if they thought they were going to visit a beautiful lake.

In the world of data lakes, as a data engineer, you want to provide an experience that is much like the pure and peaceful lake described previously, and you want to avoid your users finding that the lake looks more like a swamp. However, if you're not careful, your data lake can become a data swamp, where there are lots of different pieces of data around, but no one is sure what data is there. Then, when they do happen to find some data, they don't know where the data came from or whether it can be trusted. Ultimately, a data swamp can be a dumping ground for data that is not of much use to anyone.

How to avoid the data swamp

With some careful upfront planning and the right tools and policies, it is possible to avoid the data swamp and instead offer your users a well-structured, easy-to-navigate data lake.

Avoiding the data swamp is easy in theory – you just need two important things:

- A data catalog that can be used to keep a searchable record of all the datasets in the data lake
- Policies that ensure useful metadata is added to all the entries in the data catalog

While that may sound pretty straightforward, the implementation details matter and things are not always as simple in real life. You need to have well-structured policies to ensure all datasets are cataloged, and that a defined set of metadata is always captured along with those datasets. Ensuring this is successfully enforced will often require the buy-in of senior leadership in the organization.

Data catalogs

A data catalog enables business users to easily find datasets that may be useful to them, and to better understand the context around the dataset through metadata.

Broadly speaking, there are two types of data catalogs – business catalogs and technical catalogs. However, many catalog tools offer aspects of both business and technical catalogs.

Technical catalogs are those that map data files in the data lake to a logical representation of those files in the form of databases and tables. In *Chapter 3, The AWS Data Engineers Toolkit*, we covered the AWS Glue service, which is an example of a data catalog tool with a technical focus.

The Hive Metastore is a well-known catalog that stores technical metadata for Hive tables (such as the table schema, location, and partition information). These are primarily technical attributes of the table, and the AWS Glue data catalog is an example of a Hive-compatible Metastore (meaning analytic services designed to work with a Hive Metastore catalog can use the Glue catalog).

A technical catalog enables an analytic service to understand the schema of the dataset (the physical location of the files that make up the dataset, the columns in a dataset, and the data type for each column, for example). This enables the analytic service to run queries against the data.

In contrast to technical catalogs, some tools are designed primarily as a business catalog. A business catalog focuses on enabling business metadata regarding the datasets to be captured and providing a catalog that is easy to search.

For example, with a business catalog, you may capture details about the following:

- The owner of the dataset
- The business unit that the data relates to
- The source system(s) that the data comes from
- The confidentiality classification of the data (sensitive, confidential, PII, and so on)
- How often the data is updated (hourly, daily, or weekly)
- How this dataset is related to other datasets

Most catalog tools offer a combination of both business and technical catalog functionality, although generally, they focus on one aspect more than the other. For example, the Glue data catalog is a Hive Metastore-compatible catalog that captures information about the underlying physical files and partitions. However, the Glue catalog is also able to capture other properties of the data. For example, the Glue catalog can capture key/values about a table, and this can be used to record the data owner, whether the table contains PII data, and more.

Popular data catalog solutions outside of AWS include the **Collibra Data Catalog** and the **Informatica Enterprise Data Catalog**.

Organizational policies for capturing metadata

While a catalog provides the ability to capture technical and business metadata about the data, it is up to the organization to enforce policies that ensure the right details are captured about each dataset.

For example, a policy needs to be enforced that ensures that all the data that is added to the data lake is captured in the data catalog. If data is added to the data lake and not captured in the catalog, you can very quickly end up with a data swamp – lots of data in the data lake but users are unable to find or understand the context of the data that is there.

If the technical data is captured in the data catalog but there is no policy to enforce the capture of business data, you can still end up with a data swamp. If you have hundreds of datasets in the data catalog with technical data but no business context, then it is difficult for users to get value from the data. Users will not have any information about the source of the data or the details of the dataset owner for them to reach out to with additional questions.

Ultimately, you want to have a catalog that users can search to find datasets, and then have users be able to examine the metadata to understand the business context of the data.

The AWS Glue/Lake Formation data catalog

As discussed previously, the AWS Glue catalog is a technical data catalog that can capture some business attributes using key/value tags. For example, you can have a key called `data_owner` and an associated value as a tag on each table in the catalog.

Within AWS, there are two services for interacting with the data catalog. So far, we have only discussed the AWS Glue service, but the AWS Lake Formation service also provides an interface for the same catalog.

It is important to understand that there is only a single data catalog, but that both Glue and Lake Formation provide an interface to the catalog. For example, if you set `zone : curated` as a table property on the `film_category` table in `curatedzoneadb` using the Glue console, you will see that same property set when viewing the table using Lake Formation.

Here, we can see the table details for the `film_category` table in the AWS Glue console, and we can see that one of the tags on this dataset is `zone : curated`:

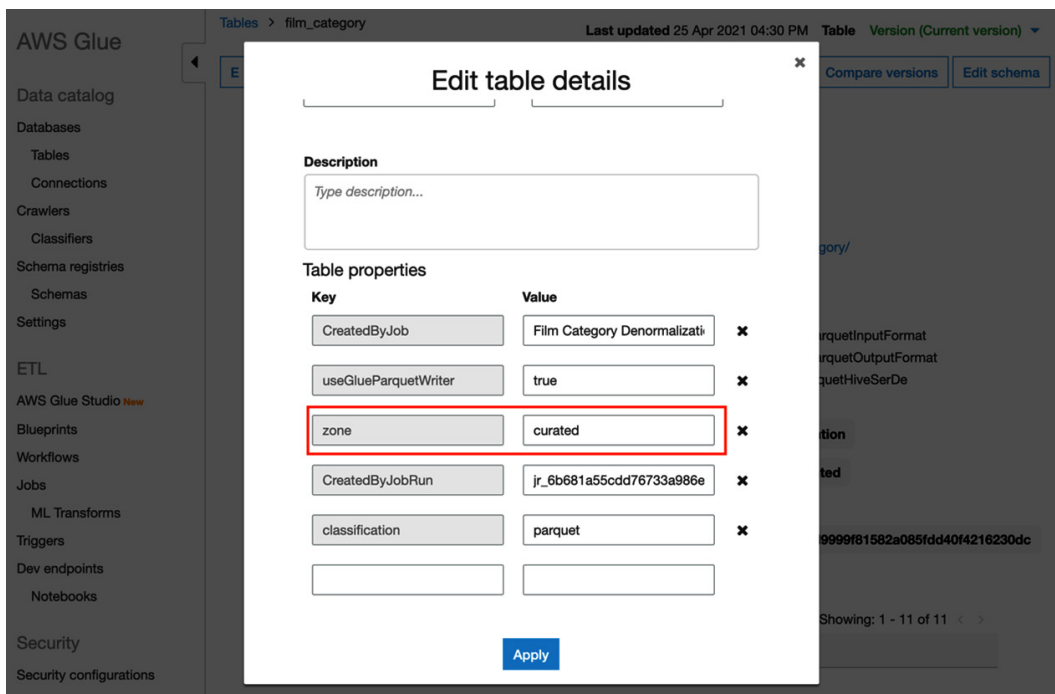


Figure 4.1 – AWS Glue console showing table properties

If we look at the same table in the Lake Formation catalog, we can also see that the `zone : curated` table property is shown:

The screenshot shows the AWS Lake Formation console interface for a table named 'film_category'. The breadcrumb navigation indicates the path: AWS Lake Formation > Tables > film_category. The table is currently at Version 1 (Current version). Action buttons include 'Actions', 'Compare versions', 'Drop table', and 'View properties'. The 'Table details' section is expanded, showing the following information:

Property	Value
Database	curatedzone
Location	s3://dataeng-curved-zone-gse89/film/film_category/
Description	-
Connection	-
Data format	parquet
Last updated	Sun, Apr 25, 2021, 8:30 PM UTC
Input format	org.apache.hadoop.hive.qi.io.parquet.MapredParquetInputFormat
Output format	org.apache.hadoop.hive.qi.io.parquet.MapredParquetOutputFormat
Serialization lib	org.apache.hadoop.hive.qi.io.parquet.serde.ParquetHiveSerDe
Serde parameters	-
Table properties	<p>CreatedByJob: Film Category Denormalization</p> <p>CreatedByJobRun: jr_6b681a55cdd76733a986e8762b99c259b0d9999f81582a085fdd40f4216230dc</p> <p>classification: parquet useGlueParquetWriter: true zone: curated</p>

The 'zone' and 'curated' properties in the table properties section are highlighted with a red box. The footer of the console includes 'Feedback', 'English (US)', 'Privacy Policy', 'Terms of Use', and 'Cookie preferences'.

Figure 4.2 – AWS Lake Formation console showing table properties

The Lake Formation console does provide a more modern design, and also provides some additional functionality that is not possible with the Glue interface. This includes the following:

- The ability to add key/value properties at the column level (with AWS Glue, you can only add properties at the table level)
- The ability to configure access permissions at the database, table, and column level (more on this later in this chapter)

The data catalog can be referenced by various analytical tools to work with data in the data lake. For example, Amazon Athena can reference the data catalog to enable users to run queries against databases and tables in the catalog. Athena uses the catalog to get the following information, which is required to query data in the data lake:

- The Amazon S3 location where the underlying data files are stored
- Metadata that indicates the file format type for the underlying files (such as CSV or Parquet)
- Details of the serialization library, which should be used to serialize the underlying data

- Metadata that provides information about the data type for each column in the dataset
- Information about any partitions that are used for the dataset

A data engineer must help put automation in place to ensure that all the datasets that are added to a data lake are cataloged and that the appropriate metadata is added.

In *Chapter 3, The AWS Data Engineers Toolkit*, we discussed AWS Glue Crawlers, a process that can be run to examine a data source, infer the schema of the data source, and then automatically populate the Glue data catalog with information on the dataset.

A data engineer should consider building workflows that make use of Glue Crawlers to run after new data is ingested, to have the new data automatically added to the data catalog. Or, when a new data engineering job is being brought into production, a check can be put in place to make sure that the Glue API is used to update the data catalog with details of the new data.

Automated methods should also be used to ensure that relevant metadata is added to the catalog whenever new data is created, such as by putting a method in place to ensure that the following metadata is added, along with all the new datasets:

- Data source
- Data owner
- Data sensitivity (public, general, sensitive, confidential, PII, and so on)
- Data lake zone (raw zone, transformed zone, enriched zone)
- Cost allocation tag (business unit name, department, and so on)

Putting this type of automation in place helps ensure that you continue to build a data lake without inadvertently letting the data lake become a data swamp.

AWS services for data encryption and security monitoring

Previously, we discussed common data protection concepts, such as data encryption. Now, we will look at some of the AWS services that can be used to help protect and secure our data.

AWS Key Management Service (KMS)

AWS KMS simplifies the process of creating and managing security keys for encrypting and decrypting data in AWS. The AWS KMS service is a core service in the AWS ecosystem, enabling users to easily manage data encryption across several AWS services.

There are a large number of AWS services that can work with AWS KMS to enable data encryption, including the following AWS analytical services:

- **Amazon AppFlow**
- **Amazon Athena**
- **Amazon EMR**
- **Amazon Kinesis Data Streams/Kinesis Firehose/Kinesis Video Streams**
- **Amazon Managed Streaming for Kafka (MSK)**
- **Amazon Managed Workflows for Apache Airflow (MWAA)**
- **Amazon Redshift**
- **Amazon S3**
- **AWS Data Migration Service (DMS)**
- **AWS Glue/Glue DataBrew**
- **AWS Lambda**

The full list of compatible services can be found at https://aws.amazon.com/kms/features/#AWS_Service_Integration.

Permissions can be granted to users to make use of the keys for encrypting and decrypting data, and all use of AWS KMS keys is logged in the AWS CloudTrail service. This enables an organization to easily audit the use of keys to encrypt and decrypt data.

For example, with Amazon S3, you can enable Amazon S3 Bucket Keys, which configures an S3 Bucket Key to encrypt all new objects in the bucket with an AWS KMS Key. This is significantly less expensive than using **Server Side Encryption – KMS (SSE-KMS)** to encrypt each object in a bucket with a unique key.

To learn more about configuring Amazon S3 Bucket Keys, see <https://docs.aws.amazon.com/AmazonS3/latest/userguide/bucket-key.html>.

It is important that you carefully protect your KMS keys and that you put safeguards in place to prevent a KMS key from being accidentally (or maliciously) deleted. If a KMS key is deleted, any data that has been encrypted with that key is effectively lost and cannot be decrypted.

Because of this, you must schedule the deletion of your KMS keys and specify a waiting period of between 7 and 30 days before the key is deleted. During this waiting period, the key cannot be used, and you can configure a CloudWatch alarm to notify you if anyone attempts to use the key.

If you use AWS Organizations to manage multiple AWS accounts as part of an organization, you can create a **Service Control Policy (SCP)** to prevent any user (even an administrative user) from deleting KMS keys in child accounts.

Amazon Macie

Amazon Macie is a managed service that uses machine learning, along with pattern matching, to discover and protect sensitive data. Amazon Macie identifies sensitive data, such as PII data, in an Amazon S3 bucket and provides alerts to warn administrators about the presence of such sensitive data. Macie can also be configured to launch an automated response to the discovery of sensitive data, such as a step function that runs to automatically remediate the potential security risk.

Macie can identify items such as names, addresses, and credit card numbers that exist in files on S3. These items are generally considered to be PII data, and as discussed previously, these should ideally be tokenized before data processing. Macie can also be configured to recognize custom sensitive data types to alert the user on sensitive data that may be unique to a specific use case.

Amazon GuardDuty

While **Amazon GuardDuty** is not directly related to analytics on AWS, it is a powerful service that helps protect an AWS account. GuardDuty is an intelligent threat detection service that uses machine learning to monitor your AWS account and provide proactive alerts about malicious activity and unauthorized behavior.

GuardDuty analyzes several AWS generated logs, including the following:

- CloudTrail S3 data events (a record of all actions taken on S3 objects)
- CloudTrail management events (a record of all usage of AWS APIs within an account)
- VPC flow logs (a record of all network traffic within an AWS VPC)
- DNS logs (a record of all DNS requests within your account)

By continually analyzing these logs to identify unusual access patterns or data access, Amazon GuardDuty can proactively alert you to potential issues, and also helps you automate your response to threats.

AWS services for managing identity and permissions

We previously defined authentication as the process of validating that a claimed identity is that identity, and authorization as the process of authorizing access to a resource, based on a validated identity.

Within AWS, there are several ways to authenticate an identity, and for analytics on AWS, there are two primary ways to manage which identities can access which resources.

AWS Identity and Access Management (IAM) service

AWS IAM is a service that provides both authentication and authorization for the AWS Console, **command-line interface (CLI)**, and **application programming interface (API)** calls.

AWS IAM also supports a federation of identities, meaning that you can configure IAM to use another identity provider for authentication, such as Active Directory or Okta.

Note that this section is not intended as a comprehensive guide to Identity and Access Management on AWS, but it does provide information on foundational concepts that are important for anyone working within the AWS cloud to understand. For a deeper understanding of the AWS IAM service, refer to the *AWS Identity and Access Management user guide* (<https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html>).

Several IAM identities are important to understand:

- **AWS account root user:** When you create an AWS account, you provide an email address to be associated with that account, and that email address becomes the root user of the account. You can log in to the AWS Management Console using the root user, and this user has full access to all the resources in the account. However, it is strongly recommended that you do not use this identity to log in and perform everyday tasks, but rather create an IAM user for everyday use.
- **IAM User:** This is an identity that you create and can be used to log in to the AWS Console, run CLI commands, or make API calls. An IAM user has a login name and password that's used for Console access and can have up to two associated access keys that can be used to authenticate this identity when using the AWS CLI or API. While you can associate IAM policies directly with an IAM user, the recommended method to provide access to AWS resources is to make the user part of a group that has relevant IAM policies attached.

- **IAM User Groups:** An IAM group is used to provide permissions that can be associated with multiple IAM users. You provide permissions (via IAM policies) to an IAM group, and all the members of that group then inherit those permissions.
- **IAM roles:** An IAM role can be confusing at first as it is similar to an IAM user. However, an IAM role does not have a username or password and you cannot directly log in or identify as an IAM role. However, an IAM user can assume the identity of an IAM role, taking on the permissions assigned to that role. An IAM Role is also used in identity federation, where a user is authenticated by an external system, and that user identity is then associated with an IAM role. Finally, an IAM role can also be used to provide permissions to AWS resources (for example, to provide permissions to an AWS Lambda function so that the Lambda function can access specific AWS resources).

To grant authorization to access AWS resources, you can attach an **IAM policy** to an IAM user, IAM group, or IAM role. These policies grant, or deny, access to specific AWS resources, and can also make use of conditional statements to further control access.

These identity-based policies are JSON documents that specify the details of access to an AWS resource. These policies can either be configured within the AWS Management Console, or the JSON documents can be created by hand.

There are three types of identity-based policies that can be utilized:

- **AWS managed policies:** These are policies that are created and managed by AWS and provide permissions for common use cases. For example, the `AdministratorAccess` managed policy provides full access to every service and resource in AWS, while the `DatabaseAdministrator` policy provides permissions for setting up, configuring, and maintaining databases in AWS.
- **Customer-managed policies:** These are policies that you create and manage to provide more precise control over your AWS resources. For example, you can create a policy and attach it to specific IAM users/groups/roles that provide access to a list of specific S3 buckets and limit that access to only be valid during specific hours of the day or for specific IP addresses.
- **Inline policies:** These are policies that are written directly for a specific user, group, or role. These policies are tied directly to the user, group, or role, and therefore apply to one specific entity only.

The following policy is an example of a customer-managed policy that grants read access to a specific S3 bucket:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": "arn:aws:s3::: de-landing-zone"
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject"
      ],
      "Resource": ["arn:aws:s3::: de-landing-zone/*"]
    }
  ]
}
```

The policy takes the form of a JSON document. In this instance, the policy does the following:

- Allow access (you can also create policies that Deny access).
- Allows access for action of `s3:GetObject` and `s3:ListBucket`, meaning authorization is given to run the Amazon S3 `GetBucket` and `ListBucket` actions (via the Console, CLI, or API).
- For `ListBucket`, the resource is set as the `de-landing-zone` bucket. For `GetObject`, the resource is set as `de-landing-zone/*`. This results in the principal being granted access to list the `de-landing-zone` bucket, and read access to all the objects inside the `de-landing-zone` bucket.

You could further limit this policy to only be allowed if the user was connecting from a specific IP address, at a certain time of day, or various other limitations. For example, to limit this permission to users from a specific IP address, you could add the following to the policy:

```
"Condition": {
  "IpAddress": {
    "aws:SourceIp": [
      "12.13.15.16/32",
      "45.44.43.42/32"
    ]
  }
}
```

Once you have created a customer-managed policy, you can attach the policy to specific IAM groups, IAM roles, or IAM users.

Traditional data lakes on AWS used IAM policies to control access to data in an Amazon S3-based data lake. For example, a policy would be created to grant access to different zones of the data lake, and then that policy would be attached to different IAM users, groups, or roles.

However, when creating a large data lake that may contain multiple buckets or S3 prefixes that relate to specific business units, it can be challenging to manage S3 permissions through these JSON policies. Each time a new data lake location is created, the data engineer would need to make sure that the JSON policy document was updated to configure permissions for the new location.

To make managing large S3-based data lakes easier, AWS introduced a new service called **AWS Lake Formation**, which enables permissions for the data lake to be controlled by the data lake administrator from within the AWS Management Console (or via the AWS CLI or AWS API).

Using AWS Lake Formation to manage data lake access

AWS Lake Formation is a service that simplifies setting up and managing a data lake. And a big part of the Lake Formation service is the ability to manage access (authorization) to data lake databases and tables without having to manage fine-grained access through JSON-based policy documents in the IAM service.

Lake Formation enables a data lake administrator to grant fine-grained permissions on data lake databases, tables, and columns using the familiar database concepts of grant and revoke for permissions management. A data lake administrator, for example, can grant SELECT permissions (effectively READ permission) for a specific data lake table to a specific IAM user or role.

Lake Formation permissions management is another layer of permissions that is useful for managing fine-grained access to data lake resources, but it works with IAM permissions and does not replace IAM permissions. A recommended way to do this is to apply broad permissions to a user in an IAM policy, but then apply fine-grained permissions with Lake Formation.

Permissions management before Lake Formation

Before the release of the Lake Formation service, all data lake permissions were managed at the Amazon S3 level using IAM policy documents written in JSON. These policies would control access to resources such as the following:

- The data catalog objects in the Glue data catalog (such as permissions to access Glue databases and tables)
- The underlying physical storage in Amazon S3 (such as the Parquet or CSV files in an Amazon S3 bucket)
- Access to analytical services (such as Amazon Athena or AWS Glue)

For example, the IAM policy would provide several Glue permissions, including the ability to read catalog objects (such as Glue tables and table partitions) and the ability to search tables. However, the resources section of the policy would restrict these permissions to the specific databases and tables that the user should have access to.

The policy would also have a section that provided permissions to the underlying S3 data. For each table that a user needed to access in the Glue data catalog, they would need both Glue data catalog permissions for the catalog objects, as well as Amazon S3 permissions for the underlying files.

The last part of the IAM policy would also require the user to have access to relevant analytical tools, such as permissions to access the Amazon Athena service.

Permissions management using AWS Lake Formation

With AWS Lake Formation, permissions management is changed so that broad access can be provided to Glue catalog objects in the IAM policy, and fine-grained access is controlled via AWS Lake Formation permissions.

With Lake Formation, data lake users do not need to be granted direct permissions on underlying S3 objects as the Lake Formation service can provide temporary credentials to compatible analytic services to access the S3 data.

It is important to note that Lake Formation permissions access only works with compatible analytic services, which, at the time of writing, includes the following AWS services:

- Amazon Athena
- Amazon QuickSight
- Apache Spark running on Amazon EMR
- Amazon Redshift Spectrum
- AWS Glue

If using these compatible services, AWS Lake Formation is a simpler way to manage permissions for your data lake. The data lake user still needs an associated IAM policy that grants them access to the AWS Glue service, the Lake Formation service, and any required analytic engines (such as Amazon Athena). However, at the IAM level, the user can be granted access to all AWS Glue objects. The Lake Formation permissions layer can then be used to control which specific Glue catalog objects can be accessed by the user.

As the Lake Formation service passes temporary credentials to compatible analytic services to read data from Amazon S3, data lake users no longer need any direct Amazon S3 permissions to be provided in their IAM policies.

Hands-on – configuring Lake Formation permissions

In this hands-on section, we will use the AWS Management Console to configure Lake Formation permissions.

However, before we implement Lake Formation permissions, we're going to create a new data lake user and configure their permissions using just IAM permissions. We'll then go through the process of updating a Glue database and table to use Lake Formation permissions, and then grant Lake Formation permissions to our data lake user.

Configuring the Glue Crawler

While not covered in this chapter, we will provide a hands-on section with details on how to configure the Glue crawler in *Chapter 6, Ingesting Batch and Streaming Data*.

Creating a new user with IAM permissions

To start, let's create a new IAM user that will become our data lake user. We will initially use IAM to grant our data lake user the following permissions:

- Permission to access a specific database and table in the Glue data catalog
- Permission to use the Amazon Athena service to run SQL queries against the data lake

First, let's create a new IAM policy that grants the required permissions for using Athena and Glue, but limits those permissions to only CleanZoneDB in the Glue catalog. To do this, we're going to copy the Amazon-managed policy for Athena Full Access, but we will modify the policy to limit access to just a specific Glue database, and we will add S3 permissions to the policy. Let's get started:

1. Log in to the AWS Management Console and access the IAM service using this link: <https://console.aws.amazon.com/iam/home>.
2. On the left-hand side, click on **Policies**, and then for **Filter Policies**, type in Athena.
3. From the filtered list of policies, expand the AmazonAthenaFullAccess policy.
4. Click inside the JSON policy box and copy the entire policy to your computer clipboard.

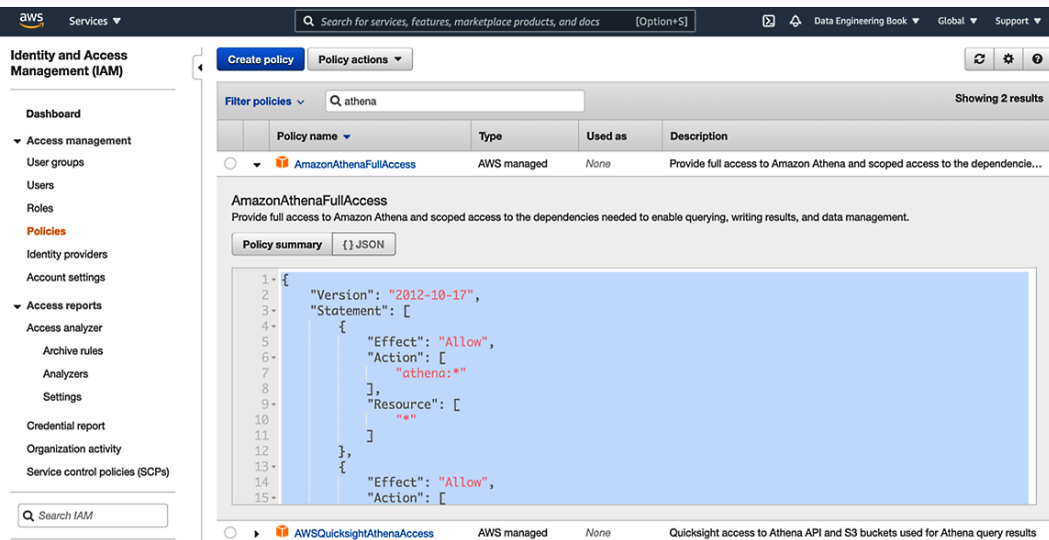


Figure 4.3 – Copying the text of the AmazonAthenaFullAccess policy

5. At the top of the page, click on **Create policy**.

6. The visual editor is selected by default, but since we want to create a JSON policy directly, click on the **JSON** tab.
7. Paste the Athena full access policy that you copied to the clipboard in *step 4* into the policy, overwriting and replacing any text currently in the policy.
8. Look through the policy to identify the section that grants permissions for several Glue actions (glue:CreateDatabase; glue>DeleteDatabase; glue:getDatabase, and so on). This section currently lists the resource that it applies to as *, meaning that the user would have access to all databases and tables in the Glue catalog. In our use case, we want to limit permissions to just the Glue CleanZoneDB database (which was created in the hands-on section of *Chapter 3, The AWS Data Engineers Toolkit*). Replace the resource section of the section that provides Glue access with the following, which will limit access to the required DB only, although it also includes all tables in that database:

```

"Resource": [
    "arn:aws:glue:*:*:catalog",
    "arn:aws:glue:*:*:database/cleanzonedb",
    "arn:aws:glue:*:*:database/cleanzonedb*",
    "arn:aws:glue:*:*:table/cleanzonedb/*"
]

```

The following screenshot shows how this looks when applied to the policy:

The screenshot shows the AWS IAM console's JSON editor. The 'JSON' tab is selected. The policy document is displayed with line numbers 28 through 48. The 'Resource' array is highlighted in yellow, showing the updated permissions for the Glue CleanZoneDB database and its tables. The status bar at the bottom indicates 0 Security, 0 Errors, 0 Warnings, and 1 Suggestion.

```

28     "glue:CreatePartition",
29     "glue>DeletePartition",
30     "glue:BatchDeletePartition",
31     "glue:UpdatePartition",
32     "glue:GetPartition",
33     "glue:GetPartitions",
34     "glue:BatchGetPartition"
35 ],
36 "Resource": [
37     "arn:aws:glue:*:*:catalog",
38     "arn:aws:glue:*:*:database/cleanzonedb",
39     "arn:aws:glue:*:*:database/cleanzonedb*",
40     "arn:aws:glue:*:*:table/cleanzonedb/*"
41 ]
42 },
43 {
44     "Effect": "Allow",
45     "Action": [
46         "s3:GetBucketLocation",
47         "s3:GetObject",
48         "s3:ListBucket"

```

Figure 4.4 – Updated policy with limited permissions for Glue resources

9. Immediately after the section that provides Glue permissions, we can add new permissions for accessing the S3 location where our CleanZoneDB data resides. Add the following section to provide these permissions, making sure to replace `<initials>` with the unique identifier you used when creating the bucket in *Chapter 3, The AWS Data Engineers Toolkit*:

```

    {
      "Effect": "Allow",
      "Action": [
        "s3:GetBucketLocation",
        "s3:GetObject",
        "s3:ListBucket",
        "s3:ListBucketMultipartUploads",
        "s3:ListMultipartUploadParts",
        "s3:AbortMultipartUpload",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::dataeng-clean-zone-
        <initials>/*"
      ]
    },

```

Here is a screenshot showing the S3 permissions added to the policy:



Figure 4.5 – S3 permissions added to the policy

10. Once you have pasted in the new S3 permissions, click on **Next:Tags** at the bottom right of the screen.
11. Optionally, add any tags for this policy, and then click on **Next: Review**.
12. For **Name**, provide a policy name of `AthenaAccessCleanZoneDB` and click **Create policy**.

Now that we have created an IAM policy for providing the required permissions to the Glue catalog and S3 buckets, we can create a new IAM user and attach our new policy to the new data lake user.

Follow these steps to create the new IAM user:

1. On the left-hand side, click on **Users**, and then click **Add user**.
2. For **User name**, enter `data-lake-user`.
3. For **Access type**, select **AWS Management Console access**.
4. For **Console password**, select **Custom password** and enter a secure password.
5. Clear the checkbox for **Require password reset** and then click **Next: Permissions**.
6. Select **Attach existing policies directly** and search for the policy you created in the previous step (`AthenaAccessCleanZoneDB`). Click the policy checkbox and then click **Next: Tags**.
7. Optionally, add any tags and click **Next: Review**.
8. Review the configuration and click **Create user**.
9. Click on **Close** to close the **Add user** dialog.

Now, let's create a new Amazon S3 bucket that we can use to capture the results of any Amazon Athena queries that we run:

1. In the AWS Management Console, use the top search bar to search for and select the **S3** service.
2. Click on **Create bucket**.
3. For **Bucket name**, enter `aws-athena-query-results-dataengbook-
<initials>`. Replace `<initials>` with your initials or some other unique identifier.
4. Ensure **AWS Region** is set to the region you have been using for the other exercises in this book.
5. Leave the others as their defaults and click on **Create bucket**.

We can now verify that our new `dataLake-user` only has access to CleanZoneDB and that the user can run Athena queries on the table in this database:

1. Sign out of the AWS Management Console, and then sign in again using the new user you just created, `dataLake-user`.
2. From the top search bar, search for and select the **Athena** service.
3. Before you can run an Athena query, you need to set up a query result location in Amazon S3. This is the S3 bucket and prefix where all the query results will be written to. From the top right of the Athena console, click on **Settings**.
4. For **Query result location**, enter the S3 path you created in *the previous Step 3* (for example, `s3://aws-athena-query-results-dataengbook-<initials>/`).
5. Click on **Save**.
6. In the **New Query** window, run the following SQL query: `select * from cleanzonedb.csvparquet`.
7. If all permissions have been configured correctly, the results of the query should be displayed in the lower window. The file we created shows names and ages.
8. Log out of the AWS Management Console since we need to be logged in as our regular user, not `dataLake-user`.

We have now set up permissions for our data lake using IAM policies to manage fine-grained access control, as was always done before the launch of the AWS Lake Formation service. In the next section, we will transition to using Lake Formation to manage fine-grained permissions on data lake objects.

Transitioning to managing fine-grained permissions with AWS Lake Formation

In the initial setup, we configured permissions for our data lake user to be able to run SQL queries using Amazon Athena, and we restricted their access to just `cleanzonedb` using an IAM permissions policy.

In this section, we are going to modify `cleanzonedb` and the tables in that database to make use of the Lake Formation permissions model.

Activating Lake Formation permissions for a database and table

As a reminder, Lake Formation adds a layer of permissions that work in addition to the IAM policy permissions. By default, every database and table in the catalog has a special permission enabled that effectively tells Lake Formation to just use IAM permissions and to ignore any permissions that may have been granted in Lake Formation. This is sometimes called the **Pass-Through** permission as it allows security checks to be validated at the IAM level, but then passes through Lake Formation without doing any additional permission checks.

With our initial setup, we granted Glue data catalog permissions to `data-lake-user` in an IAM policy. This policy allowed the user to access the `cleanzonedb` database, as well as all the tables in that database. Let's have a look at how permissions are set up on the `cleanzonedb` database and tables in Lake Formation:

1. Log in to **AWS Management Console** and search for the `Lake Formation` service in the top search bar. Make sure you are logged in as your regular user, and not as `data-lake-user`, which you created earlier in this chapter.
2. The first time you access the Lake Formation service, a pop-up box will prompt you to choose initial users and roles to be Lake Formation data lake administrators. By default, **Add myself** should be selected. Click **Get started** to add your current user as a data lake admin.

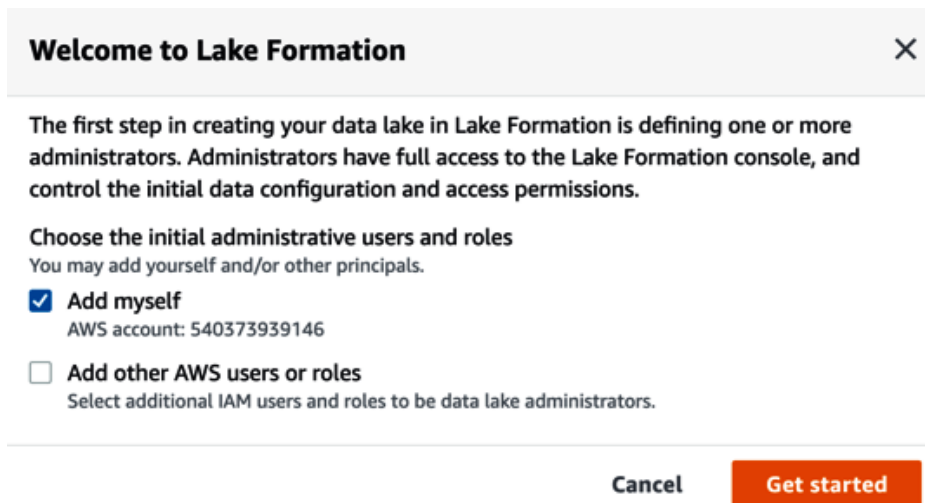


Figure 4.6 – Adding your user as a Lake Formation administrator

3. Once selected, you should be taken to the Lake Formation **Data lake administrators** screen, where you can confirm that your user has been added as a data lake administrator.

4. On the left-hand side of the Lake Formation console, click on **Databases**. In the list of databases, click on the `cleanzonedb` database.
5. This screen displays details of `cleanzonedb`. Click on **Actions**, and then **View permissions**.
6. On the **View permissions** screen, we can see that two permissions have been assigned for this database. The first one is `DataEngLambdaS3CWGlueRole`, and this IAM role has been granted full permissions on the database. The reason for this is that `DataEngLambdaS3CWGlueRole` was the role that was assigned to the Lambda function that we used to create the database back in *Chapter 3, The AWS Data Engineers Toolkit*, so it is automatically granted these permissions.

The screenshot shows the AWS Lake Formation console interface. On the left is a navigation sidebar with options like Dashboard, Data catalog, Databases, Tables, Settings, Register and ingest, Data lake locations, Blueprints, Crawlers, Jobs, Permissions, and Administrative roles and tasks. The main area is titled 'Data permissions (2)' and includes a search bar, filter buttons for 'Database: cleanzonedb' and 'Catalog ID: 540373939146', and 'Revoke' and 'Grant' buttons. Below this is a table listing the permissions:

Principal	Principal type	Resource type	Resource	Owner account ID	Permissions	Grantable	RAM Resource Share
<input type="radio"/> DataEngLambdaS3CWGlueRole	IAM role	Database	cleanzonedb	! [redacted] 6	Super, Alter, Create table, Describe, Drop	Super, Alter, Create table, Describe, Drop	-
<input type="radio"/> IAMAllowedPrincipals	Group	Database	cleanzonedb	! [redacted] 6	Super	-	-

Figure 4.7 – Lake Formation permissions for the `cleanzonedb` database

The other permission that we can see is for the `IAMAllowedPrincipals` group. This is the pass-through permission we mentioned previously, which effectively means that permissions at the Lake Formation layer are ignored. If this special permission was not assigned, only `DataEngLambdaS3CWGlueRole` would be able to access the database. However, because the permission has been assigned, any user who has been granted permissions to this database through an IAM policy, such as `datalake-user`, will be able to successfully access the database.

7. To enable Lake Formation permissions on this database, we can remove the `IAMAllowedPrincipals` permission from the database. To do this, click the selector box for the `IAMAllowedPrincipals` permission and click **Revoke**. On the pop-up box, click on **Revoke**.

Revoke permissions: cleanzonedb ✕
 Revoke access permissions to specific users and roles.

My account
 User or role from this AWS account.

External account
 AWS account or AWS organization outside of my account.

IAM users and roles
 Add one or more IAM users or roles.

Choose IAM principals to add ▼

IAMAllowedPrincipals ✕
 Group

SAML and Amazon QuickSight users and groups
 Enter a SAML user or group ARN or Amazon QuickSight ARN. Press Enter to add additional ARNs.

Ex: arn:aws:iam::<AccountId>;saml-provider/<SamlProviderName>;user/<UserName>

Database permissions
 Choose the access permissions to revoke. Access will be blocked even if IAM permissions are in place.

Create table
 Alter
 Drop
 Describe

Super
 Revoking this permission causes individual permissions on the operations above to go into effect, as well as disabling certain permissions logging in Cloudtrail. [See here](#)

Grantable permissions
 Choose the permissions that may not be granted to others.

Create table
 Alter
 Drop
 Describe

Super
 Revoking this permission causes individual grant permissions on the operations above to go into effect.

Cancel
Revoke

Figure 4.8 – Revoking the pass-through permission on cleanzonedb

8. We now want to do the same thing for our `CSVParquet` table in the database. To do this, click on **Databases** in the left-hand menu, then click on `cleanzonedb`. From the top right, click on **View tables**. Click the selector for the `CSVParquet` table and click on **Actions/View Permissions**. Click the selector for `IAMAllowedPrincipals` and click on **Revoke**. On the pop-up window, click on **Revoke**. This removes the special **Pass-Through** permission from the table.

Optional – checking permissions

If you want to see what effect this has, you can log out of the AWS Console and log in again as `dataLake-user`. Now, when you try to run a query on the `CSVParquet` table using Athena, you will receive an error message as Lake Formation permissions are in effect, and your `dataLake-user` has not been granted permissions to access the table yet.

Granting Lake Formation permissions

By removing the `IAMAllowedPrincipals` permission from the `cleanzonedb` database and the `CSVParquet` table, we have effectively enabled Lake Formation permissions on those resources. Now, if any principal needs to access that database or table, they need both IAM permissions, as well as Lake Formation permissions.

If we had enabled Lake Formation permissions on all databases and tables, then we could modify our user's IAM policy permissions to give them access to all data catalog objects. We can do this because we would know that they would only be able to access those databases and tables where they had been granted specific Lake Formation permissions.

We previously created an edited copy of the `AmazonAthenaFullAccess` managed IAM policy to limit user access to specific data catalog databases and tables in the IAM policy. However, if all databases and tables had the `IAMAllowedPrincipals` permission removed and specific permissions granted to users instead, then we could apply the generic `AmazonAthenaFullAccess` policy.

We also previously provided access to the underlying S3 files using an IAM policy. However, when using Lake Formation permissions, compatible analytic tools are granted access to the underlying S3 data using temporary credentials provided by Lake Formation. Therefore, once Lake Formation permissions have been activated, we can remove permissions to the underlying S3 data from our user's IAM policy. Then, when using a compatible tool such as Amazon Athena, we know that Lake Formation will grant Athena temporary credentials to access the underlying S3 data.

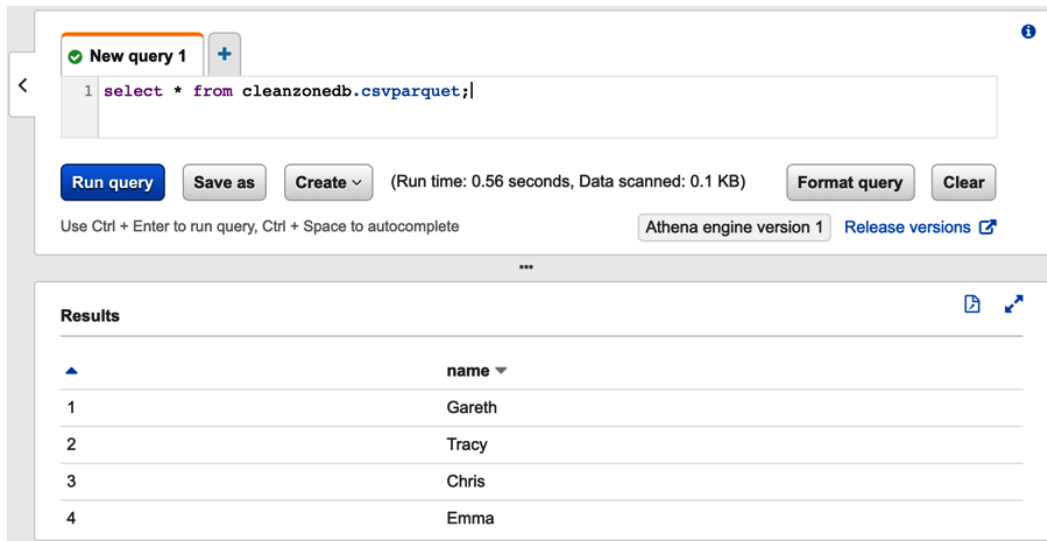
Here, we will add specific Lake Formation permissions for our `dataLake-user` to access the `CleanZoneDB` database and the `CSVParquet` table:

1. Ensure you are logged in as your regular user (the one you made a data lake admin earlier) and access the Lake Formation console.
2. Click on **CleanZoneDB**, and then click **View tables**.
3. Click on the **CSVParquet** table, and then click **Actions/Grant**.
4. From the **IAM users and roles** dropdown, click on the **dataLake-user** principal.

5. Under **Columns**, click on **Exclude columns** and then select **Age** as the column to exclude.
6. For **Table permissions**, mark the permission for **Select**.
7. Click on **Grant** at the bottom of the screen.

In the preceding steps, we granted our `dataLake-user` **Select** permissions on the **CSVParquet** table. However, we put in a column limitation, which means that `dataLake-user` will not be able to access the **Age** column. Enabling column-level permissions is not something that would be possible if we were just using IAM-level permissions, as column-level permissions is a Lake Formation-specific feature.

Now, if you log in to the AWS Management Console as `dataLake-user` and run the same Athena query we ran previously (`select * from cleanzonedb.csvparquet`), your permissions will enable the required access.



The screenshot displays the AWS Athena console interface. At the top, there is a query editor with the text "New query 1" and a plus sign. The query entered is "1 select * from cleanzonedb.csvparquet;". Below the query editor are several buttons: "Run query" (highlighted in blue), "Save as", "Create" (with a dropdown arrow), "Format query", and "Clear". To the right of the "Create" button, it shows "(Run time: 0.56 seconds, Data scanned: 0.1 KB)". Below the buttons, there is a note: "Use Ctrl + Enter to run query, Ctrl + Space to autocomplete". To the right, it says "Athena engine version 1" and "Release versions" with a link icon. Below the query editor is a "Results" section. It contains a table with one column labeled "name" and four rows of data: 1 Gareth, 2 Tracy, 3 Chris, and 4 Emma.

	name
1	Gareth
2	Tracy
3	Chris
4	Emma

Figure 4.9 – Running an Athena query with Lake Formation permissions

Note that in the results of the query, the **Age** column is not included as we specifically excluded this column when granting permissions on this table to our `dataLake-user`.

In this section, we transitioned to using Lake Formation for managing data lake permissions for the `cleanzonedb` database. We expanded IAM permissions to provide coarse-grained permissions to Glue catalog objects, but then added fine-grained permissions in Lake Formation to limit `cleanzonedb` access to just our `dataLake-user`.

Summary

In this chapter, we reviewed important concepts around data security and governance, including how a data catalog can be used to help prevent your data lake from becoming a data swamp.

Data encryption at rest and in transit, and tokenization of PII data, are important concepts for a data engineer to understand to protect data in the data lake, and a service such as AWS Lake Formation is a useful tool for easily managing authorization for datasets.

In the next chapter, we will take a step back and look at the bigger picture of how a data engineer can architect a data pipeline. We will begin exploring how to understand the needs of our data consumers, learn more about our data sources, and decide on the transformations that are required to transform raw data into useful data for analytics.

Section 2: Architecting and Implementing Data Lakes and Data Lake Houses

In this section of the book, we examine an approach for architecting a high-level data pipeline and then dive into the specifics of data ingestion and transformation. We also examine different types of data consumers, learn about the important role of data marts and data warehouses, and finally put it all together by orchestrating data pipelines. We get hands-on with various AWS services for data ingestion (Amazon Kinesis and DMS), transformation (AWS Glue Studio), consumption (AWS Glue DataBrew), and pipeline orchestration (Step Functions).

This section comprises the following chapters:

- *Chapter 5, Architecting Data Engineering Pipelines*
- *Chapter 6, Ingesting Batch and Streaming Data*
- *Chapter 7, Transforming Data to Optimize for Analytics*
- *Chapter 8, Identifying and Enabling Data Consumers*
- *Chapter 9, Loading Data into a Data Mart*
- *Chapter 10, Orchestrating the Data Pipeline*

5

Architecting Data Engineering Pipelines

Having gained an understanding of data engineering principles, the core concepts, and the available AWS tools, we can now put these together in the form of a data pipeline. A data pipeline is the process that ingests data from multiple sources, optimizes and transforms the data, and makes it available to data consumers. An important function of the data engineering role is the ability to design, or architect, these pipelines.

In this chapter, we will cover the following topics:

- Approaching the task of architecting a data pipeline
- Identifying data consumers and understanding their requirements
- Identifying data sources and ingesting data
- Identifying data transformations and optimizations
- Loading data into data marts
- Wrapping up the whiteboarding session
- Hands-on – architecting a sample pipeline

Technical requirements

For the hands-on portion of this lab, we will design a high-level pipeline architecture. You can perform this activity on an actual whiteboard, a piece of paper, or using a free online tool called diagrams.net. If you want to make use of this online tool, make sure you can access the tool at <http://diagrams.net>.

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/tree/main/Chapter05>

Approaching the data pipeline architecture

Before we get into the details of the individual components that will go into the architecture, it is helpful to get a 10,000 ft view of what we're trying to do.

A common mistake when starting a new data engineering project is to try and do everything at once, and to create a solution that covers all use cases. A better approach is to identify an initial, specific use case, and to start the project while focusing on that one outcome, but keeping the bigger picture in mind.

This can be a significant challenge, and yet it is really important to get this balance right. While you need to focus on an achievable outcome that can be completed within a reasonable time frame, you also need to ensure that you're building within a framework that can be used for future projects. If each business unit tackles the challenge of data analytics independently, with no corporate-wide analytics initiative, it will be difficult to unlock the value of corporate-wide data.

The ideal project will include sponsorship from the highest levels of the organization but will identify a limited scope project for building an initial framework. This project, when completed, can be used as an internal case study to drive forward additional analytic projects.

In the 1989 film *Field of Dreams*, a farmer (played by Kevin Costner) hears a voice saying

"If you build it, he will come."

Everyone in the town thinks he is crazy when he ends up sacrificing his crops to build a baseball field, but when he does, several long-dead baseball players come to the field to play. In business, a common mantra has become the following:

"If you build it, they will come."

This implies that if you build something really good, you will find customers for it. But this is not a recommended approach for building data analytic solutions.

Some organizations may have hundreds, or even thousands, of data sources, and many of those data sources may be useful for centralized analytics. But that doesn't mean we should attempt to immediately ingest them all into our analytics platform so that we can see how the business may use them. When organizations have taken this approach, embarking on multi-year-long projects to build out large analytic solutions covering many different initiatives, these have often failed.

Rather, once executive sponsorship has been gained and an initial project with limited scope has been identified, the data engineer can begin the process of designing a data pipeline for the project.

Architecting houses and architecting pipelines

If you were to build a new house, you would identify an appropriate piece of land, and then contract an architect to work with you to create the plans for the building. The architect would do several things:

- Discuss your requirements with you (how you want to use the home, what materials you would like, how many bedrooms, bathrooms, and so on).
- Gather information on the land where you will be building (size of the land, slope, and so on).
- Determine the type of materials that are best suited for building that environment.

As part of this, the architect may create a rough sketch showing the high-level plan. Once that high-level plan is agreed upon, the architect can gather more detailed information and then create a detailed architecture plan. This plan would include the layout of the rooms, and then where the shower, toilet, lights, and so on would go, and based on that, where the plumbing and electrical lines would run.

For a data engineer creating the architecture for a data pipeline, a similar approach can be used:

- Gather information from project sponsors and data consumers on their requirements. Learn what their objectives are, what types of tools they want to use to consume the data, required data transformations, and so on.
- Gather information on the available data sources. This may include what systems store the raw data, what format that data is in, who the system and data owner are, and so on.
- Determine what types of tools are available and may be best suited for these requirements.

A useful way to gather this information is to conduct a whiteboarding session with the relevant stakeholders.

Whiteboarding as an information-gathering tool

Running a whiteboarding session with relevant stakeholders enables the data engineer to develop a high-level plan for the data pipeline, and helps gather the information required to start working on the detailed design. The purpose of the whiteboarding exercise is not to work out all the technical details and finalize the specific services and tools that will be used. Rather, the purpose is to agree with stakeholders on the overall approach for the pipeline and to gather the information that's required for the detailed design.

In this book, we will be using an architectural approach, where we ingest data into an Amazon S3-based data lake. Data is initially ingested into a raw zone, and then we transform and optimize the data using several tools to move the data through different data lake zones. As we covered in *Chapter 2, Data Management Architectures for Analytics*, a data lake has multiple zones that the data moves through. Typically, these include zones such as raw, transformed, conformed, and enriched, but can also include zones such as staging and inference (for data science purposes). There is no specific number of zones that a data lake requires as zones should be based on business requirements, but for our whiteboarding session, we will show three zones.

Depending on data consumption requirements, we may then load subsets of the data into various data marts (such as Amazon Redshift, a cloud data warehouse service), making the data available to data consumers via various services.

The following diagram illustrates a high-level overview of the primary components of a typical data pipeline and the approach to developing the high-level pipeline architecture:

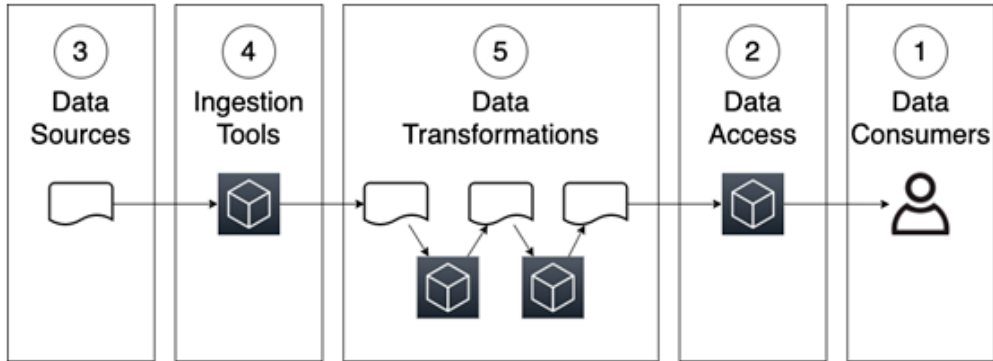


Figure 5.1 – High-level overview of a data pipeline architecture

When approaching the design of the pipeline, we can use the following sequence (which is also reflected by the numbers in the preceding diagram):

1. Understanding the business objectives and who the data consumers are and their requirements
2. Determining the types of tools that data consumers will use to access the data
3. Understanding which potential data sources may be available
4. Determining the types of toolsets that will be used to ingest data
5. Understanding the required data transformations at a high level to take the raw data and prepare it for data consumers

As you can see, we should always work backward when designing a pipeline. That is, we should start with the data consumers and their requirements, and then work from there to design our pipeline.

Conducting a whiteboarding session

Once an initial project has been identified, the data engineer should bring together relevant stakeholders for a workshop to whiteboard the high-level approach. Ideally, all stakeholders should meet in person, have a whiteboard available, and should plan for a half-day workshop. Stakeholders should include a group of people that can answer the following questions:

- Who is the executive sponsor and what are the business value and objectives for the project?
- Who is going to be working directly with the data (the data consumers)? What types of tools are the data consumers likely to use to access the data?

- What are the relevant raw data sources?
- At a high level, what types of transformations are required to transform and optimize the raw data?

The data engineer needs to understand the business objectives, and not just gather technical information during this workshop. A good place to start is to ask for a business sponsor to provide an overview of current challenges, and to review the expected business outcomes, or objectives, for the project. Also, ask about any existing solutions or related projects, and gaps or issues with those current solutions.

Once the team has a good understanding of the business value, the data engineer can begin whiteboarding to put together the high-level design. We work backward from our understanding of the business value of the project, which involves learning how the end-state data will be used to provide business value, and who the consumers of the data will be. From there, we can start understanding the raw data sources that will be needed to create the end-state data, and then develop a high-level plan for the types of transformations that may be required.

Let's start by identifying who our data consumers are and understanding their requirements.

Identifying data consumers and understanding their requirements

A typical organization is likely to have multiple different categories, or types, of data consumers. We discussed some of these roles in *Chapter 1, An Introduction to Data Engineering*, but let's review these again:

- **Business users:** A business user generally wants to access data via interactive dashboards and other visualization types. For example, a sales manager may want to see a chart showing last week's sales by sales rep, geographic area, or top product categories.
- **Business applications:** In some use cases, the data pipeline that the data engineer builds will be used to power other business applications. For example, Spotify, the streaming music application, provides users with an in-app summary of their listening habits at the end of each year (top songs, top genres, total hours of music streamed, and so on). Read the following Spotify blog post to learn more about how the Spotify data team enabled this: <https://engineering.atspotify.com/2020/02/18/spotify-unwrapped-how-we-brought-you-a-decade-of-data/>.

- **Data analyst:** A data analyst is often tasked with doing more complex data analysis, digging deeper into large datasets to answer specific questions. For example, across all customers, you may be wondering which products are most popular by different age or socio-economic demographics. Or, you may be wondering what percentage of customers have browsed the company's e-commerce store more than 5 times, for more than 10 minutes at a time, in the last 2 weeks but have not purchased anything. These users generally use structured query languages such as SQL.
- **Data scientist:** A data scientist is tasked with creating machine learning models that can identify non-obvious patterns in large datasets, or make predictions about future behavior based on historical data. To do this, data scientists need access to large quantities of diverse datasets that they may refine further.

During the whiteboarding workshop, the data engineer should ask questions to understand who the data consumers are for the identified project. As part of this, it is important to also understand the types of tools each data consumer is likely to want to use to access the data.

As information is discovered, it can be added to the whiteboard, as illustrated in the following diagram:

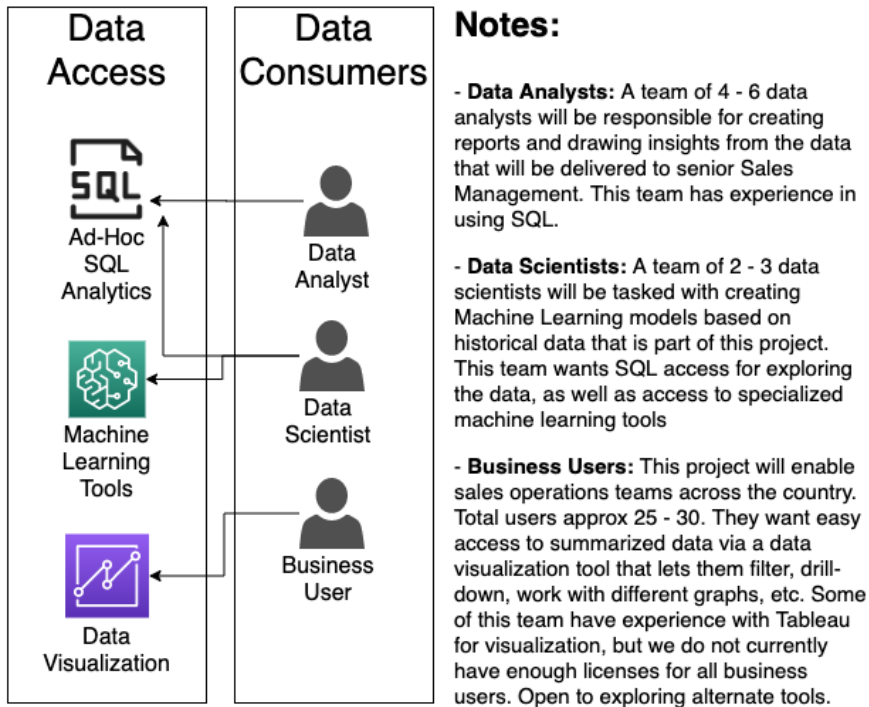


Figure 5.2 – Whiteboarding data consumers and data access

In this example, we can see that we have identified three different data consumers – a data analyst team, a data science team, and various business users. We have also identified the following:

- That the data analysts want to use ad hoc SQL queries to access the data
- That the data science team wants to use both ad hoc SQL queries and specialized machine learning tools to access the data
- That the business users want to use a **Business Intelligence (BI)** data visualization tool to access the data

It is useful to ask whether there are any existing corporate standard tools that the data consumer must use, but it is not important to finalize the toolsets at this point. For example, we should take note if a team already has experience with Tableau (a common BI application) and whether they want to use it for data visualization reporting. But if they have not identified a specific toolset they will use, that can be finalized at a later stage.

Once we have a good understanding of who the data consumers are for the project, and the types of tools they want to use to work with the data, we can move on to the next stage of whiteboarding, which is to examine the available data sources and means to ingest the data.

Identifying data sources and ingesting data

With an understanding of the overall business goals for the project, and having identified our data consumers, we can start exploring the available data sources.

While most data sources will be internal to the organization, some projects may require enriching organization-owned data with other third-party data sources. Today, there are many data marketplaces where diverse datasets can be subscribed to, or in some cases, accessed for free. When discussing data sources, both internal and external datasets should be considered.

The team that has been included in the workshop should include people that understand the data sources required for the project. Some of the information that the data engineer needs to gather about these data sources includes the following:

- Details about the source system containing the data (is the data in a database, in files on a server, existing files on Amazon S3, coming from a streaming source, and so on)?
- If this data is internal data, who is the owner of the source system within the business? Who is the owner of the data?

- What frequency does the data need to be ingested on (continuous streaming/ replication, loading data every few hours, loading data once a day)?
- Optionally, discuss some potential tools that could be used for data ingestion.
- What is the raw/ingested format of the data (CSV, JSON, native database format, and so on)?
- Does the data source contain PII or other types of data that is subject to governance controls? If so, what controls need to be put in place to protect the data?

As information is discovered, it can be captured on the whiteboard, as illustrated in the following diagram:

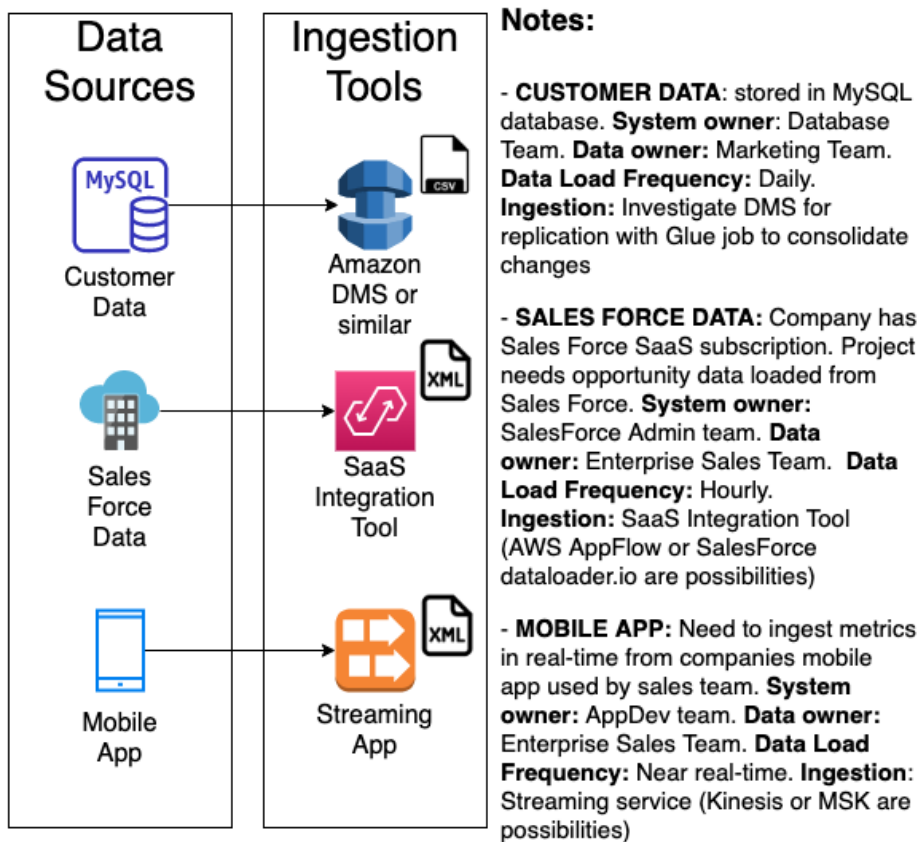


Figure 5.3 – Whiteboarding data sources and data ingestion

During the whiteboarding process, additional notes should be captured to provide more context or detail about the requirements. These can be captured directly on the whiteboard or captured separately.

In this example, we have identified three different data sources – customer data from a MySQL database, opportunity information from Salesforce, and near-real-time sales metrics from the organization's mobile application. We have also identified the following:

- The business team that owns each source system and the business team that owns the data
- The velocity of ingesting the data (how often each data source needs to be ingested)
- Potential services that can be used to ingest the data

When discussing ingestion tools, it may be worthwhile to capture potential tools if you have a good idea of which tool may be suitable. However, the objective of this session is not to come up with a final architecture and decision on all technical components. Additional sessions (as discussed later in this book) will be used to thoroughly evaluate potential toolsets against requirements and should be done in close consultation with source system owners.

During this whiteboarding session, we have been working backward, first identifying the data consumers, and then the data sources we plan to use. At this point, we can move on to the next phase of whiteboarding, which is to examine some of the data transformations that we plan to use to optimize the data for analytics.

Identifying data transformations and optimizations

In a typical data analytics project, we ingest data from multiple data sources and then perform transforms on those datasets to optimize them for the required analytics.

In *Chapter 7, Transforming Data to Optimize for Analytics* we will do a deeper dive into typical transformations and optimizations, but we will provide a high-level overview of the most common transformations here.

File format optimizations

CSV, XML, JSON, and other types of plaintext files are commonly used to store structured and semi-structured data. These file formats are useful when manually exploring data, but there are much better, binary-based file formats to use for computer-based analytics. A common binary format that is optimized for read-heavy analytics is the Apache Parquet format. A common transformation is to convert plaintext files into an optimized format, such as Apache Parquet.

Data standardization

When building out a pipeline, we often load data from multiple different data sources, and each of those data sources may have different naming conventions for referring to the same item. For example, a field containing someone's birth date may be called *DOB*, *dateOfBirth*, *birth_date*, and so on. The format of the birth date may also be stored as *mm/dd/yy*, *dd/mm/yyyy*, or in a multitude of other formats.

One of the tasks we may want to do when optimizing data for analytics is to standardize column names, types, and formats. By having a corporate-wide analytic program, standard definitions can be created and adopted across all analytic projects in the organization.

Data quality checks

Another aspect of data transformation may be the process of verifying data quality and highlighting any ingested data that does not meet the expected quality standards.

Data partitioning

A common optimization strategy for analytics is to partition the data, grouping the data at the physical storage layer by a field that is often used in queries. For example, if data is often queried by a date range, then data can be partitioned by a date field. If storing sales data, for example, all the sales transactions for a specific month would be stored in the same Amazon S3 prefix (which is much like a directory). When a query is run that selects all the data for a specific day, the analytic engine only needs to read the data in the directory that's storing data for the relevant month.

Data denormalization

In traditional relational database systems, the data is normalized, meaning that each table contains information on a specific focused topic, and associated, or related, information is contained in a separate table. The tables can then be linked through the use of foreign keys.

For data lakes, combining the data from multiple tables into a single table can often improve query performance. Data denormalization takes two (or more) tables and creates a new table with data from both tables.

Data cataloging

Another important component that we should include in the transformation section of our pipeline architecture is the process of cataloging the dataset. During this process, we ensure all the datasets in the data lake are referenced in the data catalog and can add additional business metadata.

Whiteboarding data transformation

For the whiteboarding session, we do not need to determine all the details of the required transformations, but it is useful to agree on the main transformations for the high-level pipeline design.

Some of the information that the data engineer needs to gather about expected data transformations during the whiteboarding session includes the following:

- Is there an existing set of standardized column name definitions and formats that can be referenced? If not, who will be responsible for creating these standard definitions?
- What additional business metadata should be captured for datasets? For example, data owner, cost allocation tags, data sensitivity, and so on.
- What format should optimized files be stored in? Apache Parquet is a common format, but you need to validate that the tools used by the data consumers can work with files in Apache Parquet format.
- Is there an obvious field that the data should be partitioned by?
- Are other required data transformations obvious at this point? For example, if you're ingesting data from a relational database, should the data be denormalized?
- What data transformation engines/skills does the team have? For example, does the team have experience creating Spark jobs using PySpark?

As information is discovered, it can be captured on the whiteboard, as illustrated in the following diagram:

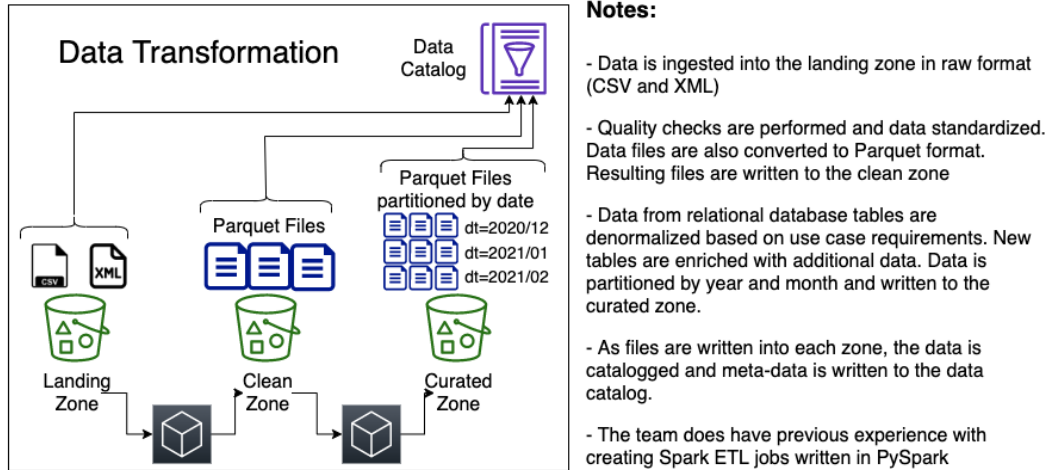


Figure 5.4 – Whiteboarding data transformation

In this example, we are creating a data lake with three zones – the landing zone, the clean zone, and the curated zone (as previously discussed in *Chapter 2, Data Management Architectures for Analytics*):

- Raw files are ingested into the landing zone and will be in plaintext formats such as CSV and XML. When the files are ingested, information about the files is captured in the data catalog, along with additional business metadata (data owner, data sensitivity, and so on).
- We haven't identified a specific data transformation engine at this point, but we did capture a note indicating that the team does have previous experience with creating Spark ETL jobs using PySpark. This means that AWS Glue may be a good solution for data transformation, but we will do further validation of this at a later stage.
- As part of our pipeline, we will have a process to run data quality checks on the data in the landing zone. If the quality checks pass, we will standardize the data (uniform column names and data types) and convert the files into Apache Parquet format, writing out the new files in the clean zone. Again, we will add the newly written-out files to our data catalog, including relevant business metadata.
- Another piece of our pipeline will now perform additional transformations on the data, as per the specific use case requirements. For example, data from a relational database will be denormalized, and tables can be enriched with additional data. We will write out the transformed data to the curated zone, partitioning the files by date as they are written out. Again, we will add the newly written-out files to our data catalog, including the relevant business metadata.

It's important to remember that the goal of this session is not to work out all the technical details, but rather to create a high-level overview of the pipeline. In the preceding diagram, we did not specify that AWS Glue will be the transformation engine. We know that AWS Glue may be a good fit, but it's not important to make that decision now.

We have indicated a potential partitioning strategy based on date, but this is also something that will need further validation. To determine the best partitioning strategy, you need a good understanding of the queries that will be run against the dataset. In this whiteboarding session, it is unlikely that there will be time to get into those details, but after the initial discussion, this appeared to be a good way to partition data, so we included it.

Having determined transformations for our data, we will move on to the last step of the whiteboarding process, which is determining whether we are going to require any data marts.

Loading data into data marts

Many tools can work directly with data in the data lake, as we covered in *Chapter 3, The AWS Data Engineer's Toolkit*. These include tools for ad hoc SQL queries (Amazon Athena), data processing tools (such as Amazon EMR and AWS Glue), and even specialized machine learning tools (such as Amazon SageMaker).

These tools read data directly from Amazon S3, but there are times where a use case may require much lower latency, higher performance reads of the data. Or, there may be times where the use of highly structured schemas may best meet the analytic requirements of the use case. In these cases, loading data from the data lake into a data mart makes sense.

In analytic environments, a data mart is most often a data warehouse system (such as Amazon Redshift), but it could also be a relational database system (such as Amazon RDS MySQL), depending on the use case's requirements. In either case, the system will have local storage (often high-speed flash drives) and local compute power, offering the best performance when needing to query across large datasets, and specifically where queries require joining across many tables.

As part of the whiteboarding session, you should spend some time discussing whether a data mart may be best suited for loading a subset of the data. For example, if you expect a large number of users to use your BI tool (for data visualizations), you may spend some time discussing which data will be used the most by these teams. You could then include a note about loading a subset of the data into a data warehouse system and connecting the data visualization tool to the data warehouse in your whiteboarding session.

Wrapping up the whiteboarding session

After completing the whiteboarding session, you should have a high-level overview architecture that illustrates the main components of the pipeline that you plan to build. At this point, there will still be a lot of questions that have been left unanswered and there will not be a lot of specific detail. However, the high-level architecture should be enough to get broad agreement from stakeholders on the proposed plans for the project. It should have also provided you with enough information that you can start on a detailed design and set up follow-up sessions as required.

Some of the information that you should have after the session includes the following:

- A good understanding of who the data consumers for this project will be
- For each category of data consumer, a good idea of what type of tools they would use to access the data (SQL, visualization tools, and so on)
- An understanding of the internal and external data sources that will be used
- For each data source, an understanding of the requirements for data ingestion frequency (daily, hourly, or near-real-time streaming, for example)
- For each data source, a list of who owns the data, and who owns the source system containing the data
- A high-level understanding of likely data transformations
- An understanding of whether loading a subset of data into a data warehouse or other data marts may be required

After the session, you should create a final high-level architecture diagram and include notes from the meeting. These notes should be distributed to all participants to request their approval and agreement on moving forward with the project based on the draft architecture.

Once an agreement has been reached on the high-level approach, additional sessions will be needed with the different teams to capture additional details and fully examine the requirements.

The final high-level architecture diagram, based on the scenario we have been looking at in this chapter, may look as follows:

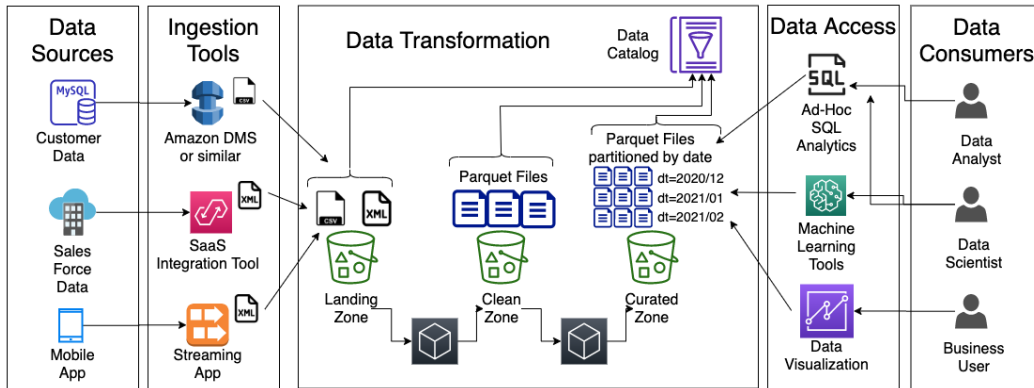


Figure 5.5 – High-level architecture whiteboard

In addition to our high-level architecture diagram on the whiteboard, we would have also captured associated notes about the various architecture components during the discussion. The notes that were captured for the scenario we discussed in this chapter may look like this:

Data Sources	Data Transformation	Data Consumers
<ul style="list-style-type: none"> - CUSTOMER DATA: stored in MySQL database. System owner: Database Team. Data owner: Marketing Team. Data Load Frequency: Daily. Ingestion: Investigate DMS for replication with Glue job to consolidate changes - SALES FORCE DATA: Company has Sales Force SaaS subscription. Project needs opportunity data loaded from Sales Force. System owner: Salesforce Admin team. Data owner: Enterprise Sales Team. Data Load Frequency: Hourly. Ingestion: SaaS Integration Tool (AWS AppFlow or Salesforce dataloader.io are possibilities) - MOBILE APP: Need to ingest metrics in real-time from companies mobile app used by sales team. System owner: AppDev team. Data owner: Enterprise Sales Team. Data Load Frequency: Near real-time. Ingestion: Streaming service (Kinesis or MSK are possibilities) 	<ul style="list-style-type: none"> - Data is ingested into the landing zone in raw format (CSV and XML) - Quality checks are performed and data standardized. Data files are also converted to Parquet format. Resulting files are written to the clean zone - Data from relational database tables are denormalized based on use case requirements. New tables are enriched with additional data. Data is partitioned by year and month and written to the curated zone. - As files are written into each zone, the data is catalogged and meta-data is written to the data catalog. - The team does have previous experience with creating Spark ETL jobs written in PySpark 	<ul style="list-style-type: none"> - DATA ANALYSTS: A team of 4 - 6 data analysts will be responsible for creating reports and drawing insights from the data that will be delivered to senior Sales Management. This team has experience in using SQL. - DATA SCIENTISTS: A team of 2 - 3 data scientists will be tasked with creating Machine Learning models based on historical data that is part of this project. This team wants SQL access for exploring the data, as well as access to specialized machine learning tools - BUSINESS USERS: This project will enable sales operations teams across the country. Total users approx 25 - 30. They want easy access to summarized data via a data visualization tool that lets them filter, drill-down, work with different graphs, etc. Some of this team have experience with Tableau for visualization, but we do not currently have enough licenses for all business users. Open to exploring alternate tools.

Figure 5.6 – Notes associated with our whiteboarding

Now that you understand the theory of how to conduct a whiteboarding session, it's time to get some practical hands-on experience. This next section provides details about a fictional whiteboarding session and allows you to practice your whiteboarding skills.

Hands-on – architecting a sample pipeline

For the hands-on portion of this chapter, you will review the detailed notes from a whiteboarding session held for the fictional company GP Widgets Inc. As you go through the notes, you should create a whiteboard architecture, either on an actual whiteboard or on a piece of poster board. Alternatively, you can create the whiteboard using a free online design tool, such as the one available at <http://diagrams.net>.

As a starting point for your whiteboarding session, you can use the following template. You can recreate this on your whiteboard or poster board, or you can access the diagrams.net template for this via the GitHub site for of this book:

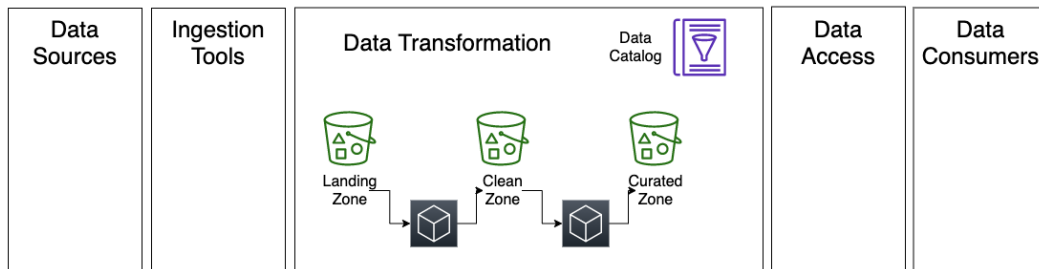


Figure 5.7 – Generic whiteboarding template

Note that the three zones included in the template (landing zone, clean zone, and curated zone) are commonly used for data lakes. However, some data lakes may only have two zones, while others may have four or more zones. The number of zones is not a hard rule but rather based on the requirements of the use case you are designing for.

As you go through the meeting notes, fill out the relevant sections of the template. In addition to drawing the components and flow for the pipeline, you should also capture notes relating to the whiteboard components, as per the example in *Figure 5.6*. At the end of this chapter, you can compare the whiteboard you have created with the one that the data engineer lead for GP Widgets has created.

By completing this exercise, you will gain hands-on experience in whiteboarding a data pipeline and learn how to identify keys points about data consumers, data sources, and required transformations.

Detailed notes from the project "Bright Light" whiteboarding meeting of GP Widgets, Inc

Here is a list of attendees participating in the meeting:

Attendees

- Ronna Parish, VP of marketing
- Chris Taylor, VP of enterprise sales
- Terry Winship, data analytics team manager
- James Dadd, data science team manager
- Owen McClave, database team manager
- Natalie Rabinovich, web server team manager
- Shilpa Mehta, data engineer lead

Meeting notes

Shilpa (SM) started the meeting by asking everyone to introduce themselves and then provided a summary of the meeting objectives:

- Plan out a high-level architecture for a new project to bring improved analytics to the marketing teams. During the discussion, Shilpa will whiteboard a high-level architecture.
- They reinforced that not all the technical details would be worked out in this meeting, but looking for agreement from all the stakeholders with a high-level approach and design is critical.
- Already agreed that the project will be built in the AWS cloud.

Shilpa asked Ronna (marketing manager) to provide an overview of the marketing team requirements for project *Bright Light*:

- Project Bright Light is intended to *improve the visibility that the marketing team has into real-time customer behavior*, as well as **longer-term customer trends**, through the use of data analytics.
- The marketing team wants to give **marketing specialists** real-time insights into interactions on the company's e-commerce website. Some examples of these **visualizations** include a heatmap to show website activity in different geographic locations; redemptions of coupons by product category; top ad campaign referrals, and spend by ZIP code in the previous period versus the current period.

- All visualizations should enable the user to select a custom period, be able to filter on custom fields, and also be able to drill down from summary information into detailed information. Data should be *refreshed on at least an hourly basis, but more often would be better.*

The **data analyst team** supporting the marketing department will run more complex investigations of current and historical trends:

- Identify the top 10% of customers over the past x days by spend and identify their top product categories for use in marketing promotions.
- Determine the average time a customer spends on the website, and the number of products they browse versus the number of products they purchase.
- Identify the top returned products over the past day/month/week.
- Compare the sales by ZIP code this month versus sales by ZIP code in the same month 1 year ago, grouped by product category.
- For each customer, keep a running total of their number of widget purchases (grouped by category), average spend per sale, average spend per month, the number of coupons applied, and the total coupon value.

We have tasked our **data science team** with building a machine learning model that can predict a widget's popularity based on the weather in a specific location:

- Research highlights how weather can impact e-commerce sales and the sales of specific products; for example, the types of widgets that customers are likely to buy on a sunny day compared to a cold and rainy day.
- The marketing team wants to target our advertising campaigns and optimize our ad spend and real-time marketing campaigns based on the current and forecasted weather at a certain ZIP code.
- We regularly add new widgets to our catalog, so the model must be updated at least once a day based on the latest weather and sales information.
- In addition to helping with marketing, the manufacturing and logistics teams have expressed interest in this model to help optimize logistics and inventory for different warehouses around the country based on 7-day weather forecasts.

James Dadd (data science team manager) spoke about some of the requirements for his team:

- They would need **ad hoc SQL access** to **weather, website clickstream logs**, and **sales** data for at least the last year.
- They have determined that a provider on AWS Data Exchange offers historical and forecast weather information for all US ZIP codes. There is a subscription charge for this data and the marketing team is working on allocating a budget for this. Data would be delivered **daily via AWS Data Exchange in CSV format**.
- James indicated he had not had a chance to speak with the database and website admin teams about getting access to their data yet.
- The team currently uses **SparkML** for a lot of their machine learning projects, but they are **interested in cloud-based tools** that may help them speed up delivery and collaboration for their machine learning products. They also **use SQL queries to explore datasets**.

Terry Winship (data analytics team manager) indicated that her team specializes in **using SQL** to gain complex insights from data:

- Based on her initial analysis, her team would need access to the **customer, product, returns, and sales databases**, as well as **clickstream data from the web server logs**.
- Her team has experience in working with on-premises data warehouses and databases. She has been reading up about data lakes and so long as the team can use SQL to query the data, they are open to using different technologies.
- She also has specialists on her team that **can create the visualizations** for the marketing team to use. This team primarily has experience with using **Tableau** for visualizations, but the marketing team **does not have licenses for using Tableau**. There would be a learning curve if a different visualization tool is used but they are **open to exploring other options**.
- Terry indicated that a **daily update of data from the databases** should be sufficient for what they need, but that they would need **near-real-time streaming for the clickstream web server log files** so that they could provide the most up-to-date reports and visualizations.

Shilpa asked Owen McClave (database team manager) to provide an overview of the databases sources that the data science and data analytics teams would need:

- Owen indicated that all their **databases run on-premises** and run **Microsoft SQL Server 2016, Enterprise Edition**.
- Owen said he doesn't know much about AWS and has some concerns about providing administrative access for his databases to the cloud since he does not believe the cloud is secure. However, he said that ultimately, it is up to the data owners to approve whether the data can be copied to the cloud. If approved, he will work with the cloud team to enable data syncing in the cloud, so long as there is no negative impact on his databases.
- Chris Taylor (VP of sales) is the data owner for the databases that have been discussed today (customer, product, returns, and sales data).

Shilpa asked Chris Taylor (VP of sales) to provide input on the use of sales data for the project:

- Chris indicated that this analytics project has executive sponsorship from senior management and visibility by the board of directors.
- He indicated that sales data can be stored in the cloud, so long as the security team review and approve it.

Shilpa indicated that AWS has a tool called **Database Migration Service** that can be used to replicate data from a relational database, such as SQL Server 2016, to Amazon S3 cloud storage. She said she would set up a meeting with Owen to discuss the requirements for this tool in more detail at a later point as there are also various other options.

Shilpa requested that Natalie Rabinovich (web server team manager) provide more information on the web server log files that will be important for this project:

- Natalie indicated that they currently run the e-commerce website on-premises on **Linux servers running Apache HTTP Server**.
- A load balancer is used to distribute traffic between **four different web servers**. Each server stores its clickstream web server logs locally.
- The log files are **plaintext files in Apache web log format**.
- Shilpa indicated that AWS has an agent called the **Kinesis Agent** that could be used to read the log files and stream their contents to the AWS cloud. She queried whether it would be possible to install this agent on the Apache web servers.

- Natalie indicated that it should be fine, but they would need more details and would need to test it in a development environment before installing it on the production servers.
- Shilpa asked who the data owner was. Natalie indicated that the marketing team owns the web server clickstream logs from a business perspective.

Shilpa led a discussion on the potential data transformations that may be required on the data that is ingested for this project:

- Based on the description from James, it appears that data should be made available daily in CSV format and can be loaded directly into the raw zone of the data lake.
- Using a tool such as Amazon DMS, we can load data from the databases into the raw zone of the data lake in Parquet format.
- The Kinesis Agent can convert the Apache HTTP Server log files into JSON format, and stream these to Kinesis Firehose. Firehose can then perform basic validation of the log (using Lambda), convert the log into Parquet format, and write directly to the clean zone.
- Shilpa indicated that an **initial transformation** could perform basic data **quality checks on incoming database data**, add contextual information as new columns (such as ingestion time), and then write the file **to the clean zone** of the data lake.
- Shilpa explained that partitioning datasets helps optimize both the performance and cost of queries. She indicated that partitions should be based on how the data is queried and led a discussion on the topic. After some discussion, it was agreed that partitioning the database and weather by day (yyyy/mm/dd) and web server logs by hour (yyyy/mm/dd/hh) may be a good partitioning strategy, but this would be investigated further and confirmed in future discussions.
- A **second transformation** could then be run against the data in the clean zone, performing tasks such as **denormalizing the relational datasets, joining tables into optimized tables, enriching data with weather data**, or performing any other required business logic. This optimized data would be **written to the curated zone** of the data lake. AWS Glue or Amazon EMR could potentially be used to perform these transformations.
- As each dataset is loaded, and then the transformed data is written out to the next zone, the **data will be added to the data catalog**. Once data has been added to the data catalog, authorized users will be able to query the data using SQL queries, enabled by a tool such as Amazon Athena. **Additional metadata could be added** at this point, including items such as data owner, source system, data sensitivity level, and so on.

- Shilpa indicated that she will arrange future meetings with the various teams to discuss the business metadata that must be added.
- Shilpa wrapped up the meeting with a brief overview of the whiteboard and committed to providing a copy of the whiteboard architecture and notes to all attendees for further review and comment. She also indicated that she would schedule additional meetings with smaller groups of people to dive deep into specific aspects of the proposed architecture to do additional validation.

Shilpa used a whiteboard in the meeting room in the meeting to sketch out a rough architecture, and then after the meeting, she created the following diagram to show the architecture:

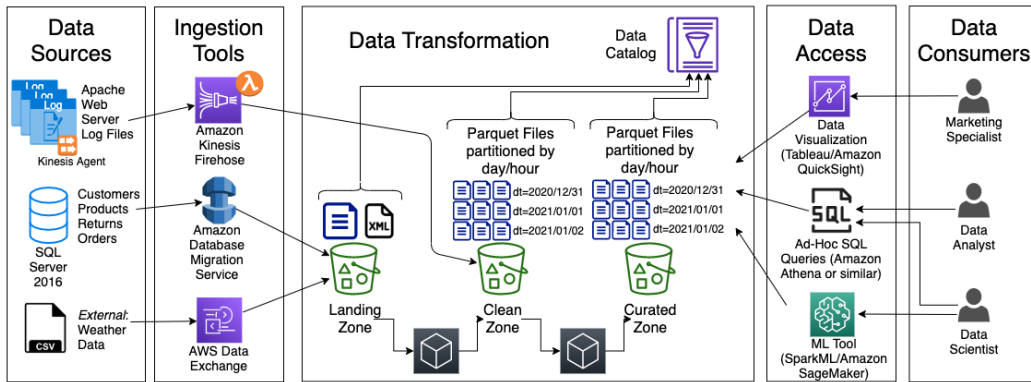


Figure 5.8 – Completed whiteboard architecture for project Bright Light

Shilpa also added some notes to go with the whiteboard and sent out both the whiteboard architecture and the notes to the meeting attendees:

Data Sources	Data Transformation	Data Consumers
<p>- Apache Web Server Log Files: From 4 Apache web servers. System Owner: Natalie Rabinovich. Data Owner: Marketing. Ingestion: Could use Kinesis Agent to transform to JSON and send to Kinesis Firehose. Firehose does validation (using Lambda function) and transforms to Parquet format. Could write direct to clean zone, partitioned by day (yyyy/mm/dd).</p> <p>- Databases: Customers, Products, Returns, Orders on SQL Server 2016 Enterprise Edition. System Owner: Owen McClave. Data Owner: Sales Team. Potentially use Amazon DMS to replicate to Amazon S3 raw zone in Parquet format.</p> <p>- Weather Data: External data source available via subscription. Data Owner: Marketing. Ingestion: Available from AWS Data Exchange marketplace. Lambda function can load data into Amazon S3 raw zone when available.</p>	<p>- Raw Zone: Database and weather data replicated into raw zone. When files ingested triggers Lambda function to perform data quality checks and then loads into Clean Zone partitioned by yyyy/mm/dd.</p> <p>- Clean Zone: Web server log files loaded directly into clean zone after Kinesis Firehose uses a Lambda function to perform data quality checks. Firehose configured to write to clean zone partitioned by yyyy/mm/dd. Database and weather files loaded from raw zone after data quality checks, and partition by yyyy/mm/dd.</p> <p>- Curated Zone: Database files denormalized, enriched (with weather data potentially), other business logic added. Partitioned by either day (databases, weather) or hour (web server log files)</p>	<p>- Marketing Specialists: Want to use business intelligence (visualization) tool to view up-to-date website analytics (ad-campaign referrals, coupon redemption, heatmap showing activity by geographic location). Refresh on at least hourly basis. Analytics team generally uses Tableau, but marketing team does not have licenses. Open to other BI tools.</p> <p>- Data Analysts: Responsible for creating reports and insights using SQL queries. Database and weather data could be refreshed daily, but they would need web server clickstream log files refreshed at least hourly.</p> <p>- Data Scientists: Need ad-hoc SQL access to databases, weather and web server log files. They currently use SparkML on-premises, but open to new cloud based tools that may make speed up delivery and collaboration for their machine learning products.</p>

Figure 5.9 – Completed whiteboard notes for project Bright Light

Compare the whiteboard you created to the whiteboard created by Shilpa, and note the differences. Are there things that Shilpa missed on her whiteboard or notes? Are there things that you missed on your whiteboard or notes?

The exercises in this chapter allowed you to get hands-on with data architecting and whiteboarding. We will wrap up this chapter by providing a summary, and then do a deeper dive into the topics of data ingestion, transformation, and data consumption in the next few chapters.

Summary

In this chapter, we reviewed an approach to developing data engineering pipelines by identifying a limited-scope project, and then whiteboarding a high-level architecture diagram. We looked at how we could have a workshop, in conjunction with relevant stakeholders in the organization, to discuss requirements and plan the initial architecture.

We approached this task by working backward. We started by identifying who the data consumers of the project would be and learning about their requirements. Then, we looked at which data sources could be used to provide the required data and how those data sources could be ingested. We then reviewed, at a high level, some of the data transformations that would be required for the project to optimize the data for analytics.

In the next chapter, we will take a deeper dive into the AWS services for ingesting batch and streaming data as part of our data engineering pipeline.

6

Ingesting Batch and Streaming Data

Having developed a high-level architecture of our data pipeline, we can now dive deep into the varied components of the architecture. We will start with data ingestion so that in the hands-on section of this chapter, we can ingest data that we can use for the hands-on activities in future chapters.

Data engineers are often faced with the challenge of the five Vs of data. These are the **variety** of data (the diverse types and formats of data); the **volume** of data (the size of the dataset); the **velocity** of the data (how quickly the data is generated and needs to be ingested); the **veracity** or **validity** of the data (the quality, completeness, and credibility of data); and finally, the **value** of data (the value that the data can provide the business with).

In this chapter, we will look at several different types of data sources and examine the various tools available within AWS for ingesting data from these sources. We will also look at how to decide between multiple different ingestion tools to ensure you are using the right tool for the job. In the hands-on portion of this chapter, we will ingest data from both streaming and batch sources.

In this chapter, we will cover the following topics:

- Understanding varied data sources
- Ingesting data from database sources

- Ingesting data from streaming sources
- Hands-on – ingesting data from a database source
- Hands-on – ingesting data from a streaming source

Technical requirements

In the hands-on sections of this chapter, we will use the Amazon DMS service to ingest data from a database source, and then we will ingest streaming data using Amazon Kinesis. To ingest data from a database, you need IAM permissions that allow your user to create an RDS database, an EC2 instance, a DMS instance, and a new IAM role and policy.

For the hands-on section on ingesting streaming data, you will need IAM permissions to create a Kinesis Data Firehose instance, as well as permissions to deploy a CloudFormation template. The CloudFormation template that is deployed will create IAM roles, a Lambda function, as well as Amazon Cognito users and other Cognito resources.

To query the newly ingested data, you will need permissions to create an AWS Glue Crawler and permissions to use Amazon Athena to query data.

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/tree/main/Chapter06>

Understanding data sources

Over the past decade, the amount and the variety of data that gets generated each year has significantly increased. Today, industry analysts talk about the volume of global data generated in a year in terms of **zettabytes (ZB)**, a unit of measurement equal to a billion **terabytes (TB)**. By some estimates, a little over 1 ZB of data existed in the world in 2012, and yet by the end of 2020, there would have been an estimated 59 ZB of data consumed globally.

In our pipeline whiteboarding session (covered in *Chapter 5, Architecting Data Engineering Pipelines*) we identified several data sources that we wanted to ingest and transform to best enable our data consumers. For each of these data sources that is identified in a whiteboarding session, you need to develop an understanding of the variety, volume, velocity, veracity, and value of data.

Data variety

In the past decade, the variety of data that has been used in data analytics projects has greatly increased. If all data was simply relational data in a database (and there was a time when nearly all data was like this), it would be relatively easy to transfer into data analytic solutions. But today, organizations find value, and often competitive advantage, by being able to analyze many other types of data (web server log files, photos, videos, and other media, geolocation data, sensor, and other IoT data, and so on).

For each data source in a pipeline, the data engineer needs to determine what type of data will be ingested. Data is typically categorized as being of one of three types, as we examine in the following subsections.

Structured data

Structured data is data that has been organized according to a data model, and is represented as a series of rows and columns. Each row represents a record, with the columns making up the fields of each record.

Each column in a structured data file contains data of a specific type (such as strings or numbers), and every row has the same number and type of columns. The definition of which columns are contained in each record, and the data type for each column, is known as the data schema.

Common data sources that contain structured data include the following:

- **Relational Database Management Systems (RDBMSes)** such as MySQL, PostgreSQL, SQL Server, and Oracle
- Delimited files such as **Comma Separated Values (CSV)** files or **Tab Separated Values (TSV)** files
- Spreadsheets such as Microsoft Excel files in `xls` format
- Data from online forms

The data shown in the following table is an example of structured data. In this case, it is data on the calorie content of several foods from the USA MyPyramid Food Raw Data, available at <https://catalog.data.gov/dataset/mypyramid-food-raw-data>. This data extract has been sorted to show some of the highest calorie content foods in the dataset:

Food_Code	Display_Name	Portion_Display_Name	Total_Calories
71411000	Potato skin with cheese & bacon	order (10 halves)	1667.4
24301010	Roasted duck	duck half	1283.52
21103120	Breaded fried steak (eat lean & fat)	large steak	1069.2
28141010	Fried chicken frozen meal	large meal (16 oz)	1024.92
27347100	Chicken or turkey pot pie	16-ounce pie (Hungry Man)	976.1
58200100	Wrap sandwich (meat, vegetables, rice)	wrap	818.37
21103120	Breaded fried steak (eat lean & fat)	medium steak	801.9
58106730	Meat & veggie pizza, thick crust	small pizza (8" across)	798.64
24401010	Roasted Cornish game hen	hen	792.54
58106530	Meat pizza, thick crust	small pizza (8" across)	785.4

Figure 6.1 – An example of structured data

Structured data can be easily ingested into analytic systems, including Amazon S3-based data lakes, and data marts such as an Amazon Redshift data warehouse.

Semi-structured data

Semi-structured data shares many of the same attributes as structured data, but the structure, or schema, does not need to be strictly defined. Generally, semi-structured data contains internal tags that identify data elements, but each record may contain different elements or fields.

Some of the data types in the unstructured data may be of a strong type, such as an integer value, while other elements may contain items such as free-form text. Common semi-structured formats include JSON and XML file formats.

The data that follows is part of a semi-structured JSON formatted file for product inventory. In this example, we can see that we have two items – a set of batteries and a pair of jeans:

```
[{
  "sku": 10001,
  "name": "Duracell - Copper Top AA Alkaline Batteries
- long lasting, all-purpose 16 Count",
  "price": 12.78,
```

```
"category": [{
  "id": "5443",
  "name": "Home Goods"
}],
"manufacturer": "Duracell",
"model": "MN2400B4Z"
},
{
  "sku": 10002,
  "name": "Levi's Men's 505 Jeans Fit Pants",
  "type": "Clothing",
  "price": 39.99,
  "fit_type": [{
    "id": 855,
    "description": "Regular"
  },
  {
    "id": 902,
    "description": "Big and Tall"
  }
],
"size": [{
  "id": 101,
  "waist": 32,
  "length": 32
},
{
  "id": 102,
  "waist": 30,
  "length": 32
}],
"category": [{
  "id": 3821,
  "name": "Jeans"
}], {
```

```
        "id": 6298,  
        "name": "Men's Fashion"  
    }],  
    "manufacturer": "Levi",  
    "model": "00505-4891"  
  }  
]
```

While most of the fields are common between the two items, we can see that the pair of jeans includes attributes for `fit_type` and `size`, which are not relevant to batteries. You will also notice that the first item (the batteries) only belongs to a single category, while the jeans are listed in two categories.

Capturing the same information in a structured data type, such as CSV, would be much more complex. CSV is not well suited to a scenario where different records have a different number of categories, for example, or for where some records have additional attributes (such as `fit_type` or `size`). JSON is structured in a hierarchical format (where data can be nested, such as for `category`, `fit_type`, and `size`) and this provides significant flexibility.

Storing data in a semi-structured format, such as JSON, is commonly used for a variety of different use cases, such as working with IoT data, as well as for web and mobile applications.

Unstructured data

As a category, unstructured data covers many different types of data where the data does not have a predefined structure or schema. This can range from free-form data (such as text data in a PDF or word processing file or emails) to media files (such as photos, videos, satellite images, or audio files).

Some unstructured data can be analyzed directly, although generally not very efficiently, unless using specialized tools. For example, it is generally not efficient to search against large quantities of free-form text in a traditional database, although there are specialized tools that can be used for this purpose (such as Amazon ElasticSearch).

However, some types of unstructured data cannot be directly analyzed with data analytic tools at all. For example, data analytic tools are unable to directly analyze image or video files. This does not mean that we cannot use these types of files in our analytic projects, but to make them useful for analytics, we need to process them further.

A large percentage of the data that's generated today is considered unstructured data, and in the past few years, a lot of effort has been put into being able to make better use of this type of data. For example, we can use image recognition tools to extract metadata from an image or video file that can then be used in analytics. Or, we could use natural language processing tools to analyze free-form text reviews on a website to determine customer sentiment for different products.

Refer to *Chapter 13 Enabling Artificial Intelligence and Machine Learning*, for an example of how Amazon Comprehend can be used to extract sentiment analysis from product reviews.

Data volume

The next attribute of data that we need to understand for each of our data sources relates to the volume of data. For this, we need to understand both the total size of the existing data that needs to be ingested, as well as the daily/monthly/yearly growth of data.

For example, we may want to ingest data from a database that includes a one-time ingest of 10 years of historical data totaling 2.2 TB in size. We may also find that data from this source generates around 30 GB of new data per month (or an average of 1 GB per day of new data). Depending on the network connection between the source system and the AWS target system, it may be possible to transfer the historical data over the network, but if we have limited bandwidth, we may want to consider using one of the devices in the Amazon Snow family of devices. For example, we could load data onto an Amazon Snowball device that is shipped to our data center and then send the device back to AWS, where AWS will load the data into S3 for us.

Understanding the volume of historical and future data also assists us in doing the initial sizing of AWS services for our use case, and helps us budget better for the associated costs.

Data velocity

Data velocity describes the speed at which we expect to ingest and process new data from the source system into our target system.

For ingestion, some data may be loaded on a batch schedule once a day, or a few times a day (such as receiving data from a partner on a scheduled basis). Meanwhile, other data may be streamed from the source to the target continually (such as when ingesting IoT data from thousands of sensors).

As an example, according to a case study on the AWS website, the BMW group uses AWS services to ingest data from millions of BMW and MINI vehicles, processing terabytes of anonymous telemetry data every day. To read more about this, refer to the AWS case study titled BMW Group Uses AWS-Based Data Lake to Unlock the Power of Data (<https://aws.amazon.com/solutions/case-studies/bmw-group-case-study/>).

We need to have a good understanding of both how quickly our source data is generated (on a schedule or via real-time streaming data), as well as how quickly we need to process the incoming data (does the business only need insights from the data 24 hours after it is ingested, or is the business looking to gather insights in near-real-time?).

The velocity of data affects both how we ingest the data (such as through a streaming service such as Amazon Kinesis), as well how we process the data (such as whether we run scheduled daily Glue jobs, or use Glue streaming to continually process incoming data).

Data veracity

Data veracity considers various aspects of the data we're ingesting, such as the quality, completeness, and accuracy of the data.

As we discussed previously, data we ingest may have come from a variety of sources, and depending on how the data is generated, the data may be incomplete or inconsistent. For example, data from IoT sensors where the sensor went offline for a while may be missing a period of data. Or, data captured from user forms or spreadsheets may contain errors or missing values.

We need to be aware of the veracity of the data we ingest so that we can ensure we take these items into account when processing the data. For example, some tools can help backfill missing data with averages, as well as tools that can help detect and remediate fields that contain invalid data.

Data value

Ultimately, all the data that's ingested and processed is done for a single purpose – finding ways to provide new insights and value to the business. While this is the last of the five V's that we will discuss, it is the most important one to keep in mind when thinking of the bigger picture of what we are doing with data ingestion and processing.

We could ingest terabytes of data and clean and process the data in multiple ways, but if the end data product does not bring value to the business, then we have wasted both time and money.

When considering the data we are ingesting, we need to ensure we keep the big picture in mind. We need to make sure that it is worth ingesting this data, and also understand how this data may add value to the business, either now or in the future.

Questions to ask

In *Chapter 5, Architecting Data Engineering Pipelines*, we held a workshop during which we identified some likely data sources needed for our data analytics project, but now, we need to dive deeper to gather additional information.

We need to identify who owns each data source that we plan to ingest, and then do a deep dive with the data source owner and ask questions such as the following:

- What is the format of the data (relational database, NoSQL database, semi-structured data such as JSON or XML, unstructured data, and so on)?
- How much historical data is available?
- What is the total size of the historical data?
- How much new data is generated on a daily/weekly/monthly basis?
- Does the data currently get streamed somewhere, and if so, can we tap into the stream of data?
- How frequently is the data updated (constantly, such as with a database or streaming source, or on a scheduled basis such as at the end of the day or when receiving a daily update from a partner)?
- How will this data source be used to add value to the business, either now or in the future?

Learning more about the data will help you determine the best service to use to ingest the data, and help with initial sizing of services, and estimating a budget.

Ingesting data from a relational database

A common source of data for analytical projects is data that comes from a relational database system such as MySQL, PostgreSQL, SQL Server, or an Oracle database. Organizations often have multiple siloed databases, and they want to bring the data from these varied databases into a central location for analytics.

It is common for these projects to include ingesting historical data that already exists in the database, as well as for syncing ongoing new and changed data from the database. There are a variety of tools that can be used to ingest from database sources, as we will discuss in this section.

AWS Database Migration Service (DMS)

The primary AWS service for ingesting data from a database is AWS **Database Migration Service (DMS)**, though there are other ways to ingest data from a database source. As a data engineer, you need to evaluate both the source and the target to determine which ingestion tool will be best suited.

AWS DMS is primarily intended for doing either a one-off ingestion of historical data from a database, or for replicating data from a relational database on an ongoing basis. When using AWS DMS, the target is either a different database engine or an Amazon S3-based data lake. In this section, we will focus on ingesting data from a relational database to an Amazon S3-based data lake.

We introduced the AWS DMS service in *Chapter 3, The AWS Data Engineer's Toolkit*, so make sure you have read the *Overview of Amazon Database Migration Service (DMS)* section in that chapter to get a good understanding of how the service works.

Note that AWS DMS is a managed service, but it is not a serverless service. DMS provisions one or more EC2 servers as **replication instances**. These replication instances connect to the source database, read data from the source, prepare the data for loading into the target, and then connect to the target and write the data.

AWS Glue

AWS Glue, a Spark processing engine that we introduced in *Chapter 3, The AWS Data Engineer's Toolkit*, can make connections to several data sources. This includes connections to JDBC sources, enabling Glue to connect to many different database engines, and through those connections transfer data for further processing.

AWS Glue is well suited to certain use cases related to ingesting data from databases. Let's take a look at some of them.

Full one-off loads from one or more tables

AWS Glue can be configured to make a JDBC connection to a database and download data from tables. Glue effectively does a select (*) from the table, reading the table contents into the memory of the Spark server. At that point, you can use Spark to write out the data to Amazon S3, optionally in an optimized format such as Apache Parquet.

Initial full loads from a table, and subsequent loads of new records

AWS Glue has a concept called bookmarks, which enables Glue to keep track of which data was previously processed, and then on subsequent runs only process new data. Glue does this by having you identify a column (or multiple columns) in the table that will serve as a bookmark key. The values in this bookmark key must always increase in value, although gaps are allowed.

For example, if you have an audit table that has a transaction ID column that sequentially increases for each new transaction, then this would be a good fit for ingesting data with AWS Glue while using the bookmark functionality.

The first time the job runs, it will load all records from the table and store the highest value for the transaction ID column in the bookmark. For illustration purposes, let's assume the highest value on the initial load was 944,872. The next time the job runs, it effectively does `select * from audit_table where transaction_id > 944872`.

Note that this process is unable to detect updated or deleted rows in the table, so it is not well suited to all use cases. An audit table, or similar types of tables where data is always added to the table but existing rows are never updated or deleted, is the optimal use case for this functionality.

Creating AWS Glue jobs with AWS Lake Formation

AWS Lake Formation includes several blueprints to assist in automating some common ingestion tasks. One of these ingestion blueprints allows you to use AWS Glue to ingest data from a database source. With a few clicks in the Lake Formation console, you can configure your ingest requirements (one-off versus scheduled, full table load or incremental load with bookmarks, and so on). Once configured, Lake Formation creates the Glue Job for ingesting from the database source, the Glue Crawlers for adding newly ingested data into the Glue data catalog, and Glue Workflow for orchestrating the different components.

Other ways to ingest data from a database

There are several other approaches for ingesting data from a database to an Amazon S3-based data lake that we will cover briefly in this section.

Amazon EMR provides a simple way to deploy several common Hadoop framework tools, and some of these tools are useful for ingesting data from a database. For example, you can run Apache Spark on Amazon EMR and use a JDBC driver to connect to a relational database to load data into the data lake (in a similar way to our discussion around using AWS Glue to connect to a database). Alternatively, in Amazon EMR, you can deploy Apache Sqoop, a popular tool for transferring data between relational database systems and Hadoop.

If you are running MariaDB, MySQL, or PostgreSQL on Amazon **Relational Database Service (RDS)**, you can use RDS functionality to export a database snapshot to Amazon S3. This is a fully managed process that writes out all tables from the snapshot to Amazon S3 in Apache Parquet format. This is the simplest way to move data into Amazon S3 if you are using one of the supported database engines on RDS.

There are also several third-party commercial tools, many containing advanced features, that can be used to move data from a relational database to Amazon S3 (although these often come at a premium price). This includes tools such as Qlik Replicate (previously known as Attunity), a well-known tool for moving data between a wide variety of sources and targets (including relational databases, data warehouses, streaming sources, enterprise applications, cloud providers, and legacy platforms such as DB2, RMS, and so on).

You may also find that your database engine contains tools for directly exporting data in a flat format that can then be transferred to Amazon S3. Some database engines also have more advanced tools, such as Oracle GoldenGate, a solution that can generate **Change Data Capture (CDC)** data as a Kafka stream. Note, however, that these tools are often licensed separately and can add significant additional expense. For an example of using Oracle GoldenGate to generate CDC data that has been loaded into an S3 data lake, search for the AWS blog post titled Extract Oracle OLTP data in real time with GoldenGate and query from Amazon Athena: <https://aws.amazon.com/blogs/big-data/extract-oracle-oltp-data-in-real-time-with-goldengate-and-query-from-amazon-athena/>.

Reminder About CDC

We introduced the concept of CDC in *Chapter 3, The AWS Data Engineer's Toolkit*, but it is an important concept, so here is a reminder. When rows in a relational database are deleted or updated, there is no practical way to capture those changes using standard database query tools (such as SQL). But when replicating data from a database to a new source, it is important to be able to identify those changes so that they can be applied to the target. This process of identifying and capturing these changes (new inserts, updates, and deletes) is referred to as CDC.

Deciding on the best approach for ingesting from a database

While all these tools can be used in one way or another to ingest data from a database, there are several points to consider when deciding on the best approach for your specific use case.

Size of the database

If the total size of the database tables you want to load is large (many tens of GBs or larger), then doing a full nightly load would not be a good approach. The full load could take a significant amount of time to run and puts a heavy load on the source system while running. In this scenario, a better approach is to do an initial load from the database and then constantly sync updates from the source using CDC.

For very large databases, you can use AWS DMS with an Amazon Snowball device to load data to the Snowball device in your data center. Once the data has been loaded, you return the device to AWS, and they will load it to Amazon S3. AWS DMS will capture all CDC changes while the Snowball device is being transferred back to AWS so that once the data is loaded, you can create an ETL job to apply changes to the full data load.

For smaller databases where you do not need to capture changes in near-real-time, you can consider using AWS Glue (or native database tools) to load the entire database to Amazon S3 on a scheduled basis. This will often be the simplest and most cost-effective method, but it is not right for every use case.

Database load

If you have a database with a consistent production load at all times, you will want to minimize the additional load you place on the server to sync to the data lake. In this scenario, you can use DMS to do an initial full load, ideally from a read replica of your database if it's supported as a source by DMS. For ongoing replication, DMS can use database log files to identify CDC changes, and this places a lower load on database resources.

Whenever you do a full load from a database (whether you're using AWS DMS, AWS Glue, or another solution), there will be a heavier load on the database as a full read of all tables is required. You need to consider this load and, where possible, use a replica of your database for the full load.

If a smaller database is running on Amazon RDS, the best solution would be to use the export to S3 from snapshot functionality of RDS, if it's supported for your database engine. This solution places no load on your source database.

Data ingestion frequency

Some analytic use cases are well suited to analyzing data that is ingested on a fixed schedule (such as every night). However, some use cases will want to have access to new data as fast as possible.

If your use case requires access to data coming from a database source as soon as possible, then using a service such as AWS DMS to ingest CDC data is the best approach. However, remember that CDC data just indicates what data has changed (new rows inserted, existing rows updated or deleted), so you still require a process to apply that to the existing data to enable querying for the most up-to-date state.

If your use case allows for regularly scheduled updates, such as nightly, you can do a scheduled full load (if the database's size and performance impacts allow), or you can have a nightly process to apply the CDC data that was collected during the day to the previous snapshot of data.

In *Chapter 7, Transforming Data to Optimize for Analytics*, we will review several approaches for applying CDC data to an existing dataset.

Technical requirements and compatibility

When evaluating different approaches and tools for ingesting data from a database source, it is very important to involve the database owner and admin team upfront to technically evaluate the proposed solution.

A data engineering team may decide on a specific toolset upfront, based on their requirements and their broad understanding of compatibility with the source systems. However, at the time of implementation, they may discover that the source database team objects to certain security or technical requirements of the solution, and this can lead to significant project delays.

For example, AWS DMS supports CDC for several MySQL versions. However, DMS does require that binary logging is enabled on the source system with specific configuration settings for CDC to work.

Another example is that AWS DMS does not support server-level audits when SQL Server 2008/2008 R2 is used as a source. Certain commands related to enabling this functionality will cause DMS to fail.

It is critical to get the buy-in of the database owner and admin team before finalizing a solution. All of these requirements and limitations are covered in the AWS DMS documentation (and other solutions or products should have similar documentation covering their requirements). Reviewing these requirements, in detail, with the admin team up front is critical to the success of the project.

In the next section, we will take a similar look at tools and approaches for ingesting data from streaming sources.

Ingesting streaming data

An increasingly common source of data for analytic projects is data that is continually generated and needs to be ingested in near-real-time. Some common sources of this type of data are as follows:

- Data from IoT devices (such as smartwatches, smart appliances, and so on)
- Telemetry data from various types of vehicles (cars, airplanes, and so on)
- Sensor data (from manufacturing machines, weather stations, and so on)
- Live gameplay data from mobile games
- Mentions of the company brand on various social media platforms

For example, Boeing, the aircraft manufacturer, has a system called **Airplane Health Management (AHM)** that collects in-flight airplane data and relays it in real time to Boeing systems. Boeing processes the information and makes it immediately available to airline maintenance staff via a web portal.

In this section, we will look at several tools and services for ingesting streaming data, as well as things to consider when planning for streaming ingestion.

Amazon Kinesis versus Amazon Managed Streaming for Kafka (MSK)

The two primary services for ingesting streaming data within AWS are Amazon Kinesis and Amazon MSK. Both of these services were described in *Chapter 3, The AWS Engineer's Toolkit*, so ensure you have read those sections before proceeding.

In summary, both Amazon Kinesis and Amazon MSK are services from AWS that offer pub-sub message processing. That is, producers create messages that are written to the streaming service (Kinesis or MSK), and consumers subscribe to receive messages from the service. This is commonly used as a way to decouple applications producing streaming data from applications that are consuming data. Both services can scale up to handle millions of messages per second.

In this section, we will examine some of the primary differences between the two services and look at some of the factors that contribute to deciding which service is right for your use case.

Serverless services versus managed services

Amazon Kinesis is a serverless service, meaning that you never need to make decisions about, manage, or know anything about the underlying servers that run the service. With Kinesis Data Streams, for example, you configure the number of shards for your stream, and AWS automatically configures the required compute infrastructure (a shard in Kinesis is the base throughput unit of a Kinesis data stream). With Amazon Kinesis Data Firehose, you don't even need to specify the number of shards to be provisioned, as Kinesis Data Firehose automatically scales up and down in response to message throughput changes without requiring any configuration.

Amazon MSK is a managed service, meaning AWS manages the infrastructure for you, but you still need to be aware of and make decisions about the underlying compute infrastructure and software. For example, you need to select from a list of instance types to power your MSK cluster, configure VPC network settings, and also fine-tune a range of Kafka configuration settings. You also need to select the version of Kafka that you want to use with the service.

As a serverless service, Kinesis is much quicker and easier to set up and configure than Amazon MSK. However, Amazon MSK provides a lot more options for configuring and fine-tuning the underlying software.

If you have a team with existing skills in using Apache Kafka, and you need to fine-tune the performance of the stream, then you may want to consider MSK. If you're just getting started with streaming and your use case does not have a requirement to fine-tune performance, then Amazon Kinesis may be a better option.

Open source flexibility versus proprietary software with strong AWS integration

Amazon MSK is a managed version of Apache Kafka, a popular open source solution. Amazon Kinesis is proprietary software created by AWS, although there are some limited open source elements, such as the Kinesis Agent.

With Apache Kafka, there is a large community of contributors to the software, and a large ecosystem providing a diverse range of connectors and integrations. Kafka provides out-of-the-box integration with hundreds of event sources and event sinks (including AWS services such as Amazon S3, but also many other popular products, such as PostgreSQL, Elasticsearch, and others).

With Amazon Kinesis, AWS provides strong integration with several AWS services, such as Amazon S3, Amazon Redshift, and Amazon Elasticsearch. Kinesis also provides integration with a limited set of external services such as Splunk and DataDog through Amazon Kinesis Data Firehose.

When deciding between the two services, ensure that you consider the types of integrations your use case requires and how that matches with the out-of-the-box functionality of either Kinesis or MSK.

At-least-once messaging versus exactly once messaging

When working with streaming technologies, some use cases have specific requirements around how many times messages may be processed by data consumers. Amazon Kinesis and Apache Kafka (and therefore Amazon MSK) provide different guarantees around message processing.

Amazon Kinesis provides an *at least once* message processing guarantee. This effectively guarantees that every message generated by a producer will be delivered to a consumer for processing. However, in certain scenarios, a message may be delivered more than once to a consuming application, introducing the possibility of data duplication.

With **Apache Kafka** (and therefore Amazon MSK), as of version 0.11, the ability to configure your streams for *exactly once* message processing was introduced. When you configure your Apache Kafka stream, you can configure the: `processing.guarantee=exactly_once` setting to enable this.

With Amazon Kinesis, you need to build the logic for anticipating and appropriately handling how individual records are processed multiple times in your application. AWS provides guidance on this in the Kinesis documentation, in the *Handling Duplicate Records* section.

If your use case calls for a guarantee that all messages will be delivered to the processing application exactly once, then you should consider Amazon MSK. Amazon Kinesis is still an option, but you will need to ensure your application handles the possibility of receiving duplicate records.

Single processing engine versus niche tools

Apache Kafka is most closely compared to Amazon Kinesis Data Streams as both provide a powerful way to consume streaming messages. While both can be used to process a variety of data types, Amazon Kinesis does include several distinct sub-services for specialized use cases.

For example, if your use case involves ingesting streaming audio or video data, then Amazon Kinesis Video Streams is custom-designed to simplify this type of processing. Or, if you have a simple use case of wanting to write out ingested streaming data to targets such as Amazon S3, Amazon Elasticsearch, or Amazon Redshift (as well as some third-party services), then Amazon Kinesis Data Firehose makes this task simple.

Deciding on a streaming ingestion tool

There are several factors to consider when deciding on which AWS service to use for processing your streaming data, as we covered in this section. Amazon Kinesis requires less upfront configuration and has fewer ongoing maintenance tasks, and it also has a subset of services for special use cases. Because of this, you should evaluate your use case against the various Kinesis services and see if one of these will meet your current and expected future requirements. If your use case has specific requirements, such as exactly once message delivery, the ability to fine-tune the performance of the stream, or needs integration with third-party products not directly available in Kinesis, then consider Amazon MSK.

In the next few sections, you will get hands-on with ingesting data from a database using AWS DMS and then ingesting streaming data using Amazon Kinesis.

Hands-on – ingesting data with AWS DMS

As we discussed earlier in this chapter, AWS DMS can be used to replicate a database into an Amazon S3-based data lake (among other uses). Follow the steps in this section to do the following:

- Create a new MySQL database instance in your account.
- Load a MySQL demo database using an EC2 instance.

- Set up a DMS replication instance and configure endpoints and tasks.
- Run the DMS full-load.
- Run a Glue Crawler to add the tables that were newly loaded into S3 into the data catalog.
- Query the data with Amazon Athena.

Note

The following steps assume the use of your AWS account's default VPC and security group. You will need to modify the steps as needed if you're not using the default.

Creating a new MySQL database instance

First, we will create a new MySQL database using the default easy create settings for a free tier eligible database instance:

1. Log into the AWS Management Console (<https://console.aws.amazon.com>).
2. In the top search bar, search for and select **RDS** to access the RDS console.
3. In the left-hand menu, click on **databases**.
4. Click **Create database**.
5. For the database creation method, select **Easy Create**.
6. For **Engine type**, select **MySQL**.
7. For **DB instance size**, select **Free tier** (`db.t2.micro`).
8. For **DB instance identifier**, provide a name, such as **dataeng-mysql-1**.
9. For **Master password**, provide a password and ensure you can recall the password you set here as this will be needed later.

10. Click **Create database**:

DB instance size

<input type="radio"/> Production db.r6g.xlarge 4 vCPUs 32 GiB RAM 500 GiB 1.017 USD/hour	<input type="radio"/> Dev/Test db.r6g.large 2 vCPUs 16 GiB RAM 100 GiB 0.231 USD/hour	<input checked="" type="radio"/> Free tier db.t2.micro 1 vCPUs 1 GiB RAM 20 GiB 0.020 USD/hour
----------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------

DB instance identifier
 Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.

dataeng-mysql-1

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 60 alphanumeric characters or hyphens (1 to 15 for SQL Server). First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

Master username [Info](#)
 Type a login ID for the master user of your DB instance.

admin

1 to 16 alphanumeric characters. First character must be a letter

Auto generate a password
 Amazon RDS can generate a password for you, or you can specify your own password

Master password [Info](#)

.....

Constraints: At least 8 printable ASCII characters. Can't contain any of the following: / (slash), '(single quote), "(double quote) and @ (at sign).

Confirm password [Info](#)

.....

► **View default settings for Easy create**

Easy create sets the following configurations to their default values, some of which can be changed later. If you want to change any of these settings now, use [Standard Create](#).

ⓘ You are responsible for ensuring that you have all of the necessary rights for any third-party products or services that you use with AWS services.

Cancel **Create database**

Figure 6.2 – A portion of the Create database screen

- Click on the name of the database you just created and take note of the **Endpoint** property (which is the database instance's hostname) under **Connectivity & Security**.
- Note that it may take a few minutes for the database to be created before the endpoint URL is displayed

In this section, we created a new MySQL database instance. In the next section, we will create an EC2 instance to load some demo data into the database.

Loading the demo data using an Amazon EC2 instance

We want to create a demo database in MySQL that we will then load into Amazon S3 using AWS DMS. To load demo data into the database, we're going to use an Amazon EC2 instance:

1. In the AWS Management Console, search for and select **EC2** using the top search bar.
2. In the left-hand menu, click on **Instances**.
3. At the top right, click on **Launch instances**.
4. Select **Amazon Linux 2 AMI (HVM), SSD Volume Type**.
5. For **Instance type**, select **t2.micro** and then select **Next: Configure Instance Details**.
6. For **Configure instance details**, make sure that **Auto-assign Public IP** is set to **Enable**.
7. At the bottom of the page is a section for **User data**. Paste the following bash script into this section; the script will be run when the instance starts for the first time.

Make sure you replace `<PASSWORD>` with the password you set in *Step 9* of the *Creating a new MySQL database instance* section and replace `<HOST>` with the name of your MySQL database instance endpoint you took note of in *Step 11* of the *Creating a new MySQL database instance* section:

```
#!/bin/bash
yum install -y mariadb
curl https://downloads.mysql.com/docs/sakila-db.zip -o
sakila.zip
unzip sakila.zip
cd sakila-db
mysql --host=<HOST> --user=admin --password=<PASSWORD> -f
< sakila-schema.sql
mysql --host=<HOST> --user=admin --password=<PASSWORD> -f
< sakila-data.sql
```

The bash script does the following:

- Installs MariaDB (which includes a MySQL client to enable us to connect to our MySQL server).
- Downloads the Sakila demo database from the MySQL website, and then unzips the file and changes to the `sakila-db` directory.

- Connects to MySQL and runs the SQL content in the `sakila-schema.sql` file, which creates the Sakila schema (database and tables, views, and so on).
 - Connects to MySQL again and runs the SQL content in the `sakila-data.sql` file, which inserts the demo data into the tables in the Sakila database.
8. Click **Next: Add storage** and leave all the default settings as-is.
 9. Click **Next: Add Tags**. Click **Add Tag** and set **Key** to Name and **Value** to `dataeng-book-ec2`.
 10. Click **Next: Configure Security Group** and for **Assign a security group**, choose to **Select an existing security group**.
 11. Select the security group named **default**.
 12. Click **Review and Launch**.
 13. Click **Launch**.
 14. In the pop-up window, select **Create a new key pair** and provide a name for your new key pair (such as `dataeng-book-key`). Then, click **Download Key Pair** and ensure you save the key pair file in a location that you can easily access from the command line.
 15. Click **Launch Instances**:

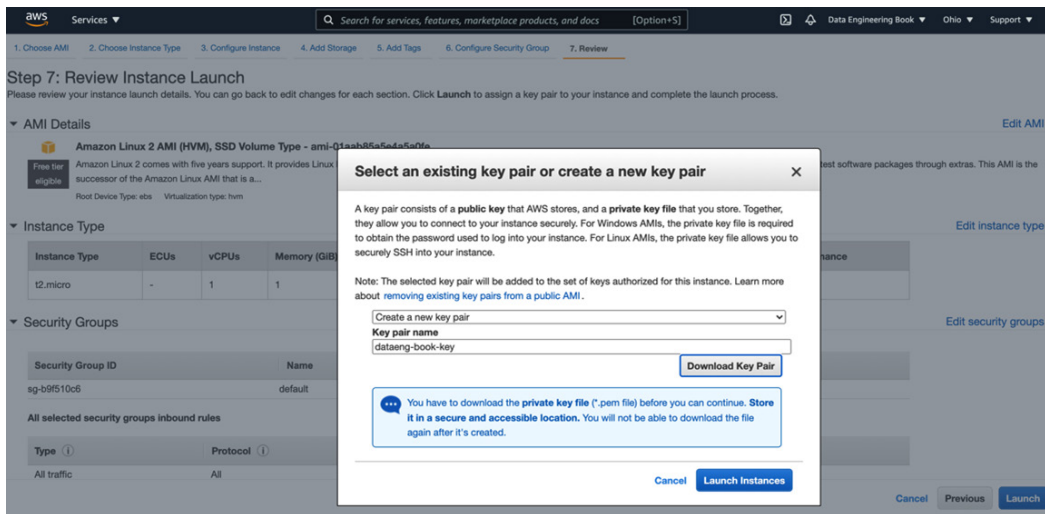


Figure 6.3 – Creating an EC2 instance and specifying the key pair

16. Click **View Instances**.

When the instance launches, it will run the script in the user data to load the Sakila demo database on our MySQL server.

Creating an IAM policy and role for DMS

In this section, we will create an IAM policy and role that will allow DMS to write to our target S3 bucket:

1. In the AWS Management Console, search for and select **IAM** using the top search bar.
2. In the left-hand menu, click on **Policies** and then click **Create policy**.
3. By default, **Visual editor** is selected, so change to a text entry by clicking on the **JSON** tab.
4. Replace the boilerplate code in the text box with the following policy definition. Make sure you replace `<initials>` in the bucket name with the correct landing-zone bucket name you created in *Chapter 3, The AWS Data Engineers Toolkit*:

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

This policy grants permissions for all S3 operations (get, put, and so on) on, and in, `dataeng-landing-zone-<initials>` bucket.

5. Click **Next: Tags**, and then click **Next: Review**.

6. Provide a descriptive policy name, such as `DataEngDMSLandingS3BucketPolicy` and click **Create policy**:

Create policy

1 2 3

Review policy

Name*

Use alphanumeric and '+,=,@-_' characters. Maximum 128 characters.

Description

Maximum 1000 characters. Use alphanumeric and '+,=,@-_' characters.

Summary

This policy defines some actions, resources, or conditions that do not provide permissions. To grant access, policies must have an action that has an applicable resource or condition. For details, choose **Show remaining**. [Learn more](#)

Service	Access level	Resource
<input type="text" value="Q Filter"/>		
Allow (1 of 300 services) Show remaining 299		
S3	Limited: List, Read, Write, Permissions management, Tagging	Multiple

Key	Value

* Required

Cancel

Previous

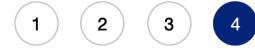
Create policy

Figure 6.4 – Creating an IAM policy to grant S3 permissions

7. In the left-hand menu, click on **Roles** and then click **Create role**.
8. For **Select type of trusted entity**, make sure **AWS service** is selected.
9. From the list of services, select **DMS** and then click **Next: Permissions**.
10. Search for and select the policy you created in *Step 6*, and then click **Next: Tags**.
11. Click **Next: Review**.

- Provide a descriptive role name, such as `DataEngDMSLandingS3BucketRole`, and click **Create role**:

Create role



Review

Provide the required information below and review this role before you create it.

Role name*	<input type="text" value="DataEngDMSLandingS3BucketRole"/>
	<small>Use alphanumeric and '+,.,@-_' characters. Maximum 64 characters.</small>
Role description	<input type="text" value="Allows Database Migration Service to call AWS services on your behalf."/>
	<small>Maximum 1000 characters. Use alphanumeric and '+,.,@-_' characters.</small>
Trusted entities	AWS service: <code>dms.amazonaws.com</code>
Policies	DataEngDMSLandingS3BucketPolicy
Permissions boundary	Permissions boundary is not set

No tags were added.

* Required Cancel Previous **Create role**

Figure 6.5 – Creating an IAM role to allow DMS to write to S3

- Click on the newly created role and copy and paste the **Role ARN** property somewhere that you can easily access it; it will be required in the next section.

Now that we have created the required IAM permissions, we will create a DMS replication instance, as well as other required DMS resources (such as source and target endpoints, as well as a database migration task).

Configuring DMS settings and performing a full load from MySQL to S3

In this section, we will create a DMS replication instance (a managed EC2 instance that connects to the source endpoint, retrieves data, and writes to the target endpoint), and also configure the source and target endpoints. We will then create a database migration task that provides the configuration settings for the migration.

In the following steps, you will configure DMS and start the full load job:

- In the AWS Management Console, search for **DMS** using the top search bar and click on **Database Migration Service**.
- In the left-hand menu, click on **Replication Instances**.
- At the top of the page, click on **Creation replication instance**.

4. Provide a **Name** for the replication instance; for example, **mysql-s3-replication**.
5. For **Instance class**, select **dms.t3.micro**.
6. For **Allocated storage**, enter 10 (the database we are replicating is very small, so 10 GB is enough space).
7. In the **VPC** dropdown, select the default VPC.
8. For Multi AZ select 'Dev or test workload (Single-AZ)
9. Leave everything else as the defaults and click **Create**.
10. In the left-hand menu, click on **Endpoints**.
11. At the top right, click on **Create endpoint**.
12. For **Endpoint type**, select **Source endpoint** and then click the box for **Select RDS DB Instance**.
13. For **RDS Instance**, use the drop-down list to select the MySQL database you created previously.
14. Under **Endpoint configuration**, for **Access to endpoint database**, select **Provide access information manually**.
15. For **Password**, provide the password that you set for the database in *Step 9* of the *Creating a new MySQL database instance* section.
16. Select **Create endpoint** at the bottom right.
17. Now that we have created the source endpoint, we can create the target endpoint by clicking on **Create endpoint** at the top right.
18. For **Endpoint type**, select **Target endpoint**.
19. For **Endpoint identifier**, type in a name for the endpoint, such as `s3-landing-zone-sakilia-csv`.
20. For **Target engine**, select **Amazon S3** from the drop-down list.
21. For **Service access role ARN**, enter the **Amazon Resource Name (ARN)** for the IAM role you recorded in *Step 13* of the previous section.
22. For **Bucket name**, provide the name of the landing zone bucket you created in *Chapter 3, The AWS Data Engineers Toolkit* (for example, `dataeng-landing-zone-<initials>`).
23. For **Bucket folder**, enter `sakila-db`.

24. Expand Endpoint settings, and click on Add new setting. Select 'AddColumnName' from the settings list, and for value type True.

Endpoint configuration

Endpoint identifier [Info](#)
A label for the endpoint to help you identify it.

Descriptive Amazon Resource Name (ARN) - optional
A friendly name to override the default DMS ARN. You cannot modify it after creation.

Target engine
The type of database engine this endpoint is connected to.

Amazon S3 ▼

Service access role ARN
Role that can access target

Bucket name
The name of an Amazon S3 bucket where DMS will read the files from

Bucket folder
The Amazon S3 bucket path where the CSV files can be found

▼ **Endpoint settings**

Define additional specific settings for your endpoints using wizard or editor. [Learn more](#)

Wizard
Enter endpoint settings using the guided user interface.

Editor
Enter endpoint settings in JSON format.

Endpoint settings

Setting	Value - A value is required	
Q AddColumnName X	Q True X	Remove
Add new setting		

Use endpoint connection attributes

Figure 6.6 – AWS DMS S3 target endpoint

25. Click **Create Endpoint**.
26. On the left-hand side, click **Database migration tasks**, and then click **Create task**.
27. For **Task identifier**, provide a descriptive name for the task, such as `dataeng-mysql-s3-sakila-task`.

28. For **Replication instance**, select the instance you created in *Step 4* of the previous section, such as **mysql-s3-replication**.
29. For **Source database endpoint**, select the source endpoint you created in *Step 11* of the previous section, such as **dataeng-mysql-1**.
30. For **Target database endpoint**, select the target endpoint you created in *Step 17* of the previous section, such as **dataeng-s3-clean-sakila-parquet**.
31. For **Migration type**, select **Migrate existing data** from the dropdown. This does a one-time migration from the source to the target.
32. Leave the defaults for **Task settings** as-is.
33. For **Table mappings**, under **Selection rules**, click **Add new selection rule**.
34. For **Schema**, select **Enter a schema**. Leave **Schema name** and **Table name** set as %.
35. Leave the defaults for **Selection rules** and all other sections as-is and click **Create task**.
36. Once the task has been created, the full load will be automatically initiated and the data will be loaded from your MySQL instance to Amazon S3. Click on the task identifier and review the **Table statistics** tab to monitor your progress.

Our previously configured S3 event for all CSV files written to the landing zone bucket will be triggered for each file that DMS loads. This will run the Lambda function we created in Chapter 3 which will create a new Parquet version of each file in the CLEAN ZONE bucket. This will also register each table in the AWS Glue data catalog.

Querying data with Amazon Athena

The Lambda function that was run for each CSV file created by DMS, also registers each new Parquet file as part of a table in the AWS Glue Database.

We can now query the newly ingested data using the Amazon Athena service. .

1. First we need to create a new Amazon S3 folder to store the results of our Athena queries. In the AWS Management Console, search for and select S3 using the top search bar.
2. Click on Create bucket, and for Bucket name enter athena-query-results-
<INITIALS>. Make sure the AWS Region is set to the region you have been using for the previous hands-on exercises. Leave all other defaults, and click on Create bucket.
3. In the AWS Management Console, search for and select **Athena** using the top search bar.

4. Expand the left-hand panel, and click on Query Editor.
5. On the Athena dashboard page, click on Explore the query editor, and then click on the Settings tab.
6. Click on Manage on the settings tab, and for Location of query result provide the path of the bucket we just created, and then click Save.
7. Return to the Editor tab, and then in the Database dropdown on the left-hand side, select `sakila` from the drop-down list.
8. In the **New query** window, run the `select * from film limit 20;` query.
9. This query returns the results of the first 20 fictional films in the Sakila database.
10. Our DMS replication instance does have a low cost per hour while it is running. So, now that we have completed our database replication to Amazon S3 and confirmed the success of this task by querying data with Athena, we can delete the replication instance. Open up the DMS service, and on the left-hand side click on Database migration tasks. We need to delete the task before we can delete the associated replication instance, so select the task, and from the Actions menu click Delete, and then confirm deletion in the pop-up box.
11. Once the replication task has been deleted, on the left-hand side, click on Replication instances. Select the replication instance you created earlier, and then from the Actions menu, select Delete. Confirm that you want to delete the replication instance by clicking on Delete in the pop-up box.

Congratulations! You have successfully replicated a MySQL database into your S3-based data lake. To learn more about ingesting data from MySQL to Amazon S3, see the following AWS documentation:

- Using Amazon S3 as a target for AWS Database Migration Service (https://docs.aws.amazon.com/dms/latest/userguide/CHAP_Target.S3.html)
- Using a MySQL-compatible database as a source for AWS DMS (https://docs.aws.amazon.com/dms/latest/userguide/CHAP_Source.MySQL.html)

Now that we have got hands-on with ingesting batch data from a database into our Amazon S3 data lake, let's look at one of the ways to ingest streaming data into our data lake.

Hands-on – ingesting streaming data

Earlier in this chapter, we looked at two options for ingesting streaming data into AWS, namely Amazon Kinesis and Amazon MSK. In this section, we will use the serverless Amazon Kinesis service to ingest streaming data. To generate streaming data, we will use the open source Amazon Kinesis Data Generator (KDG) In this section:

- Configure **Amazon Kinesis Data Firehose** to ingest streaming data, and write the data out to Amazon S3.
- Configure **Amazon KDG** to create a fake source of streaming data.

To get started, let's configure a new Kinesis Data Firehose to ingest streaming data and write it out to our Amazon S3 data lake.

Configuring Kinesis Data Firehose for streaming delivery to Amazon S3

Kinesis Data Firehose is designed to enable you to easily ingest data from streaming sources, and then write that data out to a supported target (such as Amazon S3, which we will do in this exercise). Let's get started:

1. In the AWS Management Console, search for and select **Kinesis** using the top search bar.
2. The Kinesis landing page provides links to create new streams using the Kinesis features of Kinesis Data Streams, Kinesis Data Firehose, or Kinesis Data Analytics. Select the Kinesis Data Firehose service, and then click on Create delivery stream.
3. In this exercise, we are going to use the KDG to send data directly to Firehose, so for Source, select Direct PUT from the drop-down list. For Destination, select Amazon S3 from the drop-down list.
4. For **Delivery stream name**, enter a descriptive name, such as `dataeng-firehose-streaming-s3`.
5. For **Transform records with AWS Lambda**, leave the default of **Disabled** as-is. This functionality can be used to run data validation tasks or perform light processing on incoming data with AWS Lambda, but we want to ingest the data without any processing, so we will leave this disabled.
6. For **Convert record format**, we will also leave the default of **Disabled** as-is. This can be used to convert incoming data into Apache Parquet or Apache ORC format. However, to do this, we would need to specify the schema of the incoming data upfront. We are going to ingest our data without changing the file format, so we will leave this disabled.

7. For **S3 bucket**, select the Landing Zone bucket you created previously; for example, `dataeng-landing-zone-<initials>`.
8. By default, Kinesis Data Firehose writes the data into S3 with a prefix to split incoming data by `YYYY/MM/dd/HH`. For our dataset, we want to load streaming data into a `streaming` prefix, and we only want to split data by the year and month that it was ingested. Therefore, we must set **S3 bucket prefix** to `streaming/!{timestamp:yyyy/MM/}`. For more information on custom prefixes, see <https://docs.aws.amazon.com/firehose/latest/dev/s3-prefixes.html>.
9. If we set a custom prefix for incoming data, we must also set a custom error prefix. Set **S3 bucket error output prefix** to `!{firehose:error-output-type}/!{timestamp:yyyy/MM/}`.
10. Expand the Buffer hints, compression and encryption section
11. The S3 buffer conditions allow us to control the parameters for how long Kinesis buffers incoming data, before writing it out to our target. We specify both a buffer size (in MB) and a buffer interval (in seconds), and whichever is reached first will trigger Kinesis to write to the target. If we used the maximum buffer size of 128 MB and a maximum buffer interval of 900 seconds (15 minutes), we would see the following behavior. If we receive 1 MB of data per second, Kinesis Data Firehose will trigger after approximately 128 seconds (when 128 MB of data has been buffered). On the other hand, if we receive 0.1 MB of data per second, Kinesis Data Firehose will trigger after the 900-second maximum buffer interval. For our use case, we will set **Buffer size** to 1 MB and **Buffer interval** to 60 seconds.
12. For all the other settings, leave the default settings as-is and click on **Create delivery stream**.
13. Our Kinesis Data Firehose stream is now ready to receive data. So, in the next section, we will generate some data to send to the stream using the **KDG (KDG)** tool.

Configuring Amazon Kinesis Data Generator (KDG)

Amazon **KDG** is an open source tool from AWS that can be used to generate customized data streams and can send that data to Kinesis Data Streams or Kinesis Data Firehose.

The Sakila database we previously loaded was for a company that produced classic movies and rented those out of their DVD stores. The DVD rental stores went out of business years ago, but the owners have now made their classic movies available for purchase and rental through various streaming platforms.

The company receives information about their classic movies being streamed from their distribution partners in real time, in a standard format. Using KDG, we will simulate the streaming data that's received from partners, including the following:

- Streaming timestamp
- Whether the customer rented, purchased, or watched the trailer
- `film_id` that matches the Sakila film database
- The distribution partner name
- Streaming platform
- The state that the movie was streamed in

KDG is a collection of HTML and JavaScript files that run directly in your browser and can be accessed as a static site in GitHub. To use KDG, you need to create an Amazon Cognito user in your AWS account, and then use that user to log into KDG on the GitHub account.

AWS had created an Amazon CloudFormation template that you can deploy in your AWS account to create the required Amazon Cognito user. This CloudFormation template creates an AWS Lambda function in your account to perform the required setup.

Follow these steps to deploy the CloudFormation template, create the required Cognito user, and configure the KDG:

1. Open the KDG help page in your browser by going to <https://awslabs.github.io/amazon-kinesis-data-generator/web/help.html>.
2. Read the information about how the CloudFormation template works to create Cognito credentials in your account. When you're ready, click on the **Create a Cognito user with CloudFormation** button.
3. The AWS Management Console will open to the CloudFormation **Create Stack** page. When opening the link, the region may default to Oregon (us-west-2-), so if necessary, change **region** to the region you are using for the exercises in this book, and then accept the CloudFormation defaults and click **Next**.
4. On the **Specify stack details** page, provide a **Username** and **Password** for your Cognito user and click **Next**.
5. For **Configure stack options**, leave all the default settings as-is and click **Next**.
6. Review the details of the stack to be created, and then click the box to acknowledge that AWS CloudFormation may create IAM resources. Click **Create stack**.

Refresh the web page and monitor it until the stack's status is `CREATE_COMPLETE`.

7. Once the stack has been successfully deployed, go to the **Outputs** tab and take note of the `KinesisDataGeneratorUrl` value. Click on the link and open a new tab.
8. Use the username and password you set as parameters for the CloudFormation template to log into the Amazon KDG portal.
9. Set **Region** to be the same region where you created the Kinesis Data Firehose delivery stream.
10. For **Stream/delivery stream**, from the dropdown, select the Kinesis Data Firehose stream you created in the previous section.
11. For **Records per second**, set this as a constant of 10 records per second.
12. For the record template, we want to generate records that simulate what we receive from our distribution partners. Paste the following into the template section of the KDG:

```
{
  "timestamp": "{{date.now}}",
  "eventType": "{{random.weightedArrayElement(
    {
      "weights": [0.3,0.1,0.6],
      "data": ["rent","buy","trailer"]
    }
  )}}",
  "film_id":{{random.number(
    {
      "min":1,
      "max":1000
    }
  )}},
  "distributor": "{{random.arrayElement(
    ["amazon prime", "google play", "apple itunes",
    "vudo", "fandango now", "microsoft", "youtube"]
  )}}",
  "platform": "{{random.arrayElement(
    ["ios", "android", "xbox", "playstation", "smart
    tv", "other"]
  )}}",
```

```
"state": "{address.state}"  
}
```

13. Click **Send data** to start sending streaming data to your Kinesis Data Firehose delivery stream. Because of the configuration that we specified for our Firehose stream, the data we are sending is going to be buffered for 60 seconds, and then a batch of data written to our Landing Zone S3 bucket. This will continue for as long as we leave the KDG running.
14. Allow KDG to send data for 5-10 minutes, and then click on **Stop Sending Data to Kinesis**.

During the time that the KDG was running, it will have created enough data for us to use in later chapters, where we will join this data with data we migrated from our MySQL database.

We can now use a Glue crawler to create a table in our data catalog for the newly ingested streaming data.

Adding newly ingested data to the Glue Data Catalog

In this section, we will run a Glue crawler to examine the newly ingested data, infer the schema, and automatically add the data to the Glue catalog. Once we do this, we can query the newly ingested data using services such as Amazon Athena. Let's get started:

1. In the AWS Management Console, search for and select **Glue** using the top search bar.
2. In the left-hand menu, click on **Crawlers**.
3. Click on **Add crawler**.
4. Enter a descriptive name for **Crawler name**, such as `dataeng-streaming-crawler`.
5. Leave the defaults for **Specify crawler source type** as-is and click on **Next**.
6. For **Add a data store**, enter the path for the new data in **Include Path** (or click the folder icon to browse your S3 folders and select from there). For **Include Path**, the folder should be similar to `s3://dataeng-landing-zone-<initials>/streaming`.
7. Click **Next**, then for **Add another data store**, leave the default of **No** as-is and click **Next**.

8. For **Choose an IAM role**, leave the default to create a new IAM role as-is and enter a suffix for the IAM role (such as `glue-crawler-streaming-data-role`). Then, click **Next**.
9. For **Create a schedule for this crawler**, leave the default of **Run on-demand** as-is and click **Next**.
10. For **Configure the crawler's output**, click on **Add database** to create a new Glue catalog database for storing streaming data tables.
11. On the **Add database** page, provide a descriptive database name, such as `streaming-db`, and click on **Create**. Then, click **Next**.
12. Review the Glue crawler settings and click **Finish**.
13. Select your new crawler from the list and click **Run crawler**.

When the crawler finishes running, it should have created a new table for the newly ingested streaming data.

Querying the data with Amazon Athena

Now that we have ingested out new streaming data and added the data to the AWS Glue data catalog using the AWS Glue crawler, we can query the data using Amazon Athena:

1. In the AWS Management Console, search for and select **Athena** using the top search bar.
2. On the left-hand side, from the **Database** drop-down list, select the database you created in the previous step (such as `streaming-db`).
3. In the query window, type in `select * from streaming limit 20`.

The result of the query should show 20 records from the newly ingested streaming data, matching the pattern that we specified for KDG. Note how the Glue Crawler automatically added the YYYY and MM prefixes we created as partitions.

Summary

In this chapter, we reviewed several ways to ingest common data types into AWS. We reviewed how AWS DMS can be used to replicate a relational database to S3, and how Amazon Kinesis and Amazon MSK can be used to ingest streaming data.

In the hands-on section of this chapter, we used both the AWS DMS and Amazon Kinesis services to ingest data and then used AWS Glue to add the newly ingested data to the data catalog and query the data with Amazon Athena.

In the next chapter, *Chapter 7, Transforming Data to Optimize for Analytics*, we will review how we can transform the ingested data to optimize it for analytics, a core task for data engineers.

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 and to create value for an organization.

Transforming data to optimize for analytics and to 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. However, 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:

- An overview of how transformations can create new valuable datasets
- A look at the different types of transformation engines available
- Common data-preparation transformations
- Common business use case transformations
- How to handle **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/tree/main/Chapter07>

Transformations – making raw data more valuable

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 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.

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:

- **Data consumers:** Who was going to be consuming the data we created and what tools would they use for data gathering (our guests)?
- **Data sources:** Which data sources did we have access to that we could use to create our new dataset (our raw ingredients)?
- **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 *Chapter 3, The AWS Data Engineer's Toolkit*, so make sure to review that chapter, 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 a managed service for deploying a cluster for running Spark. 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 running 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. Apache 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 many 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. In addition to Apache Hive, one of the other Hadoop tools is the **Hadoop Distributed File System (HDFS)**, and this is still commonly used as the ingest and target storage for Apache Spark processing jobs.

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 tools 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 intermediary 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 generally do 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 to developing Apache Spark transformations. This can be used by people who do not have any current experience with Spark but 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.

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, Cataloging, Security, and Governance*. 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 used to transform PII data for anonymization, there may be multiple different toolsets that can be used. This includes open source libraries to create an Apache Spark job to do the anonymization, and, as you already know, that could be run on AWS Glue or Amazon EMR. If your requirements are just for a simple **SHA-256 hash** of a column, you can achieve this by creating a new table using **Amazon Athena**, as outlined in the AWS blog post *Anonymize and manage data in your data lake with Amazon Athena and AWS Lake Formation*. Note, however, that an SHA-256 hash is often not regarded as a secure way of anonymizing data – for example, a court in Germany ruled that using an SHA-256 hash to anonymize data was not sufficient to comply with privacy requirements. For a more secure way to anonymize data, or 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 columnar-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, both in terms of storage space used and for 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 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 in 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=2021/file1.parquet
datalake_bucket/year=2020/file1.parquet
datalake_bucket/year=2019/file1.parquet
datalake_bucket/year=2018/file1.parquet
```

If you then run a SQL query and include a `WHERE Year = 2018` clause, for example, the analytics engine only needs to open up the single file in the `datalake_bucket/year=2018` folder. Because less data needs to be scanned by the query, it costs less and completes quicker.

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 Parquet files in a data lake is 128 MB–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, reading metadata, scanning data, and closing each file, and can significantly impact performance.

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:

- **Ensuring consistent column names:** When ingesting data from multiple datasets, you may find that the same data in different datasets have 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`.
- **Changing column data type:** 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.
- **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 of `DD-MM-YYYY`, or whatever your corporate standard is for analytics.
- **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.
- **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.

Business use case transforms

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 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 Analytics 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 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 to understand 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 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 preprocess 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.

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 that 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.

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 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 the 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 load of data is completed to the data lake, 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):

- **I – Insert:** This indicates that this row contains data that was a new insert to the table.
- **U – Update:** This indicates that this row contains data that updates an existing record in the table.
- **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 to individual records within the 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 traditional approaches to handling updates to data in a data lake.

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 in existing data to 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 that 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 runs a 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 – the transactional data lake

Over the past few years, the concept of a *transactional* data lake has become popular, and a number of different companies and organizations have created new **table formats** to support the goal of **transactional data lakes**. 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 table formats bring **ACID** semantics to the data lake:

- **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
- **Consistency:** The expectation that even if a failure occurs, the dataset will stay consistent
- **Isolation:** The expectation that one transaction on the dataset will not be affected by another transaction that is requested at the same time
- **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 do 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 potentially 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 updates to a dataset, and providing a consistent, up-to-date view of data with lower latency, is handled by these solutions.

Let's have a brief look at some of the most common offerings for these new styles of transactional data lakes.

AWS Lake Formation governed tables

In December 2020, AWS announced the public preview of new functionality for **Lake Formation** with the introduction of **governed tables**. This new Amazon S3 table type has been designed to support ACID transactions within an S3-based data lake environment. When a table is created and configured as a governed table, Lake Formation handles the complexities of allowing multiple users to simultaneously and reliably insert, delete, and modify records across these tables. In addition, Lake Formation works behind the scenes to automatically compact and optimize the files behind the table on a regular basis.

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.

Apache Iceberg

Apache Iceberg was created by engineers at **Netflix** and **Apple**, and is designed as an open-table format for very large datasets. The code was donated to the Apache Software Foundation and became a top-level project in May 2020.

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. At the time of writing, the platform has not yet been launched, but the founders have secured Series A funding for their start-up.

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-format storage layer 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 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 emerged to handle these challenges, but each organization had to effectively *reinvent the wheel* to implement its 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 solutions, as outlined in this section.

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 Glue job that joins the streaming data with the 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 the 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, written 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:

1. Log into the **AWS Management Console** (<https://console.aws.amazon.com>).
2. In the **top search bar**, search for and select **S3** to access the S3 console.
3. In the top right, click on **Create bucket**.
4. For bucket name, enter `dataeng-curated-zone-<initials>`, replacing `<initials>` with your initials as you did for the landing zone and clean zone.
5. Ensure the region is set to the region you have been using for the other hands-on exercises. For the examples in this book, we use `us-east-2` (Ohio).

6. Accept all other defaults and click **Create bucket**.
7. In the **top search bar**, search for and select **Glue** to access the Glue console.
8. On the left-hand side, select **Databases**, and then click **Add database**.
9. For **Database name**, type `curatedzonedb`, and then click **Create**.

We have now created a new curated zone 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:

- Read our source S3 bucket (for example, `dataeng-landing-zone-<initials>` and `dataeng-clean-zone-<initials>`)
- Write to our target S3 bucket (for example, `dataeng-curated-zone-<initials>`)
- Access to Glue temporary directories
- Write logs to **Amazon CloudWatch**
- 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. 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 service select **Glue**, and then click **Next: Permissions**. Listing Glue as a trusted entity for this role enables the AWS Glue service to assume this role to run transformations.
9. Under **Attach permissions** policies, select the policy we just created (for example, `DataEngGlueCWS3CuratedZoneWrite`) by searching and then clicking in the tick box.
10. Also, search for `AWSGlueServiceRole` and click on the tick box to select this role. This managed policy provides access to temporary directories used by Glue, as well as **CloudWatch** logs and Glue resources.
11. Then, click **Next: Tags**.
12. Provide any tags you would like associated with this policy (optional) and click **Next: Review**.
13. Provide a role name, such as `DataEngGlueCWS3CuratedZoneRole`, and click **Create role**.

We have now created 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 about 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:

- `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.
- `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.
- `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, under the ETL section, click on **AWS Glue Studio**. Expand the left-hand panel, and click on **Jobs**.
3. Select the option for Visual with a blank canvas, and click **Create**.
4. Click on the **Source** dropdown, and then select **S3**.
5. On the right-hand side, under **Data source properties – S3**, ensure **Data Catalog table** is selected, and from the dropdown select the `sakila` database.
6. For the **Table** dropdown, select `film_category`.
7. Click on the **Node properties** tab in the transform designer and set **Name** to `S3 – Film-Category`.

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

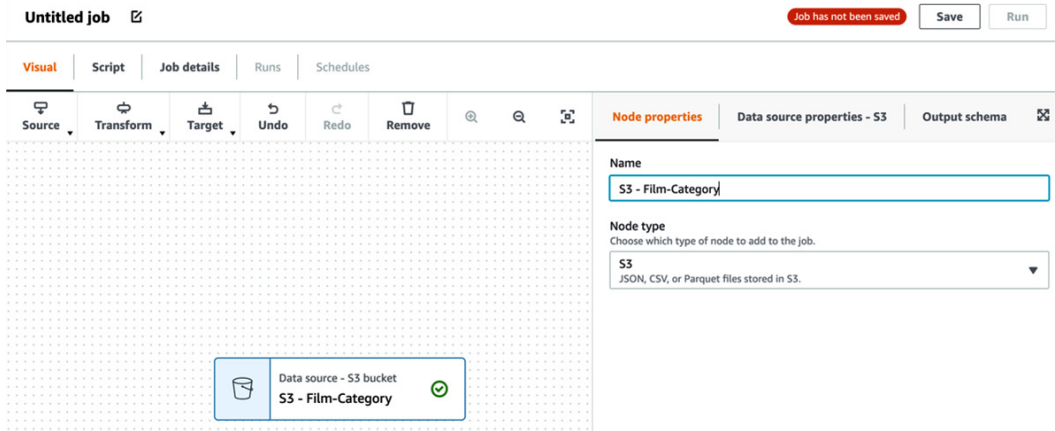


Figure 7.4 – Glue Studio with first S3 data source

- Repeat *steps 4-7*, adding another S3 source for the `film` table, and under Node properties, set the Name to `S3 - Film`. Once done, your Glue studio screen should look as follows:

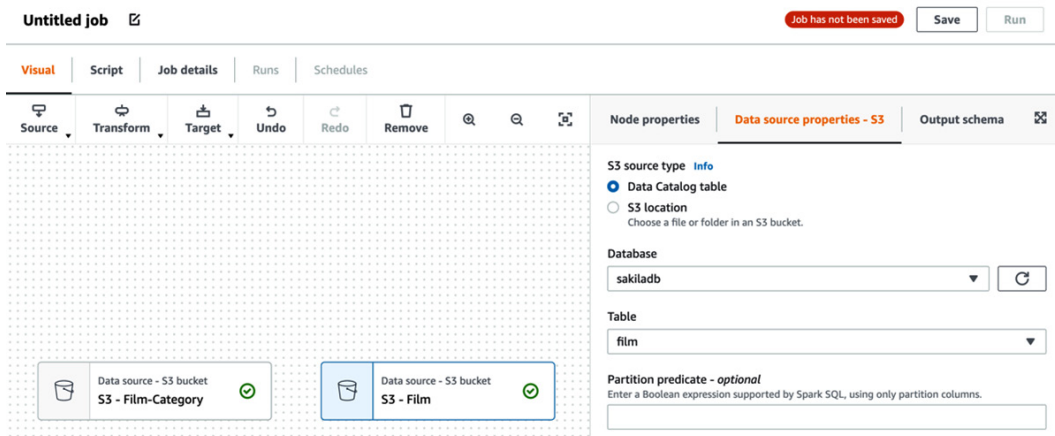


Figure 7.5 – Glue Studio with two S3 data sources

- In the **Designer** window, click on the **Transform** dropdown and 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, click on **Node properties**, and use the **Node parents** dropdown to select the **S3 – Film** and **S3 – Film-Category** tables.
11. You will see a red check-mark 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 (`film_category`). Click on **Resolve it** to have Glue Studio automatically add a new transform that renames the columns in the right-hand table (`film_category`).
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:

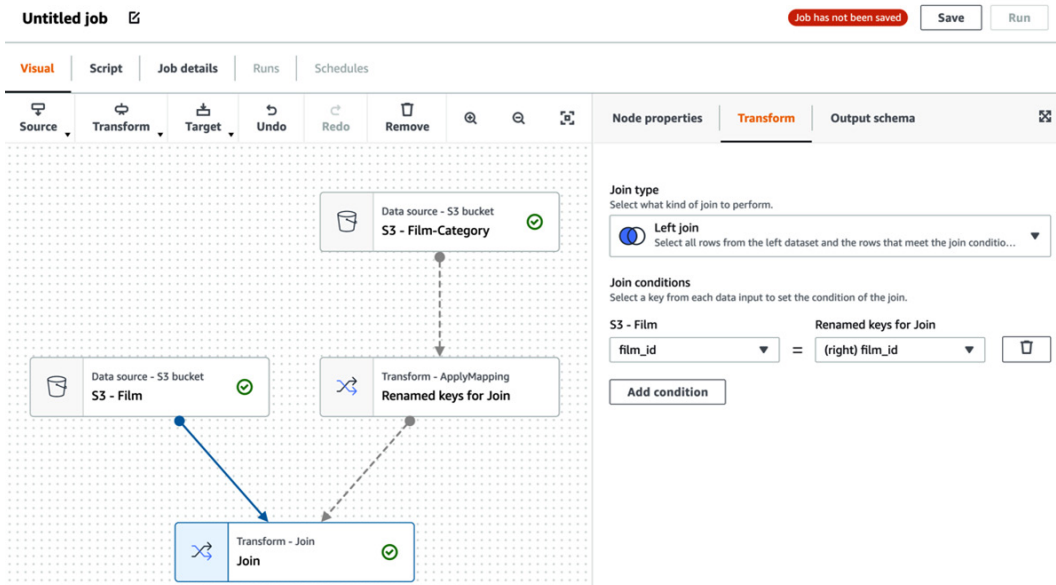


Figure 7.6 – Glue Studio after first table join

13. Let's provide a name for the temporary table created as a result of the join. On the **Node properties** tab, change the 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 **ApplyMapping** transform to drop the columns we don't need, rename fields, and so on. From the **Transform** menu, select **ApplyMapping**.
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:
 - `rental_duration`
 - `rental_rate`
 - `replacement_cost`
 - `last_update`
 - `(right) film_id`
 - `(right) last_update`
16. We can now add a transform, which will join the results of the **ApplyMapping** transform with our category table, adding the name of the category for each film. To add the **Category** table into our transform, from the **Source** drop-down menu, select **S3**. For **Database**, select `sakila`, and for **Table**, select **Category**. To provide a descriptive name, on the **Node properties** tab, change the **Name** to **S3 - Category**.
17. We can now add our final transformation. From the **Transform** drop-down menu, select **Join**.
18. We always need two tables for a join, so from the **Node properties** tab, use the **Node parents** to add the **ApplyMapping** 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 **ApplyMapping** table, select the `(right) category_id` field.

20. Now we will add one last **ApplyMapping** transform, again remove unneeded fields, and rename fields where appropriate. From the **Transform** dropdown, select **ApplyMapping**. Click the checkbox next to the following columns in order to drop them:

- `last_update`
- `(right) category_id`

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 where we want to write out our data to:

1. Add a target by clicking on the Target drop-down, and selecting Amazon S3
2. On the **Data target properties – S3** tab, select **Parquet** for **Format**, and **Snappy** for **Compression type**. Click on **Browse S3** for **S3 Target Location** and select the `dataeng-curated-zone-<initials>` bucket. Add a prefix after the bucket 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:

Node properties	Data target properties - S3	Output schema	Data preview
Format			
Parquet			
Compression Type			
Snappy			
S3 Target Location			
Choose an S3 location in the format <code>s3://bucket/prefix/object/</code> with a trailing slash (/).			
<input type="text" value="s3://dataeng-curved-zone- /filmdb/film_category/"/> <input type="button" value="View"/> <input type="button" value="Browse S3"/>			
Data Catalog update options Info			
Choose how you want to update the Data Catalog table's schema and partitions. These options will only apply if the Data Catalog table is an S3 backed source.			
<input type="radio"/> Do not update the Data Catalog <input checked="" type="radio"/> Create a table in the Data Catalog and on subsequent runs, update the schema and add new partitions <input type="radio"/> Create a table in the Data Catalog and on subsequent runs, keep existing schema and add new partitions			
Database			
Choose the database from the AWS Glue Data Catalog.			
curatedzonedb <input type="button" value="Refresh"/>			
Table name			
Enter a table name for the AWS Glue Data Catalog.			
film_category			
Partition keys - optional			
Add partition keys.			
<input type="button" value="Add a partition key"/>			

Figure 7.7 – Data target properties – S3 configuration

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.
10. For **Job bookmark**, change the setting 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**, change this to 0. If our job fails to run, we don't want it to automatically repeat.
12. Leave all other defaults, and in 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 Amazon S3 and review the output location to validate that the files were created. Also, navigate to the AWS Glue console to confirm that the new table was created in `curatedzoneadb`.

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, under the ETL section, click on **AWS Glue Studio**.
3. Click on **Create and manage jobs**, then select **Blank graph**, and click **Create**.

4. Click on the **Source** dropdown, and then select **S3**.
5. On the right-hand side, under **Data source properties – S3**, ensure **Data Catalog table** is selected, and from the dropdown select the `curatedzonedb` database.
6. For the **Table** dropdown, select `film_category`.
7. Click on the **Node properties** tab in the transform designer, and set **Name** to `S3 - Film_Category`.
8. Repeat *steps 5–7*, adding another S3 source for the streaming table from the `streamingdb` database, and setting the name to `S3 - Streaming`.
9. From the **Transform** dropdown, add an **ApplyMapping** transform for the **S3 – Streaming** data source.
10. Change the name of the `film_id` key to `film_id_streaming` and under **Node properties** set the name to `ApplyMapping - Streaming`.
11. From the **Transform** dropdown, add a **Join** transform and set **Join type** to **Left join**.
12. Under **Node properties**, add the **S3 – Film-Category** data source as a node parent.
13. Under the **Transform** tab, for **Join conditions**, click on **Add condition**.
Select `film_id_streaming` for the left-hand table, and `film_id` for the right-hand table.

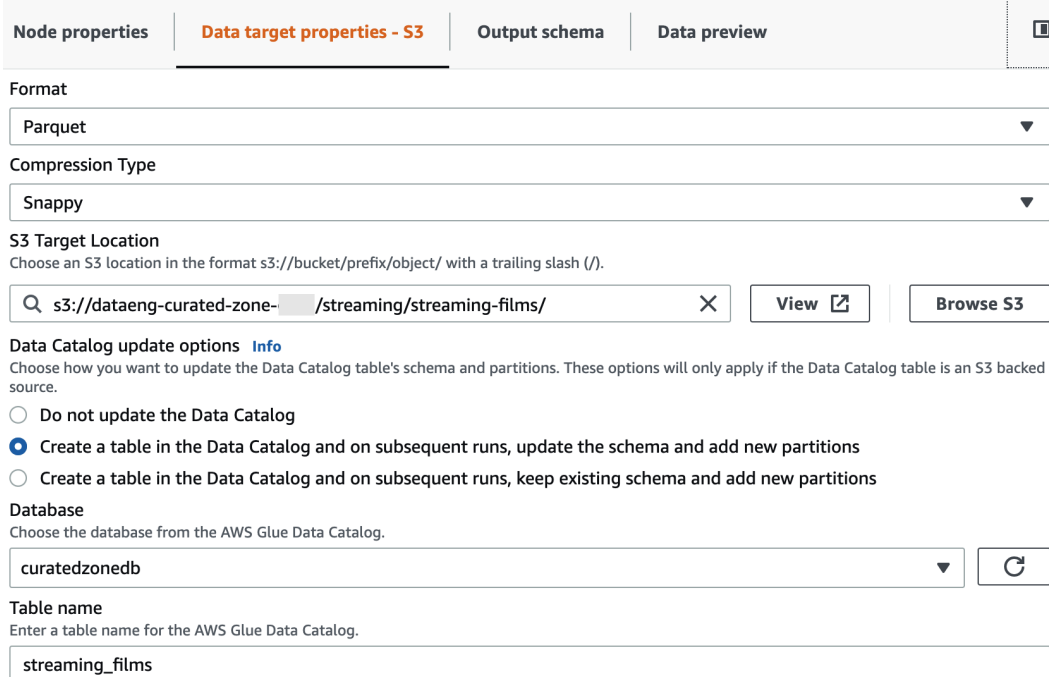
Your Glue Studio visual designer should look as follows:

The screenshot displays the AWS Glue Studio interface. At the top, there's a header with 'Untitled job' and a 'Job has not been saved' warning. Below the header are tabs for 'Visual', 'Script', 'Job details', 'Runs', and 'Schedules'. A toolbar contains icons for Source, Transform, Target, Undo, Redo, and Remove. The main workspace shows a job graph with three nodes: 'Data source - S3 bucket S3 - Streaming', 'Transform - ApplyMapping ApplyMapping - Stre...', and 'Data source - S3 bucket S3 - Film_Category'. Arrows indicate data flow from the two data sources into the ApplyMapping transform, and from the ApplyMapping transform into a 'Transform - Join Join' node. The right-hand panel is set to the 'Transform' tab for the 'Join' node. It shows the 'Join type' set to 'Left join' and 'Join conditions' configured with 'ApplyMapping - Streaming' on the left and 'S3 - Film_Category' on the right, with the condition 'film_id_streaming = film_id'. An 'Add condition' button is visible at the bottom of the panel.

Figure 7.8 – Glue Studio interface showing the first join

14. Click on the Target drop-down, and select Amazon S3. For **Format**, select **Parquet** from the dropdown, and for **Compression type**, select **Snappy**.
15. For **S3 Target Location**, click **Browse S3**, select the `dataeng-curatedzone-<initials>` bucket, and click **Choose**. Add a prefix after the bucket of `/streaming/streaming-films/`.
16. For **Data Catalog update options**, select **Create a table in the Data Catalog, and on subsequent runs, update the schema and add new partitions**.
17. For **Database**, select `curatedzonedb` from the drop-down list.
18. For **Table name**, type in `streaming_films`.

Our **Data Target Properties – S3** configuration should look as follows:



The screenshot shows the AWS Glue Studio configuration interface for a Data Target. The 'Data target properties - S3' tab is active. The configuration is as follows:

- Format:** Parquet
- Compression Type:** Snappy
- S3 Target Location:** s3://dataeng-curated-zone-.../streaming/streaming-films/
- Data Catalog update options:**
 - Create a table in the Data Catalog and on subsequent runs, update the schema and add new partitions
 - Do not update the Data Catalog
 - Create a table in the Data Catalog and on subsequent runs, keep existing schema and add new partitions
- Database:** curatedzonedb
- Table name:** streaming_films

Figure 7.9 – Glue Studio interface showing target configuration

19. 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.
20. Set the name of the job to be `Streaming Data Film Enrichment`.
21. For **IAM Role**, from the dropdown select the role we created previously (`DataEngGlueCWS3CuratedZoneRole`).
22. For **Number of workers**, change this to 2.
23. For **Job bookmark**, change the setting to **Disable**.
24. For **Number of retries**, change this to 0.
25. Leave all other defaults, and in the top right click on **Save**. Then click on **Run** to run the transform job.
26. Click on the **Runs** tab in order to monitor the job run.
27. When the job completes, navigate to Amazon S3 and review the output location to validate that the files were created. Also, navigate to the AWS Glue console to confirm that the new table was created in `curatedzonedb`.

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 the business transforms to enrich and denormalize datasets.

This chapter is 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.

8

Identifying and Enabling Data Consumers

A data consumer can be defined as a person, or application, within an organization that needs access to data. Data consumers can vary from staff that pack shelves and need to know stock levels, to the CEO of an organization that needs data to make a decision on which projects to invest in. A data consumer can also be a system that needs data from a different system.

Everything a data engineer does is to make datasets useful and accessible to data consumers, which, in turn, enables the business to gain useful insights from their data. This means delivering the right data, via the right tools, to the right people or applications, at the right time, to enable the business to make informed decisions.

Therefore, when designing a data engineering pipeline (as covered in *Chapter 5, Architecting Data Engineering Pipelines*), data engineers should start by understanding business objectives, including who the data consumers are and what their requirements are.

We can then work backward from these requirements to ensure that we use the appropriate tools to ingest data at the required frequency (streaming or batch, for example). We can also ensure that we create transformation pipelines that transform raw data sources into data that meets the consumer's specific requirements. And finally, understanding our data consumers will guide us in selecting a target location and format for our transformed data that is compatible with the tools that best enable our data consumers.

Maintaining an understanding of how the data is consumed, as well as knowledge of any downstream dependencies, will also help data engineers support different types of data consumers as they work with a variety of datasets.

In this chapter, we will do a deep dive into data consumers by covering the following topics:

- Understanding the impact of data democratization
- Meeting the needs of business users with data visualization
- Meeting the needs of data analysts with structured reporting
- Meeting the needs of data scientists and ML models
- Hands-on – transforming data using AWS Glue DataBrew

Technical requirements

For the hands-on exercise in this chapter, you will need permission to use the AWS Glue DataBrew service. You will also need to have access to the AWS Glue Data Catalog and any underlying Amazon S3 locations for the databases and tables that were created in the previous chapters.

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/tree/main/Chapter08>

Understanding the impact of data democratization

At a high level, business drivers have not changed significantly over the past few decades. Organizations are still interested in understanding market trends, customer behavior, increasing customer retention, improving product quality, and improving speed to market. However, the analytics landscape, the teams and individual roles that deliver business insights, and the tools that are used to deliver business value have evolved.

Data democratization – the enhanced accessibility of data for a growing audience of users, in a timely and cost-efficient manner – has become a standard expectation for most businesses. Today's varied data consumers expect to be able to get access to the right data promptly using their tool of choice to consume the data.

In fact, as datasets increase in volume and velocity, their gravity will attract more applications and consumers. This is based on the concept of *data gravity*, a term coined by Dave McCrory, which suggests that data has mass. That is, as datasets increase in size, they attract more users and become more difficult to move.

To not be inhibited by a dataset's mass, a modern data pipeline should be based on a storage solution that allows users to interact with data in place, minimizing any heavy lifting and latency associated with moving data. And, due to data democratization and the existence of data gravity, both analytic teams and business users require access to more data, and a greater variety of data, at a faster rate to stay competitive. In effect, organizations have an increasing thirst for data.

A growing variety of data consumers

Over the past few years, we have seen an increase in the number and type of data consumers within an organization, and these data consumers are constantly looking for new data sources and tools. As a result, in today's modern organizations, we can expect to find a wide variety of data consumers – from traditional business users and data analysts to data scientists, machine-to-machine applications, as well as new types of business users.

Beyond just the ability to run SQL queries and generate scheduled reports based on a pre-existing dataset, we see data analysts that also want the ability to do ad hoc data cleansing and exploration, as well as the ability to join structured data with semi-structured data or metadata extracted from unstructured data. For example, they may want to evaluate sales trends concerning social media.

And business users now expect dashboards to be refreshed with real, or near-real-time, data. They also want these dashboards to be accessible from anywhere, on a plethora of mobile devices. Furthermore, they are interested in more than just sales or ERP data. Analysts and business users are interested in social media data to identify consumer trends, and insurance and real estate companies are looking for data to be extracted from documents (such as medical reports or property appraisals). In the manufacturing industry, a variety of data consumers want access to data that's been collected from machines, devices, and vehicles for use cases such as proactively anticipating maintenance requirements.

Data consumers are also no longer limited to individual humans or teams. We are seeing a growing need for business applications to access data, be fed data, or be triggered based on an event or trend in the data. Call centers are interested in real-time transcripts of audio calls for sentiment analysis and tagging calls for manager review. They are also looking at applications and integrations that would use real-time call transcriptions, or full-text analysis of corporate documents, to reduce the time agents spend searching for answers. Engagement platforms are mapping the customer journey and using every event delivery (for example, email opened or email ignored) to tailor the customer experience.

Finally, the availability and importance of data scientists is a growing need and role in many companies. They develop machine learning (ML) models that can identify non-obvious patterns in large datasets or make predictions about future behavior based on historical data. Data scientists usually require access to a large volume of raw, non-aggregated data. They also require enough data to train a machine learning model and test the model for accuracy.

Let's take a deeper dive into some of the different types of data consumers that we can find in today's organizations. We will also look at how data engineers can help enable each of these data consumers.

Meeting the needs of business users with data visualization

Some roles within an organization, such as data analysts, have always had easy access to data. For a long time, these roles were effectively gatekeepers of the data, and any "ordinary" business users that had custom data requirements would need to go through the data gatekeepers.

However, over the past few years, the growth of big data has expanded the thirst and need for custom data among a growing number of **business users**. Business users are no longer willing to tolerate having to go through long, formal processes to access the data they need to make decisions. Instead, users have come to demand easier, and more immediate, access to wider sets of data.

To remain competitive, organizations need to ensure that they enable all the decision-makers in their business to have easy and direct access to the right data. At the same time, organizations need to ensure that good data governance is in place, and that data consumers only have access to the data they need (as we discussed in *Chapter 4, Data Cataloging, Security, and Governance*). Data engineers are key to enabling this.

AWS tools for business users

Business users have mixed skill sets, ranging from those that are Excel power users and are comfortable with concepts such as pivot tables, to executives who want easy access to dashboards that provide visualizations that summarize complex data.

As a data engineer, you need to be able to provide solutions that meet the needs of these diverse business users. Within AWS, the primary tool that's used by business users is Amazon QuickSight, a cloud-based **Business Intelligence (BI)** application. QuickSight enables the creation of easy-to-access visualizations, but also provides functionality for advanced users to dig deeper into the data while providing strong security and governance controls. Amazon QuickSight is cloud-based and can easily be provisioned for hundreds, or even thousands, of users in an organization.

A quick overview of Amazon QuickSight

We will do a deep dive into **Amazon QuickSight** in *Chapter 12, Visualizing Data with Amazon QuickSight*, but in this section, we will have a brief look at some of the primary ways that business users can use this tool.

Amazon QuickSight provides interactive access to data for business users, with many different types and styles of charts supported. A dashboard can display data from multiple different data sources, and users can filter data, sort data, and even drill down into specific aspects of a dataset.

Business users can elect to receive **dashboards** via regular emails or can access and interact with dashboards on-demand via the QuickSight portal or the QuickSight mobile app. Dashboards can also be embedded into existing web portals and apps, making these rich data visualizations accessible via existing tools that business users have access to.

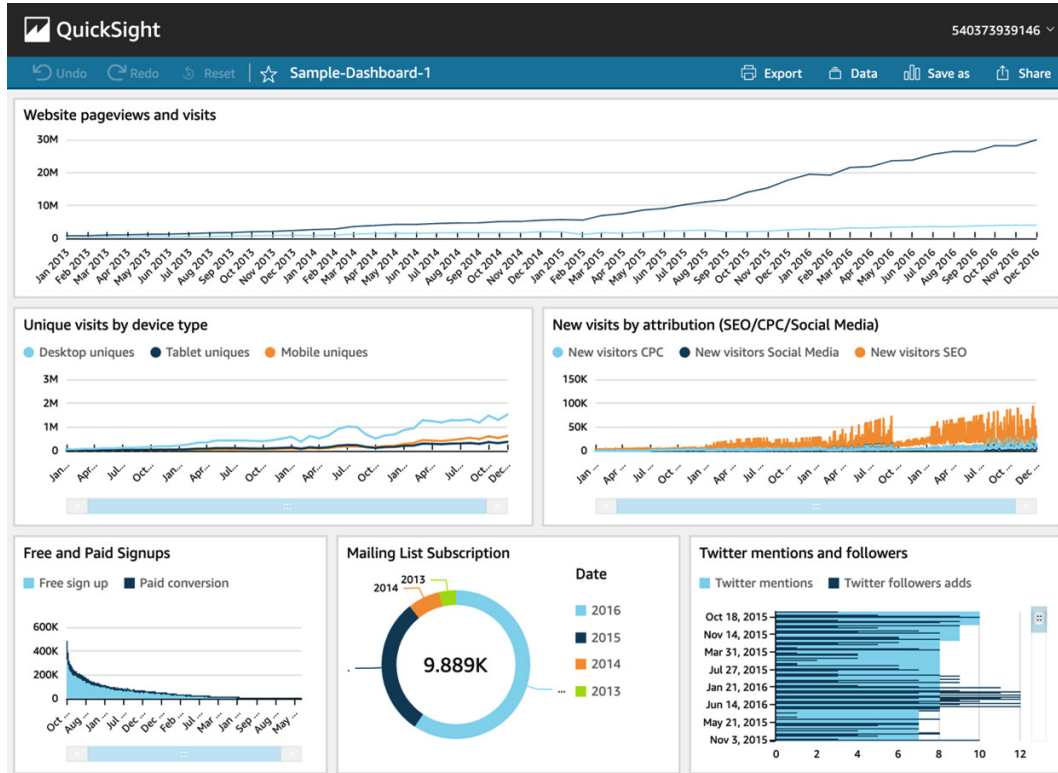


Figure 8.1 – A sample QuickSight dashboard

While some users may have previously used spreadsheets to explore datasets using custom-built charts and pivot tables, QuickSight can provide the same functionality but in a much easier-to-use way. QuickSight also provides security, governance, and auditability, which is not possible when users share ad hoc spreadsheets.

QuickSight can use data from many different sources, including directly from an S3-based data lake, databases (such as Redshift, MySQL, and Oracle), SaaS applications (including Salesforce, ServiceNow, Jira, and others), as well as numerous other sources.

As a data engineer, you may be involved in helping set up QuickSight and may need to configure access to the various data sources. QuickSight users with relevant access can combine different data sources directly, thereby enabling them to build the visualizations the business requires without going through traditional data gatekeepers. However, there may also be times where you are asked to create new datasets in a data lake or data warehouse (such as Redshift or Snowflake) so that QuickSight users can access the required data without needing to combine and transform datasets themselves.

We are now going to move on and explore a different type of data consumer – the data analyst. But for a deeper dive into QuickSight, including a hands-on exercise on creating a QuickSight visual, refer to *Chapter 12, Visualizing Data with Amazon QuickSight*.

Meeting the needs of data analysts with structured reporting

While business users make use of data to make decisions related to their job in an organization, a data analysts' full-time job is all about the data – analyzing datasets and drawing out insights for the business.

If you look at various job descriptions for **data analysts**, you may see a fair amount of variety, but some elements will be common across most descriptions. These include the following:

- Cleansing data and ensuring data quality when working with ad hoc data sources.
- Developing a good understanding of their specific part of the business (sometimes referred to as becoming a domain specialist for their part of the organization). This involves understanding what data matters to their part of the organization, which metrics are important, and so on.
- Interpreting data to draw out insights for the organization (this may include identifying trends, highlighting areas of concern, and performing statistical analysis of data). The data analyst also needs to present the information they've gathered, as well as their conclusions, to business leaders.
- Creating visualizations using powerful BI software (such as Amazon QuickSight) that other business users can then interact with.
- Doing an ad hoc analysis of data using structured query languages such as SQL.

A data analyst is often tasked with doing complex data analysis to answer specific business questions. Examples, as described earlier in this book, include identifying which products are the most popular by different age or socio-economic demographics. Another example is what percentage of customers have browsed the company's e-commerce store more than 5 times, for more than 10 minutes at a time, in the last 2 weeks, but have not purchased anything.

At times, a data analyst may make use of data in the data lake that has already been through formal data engineering pipelines, which means it has been cleaned and checked for quality. At other times, a data analyst may need to ingest new raw data, and in these cases, they may be responsible for data cleansing and performing quality checks on the data.

Some of the work a data analyst does may be to use ad hoc SQL queries to answer very specific queries for a certain project, while at other times they may create reports, or visualizations, that run on a scheduled basis to provide information to business users.

AWS tools for data analysts

Data analysts may use a variety of tools as they work with diverse datasets. This includes using query languages, such as SQL, to explore data in a data warehouse such as Redshift or data in a traditional database. A data analyst may also use advanced toolsets such as **Python** or **R** to perform data manipulation and exploration. Visual transformation tools may also be used by the data analyst to cleanse and prepare data when working with ad hoc data sources that have not been through formal data engineering pipelines.

Data analysts also use BI tools, such as **Amazon QuickSight**, to create advanced visualizations for business users. We covered Amazon QuickSight previously, so let's explore some of the other tools in AWS that can be used by data analysts.

Amazon Athena

Amazon Athena is a service that enables users to run complex SQL queries against a variety of data sources. This can be used to perform ad hoc exploration of data, enabling the data analyst to learn more about the data and test out different queries.

Using Athena, a data analyst can run queries that join data from across tables in different data sources. For example, using Athena, you can run a single query that brings data in from S3 and join that with data from Redshift.

In *Chapter 11, Ad Hoc Queries with Amazon Athena*, we will do a deeper dive into the Athena service.

AWS Glue DataBrew

Data analysts often need to use new sources of data to answer new questions and may need to perform some data transformation on these datasets. While creating these new insights, the data analyst may work closely with business users to develop the reports, visualizations, metrics, or other data as needed. Part of this iterative process may involve creating ad hoc transformation pipelines to ingest, cleanse, join, and transform data.

Once the deliverable has been finalized (data sources identified, transformations determined, and so on), the data analyst may work with their data engineering team to formalize the pipeline. This is a recommended best practice to ensure that all pipelines are contained in a source control system, are part of formal deployment processes, and so on. As such, data engineers should work closely with data analysts, and always be ready to help formalize the ad hoc pipelines that a data analyst may create and that the business has come to depend on.

One of the AWS tools that is very popular with data analysts is the **AWS Glue DataBrew** service. Using DataBrew, data analysts can easily cleanse new data sources and transform and join data from different tables to create new datasets. This can all be done with the Glue DataBrew **visual interface**, without the data analyst needing to write any code:

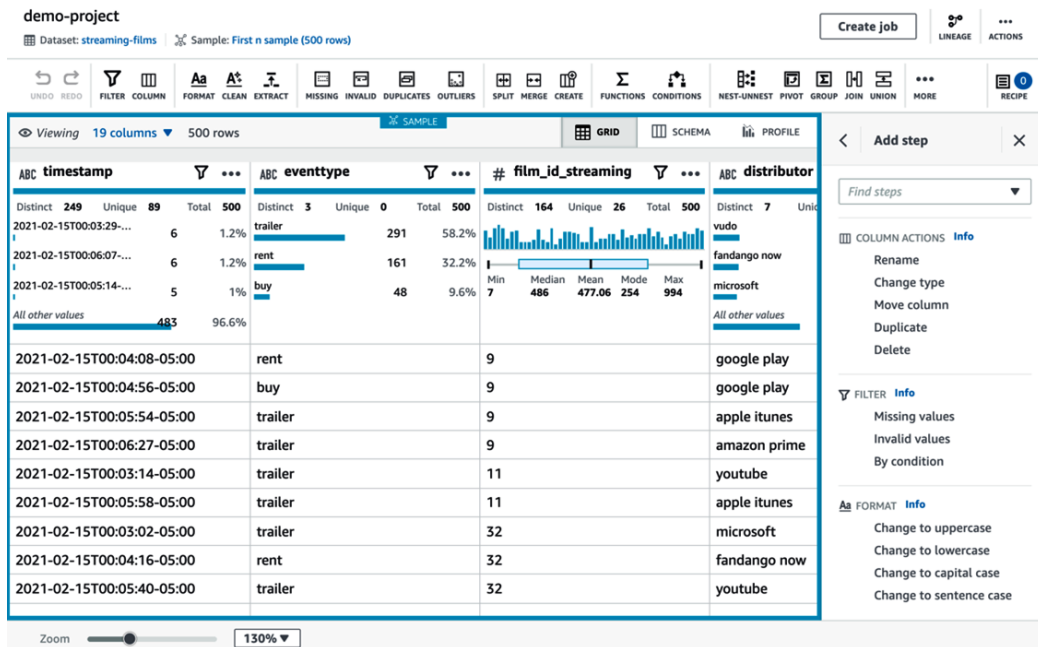


Figure 8.2 – The AWS Glue DataBrew visual transform designer

Glue DataBrew can connect to many different data sources, including Redshift and Snowflake, JDBC databases, S3, Glue/Lake Formation tables, as well as other Amazon services such as AWS Data Exchange and Amazon AppFlow. DataBrew also includes over 250 **built-in transforms** that can be used by data analysts to easily perform common data cleansing tasks and transformations. In the hands-on section of this chapter, you will get to use some of these built-in transforms.

Running Python or R in AWS

Some data analysts have advanced coding skills that they put to use to explore and visualize data using popular programming languages such as Python and R. These languages include many functions for statistically analyzing datasets and creating advanced visualizations.

Python code can be run using multiple services in AWS, including the following:

- **AWS Lambda:** Can run Python code in a serverless environment, for up to a maximum of 15 minutes of runtime
- **AWS Glue Python Shell:** Can run Python code in a serverless environment, with no limit on how long it runs
- **Amazon EC2:** A compute service where you can install Python and run Python code

RStudio, a popular IDE which can be used for creating data analytic projects based on the R programming language, can also be run using multiple services in AWS.

- **RStudio** can be run on Amazon EC2 compute instances, enabling data analysts to create R-based projects for data analysis. See the AWS blog titled *Running R on AWS* (<https://aws.amazon.com/blogs/big-data/running-r-on-aws/>) for more information on how to set this up.
- If you're working with very large datasets, RStudio can also be run on Amazon EMR, which uses multiple compute nodes to process large datasets. See the AWS blog titled *Statistical Analysis with Open-Source R and RStudio on Amazon EMR* (<https://aws.amazon.com/blogs/big-data/statistical-analysis-with-open-source-r-and-rstudio-on-amazon-emr/>) for more information on how to use R with Amazon EMR.

Data engineers can help enable data analysts that have strong Python or R skills by helping them configure these coding environments in AWS. Data engineers can also help formalize data transformation pipelines in those cases where a data analyst has created an ad hoc pipeline for processing, that the business has subsequently come to use on an ongoing basis.

While data analysts are primarily responsible for deriving insights out of data that reflect current trends, as well as the current state of the business, data scientists generally use data to predict future trends and requirements. In the next section, we will dive deeper into the role of the data scientist.

Meeting the needs of data scientists and ML models

Over the past decade, the field of **ML** has significantly expanded, and the majority of larger organizations now have **data science** teams that use ML techniques to help drive the objectives of the organization.

Data scientists use advanced mathematical concepts to develop ML models that can be used in various ways, including the following:

- Identifying non-obvious patterns in data (based on the results of a blood test, what is the likelihood that this patient has a specific type of cancer?)
- Predicting future outcomes based on historical data (is this consumer, with these specific attributes, likely to default on their debt?)
- Extracting metadata from unstructured data (in this image of a person, are they smiling? Are they wearing sunglasses? Do they have a beard?)

Many types of ML approaches require large amounts of raw data to train the **machine learning model** (teaching the model about patterns in data). As such, data scientists can be significant consumers of data in modern organizations.

AWS tools used by data scientists to work with data

Data scientists will use a wide variety of tools with many different purposes, such as tools for developing ML models, tools for fine-tuning those models, and tools for preparing data to train ML models.

Amazon SageMaker is a suite of tools that helps data scientists and developers with the many different steps required to build, train, and deploy ML models. In this section, we will only focus on the tools that are used in data preparation, but in *Chapter 13, Enabling Artificial Intelligence and Machine Learning*, we will do a deeper dive into some of the other AWS tools related to ML and AI.

SageMaker Ground Truth

Most ML models today rely on training the model using labeled data. That is, a dataset that includes the attribute that we are trying to predict is available to help train our model.

Let's use an example of a data scientist named Luna that is looking to create an ML model to identify if an image was of a dog or a cat. To train the model, Luna would need loads of pictures of dogs and cats and would need each image to be labeled to indicate whether it was a picture of a dog or a cat. Once Luna has this information, she could train her ML model to recognize both dogs and cats.

For our example, let's imagine that Luna was able to acquire a set of 10,000 images of dogs and cats, but the images are unlabeled, which means they cannot be used to train the model. And it would take weeks for Luna to go through the 10,000 images on her own to label each one correctly.

Luckily, Luna has heard about **SageMaker Ground Truth**, a fully managed service for labeling datasets. Ground Truth uses its own ML model to automatically label datasets, and when it comes across data that it cannot confidently label, it can route that data to a team of human data labelers to be manually labeled. You can route data to either your pre-selected team of data labelers or make use of the over 500,000 independent contractors that are part of the **Amazon Mechanical Turk** program and have them label the data according to your instructions.

Using Amazon Ground Truth, Luna can quickly and accurately get her 10,000 images of dogs and cats labeled, ready to help train her ML model.

SageMaker Data Wrangler

It has been estimated that data scientists can spend up to 70% of their time cleaning and preparing raw data to be used to train ML models. To simplify and speed up this process, AWS announced **SageMaker Data Wrangler** at their *re:Invent conference* in 2020.

In most organizations, there will be formal datasets that data engineering teams have prepared for consumption by the organization. However, the specific data that a data scientist needs for training a specific model may not be available in this repository, may not be in the required format, or may not contain the granular level of data that is needed. To best enable data scientists to be self-sufficient without needing to depend on other teams, many organizations enable their data science teams to directly ingest and process raw data.

Data Wrangler supports directly ingesting data from sources, including Amazon S3, Athena, Redshift, as well as the Snowflake data warehouse. Once imported, a data scientist can use the SageMaker Studio interface to transform the data, selecting from a library of over 300 built-in data transformations. Data Wrangler also supports writing custom transformations using PySpark and popular Python libraries such as pandas.

Once a Data Wrangler flow has been created in the SageMaker Studio **visual interface**, a user can export the Data Wrangler flow into a **Jupyter Notebook** and run it as a Data Wrangler job, or even export the code as Python code and run it elsewhere.

SageMaker Clarify

SageMaker Clarify is a tool for examining raw data to identify potential bias in data that is going to be used to train ML models. For example, let's say that you were developing a new ML model to detect credit risk for new customers. If your proposed training dataset contains data mostly on middle-aged people, then the resulting ML model may be less accurate when making predictions for younger or older people.

SageMaker Clarify has been integrated with **SageMaker Data Wrangler**, enabling users to evaluate their datasets for potential bias as part of the data preparation process. Users can specify the attributes that they want to evaluate for bias (such as gender or age) and SageMaker Clarify will use several built-in algorithms to detect potential bias. SageMaker Clarify also provides a visual report with details on the measurements and potential bias identified.

So far, we have had a look at several types of data consumers that are common in organizations. Now, we will look at this chapter's hands-on exercise – creating a simple data transformation using AWS Glue DataBrew.

Hands-on – creating data transformations with AWS Glue DataBrew

In *Chapter 7, Transforming Data to Optimize for Analytics*, we used AWS Glue Studio to create a data transformation job that took in multiple sources to create a new table. In this chapter, we discussed how **AWS Glue DataBrew** is a popular service for data analysts, so we'll now make use of Glue DataBrew to transform a dataset.

Differences between AWS Glue Studio and AWS Glue DataBrew

Both AWS Glue Studio and AWS Glue DataBrew provide a visual interface for designing transformations, and in many use cases either tool could be used to achieve the same outcome. However, Glue Studio generates Spark code that can be further refined in a code editor and can be run in any compatible environment. Glue DataBrew does not generate code that can be further refined, and Glue DataBrew jobs can only be run within the Glue DataBrew service. Glue Studio has fewer built-in transforms, and the transforms it does include are generally aimed at data engineers. Glue DataBrew has over 250 built-in transforms, and these are generally aimed at data analysts.

In this hands-on task, we will be playing the role of a data analyst that has been tasked with creating a mailing list that can be used to send marketing material to the customers of our now-closed video store, to make them aware that our catalog of movies is now available for streaming.

Configuring new datasets for AWS Glue DataBrew

To start with, we're going to access the Glue DataBrew console and connect to two existing S3-based data sources (the customer and address tables that we ingested from our MySQL database in *Chapter 6, Ingesting Batch and Streaming Data*):

1. Log into the AWS Management Console and access the Glue service at `https://console.aws.amazon.com/databrew`.
2. From the left-hand side menu, click on **Datasets**.
3. Click on **Connect new dataset**.
4. Provide a **Dataset name** for the customer table (such as `customer-dataset`).
5. In the **Connect to new dataset** section of the window, click on **Data Catalog S3 tables** on the left-hand side. Then, click on `sakila` from the list of Glue databases:

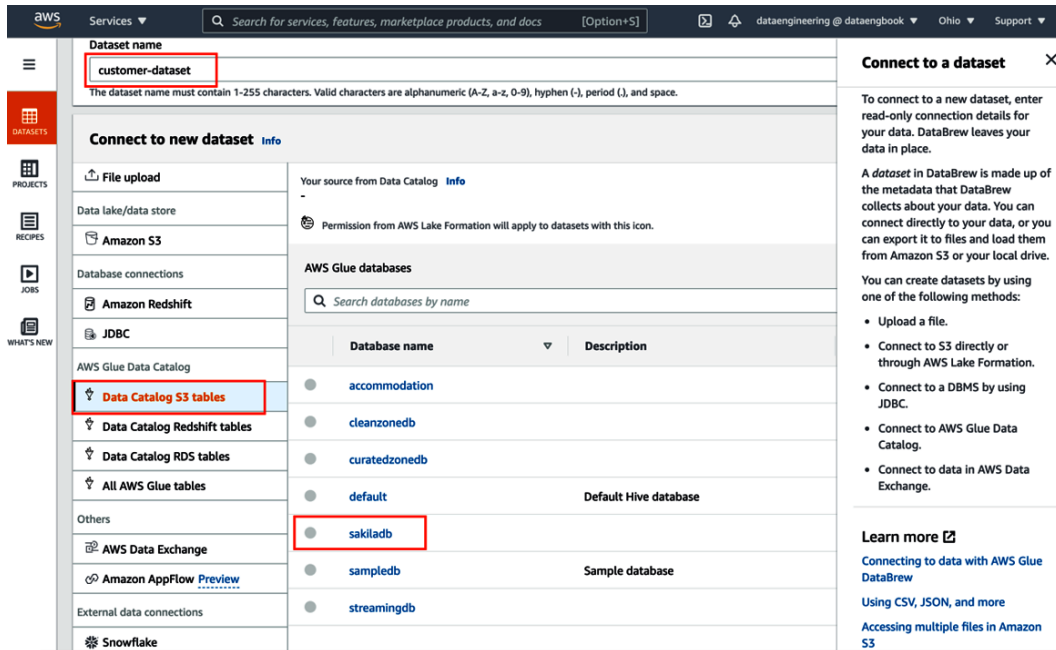


Figure 8.3 – Glue DataBrew – Dataset console

6. From the list of tables, click the selector for the **customer** table, and then click **Create dataset** at the bottom right.
7. Repeat *Steps 1 – 6*, but this time, name the dataset **address-dataset**, select **Data Catalog S3 tables** and **sakila** again, but select the **address** table, and then **Create dataset**.

Now that we have configured the two datasets we plan to use, we will start creating the transform steps in a new DataBrew project.

Creating a new Glue DataBrew project

Now, let's create a new Glue DataBrew project where we can join our customer and address tables, and then clean the dataset:

1. In the AWS Glue DataBrew console, click on **Projects** from the left-hand side menu. Then, click **Create project**.
2. For **Project name**, provide a name (such as `customer-mailing-list`).
3. Under **Recipe details**, leave the default of **Create new recipe** as-is.

4. Under **Select a dataset**, select **customer-dataset**:

DataBrew > Projects > Create project

Create project [Info](#)

Project details

Project name

customer-mailing-list

The project name must contain 1-255 characters. Valid characters are alphanumeric (A-Z, a-z, 0-9), hyphen (-), period (.), and space.

Recipe details [Info](#)

Data cleaning steps in DataBrew are stored as a recipe. A recipe is connected to a project by default. An existing recipe with no associated project could also be applied to a project.

Attached recipe

Create new recipe ▼

Recipe name

customer-mailing-list-recipe

The recipe name must contain 1-255 characters. Valid characters are alphanumeric (A-Z, a-z, 0-9), hyphen (-), period (.), and space.

Import steps from recipe
Import recipe steps from an existing recipe into your project. The existing recipe that you chose will not be edited.

Select a dataset

Select the dataset that you want to work on

My datasets
Your imported datasets

Sample files
Explore example files for your dataset

New dataset
Import new dataset

Find datasets

	Dataset name	Data type	Source	Create date
<input type="radio"/>	address-dataset	Data Catalog table	Data Catalog	31 minutes ago September 21, 2021, 10:57:28 pm
<input checked="" type="radio"/>	customer-dataset	Data Catalog table	Data Catalog	31 minutes ago September 21, 2021, 10:57:11 pm

Figure 8.4 – Creating a new Glue DataBrew project (1)

- Under **Permissions**, from the drop-down list, select **Create new IAM role**.
- For **New IAM role suffix**, provide a suitable suffix, such as dataengbook.
- At the bottom right, click on **Create project**:

The screenshot shows the AWS Glue DataBrew console interface. The top navigation bar includes a hamburger menu, a breadcrumb trail 'customer-dataset > Data Catalog table > Data Catalog', and a timestamp '31 minutes ago September 21, 2021, 10:57:11 pm'. The left sidebar contains navigation icons for DATASETS, PROJECTS (highlighted in orange), RECIPES, JOBS, and WHAT'S NEW. The main content area is titled 'Permissions info' and contains the following sections:

- Sampling - optional**: Select the type and size of your sample.
- Tags - optional**: Metadata that you can define and assign to AWS resources. Each tag is a simple label consisting of a customer-defined key (name) and an optional value. Using tags can make it easier for you to manage, search for, and filter resources by purpose, owner, environment, or other criteria.
- Permissions info**: DataBrew needs permission to connect to data on your behalf. Use an IAM role with the [required policy](#) attached.

The 'Role name' section includes a dropdown menu with 'Create new IAM role' selected and a refresh button. Below it, the 'New IAM role suffix' section shows a text input field containing 'dataengbook'. A note states: 'Your role will be prefixed with "AWSGlueDataBrewServiceRole-"'.

At the bottom, a blue information banner reads: 'As soon as you create a DataBrew project, the project opens and costs begin to accrue to your AWS account. [Pricing details](#)'. Below the banner are 'Cancel' and 'Create project' buttons.

Figure 8.5 – Creating a new Glue DataBrew project (2)

Note that there are session costs associated with Glue DataBrew projects (\$1.00 per 30-minute session). However, at the time of writing, AWS was offering the first 40 sessions at no charge to new Glue DataBrew customers. For the current pricing, see <https://aws.amazon.com/glue/pricing/>.

Building your Glue DataBrew recipe

We can now use the interactive Glue DataBrew project session to build out a recipe for our transformation (a recipe is the steps that are taken to transform our data). Note that it may take a few minutes before the session is provisioned and ready.

In the interactive project session window, as shown in the following screenshot, we can see a sample of our customer table data and a panel to the right that allows us to build our recipe:

The screenshot displays the AWS Glue DataBrew interface. The top bar indicates the project name 'customer-mailing-list'. Below the toolbar, the main data preview shows the following data:

#	customer_id	store_id	first_name
1	1	1	MARY
2	1	1	PATRICIA
3	1	1	LINDA
4	2	2	BARBARA
5	1	1	ELIZABETH
6	2	2	JENNIFER
7	1	1	MARIA
8	2	2	SUSAN
9	2	2	MARGARET
10	1	1	DOROTHY
11	2	2	LISA
12	1	1	NANCY
13	2	2	KAREN

The right-hand panel shows a 'Recipe (0)' section with a 'Build your recipe' button and instructions: 'Start applying transformation steps to your data. All your data preparation steps will be tracked in the recipe.' An 'Add step' button is visible at the bottom of the recipe panel.

Figure 8.6 – AWS Glue DataBrew interactive project session

For our recipe, we want to join this data with our address table, and then make the following changes to the dataset to create a mailing list for our marketing team:

- Change the `first_name` and `last_name` columns to capital case.
- Change the email addresses so that they're all in lowercase.

Follow these steps to create the recipe:

1. Click on **Add step** in the recipe panel on the right-hand side of the console.
2. Scroll down through the list of transformations and select **Join multiple datasets**.
3. From the **Select dataset** dropdown, select **address-dataset**. Dataset metadata, as well as a sample of the dataset, will be displayed. Click on **Next** at the bottom right.
4. For **Select join type**, select **Left join**. This takes all the rows in our left-hand table (the customer table) and joins each row with the matching row in the address table, based on the join keys we specify.
5. For **Join keys**, for **Table A**, select `address_id`. For **Table B**, also select `address_id`.

6. Under **Column list**, deselect all the columns, and then select only the following columns (these will be the only columns that our marketing team needs for the mailing list):
 - A. **Table A**, `customer_id`
 - B. **Table A**, `first_name`
 - C. **Table A**, `last_name`
 - D. **Table A**, `email`
 - E. **Table B**, `address`
 - F. **Table B**, `district`
 - G. **Table B**, `postal_code`

7. Click **Finish**.

We will now see a preview of our new table, with the customer and address tables joined, and only the columns selected previously showing.

You may notice that our customer list includes addresses from many different countries (look at some of the entries under the `district` column), and yet we don't have a column for the country. This is because our original data source (a MySQL database) was highly normalized. The address table has a `city_id` field, and we could have included that and then joined our new dataset with the city table to include the city name and `country_id` fields. However, we would need to have joined that dataset with the country table (joining on the `country_id` column) to get the country name. We will not be covering those steps here, but feel free to give that a try on your own.

All the first names and last names were captured in all uppercase in the original data source (MySQL), so let's transform these into capital case.

8. In the **Recipe** panel, click on **Add step icon** next to **Applied steps**.
9. From the list of transforms, scroll down and select the **FORMAT / Change to capital case** transform.
10. For **Source column**, select the `first_name` column. Ensure that **Format column to** has **Capital case** selected and then click **Apply**.
11. Repeat *Steps 7 – 9*, but this time select the `last_name` column as **Source column**.
12. Repeat *Steps 7 – 9*, but this time select the **FORMAT / Change to lowercase** transform and select the `email` column as the **Source column**.

Your Glue DataBrew recipe should look as follows:

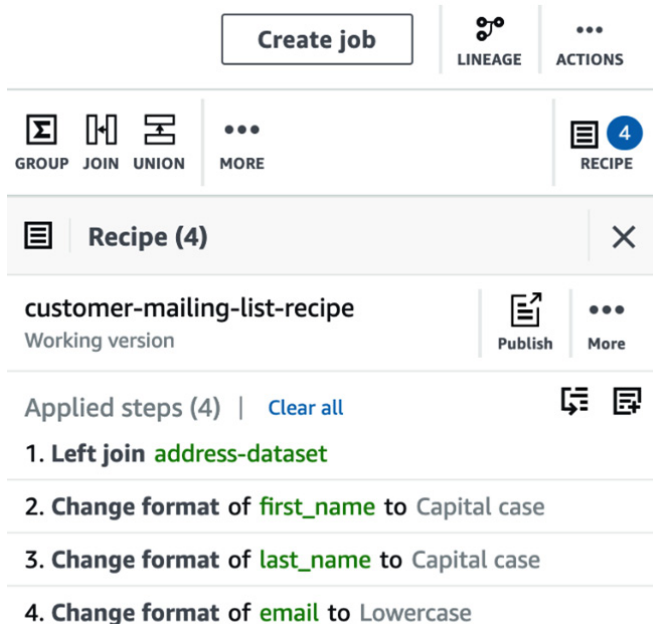


Figure 8.7 – Completed Glue DataBrew recipe

With that, we have created our recipe and been able to preview the results of our transform. Our final step will be to run our recipe in a Glue DataBrew job and write out the results to Amazon S3 so that we can provide the mailing list file to our marketing team.

Creating a Glue DataBrew job

In this final section of our hands-on activity, we will run our recipe in a job and write the results of our transform to a file in Amazon S3:

1. In the AWS Glue DataBrew console, click on **Jobs** from the left-hand side menu. Then, click **Create job**.
2. For **Job name**, provide a name for your job (such as `mailing-list-job`).
3. For **Job input**, select **Project**, and then select your **customer-mailing-list** project.
4. For **Job output settings**, leave the default settings as-is (output to Amazon S3, with CSV set as the file type, the delimiter as a comma, and no compression).
5. For **S3 location**, select a location (such as `s3://dataeng-clean-zone-<initial>/mailing-list`).

6. For **Permissions**, select the role that was created previously in this exercise (such as **AWSGlueDataBrewServiceRole-dataengbook**).
7. For **Permissions**, select **Create new IAM role** and provide a suffix (such as **mailing-list-job**). By having Glue DataBrew create a new role for this job, DataBrew will automatically provide write access to the location you specified for S3 output.
8. Click **Create and run job**.

When the job finishes running, the **Job run history** screen will be displayed, showing the status of the job:

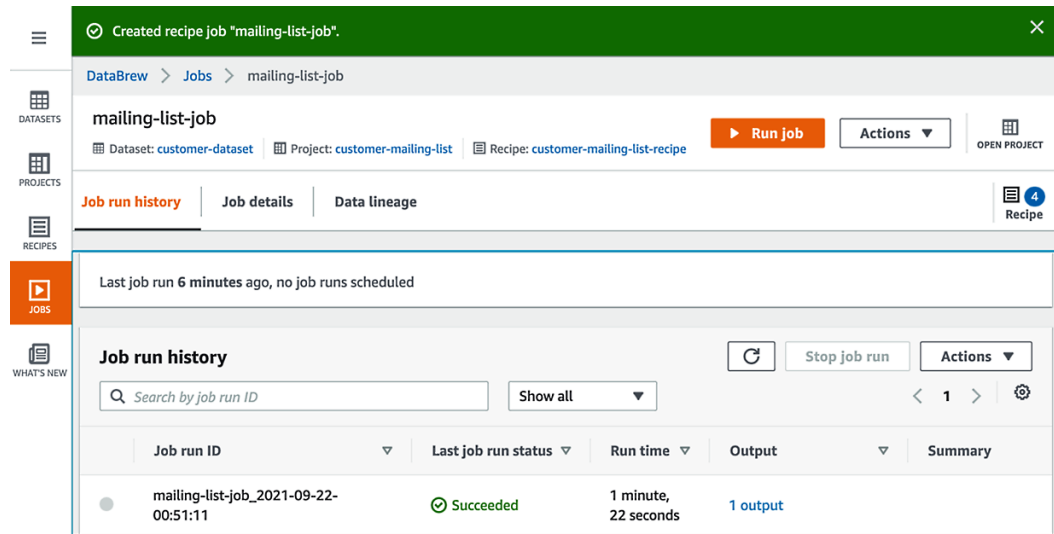


Figure 8.8 – Job run history screen showing the job's status

Click on **1 output** in the **Output** column to view the S3 destination that you selected for this job. Click on **S3 destination path** to open a new browser tab showing the output's location in the S3 console. Download the CSV file and open it with a text editor or spreadsheet application to verify the results.

In this hands-on exercise, you created a new Glue DataBrew job that joined two tables (customer and address). You then ran various transforms on the dataset to format the columns as needed by the marketing team and created a new CSV output file in Amazon S3.

Summary

In this chapter, we explored a variety of data consumers that you are likely to find in most organizations, including business users, data analysts, and data scientists. We briefly examined their roles, and then looked at the types of AWS services that each of them is likely to use to work with data.

In the hands-on section of this chapter, we took on the role of a data analyst, tasked with creating a mailing list for the marketing department. We used data that had been imported from a MySQL database into S3 in a previous chapter, joined two of the tables from that database, and transformed the data in some of the columns. Then, we wrote the newly transformed dataset out to Amazon S3 as a CSV file.

In the next chapter, *Loading Data into a Data Mart*, we will look at how data from a data lake can be loaded into a data warehouse, such as Amazon Redshift.

9

Loading Data into a Data Mart

While the **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 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 data store.

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** 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 the data lake and Redshift
- Hands-on – loading data into an Amazon Redshift cluster and running queries

Technical requirements

For the hands-on exercises in this chapter, you will need permissions to create a new IAM role, as well as permissions 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/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. And while this enables us to query very large datasets that exist on **Amazon S3**, 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.

Not all queries require this kind of high performance, and we can categorize our queries and data into three categories. Let's take a look.

Cold data

This is data that is not frequently accessed, but it is mandatory to store it 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 models).

An example of this is the logs from a banking website. Unless there is a breach, or the customer requests account access history, there is a good chance that after a while, we may not need to query this data again.

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 machine learning project where it would be useful to train the machine learning model with rich, historical data.

In AWS, **cold data** can be stored in the Amazon S3 service, which provides different classes of storage based on your requirements. The following classes of S3 storage are commonly used for cold data, and S3 life cycle rules can be used to move data into these classes automatically after a certain time. For example, you can move certain datasets from warm storage to one of the following cold storage classes:

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

Selecting the appropriate class of S3 storage for your data is important. Storing cold data that is infrequently accessed outside of the Glacier class means you are paying more for that storage than needed, and this is not frugal. Significant savings can be achieved by storing cold data in an appropriate storage class.

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 our SAP or another 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.

In AWS, warm data can also be stored in the Amazon S3 service, but would most likely be stored in the standard storage class. The following classes of S3 storage are commonly used for warm data requirements:

- **Amazon S3 Standard (S3 Standard):** The S3 Standard storage class provides immediate access to data with performance that is ideal for ETL jobs and ad hoc SQL queries with Amazon Athena or **Redshift Spectrum**. With S3 Standard, costs are based on the amount of data stored, and there are no per-GB data retrieval costs (although there is a cost for API GET calls).
- **Amazon S3 Standard-Infrequent Access (S3 Standard-IA):** This storage class offers the same immediate access to data, as well as the same fast retrieval speed, as Amazon S3 Standard. With S3 Standard-IA, the cost per GB for storage is lower than S3 Standard, but there is a per-GB cost for retrieving data. Data in this class can be directly accessed via Glue jobs, Amazon Athena, Redshift Spectrum, and more.
- **Amazon S3 Intelligent-Tiering (S3 Intelligent Tiering):** This storage class is useful when you are unsure of data access patterns. With Intelligent Tiering, data is automatically moved from the Standard tier to the Infrequent Access tier if the data object has not been accessed in 30 days. Optionally, you can enable archive tiering as well, in which case objects that haven't been accessed in 90 days will be moved to S3 Glacier, and after 180 consecutive days without access will be moved to S3 Glacier Deep Archive.

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 also has 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/>.

When you know the access patterns for your data, you should select either the S3 Standard or S3 Standard-Infrequent Access class. However, if you are unsure of data access patterns, you should strongly consider storing the data in the S3 Intelligent Tiering class and allow the Amazon S3 service to automatically move data between classes based on how the data is accessed by your data consumers.

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 business intelligence 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 that 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 **Elasticsearch** (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 business intelligence 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 on 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 massage 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 things that organizations may be tempted to do that are not good for a data warehouse.

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 data store

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, a data warehouse is primarily designed for append-only 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 only support SQL 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 on 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.

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.

Using data warehouses for real-time, record-level use cases

Data warehouses are optimized to load data in batches and are not well-suited to ingesting data as individual records. As such, a data warehouse should not be used as a direct target for large amounts of IoT data (or other real-time data sources), for example.

If there is a requirement to load this kind of data in Redshift, it would be recommended to buffer the data and load the data in batches to Redshift. One way to do this would be by sending the data to **Kinesis Firehose** in real time, where Kinesis Firehose could then buffer the data for up to 15 minutes, or up to 128 MB of data, whichever comes first. Once the buffer is full, Kinesis can instruct Redshift to load the batch of data. However, in most cases, you would still need to design the schema/table structure upfront, whereas with data lakes, you can ingest data directly, without needing any schema design.

Storing unstructured data

While some data warehouses (such as Amazon Redshift) can store semi-structured data (such as JSON data), data warehouses should not 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, 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 addressable storage capacity is listed as 2.56 TB, the actual underlying storage may be closer to 7.5TB).

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 working 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, 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.

Data distribution across slices

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

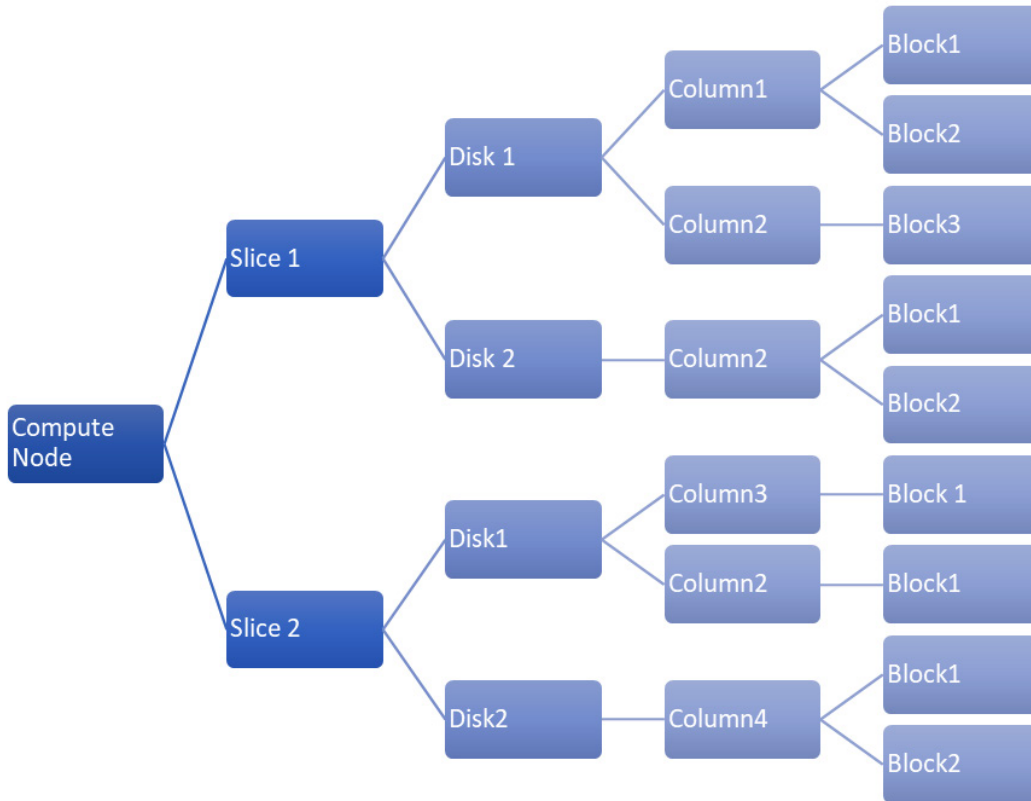


Figure 9.1 – Data distribution across slices on a compute node

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).

The small dimension table can easily fit into the storage of a single node, while the large 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) around 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 regards 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"`. Then, 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.

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 or the size of storage 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.

Selecting the optimal Redshift node type

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:

- **RA3 nodes:** When used with managed storage, you can decouple 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.
- **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).
- **DS2 nodes:** These are legacy nodes that offer compute with attached large hard disk drives. With DS2 nodes, compute and storage is also coupled.

AWS recommends that small data warehouses (under 1 TB in size) use DC2 nodes, while larger data warehouses make use of the RA3 nodes with managed storage. 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 Redshift cluster in the console, you have the option of entering information about your data's size, type of data, and data retention, which 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 analyses 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.

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. 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 take a look.

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:

- `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:

<code>CHAR (8)</code>	
<code>"ABC"</code>	<code>"</code>
<code>"DEF"</code>	<code>"</code>

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

- `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, with the maximum number of characters you need to store.

A column with `VARCHAR (8)`, for example, can store up to 8 single-byte characters, 4 two-byte characters, or 2 four-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 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:

- **SMALLINT/INT2:** These integers have a range of -32,768 to +32,767.
- **INTEGER/INT/INT4:** These integers have a range of -2147483648 to +2147483647.
- **BIGINT/INT8:** These integers have a range of - 9223372036854775808 to +9223372036854775807.

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. **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 -9999.999 to +9999.999.

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 specific 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:

- **REAL/FLOAT4:** These support values of up to 6 digits of precision.
- **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 **datetime** types are supported in Redshift:

- **DATE:** This column type supports storing a date without any associated time. Data should always be inserted enclosed in double quotation marks.
- **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 six-digit precision for fractional seconds.
- **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 missing values will be stored as 00. For example, a `TIMESTAMP` of 2021-05-23 will be stored as 2021-05-23 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` are `{TRUE, 't', 'true', 'y', 'yes', '1'}`. The valid set of specifiers for `False` are `{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 – 0.6%). Using this algorithm means you can now work with extremely 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.

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.

Coupling storage and compute – local Redshift tables

The most common and default table type in Redshift is a table that is permanently stored on the disk local to a compute node and is automatically replicated for fault tolerance purposes.

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.

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 machine learning-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**, 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.

External tables for querying data in Amazon S3

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 on a local disk on the Redshift compute nodes.

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 **Massive Parallel Processing (MPP)** query engine in Redshift to query the data. With Redshift Spectrum, we can still get impressive performance while directly 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 rely on also 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.

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**, **RFile**, and **Sequence** files.

An important point to keep in mind when planning your use of external tables is that Redshift Spectrum charges are based on the amount of data that's scanned by a query, whereas Redshift cluster charges are fixed, based on the cluster node's type and storage. Also, query performance, while impressive, may not match the performance when querying data stored locally in the cluster. Therefore, you should consider loading frequently queried data directly into Redshift local storage, rather than only relying on external tables. This is especially true for datasets that are used for things such as constantly refreshing dashboards, or datasets that are frequently queried by a large group of users.

In the hands-on section of this chapter, we will configure a Redshift Spectrum external table and load data from that table into our Redshift cluster.

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.

If you read in new data and insert that data into a table where there is a matching existing record, this may result in a duplicate record being inserted. As a result, you may end up with multiple versions of the same record, with a number of those records being out of date.

An alternative approach for handling CDC operations in Redshift is to load the new data into a temporary table, and then perform an **INNER JOIN** of the temporary table with the existing table. See Performing a merge operation by replacing existing rows (<https://docs.aws.amazon.com/redshift/latest/dg/merge-replacing-existing-rows.html>) in the Amazon Redshift documentation for more details on how to achieve this.

Data caching using Redshift materialized views

Data warehouses are often used as the backend query engine for business intelligence 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.

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.

However, note that the materialized views are not updated when the underlying data tables are updated, and a `refresh materialized view` Redshift SQL statement needs to be run to refresh the view after full or incremental loads of the underlying tables.

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 for 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 Map Reduce (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 Redshift, 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:

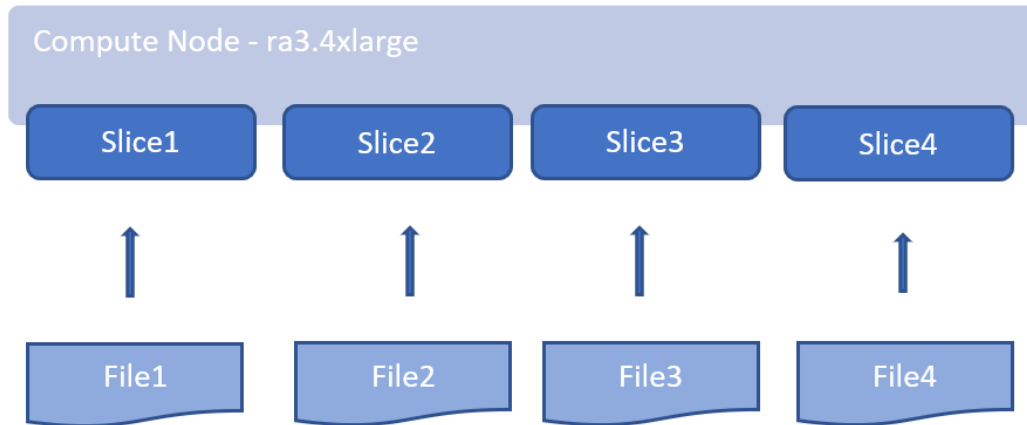


Figure 9.2 – Slices in a Redshift compute node

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 4 slices per ra3.4xlarge node). If your ingest file was 64 GB in size, you should 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 where 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 running on EMR using the Spark-Redshift JDBC driver. 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.

When using AWS Glue, you can configure a Glue connection for your Redshift cluster. This uses built-in drivers within Glue to connect to your Redshift cluster, in a similar way to using the Spark-Redshift JDBC driver in Amazon EMR.

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. 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.

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

Hands-on – loading data into an Amazon Redshift cluster and running queries

In our Redshift hands-on exercise, we're going to create a new Redshift cluster and set up **Redshift Spectrum** so that we can query data in external tables on Amazon S3. We'll then use Redshift Spectrum to read data from S3 and load a subset of that data into a local table in Redshift, after which we'll run some complex queries.

In this exercise, we will be setting up a Redshift cluster for a travel agency. Agents need to ensure that they can find the best deal for accommodation in New York City and Jersey City that is close to specific popular tourist attractions, such as the Freedom Tower and the Empire State Building.

Uploading our sample data to Amazon S3

For this exercise, we will use a dataset from an organization called **Inside Airbnb** (<http://insideairbnb.com/about.html>) that provides Airbnb data under the Creative Commons Attribution 4.0 International License (<https://creativecommons.org/licenses/by/4.0/>) license, which means that the data can be shared and adapted, as long as attribution is given.

For this exercise, we will use the Inside Airbnb data for New York City and Jersey City. Let's get started:

1. Download the **Jersey City** and **New York City** `listings.csv Summary Information and metrics for listings` from <http://insideairbnb.com/get-the-data.html>. Rename each file so that you can identify the Jersey City and New York City listings (for example, `ny-listings.csv` and `jc-listings.csv`).
2. Copy the listing files to the data lake's Landing Zone, creating a partition for each city, as follows. Remember to replace the Landing Zone path with the name of the bucket you created in *Chapter 3, The AWS Data Engineers Toolkit*:

```
aws s3 cp jc-listings.csv s3://dataeng-landing-zone-  
initials/listings/city=jersey_city/jc-listings.csv  
aws s3 cp ny-listings.csv s3://dataeng-landing-zone-  
initials/listings/city=new_york_city/ny-listings.csv
```

3. To verify that the files have been uploaded correctly, we can use **S3 Select** to directly query uploaded files. Open the Amazon S3 console and navigate to the `<bucket>/city=jersey_city/jc-listings.csv` file. Select the file and, from the **Actions** menu, click on **Query with S3 Select**. Leave all the options as their defaults and click **Run SQL query**:

SQL query

Amazon S3 Select supports only the SELECT SQL command. Using the S3 console, you can extract up to 40 MB of records from an object that is up to 128 MB in size. To work with larger files or more records, use the AWS CLI, AWS SDK, or Amazon S3 REST API. For more complex SQL queries, use [Amazon Athena](#).

```

1 /* To create reference point for writing SQL queries, you can display the first 5 records of input data by running the following SQL query: SELECT * FROM s3object s LIMIT
2 SELECT * FROM s3object s LIMIT 5

```

Query results

Query results are not available after you choose **Close** or navigate away. Choose **Download results** to download a copy of the following query results.

Status
✔ Successfully returned 5 records in 358 ms
 Bytes returned: 787 B

Raw

id	name	host_id	host_name	neighbourhood_group	neighbourhood	latitude	long
40669	Skyy's Lounge / Cozy	175412	Skyy		Ward C (councilmember Richard Boggiano)	40.73742	-74.
63282	2bed/2bath,furnished,doorman, by NY	304762	Gil		Ward B (councilmember Mira Prinz-Arey)	40.72813	-74.
146144	Shared Room	266070	Patricia		Ward E (councilmember James Solomon)	40.71077	-74.
215768	Minutes to Manhattan & Jersey Shore	846837	Charlaine		Ward F (councilmember Jermaine D. Robinson)	40.71663	-74.

Figure 9.3 – Running SQL Select against the Airbnb Jersey City listings file

Repeat this again but select `<bucket>/city=new_york_city/ny-listings.csv` instead.

Having uploaded our listings 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

For our Redshift cluster to be able to create EC2 networking resources behind the scenes, our Redshift cluster needs permissions to access specific EC2 networking resources. When we create the first Redshift cluster in our account, Redshift will automatically create an IAM service-linked role called `AWSServiceRoleForRedshift` and attach the managed policy called `AmazonRedshiftServiceLinkedRolePolicy` to the role, providing the required permissions. Therefore, we do not need to create this role manually.

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 directly. 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 select the **Redshift** service from the list of services. For **Select your use case**, select **Redshift – Customizable**. Click on **Next: Permissions**.
3. Attach the following three policies to the role:
 - `AmazonS3FullAccess`
 - `AWSGlueConsoleFullAccess`
 - `AmazonAthenaFullAccess`

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, then 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.*

Then, click on **Next: Tags** and then **Next: Review**.

4. Provide a **Role name**, such as `AmazonRedshiftSpectrumRole`. Make sure that the three policies listed in *Step 3* are included and that **Trusted entities** is set to **AWS service: redshift.amazonaws.com**. Once confirmed, click **Create role**:

Create role



Review

Provide the required information below and review this role before you create it.

Role name*

AmazonRedshiftSpectrumRole

Use alphanumeric and '+,.,@-_' characters. Maximum 64 characters.

Role description

Allows Redshift clusters to call AWS services on your behalf.

Maximum 1000 characters. Use alphanumeric and '+,.,@-_' characters.

Trusted entities

AWS service: redshift.amazonaws.com

Policies

 AmazonS3FullAccess [↗](#)

 AWSGlueConsoleFullAccess [↗](#)

 AmazonAthenaFullAccess [↗](#)

Permissions boundary

Permissions boundary is not set

No tags were added.

* Required

Cancel

Previous

Create role

Figure 9.4 – Creating an IAM Role for Redshift Spectrum

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 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 Redshift trial, enabling you to create and test out a new Redshift cluster for up to 2 months at no charge. However, this is only available if your organization has not previously created a Redshift cluster. If your account is part of an organization that has previously created an Amazon Redshift cluster, you are not eligible for the free trial and your usage of Redshift will be billed for. For a single `dc2.xlarge` node, the cost at the time of writing would be \$182.50 per month. 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 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/> and click on **Create cluster**.
2. You can change, or keep, the default Redshift cluster name (`redshift-cluster-1`), but make sure to select **Free trial** for **What are you planning to use this cluster for?**.
3. Leave the default **Admin user name** (`awsuser`) as-is, but provide an **Admin user password**. Make sure you can recall this password later as it will be needed in future steps. Click on **Create cluster**.
4. Wait until your cluster is listed with a **Status** of **Available**, and then click on the cluster's name:

The screenshot shows the Amazon Redshift console interface. At the top, there are navigation tabs for 'In my account' and 'From other accounts'. The main section is titled 'Connect to Redshift clusters' and is divided into three columns: 'Query data using Redshift query editor', 'Work with your client tools', and 'Choose your JDBC or ODBC driver'. Below this, there is a 'Clusters (1)' section with a search bar and a table of clusters. The table has columns for Cluster, Cluster namespace, Status, Storage capacity, CPU utilization, and Snapshots. A single cluster is listed: 'redshift-cluster-1' with namespace 'dc2.large | 1 node | 160 GB', ID 'fbbd8441-8434-4347-...', status 'Available', and no snapshots.

Figure 9.5 – Created Redshift cluster

5. Click on the **Properties** tab, scroll down to the **Cluster permissions** section, and click the **Attach IAM roles** button.
6. Select the role you created previously for Redshift Spectrum (such as `AmazonRedshiftSpectrumRole`) and click on the **Associate IAM role** button. Then, click on **Save changes**.

Note that it may take a few minutes for the permissions modification to be applied. Click on Clusters on the left-hand side menu, and wait until Status changes from Available - Modifying, to just Available. Once the change has been fully applied, you can continue and test the Redshift connection. .

7. On the left-hand side of the Redshift console, click on **Editor**, and then click on **Connect to database**.
8. Leave the default of **Create a new connection** as-is and set **Authentication** to **Temporary credentials**. Make sure your cluster is selected from the drop-down list, and then enter `dev` for **Database name** and `awsuser` for **Database user**. Then, click **Connect**.
9. Once connected, ensure `dev` is set for **Select database** and `public` is set for **Select schema**. Then, in the query editor, run `select * from sales limit 10` and click **Run**.

The preceding query should have returned 10 rows from the sample database that was loaded when we created our trial Redshift cluster.

Now that we have created and verified our Redshift cluster, we can create the external tables that will enable us to query data in S3.

Creating external tables for querying data in S3

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 catalog), and we specify that we want to create a new database in the Glue catalog called **accommodations**. For Redshift to be able to write to the Glue data catalog and to access objects in S3, we need to specify the ARN for the Redshift Spectrum role that we previously created:

1. Run the following command in the Redshift query editor to create a new external schema called `spectrum_schema`, and to also create a new database in the Glue catalog called **accommodations**. 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 'accommodation'
iam_role 'arn:aws:iam::1234567890:role/
AmazonRedshiftSpectrumRole'
create external database if not exists;
```

Note that because we made a connection to the **dev** database in Redshift previously, the external schema is created as an object in the **dev** database.

2. We can now define an **external table** that will be registered in our Glue data catalog under our **accommodations** database. When defining the table, we specify the columns that exist, the column that we have partitioned our data by (city), the format of the files (text, comma delimited), and the location in S3 where the text files were uploaded. Make sure to replace the bucket name of the S3 location with the name of the bucket you created:

```
CREATE EXTERNAL TABLE spectrum_schema.listings(  
  listing_id INTEGER,  
  name VARCHAR(100),  
  host_id INT,  
  host_name VARCHAR(100),  
  neighbourhood_group VARCHAR(100),  
  neighbourhood VARCHAR(100),  
  latitude Decimal(8,6),  
  longitudes Decimal(9,6),  
  room_type VARCHAR(100),  
  price SMALLINT,  
  minimum_nights SMALLINT,  
  number_of_reviews SMALLINT,  
  last_review DATE,  
  reviews_per_month NUMERIC(8,2),  
  calculated_host_listings_count SMALLINT,  
  availability_365 SMALLINT)  
partitioned by(city varchar(100))  
row format delimited  
fields terminated by ','  
stored as textfile  
location 's3://dataeng-landing-zone-initials/listings/';
```


- Verify that the table was created correctly by selecting `spectrum_schema` from the dropdown on the left-hand side and by expanding the **listings** table to view the defined columns:

The screenshot shows the Amazon Redshift console interface. On the left, the 'Resources' sidebar is visible, with the 'Select schema' dropdown set to 'spectrum_schema' and the 'listings' table expanded to show its columns. The main area displays the SQL query for creating the external table:

```

1 CREATE EXTERNAL TABLE spectrum_schema.listings(
2 listing_id INTEGER,
3 name VARCHAR(100),
4 host_id INT,
5 host_name VARCHAR(100),
6 neighbourhood_group VARCHAR(100),
7 neighbourhood VARCHAR(100),
8 latitude Decimal(8,6),
9 longitudes Decimal(9,6),
10 room_type VARCHAR(100),
11 price SMALLINT,
12 minimum_nights SMALLINT,
13 number_of_reviews SMALLINT,
14 last_review DATE,
15 reviews_per_month NUMERIC(8,2),

```

Below the query, there are buttons for 'Run', 'Save', 'Schedule', and 'Clear'. The 'Table details' tab is active, showing a message: 'No data selected. To view details, choose data from navigator.'

Figure 9.6 – External table in the Redshift console

- We now need to add the specific partitions that we created, which we can do by running the following two commands in the Redshift query editor. Make sure to update the location so that it references your bucket name:

```
alter table spectrum_schema.listings add
partition(city='jersey_city')
```

```
location 's3://dataeng-landing-zone-initials/listings/
city=jersey_city/'
```

```
alter table spectrum_schema.listings add
partition(city='new_york_city')
```

```
location 's3:// dataeng-landing-zone-initials /listings/
city=new_york_city/'
```

5. Verify that the table and partitions have been created correctly by viewing them in the AWS Glue console. Open **AWS Glue console** in a new browser window and click on **Databases** via the left-hand side navigation menu.
6. Click on the Glue database we created earlier called **accommodation**, and then click on **Tables in accommodation**.
7. Click on the **listings** table, which will list the columns as we define them:

The screenshot shows the AWS Glue console interface for the 'listings' table. The left sidebar contains navigation options like 'Data catalog', 'Databases', 'Tables', 'Connections', 'Crawlers', 'Schema registries', 'Settings', 'ETL', 'AWS Glue Studio', 'Blueprints', 'Workflows', 'Jobs', 'ML Transforms', 'Triggers', 'Dev endpoints', 'Notebooks', 'Security', 'Security configurations', 'Tutorials', and 'Add crawler'. The main content area shows the table details for 'listings' in the 'accommodation' database. The 'Table properties' are displayed as a row of buttons: EXTERNAL, TRUE, transient_lastDdlTime, and 1626466031. Below this is a 'Schema' table with 5 columns.

	Column name	Data type	Partition key	Comment
1	listing_id	int		
2	name	varchar(100)		
3	host_id	int		
4	host_name	varchar(100)		
5	neighbourhoo...	varchar(100)		

Figure 9.7 – Viewing the listings table in the AWS Glue console

- Click on the **View Partitions** button to view the partitions that have been defined:

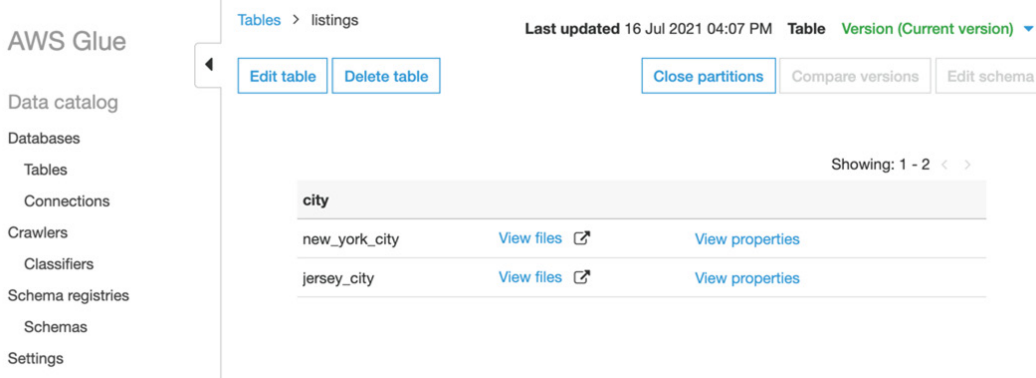


Figure 9.8 – Viewing table partitions in the AWS Glue console

Note that instead of defining the table manually in the Redshift console and adding the partitions, we could have used a Glue Crawler to crawl the S3 location and automatically add the table and partitions to the Glue data catalog. If we had done that, we would have still needed to create the Redshift external schema and define the database, but we would not have had to specify the column details for the table.

- To confirm that everything has been set up correctly, we can query the data using both Redshift Spectrum and Amazon Athena. In the Redshift query editor, run the following query:

```
select * from spectrum_schema.listings limit 100;
```

Note that when querying the table in Redshift, we query based on `<redshift_external_schema_name>.<table_name>`.

- Open the Amazon Athena console in a new browser window and run the following query:

```
select * from accommodation.listings limit 100;
```

Note that when querying the table with Athena, we query based on `<glue_database_name>.<table_name>`.

Now that we have configured Redshift Spectrum to be able to read the files we uploaded to Amazon S3, we can design a Redshift table to store just the data we need locally.

Creating a schema for a local Redshift table

With Redshift Spectrum, we pay for each query we run, based on the amount of data scanned. If we have a hot dataset that is going to be queried regularly, we may want to move required data into a local Redshift table so that we are not charged for every query. Depending on the types of queries we run, we may also find that the performance of queries against data stored locally is better than querying data on Amazon S3 via Redshift Spectrum.

If we wanted to load the full dataset from Amazon S3 into Redshift, we could use the Redshift COPY command to read the data from S3 and load it into a local table. However, we only want to query a subset of the data in our Amazon S3 files, so in this exercise, we will use Redshift Spectrum to read in just the required data and write it out to a local Redshift table. Let's get started:

1. First, we must create a new Redshift schema to store our local tables. Run the following command in the Redshift query editor:

```
create schema if not exists accommodation_local;
```

2. We can now create a new local table that contains just the fields that we require for our use case. Run the following in the Redshift query editor to create the new table:

```
CREATE TABLE dev.accommodation_local.listings(  
  listing_id INTEGER,  
  name VARCHAR(100),  
  neighbourhood_group VARCHAR(100),  
  neighbourhood VARCHAR(100),  
  latitude Decimal(8,6),  
  longitudes Decimal(9,6),  
  room_type VARCHAR(100),  
  price SMALLINT,  
  minimum_nights SMALLINT,  
  city VARCHAR(40))  
  distkey(listing_id)  
  sortkey(price);
```

With the preceding command, we have created a new local table in our **dev** database, and in `accommodation_local` schema, called **listings**. We defined the various columns, specified that we want the table distributed across the compute nodes of our cluster based on `listing_id`, and then specified that we want the table sorted on the **price** column.

3. To load data from our external Spectrum table into our new local table, we can run the following query:

```
INSERT into accommodation_local.listings
(SELECT listing_id,
 name,
 neighbourhood_group,
 neighbourhood,
 latitude,
 longitudes,
 room_type,
 price,
 minimum_nights
FROM spectrum_schema.listings);
```

This query inserts data into our new local table, based on a query of the data in our external Spectrum schema. We just select the columns that we need for our planned queries.

Running complex SQL queries against our data

We can now run some advanced queries against the local listings table we just loaded data into. Our goal here is to be able to easily identify Airbnb listings in the New York City and Jersey City areas that are close to specific tourist attractions. We will split our query into different parts to explain what each part is doing:

1. The first part of our query, which you can paste into the Redshift query editor, is as follows:

```
WITH touristspots_raw(name,lon,lat) AS (
(SELECT 'Freedom Tower', -74.013382,40.712742) UNION
(SELECT 'Empire State Building', -73.985428, 40.748817)),
touristspots (name,location) AS (SELECT name,
ST_Point(lon, lat) FROM touristspots_raw)
select name, location from touristspots
```

This part of the query creates a new temporary table called `touristspots_raw`, and inserts the names and longitude and latitude of two popular New York City tourist attractions. It then uses a Redshift function called `ST_Point` to convert the longitude and latitude of the tourist attractions into **point geometry** (which can be used in distance calculations). This portion of the query results in a new virtual table called `touristspots` that has two fields – name and location.

- Now, we want to convert the latitude and longitude of the values in our accommodation table into point geometry. We can do this with the following query, which you can run in the Redshift query editor (note that each block below should be on a single line in the Redshift query editor, so if copying and pasting be careful of inserted line breaks)

```
WITH accommodation(listing_id, name, room_type,
location) AS (SELECT listing_id, name, room_type, ST_
Point(longitudes, latitude) from accommodation_local.
listings)
select listing_id, name, room_type, location from
accommodation
```

This query creates a new temporary table called `accommodation` with data from our listings table, but again, it uses the `ST_Point` function to convert longitude and latitude into point geometry, as a field called `location`.

- Now, we can combine the preceding two queries in a modified form and add the final part of our query. This query will calculate the distance between a listing from our listings table containing Airbnb data, and either the Freedom Tower or Empire State Building. Then, we can sort the result by distance and return the 100 closest listings. Run the following query in the Redshift query editor:

```
WITH touristspots_raw(name,lon,lat) AS (
(SELECT 'Freedom Tower', -74.013382,40.712742) UNION
(SELECT 'Empire State Building', -73.985428, 40.748817)
),
touristspots(name,location) AS (
SELECT name, ST_Point(lon, lat)
FROM touristspots_raw),
accommodation(listing_id, name, room_type, price,
location) AS
(
SELECT listing_id, name, room_type, price,
ST_Point(longitudes, latitude)
```

```

FROM accommodation_local.listings)
SELECT
  touristspot.name as tourist_spot,
  accommodation.listing_id as listing_id,
  accommodation.name as location_name,
  (ST_DistanceSphere(touristspot.location,
  accommodation.location) / 1000)::decimal(10,2) AS
  distance_in_km,
  accommodation.price AS price,
  accommodation.room_type as room_type
FROM touristspot, accommodation
WHERE tourist_spot like 'Empire%'
ORDER BY distance_in_km
LIMIT 100;

```

In this final query, we combined our previous queries (to create two temporary tables – `touristspot` and `accommodation`) and we added new statements. We used the Redshift `ST_DistanceSphere` function to calculate the distance between a tourist spot and one of our listings, and then we converted the result into a decimal data type with two decimal places and named that column `distance_in_km`.

We then used a `WHERE` clause to filter out results to just the Empire State Building, sorted (or ordered) the result by distance, and limited the query to just the first 100 results.

- As our agents will be regularly running these queries to find the right Airbnb listing for our customers, we can create a materialized view that contains all of our listings, along with the distance between the listing and both the Empire State Building and the Freedom Tower. This will save us from having to calculate the distance each time the query is run. Run the following in the Redshift query editor to create the materialized view:

```

CREATE MATERIALIZED VIEW listings_touristspot_distance_
view AS
WITH touristspot_raw(name,lon,lat) AS (
  (SELECT 'Freedom Tower', -74.013382,40.712742) UNION
  (SELECT 'Empire State Building', -73.985428, 40.748817)
),
touristspot(name,location) AS (

```

```
SELECT name, ST_Point(lon, lat)
FROM touristspots_raw),
accommodation(listing_id, name, room_type, price,
location) AS
(
SELECT listing_id, name, room_type, price,
ST_Point(longitudes, latitude)
FROM accommodation_local.listings)
SELECT
touristspots.name as tourist_spot,
accommodation.listing_id as listing_id,
accommodation.name as location_name,
(ST_DistanceSphere(touristspots.location,
accommodation.location) / 1000)::decimal(10,2) AS
distance_in_km,
accommodation.price AS price,
accommodation.room_type as room_type
FROM touristspots, accommodation
```

In a system where there are a lot of queries with complex calculations, creating materialized views can help manage the CPU and memory pressure on the system.

5. Now, we can query the view we have created by running the following in the Redshift query editor:

```
select * from listings_touristspot_distance_view where
tourist_spot like 'Empire%' order by distance_in_km limit
100
```

This query returns the top 100 listings that are closest to the Empire State Building. You do not need to calculate the distance for each point as part of the query.

In these hands-on exercises, we created a Redshift cluster, ingested data, and ran several queries. Feel free to experiment with other queries, such as loading in the latitude and longitude for other tourist spots, and finding a query that finds listings for a certain room type or within a specific price range.

Summary

In this chapter, we learned how a cloud data warehouse can be used to store hot data to optimize performance and manage costs. 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 an optimized schema in Redshift, before reviewing ingested unloaded from Redshift.

Then, we performed a hands-on exercise where we created a new Redshift cluster, configured Redshift Spectrum to query data from Amazon S3, and then loaded a subset of data from S3 into Redshift. We then ran some complex queries to calculate the distance between two points before creating a materialized view with the results of our complex query.

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

10

Orchestrating the Data Pipeline

Throughout this book, we have been discussing various services that can be used by data engineers to ingest and transform data, as well as make it available for consumers. We looked at how we could ingest data via Amazon Kinesis Data Firehose and Amazon Database Migration Service, and how we could run AWS Lambda and AWS Glue functions to transform our data. We also discussed the importance of updating a data catalog as new datasets are added to a data lake, and how we can load subsets of data into a data mart for specific use cases.

For the hands-on exercises, we made use of various services, but for the most part, we triggered these services manually. However, in a real production environment, it would not be acceptable to have to manually trigger these tasks, so we need a way to automate various data engineering tasks. This is where data pipeline orchestration tools come in.

Modern-day ETL applications are designed with a modular architecture to facilitate the use of the best purpose-built tool to complete a specific task. A data engineering pipeline (also sometimes referred to as a workflow) stitches all of these components together to create an ordered execution of related tasks.

To build our pipeline, we need an orchestration engine to define and manage the sequence of tasks, as well as the dependencies between tasks. The orchestration engine also needs to be intelligent enough to perform different actions based on the failure or success of a task and should be able to define and execute tasks that run in parallel, as well as tasks that run sequentially.

In this chapter, we will look at how to manage data pipelines with different orchestration engines. First, we will examine some of the core concepts of pipeline orchestration and then review several different options within AWS for orchestrating data pipelines. In the hands-on activity for this chapter, we will orchestrate a data pipeline using the AWS Step Function service.

In this chapter, we will cover the following topics:

- Understanding the core concepts for pipeline orchestration
- Examining the options for orchestrating pipelines in AWS
- Hands-on – orchestrating a data pipeline using AWS Step Function

Technical requirements

To complete the hands-on exercises in this chapter, you will need an AWS account where you have access to a user with administrator privileges (as covered in *Chapter 1, An Introduction to Data Engineering*). We will make use of various AWS services, including **AWS Lambda**, **AWS Step Function**, and **Amazon Simple Notification Service (SNS)**.

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/tree/main/Chapter10>

Understanding the core concepts for pipeline orchestration

In *Chapter 5, Architecting Data Engineering Pipelines*, we architected a high-level overview of a data pipeline. We examined potential data sources, discussed the types of data transformations that may be required, and looked at how we could make transformed data available to our data consumers.

Then, we examined the topics of data ingestion, transformation, and how to load transformed data into data marts in more detail in the subsequent chapters. As we discussed previously, these steps are often referred to as an **extract, transform, load (ETL)** process.

We have now come to the part where we need to combine the individual steps involved in our ETL processes to operationalize and automate how we process data. But before we look deeper at the AWS services for enabling this, let's examine some of the key concepts around pipeline orchestration.

What is a data pipeline, and how do you orchestrate it?

A simple definition is that a **data pipeline** is a collection of data processing tasks that need to be run in a specific order. Some tasks may need to run sequentially, while other tasks may be able to run in parallel. You could also refer to the sequencing of these tasks as a Workflow.

Data pipeline orchestration refers to automating the execution of tasks involved in a data pipeline Workflow, managing dependencies between the different tasks, and ensuring that the pipeline runs when it is meant to.

Think of the data pipeline as the smallest entity for performing a specific task against a dataset. For example, if you are receiving data from a partner regularly, your first data pipeline may involve validating that the data that's received is valid, and then converting the data file into an optimized format, such as **Parquet**. If you have hundreds of partners sending you data files, then this same pipeline may run for each of those partners.

You may also have a second data pipeline that runs at a specific time of day that validates that the data from all your partners has been received, and then runs a Spark job to join the datasets and enrich the data with additional proprietary data.

Once that data pipeline finishes running, you may have a third pipeline that loads the newly enriched data into a data warehouse.

While you could place all of these steps in a single data pipeline, it is a recommended best practice to split pipelines into the smallest logic grouping of steps. In our example, our first step is getting newly received files converted into Parquet format, but we only want to do that if we can confirm that the file that's been received is valid. As such, we group those two tasks into a single pipeline. The goal of our second pipeline is to join the files we have received and enrich the new file with additional data, but we must also include a step to validate and report on whether all the expected partner files were received.

What is a directed acyclic graph?

When talking about data pipelines, you may hear the term **directed acyclic graph**, commonly referred to as **DAG**. If you Google this term, you may find a lot of complex mathematical explanations of what a DAG is. This is because this term does not only apply to data pipelines, but is used to define many different types of ordered processes. For example, DAGs are also used to design compilers.

A simple explanation of a DAG is that it represents connections between nodes, with the flow between nodes always occurring in only one direction and never looping back to an earlier node (acyclic means not a cycle).

The following diagram shows a simple DAG:

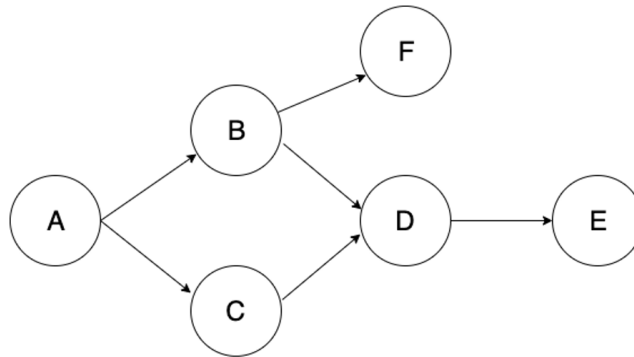


Figure 10.1 – A simple example of a directed acyclic graph

If this DAG represented a data pipeline, then the following would take place:

- When event A completes, it triggers event B and event C.
- When event B completes, it triggers event F.
- When events B and C are complete, they trigger event D.
- When event D completes, it triggers event E.

In the preceding example, event F could never loop back to event A, B, or C as that would break the acyclic part of the DAG definition.

No rule says that data pipelines have to be defined as DAGs, although certain orchestration tools do require this. For example, Apache Airflow (which we will discuss in more detail later in this chapter) requires pipelines to be defined as a DAG, and if there is a cycle in a pipeline definition where a node loops back to a previous node, this would not run. However, AWS Step Function does allow for loop cycles in the definition of a state machine, so Step Function-based pipelines do not enforce that the pipeline should be a DAG.

How do you trigger a data pipeline to run?

There are two primary types of triggers for a pipeline – **schedule-based pipelines** and **event-based pipelines**.

Traditionally, pipelines were all triggered on a schedule. This could be once a day, or every hour, or perhaps even every 15 minutes. This is still a common approach, especially for batch-orientated pipelines. In our example pipeline, the second pipeline could be an example of a scheduled pipeline that runs once per day to join and enrich partner files that are received throughout the day.

Today, however, a lot of pipelines are created to be event-driven. In other words, the pipeline is triggered in response to some specific event being completed. Event-based Workflows are useful for reducing the latency between data becoming available and the pipeline processing that data. For example, if you expect that you will have received the data files you need at some point between 4 A.M. and 6 A.M., you could schedule the pipeline to run at 6 A.M. However, if all the data is available by 5 A.M. on some days, using an event-based trigger can get your pipeline running earlier.

In our earlier example of a pipeline, the first pipeline would be an event-driven pipeline that runs in response to a partner having uploaded a new file. Within AWS, there is strong support for creating event-driven activities, such as triggering an event (which could be a pipeline) based on a file being written to a specific Amazon S3 bucket.

Using manifest files as pipeline triggers

A **manifest** is often used to refer to a list of cargo carried by ship, or other transport vehicles. The manifest document may be reviewed by agents at a border crossing or port to validate what is being transported.

In the world of data pipelines, a common concept is to create a **manifest file** that contains information about other files that form part of a batch of files.

In our data pipeline example of receiving files from our partners, we may find that the partner sends hundreds of small CSV files in a batch every hour. We may decide that we do not want to run our pipeline on each file that we receive, but that we want to process all the small CSV files of a batch together and convert them into a single Parquet file.

In this case, we could instruct our partners to send a manifest file at the end of each batch of files that they send to us. This manifest file would list the name of each file that's transferred, as well as potentially some validation data, such as file size, or a calculated SHA-256 hash of the file.

We could then configure our S3 event notification to only trigger when a file that begins with the name `manifest` is written to our bucket. When this happens, we will trigger our pipeline to run, and perhaps the first step in our pipeline would be to read the manifest file, and then for each file listed in the manifest, verify that it exists. We could also calculate an SHA-256 hash of the file, and verify that it matches what is listed in the manifest. Once the files have been verified, we could run our ETL job to read in all the files and write the files out in Parquet format.

This process would still be considered an event-driven pipeline, even though we are not responding to every file upload event, just the completion of a batch of uploads, as represented in the manifest file.

How do you handle the failures of a step in your pipeline?

As part of the orchestration process to automate processing of steps in a pipeline, we need to ensure that failures are handled correctly. As part of this, it is also important that log files related to each step of the pipeline are easily accessible. In this section, we will look at some important concepts for failure handling and logging.

Common reasons for failure in data pipelines

There are many reasons why a specific step in a data pipeline may fail. Some common reasons for errors include the following:

- **Data quality issues:** If one of the steps in your pipeline is expecting to receive CSV files to process, but instead receives a file in JSON format that it does not know how to process, this would lead to a hard failure (that is, a failure that your job cannot recover from until the data quality issue is resolved).
- **Code errors:** When you update a job, it is possible to introduce a syntax, or logic, error into the code. Testing your code before deploying it into production is very important, but there may be times when your testing does not catch a specific error. This would also be a hard failure, requiring you to redeploy fixed code.
- **Endpoint errors:** One of the steps in your pipeline may need to either read or write data to or from a specific endpoint (such as reading a file in S3 or writing data into a data warehouse). Sometimes, these errors may be due to a temporary problem, such as a temporary network error, and this could be considered a soft failure (that is, one that may be overcome by retrying). At other times, the error may be a hard failure, such as your job not being configured with the correct permissions to access the endpoint.

- **Dependency errors:** Data pipelines generally consist of multiple steps with complex dependencies. This includes dependencies within the pipeline, as well as dependencies between different pipelines. If your job is dependent on a previous step, then the job it is dependent on is referred to as an upstream job. If your job fails, any jobs that depend on it are considered downstream jobs. Dependency errors can be hard failures (such as an upstream job or pipeline having a hard failure) or soft failures (the upstream job is taking longer than expected to complete, but if you retry your step, it may complete later).

Hard failures generally interrupt processing (and also likely cause failures in downstream jobs) until someone takes a specific action to resolve the error. Soft failures (such as intermittent networking issues), however, can benefit from having a good retry strategy, as we will discuss next.

Pipeline failure retry strategies

When you're designing your pipeline, you should consider implementing a retry strategy for failed steps. Many orchestration tools (such as Apache Airflow and AWS Step Function) will allow you to specify the number of retries, the interval between retry attempts, as well as a backoff rate.

The **retry backoff rate** (also known as **exponential backoff**) causes the time between retry attempts to be increased on each retry. With AWS Step Function, for example, you can specify a `BackOffRate` value that will multiply the delay between retries by that value. For example, if you specify a retry interval of 10 seconds and a backoff rate of 1.5, Step Function will wait 15 seconds (10 seconds x 1.5) for the second retry, 22.5 seconds (15 seconds x 1.5) for the third retry, and so on.

Having reviewed some of the core concepts of data pipelines and orchestration, we can now examine the tools that are available in AWS for creating and orchestrating pipelines.

Examining the options for orchestrating pipelines in AWS

As you will have noticed throughout this book, AWS offers many different building blocks for architecting solutions. When it comes to pipeline orchestration, AWS provides native serverless orchestration engines with AWS Data Pipeline and AWS Step Function, a managed open source project with **Amazon Managed Workflows for Apache Airflow (MWAA)**, and service-specific orchestration with AWS Glue Workflows.

There are pros and cons to using each of these solutions, depending on your use case. And when you're making a decision, there are multiple factors to consider, such as the level of management effort, the ease of integration with your target ETL engine, logging, error handling mechanisms, and cost and platform independence.

In this section, we'll examine each of the four pipeline orchestration options.

AWS Data Pipeline for managing ETL between data sources

AWS Data Pipeline is one of the oldest services that AWS has for creating and orchestrating data pipelines, having been originally released in 2012.

Using AWS Data Pipeline, you can extract, transform, and load data between certain AWS data sources – even on-premises data sources. To use this service, you define your data sources, schedule transform activities, and select a data target for writing to. Data Pipeline will then manage the scheduling of the pipeline, automatically provision the required AWS resources (such as an EMR cluster), and enable you to monitor pipelines with configurable retry logic and alerting.

The following AWS data services are supported as sources and targets by Data Pipeline:

- Amazon DynamoDB
- Amazon Relational Database System
- Amazon Redshift
- Amazon S3

In addition to these data sources, Data Pipeline is also able to read and write to other JDBC data stores, such as an on-premises database.

The following compute services can be used to run jobs to transform your data:

- Amazon EC2
- Amazon EMR
- On-premises compute resources (by installing the Java-based Data Pipeline task runner software)

If you review the AWS documentation for the Data Pipeline service, you may notice that there have not been many recent updates to the service. For example, the last update to the documentation was in 2018 (as per <https://docs.aws.amazon.com/datapipeline/latest/DeveloperGuide/DocHistory.html>), the default EC2 instance in most regions is the m1 instance family (although newer generations, such as m5 instances, can be used), and the task runner software is only supported by Java 1.6 and Java 1.8 versions. Also, the Data Pipeline service is only supported in five AWS regions (Northern Virginia, Oregon, Sydney, Tokyo, and Ireland).

Because of these limitations, it is generally recommended to use the newer AWS services for building and orchestrating data pipelines.

AWS Glue Workflows to orchestrate Glue resources

In *Chapter 3, The AWS Data Engineers Toolkit*, we introduced the **AWS Glue Workflows** service. As a reminder, this is a part of the AWS Glue service and can be used to build a data pipeline consisting of Glue components (Glue Crawlers and Glue Spark or Python jobs).

For use cases where you are creating a data pipeline that only uses AWS Glue components, the use of Glue Workflows can be a good fit. For example, you could create the following pipeline using Glue Workflows:

- Run a Glue Crawler to add CSV files that have been ingested into a new partition to the Glue Data Catalog.
- Run a Glue Spark job to read the new data using the catalog, and then transform the CSV files into Parquet files.
- Run another Glue Crawler to add the newly transformed Parquet files to the Glue Data Catalog.
- Run two Glue jobs in parallel. One Glue job aggregates data and writes the results into a DynamoDB table. The other Glue job creates a newly enriched dataset that joins the new data to an existing reference set of data.
- Run another Glue Crawler to add the newly enriched dataset to the Glue Catalog.
- Run a Glue Python Shell job to send a notification about the success or failure of the job.

While a fairly complex data pipeline can be created using Glue Workflows (as shown here), many use cases require the use of other AWS services, such as EMR for running Hive jobs or writing files to an SQS queue. While Glue Workflows don't support integration with non-Glue services directly, it is possible to run a Glue Python Shell job that uses the Boto3 library to interact with other AWS services. However, this is not as feature-rich or as obvious to monitor as interacting with those services directly.

Monitoring and error handling

Glue Workflows includes a graphical UI that can be used to monitor job progress. With the UI, you can see whether any step in the pipeline has failed, and you can also resume the Workflow from a specific step once you have resolved the issue that caused the error. While Glue Workflows does not include a retry mechanism as part of the Workflow definition, you can specify the number of retries in the properties of individual Glue jobs.

CloudWatch Events provides a real-time stream of change events that can be generated by some AWS services, including AWS Glue. While, at the time of writing, Glue does not generate any events from Glue Workflows, events are generated from individual Glue jobs. For example, there is a *Glue Job State Change* event that is generated for Glue jobs that reflects one of the following states: SUCCEEDED, FAILED, TIMEOUT, or STOPPED.

Using Amazon EventBridge, you can automate actions to take place when a new event and status you are interested in is generated. For example, you can create an EventBridge rule that picks up Glue Job FAILED events, then triggers a Lambda function to run, and sends an email notification with details of the failure.

Triggering Glue Workflows

When you create a Glue Workflow, you can select the mechanism that will cause the Workflow to run. There are three ways that a Glue Workflow run can be started.

If set to **on-demand**, the Workflow will only run when it's started manually from the console, or when it's started using the Glue API or CLI.

If set to **scheduled**, you can specify a frequency for running the job, such as hourly, daily, monthly, or for specific days of the week (such as Mondays to Fridays). Alternatively, you can set a custom schedule using a *cron* expression, which uses a string to set a frequency to run. For example, if you set the cron expression to `* / 30 8 - 16 * * 2 - 6`, the Workflow will run *every 30 minutes between 8 A.M and 4:59 P.M., Mondays to Fridays*.

Glue Workflows also support an **event-driven approach**, where the Workflow is triggered in response to an EventBridge event. With this approach, you can configure an Amazon EventBridge rule to send events to Glue Workflows, such as an S3 PutObject event for a specific S3 bucket and prefix.

When configuring your Workflow, you can also specify triggering criteria where you specify that you only want the Workflow to run after a certain number of events are received, optionally specifying a maximum amount of time to wait for those events.

For example, if you have a business partner that sends many small CSV files throughout the day, you may not want to process each file individually, but rather process a batch of files. For this use case, you can configure the Workflow to trigger once 100 events have been received and specify a time delay of 3,600 seconds (1 hour).

This time delay starts when the first unprocessed event is received. If the specified number of events is not received within the time delay you entered, the Workflow will start anyway and process the events that have been received.

If you receive 100 events between 8 A.M. and 8:40 A.M., the first run of the Workflow will be triggered at 8:40 A.M. If you receive only 75 events between 8:41 A.M. and 9:41 A.M., the Workflow will run a second time at 9:41 A.M. anyway and process the 75 received events since the time delay of 1 hour has been reached.

Functionality such as the ability to easily restart a Workflow from a specific step, as well as the ability to batch events before triggering the running of a Workflow, makes Glue Workflows a good data pipeline orchestration solution for pipelines that only use the Glue service. However, if you are looking for a more comprehensive solution that can also orchestrate other AWS services and on-premises tools, then you should consider AWS Step Function or Apache Airflow, which we will discuss next.

Apache Airflow as an open source orchestration solution

Apache Airflow is a piece of open sourced orchestration software, originally developed at Airbnb, that provides functionality for authoring, monitoring, and scheduling Workflows. Some of the features available in Airflow include stateful scheduling, a rich user interface, core functionality for logging, monitoring, and alerting, and a code-based approach to authoring pipelines.

Within AWS, a managed version of Airflow is available as a service called **Amazon Managed Workflows for Apache Airflow (MWAA)**. This service simplifies the process of getting started with Airflow, as well as the ongoing maintenance of Airflow infrastructure since the underlying infrastructure is managed by AWS. Like other AWS managed services, AWS ensures the scalability, availability, and security of the Airflow software and infrastructure. Please refer to the overview of Amazon MWAA in *Chapter 3, The Data Engineers Toolkit*, for more information on the architecture of this managed service.

When deploying the managed MWAA service in AWS, you can choose from multiple supported versions of Apache Airflow. At the time of writing, Airflow v1.10.12 and Airflow v2.0.2 are supported in the managed service.

Core concepts for creating Apache Airflow pipelines

Apache Airflow uses a code-based (Python) approach to authoring pipelines. This means that to work with Airflow, you do need some Python programming skills. However, having pipelines as code is a natural fit for saving pipeline resources in a source control system, and it also helps with creating automated tests for pipelines.

The following are some of the core concepts that are used to create Airflow pipelines.

Directed acyclic graphs (DAGs)

We introduced the concept of a **directed acyclic graph (DAG)** earlier in this chapter. In the context of Airflow, a data pipeline is created as a DAG (using Python to define the DAG), and the DAG provides the tasks in the pipeline and the dependencies between tasks.

In the Airflow user interface, you can also view a pictorial representation of the DAG – the pipeline tasks and their dependencies, with tasks represented as nodes and arrows showing the dependencies between tasks.

Airflow Connections and Hooks

Airflow Hooks define how to connect to remote source and target systems, such as a database, or a system such as Zendesk. This hook contains the code that controls the connection to the remote system, and while Airflow includes several built-in Hooks, it also lets you define custom hooks. Built-in hooks include hooks for Amazon S3, HTTP systems, various databases (such as Oracle, MySQL, and Postgres), as well as systems such as Slack, Presto, and Hive.

Open source contributors can also create and share hooks, and this includes hooks for AWS services such as Athena, DynamoDB, Firehose, and Glue, as well as for non-AWS services such as Google BigQuery, DataBricks, Jenkins, and many others.

A related concept is **Airflow Connections**, which defines the URL/hostname, username, and password that is used to make a connection to a remote system.

Hooks and Connections contain the code to connect to and authenticate with remote systems, keeping that code separate from pipeline definitions.

Airflow Tasks

Airflow Tasks defines the basic unit of work that a DAG performs. Each task is defined in a DAG with upstream and downstream dependencies, which defines the order in which the tasks should run.

When a DAG runs, the tasks in the DAG move through various states, from `None` to `Scheduled`, to `Queued`, to `Running`, and finally to `Success` or `Failed`.

Airflow Operators

Airflow Operators provide predefined task templates that provide a pre-built interface for performing a specific task. Airflow includes several built-in core operators (such as `BashOperator` and `PythonOperator`, which execute a bash command or Python function). There is also an extensive collection of additional operators that are released separately from Airflow Core (such as `JdbcOperator`, `S3FileTransformOperator`, `S3toRedshiftTransfer`, and `DockerOperator`).

Airflow Sensors

Airflow Sensors provides a special type of Airflow operator that is designed to wait until a specific action takes place. The sensor will regularly check whether the activity it is waiting on has been completed, and can be configured to time out after a certain period.

Using Airflow Sensors enables you to create event-driven pipelines. For example, you could use `S3KeySensor`, which waits for a specific key to be present at an S3 path and, once present, triggers a specific DAG to run.

Pros and cons of using MWAA

One of the key differentiators for Airflow is active development support from the open source community, with over 1,500 contributors. As a result of this active community, Airflow supports a wide range of integrations with many different services, including services from AWS, Google, and Microsoft Azure cloud. If your pipelines need integration with lots of services from multiple providers, then the number of supported integrations in Airflow is one of the most significant benefits you will find from using Airflow.

Airflow is also a mature service, with built-in functionality for retrying tasks, alerting on failures, and scaling to handle large and complex Workflows. It has a well-developed UI for monitoring and managing pipelines. Airflow is widely used and proven across many large enterprises, such as Airbnb.

The managed version of Airflow that is available from AWS significantly simplifies the time and effort for deploying an Airflow environment. AWS also provides built-in functionality for scaling Airflow workers, automatically adding or removing additional workers based on demand.

However, you do need to have some Python skills to use Airflow, so the learning curve for using Airflow may be higher than when using an orchestration tool that provides a graphical user interface for creating pipelines. Airflow also has a certain amount of fixed infrastructure that is used for delivering the service, and this comes with an associated fixed cost. So, whether your Airflow environment is actively running a pipeline, or whether it is idle for hours between pipeline runs, there is an ongoing cost for the environment.

Now, let's look at the final option within AWS for orchestrating data pipelines: the AWS Step Function service.

AWS Step Function for a serverless orchestration solution

AWS Step Function is a comprehensive serverless orchestration service that uses a low-code approach to develop data pipelines and serverless applications. Step Function provides a powerful visual design tool that allows you to create pipelines with a simple drag and drop approach. Or, if you prefer, you can define your pipeline using **Amazon States Language (ASL)** directly using JSON.

AWS has built optimized, easy-to-use integrations between many different AWS services and Step Function. For example, you can easily add a step that runs a Lambda function and select the name of the Lambda function to run from a drop-down list.

Step Function also makes it easy to specify how to handle the failure of a state with custom retry policies, lets you specify catch blocks to catch specific errors, and takes custom actions based on the error.

For services where AWS has not built an optimized integration, you can still run the service by using the AWS SDK integration built into Step Function. For example, there is no direct Step Function integration for running Glue Crawlers, but you can add a state that calls the `Glue StartCrawler` API and specify the parameters that are needed by that API call.

Step Function also includes strong support for error handling and has a visual interface for monitoring the status of a Step Function state machine run. However, Step Function does not currently support the ability to restart a state machine from a specific step.

A sample Step Function state machine

With Step Function, you create a **state machine** that defines the various tasks that make up your data pipeline. Each task is considered a state within the state machine, and you can also have states that control the flow of your pipeline, such as a **choice state** that executes a branch of the pipeline, or a **wait state** to pause the pipeline for a certain period.

When you're executing a Step Function state machine, you can pass in a payload that can be accessed by each state. Each state can also add additional data to the payload, such as a status code indicating whether a task succeeded or failed.

The following diagram shows a sample state machine in Step Function:

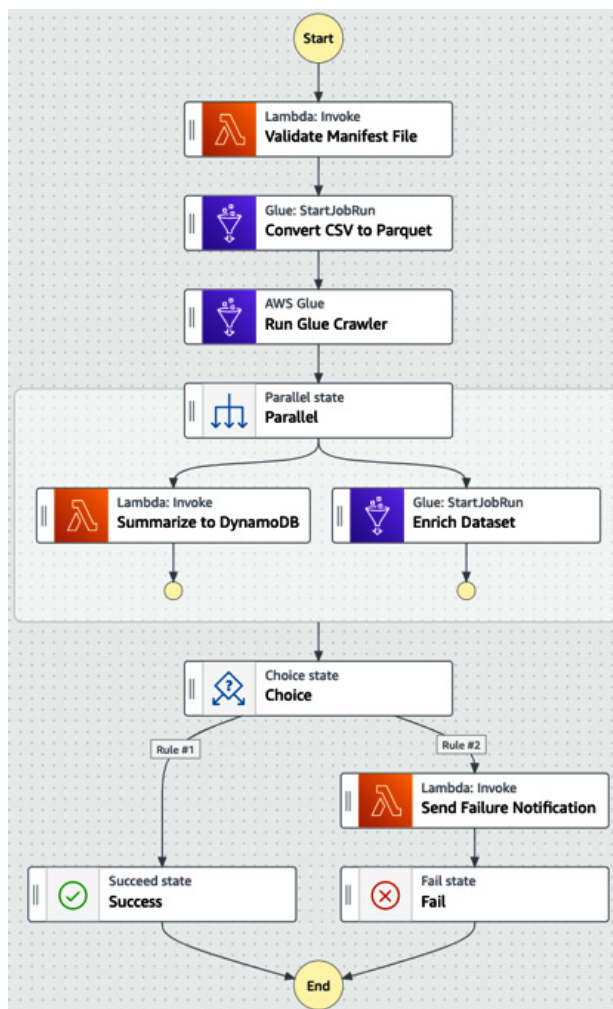


Figure 10.2 – Sample Step Function state machine

In this state machine definition, we can see the following states:

1. We start with a **Task state** that executes a Lambda function that validates a manifest file that has been received (ensuring all the files listed in the manifest exist, for example).
2. We then have another **Task state**, this time to execute a Glue Job that will convert the files we received from CSV format into Parquet format.
3. Our next step is another **Task state**. This time, the task executes a Glue crawler to update our data catalog with the new Parquet dataset we have generated.
4. We then enter a **Parallel state**, which is used to create parallel branches of execution in our state machine. In this case, we're executing a Lambda function (to summarize data from the Parquet file and store the results in a DynamoDB table) and then trigger a Glue job to enrich our new Parquet dataset with additional data.
5. We then enter a **Choice state**. The choice state specifies rules that get evaluated to determine what to do next. In this case, if our Lambda and Glue jobs succeeded, we end the state machine with a **Success state**. If either of them failed, we run a Lambda function to send a failure notification, and we end the state machine with a **Fail state**.

The visual editor that can be used in the console to create a state machine ultimately ends up generating an **Amazon States Language (ASL)** JSON file that contains the definition of the pipeline. You can store the JSON definition file for your data pipeline in a source control system, and then use the JSON file in a CI/CD pipeline to deploy your Step Function state machine.

Pros and cons of using AWS Step Function

AWS Step Function provides a native AWS solution for defining and orchestrating pipelines that are easy to use with a powerful visual design tool. Step Function is offered as a serverless service, which means that you only pay for the service while actively using it; you do not have any infrastructure to manage or even any infrastructure decisions to make.

AWS offers Step Function as a highly available service within a region, and even provides an SLA indicating that AWS will make *commercially reasonable efforts to make AWS Step Function available with a monthly uptime percentage for each AWS region, during any monthly billing cycle, of at least 99.9%*. For more information on this SLA, see <https://aws.amazon.com/step-functions/sla/>.

However, Step Function does not natively let you resume a pipeline from its point of failure, which tools such as Apache Airflow do offer. Also, while Step Function is very well integrated with AWS services and lets you orchestrate even on-premises workloads, if you are looking for an orchestration tool with strong integration to non-AWS third-party services, then Apache Airflow has the strongest offering for that.

In the hands-on exercises for this chapter, you will get the opportunity to build out a data pipeline using AWS Step Function. However, before we do that, let's summarize your choices for data pipeline orchestration within AWS.

Deciding on which data pipeline orchestration tool to use

As we have discussed in this chapter, there are multiple options for creating and orchestrating data pipelines within AWS. And while we have looked at four different options offered by AWS directly, there are countless other options from AWS partners that could also be considered.

For less complex environments that only use the services supported by either AWS Data Pipeline or AWS Glue Workflows, these services can be a good choice. However, for larger and more complex environments, it is worth examining both Amazon MWAA and AWS Step Function.

The following tables show a comparison of Step Function and Amazon MWAA based on several different key attributes:

Criteria	AWS Step Functions	Amazon Managed Workflows for Apache Airflow (MWAA)
Short description	Serverless AWS native orchestration service	Managed AWS service for open source Apache Airflow
Graphical pipeline development	Yes	No
Graphical run visualization	Yes	Yes
Error and retry single step	Yes	Yes
Re-run from failed step	Custom workaround	Yes
Open source community support	No	Yes
Cost	Usage-based cost that depends on the complexity of the workflow	Constant base infrastructure cost, plus worker costs that can scale up and down
Scalability	Highly scalable, fully automatic	Highly scalable, managed by user or autoscaling groups, and can be configured
Infrastructure management	No infrastructure management or provisioning as everything handled by AWS	Requires making choices about infrastructure, but AWS manages the infrastructure and software
Language for pipeline development	JSON (or use of visual designer)	Python
Serverless/managed	Serverless	Managed
Integration	Seamlessly integrates with AWS services and manual integration with non-AWS services	Strong integration support for many AWS services, as well as extensive third-party services

Figure 10.3 – Comparison of AWS Step Function and Amazon MWAA

Now, let's get hands-on with AWS Step Function and see how this service can let us visually build pipelines.

Hands-on – orchestrating a data pipeline using AWS Step Function

In this section, we will get hands-on with the AWS Step Function service, which can be used to orchestrate data pipelines. The pipeline we're going to orchestrate is relatively simple, but Step Function can also be used to orchestrate far more complex pipelines with many steps. To keep things simple, we will only use Lambda functions to process our data, but you could replace Lambda functions with Glue jobs in production pipelines that need to process large amounts of data.

For our Step Function state machine, we are going to start by using a Lambda function that checks the extension of an incoming file to determine the type of file. Once determined, we'll pass that information on to the next state, which is a CHOICE state. If it is a file type we support, we'll call a Lambda function to process the file, but if it's not, we'll send out a notification, indicating that we cannot process the file.

If the Lambda function fails, we'll send a notification to report on the failure; otherwise, we will end the state machine with a SUCCESS status.

Creating new Lambda functions

Before we can create our Step Function, we need to create the Lambda functions that we will be orchestrating. We will create three separate Lambda functions in this section.

Using a Lambda function to determine the file extension

Our first Lambda function will check the extension of any file that's uploaded to an Amazon S3 bucket. Then, it will return it that to the state machine. Let's get started:

1. Log in to **AWS Management Console** and navigate to the **AWS Lambda** service at <https://console.aws.amazon.com/lambda/home>.
2. Ensure that you are in the region that you have been using for all the exercises in this book.
3. Click on **Create function**.
4. Select **Author from scratch**. Then, for **Function name**, enter `dataeng-check-file-ext`.
5. For **Runtime**, select **Python 3.9**. Leave the defaults for **Architecture** and **Permissions** as-is and click **Create function**.

6. In the **Code source** block, replace any existing code with the following code. This code receives an EventBridge event when a new S3 file is uploaded and uses the metadata included within the event to determine the extension of the file:

```
import urllib.parse
import json
import os
print('Loading function')

def lambda_handler(event, context):
    print("Received event: " + json.dumps(event,
    indent=2))

    # Get the object from the event and show its content
    type

    bucket = event['detail']['requestParameters']
    ['bucketName']

    key = urllib.parse.unquote_plus(event['detail']
    ['requestParameters']['key'], encoding='utf-8')
    filename, file_extension = os.path.splitext(key)
    print(f'File extension is: {file_extension}')
    payload = {
        "file_extension": file_extension,
        "bucket": bucket,
        "key": key
    }
    return payload
```

7. Click the **Deploy** button above the code block section to save and deploy our Lambda function.

Now, we can create a second Lambda function that will process the file we received. However, for this exercise, the code in this Lambda function will randomly generate failures.

Lambda to randomly generate failures

For this Lambda function, we will use a random number generator to determine whether to cause an error in the Lambda function or to succeed. We will do this by generating a random number that will be either 0, 1, or 2 and then dividing our random number by 10. When the random number is 0, we will get a "divide by zero" error from our function. Let's get started:

1. Repeat *Steps 1 to 5* of the previous section to create the first Lambda function, but this time, for **Function name**, enter `dataeng-random-failure-generator`.
2. In the **Code source** block, replace any existing code with the following code:

```
from random import randint
def lambda_handler(event, context):
    print('Processing')
    #Our ETL code to process the file would go here
    value = randint(0, 2)
    # We now divide 10 by our random number.
    # If the random number is 0, our function will fail
    newval = 10 / value
    print(f'New Value is: {newval}')
    return(newval)
```

3. Click the **Deploy** button above the code block section.

We now have two Lambda functions that we can orchestrate in our Step Function state machine. But before we create the state machine, we have a few additional resources to create.

Creating an SNS topic and subscribing to an email address

If there is a failure in our state machine, we want to be able to send an email notification about the failure. We can use the SNS service to send an email. To do this, we need to create an SNS topic that we will send the notification to. Then, we can subscribe one or more email addresses to that topic. Let's get started:

1. Navigate to the **Amazon SNS** service at <https://console.aws.amazon.com/sns>.
2. Ensure that you are in the region that you have been using for all the exercises in this book.

3. In the menu on the left-hand side, click on **Topics**, then **Create topic**.
4. For **Type**, select **Standard**.
5. For **Name**, enter `dataeng-failure-notification`.
6. Leave all the other items as-is and click on **Create topic**.
7. In the **Topic details** section, click **Create subscription**.
8. For **Protocol**, select **Email**.
9. For **Endpoint**, enter your email address. Then, click on **Create subscription**.
10. Access your email and look for an email from `no-reply@sns.amazonaws.com`. Click the **Confirm subscription** link in that email. You need to do this to receive future email notifications from Amazon SNS.

We now have an SNS topic with a confirmed email subscription that can receive SNS notifications.

Creating a new Step Function state machine

Now, we can orchestrate the various components that we have created so far (our two Lambda functions and the SNS topic we will use for sending messages):

1. Navigate to the **Amazon Step Function** service at `https://console.aws.amazon.com/states/home`.
2. Ensure that you are in the region that you have been using for all the exercises in this book.
3. Click on **Create state machine**.
4. Leave the default of **Design your Workflow visually** as-is and set the type to **Standard**. Then, click **Next**.
5. This will show a visual editor with a **Start** block and an **End** block. Drag the **AWS Lambda Invoke** block into the visual designer, between the **Start** and **End** blocks.
6. On the right-hand side of the screen, set **State name** to `Check File Extension`.
7. Under **API Parameters**, use the drop-down list to select the Lambda function that extracts the file extension (such as `dataeng-check-file-ext`).
8. Click on the **Output** tab, click the selector for **Filter output with OutputPath**, and provide a value of `$.Payload`. Selecting this option configures our **Check File Extension** state to have an output of whatever was returned by our Lambda function (in our case, we have configured our Lambda function to return some JSON that contains the S3 bucket, object, and file extension of the file to process).

The screenshot shows the AWS Step Functions console interface for designing a workflow. The main area displays a state machine diagram with a 'Start' state, a 'Lambda: Invoke Check File Extension' state, and an 'End' state. The right-hand side shows the configuration for the 'Lambda: Invoke' task, including a 'Check File Extension' label and a 'Lambda: Invoke' task result example. The 'Output' tab is selected, showing a JSON example of a task result and configuration options for 'Transform result with ResultSelector', 'Add original input to output using ResultPath', and 'Filter output with OutputPath'. The 'Filter output with OutputPath' option is checked, and the output path is set to '\$.Payload'.

Figure 10.4 – Building out a Step Function state machine

9. On the left-hand side, click on the **Flow** tab. Then, drag the **Choice** state to between the **Lambda Invoke** function and the **End** state. We use the **Choice** state to branch out our pipeline to run different processes, based on the output of a previous state. In this case, our pipeline will do different things depending on the extension of the file we are processing.
10. On the right-hand side, under **Configuration** for our new choice state, click the **Pencil Edit** icon next to **Rule #1** and then click **Add conditions**.
11. On the pop-up screen, under **Variable**, enter `$.file_extension` (our Lambda function returned some JSON, including a JSON path of `file_extension` that contains a string with the extension of the file we are processing). Set **Operator** to **matches string** and for **value**, enter `.csv`. Then, click **Save conditions**.
12. On the left-hand side, switch back to the **Actions** tab and drag the **AWS Lambda Invoke** state to the **Rule #1** box in the flow diagram.
13. For our new **Lambda Invoke** state for **Rule #1**, set **State name** to `Process CSV` (since our **Choice** function is going to invoke this Lambda for any file that has an extension of `.csv`, as we set in *Step 11*).

- Under **API Parameters**, use the dropdown to set **Function name** to our second Lambda function (`dataeng-random-failure-generator`). In a real pipeline, we would have a Lambda function (or Glue job) that would read the CSV file that was provided as input and process the file. In a real pipeline, we may have also added additional rules to our **Choice** state for other file types (such as XLS or JPG) and had different Lambda functions or Glue jobs invoked to handle each file type.

However, in this exercise, we are only focusing on how to orchestrate pipelines, so our Lambda function code is designed to simply divide 10 by a random number, resulting in random failures when the random number is 0.

- On the left-hand side, switch back to the **Flow** tab and drag the **Pass** state to the **Default** rule box leading from our **Choice** state. The default rule is used if the output of our Lambda function does not match any of the other rules. In this case, our only other rule is for handling files with a `.csv` extension, so if a file has any other extension besides `.csv`, the default rule will be used.
- On the right-hand side, for the **Pass** state configuration, change **State name** to `Pass - Invalid File Ext`. Then, click on the **Output** tab and paste the following into the **Result** textbox:

```
{
  "Error": "InvalidFileFormat"
}
```

The **Pass** state is used in a state machine to modify the data that is passed to the next state. In this case, we want to pass an error message about the file format being invalid to the next state in our pipeline.

Ensure that the selector for **Add original input to output using ResultPath** is selected and that the dropdown is set to **Combine original input with result**. In the textbox, enter `$.Payload`.

- If we receive an **InvalidFileFormat** error, we want to send a notification using the Amazon SNS service. To do so, on the left-hand side, under the **Actions** tab, drag the **Amazon SNS Publish** state to below our **Pass - Invalid File Ext** state.

On the right-hand side, on the **Configuration** tab for the **SNS Publish** state, under **API Parameters**, set **Topic** to our previously created SNS topic (`dataeng-failure-notification`). Your state machine should now look as follows:

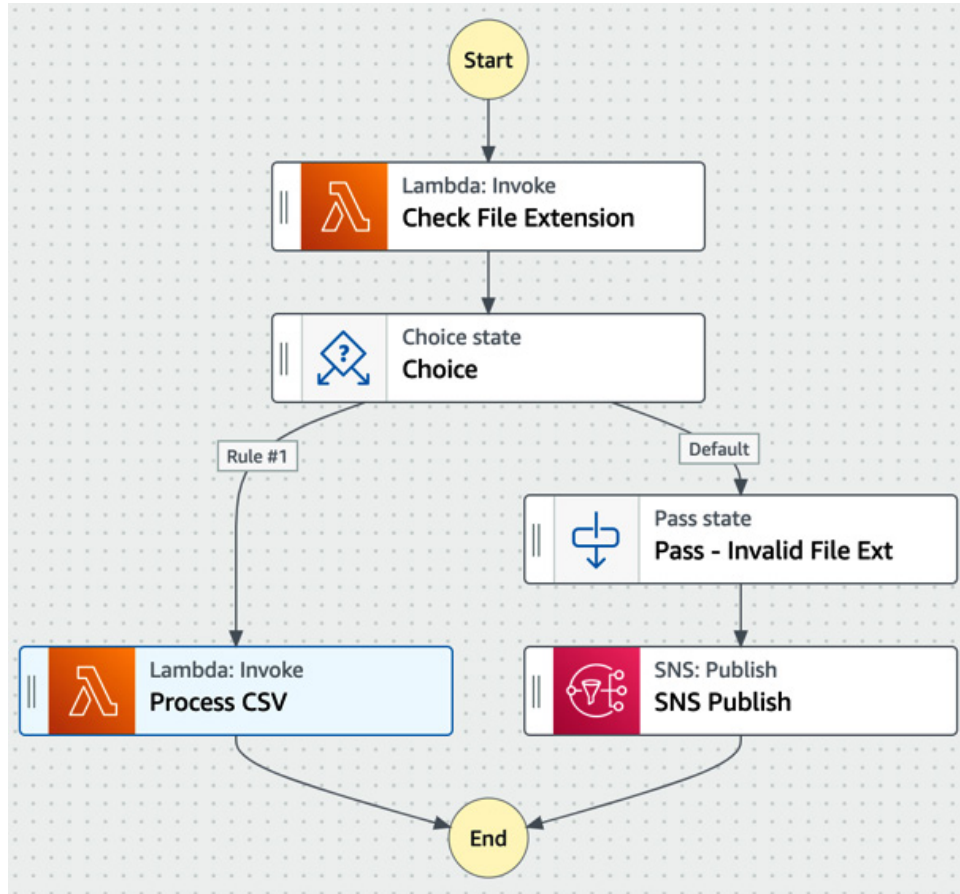


Figure 10.5 – The current status of our Step Function state machine

18. We can now add error handling for our **Process CSV** state. Click on the **Process CSV** state and, on the right-hand side, click on the **Error handling** tab. Under **Catch errors**, click on the + **Add new catcher** button. For **Errors**, select **States.ALL**, for **Fallback state**, select our **SNS Publish** state, and for **result path**, enter `$.Payload`. This configuration means that if our Lambda function fails for any reason (**States.ALL**), we will add the error message to our JSON under a **Payload** and pass this to our SNS notification state.

19. On the left-hand side, click on the **Flow** tab and drag **Success state** under the **Process CSV** state. Then, drag **Fail state** under the **SNS Publish** state. We are doing this as we want our Step Function to show as having failed if, for any reason, something failed and we ended up sending a failure notification using SNS. Your finalized state should look as follows:

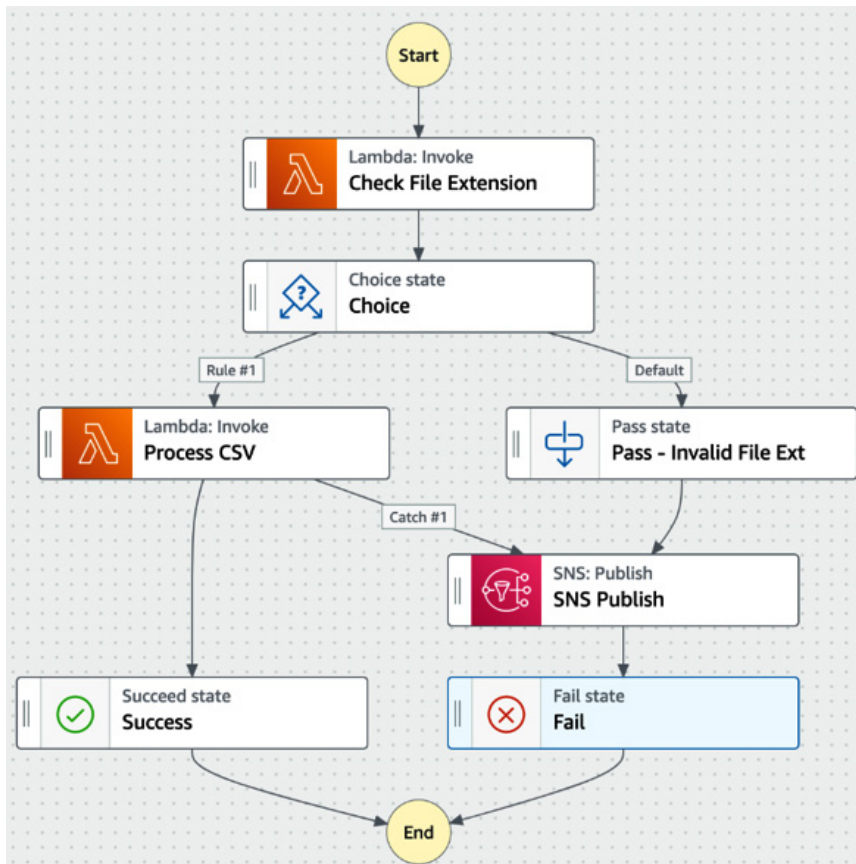


Figure 10.6 – The final status of our Step Function state machine

20. At the top right, click on **Next**. This screen shows the JSON Amazon States Language code that has been generated for your state machine. Click on **Next**.
21. For **State machine name**, enter `ProcessFilesStateMachine`. Leave all the other settings as-is and click **Create state machine**.

With that, we have created our pipeline orchestration using Step Function. Now, we want to create an event-driven Workflow for triggering the Step Function. In the next section, we will create a new EventBridge rule that will trigger our state machine whenever a new file is uploaded to a specific S3 bucket.

The Amazon EventBridge service is a serverless event bus that can be used to build event-driven Workflows. EventBridge can detect events from various AWS services (such as a new file being uploaded to S3) and can be configured to trigger a variety of different targets in response to an event. In our case, we will configure our Step Function as a target.

Configuring AWS CloudTrail and Amazon EventBridge

The AWS CloudTrail service is used to log the activities that are performed in an AWS account in near-real time. For example, when a new file is uploaded to Amazon S3, a CloudTrail event can be logged with details of the activity that took place.

Amazon EventBridge can monitor CloudTrail logs to detect certain events and respond to those. In the case of Amazon S3, however, object-level data events are not logged in CloudTrail by default, so we will need to configure our S3 bucket to generate CloudTrail data events.

Configuring Amazon S3 data events

For this exercise, we want to detect new files being created in our S3 Clean Zone bucket and have that trigger our Step Function state machine. The following steps will take you through the process of configuring CloudTrail data events:

1. Navigate to the **Amazon CloudTrail** service at `https://console.aws.amazon.com/cloudtrail/home`.
2. Ensure that you are in the region that you have been using for all the exercises in this book.
3. Expand the left panel and click on **Dashboard**.
4. Under **Trails**, click on **Create trail**.
5. For **Trail name**, enter `s3-data-events`.
6. Under **Customer managed AWS KMS key**, enter `s3-data-events-key` for **AWS KMS alias**.
7. Leave all the other options as-is and click **Next**.
8. For **Event type**, deselect **Management events** and select **Data events** instead.
9. Under **Data event: S3**, deselect the **Read** and **Write** options for **All current and future S3 buckets**.

10. Under **Individual bucket selection**, enter (or browse for) the name of your clean-zone bucket (such as `dataeng-clean-zone-<initials>`). Deselect **Read**, leaving only **Write** events selected. Then, click on **Next**.
11. After reviewing the summary screen, click on **Create trail**.

With the preceding steps, we have configured CloudTrail to record a log of all Write type events to our clean-zone bucket. In the next section, we will create a new EventBridge event that will detect Write events to the clean-zone bucket and trigger our Step Function in response.

Create an EventBridge rule for triggering our Step Function state machine

Our final task, before testing our pipeline, is to configure the EventBridge rule that will trigger our Step Function state machine. Let's get started:

1. Navigate to the **Amazon EventBridge** service at `https://console.aws.amazon.com/events/home`.
2. Ensure that you are in the region that you have been using for all the exercises in this book.
3. From the left-hand panel, click on **Rules**. Then, click on **Create rule**.
4. For the rule's name, enter `dataeng-s3-trigger-rule`.
5. Under **Define pattern**, select **Event pattern**, and then select **Pre-defined pattern by service**.
6. For **Service provider**, select **AWS**. For **Service name**, select **Simple Storage Service (S3)**. For **Event type**, select **Object level operations**.
7. Select **Specific operations**. Then, from the drop-down list, select the **PutObject**, **CopyObject**, and **CompleteMultipartUpload** operations.
8. Select **Specific bucket(s) by name** and enter the name of your clean-zone bucket (for example, `dataeng-clean-zone-<initials>`):

Build or customize an Event Pattern or set a Schedule to invoke Targets.

Event pattern [Info](#)
 Build a pattern to match events

Schedule [Info](#)
 Invoke your targets on a schedule

Event matching pattern
 You can use pre-defined pattern provided by a service or create a custom pattern

Pre-defined pattern by service
 Custom pattern

Service provider
 AWS services or custom/partner services

Service name
 The name of partner service selected as the event source

Event type
 The type of events as the source of the matching pattern

Info AWS API Call Events sent by CloudTrail will only match your rules if you have trail(s) (optionally with event selectors) configured to receive those events. See [CloudTrail](#) for further details.

Any operation
 Specific operation(s)

Any bucket
 Specific bucket(s) by name

Event pattern

```

1 {
2   "source": ["aws.s3"],
3   "detail-type": ["AWS API Call via CloudTrail"]
4   "detail": {
5     "eventSource": ["s3.amazonaws.com"],
6     "eventName": ["PutObject", "CopyObject", "Co
7     "requestParameters": {
8       "bucketName": ["dataeng-clean-zone-gs
9     }
10  }
11 }
  
```

Figure 10.7 – Specifying the event pattern for an EventBridge rule

- Scroll down to **Select targets** and for **Target**, select **Step Function state machine** from the drop-down list.

10. For **State machine**, select **ProcessFileStateMachine**, which we created previously.
11. Leave all the other settings as-is and click **Create**.

With that, we have put together an event-driven Workflow to orchestrate a data pipeline using Amazon Step Function. Our last task is to test our pipeline.

Testing our event-driven data orchestration pipeline

To test our pipeline, we need to upload a file to our clean-zone S3 bucket. Once the file has been uploaded, the rule we created in Amazon EventBridge will cause our Step Function state machine to be triggered:

1. Navigate to the Amazon S3 service at `https://s3.console.aws.amazon.com/s3`.
2. From the list of buckets, click on the **dataeng-clean-zone-<initials>** bucket.
3. Optionally, create a new folder in this bucket for testing.
4. Click on **Upload**, then **Add files**. Browse your computer for a file with a CSV extension (if you cannot find one, create a new, empty file and make sure that the file is saved with an extension of CSV).
5. Leave the other settings as-is and click **Upload**.
6. Navigate to the AWS Step Function service at `https://console.aws.amazon.com/states`.
7. Click on the state machine we created earlier (`ProcessFilesStateMachine`). From the list of **Executions**, see whether the state machine **Succeeded** or **Failed**. Click on the **Name** property of the execution for more details.
8. Reupload the same .csv file (multiple times if necessary) and notice how some executions succeed and some fail. The random number generator has a 66% chance of generating the number 1 or 2 and a 33% chance of generating the number 0. When the number 0 is generated, the function will fail, so throughout many executions, approximately one-third should fail.

The following diagram shows an example of what our state machine looks like after an execution where 0 was generated as a random number, causing the Lambda function to fail:

Graph inspector

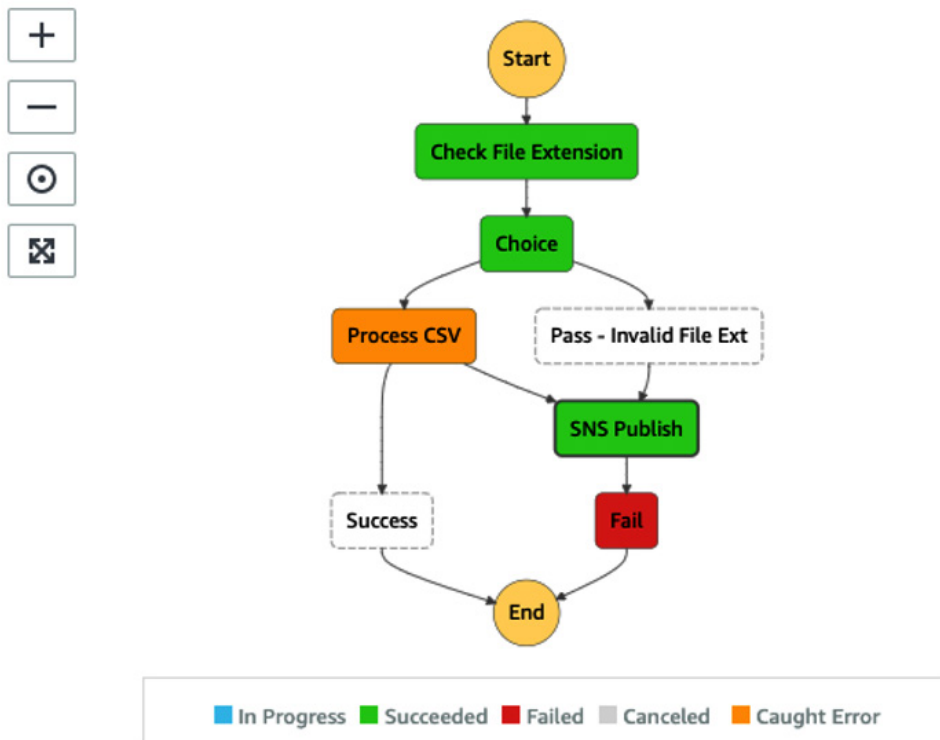


Figure 10.8 – An example of a state machine run when the random number generator generated a 0, resulting in a failed state machine

9. After a failed execution, check the email address that you specified when you configured the Amazon SNS notification service. If you previously confirmed your SNS subscription, you should receive an email each time the state machine fails.
10. Now, upload another file to the same Amazon S3 bucket, but ensure that this file has an extension other than .csv (for example, PDF). When you're viewing the execution details for your state machine, you should see that the choice state proceeded to the **Pass – Invalid File Ext** state and then also published an SNS notification to your email.

In the hands-on activity for this chapter, we created a serverless pipeline that we orchestrated using the AWS Step Function service. Our pipeline was configured to be event-driven via the Amazon EventBridge service, which let us trigger the pipeline in response to a new file being uploaded to a specific Amazon S3 bucket.

This was a fairly simple example of a data pipeline. However, AWS Step Function can be used to orchestrate far more complex data pipelines, with advanced error handling and retries. For more information on advanced error handling, see the AWS blog titled *Handling Errors, Retries, and Adding Alerting to Step Function state machine Executions* (<https://aws.amazon.com/blogs/developer/handling-errors-retries-and-adding-alerting-to-step-function-state-machine-executions/>).

Summary

In this chapter, we looked at a critical part of a data engineers' job: designing and orchestrating data pipelines. First, we examined some of the core concepts around data pipelines, such as scheduled and event-based pipelines, and how to handle failures and retries.

We then looked at four different AWS services that can be used for creating and orchestrating data pipelines. This included Amazon Data Pipeline, AWS Glue Workflows, Amazon Managed Workflows for Apache Airflow (MWAA), and AWS Step Function. We discussed some of the use cases for each of these services, as well as the pros and cons of them.

Then, in the hands-on section of this chapter, we built an event-driven pipeline. We used two AWS Lambda functions for processing and an Amazon SNS topic for sending out notifications about failure. Then, we put these pieces of our data pipeline together into a state machine orchestrated by AWS Step Function. We also looked at how to handle errors.

So far, we have looked at how to design the high-level architecture for a data pipeline and examined services for ingesting, transforming, and consuming data. In this chapter, we put some of these concepts together in the form of an orchestrated data pipeline.

In the remaining chapters of this book, we will take a deeper dive into some of the services for data consumption, including services for ad hoc SQL queries, services for data visualization, as well as an overview of machine learning and artificial intelligence services for drawing additional insights from our data.

In the next chapter, we will do a deeper dive into the Amazon Athena service, which is used for ad hoc data exploration, using the power of SQL.

Section 3: The Bigger Picture: Data Analytics, Data Visualization, and Machine Learning

In Section 3 of the book, we examine the bigger picture of data analytics in modern organizations. We learn about the tools that data consumers commonly use to work with data transformed by data engineers, and briefly look into how **machine learning (ML)** and **artificial intelligence (AI)** can draw rich insights out of data. We also get hands-on with tools for running ad hoc SQL queries on data in the data lake (Amazon Athena), for creating data visualizations (Amazon QuickSight), and for using AI to derive insights from data (Amazon Comprehend). We then conclude by looking at data engineering examples from the real world and explore some emerging trends in data engineering.

This section comprises the following chapters:

- *Chapter 11, Ad Hoc Queries with Amazon Athena*
- *Chapter 12, Visualizing Data with Amazon QuickSight*
- *Chapter 13, Enabling Artificial Intelligence and Machine Learning*
- *Chapter 14, Wrapping Up the First Part of Your Learning Journey*

11

Ad Hoc Queries with Amazon Athena

In *Chapter 8, Identifying and Enabling Varied Data Consumers*, we explored a variety of data consumers. Now, we will start examining the AWS services that some of these different data consumers may want to use, starting with those that need to use SQL to run ad hoc queries on data in the **data lake**.

SQL syntax is widely used for querying data in a variety of databases, and it is a skill that is easy to find. As a result, there is significant demand from various data consumers for the ability to query data that is in the data lake using SQL, without having to first move the data into a dedicated traditional database.

Amazon Athena is a serverless, fully managed service that lets you use SQL to directly query data in the data lake, as well as query various other databases. It requires no setup, and the cost is based purely on the amount of data that is scanned to complete the query.

In this chapter, we will do a deep dive into Athena, examining how Athena can be used to query data directly in the data lake, query data from other data sources with Query Federation, and how Athena provides workgroup functionality to help with governance and cost management.

In this chapter, we will cover the following topics:

- An introduction to Amazon Athena
- Tips and tricks to optimize Amazon Athena queries
- Federating the queries of external data sources with Amazon Athena Query Federation
- Managing governance and costs with Amazon Athena Workgroups
- Hands-on – creating an Amazon Athena workgroup and configuring Athena settings
- Hands-on – switching Workgroups and running queries

Technical requirements

In the hands-on sections of this chapter, you will perform administrative tasks related to Amazon Athena (such as creating a new Athena workgroup) and run Athena queries. As mentioned at the start of this book, we strongly recommend that, for the exercises in this book, you use a sandbox account where you have full administrative permissions.

For this chapter, at a minimum, you will need permissions to manage Athena Workgroups, permissions to run Athena queries, access to the **AWS Glue data catalog** for databases and tables to be queried, and read access to the relevant underlying S3 storage.

A user that has the **AmazonAthenaFullAccess** and **AmazonS3ReadOnlyAccess** policies attached should have sufficient permissions for the exercises in this chapter. However, note that a user with these roles will have access to all S3 objects in the account, all Glue resources, all Athena permissions, as well as various other permissions, so this should only be granted to users in a sandbox account. Such broad privileges should be avoided for users in production accounts.

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/tree/main/Chapter11>

Amazon Athena – in-place SQL analytics for the data lake

Structured Query Language (SQL) was invented at **IBM** in the 1970s but has remained an extremely popular language for querying data throughout the decades. Every day, millions of people across the world use SQL directly to explore data in a variety of databases, and many more use applications (whether business applications, mobile applications, or others) that, under the covers, use SQL to query a database.

Over the years, the **American National Standards Institute (ANSI)** has created various versions of an ANSI-SQL standard that database vendors can use to build ANSI-SQL-compliant databases. Database vendors often declare that their database is compatible with a large subset of ANSI-SQL, meaning that different database engines support different aspects of the ANSI-SQL standard.

Facebook, the social media network, has very large datasets and complex data analysis requirements and found that existing tools in the Hadoop ecosystem were not able to meet their needs. As a result, Facebook created an internal solution for being able to run SQL queries on their very large datasets, using standard ANSI SQL semantics, and in 2013, Facebook released this as an open source solution called Presto.

In late 2016, AWS announced the launch of Amazon Athena, a new service that would enable customers to directly query structured and semi-structured data that exists in Amazon S3. In the launch announcement, Amazon indicated that Athena was a managed version of Presto, with full standard SQL support. This provided the power of the **Presto SQL analytics engine** as a serverless service to AWS customers.

SQL is broadly broken into two parts:

- **Data Definition Language (DDL)**, which is used to create and modify database objects.
- **Data Manipulation Language (DML)**, which is used to query and manipulate data.

In 2021, AWS upgraded the Amazon Athena engine to v2, which is based on HiveQL for DDL statements, and Presto version 0.217 for DML statements.

Amazon Athena requires a Hive-compatible data catalog that provides the metadata for the data being queried. The catalog provides a logical view (databases that contain tables, which consist of rows, along with columns of a specific data type), and this maps to physical files stored in Amazon S3. Athena originally had its own data catalog, but today, it requires the use of the AWS Glue data catalog as its Hive compatible data store.

Amazon Athena makes it easy to quickly start querying data in an Amazon S3-based data lake, but there are some important things to keep in mind to optimize SQL queries with Amazon Athena, as we will discuss in the next section.

Tips and tricks to optimize Amazon Athena queries

When **raw data** is ingested into the data lake, we can immediately create a table for that data in the AWS Glue data catalog (either using a **Glue crawler** or by running DDL statements with Athena to define the table). Once the table has been created, we can start exploring the table by using Amazon Athena to run SQL queries against the data.

However, raw data is often ingested in plaintext formats such as CSV or JSON. And while we can query the data in this format for ad hoc data exploration, if we need to run complex queries against large datasets, these raw formats are not efficient to query. There are also ways that we can optimize the SQL queries that we write to make the best use of the underlying Athena query engine.

Amazon Athena's cost is based on the amount of compressed data that is scanned to resolve the query, so anything that can be done to reduce the amount of data scanned improves query performance and reduces query cost.

In this section, we will review several ways that we can optimize our analytics for increased performance and reduced cost.

Common file format and layout optimizations

The most impactful and easiest transformations that a data engineer can apply to raw files are those that transform the raw files into an optimized file format, and that structure the layout of files in an optimized way.

Transforming raw source files to optimized file formats

As we discussed in *Chapter 7, Transforming Data to Optimize for Analytics*, file formats such as **Apache Parquet** are designed for analytics and are much more performant than raw data formats such as CSV or JSON. So, transforming your raw source files into a format such as Parquet is one of the most important things a data engineer can do to improve the performance of Athena queries. Review the *Optimizing the file format* section of *Chapter 7, Transforming Data to Optimize for Analytics*, for a more comprehensive look at the benefits of Apache Parquet files.

In *Chapter 7, Transforming Data to Optimize for Analytics* we reviewed how the AWS Glue service can be used to transform your files into optimized formats. However, Amazon Athena can also transform files using a concept called **Create Table As Select (CTAS)**. With this approach, you run a CTAS statement using Athena, and this instructs Athena to create a new table based on a SQL select statement against a different table.

In the following example, `customers_csv` is the table that was created on the data we imported from a database to our data lake, and the data is in CSV format. If we want to create a Parquet version of this table so that we can effectively query it, we could run the following SQL statement using Athena:

```
CREATE TABLE customers_parquet
WITH (
    format = 'Parquet',
    parquet_compression = 'SNAPPY')
AS SELECT *
FROM customers_csv;
```

The preceding statement will create a new table called `customers_parquet`. The underlying files for this table will be in Parquet format and compressed using the Snappy compression algorithm. The contents of the new table will be the same as the `customers_csv` table since our query specified `SELECT *`, meaning select all data.

If you are bringing in specific datasets regularly (such as every night), then in most scenarios, it would make sense to configure and schedule an AWS Glue job to perform the conversion to Parquet format. But if you're doing ad hoc exploratory work on various datasets, or a one-time data load from a system, then you may want to consider using Amazon Athena to perform the transformation. Note that there are some limitations in using Amazon Athena to perform these types of transforms, so refer to the *Considerations and Limitations for CTAS Queries* (<https://docs.aws.amazon.com/athena/latest/ug/considerations-ctas.html>) page in the Amazon Athena documentation for more details.

Partitioning the dataset

This is also a concept that we covered in more detail in *Chapter 7, Transforming Data to Optimize for Analytics* but we will discuss it again now briefly as, after using an optimized file format such as Parquet, this is the next most impactful thing you can do to increase the performance of your analytic queries. Review the *Optimizing with Data Partitioning* section of *Chapter 7, Transforming Data to Optimize for Analytics*, for more details on partitioning.

A common data partitioning strategy is to partition files by columns related to date. For example, in our `sales` table, we could have a `YEAR` column, a `MONTH` column, and a `DAY` column that reflect the year, month, and day of a specific sales transaction, respectively. When the data is written to S3, all sales data related to a specific day will be written out to the same S3 prefix path.

Our partitioned dataset may look as follows:

```
/datalake/transform_zone/sales/YEAR=2021/MONTH=9/DAY=29/sales1.parquet
```

```
/datalake/transform_zone/sales/YEAR=2021/MONTH=9/DAY=30/sales1.parquet
```

```
/datalake/transform_zone/sales/YEAR=2021/MONTH=10/DAY=1/sales1.parquet
```

```
/datalake/transform_zone/sales/YEAR=2021/MONTH=10/DAY=2/sales1.parquet
```

Note

The preceding partition structure is a simple example because generally, with large datasets, you would expect to have multiple Parquet files in each partition.

Partitioning provides a significant performance benefit when you filter the results of your query based on one or more partitioned columns using the `WHERE` clause. For example, if a data analyst needs to query the total sales for the last day of September 2021, they could run the following query:

```
select sum(SALE_AMOUNT) from SALES where YEAR = '2021' and  
MONTH = '9' and DAY = '30'
```

Based on our partitioning strategy, the preceding query would only need to read the file (or files) in the single S3 prefix of `/datalake/transform_zone/sales/YEAR=2021/MONTH=9/DAY=30`.

Even if we want to query the data for a full month or year, we still significantly reduce the number of files that need to be scanned, compared to having to scan all the files for all the years if we did not partition our data.

As covered in *Chapter 7, Transforming Data to Optimize for Analytics*, you can specify one or more columns to partition by when writing out data using **Apache Spark**. Alternatively, you can use Amazon Athena CTAS statements to create a partitioned dataset. However, note that a single CTAS statement in Athena can only create a maximum of 100 partitions.

Other file-based optimizations

Using an optimized file format (such as Apache Parquet) and partitioning your data are generally the two strategies that will have the biggest positive impact on analytic performance. However, several other strategies can fine-tune performance, which we will cover here briefly.

Optimizing file size: It is important to avoid having a large number of small files if you want to optimize your analytic queries. For each file in S3, the analytic engine (in this case, Amazon Athena) needs to do the following:

- Open the file.
- Read the Parquet metadata to determine whether the query needs to scan the contents of the file.
- Scan the contents of the file if the file contains data needed for the query.
- Close the file.

There can be significant **Input/Output (I/O)** overhead in listing out very large numbers of files and then processing each file. **Airbnb** has an interesting blog post on *Medium* (<https://medium.com/airbnb-engineering/on-spark-hive-and-small-files-an-in-depth-look-at-spark-partitioning-strategies-a9a364f908>) that explains an issue they had where one of their data pipeline jobs ended up creating millions of files, and how this caused significant outages for them.

To optimize for analytics, you should aim for file sizes of between 128 MB and 1,024 MB.

Bucketing: Bucketing is a concept that is related to partitioning. However, with bucketing, you group rows of data together in a specified number of buckets, based on the hash value of a column (or columns) that you specify. Currently, Athena is not compatible with the bucketing implementation that's used in Spark, so you should use Athena CTAS statements to bucket your data. Refer to the Amazon Athena documentation on *Bucketing versus Partitioning* for more information (<https://docs.aws.amazon.com/athena/latest/ug/bucketing-vs-partitioning.html>).

Partition Projection: In scenarios where you have a very large number of partitions, there can be a significant overhead for Athena to read all the information about partitions from the Glue catalog. To improve performance, you can configure partition projection, where you provide a configuration pattern to reflect your partitions. Athena can then use this configuration information to determine possible partition values, without needing to read the partition information from the catalog.

For example, if you have a column called `YEARMONTH` that you partition on, and you have data going back to 2005, you could configure the partition projection range as `200501, NOW` and the partition projection format as `YYYYMM`. Athena would then be able to determine all possible valid partitions for that period without needing to read the partition information from the Glue catalog. For more information on partition projection, see the AWS documentation titled *Partition Projection with Amazon Athena* (<https://docs.aws.amazon.com/athena/latest/ug/partition-projection.html>).

In addition to the file and layout optimizations, there are also ways to write SQL queries so that the queries are optimized for the Presto analytic engine. We will cover some of these optimizations in the next section.

Writing optimized SQL queries

The way that SQL queries are written can also have a significant impact on the performance of the query.

In this section, we will review the top three best practices that will help provide optimal performance of queries. It's recommended that, as a data engineer, you share these best practices with data analysts and others using Athena to run queries.

That said, there are other ways, beyond the three best practices we will outline here, to go deep into query optimization. For example, you (or your end user data analysts) can use the `EXPLAIN` statement as part of an Athena query to view the logical execution plan of a specific SQL statement. You can then make modifications to your SQL statement and review the `EXPLAIN` query plan to understand how that changes the underlying execution plan. For more information, see the AWS Athena documentation titled *Using the EXPLAIN Statement in Athena*: <https://docs.aws.amazon.com/athena/latest/ug/athena-explain-statement.html>.

We don't have space to cover these additional query optimization techniques in this chapter. However, the AWS documentation provides a deeper dive into these optimizations, so for more information, please refer to the AWS Athena documentation titled *Performance Tuning in Athena: Performance Tuning in Athena*

Selecting only the specific columns that you need

When exploring data, it is common to run queries that select all columns by specifying the start of the query as `select *`. However, remember that the Parquet format that we recommend for analytics is a columnar-based file format, meaning that data stored on disk is grouped by columns rather than rows. When you specify a specific column to query, the analytic engine (such as Athena) can read the data for that column only.

If you have a table with a lot of columns, specifying just the columns that are important to your query can significantly increase the performance of your query. This is because Athena does not need to process the data for all columns. Since Athena's cost is based on the amount of data that's scanned, selecting just specific columns can also result in significant cost savings.

Take a scenario where you have a table with 150 columns, but your specific query only needs data from 15 of the columns. In this scenario, Athena would scan approximately 10% of the dataset, compared to a query that uses a `select *` to query all columns.

Using approximate aggregate functions

The Presto database engine (and therefore Athena) supports a wide variety of functions and operators that can be used in queries. These include functions that can be used in calculations against large datasets in a data lake. They are used for tasks such as the following:

- Working out the sum of all sales for this month compared to last month
- Calculating the average number of orders per store
- Determining the total number of unique users that accessed our e-commerce store yesterday
- Other advanced statistical calculations

For some calculations, you may need to get a fully accurate calculation, such as when determining sales figures for formal financial reporting. At other times, you may just need an approximate calculation, such as for getting an estimate on how many unique visitors came to our website yesterday.

For those scenarios, where you can tolerate some deviation in the result, Presto provides approximate aggregate functions, and these offer significant performance improvements compared to the equivalent fully accurate version of the function.

For example, if we needed to calculate the approximate number of unique users that browsed our e-commerce store in the past 7 days, and we could tolerate a standard deviation of 2.3% in the result, we could use the `approx_distinct` function, as follows:

```
SELECT
    approx_distinct (userid)
FROM
    estore_log
WHERE
    visit_time > to_iso8601(current_timestamp - interval '7'
day)
```

For more information on supported Presto functions in Athena, including approximate functions, refer to the Athena documentation titled *Presto Functions in Amazon Athena*: <https://docs.aws.amazon.com/athena/latest/ug/presto-functions.html>

Using regular expressions instead of using the like operator

A common way to select rows of data that match a specific pattern is to use the `like` operator, as shown in the following query:

```
select
    category_name, count(category_name)
from
    film_category
where
    category_name like 'Comedy' or category_name like 'Drama'
or category_name like 'Music' or category_name like 'New'
group by
    category_name
```

The preceding query returns the selected categories (Comedy, Drama, Music, and New) along with the count of how many movies are in each of the categories.

An alternative approach to using the `like` operator is to use regular expressions for pattern matching, and this can both simplify the statement as well as increase the performance of the query.

The following query returns the same results as our previous query, but it uses the `regexp_like` operator instead, and also includes regular expression syntax (`?i`) to make the pattern matching case insensitive:

```
select
  category_name, count(category_name)
from
  film_category
where
  regexp_like(category_name, '(?i)^comedy|drama|music|new')
group by
  category_name
```

Using `regexp_like` instead of a regular `like` is recommended to improve query performance, especially in scenarios where you need to do several comparisons that would require many `like` operators.

Now that we have reviewed some of the important factors that can affect Athena's performance, we can look at more advanced features in Athena, such as the ability to federate queries, which we will discuss next.

Federating the queries of external data sources with Amazon Athena Query Federation

As we've discussed several times in this book, Athena lets you query data that has been loaded into the data lake using standard SQL semantics. But since the launch of Athena, AWS has added additional functionality to enhance Athena to make it an even more powerful query engine.

One of those major enhancements, which became available in 2021 with Athena query engine v2, was the ability to run federated queries, which we will look at next.

Querying external data sources using Athena Federated Query

Query federation, also sometimes referred to as **data virtualization**, is the process of querying multiple external data sources, in different database engines or other systems, through a single SQL query statement. In November 2019, AWS announced the preview of **Federated Query** in Amazon Athena, which enables a single Athena query to query data in data lakes, as well as data from external sources.

Data lakes are designed to collect data from multiple systems in an organization and bring it into centralized storage, where the data can be combined in ways that unlock value for the business.

However, it is not practical to bring every single dataset that an organization has created into the centralized storage of the data lake. For some datasets, the organization either does not need to keep all historical data for a dataset, or the data is currently in a system that already stores historical data. In these scenarios, it may make more sense to query the source dataset directly and combine data from the source with data in the data lake on the fly.

If a dataset needs to be queried by multiple teams, is queried often, and queries need to return very large amounts of data, then it may be best to load that dataset directly into the data lake. Also, if you need to repeatedly query a system that is already under a relatively heavy load, you can reduce that load by loading data from the system to the data lake in off-peak hours, rather than running federated queries during peak times.

But if you're performing ad hoc queries, or if the data only needs to be queried by a small number of teams with a relatively low frequency of querying, then using the *Athena Federated Query* functionality to access the data makes sense. Several people have run performance testing with Athena Federated Query and have proven the ability to query many tens of thousands of records per second.

For example, Athena Federated Query could enable a data analyst to run a single SQL query that combines the following datasets:

- Master customer data in Amazon S3
- Current order information in Amazon Aurora
- Shipment tracking data in Amazon DynamoDB
- Product catalog data in Amazon Redshift:

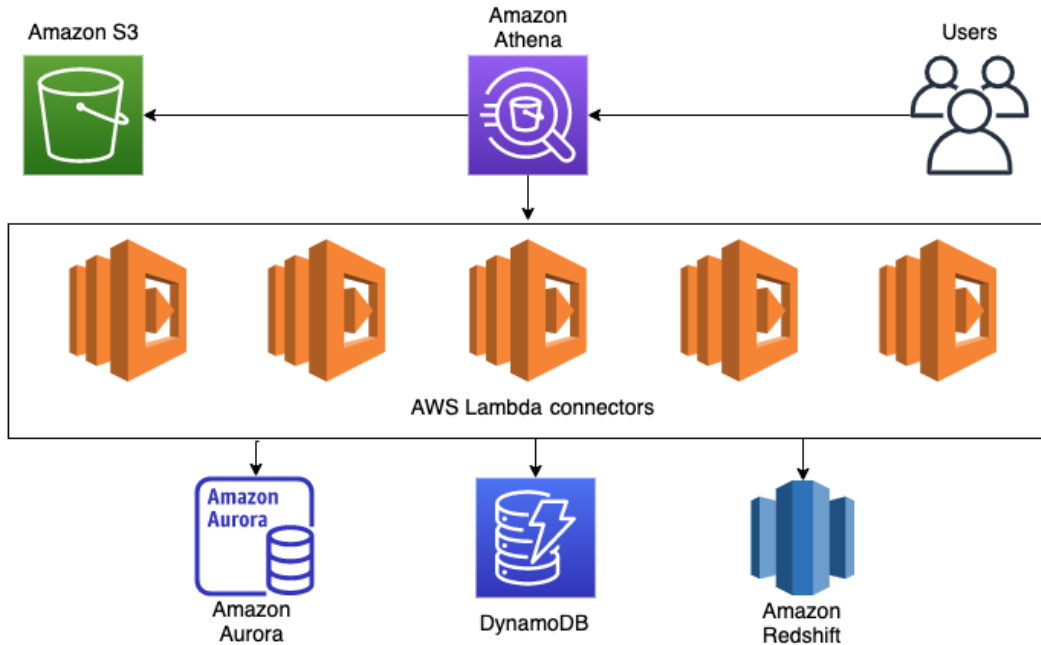


Figure 11.1 – Amazon Athena query federation

Another use case would be if you do a nightly load of data from an external system into your data lake, but a few of your queries need to be able to reference some real-time data. For example, if supplier order information was loaded into the data lake each night, but you had a query that needed to calculate the total number of orders for a specific supplier for the year up to the present time, your query could do the following:

- Read supplier order information from the S3 data lake for all orders from the beginning of the year up until yesterday.
- Read any orders from today from your SAP HANA system.

There are many other potential use cases where data processing can be simplified by having the ability to use pure SQL to read and manipulate data from multiple systems. Rather than having to read independently from multiple systems, and then programmatically process the data, data processing is simplified by processing the data with a single SQL query.

Pre-built connectors and custom connectors

Athena Federated Query uses code running in **AWS Lambda** to connect to and query data, as well as metadata, from the external systems. When a query runs that uses a connected data source, Athena invokes the relevant Lambda function/s to read metadata, identifies parts of the tables that need to be read, and launches multiple Lambda functions to read the data in parallel.

AWS has open sourced several connectors that enable federated queries against many popular data sources, including the following:

- A **JDBC connector** for connecting to sources such as MySQL, Postgres, and Redshift.
- A **DynamoDB connector** for reading from the Amazon-managed NoSQL database.
- A **Redis connector** for reading data from Redis instances.
- A **CloudWatch logs and CloudWatch metrics connector**, enabling you to query your application log files and metrics using SQL.
- An **AWS CMDB connector** that integrates with several AWS services to enable SQL queries against your AWS resources. Integrated services include EC2, RDS, EMR, and S3.

The full list of connectors can be found on, and downloaded from, the AWS Labs Athena Query Federation GitHub page at <https://github.com/aws-labs/aws-athena-query-federation/wiki/Available-Connectors>.

In addition to the connectors made available by AWS, anyone can create custom connectors to connect to external systems. If you can make a network connection from AWS Lambda to the target system, whether on-premises or in the cloud, you could potentially create an Athena Federated Query connector for that system.

Third-party companies are also able to create connectors for Athena Federated Query. For example, a company called **Trianz** has created connectors for **Terradata**, **Snowflake**, **Google BigQuery**, **Cloudera**, **Oracle**, and other systems.

To learn more about building custom connectors, see the Athena Query Federation GitHub page titled *Amazon Athena Query Federation SDK*: <https://github.com/aws-labs/aws-athena-query-federation/blob/master/athena-federation-sdk/README.md>

So far, we have looked at the core Athena functionality for querying data inside and outside of the data lake. Now, let's take a look at some of the Athena functionality for managing governance and costs.

Managing governance and costs with Amazon Athena Workgroups

Athena costs are based on the amount of data that is scanned by a query, and in the first section of this chapter, we looked at some of the ways that data can be optimized so that queries would scan less data, and therefore reduce costs.

However, some of those optimizations are based on writing efficient SQL queries, and it's not unusual for organizations to be concerned that users are going to accidentally run SQL queries that are not optimized and end up scanning massive amounts of data. As such, organizations want a way to control the amount of data that's scanned by different users or teams.

Organizations also have concerns around governance and security. Some of these concerns include the following:

- Athena saves the results of all queries, as well as associated metadata, on S3. These results could contain confidential information, so organizations want to ensure this data is protected.
- Multiple teams in an organization may use Athena in the same AWS account, and organizations want items such as query history to be stored separately for each team.

Athena Workgroups overview

To help organizations manage these governance concerns, AWS introduced the concept of **Athena Workgroups**. Workgroups are a resource type that enables the separation of query execution and query history between different users, teams, or systems.

In the Athena console, users can save queries that they frequently run, and a list of historical queries that they have run are also available. However, these lists only show queries for the Workgroup where the query is run, so splitting up teams or projects into different Workgroups ensures that query history and saved queries are visible only to the specific team associated with the workgroup.

Workgroups also enable an organization to control the amount of data that's scanned (and therefore Athena costs) for each Workgroup. In addition, Workgroups can also be used to enforce several settings, including the S3 path for query results, and can control whether query results are encrypted or not.

Enforcing settings for groups of users

One of the primary uses of Athena Workgroups is to separate groups of different Athena users and to enforce settings for each group. These could be separate Workgroups for each team, separate Workgroups for different applications, or separate Workgroups for different types of users.

Workgroups enable an administrator to enforce various settings for each different group of users or different projects or use cases. By default, each user can control several settings, but Workgroups enable an administrator to override the users' settings, forcing them to use the Workgroup settings.

The following are the configuration items that an administrator can enforce for members of a Workgroup:

- **Query Result Location:** This is the S3 path where the results of Athena queries will be written. Users can set a query result location, but if this is set for the workgroup and **Override client-side settings** is set on the Workgroup, then this location will be used for all the queries that are run in this Workgroup.

This enables an organization to control where query result files are stored in S3, and the organization can set strict access control options on this location to prevent unauthorized users from gaining access to query results.

For example, each team can be assigned a different Workgroup, and their IAM access policy can be configured to only allow read access to their query results.

- **Encrypt Query Results:** This option can be used to enforce that query results are encrypted, helping organizations keep in line with their corporate security requirements.
- **Metrics:** You can choose to send metrics to CloudWatch logs, which will reflect items such as the number of successful queries, the query runtime, and the amount of data that's been scanned for all the queries that are run within this workgroup.
- **Override client-side settings:** If this item is not enabled, then users can configure their user settings for things such as query result location, and whether query results are encrypted. Therefore, it is important to enable this setting to ensure that query results are protected and corporate governance standards are met.
- **Requester pays S3 buckets:** When creating a bucket in Amazon S3, one of the options that's available is to configure the bucket so that the user that queries the bucket pays for the API access costs. By default, Athena will not allow queries against buckets that have been configured for requester pays, but you can allow this by enabling this item.

- **Tags:** You can provide as many `key:value` tags as needed to help with items such as cost allocation, or for controlling access to a Workgroup. For example, you may have two Workgroups that have different settings for the query output location, based on different projects or use cases. You could provide a tag with the name of the team and then, through IAM policies, provide team members access to all Workgroups that are tagged with their team name.

For examples of how to manage access to Workgroups using tags or workgroup names, see the Amazon Athena documentation titled *Tag-Based IAM Access Control Policies* (<https://docs.aws.amazon.com/athena/latest/ug/tags-access-control.html>).

In addition to using Workgroups to enforce different settings for different groups of users or use cases, Workgroups can also be used to manage costs by limiting the amount of data that's scanned.

Enforcing data usage controls

As Athena pricing is based on the amount of data scanned (at the time of writing, the cost is \$5 per TB of data scanned), limiting the amount of data that's scanned helps manage costs. To enable this, Athena Workgroups includes functionality for data usage controls, and two types of controls can be implemented.

Per query data usage control

You can configure the maximum amount of data that can be scanned by a single query using per query data usage controls. If a user runs a query and Athena ends up trying to scan more data than what's allowed in the control, the query is canceled. However, note that the AWS account is still billed for the amount of data that was scanned up until the query was canceled.

As a practical example, you may have a group of users that are relatively inexperienced with SQL and want to have a sandbox environment where they can run ad hoc queries safely. In this scenario, you could create an Athena Workgroup called *sandbox* and configure these users to have access to the sandbox Workgroup. You could configure the Workgroup to have a per-query limit of 100 GB, for example, which would ensure that no individual query would cost more than \$0.50.

Per query data limits are useful for scenarios where you want to have hard control over the amount of data that's scanned by each query. However, this control is restrictive in that it automatically cancels any query that exceeds the specified amount of data that's scanned. An alternative option for controlling costs is to configure Workgroup data usage controls.

Workgroup data usage controls

With Workgroup data usage controls, you have the flexibility to configure the maximum amount of data that's scanned by the entire workgroup, within a specified period. And instead of it being a hard cancel of the query, workgroup data usage controls use **Amazon Simple Notification Service (SNS)** to trigger actions when the limit is exceeded.

For example, you can configure a workgroup data usage control for a maximum data scan of 3 TB per day. Then, you can configure an SNS topic that will email an administrator to inform them if the data scanned limit for the workgroup has been exceeded.

However, since multiple targets can be triggered by an SNS message, you can also do things such as automate a programmatic action when the limit you have set is reached. For example, you can create a Lambda function that can programmatically disable the workgroup, which would prevent any additional queries from being run in the Workgroup.

Now, let's get hands-on with Athena by creating and configuring a new workgroup, as well as running some SQL queries.

Hands-on – creating an Amazon Athena workgroup and configuring Athena settings

In this section, we're going to create and configure a new Athena Workgroup and learn more about how Workgroups can help separate groups of users:

1. Log into **AWS Management Console** and access the Athena service using this link: <https://console.aws.amazon.com/athena>.
2. Expand the left-hand menu, and click on Workgroups to access the workgroup management page.

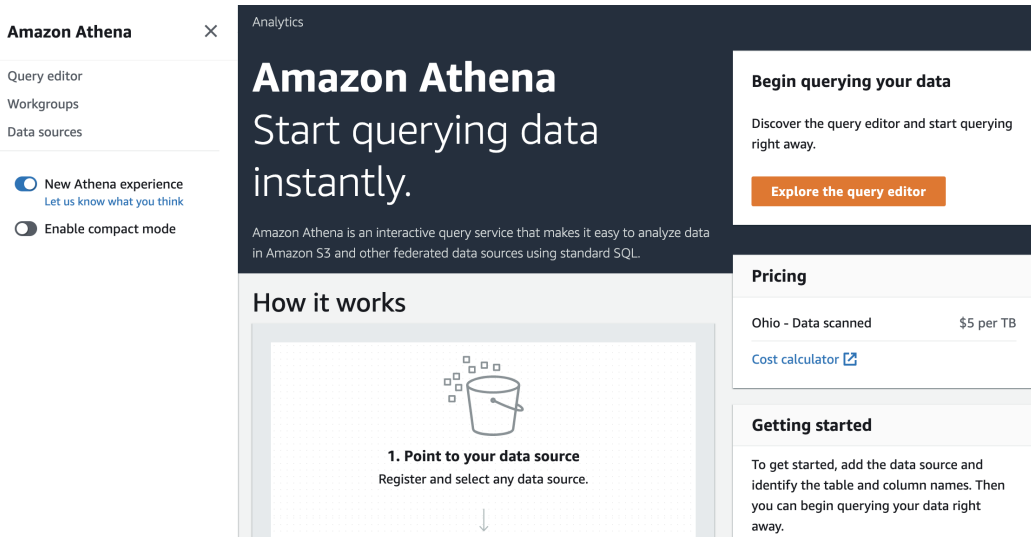


Figure 11.2 – Athena Console showing Workgroups

3. On the **Workgroup management** page, click on **Create workgroup** and enter the following values for our new Workgroup. For the items not listed here, leave the defaults as-is:
 - **Workgroup name:** Provide a descriptive name for the Workgroup, such as **datalake-user-sandbox**.
 - **Description:** Optionally, provide a description for this Workgroup, such as **Sandbox Workgroup for new datalake-users**.
 - **Query result location:** In the hands-on exercises in *Chapter 4, Data Cataloging, Security, and Governance*, we created a bucket to store our Athena query results in (named `aws-athena-query-results-dataengbook-<initials>`). Click the **Browse S3 button** next to **Query result location**, and select the previously created query result bucket selector, and then click Choose. To make the location of our query results unique for this Workgroup, add the Workgroup name to the end of the path. For example, the full path should be something like `s3://aws-athena-query-results-dataengbook-xxxxx/datalake-user-sandbox/`. Make sure that you include the trailing slash at the end of the path.

- **Encrypt Query Results:** Tick this box to ensure that our query results are encrypted. When you select this, you will see several options for controlling the type of encryption. For our purposes, select **SSE-S3** for **Encryption type** (this specifies that we want to use S3 Server-Side Encryption rather than our own unique KMS encryption key).
 - **Override client-side settings:** If we want to prevent our users from changing items such as the query result's location or encryption settings, we need to ensure that we select this option.
4. In the Per query data usage control section, we can specify a Data limit to limit the scan size for individual queries run in this workgroup. If a query scans more than this amount of data, the query will be canceled. Set the Data limit size to 10 GB.

Per query data usage control - optional [Info](#)

Sets the limit for the maximum amount of data a query is allowed to scan. You can set only one per query limit for a workgroup. The limit applies to all queries in the workgroup and if query exceeds the limit, it will be cancelled.

Data limit

Minimum limit is 10 MB and maximum limit is 7 EB per workgroup. Numeric characters only.

▶ **Workgroup data usage alerts - optional** [Info](#)

Set multiple alert thresholds when queries running in this workgroup scan a specified amount of data within a specific period. Alerts are implemented using [Amazon CloudWatch alarms](#) and applies to all queries in the workgroup.

Tags - optional [Info](#)

You can edit tag keys and values, and you can remove tags from a data source at any time. Tag keys and values are case-sensitive. For each tag, a tag key is required, but tag value is optional. Do not use duplicate tag keys in the same data source.

Key	Value	
<i>Enter key</i>	<i>Enter value</i>	Remove
Use 1 - 128 characters. (A-Z,a-z,0-9, _.,:./,=,*,-,@)	Use up to 256 characters. (A-Z,a-z,0-9, _.,:./,=,*,-,@)	
<input type="button" value="Add key/value"/>		
You can add up to 50 items		

Figure 11.3 – Athena Workgroup – Per query data usage control

5. Optionally add any Tags you want to specify, and then click on Create workgroup.

We can also set a **Workgroup data usage control** to manage the total amount of data that is scanned by all users of the Workgroup over a specific period. We are not going to cover this now, but if you would like to explore setting this up, refer to the AWS documentation titled *Setting Data Usage Controls Limits*: <https://docs.aws.amazon.com/athena/latest/ug/workgroups-setting-control-limits-cloudwatch.html>

Hands-on – switching Workgroups and running queries

By default, all users operate in the **primary** Workgroup, but users can switch between any workgroup that they have access to. You can control Workgroup access via IAM policies, as detailed in the AWS documentation titled *IAM Policies for Accessing Workgroups*: <https://docs.aws.amazon.com/athena/latest/ug/workgroups-iam-policy.html>

In the previous section, we created and configured a new Workgroup, so we can now run some SQL queries and explore Athena's functionality further:

1. In the left-hand menu, click on Query editor. Once in the Query editor, use the Workgroup drop-down list selector to change to the your newly created sandbox workgroup.

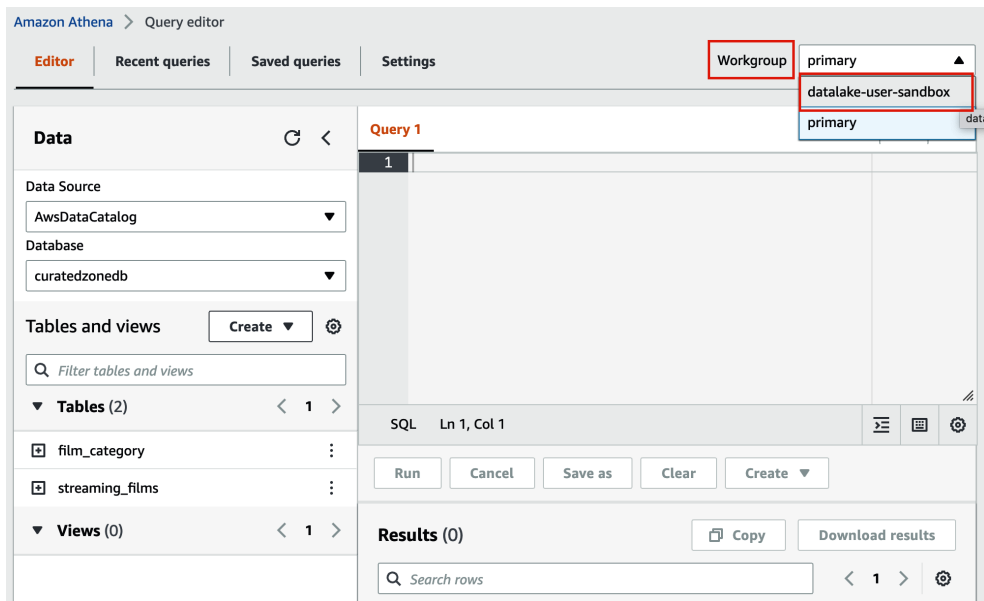


Figure 11.4 – Switching Workgroups in the Athena Console

2. A pop-up dialog may appear for you to acknowledge that all the queries that are run in this Workgroup will use the settings we configured previously. This is because we chose to **Overwrite client-side settings** when creating the workgroup. Click **Acknowledge** to confirm this.
3. In the Query editor, let's run our first query to determine which category of films is most popular with our streaming viewers. We're going to query the `streaming_films` table, which was the denormalized table we created in *Chapter 7, Transforming Data to Optimize for Analytics*. On the left-hand side of the Athena query editor, select the `curatedzonedb` database from the dropdown, and then run the following query in the query editor:

```
SELECT category_name,  
       count(category_name) streams  
FROM streaming_films  
GROUP BY category_name  
ORDER BY streams DESC
```

This query performs the following tasks:

- It selects the category name and a count of the total number of entries of that category in the table, and then it renames the count of queries column to create a new column heading of `streams`.
- It selects this data from the `streaming_films` table. Since we selected the `curatedzonedb` from the dropdown on the left-hand side, Athena automatically assumes that the table we are querying is in that selected database, so we don't need to specifically reference `curatedzonedb` in our query, although we could.
- Then, it groups the results by `category_name`, meaning that one record will be returned per category.
- Finally, it sorts the results by the `streaming` column, in descending order, so that the first result is the category with the highest number of streams. In the following screenshot, we can see that `Sports` was the most popular category from our streaming catalog:

The screenshot shows the Amazon Athena console interface. At the top, there are tabs for 'Editor', 'Recent queries', 'Saved queries', and 'Settings'. The 'Workgroup' is set to 'datalake-user-sand...'. The main area is divided into several sections:

- Data Source:** 'AwsDataCatalog' and 'Database: 'curatedzonedb'.
- Query Editor:** Contains the SQL query:


```
1 SELECT category_name,
2 count(category_name) streams
3 FROM streaming_films
4 GROUP BY category_name
5 ORDER BY streams DESC
```
- Execution Controls:** 'Run again', 'Cancel', 'Save as', 'Clear', and 'Create' buttons.
- Execution Status:** 'Completed', 'Time in queue: 0.124 sec', 'Run time: 0.415 sec', 'Data scanned: 2.59 KB'.
- Results:** A table with 16 rows. The first five rows are:

category_name	streams
Sports	258
Foreign	252
Documentary	248
Family	241
Sci-Fi	233
- Tables and views:** A sidebar on the left showing a list of tables, including 'streaming_films' with its schema details.

Figure 11.5 – Athena query for the top streaming categories

Note that the data in the `streaming_films` table was randomly generated by the Kinesis Data Generator utility in *Chapter 6, Ingesting Streaming and Batch Data*, so your results regarding the top category may be different.

- If we have a query that we think we may want to run regularly (such as seeing the top category each day), we can save the query so that we don't need to retype it each time we want to run it. To do so, just click on the **Save as** button below the query.
- Provide a name for the query (such as Overall-Top-Streaming-Categories) and a description (such as Returns a list of all categories, sorted by highest number of views). Then, click **Save query**.

- Now, let's modify our query slightly to find out which `State` streamed the most movies out of our streaming catalog. Click on the plus (+) sign to open a new query window and enter the following query:

```
SELECT state,
        count(state) count
FROM streaming_films
GROUP BY state
ORDER BY count desc
```

Running this query using the data I generated returns the following results. We can see that our catalog of films was most popular with viewers in Louisiana Again, though, your results may be different due to the random data we generated using the **Kinesis Data Generator**:

The screenshot shows the Athena Query Editor interface. The query editor contains the following SQL query:

```
1 SELECT state,
2     count(state) count
3 FROM streaming_films
4 GROUP BY state
5 ORDER BY count desc
```

The query has been executed successfully. The results are displayed in a table with the following data:

state	count
Louisiana	89
North Carolina	86
Washington	86
Wisconsin	85
Kentucky	84

Figure 11.6 – Athena Query Editor showing the total streams per state

Click on **Save as** and provide a name and description for this query.

7. Close the currently open query tabs, and then, via the top Athena menu, click on **Saved queries**. Here, we can see the list of queries that we have previously saved, and we can easily select a query from the list if we want to run that query again. Note that saved queries are saved as part of the Workgroup, so any of our team members that have access to this Workgroup will also be able to access any queries that we have saved. If you click on one of the saved queries, it will open the query in a **New query** tab.
8. At the top of the Athena menu, click on **Recent queries**. Here, we can see a list of all the recent queries that have been run in this Workgroup. There are several actions we can take:

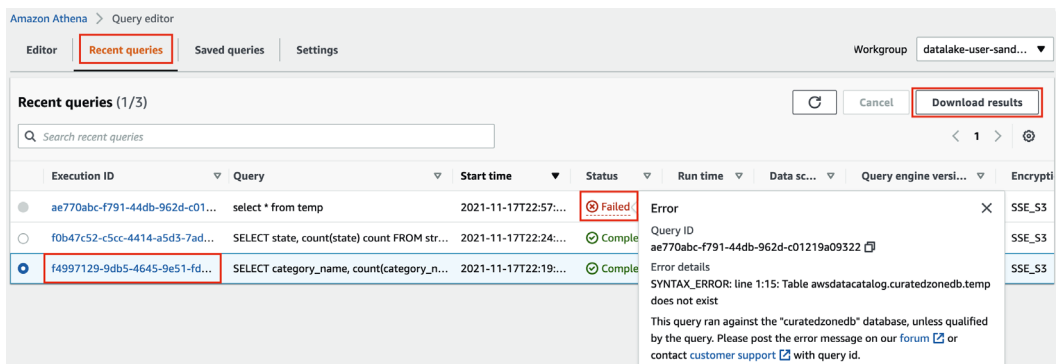


Figure 11.7 – Amazon Athena - Recent queries tab

These actions are as follows:

- A. If we want to rerun a query, we can click on the Execution ID of the query and it will open the query in a new query window.
- B. To download the results that the query generated as a CSV file, click on **Download Results**. Remember that query results are always stored on S3 in the location set for **Query Result Location**.
- C. To see the details of why a query failed, click on **Failed under the Status column**. **A pop-up box will provide details of the error message that caused the failure.**

Note that the **Recent queries** tab keeps a record of all the queries that have been run in the past 45 days.

In these hands-on exercises, you configured an Athena workgroup and made use of that workgroup to run several queries. You also learned how to save queries and view query history.

Summary

In this chapter, we had a deeper look at the Amazon Athena service, which is an AWS-managed version of Apache Presto. We looked at how to optimize our data and queries to increase query performance and reduce costs.

Then, we explored how Athena can be used as a SQL query engine – not only for data in an Amazon S3 data lake but also for external data sources such as other database systems, data warehouses, and even CloudWatch logs using Athena Query Federation.

Finally, we explored Athena Workgroups, which let us manage governance and costs. Workgroups can be used to enforce specific settings for different teams or projects, and can also be used to limit the amount of data that's scanned by queries.

In the next chapter, we will take a deeper dive into another Amazon tool for data consumers as we look at how we can create rich visualizations and dashboards using **Amazon QuickSight**.

12

Visualizing Data with Amazon QuickSight

In *Chapter 11, Ad Hoc Queries with Amazon Athena*, we looked at how **Amazon Athena** enables data analysts to run ad hoc queries against data in the data lake using the power of SQL. And while SQL is an extremely powerful tool for querying large datasets, often, the quickest way to understand a summary of a dataset is to visualize the data in graphs and dashboards.

In this chapter, we will do a deeper dive into **Amazon QuickSight**, a **Business Intelligence (BI)** tool that enables the creation of rich visualizations that summarize data, with the ability to filter and drill down into datasets in numerous ways.

In smaller organizations, a data engineer may be tasked with setting up and configuring a BI tool that data consumers can use. Things may be different in larger organizations, where there may be a dedicated team to manage the BI system. However, it is still important for a data engineer to understand how these systems work, as these systems will be consuming data that the data engineer will have played a part in creating.

The purpose of BI tools is to enable users to quickly understand complex datasets by enabling the exploration of data visually. And while we will focus on Amazon QuickSight in this chapter, many of the concepts in this chapter can be applied to other popular BI applications, such as Tableau, Microsoft Power BI, and Qlik.

Amazon QuickSight is a serverless BI solution and is fully managed by AWS. Organizations don't need to pay for any infrastructure costs, but rather pay a fixed amount per QuickSight user on a subscription basis.

In this chapter, we will cover the following topics:

- Representing data visually for maximum impact
- Understanding Amazon QuickSight's core concepts
- Ingesting and preparing data from a variety of sources
- Creating and sharing visuals with QuickSight analyses and dashboards
- Understanding QuickSight's advanced features – ML Insights and embedded dashboards
- Hands-on – creating a simple QuickSight visualization

Technical requirements

At the end of this chapter, you will get hands-on by creating a QuickSight visual from scratch. To complete the steps in the hands-on section, you will need the appropriate user permissions to sign up for a QuickSight subscription.

If you have administrator permissions for your AWS account, these permissions should be sufficient to sign up for a QuickSight subscription. If not, you will need to work with your IAM security team to create a custom policy. See the AWS documentation titled *IAM Policy Examples for Amazon QuickSight* and refer to the *All Access for Standard Edition* example policy as a reference.

At the time of writing, Amazon QuickSight includes a free trial subscription for 30 days for new QuickSight subscriptions. If you do not intend to use QuickSight past these 30 days, ensure that your user is also granted the `quicksight:Unsubscribe` permission so that you can unsubscribe from QuickSight after completing the hands-on section.

Note that the *All Access for Standard Edition* example policy has a specific deny for the unsubscribe permission, so this may need to be modified based on your requirements. Work with your security team to implement a custom IAM policy.

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/tree/main/Chapter12>

Representing data visually for maximum impact

Data lakes are designed to capture large amounts of raw data and enable the processing of that data to draw out new insights that provide business value. The insights that are gained from a data lake can be represented in many ways, such as reports that summarize sales data and top sales items, **machine learning (ML)** models that can predict future trends, and visualizations and dashboards that effectively summarize data. Each of these ways of representing data offers different benefits, depending on the business purpose:

- If you're a data analyst that needs to report sales figures, profit margins, inventory levels, and other data for each category of product the company produces, you would probably want access to detailed tabular data. You would want the power of SQL to run powerful queries against the data to draw varied insights so that you can provide this data to different departments within the organization.
- If you're a logistics manager and are responsible for supplying all your retail stores with the correct amount of inventory, you would want your data science team to develop an ML model that can predict inventory requirements for each store. The model could take in raw data from the data lake and predict how much inventory each store may require.
- If you're a sales manager for a specific product category, you need to have an updated view of sales for the products in your category at all times. You need to be able to determine which products are selling well, and which marketing campaigns are most effective. Seeing a visual representation of relevant data provides you with the most effective way to quickly understand the product and campaign's performance at a high level.

Having raw, granular data available to an organization is important, but when you need to make decisions quickly based on that data, having a visual representation of the data is critical.

It is not practical to identify trends or outliers in a dataset by examining a spreadsheet containing 10,000 rows. However, if you aggregate and summarize the data into a well-designed visual representation of the data, it becomes very easy to identify those trends and outliers.

Benefits of data visualization

A well-designed visual representation of data can reflect multiple different datasets in a single picture. It can do so in a way that enables the consumer of the visual to immediately gain insights that would take significant time and effort to gather from raw data.

There is a common fact often mentioned in articles on the internet that people can process images 60,000 times faster than they can process text. As it turns out, this is just an often-repeated claim with no evidence to back it up. However, while the number may be exaggerated, the basic claim that the human brain can process images quicker than text is without a doubt true.

And you don't need to look too far to validate this claim. For example, look at the rise of visual-based social media sites such as Instagram and Pinterest, or how people use emojis and animated GIFs to quickly and effectively communicate how they feel about something.

In the same way, we can use the power of visuals (images, graphs, word clouds, and many other types) to effectively communicate data from our data lake in a way that makes it easy for the consumer of the visual to quickly draw insights from the data.

Let's examine some common uses of visualizations that enable a user to quickly understand complex information.

Popular uses of data visualizations

Visualizations can be used to draw insights from many different types of data, in various ways. In this section, we look at a few examples of some common types of visualizations to demonstrate the impact of a well-designed visual.

Trends over time

A common usage of analytic tools is to crunch through raw data to help surface trends, or changes in the data, over time. For example, we may want to understand how our spending on the AWS platform is changing over time, as this can help identify areas where we need to focus on cost optimizations. A line graph can be a useful way to illustrate changes in data over a certain period.

The following diagram was created using a popular spreadsheet application and provides a visual of raw Amazon S3 spend per month, over 9 months, for a fictional company:

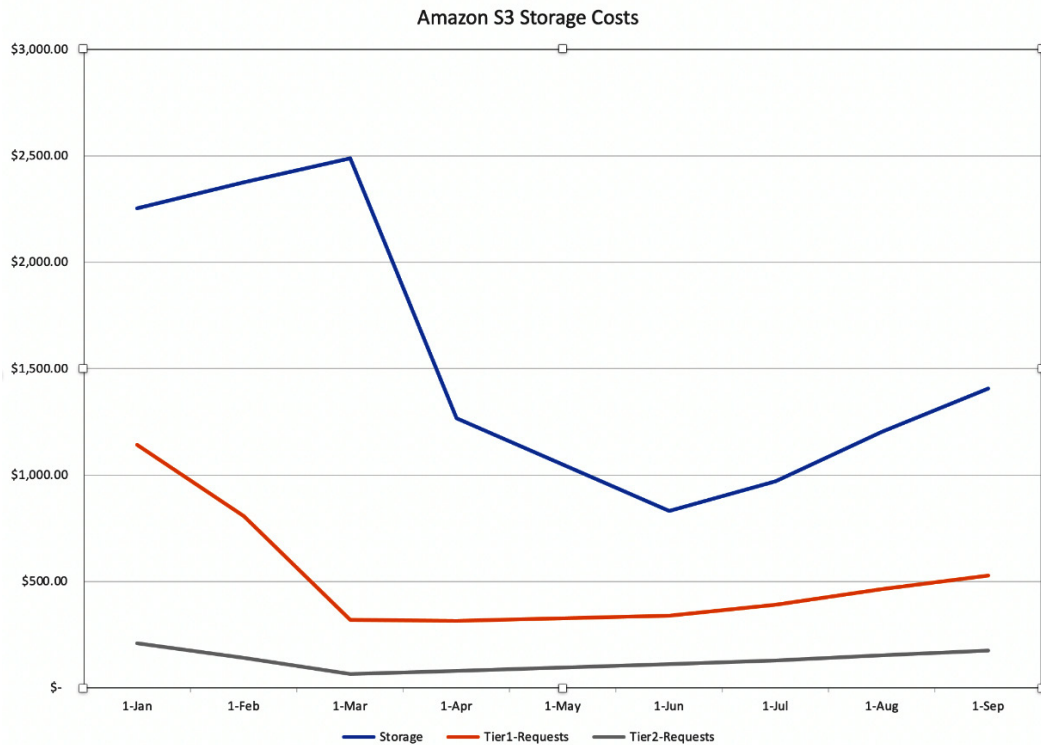


Figure 12.1 – Line chart showing data over a certain period

In this visualization, we can see that our **Tier1-Requests** cost (middle line) significantly decreased from January to March. These costs are for API calls for operations such as `PUT`, `COPY`, `POST`, and `LIST`. Before February, we used to ingest a large number of small files, resulting in millions of `PUT` requests when writing these files to Amazon S3. After changing our transformation pipeline to write out fewer, larger files, this visualization clearly shows how those costs decreased.

In the visualization, we can also see that in March, our storage consumption (top line) significantly decreased. This makes sense as, during March, our fictional company had a project to implement Amazon S3 life cycle rules that deleted older versions of data from S3.

Showing summarized data over a certain period in a visual format makes it much easier to track and understand trends in our data, as well as to spot anomalies.

Data over a geographic area

In our first example, we looked at how we could graph trends over time, but another really useful visualization is to look at trends over a geographic area. There are many uses for this type of visualization, such as the following:

- Understanding the popularity of a certain product in different geographic regions.
- Quickly visualizing hotspots for the spread of an infectious disease (such as flu outbreaks) in different geographic regions.
- Visualizing the population sizes of different cities in different regions.
- Showing differences in temperature in different geographic areas.

These types of charts are often known as *geospatial charts*, although they go by many different names. The chart may also come in different formats, but a common format is to use circles of different sizes on the map, with the size of each circle representing the value of one of the columns in the dataset (the larger the value, the bigger the circle). Circles may also be different colors to represent different rows in the dataset.

For example, the following chart (created with Amazon QuickSight) uses city population data from <https://simplemaps.com/data/world-cities>. In this chart, we have filtered the data to show all cities with a population above 3 million people, and the size of the circle represents the relative population size. In the hands-on section of this chapter, you will use Amazon QuickSight to recreate this chart so that you can interact with the chart (filter for different values and so on):

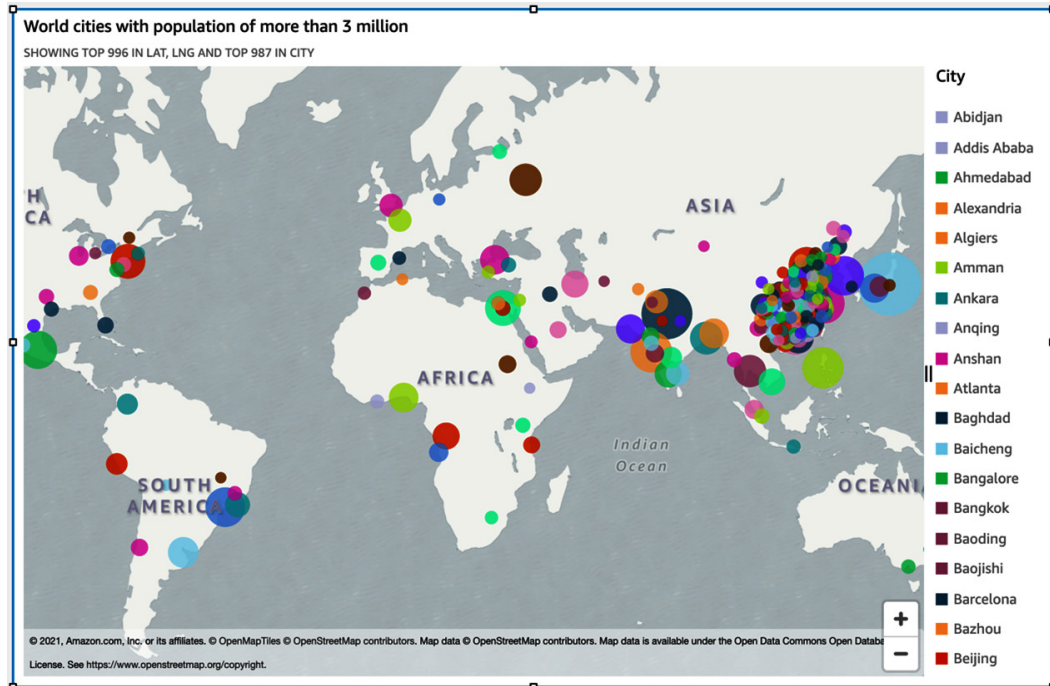


Figure 12.2 – Map chart showing data by geographic region

The preceding diagram of a map chart enables us to quickly understand which parts of the world have the most populated cities, and which parts are less populated. The same type of chart could be used to show the spread of disease, vaccination rates, poverty levels, water quality, or just about any other data that is associated with a specific location.

Heat maps to represent the intersection of data

Another common use of visualization tools is to understand the relationship between different sets of data. Often, we may have a gut feeling that there could be a correlation between two different datasets, but it is only when we explore the data more fully that we can understand those relationships.

As a very simple example, we would probably suspect that sales of ice cream, water, and other cold goods would be more popular in the summer months and that the sales of coffee, hot chocolate, and soup would be more popular in the winter months:

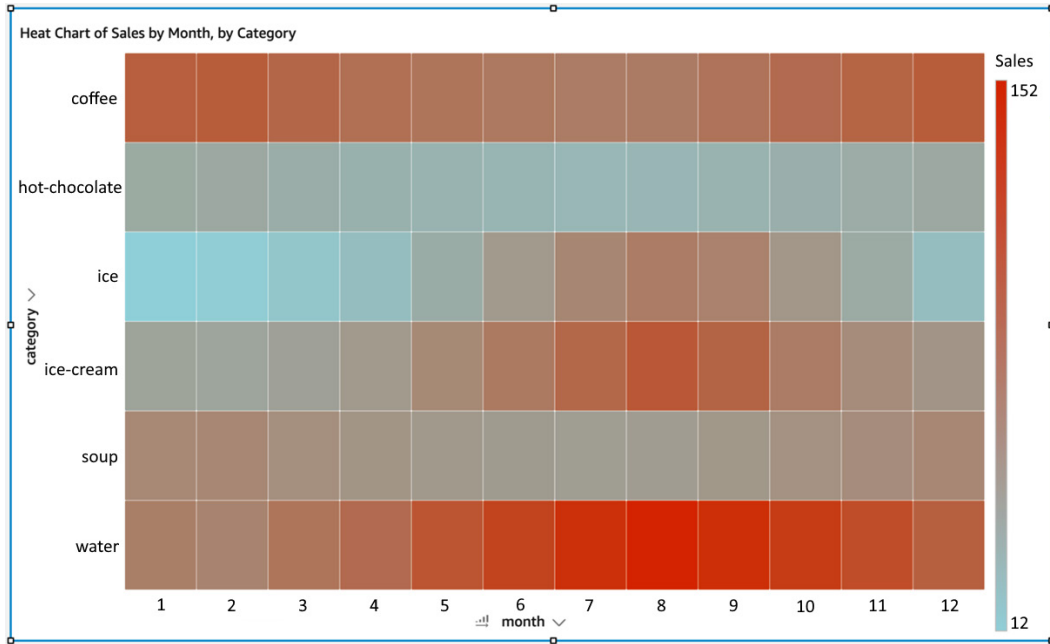


Figure 12.3 – Heat map showing product sales by category and month

The preceding diagram shows a heat map that plots the relationship between sales in different categories, by month. The darker squares illustrate a higher sales value, while the lighter squares represent lower sales values.

As you can see, the sales of both coffee and water are strong throughout the year, but we can see that water has higher sales in the Northern Hemisphere summer (months 6 – 9), while coffee has higher sales in winter (months 11 – 2). Another insight we can gain quickly is that sales of ice are very low in the winter months and only have strong sales for a few summer months of the year (months 6 – 10). What other insights can you gain about sales of hot chocolate, ice cream, and soup by examining the heat map?

While this example may have been a fairly simple one, there are many other relationships between datasets that are not always as obvious, and heat maps can be useful to highlight these relationships visually.

We do not have sufficient space in this chapter to cover all the many varied types of charts that can be used to visually represent data, but as we continue into the other sections of this chapter, we will explore some other common chart types along the way. In the next section, we are going to dive deeper into Amazon QuickSight's core concepts.

Understanding Amazon QuickSight's core concepts

At its core, QuickSight lets us ingest data from a wide variety of sources, perform some filtering or other transformation tasks on the data, and then create dashboards with multiple types of visuals that can be easily shared with others.

The QuickSight service is fully managed by AWS, and there are no upfront costs for using the service. Instead, the service uses a pricing model of a monthly cost per user and offers both Standard and Enterprise Editions. QuickSight also includes a powerful in-memory storage and computation engine to enable the best performance for working with a variety of data sources.

In this section, we'll examine the differences between the standard and enterprise editions of QuickSight and also do a deeper dive into SPICE, the in-memory storage and computation engine.

Standard versus enterprise edition

The Standard Edition of QuickSight is useful for those that are just starting to explore the power of a BI tool and enables users to create visualizations from a variety of sources. However, for larger organizations, the Enterprise Edition of QuickSight provides several additional features that most large organizations would want to make use of.

The following is a subset of some of the additional functionality available in the enterprise edition, but refer to the Amazon QuickSight pricing page for full details on the differences between the versions. The following features are only available in the enterprise edition:

- Integration with **Active Directory (AD)** and the ability to use AD groups for management of QuickSight resources
- The ability to embed dashboards into custom applications
- The ability to email reports to QuickSight users on a schedule
- Fine-grained access control over AWS data sources (such as S3 and Athena)

- Automatic insight generation using ML Insights
- Encryption of data at rest

Another benefit of the Enterprise Edition of QuickSight is that pricing is lower for those users that just need to access and interact with created visualizations (readers), but do not need to author new visuals from scratch (authors).

With the enterprise edition, there is a fixed monthly cost for users that have the author role, while users with the reader role are charged per session. Each session provides access to QuickSight dashboards for a user for up to 30 minutes after they have logged in. During this time, readers can fully interact with the dashboards (filtering data, doing drill-downs into data, and so on). At the time of writing, a session costs \$0.30, and there is a maximum monthly cost of \$5 per reader, no matter how many sessions are used. In comparison, the Standard Edition has a fixed cost (at the time of writing) of \$9 per user and all users have full read and author capabilities. Refer to the QuickSight pricing page (<https://aws.amazon.com/quicksight/pricing/>) for the current pricing for your region, as pricing may change occasionally.

SPICE – the in-memory storage and computation engine for QuickSight

Like many other BI tools, Amazon QuickSight provides a storage engine for storing imported data and performing rapid calculations on that data. In QuickSight, **SPICE** is the acronym that's used to reference this engine, and it stands for **Super-fast, Parallel, In-memory, Calculation Engine**. When you're creating a new dataset in QuickSight, you can select whether to perform direct queries of the dataset, or whether you want to import data into SPICE.

If you choose to query the dataset, then each time the visualization is accessed, QuickSight will make a connection to the data source (such as an Amazon RDS MySQL database) and query the data. This ensures that the dashboard always reflects the latest data. However, there is some latency in making the connection to the data source and retrieving data.

Alternatively, you can choose to import data into the SPICE engine. That way, when the visualization is accessed, QuickSight can read the data directly from SPICE, and this can significantly improve performance.

You also have the option of scheduling a refresh of the data in SPICE so that QuickSight will regularly connect to the data source and retrieve the latest data to store in SPICE. With both the standard and enterprise editions of QuickSight, you can schedule the refresh to be done daily, weekly, or monthly. With the enterprise edition of QuickSight, however, you gain the additional option of performing incremental refreshes, and the ability to refresh data as often as every 15 minutes. You can also use an API call to trigger the refresh of SPICE data, enabling you to build an event-driven strategy for refreshing SPICE data. For more information, see the AWS blog post titled *Event-driven refresh of SPICE datasets in Amazon QuickSight* at <https://aws.amazon.com/blogs/big-data/event-driven-refresh-of-spice-datasets-in-amazon-quicksight/>.

Note

There is a 2-minute timeout for generating visuals in QuickSight. Therefore, if your direct query takes 2 minutes or longer to perform the query and generate the visualization, a timeout will occur. In these cases, you either need to improve the performance of the query (filtering data, only selecting specific columns, and so on) or you should import the data into SPICE.

If you're using a data source (such as Amazon Athena or Amazon Redshift Spectrum) that charges for each query, importing the data into SPICE can help reduce costs. Storing the data in SPICE means you only pay for the query when the data is initially loaded, as well as for when the data is refreshed. With a direct query, you would pay for the query each time the visualization is accessed.

Managing SPICE capacity

Your account is granted 10 GB of SPICE storage for every paid user that has the author role (this would be every user in the Standard Edition, and users with the Author role in the enterprise edition). SPICE storage is shared by all QuickSight users in an account and is on a per-region basis.

For example, if you have QuickSight enterprise edition and you have 10 users with the Author role and 100 users with the Reader role, all in the Northern Virginia (us-east-1) region, then your QuickSight account in us-east-1 would have 100 GB of SPICE storage available.

If additional SPICE storage is needed, you can purchase additional SPICE capacity. For example, if you needed 130 GB of total SPICE storage for the datasets you wanted to import, you could purchase an additional 30 GB of capacity each month. At the time of writing, additional SPICE capacity for the Enterprise Edition is charged at \$0.38 per GB.

There are also limits on the size of a single dataset in SPICE. At the time of writing, datasets are limited to a maximum of 500 million rows, or 500 GB, for QuickSight enterprise edition. For the Standard Edition, the limit is 25 million rows or 25 GB of data. There are also other limits for each dataset (such as the number of columns and the length of column names), so ensure you refer to the latest QuickSight documentation for updated information on these limits.

Now that we have reviewed the core Amazon QuickSight concepts, let's move on and review QuickSight's functionality for importing and preparing data.

Ingesting and preparing data from a variety of sources

Amazon QuickSight can use other AWS services as a source, as well as on-premises databases, imported files, and even some **Software as a Service (SaaS)** applications.

For example, you can easily connect to Oracle, Microsoft SQL Server, Postgres, and MySQL databases, either running as part of the Amazon RDS managed database service or as instances running on Amazon EC2, or in your own data centers. You can also connect to data warehouse systems such as Amazon Redshift, Snowflake, and Teradata. Other AWS services are also supported as data sources, including Amazon S3, Amazon Athena, Amazon Elasticsearch Service, Amazon Aurora, and AWS IoT Analytics.

In addition to these traditional data sources, QuickSight can also connect to various SaaS offerings, including ServiceNow, Jira, Adobe Analytics, Salesforce, GitHub, and Twitter.

Data stored in files, such as a Microsoft Excel Spreadsheet (XLSX files), JSON documents, and CSV files, can also be imported into QuickSight. These files can be directly uploaded through the QuickSight console, or they can be imported from Amazon S3.

The rich variety of potential data sources for QuickSight is shown in the following screenshot:

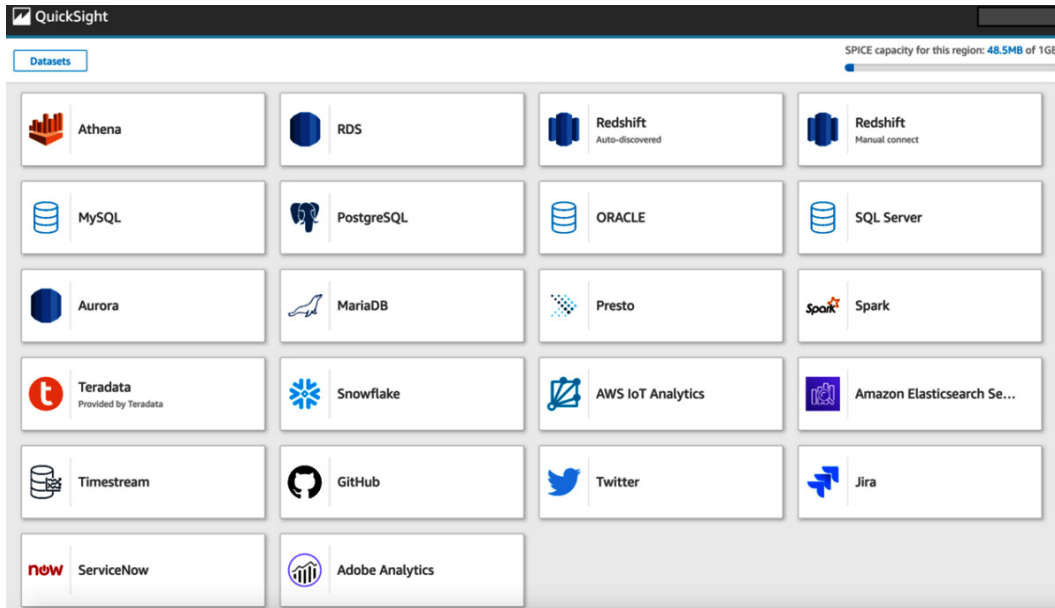


Figure 12.4 – Data sources that can be imported into Amazon QuickSight

For data sources not directly supported, you can use other ingestion methods (such as those discussed in *Chapter 6, Ingesting Batch and Streaming Data*) to ingest data into your S3-based data lake. You can then create visualizations of that data by using the Amazon Athena data source integration to enable QuickSight to query the data.

Preparing datasets in QuickSight versus performing ETL outside of QuickSight

QuickSight includes functionality for performing data transformations on imported data. For example, you can do the following:

- Join two different datasets.
- Exclude specific fields.
- Filter data.
- Change the data type or name of a field.
- Create a new calculated field.

All of these data preparation tasks can be done using a simple visual interface.

If you select to join two different datasets, then you need to import the data into SPICE. However, if you're just working with a single data source, the transformations you specify will be applied when the data is read from the data source.

Ultimately, you need to decide on whether you should perform data transformation and joins in QuickSight, or whether you should perform those transformations outside of QuickSight. For example, you could join two datasets, drop unneeded columns, change the data types and column names, and create new calculated fields using tools such as AWS Glue, AWS Glue DataBrew, or AWS Glue Studio.

There are several factors to consider when making this decision, including the following:

- If this dataset may be used outside of QuickSight, such as for queries using Amazon Athena, then it makes sense to perform the ETL with other tools before using the dataset in QuickSight.
- If the required transformations are relatively simple and the resulting dataset will only be used in QuickSight, then you may choose to perform the transformation using QuickSight. This could include transforms such as adding additional calculated fields, changing the names or data types of a few columns, dropping a few columns, and so on.

The decision about where to perform data transformations can be complex, and it may not be an easy decision. However, an important factor to take into account is the controls that may be in place for formal data pipelines, versus those for more informal transformations (such as those performed by data analysts using tools such as Amazon QuickSight).

If you have strong governance controls around your formal data engineering pipelines (such as code reviews and change control), then you may choose to ensure that all the transformations are done within formal processes. However, you need to balance this against ensuring that you don't tie up your end user teams in formal processes that slow the business down.

Often, you need to balance the two sides – ensuring that your business teams have the flexibility to perform minor transformations using tools such as QuickSight, while also ensuring that new datasets or visualizations that business users may use to make important business decisions have the correct governance controls around them.

It is not always easy to find this balance, and there are no specific rules that apply universally when making this decision. Therefore, much thought needs to be given to this decision to find the right balance between enabling the business to make decisions quickly, without being constrained by overly formal processes for even minor data transformations.

The business ultimately needs to take the time required to put in place governance and controls that communicate the types of ad hoc data transformations that data analysts and others can perform. These policies should also make it clear as to when transformations need to be performed within formal processes by data engineering teams.

Creating and sharing visuals with QuickSight analyses and dashboards

Once a dataset has been imported (and optionally transformed), you can create visualizations of this data using **QuickSight analyses**. This is the tool that is used by QuickSight authors to create new dashboards, with these dashboards containing one or more visualizations that can be shared with others in the business.

When you create a new analysis/dashboard, you choose one or more datasets to include in the analysis (up to a maximum of 50 datasets per dashboard). Each analysis consists of one or more sheets (or tabs, much like browser tabs) that display a group of visualizations. You can have up to 20 sheets (tabs) per dashboard, and each sheet can have up to 30 visualizations.

Once you have created an analysis (consisting of multiple visuals, optionally across multiple sheets), you can choose to publish the analysis as a dashboard. When you're publishing a dashboard, you can select various parameters related to how readers can interact with the dashboard, including the following:

- If they can apply their ad hoc filters to the data in the dashboard
- If they can download data in the dashboard as a CSV file
- If they can perform drill-down and drill-up actions (when supported in a dashboard)
- If they can sort the data

Once the dashboard has been published, you can select who to share the dashboard with. You can either share the dashboard with everyone in the account (providing them with read access to the dashboard) or you can select specific users and groups to share with.

By default, when you create a new analysis, the analysis contains a single sheet, with a single empty visualization that is set to a type of **AutoGraph**:

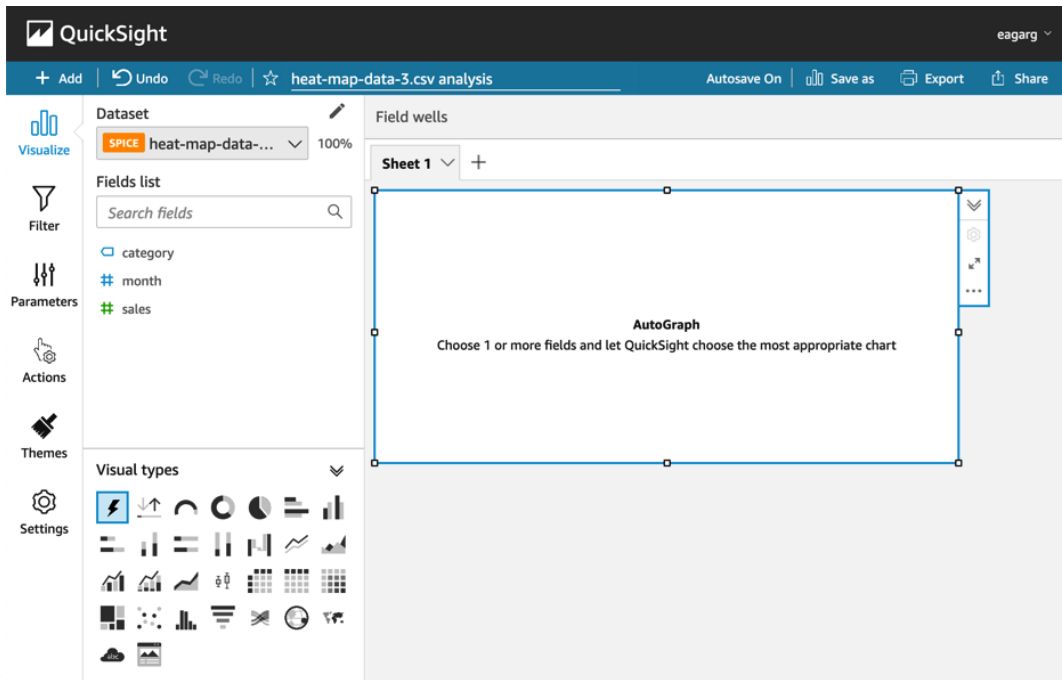


Figure 12.5 – Amazon QuickSight – New Analysis screen

QuickSight supports many different types of visualizations (as can be seen in the **Visual types** section of the preceding screenshot). Let's dive deeper into some of these visual types.

Visual types in Amazon QuickSight

In this section, we will discuss several data visualization types supported by Amazon QuickSight. There are many different types of visualizations that are supported, and we will not cover all of them here, so check out the *Amazon QuickSight documentation* (<https://docs.aws.amazon.com/quicksight/latest/user/working-with-visual-types.html>) for a full list of supported visualizations.

AutoGraph for automatic graphing

While this is not an actual type of visual, you can select AutoGraph as a visual type to let QuickSight automatically choose the visual type for you. Based on the number of fields you select, and the data type of each field that is selected, QuickSight automatically uses the most appropriate visual type for your data. This is often a good way to start exploring your data if you're unsure of the specific type of graph you want to use.

Line, geospatial, and heat maps

Earlier in this chapter, we discussed three common types of visualizations:

- **Line charts:** Displays data as a series of data points and is often used to plot data over a certain period
- **Geospatial charts:** Displays data points overlaid on a map, combining geospatial data with other data
- **Heat maps:** Displays data in a chart with values represented by darker or lighter colors

All three of these types of charts (and variations of these charts) are supported by Amazon QuickSight and can be used to create rich visualizations from many different data sources.

Bar charts

Bar charts are a common visualization type, and QuickSight supports multiple types of bar charts. For example, you can have a simple bar chart showing a single value for a dimension (such as sales per region) or a multi-measure bar chart that shows multiple measures for a dimension (such as sales goal and achieved sales per region).

There are also additional bar chart types that are supported, such as stacked bar charts and clustered bar charts. Bar charts can be displayed horizontally or vertically.

Custom visual types

QuickSight lets you include several **custom visuals** within a dashboard, including the following:

- Custom images (such as a company or product logo)
- Custom videos
- An online form
- An embedded web page

Note that when you're embedding custom content in an analysis/dashboard, you need to specify the HTTP URL of the resource. Also, while QuickSight does include functionality for emailing dashboards to users, embedded custom visual types (pictures, videos, forms, and web pages) will not be displayed in the email copy of a dashboard.

There are also other limitations to using embedded content. For example, the web content needs to support opening the content in an iFrame; otherwise, the content may not appear in QuickSight. When you're looking to embed content into a QuickSight analysis/dashboard, you should look for content that has an embeddable URL (which is often available when you choose to share content).

Key Performance Indicators

A **Key Performance Indicator (KPI)** is often used to show progress against a specific goal. For example, you may have a goal of achieving a specific amount of revenue in a quarter.

A KPI visual could display the current revenue as a percentage of the target revenue in a visual. A dashboard showing this KPI (or multiple KPIs) can help management keep track of how the business is performing based on several key metrics.

In QuickSight, a KPI displays a comparison of two values and includes a progress bar indicating the percentage difference of the values:

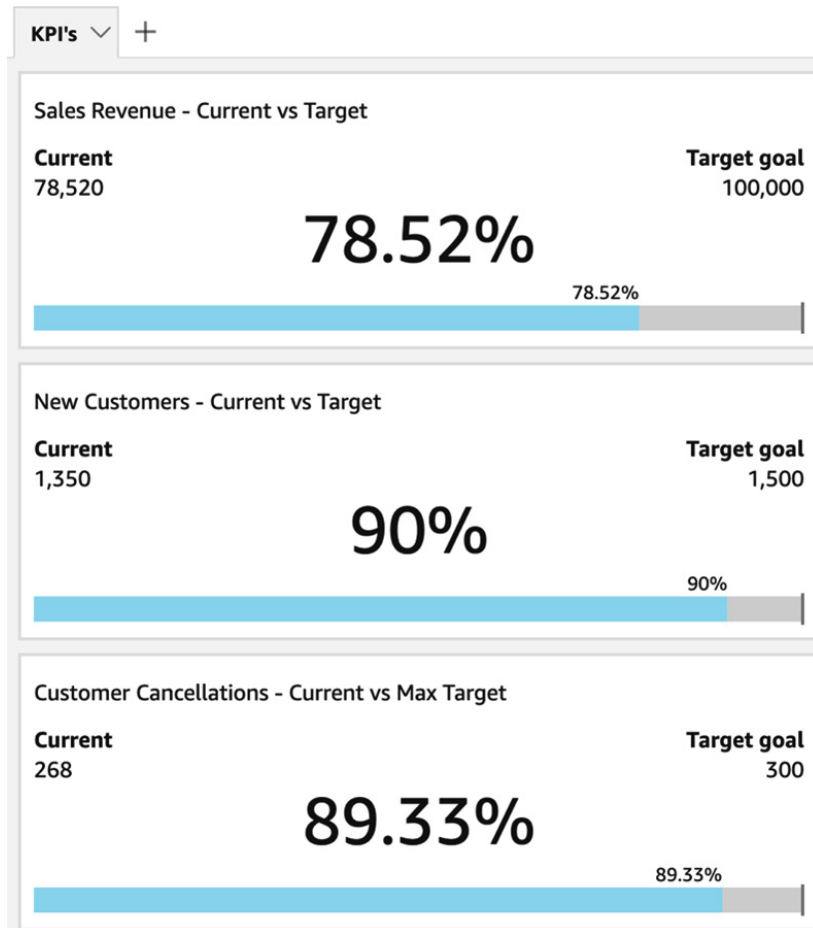


Figure 12.6 – Dashboard with KPI visuals

In the preceding screenshot, a sales manager can quickly view how their organization is performing against several key metrics. This chart shows that revenue is nearly at 80% of the target, new customers are at 90% of the target, and that they are within 11% of the target maximum customer cancellations for that period.

Tables as visuals

There may be use cases where you want to display the raw data of a table on a dashboard, without converting the data into a specific visual.

QuickSight supports displaying tables directly within an analysis/dashboard and supports up to 200 columns in the visual. However, directly displaying raw table data should ideally only be done with small tables, where you display just a limited amount of raw data.

Other visual types

There are many other types of charts that are supported in QuickSight, and new types are added over time. These include the following common chart types:

- Pie charts
- Box plots
- Gauge charts
- Histograms
- Pivot tables
- Sankey diagrams
- Tree maps
- Word clouds

Not all the supported visual types have been listed in this chapter, so to review the full list of supported visual types, see the Amazon QuickSight documentation titled *Working with Visual Types in Amazon QuickSight*: <https://docs.aws.amazon.com/quicksight/latest/user/working-with-visual-types.html>.

As we have discussed in this section, QuickSight lets us create many different types of visuals and publish and then share those visuals as dashboards. However, QuickSight also includes advanced functionality that can automatically reveal new insights in your data and lets you embed dashboards into custom applications, as we will see in the next section.

Understanding QuickSight's advanced features – ML Insights and embedded dashboards

The enterprise edition of Amazon QuickSight includes two advanced features that can help you draw out additional insights from your data, and that can enable you to widely share your data by embedding dashboards into applications.

Amazon QuickSight ML Insights

QuickSight ML Insights uses the power of machine learning algorithms to automatically uncover insights and trends, forecast future data points, and identify anomalies in your data.

All of these ML Insights can be easily added to an analysis/dashboard without the author needing to have any machine learning experience or any real understanding of the underlying ML algorithms. However, for those who are interested in the underlying ML algorithms used by QuickSight, Amazon provides comprehensive documentation on this topic. Review the Amazon QuickSight documentation titled *Understanding the ML Algorithm Used by Amazon QuickSight* for more information: <https://docs.aws.amazon.com/quicksight/latest/user/concept-of-ml-algorithms.html>.

However, to make use of ML Insights, there are specific requirements for your data, such as having at least one metric and one category dimension. For ML forecasting, the more historical data you have the better. For example, if you want to forecast based on daily data, you need at least 38 daily data points, or to forecast on quarters, you need at least 35 quarterly data points. The full details on the data requirements are documented in the Amazon QuickSight documentation titled *Dataset Requirements for Using ML Insights with Amazon QuickSight*: <https://docs.aws.amazon.com/quicksight/latest/user/ml-data-set-requirements.html>.

Let's examine some of the different types of ML Insights in more detail.

Amazon QuickSight autonarratives

Autonarratives provide natural language insights into your data, providing you with an easy-to-read summary of what is displayed in a visual. Effectively, autonarratives enables you to provide a plainly stated summary of your data, as the following autonarrative examples show:

- Year-to-date revenue *decreased* by 4.6% from \$906,123 to \$864,441 compared to the same period last year. We are at 89.3% achievement for the YTD goal and 77.9% achievement for the annual goal.
- Daily revenue for *Accessories / Cell Phone Covers* on September 3, 2021 was higher than expected at \$3,461.21.

You can add a variety of autonarratives to an analysis, such as bottom-ranked items, growth rate, anomaly detection, top movers, and many others. For the full list of available autonarratives, see the Amazon QuickSight documentation titled *Insights that include autonarratives*: <https://docs.aws.amazon.com/quicksight/latest/user/auto-narratives.html>.

ML-powered anomaly detection

Amazon QuickSight can perform **anomaly detection** across millions of metrics contained in your data, identify non-obvious trends, and highlight outliers in the data. These types of insights are difficult to draw out of data without using the power of modern ML algorithms.

You can add an autonarrative widget to an analysis and specify the type as being anomalies. Then, you can configure several settings related to how QuickSight detects outliers in the data and can set a schedule for when outliers are calculated (ranging from once an hour to once a month). You can also configure QuickSight to analyze the top items that contributed to the anomaly.

Once an anomaly has been detected, you can choose to **explore the anomalies** on the insight. This opens a screen where you can change various settings related to anomaly detection, enabling you to explore different types of anomalies in the dataset.

ML-powered forecasting

Amazon QuickSight can use the power of ML algorithms to provide reliable forecasts against your data. When you create a visual that uses a date field and contains up to three metrics, you can select an option in the widget to add a forecast of future values.

QuickSight will automatically analyze historical data using an ML model and graph out future predicted values for each metric. You can also configure the forecast properties by setting items such as forecast length (how many future periods to forecast and how much historical data to analyze).

The machine learning model that's used by QuickSight for forecasting automatically excludes data that it identifies as outliers and automatically fills in any missing values. For example, if you had a short spike in sales due to a promotion, QuickSight could exclude that spike when calculating the forecast. Or, if there were a few days where historical data was missing, QuickSight could automatically determine likely values for the missing period.

It is important to remember that the QuickSight ML Insight features (including autonarratives, anomaly detection, and forecasting) are available in the Enterprise Edition of QuickSight, and will not be available if you only have a Standard Edition subscription.

In this section, we looked at how QuickSight enables you to draw out powerful new insights from your data. In the next section, we will look at another popular feature of the enterprise edition of QuickSight, a feature that enables you to easily distribute your published dashboards more widely.

Amazon QuickSight embedded dashboards

For use cases where you don't want your users to have to log into QuickSight via the AWS Management Console or QuickSight portal, you can embed QuickSight directly into your applications or website.

You can embed either the full console experience (including authoring tools for creating new analyses and managing datasets) or embed published dashboards only. Embedded dashboards have the full interactive capabilities that they do in the console, which means that users can filter and sort data, and even drill down into data (so long as the author enabled those levels of interactivity when they published the dashboard).

Embedding for registered QuickSight users

QuickSight supports several authentication methods, including AD SAML 2.0, as well as SSO using AWS Single Sign-on (or other identity providers such as Okta, Auth0, and PingOne).

As such, your users can authenticate with your existing website or HTML-based application using one of the supported authentication methods and, using that identity, map to an existing QuickSight user. If that user has not accessed QuickSight before, a new QuickSight user will be created for the user.

You can elect to either embed the full console experience or only embed dashboards. Users will be able to open any dashboards that their QuickSight user has been given access to.

With the QuickSight embedding experience, you can optionally customize the display theme using your branding. This enables the embedded QuickSight objects to appear as a direct part of your application, rather than looking like an embedded external application. However, even when you have a customized theme, the embedded QuickSight application does display a **Powered by QuickSight** label.

Embedding for unauthenticated users

For use cases where your users do not authenticate with your website or application, you still have the option of embedding QuickSight dashboards for anonymous user access.

To enable anonymous access, you need to purchase reader session capacity pricing. This offers a set number of QuickSight sessions per month, or per year (depending on your plan), and these sessions can be consumed by anonymous users. The bonus of purchasing an annual plan for QuickSight sessions is that the **Powered by QuickSight** label can be removed from embedded resources.

An example use case for this functionality is for a local government health department that wants to share the latest information on a virus outbreak with their community. The health department could embed an Amazon QuickSight dashboard into their website that is linked to the latest data on the spread of the virus.

Users accessing the website could interact with the dashboard, filtering data for their specific location, sorting data, or even downloading a CSV version of the data for their additional analysis. These users would not need to log into the health department website to access the dashboard, and the health department could use an annual plan for reader session capacity. For more information on pricing for reader session plans, see the Amazon QuickSight pricing page: <https://aws.amazon.com/quicksight/pricing/>.

Having learned more about QuickSight's core functionality, let's get hands on by creating a QuickSight visualization and publishing a dashboard.

Hands-on – creating a simple QuickSight visualization

Earlier in this chapter, we discussed how data can be represented over a geographic area. We used the example of data containing information on the population of world cities, and how we could use that to easily visualize how large cities are geographically distributed. The example visual in *Figure 12.2* showed cities with a population of over 3 million people, displayed on top of a map of the world.

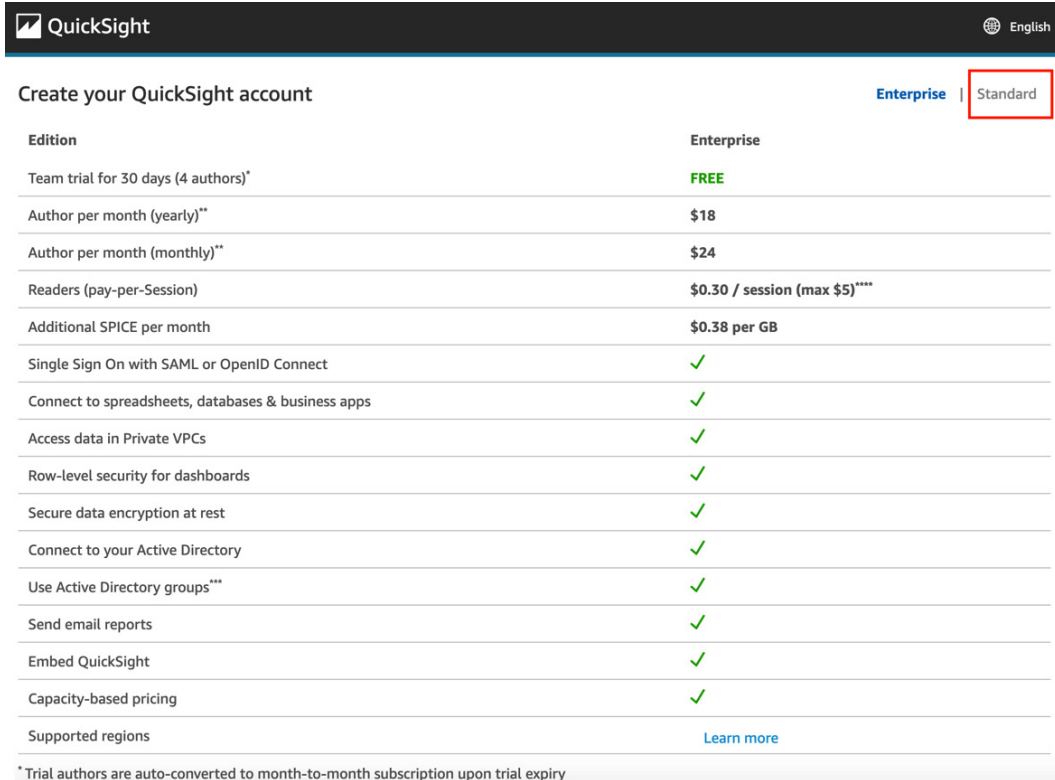
For the hands-on section of this chapter, we are going to recreate that visual using Amazon QuickSight.

Setting up a new QuickSight account and loading a dataset

Before we start creating a new dashboard, we need to download a sample dataset of world city populations. We are going to use the basic dataset available from <https://simplemaps.com/>, which is freely distributed under the *Creative Commons Attribution 4.0* license (<https://creativecommons.org/licenses/by/4.0/>):

1. Use the following link to download the basic dataset from simplemaps.com: <https://simplemaps.com/data/world-cities>. If the file downloaded is a ZIP file, make sure to extract the actual city data CSV file.
2. Log into the AWS Management Console and use the top search bar to search for, and open, the **QuickSight** service.

3. If you have not used QuickSight before in this account, you will be prompted with a **Sign up for QuickSight button**. Click the button to start the signup process.
4. The default page opens to QuickSight enterprise edition. For this exercise, only the Standard Edition is needed, so click on **Standard** at the top right of the screen:



The screenshot shows the QuickSight account creation interface. At the top, the QuickSight logo is on the left and 'English' is on the right. Below the header, the title 'Create your QuickSight account' is on the left, and 'Enterprise | Standard' is on the right, with 'Standard' highlighted by a red box. A table compares features between the Standard and Enterprise editions. The Standard edition is marked as 'FREE' for the trial period. A footnote at the bottom states: '* Trial authors are auto-converted to month-to-month subscription upon trial expiry'.

Edition	Enterprise
Team trial for 30 days (4 authors)*	FREE
Author per month (yearly)**	\$18
Author per month (monthly)**	\$24
Readers (pay-per-Session)	\$0.30 / session (max \$5)****
Additional SPICE per month	\$0.38 per GB
Single Sign On with SAML or OpenID Connect	✓
Connect to spreadsheets, databases & business apps	✓
Access data in Private VPCs	✓
Row-level security for dashboards	✓
Secure data encryption at rest	✓
Connect to your Active Directory	✓
Use Active Directory groups****	✓
Send email reports	✓
Embed QuickSight	✓
Capacity-based pricing	✓
Supported regions	Learn more

* Trial authors are auto-converted to month-to-month subscription upon trial expiry

Figure 12.7 – Setting up a new QuickSight account

- For **Authentication method**, select **Use IAM federated identities only**, and then select your preferred **AWS region**. Under **Account info**, provide a unique name for your **QuickSight account** (such as `data-engineering-<initials>`) and provide a **Notification email address** that can be used to send QuickSight notifications to you. Leave all other settings as-is and click **Finish**:

Create your QuickSight account

Standard

[Back](#)

Authentication method

- Use IAM federated identities & QuickSight-managed users
Authenticate with single sign-on (SAML or OpenID Connect), AWS IAM credentials, or QuickSight credentials
- Use IAM federated identities only
Authenticate with single sign-on (SAML or OpenID Connect) or AWS IAM credentials

QuickSight region

Select a region

US East (Ohio) ⌵

Account info

QuickSight account name

You will need this for you and others to sign in

data-

Notification email address

For QuickSight to send important notifications

gare l.com

Figure 12.8 – Configuring a new QuickSight account

- After a while, you should receive a message confirming that you have signed up for Amazon QuickSight. Click on the **Go to Amazon QuickSight link**, and then click through the welcome screens, which provide an overview of Amazon QuickSight's functionality.
- From the left-hand side menu, click on **Datasets** to go to the dataset management screen. On this screen, you will see several pre-loaded sample datasets:

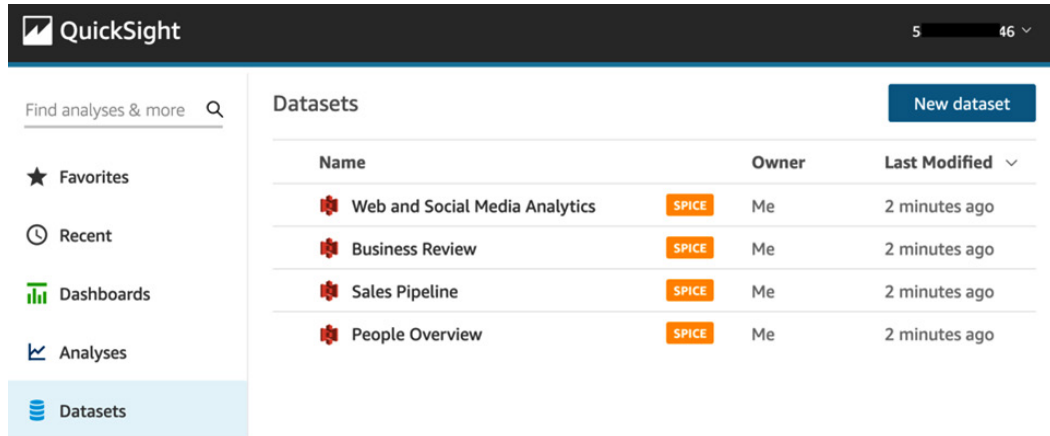


Figure 12.9 – Pre-loaded datasets for a new QuickSight account

8. Click on **New dataset** to create a new dataset. On the new dataset screen, click on **Upload a file**.
9. When you're prompted to provide the file to upload, navigate to where you downloaded the World Cities data from simplymaps.com (in *Step 1* of this exercise) and upload the `worldcities.csv` CSV file.
10. Once the file has been uploaded, you will be presented with a popup to confirm the file upload settings. Click on **Next**.
11. On the next screen, click on **Visualize**. This will open a new analyses screen where you can create your analysis/dashboard based on the World Cities dataset.

Now that we have subscribed to QuickSight, downloaded our World Cities dataset, and uploaded the dataset into QuickSight, we are ready to create our first visual.

Creating a new analysis

We are now on the analysis authoring page for QuickSight. Using this interface, we can build out new analyses consisting of multiple visualizations and, optionally, containing multiple sheets (tabs). Then, we can publish our analysis as a dashboard that can be consumed by QuickSight readers.

The following screenshot shows the analysis workspace after importing our `worldcities.csv` dataset:

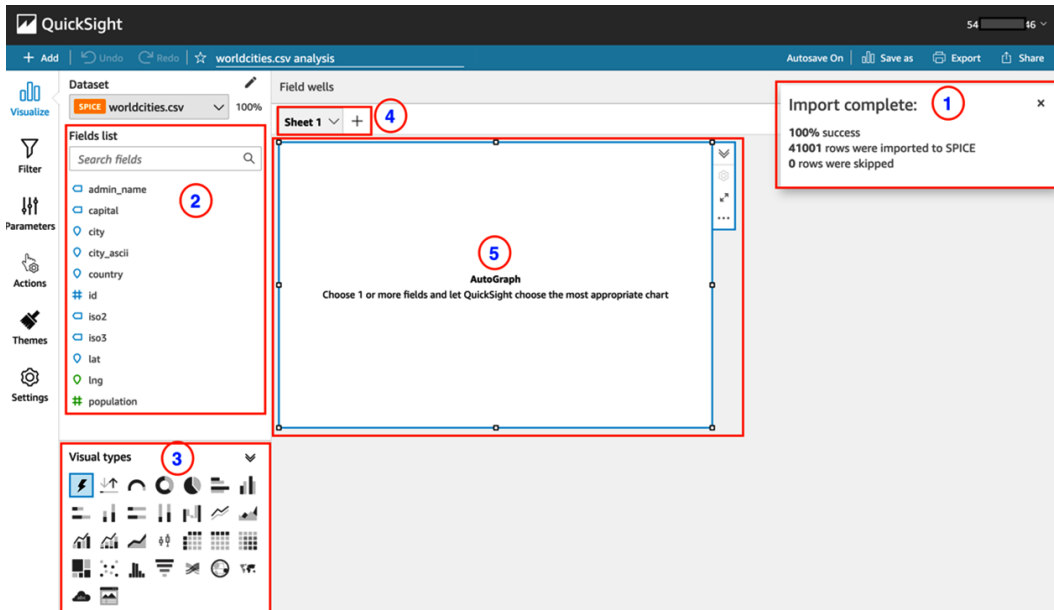


Figure 12.10 – The different parts of a new QuickSight analysis

In this screenshot, we can see the following components of the analysis workspace. Note that the numbers in this section correspond to the component number shown in the preceding screenshot:

1. A popup message indicating that the dataset import is complete. This shows us that approximately 41,000 rows were ingested into the SPICE storage engine. You can click on the **X** button to close the popup.
2. A list of fields in our selected dataset (`worldcities.csv`).
3. A list of different types of charts that we can use in our visuals (bar, pie, heat map, and so on).
4. The sheet bar, which shows us our current sheet (**Sheet 1**). Clicking the + sign would enable us to create additional sheets (much like tabs in a browser). We can also rename sheets.

5. The visual display area. Once we select a chart type and add some fields to the visual, the chart will be displayed here. Notice that the size of the visual area can be dragged to be larger or smaller, and we can click on **+ Add** in the top menu bar if we want to add additional visuals to this sheet.

To create our map of the world showing cities with populations greater than 3 million people, perform the following steps:

1. In the **Visual types** box, find and select the **Points on map** visual type.
2. From **Fields list**, drag **lat** into the **Geospatial** field well (at the top of the visual-designed workspace), and then drag **lng** into the same **Geospatial** field well. Make sure that you drag **lng** either above or below **lat**; otherwise, you will end up replacing the existing **lat** field.
3. Drag **population** into the **Size** field well and drag **city** into the **Color** field well.

Your visual designer should look as follows at this point:

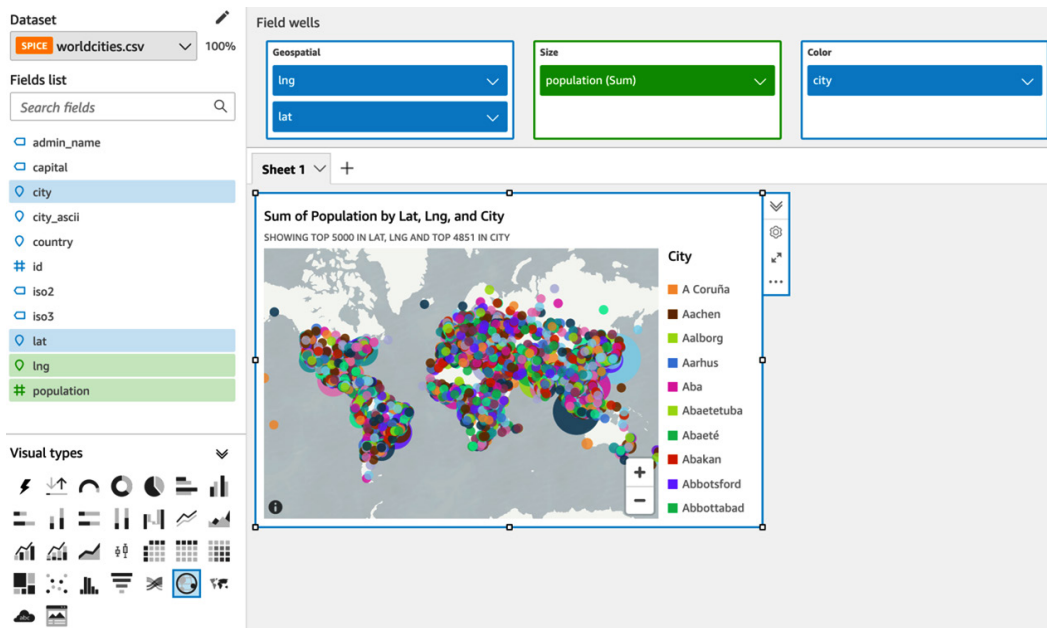


Figure 12.11 – Creating a new Points on Map visual

At this point, our visual is displaying population data for all 41,000 cities in the dataset. However, for our use case, we only want to display data for cities that have a population of above 3 million people. Perform the following steps to filter the data to just cities with a population above a certain size.

- From the left-hand side QuickSight menu, click on **Filter**, and then click **Create one...** (as shown in the following screenshot):

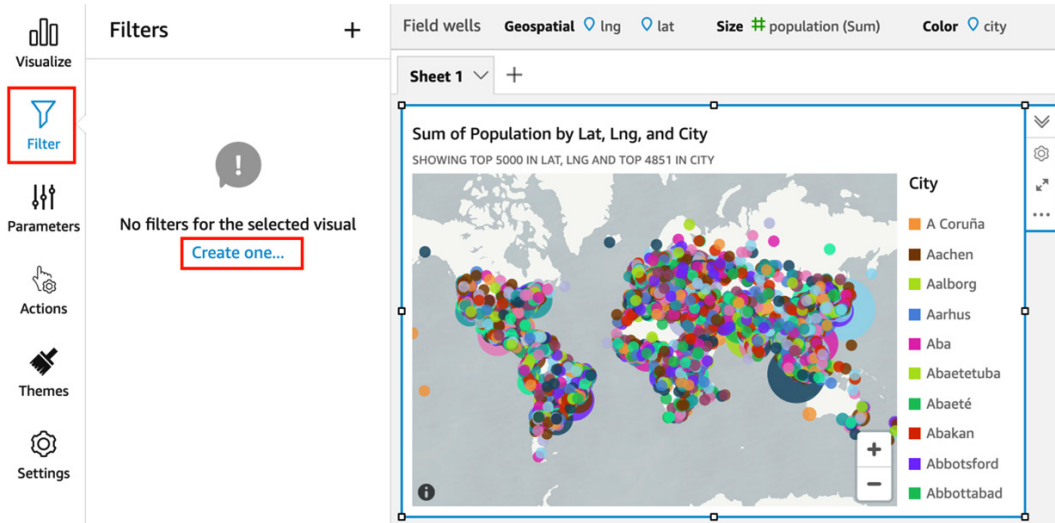


Figure 12.12 – Configuring a filter for a visual

- In the popup that shows the list of fields, click on the **population** field. This displays a filters list with **population** showing as the only filter.
- From the filters list, click on **population**. Change the **Equals** dropdown to **Greater than or equal to** and enter a value of 3000000, as shown in the following screenshot. Then, click on **Apply**:

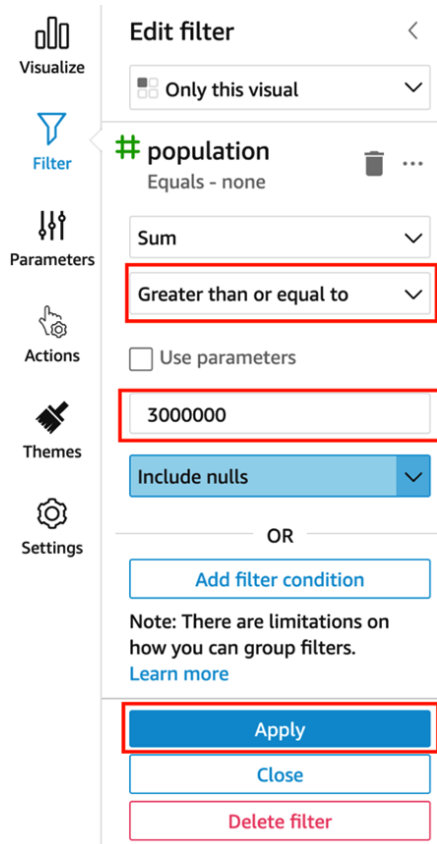


Figure 12.13 – Editing the filter for a visual

Our visual now displays only those cities that have a population of 3 million people or more. Note how you can position the mouse over a city to get a popup of the city's name, along with its latitude, longitude, and population details.

You can also modify the following aspects of the visual:

- Drag the corners of the visual to increase the size of the visual.
- Experiment with the visual by changing the filter on population size (for example, change the filter to 5 million people).
- Zoom in and out on the map to size the map to display just the parts of the map you want to show.
- Double-click on the title of the visual to change the title.
- Click the down arrow next to the title of the sheet (by default, **Sheet 1**) and rename the sheet (for example, changing the name to **City Populations**).

The completed visual now looks as follows:

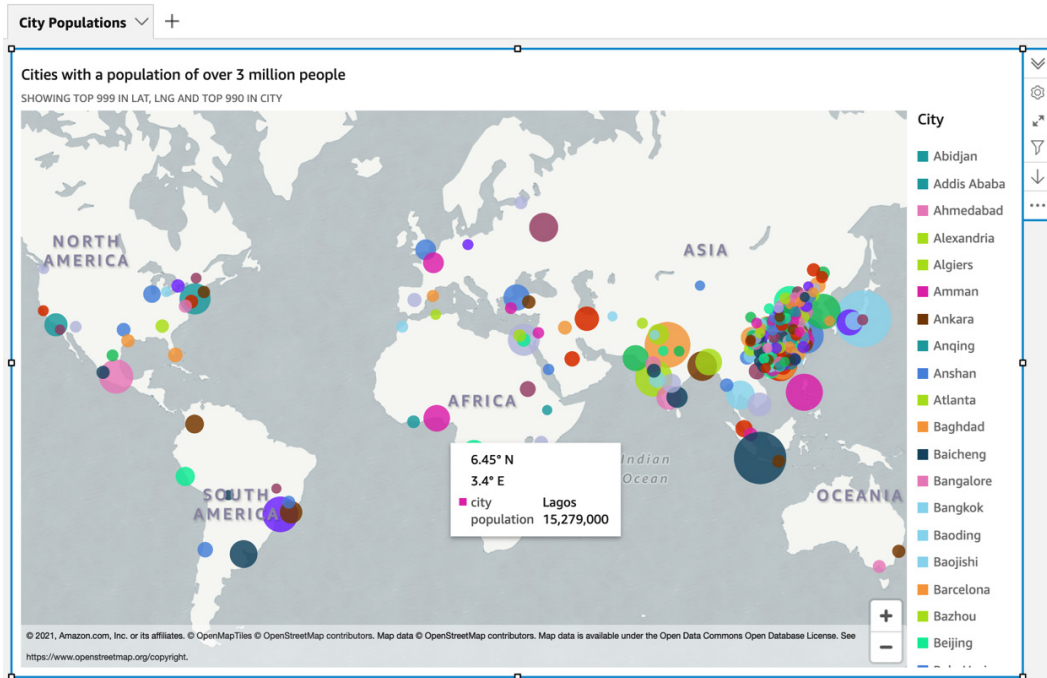


Figure 12.14 – A completed visual showing cities with a population of over 3 million people. We could now share this analysis as a dashboard, making the visual available to a set of QuickSight users that we select.

In the hands-on section of this chapter, you signed up for a new QuickSight account and imported a new file-based dataset that contained information on world cities. This included geospatial data (latitude and longitude), as well as the size of the population of the city. Then, you created a new visual based on this data, filtering the data to only show cities with a population of 3 million or more people.

Important – Avoiding Future QuickSight Subscription Costs

If you do not intend to use QuickSight after the initial 30-day subscription, ensure that you unsubscribe from QuickSight to avoid future subscription charges. For more information, see the AWS documentation titled *Canceling your Amazon QuickSight subscription and closing the account* (<https://docs.aws.amazon.com/quicksight/latest/user/closing-account.html>).

Summary

In this chapter, you learned more about the Amazon QuickSight service, a **BI** tool that is used to create and share rich visualizations of data.

We discussed the power of visually representing data, and then explored core Amazon QuickSight concepts. We looked at how various data sources can be used with QuickSight, how data can optionally be imported into the SPICE storage engine, and how you can perform some data preparation tasks using QuickSight.

We then did a deeper dive into the concepts of analyses (where new visuals are authored) and dashboards (published analyses that can be shared with data consumers). As part of this, we also examined some of the common types of visualizations available in QuickSight.

We then looked at some of the advanced features available in QuickSight, including ML Insights (which uses machine learning to detect outliers in data and forecast future data trends), as well as embedded dashboards (which enable you to embed either the full QuickSight console or dashboards directly into your websites and applications).

We wrapped up this chapter with a hands-on section that took you through the steps of configuring QuickSight within your AWS account and creating a new visualization.

In the next chapter, we will do a deeper dive into some of the many AWS machine learning and artificial intelligence services that are available. We will also review how these services can be used to draw new insights and context out of existing structured and unstructured datasets.

13

Enabling Artificial Intelligence and Machine Learning

For a long time, organizations could only dream of the competitive advantage they would get if they could accurately forecast demand for their products, personalize recommendations for their customers, and automate complex tasks.

And yet, advancements in **machine learning (ML)** over the past decade or so have made many of these things, and much more, a reality.

ML describes the process of training computers in a way that mimics how humans learn to perform several tasks. ML uses a variety of advanced algorithms and, in most cases, large amounts of data to develop and train an ML model. This model can then be used to examine new data and automatically draw insights from that data.

ML offers a wide range of interesting use cases that are expected to have a growing impact on many different aspects of life. For example, scientists are using ML to analyze a patient's retina scan to identify early signs of Alzheimer's disease. It is also the power of ML, and specifically computer vision, that is enabling advances in self-driving vehicles so that a car can navigate itself along a highway or, in the future, even navigate complicated city streets unaided.

While not as exciting perhaps, organizations have been using the power of ML more and more over the past decade to improve things such as fraud detection, or to predict whether a consumer with a specific set of attributes is likely to default on a loan.

AWS offers several services to help developers build their own custom advanced ML models, as well as a variety of pretrained models that can be used for specific purposes. In this chapter, we'll examine why **artificial intelligence (AI)** and ML matter to organizations, and we'll review a number of the AWS AI and ML services, as well as how these services use different types of data.

In this chapter, we will cover the following topics:

- Understanding the value of ML and AI for organizations
- Exploring AWS services for ML
- Exploring AWS services for AI
- Hands-on – reviewing the reviews with Amazon Comprehend

Before we get started, review the following *Technical requirements* section, which lists the prerequisites for performing the hands-on activity at the end of this chapter.

Technical requirements

In the last section of this chapter, we will go through a hands-on exercise that uses **Amazon SQS** and **AWS Lambda**, to send some text to the **Amazon Comprehend** service so that we can extract insights from it.

As with the other hands-on activities in this book, if you have access to an administrator user in your AWS account, you should have the permissions needed to complete these activities. If not, you will need to ensure that your user is granted access to create Amazon SQS and AWS Lambda resources, as well as at least read-only permissions for Amazon Comprehend APIs.

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/tree/main/Chapter13>

Understanding the value of ML and AI for organizations

More and more companies, of all sizes, are in various stages in the journey of discovering how ML and AI can positively impact their business. While initially, only the largest of organizations had the money and expertise to invest in ML projects, over time, the required technology has become more affordable and more accessible to non-specialist developers.

Cloud providers, such as AWS, have played a big part in making ML and AI technology more accessible to a wider group of users. Today, a developer with no previous ML education or experience can use a service such as **Amazon Lex** to create a customer service **chatbot**. This chatbot will allow customers to ask questions using natural language, rather than having to select from a menu of preset choices. Not all that long ago, anyone wanting to create a chatbot like this would have needed a Ph.D. in ML!

Many large organizations still look to build up data science teams with specialized AI and ML education and experience, and these developers are often involved in cutting-edge research and development. However, organizations of just about any size can use non-specialist developers to harness the power of ML to improve customer experience, financial forecasting, and other aspects of their business.

Let's have a look at some of the ways that ML is having an impact on different types of organizations.

Specialized ML projects

Large organizations in specialized industries make use of advanced ML technologies to develop cutting-edge ML advances. In this section, we'll have a look at a few examples of these technologies.

Medical clinical decision support platform

Cerner, a health information technology services company, has built an ML-powered clinical decision support system to help hospitals streamline their workflows. This solution, built on AWS, uses ML models to predict how busy an emergency room may get on any given day, or time. This helps ensure that the right patients are prioritized for care, that patients are discharged at the right time, and that real-time data is used to create a **Centralized Operations Center** dashboard. This dashboard provides critical, near-real-time information on important metrics for managing hospital workflows, as well as predictions for what these metrics may look like over time.

Cerner has built their *Cerner Machine Learning Ecosystem* platform using **Amazon SageMaker**, as well as other AWS services. As with just about all ML projects, getting the right data to train the ML model is critical, and data engineers play an important role in this. In addition, data engineers are needed to build pipelines that enable near-real-time data to be ingested from multiple sources and fed into the platform. If the pipeline fails to ingest the right data at the right frequency, then the ML models cannot make the predictions that an organization may have come to depend on.

To learn more about the *Cerner clinical decision support system*, you can watch a pre-recorded webinar, available at <https://www.youtube.com/watch?v=TZB8W7BL0eo>.

Early detection of diseases

One of the areas of ML and AI that has massive potential for impacting a significant number of people is the early detection of serious diseases.

A November 2020 article in the international peer-reviewed journal *Nature*, titled *Artificial intelligence is improving the detection of lung cancer* (<https://www.nature.com/articles/d41586-020-03157-9>), provides an in-depth look into how AI is positively impacting the medical field. In this article, the author (Elizabeth Svoboda) provides an example of how a **deep learning** ML model was able to correctly detect the early stages of lung cancer on CT scans 94% of the time, which was better than a panel of six veteran radiologists.

With many terminal diseases, early detection can make a significant difference in the outcome for the patient. For example, early detection, combined with appropriate medical interventions, can significantly increase the chance of survival beyond 5 years for certain cancer patients.

Making sports safer

Another area that ML is having an impact on is improving the safety of athletes for competitive sports. For example, the **National Football League (NFL)** in the United States is using Amazon AI and ML services to derive new insights into player injuries, rehabilitation, and recovery.

NFL has started a project that uses **Amazon SageMaker** to develop a deep learning model to track players on a field, and then detect and classify significant injury events and collisions. There is an expectation that these advanced ML models, along with vast quantities of relevant data (including video data), can be used to significantly improve player safety over time.

To learn more about how NFL is using ML to improve player safety, you can watch a short video on YouTube from the AWS re:Invent 2020 conference titled *AWS re:Invent 2020 – Jennifer Langton of the NFL on using AWS to transform player safety* (<https://www.youtube.com/watch?v=hXxfCn4tGp4>).

Having had a look at a few specialized use cases, let's look at how everyday businesses are using ML and AI to impact their organizations and customers.

Everyday use cases for ML and AI

Just about every business, ranging from those with tens of employees to those with thousands of employees, is finding ways to improve through the use of ML and AI technologies.

One of the big reasons for this is that ML and AI have become more democratized over the past few years. Whereas ML and AI were once solely the domains of experts with years of experience in the field, today, a developer without specialized ML experience can harness the power of these technologies in impactful ways.

Let's have a look at a few examples of how ML and AI are widely used across different business sectors.

Forecasting

Just about every organization needs to do forecasting to anticipate a variety of factors that influence their business. This includes financial forecasting (such as sales and profit margin), people forecasting (such as employee turnover, and how many staff are needed for a particular shift), and inventory forecasting (such as how many units we are likely to sell, how many units we need to manufacture next month, and so on).

Forecasting uses historical data over a period (often referred to as time series data) and attempts to predict likely future values over time. Forecasting has been around since long before ML, but traditional forecasts often lacked accuracy due to things such as irregular trends in historical data. Traditional forecasting also often failed to take into account variable factors, such as weather, promotions, and more.

ML has introduced new approaches and techniques to forecasting that offer increased accuracy and the ability to take several variable factors into account. AWS offers several services that help bring the power of ML to forecasting problems, as we will discuss later in this chapter.

Personalization

Personalization is all about tailoring communication and content for a specific customer or subscriber. A good example of personalization is the effort Netflix has invested in to provide personalized recommendations about other shows a specific subscriber may be interested in watching, based on the shows they have watched in the past.

Other examples of where ML is used to power personalized recommendations are the recommended products on the Amazon.com storefront, as well as the recommended travel destinations on booking.com.

Natural language processing

Natural language processing (NLP) is a branch of AI/ML that is used to analyze human language and draw automated insights and context from the text.

A great example of NLP is the Alexa virtual assistant from Amazon. Users can speak to Alexa using natural language, and Alexa uses NLP algorithms to understand what the user is asking. While voice recognition systems have been around for a long time, these generally required users to say very specific phrases for the system to understand them. With modern NLP approaches, 10 different users could ask the same question in 10 slightly different ways, and the system would be able to understand what is being asked.

Image recognition

Another area where ML is having an impact on many businesses is through the use of image recognition ML models. With these models, images can be analyzed by the model to recognize objects within them. This can be used for many different types of tasks, such as ensuring employees are wearing appropriate safety gear, or as part of the process of validating the identity of a customer. These models are also able to automatically label images based on what is in the image, such as the breed of dog in a collection of dog photos.

Now that we have reviewed some examples of the typical use cases for ML and AI, we can do a deeper dive into some of the AWS services that enable these use cases.

Exploring AWS services for ML

AWS has three broad categories of ML and AI services, as illustrated in the following diagram (note that only a small sample of AI and ML services are included in this diagram, due to space constraints):

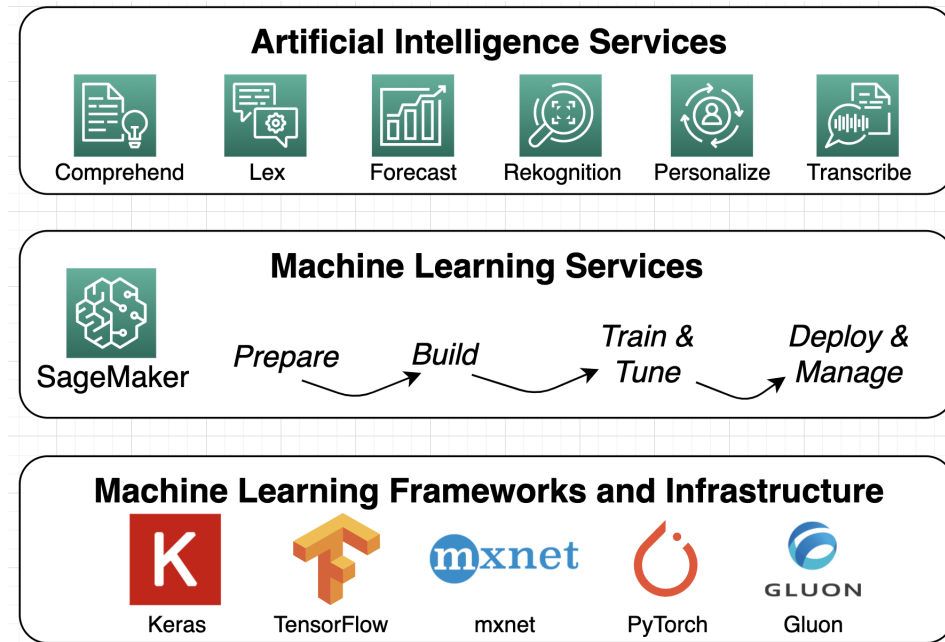


Figure 13.1 – Amazon ML/AI stack

In the preceding diagram, we can see a subset of the services that AWS offers in each category – **Artificial Intelligence Services**, **Machine Learning Services**, and **Machine Learning Frameworks and Infrastructure**.

At the ML framework level, AWS provides **Amazon Machine Images (AMIs)** and prebuilt Docker containers that have popular deep learning ML frameworks pre-installed and optimized for the AWS environment. While these are useful for advanced use cases that require custom ML environments, these use cases are beyond the scope of this book.

For more information on these ML frameworks, refer to the AWS documentation on *AWS Deep Learning AMIs* (<https://aws.amazon.com/machine-learning/amis/>) and *AWS Deep Learning Containers* (<https://aws.amazon.com/machine-learning/containers/>).

In the remainder of this chapter, we will explore some of the services in the AWS ML services and AWS AI services categories.

AWS ML services

While working in the **Machine Learning Frameworks and Infrastructure** layer (as shown in the preceding diagram) requires advanced ML skills and experience, AWS makes developing ML models more accessible in the **Machine Learning Services** layer.

In this layer, Amazon SageMaker enables users to prepare, build, train, tune, deploy, and manage ML models, without needing to manage the underlying infrastructure. SageMaker is designed to simplify each step of building an ML model for both data scientists and everyday developers.

SageMaker includes several underlying tools to help with each of the stages of building an ML model.

SageMaker in the ML preparation phase

Several capabilities within SageMaker simplify and speed up the tasks involved in preparing to build an ML model. We covered these services in *Chapter 8, Identifying and Enabling Data Consumers*, so review that chapter for more information, but here is a quick reminder of these services.

Amazon SageMaker Ground Truth

The majority of ML models *learn* by being trained on labeled data. That is, the model is effectively given data that includes the attribute the model is designed to predict. Once trained, the model can then predict data where the attribute to be predicted is missing. For example, to train a model that can identify different breeds of dogs in a photo, you would train the model using photos of dogs that are labeled with the breed of dog. Once trained, you could provide a picture of a dog and the model could predict the breed.

SageMaker Ground Truth is a service that uses both ML and/or human curators to label data; for example, labeling the breed of a dog in a photo. This significantly speeds up the process of preparing data to use to train new ML models.

Amazon SageMaker Data Wrangler

The **SageMaker Data Wrangler** service is a visual data preparation tool that data scientists can use to prepare raw data for ML use. The service enables data scientists to select relevant datasets, explore the data, and then select from over 300 built-in transformations that they can easily apply to the dataset, without writing any code.

SageMaker Data Wrangler also includes visualization templates that enable you to preview the results of transformations in **SageMaker Studio**, a full-fledged **integrated development environment (IDE)** for ML.

Amazon SageMaker Clarify

When training an ML model with a training dataset, the dataset may be biased through either a concentration of specific data or because it is missing specific data.

For example, if a dataset is intended to be used to predict responses from people with a wide age range, but the training dataset primarily contains data from people aged 35 – 55, then predictions may be inaccurate for both younger people (under 35) and/or older people (over 55).

The same could be applied to datasets that tend to concentrate on a specific gender, sexual orientation, married versus un-married, or just about any other attribute. To help avoid this type of potential bias in a dataset, **SageMaker Clarify** can examine specified attributes in a dataset and use advanced algorithms to highlight the existence of potential bias.

SageMaker in the ML build phase

Once data has been labeled and prepared, a data scientist can move on to building ML models. The following capabilities in SageMaker are used to build new ML models.

SageMaker Studio notebooks

Data scientists typically use notebooks to develop the code for their ML models. A notebook is an interactive web-based environment where developers can run their code and immediately see the results of the running code. An interactive notebook is backed by a compute engine that runs a kernel where notebook code is executed.

With **SageMaker Studio Notebooks**, you can quickly launch a new notebook, backed by an EC2 instance type of your choosing. The notebook environment uses **Amazon Elastic File System (EFS)**, which is network-based storage that persists beyond the life of the instance running the notebook. This enables you to easily start and stop different notebook instances, and have your notebook project files available in each notebook instance.

SageMaker Studio Notebooks also enables users to easily share notebooks, enabling collaborative work between data scientists on a team. In addition, SageMaker Studio Notebooks provides sample projects that can be used as a starting point for developing a new model.

SageMaker Autopilot

For developers that do not have extensive ML experience, **SageMaker Autopilot** can be used to automatically build, train, and tune several different ML models, based on your data.

The developer needs to provide a tabular dataset (rows and columns) and then indicate which column value they want to predict. This could be predicting a number (such as expected spend), a binary category (fraud or not fraud), or a multi-label category (such as favorite fruit, which could be banana, peach, pear, and so on).

SageMaker Autopilot will then build, train, and tune several ML models and provide a model leaderboard to show the results of each model. Users can view each of the models that were generated and explore the results that were generated by each model. From here, a user can select the model that best meets their requirements and deploy it.

SageMaker JumpStart

SageMaker JumpStart provides several preselected end-to-end solutions, ML models, and other resources to help developers and data scientists get their ML projects up and running quickly.

By using these prebuilt resources, developers can easily deploy solutions and models with all the infrastructure components managed for them. Once deployed, the model can be opened with SageMaker Studio Notebooks, and the model can be tested through a notebook environment.

Prebuilt solutions include sample datasets that can be used to test the model, and you can also provide your own dataset to further train and tune the model. Some examples of prebuilt solutions available in JumpStart include the following:

- Churn prediction
- Credit risk prediction
- Computer vision
- Predictive maintenance

For more information on SageMaker JumpStart, including an example of how a solution can easily be deployed, see the AWS blog post by Julien Simon titled *Amazon SageMaker JumpStart Simplifies Access to Pre-built Models and Machine Learning Solutions* (<https://aws.amazon.com/blogs/aws/amazon-sagemaker-jumpstart-simplifies-access-to-prebuilt-models-and-machine-learning-models/>).

SageMaker in the ML training and tuning phase

Once you have built an ML model, you need to train the model on a sample dataset, and then further tune and refine the model until you get the results that meet your requirements.

Training a model is core functionality that's built into SageMaker. You point SageMaker to the location of your training data in Amazon S3, and then specify the type and quantity of SageMaker ML instances you want to use for the training job. SageMaker will provision a distributed compute cluster and perform the training, outputting the results to Amazon S3. The training cluster will then be automatically removed.

SageMaker can also automatically tune your ML model by testing the model with thousands of different combinations of algorithm parameters to determine which combination of parameters provides the most accurate results. This process is referred to as **hyperparameter tuning**, and with SageMaker, you can specify the range of hyperparameters that you want to test.

To keep track of the results of different training jobs, SageMaker also includes something called SageMaker Experiments.

SageMaker Experiments

This process of tracking different ML experiments can be made significantly easier using **SageMaker Experiments**. This feature of SageMaker automatically tracks items such as inputs, parameters, and configurations, and stores the result of each experiment. This helps reduce the overhead and time needed to identify the best performing combinations for your ML model.

When running a training job on SageMaker, you can pass in an extra parameter, defining the name of the experiment. By doing this, all the inputs and outputs of the job will be automatically logged.

This data can then be loaded into a pandas DataFrame (a popular Python data structure for working with data), and you can use the built-in analytics features of pandas to analyze your results. Amazon SageMaker Studio also includes integration with SageMaker Experiments, enabling you to run queries on experiments data, and view leaderboards and metrics.

SageMaker in the ML deployment and management phase

Once you have prepared your data, developed your model, and then trained and tuned the model, you are finally ready to deploy the model. There are several different ways that you can select to deploy the model using SageMaker.

For example, if you want to get predictions on a large dataset, you can use SageMaker's batch transform process. Using this, you point SageMaker to the dataset on S3, select the type of compute instance you want to use to power the transform, and then run the transform job, which will make a prediction for each record in the dataset and write out the transformed dataset to S3.

Alternatively, you can deploy an endpoint for your model that can be used by your applications to pass data to the model to get an ML-powered prediction in real time. For example, you can pass information to the endpoint of a specific credit card transaction (date, time, location, vendor, amount, and so on), and the ML model can predict whether this is a fraudulent or genuine transaction.

ML models can become less accurate over time due to changing trends in your customer base, for example, or because of data quality issues in upstream systems. To help monitor and manage this, you can use SageMaker Model Monitor.

SageMaker Model Monitor

SageMaker Model Monitor can be configured to continuously monitor the quality of your ML models and can send notifications when there are deviations in the model's quality. Model Monitor can detect issues with items such as data quality, model quality, and bias drift.

To resolve issues with model quality, a user may take steps such as retraining the model using updated data or investigating potential quality issues with upstream data preparation systems.

Having briefly covered some of the extensive functionality available for creating custom models using Amazon SageMaker, let's look at some of the AWS AI services that provide prebuilt ML models as a service.

Exploring AWS services for AI

While Amazon SageMaker simplifies building custom ML models, there are many use cases where a custom model is not required, and a generalized ML model will meet requirements.

For example, if you need to translate from one language into another, that will most likely not require a customized ML model. Existing, generalized models, trained for the languages you are translating between, would work.

You could use SageMaker to develop a French to English translation model, train the model, and then host the model on a SageMaker inference endpoint. But that would take time and would have compute costs associated with each phase of development (data preparation, notebooks, training, and inference).

Instead, it would be massively simpler, quicker, and cheaper to use an AI service such as **Amazon Translate**, which already has a model trained for this task. This service provides a simple API that can be used to pass in text in one language and receive a translation in a target language. And there would be no ongoing compute costs or commitments – just a small per-character cost for the translation (currently \$ 0.000015 per character).

Also, AWS is constantly working to improve the underlying ML algorithms, monitoring data quality, and maintaining the availability of the API endpoints, at no additional cost to you. And if you do need to customize the model (for example, based on specific industry terminology, or a preferred style or tone for the translation), you can provide additional training data for customized translations, although this comes at a slightly higher cost (currently \$0.00006 per character).

These types of AI services have gained in popularity over the past few years, and all of the major cloud providers now offer a range of pretrained ML models as a service. We don't have space in this chapter to cover all of the AWS AI services, but we'll look at a few of the most popular services in this section.

We started with Amazon Translate as an example of an AWS AI service, so now, let's explore some of the other AI offerings from AWS.

AI for unstructured speech and text

One of the primary benefits of a data lake is the ability to store all types of data, including **unstructured data** such as PDF documents, as well as audio and video files, in the data lake. And while this type of data can be easily ingested and stored in the data lake, the challenge for the data engineer is in how to process and make use of this data.

For example, a large enterprise company may have hundreds of thousands of invoices from a variety of vendors, and they may want to perform analysis or fraud detection on those. Or a busy call center may want to automatically transcribe recorded customer calls to perform sentiment analysis and identify unhappy customers.

For these use cases, AWS offers several AI services designed to extract metadata from text or speech sources to make this data available for additional analysis.

Amazon Transcribe for converting speech into text

Amazon Transcribe is an AWS AI service that can produce text transcription from audio and video files. This can be used to generate subtitles for a video file, to provide a transcription of a recording of a meeting or speech, or to get a transcript of a customer service call.

Transcribe uses **automatic speech recognition (ASR)**, a deep learning process, to enable highly accurate transcriptions from audio files, including the ability to identify different speakers in the transcript. Transcribe can also detect and remove sensitive personal information (such as credit card numbers or email addresses) from transcripts, as well as words that you don't want to be included in a transcription (such as curses or swear words). Transcribe can also generate a new audio file that replaces these unwanted words with silence.

A data engineer can build a pipeline that processes audio or video files with Transcribe, ensuring that text transcripts from audio sources are generated shortly after new audio sources are ingested into the data lake. Other ML models or AWS AI services can also be built into the pipeline to further analyze the transcript to generate additional metadata.

Amazon Transcribe also includes functionality targeted at specific types of audio. For example, **Amazon Transcribe Medical** uses an ML model specifically trained to identify medical terminologies such as medicine names, diseases, and conditions. And **Amazon Transcribe Call Analytics** has been specifically designed to understand customer service and sales calls, as well as to identify attributes such as agent and customer sentiment, interruptions, and talk speed.

Amazon Textract for extracting text from documents

Amazon Textract is an AI service that can be used to automatically extract text from unstructured documents, such as PDF or image files. Whether the source document is a scan of printed text or a form that includes printed text and handwriting, Textract can be used to create a semi-structured document for further analysis.

A data engineer may be tasked, for example, with building a pipeline that automatically analyzes uploaded expense receipts to extract relevant information. This may include storing that information in semi-structured files in the data lake, or a different target such as DynamoDB or a relational database.

For example, the following screenshot shows a portion of a hotel receipt bill contained in a PDF file:

DATE	REF NO	DESCRIPTION	CHARGES
4/15/2019	2559498	GUEST ROOM	\$179.00
4/15/2019	2559498	STATE TAX	\$10.74
4/15/2019	2559498	CITY TAX	\$16.11
4/16/2019	2559777	C3 FOOD DRINK	\$7.00
4/16/2019	2559811	VS	(\$212.85)
BALANCE			\$0.00

Hilton Honors(R) stays are posted within 72 hours of checkout. To check your earnings or book your next stay at more than 4,000 hotels and resorts in 100 countries, please visit Honors.com

Thank you for choosing Doubletree! Come back soon to enjoy our warm chocolate chip cookies and relaxed hospitality. For your next trip visit us at doubletree.com for our best available rates!

Figure 13.2 – Extract from a PDF document of a hotel invoice

Most traditional analytic tools would not be able to process this data contained within a PDF file, but when this file is sent to the Amazon Textract service, a semi-structured file can be created containing relevant data. For example, the ML model powering Textract can extract information from the preceding table as a CSV file that can be further analyzed in a data engineering pipeline.

The following table shows the CSV file when opened in a spreadsheet application:

DATE	REF NO	DESCRIPTION	CHARGES
4/15/2019	2559498	GUEST ROOM	\$179.00
4/15/2019	2559498	STATE TAX	\$10.74
4/15/2019	2559498	CITY TAX	\$16.11
4/16/2019	2559777	C3 FOOD DRINK	\$7.00
4/16/2019	2559811	VS	(\$212.85)

Figure 13.3 – CSV formatted data extracted from a PDF invoice

Textract has been designed to work well with various types of documents, including documents that contain handwritten notes. For example, a medical intake form at a doctor's office, where patients fill out the form by hand, can be sent to Textract to extract data from the form for further processing.

Amazon Comprehend for extracting insights from text

We have looked at how Amazon Transcribe can create electronic text from speech, as well as how Amazon Textract can create semi-structured documents from scanned documents and images. Now, let's look at how to extract additional insights from text.

Amazon Comprehend is an AI service that uses advanced ML models to generate additional insights from text documents, such as sentiment, topics, place names, and more. With Comprehend, you can build a near-real-time pipeline that passes in 1 – 25 documents in a single API call for analysis or build a batch pipeline that configures Comprehend to analyze all documents in an S3 bucket.

When you call the API or run an asynchronous batch job, you specify the type of comprehension that you want in the results. For example, you can have Comprehend analyze text to detect the dominant language, entities, key phrases, PII data, sentiment, or topics (each type of comprehension has a different API call).

Comprehend can be used for several use cases, such as identifying important entities in lengthy legal contracts (such as location, people, and companies), or understanding customer sentiment when customers interact with your call center. As a data engineer, you may be tasked with building a pipeline that uses Amazon Transcribe to convert the audio of recorded customer service calls into text, and then run that text through Comprehend to capture insight into customer sentiment for each call.

Another use case could be to analyze social media posts to identify which organizations were being referenced in a post, and what the sentiment of the review was. For example, we could analyze the following fictional post made to a social media platform:

"I went to Jack's Cafe last Monday, and the pancakes were amazing! You should try this place, it's new in downtown Westwood. Our server, Regina, was amazing."

When Amazon Comprehend analyzes this text, it returns the following insights:

- **Entities detected:**
 - Jack's café, Organization, 93% confidence
 - Westwood, Location, 71% confidence
 - Regina, Person, 99% confidence
 - last Monday, Date, 94% confidence
- **Sentiment:**
 - Positive, 99% confidence

As we can see from the previous results, Comprehend can accurately detect entities and sentiment. At the end of this chapter, we will go through an exercise with Amazon Comprehend to determine customer sentiment from online reviews, which will allow you to get hands-on with how Amazon Comprehend works.

Note that there is also a specialty version of Comprehend, called **Amazon Comprehend Medical**, that has been designed to extract medical information from electronic text, such as medical conditions, medications, treatments, and protected health information. You can also train a **Comprehend custom entity detection** model using your data to recognize specialized entities (such as a model trained to recognize different makes and models of cars and motorbikes).

AI for extracting metadata from images and video

In the previous section, we reviewed AI services for processing text – including audio transcribed into text (Amazon Transcribe), images and scanned documents converted into text (Amazon Textract), and insights drawn out of electronic text (Amazon Comprehend).

In this section, we will change focus and look at how we can extract insights out of videos and images using the power of AI.

Amazon Rekognition

Amazon Rekognition uses the power of pretrained ML models to extract metadata from images and videos, enabling users to get rich insights from this unstructured content.

With traditional data warehouses and databases, the ability to store unstructured data, such as images and videos, was very limited. In addition, until recently, it was difficult to extract rich metadata from these unstructured sources, without having humans manually label data. And, as you can imagine, this was a very time-consuming and error-prone process.

For organizations that stored a lot of images or videos, they needed to manually build catalogs to tag the media appropriately. For example, these organizations would need someone to manually identify celebrities in photos or add labels to an image to tag what was shown in the image.

As ML technologies advanced, these organizations could build and train ML models to automatically tag images (or stills from a video), but this still required deep expertise and an extensive labeled catalog for training the ML model.

With new AI services, such as **Amazon Rekognition**, vendors do the hard work of building and training the ML models, and users can then use a simple API to automatically extract metadata from images. And, with **Amazon Rekognition Video**, users can also gain these insights from video files. When passed a video file for analysis, the results that are returned include a timestamp of where the object was detected, enabling an index of identified objects to be created.

For example, the following photo could be sent to the Amazon Rekognition service to automatically identify elements in the photo:



Figure 13.4 – Photo of a dog and a Jeep in the snow

When passed to Amazon Rekognition, the service can automatically identify objects in the photo. The following is a partial list of the identified objects (with the confidence level of the ML model shown in brackets):

- Outdoors (99.6%)
- Snow (99.2%)
- Blizzard (99.2%)
- Winter (99.2%)
- Wheel (95.3%)
- Dog (93.6%)
- Car (90.8%)

A data engineer could use this type of service to build a data pipeline that ingests images and/or video, and then calls the Amazon Rekognition service for each file, building an index of objects in each file, and storing that in DynamoDB, for example.

The AI services we have discussed so far are used to extract data from unstructured files such as PDF scans and image and video files. Now, let's take a look at AWS AI services that can be used to make predictions based on semi-structured data.

AI for ML-powered forecasts

A common business need is to forecast future values, whether these be the number of staff an entertainment venue is likely to need next month, or how much revenue an organization is likely to receive on a specific product line over the next 12 months.

For many years, organizations would use formulas to forecast future values, based on historical data that they had built up. However, these formulas often did not take into account seasonal trends and other third-party factors that could significantly influence the actual values that are realized.

Modern forecasting tools, such as Amazon Forecast, can provide significantly more accurate forecasts by using the power of ML.

Amazon Forecast

Amazon Forecast is a powerful AI service for predicting future time series data, based on complex relationships between multiple datasets. Using Forecast, a developer can train and build a customized forecast ML model, without needing ML expertise.

To train the custom model, a user would provide historical data for the attribute that they want to predict (for example, daily sales at each store over the past 12 months). In addition, they can include related datasets, such as a dataset listing the total number of daily visitors to each store.

If the primary dataset also includes geolocation data (identifying, for example, the location of the store) and timezone data, Amazon Forecast can automatically use weather information to help further improve prediction accuracy. For example, the model can take into account how the weather has affected sales in the past, and use the latest 14-day weather forecast to optimize predictions for the upcoming period based on the weather forecast.

A data engineer may be involved in building a pipeline that uses Amazon Forecast. The following could be some of the steps in a pipeline that the data engineer architects and implements:

- Use an **AWS Glue** job to create an hourly aggregation of sales for each store, storing the results in Amazon S3.
- Use **AWS Step Functions** to call Lambda functions that clean up previous predictions, and generate new predictions based on the latest data. Use a Lambda function to create an export job to export the newly generated predictions to Amazon S3.
- Use **Amazon AppFlow** to load the newly generated predictions from Amazon S3 to Amazon Redshift for further analysis.

Refer to the AWS blog post titled *Automating your Amazon Forecast workflow with Lambda, Step Functions, and CloudWatch Events rule* (<https://aws.amazon.com/blogs/machine-learning/automating-your-amazon-forecast-workflow-with-lambda-step-functions-and-cloudwatch-events-rule/>) for more details on building a pipeline that incorporates Amazon Forecast.

AI for fraud detection and personalization

The AI services we discussed previously are often incorporated into data engineering pipelines as these services are useful for advanced analytics (such as extracting metadata from images, text transcripts from audio files, or making forecasts). However, other AI services are often used as a part of transactional systems, rather than data engineering pipelines, which we will briefly look at in this section.

Amazon Fraud Detector

Amazon Fraud Detector is an AI service that helps organizations detect potentially fraudulent transactions and fake account registrations.

Fraud Detector enables an organization to upload its historical data regarding fraudulent transactions. It then adds this to a model trained with fraud data from Amazon and AWS to optimize fraud detection.

Using Fraud Detector, an organization can build fraud prediction into their checkout process, getting a prediction within milliseconds as part of the checkout process.

Amazon Personalize

Amazon Personalize is an AI service that helps organizations provide personalized recommendations to their customers. Using Personalize, developers can easily integrate personalized product recommendations, marketing initiatives, and other ML-powered personal recommendations into existing customer-facing systems.

With Personalize, developers can design systems that capture live events from users (such as data extracted from a website click-stream) and combine this with historical user profile information to recommend the most relevant items for a user. This can be used to recommend other products a customer may be interested in, or the next movie or TV show a customer may like to watch.

Having reviewed several AWS AI services, let's get hands-on with using one of these services: Amazon Comprehend.

Hands-on – reviewing reviews with Amazon Comprehend

Imagine that you work for a large hotel chain and have been tasked with developing a process for identifying negative reviews that have been posted on your website. This will help the customer service teams follow up with the customer.

If your company was getting hundreds of reviews every day, it would be time-consuming to have someone read the entire review every time a new review was posted. Luckily, you have recently heard about Amazon Comprehend, so you decide to develop a small **Proof of Concept (PoC)** test to see whether Amazon Comprehend can help.

If your PoC is successful, you will want to have a decoupled process that receives reviews once they have been posted, calls Amazon Comprehend to determine the sentiment of the review, and then takes a follow-up action if the review is negative or mixed. Therefore, you decide to build your PoC in the same way, using Amazon **Simple Queue Service (SQS)** to receive reviews and have this trigger a Lambda function to perform analysis with Comprehend.

Setting up a new Amazon SQS message queue

Create a new Amazon SQS message queue for receiving reviews by following these steps:

1. Log into **AWS Management Console** and navigate to the **Amazon SQS** service at <https://console.aws.amazon.com/sqs/v2/>.
2. Click on **Create queue**.

3. Leave the default of a **Standard** queue as-is and provide a queue **Name** (such as `website-reviews-queue`):

The screenshot shows the 'Create queue' page in the Amazon SQS console. The breadcrumb navigation is 'Amazon SQS > Queues > Create queue'. The main heading is 'Create queue'. Under the 'Details' section, the 'Type' is set to 'Standard' (selected with a radio button). A warning message states: 'You can't change the queue type after you create a queue.' Below this, two options are shown: 'Standard' (selected) and 'FIFO'. The 'Standard' option includes the text 'At-least-once delivery, message ordering isn't preserved' and a bulleted list: 'At-least once delivery' and 'Best-effort ordering'. The 'FIFO' option includes the text 'First-in-first-out delivery, message ordering is preserved' and a bulleted list: 'First-in-first-out delivery' and 'Exactly-once processing'. Below the 'Type' section, the 'Name' field contains the text 'website-reviews-queue'. A note below the name field states: 'A queue name is case-sensitive and can have up to 80 characters. You can use alphanumeric characters, hyphens (-), and underscores (_).'.

Figure 13.5 – Creating a new Amazon SQS message queue

4. Leave all other options as their default values and click on **Create queue** at the bottom of the page.

Now that our queue has been created, we want to create a Lambda function that will read items from the queue and submit the website review text to Amazon Comprehend for analysis.

Creating a Lambda function for calling Amazon Comprehend

The following steps will create a new Lambda function for calling Amazon Comprehend to analyze the text that's passed in from the SQS queue:

1. In the Amazon Management Console, navigate to the **AWS Lambda** service at <https://us-east-2.console.aws.amazon.com/lambda/>.
2. Click on **Create function**.
3. Select the option to **Author from scratch**.
4. Provide a **Function name** value (such as `website-reviews-analysis-function`) and select the most recent version of **Python** for **Runtime**.

5. For **Execution role**, select **Create a new role from AWS policy templates**.
6. Provide a **Role name value** (such as `website-reviews-analysis-role`).
7. For **Policy templates**, search for SQS and add **Amazon SQS poller permissions**.
8. Leave everything else as the defaults and click on **Create function**.

Having created our function, we can add our custom code, which will receive the SQS message, extract the review text from the message, and then send it to Amazon Comprehend for sentiment and entity analysis.

9. Replace **Code source** in Lambda with the following block of code:

```
import boto3
import json
comprehend = boto3.client(service_name='comprehend',
                           region_name='us-east-2')
def lambda_handler(event, context):
    for record in event['Records']:
        payload = record["body"]
        print(str(payload))
```

In this preceding block of code, we imported the required libraries and initialized the Comprehend API, which is part of `boto3`. Make sure that you modify the preceding Comprehend API initialization code to reflect the region you are using for these exercises. Then, we defined our Lambda function and read in the records that we received from Amazon SQS. Finally, we loaded `body` of record into a variable called `payload`.

Continue your Lambda function with the following block of code:

```
print('Calling DetectSentiment')
response = comprehend.detect_sentiment(
    Text=payload, LanguageCode='en')
sentiment = response['Sentiment']
sentiment_score = response['SentimentScore']
print(f'SENTIMENT: {sentiment}')
print(f'SENTIMENT SCORE: {sentiment_score}')
```

In this preceding block of code, we called the Comprehend API for **sentiment detection**, passed in the review text (`payload`), and specified that the text is in English. In the response we receive from Comprehend, we extracted the `sentiment` property (positive, mixed, or negative), as well as the `SentimentScore` property.

Now, let's look at our last block of code:

```
print('Calling DetectEntities')
response = comprehend.detect_entities(
    Text=payload, LanguageCode='en')
print(response['Entities'])
for entity in response['Entities']:
    entity_text = entity['Text']
    entity_type = entity['Type']
    To the PD: Please add this to the next line
    in p-regular style
```

To correctly print over two lines we need the following code:

```
print(f'ENTITY: {entity_text}, '
      f'ENTITY TYPE: {entity_type}')
      ENTITY TYPE: {entity_type}')

return
```

In this final part of our code, we called the Comprehend API for entity detection, again passing in the same review text (`payload`). Multiple entities may be detected in the text, so we looped through the response and printed out some information about each entity.

Then, we returned without any error, which indicates success, which means the message will be deleted from the SQS message queue. Note that for a production implementation of this code, you would want to add error-catching code to raise an exception if there were any issues when calling the Comprehend API.

10. Click **Deploy** in the Lambda console to deploy your code.

Now, we just need to add permissions to our Lambda function to access the Comprehend API and add our function as a trigger for our SQS queue. Then, we can test it out.

Adding Comprehend permissions for our IAM role

When we created our Lambda function, we were able to select from a preset list of common permissions to add permission for our Lambda function to poll an SQS message queue. However, our function also needs to call the Comprehend API, so let's add permission for that as well:

1. In **AWS Lambda console**, with your website reviews analysis function open, click on the **Configuration** tab along the top, and then the **Permissions** tab on the left.
2. The name of the role you specified when creating the Lambda function will be shown as a link. Click on **Role name** (such as `website-reviews-analysis-role`) to open the IAM console so that we can edit the permissions:

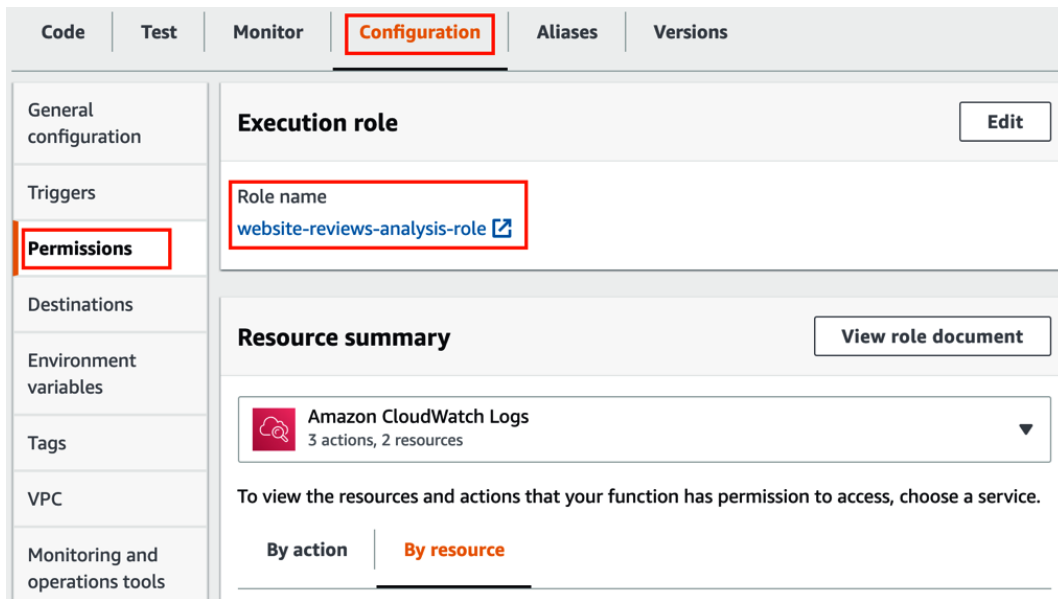


Figure 13.6 – Lambda Permissions > Configuration tab > Execution role

3. In the IAM console, click on **Attach policies**.
4. Search for a policy called **ComprehendReadOnly**, which has sufficient permissions to enable us to call the Comprehend API from our Lambda function.

5. Select the tick box for **ComprehendReadOnly**, and then click on **Attach policy**:

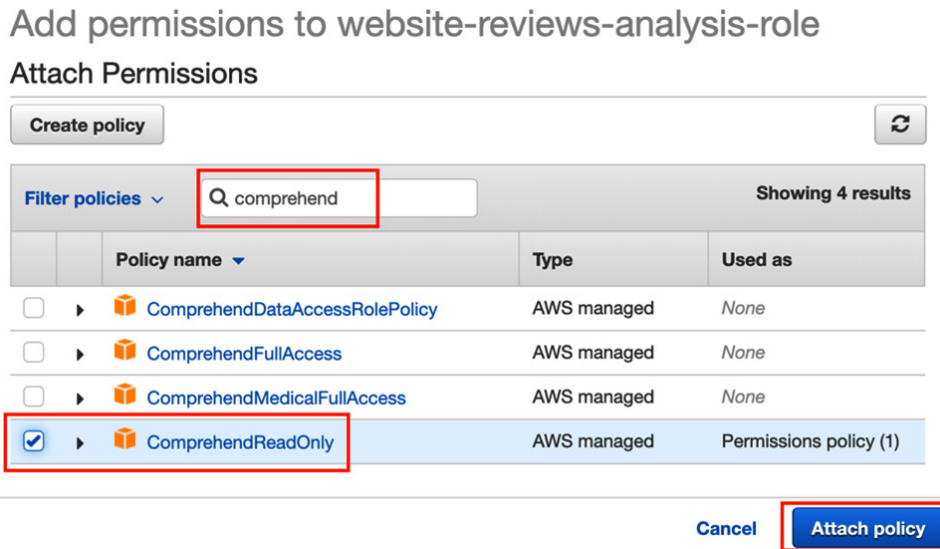


Figure 13.7 – Finding and selecting the required Comprehend permissions in IAM

We are just about ready to test our function. Our last step is to link our SQS queue and our Lambda function.

Adding a Lambda function as a trigger for our SQS message queue

With the following steps, we'll configure our Lambda function to be able to pick up new messages that are added to our SQS message queue for processing:

1. Navigate back to the Amazon SQS message queue console at <https://us-east-2.console.aws.amazon.com/sqs/v2/home>.
2. Click on the name of the SQS queue you previously created (such as `website-reviews-queue`).
3. Click on the **Lambda triggers** tab, and then click **Configure Lambda function trigger**.
4. Make sure that **Region** is set to the region you have been using for the exercises in this chapter, and then select your Lambda function from the drop-down list.
5. Click **Save** to link your SQS queue and Lambda function.

And with that, we are now ready to test out our solution and see how Amazon Comprehend performs.

Testing the solution with Amazon Comprehend

Using the following steps, test the solution and get Amazon Comprehend to analyze the text you have provided for both sentiment and entity detection:

1. Ensure that you are still on the Amazon SQS console and that your SQS queue is open.
2. At the top right, click on **Send and receive messages**:

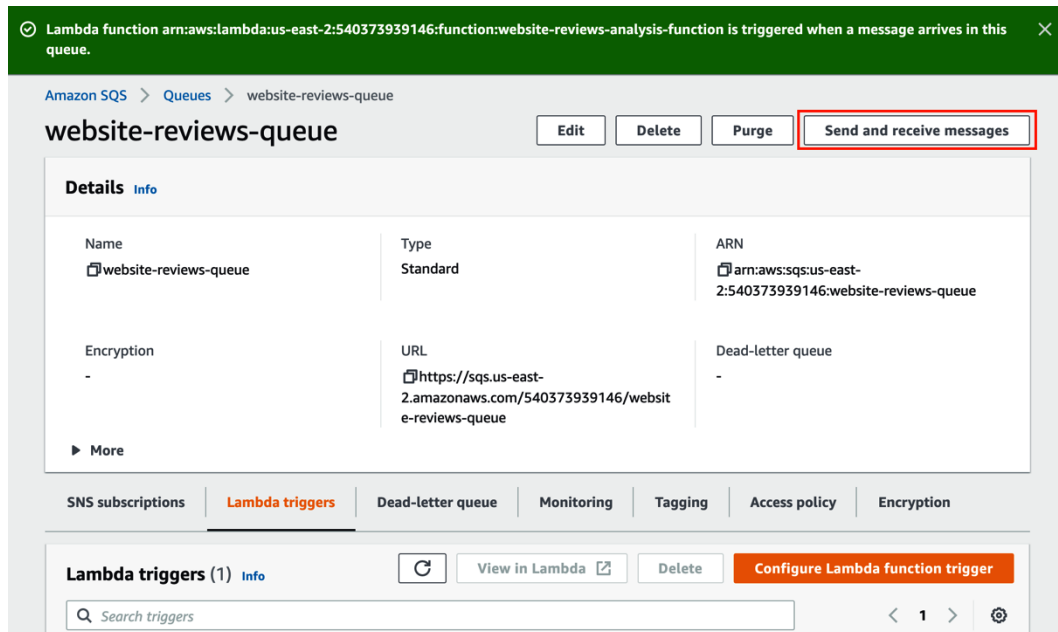


Figure 13.8 – Amazon SQS queue detail view

We can now send a message directly to our SQS queue, which will trigger our Lambda function to process the message and send it to Amazon Comprehend. When moved to production, we would build integration into our website to automatically send all new reviews to our Amazon SQS message queue as the reviews are posted.

3. Paste the following text (or your own, similar text) into the **Message Body** section of **Send and receive messages**:

"I recently stayed at the Kensington Hotel in downtown Cape Town and was very impressed. The hotel is beautiful, the service from the staff is amazing, and the sea views cannot be beaten. If you have the time, stop by Mary's Kitchen, a coffee shop not far from the hotel, to get a coffee and try some of their delicious cakes and baked goods."

Then, click on the **Send message** option at the top right.

- To view the results of the Comprehend analysis, we can review the output of our Lambda function in CloudWatch Logs. If it's not already open in a separate browser tab, open a new browser tab and navigate back to your **Lambda function**. Click on the **Monitor** tab, and then click **View logs in CloudWatch**. This will open the CloudWatch console in a new browser tab.

The **CloudWatch** console should have opened at the log group for your Lambda function (for example, the log group named `/aws/lambda/website-reviews-analysis-functions`). Click on the latest log stream to open the log:

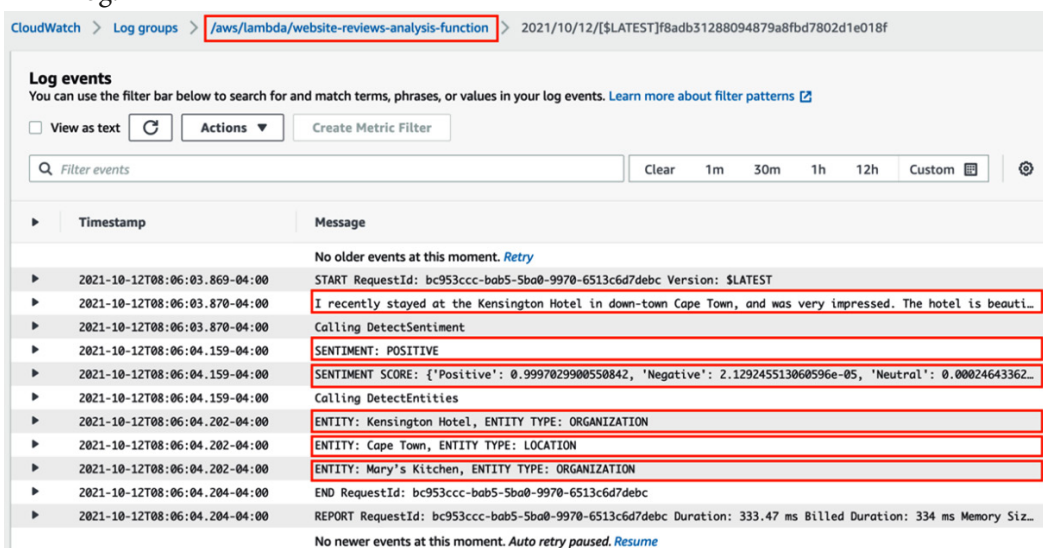


Figure 13.9 – Amazon CloudWatch logs for our Lambda function

In the CloudWatch logs, you can see the output of our Lambda function. This includes the text that was analyzed, the sentiment (POSITIVE), the sentiment score, as well as the three entities Comprehend detected in our text (the hotel and coffee shop names, and the city location).

- Go back to your browser tab for the SQS console and modify the review text with a negative review. You can either write your own fictional negative review or copy and paste a negative review that you find via Google. Send the message via SQS and review the analysis results in CloudWatch to see how Comprehend detects the negative sentiment, and see which other entities Comprehend can detect.

After testing and validating that Amazon Comprehend can reliably detect sentiment from published reviews, you may decide to move forward with implementing this solution in production. If you do decide to do this, you could use Amazon Step Functions to build a workflow that runs a Lambda function to do the sentiment analysis. Then, depending on the results (positive, negative, neutral, or mixed), the Step Function state machine could run different Lambda functions based on the next steps (such as sending a negative review to customer service to follow up with the customer or sending a mixed review to a manager to decide on the next steps).

With this hands-on exercise, you got to experiment with how Amazon Comprehend can detect both sentiment and entities in written text. If you have time, you can explore the functionality of other Amazon AI services directly in the console. This includes Amazon Rekognition, Amazon Transcribe, Amazon Textract, and Amazon Translate.

Summary

In this chapter, you learned more about the broad range of AWS ML and AI services and had the opportunity to get hands-on with Amazon Comprehend, an AI service for extracting insights from written text.

We discussed how ML and AI services can apply to a broad range of use cases, both specialized (such as detecting cancer early) and general (business forecasting or personalization).

We examined different AWS services related to ML and AI. We looked at how different Amazon SageMaker capabilities can be used to prepare data for ML, build models, train and fine-tune models, and deploy and manage models. SageMaker makes building custom ML models much more accessible to developers without existing expertise in ML.

We then looked at a range of AWS AI services that provide prebuilt and trained models for common use cases. We looked at services for transcribing text from audio files (Amazon Transcribe), for extracting text from forms and handwritten documents (Amazon Textract), for recognizing images (Amazon Rekognition), and for extracting insights from text (Amazon Comprehend). We also briefly discussed other business-focused AI services, such as Amazon Forecast and Amazon Personalize.

We're near the end of a journey that has had us look, at a high level, at several tasks, activities, and services that are part of the life of a data engineer. In the next chapter, we will conclude this book by looking at some additional examples of data engineering pipelines, and briefly introduce other topics that a data engineer may wish to explore for further learning.

Further reading

To learn more about the impact that ML is having on the medical field regarding automatically detecting serious diseases, check out these articles:

- *Machine Learning and Deep Learning Approaches for Brain Disease Diagnosis: Principles and Recent Advances* (<https://ieeexplore.ieee.org/document/9363896>)
- *AI Algorithm Can Accurately Predict Risk, Diagnose Alzheimer's Disease* (<https://www.bumc.bu.edu/busm/2020/05/04/ai-algorithm-can-accurately-predict-risk-diagnose-alzheimers-disease/>)
- *Implementing AI models has made critical disease diagnosis easy* (<https://www.analyticsinsight.net/implementing-ai-models-has-made-critical-disease-diagnosis-easy/>)
- *Apple hiring machine learning scientist for early disease detection* (<https://www.healthcareitnews.com/news/apple-hiring-machine-learning-scientist-early-disease-detection>)

The organizations behind these articles and headlines use advanced ML principles and technologies to further advance medical diagnosis. It's still early days in this field, and this will be an interesting space to watch over the next few years.

14

Wrapping Up the First Part of Your Learning Journey

In this book, we have explored many different aspects of the data engineering role by learning more about common architecture patterns, understanding how to approach designing a data engineering pipeline, and getting hands-on with many different AWS services commonly used by data engineers (for data ingestion, data transformation, and orchestrating pipelines).

We examined some of the important issues surrounding data security and governance and discussed the importance of a data catalog to avoid a data lake turning into a data swamp. We also reviewed data marts and data warehouses and introduced the concept of a data lake house.

We learned about data consumers – the end users of the *product* that's produced by data engineering pipelines – and looked into some of the tools that they use to consume data (including Amazon Athena for ad hoc SQL queries and Amazon QuickSight for data visualization). Then, we briefly explored the topic of **machine learning (ML)** and **artificial intelligence (AI)** and learned about some of the AWS services that are used in these fields.

In this chapter, we're going to introduce some important real-world concepts to help manage the data infrastructure/pipeline development process, have a look at some examples of real-world data pipelines, and discuss some emerging trends in the field. We'll then look at how to clean up your AWS account in the hands-on portion of this chapter.

In this chapter, we will cover the following topics:

- Looking at the data analytics big picture
- Examining examples of real-world data pipelines
- Imagining the future – a look at emerging trends
- Hands-on – cleaning up your AWS account

Technical requirements

There are no specific technical requirements for the hands-on section of this chapter as we will just be cleaning up resources that we have created throughout this book. Optionally, however, there will be a section that covers deleting your AWS account. If you choose to do this, you will need access to the account's root user to log in with the email address that was used to create the account.

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/tree/main/Chapter14>

Looking at the data analytics big picture

This book was never intended as a deep dive into one specific area of data engineering, although there are many other great books and resources out there that do focus on a single area (such as a deep dive on Spark programming, or on how to use Kafka to ingest streaming data).

Because of this broad topic coverage, you have probably already begun to form a good idea of the different aspects of the bigger picture of data analytics. While it is quite common for data engineering roles to focus on just writing data transform jobs, or just managing the infrastructure to ingest and process streaming data, it is helpful to understand how this integrates with data warehouses/data marts, how different data consumers use data, and how ML and AI fit into the bigger data picture, as we have reviewed in this book.

We have also been focusing on the tasks from the perspective of a single data engineer, but in reality, most data engineers will work as part of a larger team. There may be different teams, or team members, focused on different aspects of the data engineering pipeline, but all team members need to work together.

In most organizations, there are also likely to be multiple environments, such as a development environment, a test/**quality assurance (QA)** environment, and a production environment. The data infrastructure and pipelines must be deployed and tested in the development environment first, and then any updates should be pushed to a test/QA environment for automated testing, before finally being approved for deployment in the production environment.

In the following diagram, we can see that there are multiple teams responsible for different aspects of data engineering resources. We can also see that the data engineering resources are duplicated across multiple different environments (which would generally be different AWS accounts), such as the development environment, test/QA environment, and production environment. Each organization may structure its teams and environments a little differently, but this is an example of the complexity of data engineering in real life:

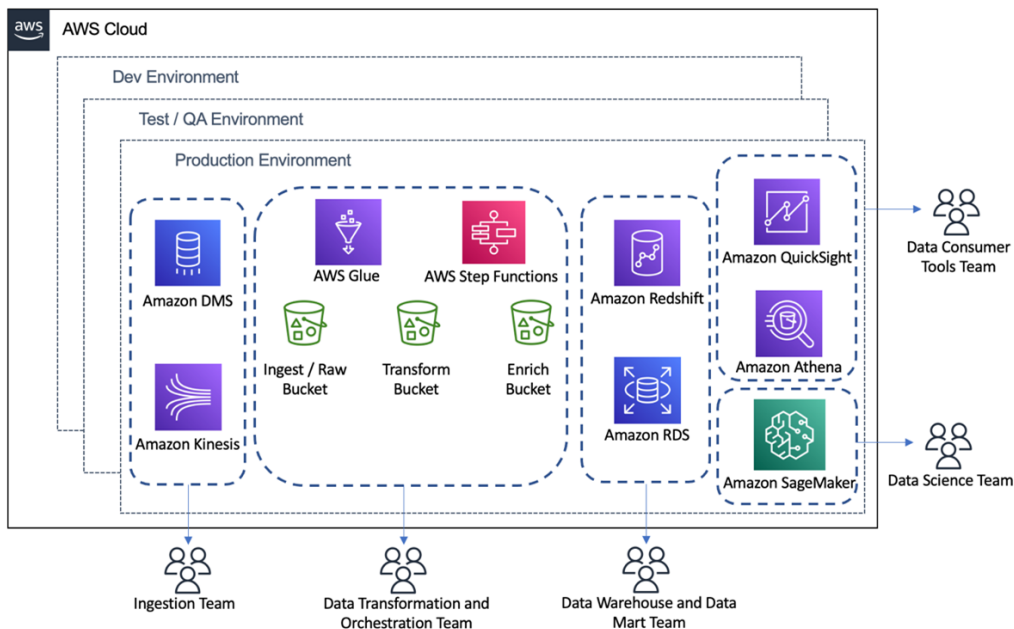


Figure 14.1 – Data engineering teams and environments

It is a challenge to work in these kinds of complex environments, and an organized approach is required to be successful. Part of understanding the bigger picture of data analytics is to understand these types of challenges and how to overcome them, as we will look at in this section.

Managing complex data environments with DataOps

DataOps is a set of processes and principles that can be applied to manage how changes to data infrastructure (including pipelines) are deployed to a production environment. The purpose of DataOps is to bring repeatability and reliability to the process of transforming data to increase its value, as well as to make it production-ready for use by data consumers in the shortest possible time and with the highest data quality possible.

The *opposite* of DataOps would be entirely manual processes for making changes to data infrastructure that can vary each time a change is made, and with no formal controls for testing and approving changes that are made in a production environment, or for ensuring data quality.

DataOps builds on the well-known DevOps processes and principles for software engineering and applies similar processes to data. We don't have time in this book to do a deep dive into DataOps, but we will introduce some of the important concepts here and encourage you to read up more on this topic.

Data infrastructure and pipelines as source control-managed code

One of the big benefits of running workloads in the cloud is the ability to automate all the aspects of infrastructure deployment. While traditionally, you may think of code as being software engineering code, such as mobile phone or web applications, infrastructure deployments can also be encapsulated in code.

Infrastructure as Code (IaC) refers to the process of using code, or definition templates, to control the deployment and configuration of infrastructure. In AWS, the **AWS CloudFormation (CFN)** service uses template files to specify the definition and configuration of infrastructure that you want to deploy to an AWS account.

With CloudFormation, you create a template file (using either YAML or JSON formatted text) that specifies the details of the resources you want to deploy.

For example, you can create a CFN template that can be used to automatically deploy the following resources into an AWS account:

- An S3 bucket that is configured to block public access
- An EventBridge rule to monitor files being written to that bucket and triggers a Lambda function when a new file is written
- An SNS topic that is used for sending failure notifications

- A Lambda function that validates the new file that is received, and then launches a Step Function state machine
- A state machine that launches a Glue job to process the file, which then runs a Glue crawler to update the Glue catalog and sends an SNS notification if anything fails

Once you have created the template definition file, you can commit that file to a **source code repository**, such as **AWS CodeCommit**, **Azure DevOps (ADO)**, **BitBucket**, or **GitLab**. The source code repository enables other team members to access and modify the template file you committed, and the source code system helps manage changes and conflicts in different versions of the template file.

In the same way, code that's used for data transformation jobs (such as Python, PySpark, or Scala code) or orchestration jobs (such as Step Function state machine definition files) can also be committed to a source control repository.

Continuous integration/continuous delivery

Continuous integration (CI) refers to automated processes that are run when a new version of a file is committed to a source control repository. These automated processes are used to build the code when required (such as building a JAR file) and integrate the newly committed code into existing code that makes up the target system.

For example, when a new Python file is committed to the source control repository, automated tests can be done to validate that the code in the pipeline meets certain quality standards, syntax style requirements, and more. At this stage, unit tests can also be run to test the quality of the code (a simple test to make sure that the code works as expected – such as ensuring that a specific function returns the expected output, based on a given input).

Some organizations prefer to run the automated test process on every commit to the repository, and in others, the tests will only be run when a `pull` request is raised to merge new code from a developer's branch into a main branch of the repository.

Continuous delivery (CD) refers to the process of automatically deploying code changes into target environments, generally with additional automated end-to-end testing. For example, after code is merged from a developer's branch into a main branch of the repository, the full repository may be deployed into a test environment. In this test environment, automated tests will run that do end-to-end testing (such as ingesting files, running transformation pipelines, and validating output files).

While in some cases, CD processes may be triggered whenever a `pull` request finishes merging code from developer branches into the main repository, some organizations prefer to kick off this process manually (although in this case, it could not strictly be called CD).

For example, an organization may choose to do a once-daily, or even weekly, deployment of all merged code changes into the test environment. Once testing has been completed, they will then manually kick off the process to deploy the updated code into the production environment. However, the processes to deploy the code and perform validation testing will still be automated. There will also be automated processes to roll back the changes to the previous version in case of failure.

DataOps brings source control repositories and CI/CD processes together as part of an agile approach to developing and deploying data infrastructure, transformation pipelines, and orchestration. The teams that develop the code (whether transformation code or infrastructure code) are also responsible for overseeing the process of deploying code to the production environment and managing any issues that arise.

We have only briefly introduced the core concepts of DataOps, but there is much more to learn, and there are many good online resources that can enable you to dive deeper into this topic.

In the next section, we will look at some real-life examples of complex data engineering pipelines.

Examining examples of real-world data pipelines

The data pipeline examples that we have used in this book have been based on common types of transformations and pipelines, but they have been relatively simple examples. As you can imagine, in large organizations, the types of data pipelines that are built can be a lot more complex and may end up processing extremely large sets of data.

In this section, we will examine two examples of more complex data engineering pipelines from two very well-known organizations – **Spotify** and **Netflix**. Both of these companies have public blogs that cover software and data engineering, and the details provided about their pipelines in this section have been taken from the public information that's been made available in a variety of blog posts and articles.

A decade of data wrapped up for Spotify users

Every year, for the past few years, the music streaming service *Spotify* has used the extensive data they have on their user's listening history to generate interesting stats for each user. This information is made available to each user at the end of the year and includes information such as how many minutes of Spotify audio they streamed that year, as well as their top artist, top track, and top genre for the year.

This information is marketed to users as **Spotify Wrapped**, which is a massive undertaking for multiple teams at Spotify, including marketing, frontend app engineering, and, of course, data engineering.

While Spotify has been presenting the Spotify Wrapped feature for several years, in 2019, they decided to add a new feature by reporting on a user's listening trends for each year of the past decade (2010 – 2019). In an official Spotify blog post, *Spotify Unwrapped: How we brought you a decade of data* (<https://engineering.atspotify.com/2020/02/18/spotify-unwrapped-how-we-brought-you-a-decade-of-data/>), the Spotify data engineering team revealed some of the behind-the-scenes work they did to aggregate user data by year, over 10 years.

In this blog post, the data engineering team talks about some challenges they faced with the wrapped project in 2018, and how they had to work closely with Google (their cloud provider) to be able to achieve the required processing scale. For 2019, they were planning to do something similar to 2018, but they had more users (totaling 248 million monthly active users) and were planning to do this for 10 years of listening history. As a result, they used the lessons they had learned from their 2018 experience to modify their approach for 2019.

Spotify considers each statistic they want to report for an individual user (such as top artist or top track) as a separate data story. So, to meet the scale requirements for a decade of data, they decided to persist intermediate data and final data for Spotify Wrapped 2019 in **Google BigTable** (a NoSQL database that is somewhat similar to Amazon DynamoDB). For every Spotify user, they had a row in BigTable with a column for each data story, for each year of the decade. This was a significant change from how they had processed and collected different data stories for each user in previous years, but this led to a significantly improved process as data was now pre-grouped and collated per user in BigTable.

They could then write separate jobs for most data stories (decoupling the data stories from each other) and run these individually, but could also run multiple different data story jobs in parallel. The output of each of these data story jobs would then be saved to the same `userid` row in BigTable. End-of-decade top statistics could then be aggregated directly from the data in BigTable.

The key takeaways that we can learn from this example are as follows:

- It is good to iterate on data engineering pipelines and continually reevaluate the architecture and approach you use to identify better ways to do things.
- Breaking down large jobs into smaller, decoupled jobs can lead to improved efficiencies. Keep a modular design for your jobs and avoid the temptation to create a single job that does everything.
- Be versatile and flexible in the tools you use. While we did not have space to cover NoSQL databases in any significant way in this book, a NoSQL database may be an ideal target for storing some of the output from your big data processing jobs. For example, DynamoDB was designed to handle billions of rows of data in a table, as well as enable extremely fast access to individual rows from that large dataset.

Data engineers are often challenged to come up with innovative new ways to draw insights out of extremely large datasets, as demonstrated in this real-world example from Spotify. Now, let's look at another real-life data processing example, this time from Netflix.

Ingesting and processing streaming files at Netflix scale

Netflix, the world's leading streaming video platform with over 200 million subscribers worldwide, predominantly uses AWS for its compute infrastructure. As you can imagine, it takes a lot of compute power and many different microservices and applications to support a user base of that size.

Monitoring and understanding how network traffic flows between all the different Netflix microservices, across many separate AWS accounts, is key for the following:

- Maintaining a resilient service
- Understanding dependencies between services
- Troubleshooting when things do go wrong
- Identifying ways to improve the user experience

One of the features of the Amazon **Virtual Private Cloud (VPC)** service is the ability to generate **VPC FlowLogs**, which capture details on network traffic between all network interfaces in a VPC (a private cloud-based network environment in an AWS account).

However, most AWS services make use of dynamic IP addresses, meaning that the IP address that's used by a system can frequently change. So, while VPC FlowLogs provide rich information on network communications between IP addresses, if you don't know which applications or services had the IP addresses being reported on at that time, the flow logs are largely meaningless.

Enriching VPC FlowLogs with application information

To have data that was meaningful, Netflix determined that they needed to enrich VPC FlowLogs with information about which application was using a specific IP address at the point in time recorded in the VPC flow log. To capture this information, Netflix created an internal system called **Sonar** that uses CloudWatch Events, Netflix Events, API calls, and various other methods to capture a stream of IP change events.

In 2017, AWS featured the Netflix solution for this in a case study on their website titled *Netflix & Amazon Kinesis Data Streams Case Study* (<https://aws.amazon.com/solutions/case-studies/netflix-kinesis-data-streams/>). In this case study, it was explained that Netflix used a large **Kinesis Data Streams cluster** (of up to 1,000 shards) to process incoming VPC FlowLogs. An internal Netflix application known as **Dredge** was created to read incoming data from the Kinesis Data Stream, as well as enrich the VPC flow log data with application metadata from the Sonar stream of IP change events, identifying the applications or microservices involved with each VPC flow log record. This enriched data was then loaded into an open source, high-performance, real-time analytics database called **Druid**, where users could efficiently analyze network data for troubleshooting and to gain improved insights into network performance.

Amazon VPC enhancements and changing the architecture

In the cloud, things change frequently, and AWS is constantly enhancing its services and adding additional services in response to customer feedback. In August 2018, AWS enhanced the VPC Flow Logs service so that logs could be delivered directly to Amazon S3, without needing to be processed via Kinesis first.

In May 2020, Netflix posted a public blog post titled *How Netflix is able to enrich VPC Flow Logs at Hyper Scale to provide Network Insight* (<https://netflixtechblog.com/hyper-scale-vpc-flow-logs-enrichment-to-provide-network-insight-e5f1db02910d>). This blog post shows how Netflix has changed its architecture to make the best use of the updated functionality in the VPC Flow Logs feature.

In this blog post, Netflix talks about a common pattern that they have for processing newly uploaded S3 files. When a new file is uploaded to S3, it is possible to configure an action to take place in response to the newly uploaded file (as we did in *Chapter 3, The AWS Data Engineer's Toolkit*, where we triggered a Lambda function to transform a CSV file into Parquet format whenever a new CSV file was uploaded to a specific S3 bucket prefix).

Netflix commonly uses this pattern to write details of newly uploaded files to an Amazon SQS queue, and they can then read events from the queue to process the newly arrived files. This enables them to decouple the S3 event from the action that they wish to perform in response to this event.

In this case, Netflix intended to read through the entries on the SQS queue and use the file size information included in the event notification to determine the number of newly ingested VPC flow log files to process in a batch (which they refer to as a *mouthful* of files). They intended to use an Apache Spark job that would enrich the VPC flow log with application metadata based on the IP addresses recorded in each record. They would tune the Apache Spark job to optimally process a certain amount of data, which is why they would read the file size information contained in the SQS messages to create an optimally sized mouthful of files to send to the Spark job.

With the Amazon SQS service, messages are read from the queue and processed. If the processing is successful, the processed messages are deleted from the queue. During this processing time, the messages are considered to be *in flight* and will be hidden from the queue so that no other application attempts to process the same files. If something goes wrong and the files are not successfully processed and deleted from the queue, the messages will become visible again after a certain amount of time (known as the visibility timeout period) so that they can be picked up by an application again for processing.

In the case of Netflix, they would send a mouthful of files to an Apache Spark job, and once the Spark job successfully processed the messages, the messages would be deleted from the queue.

However, the Amazon SQS service has a limit on the number of files that can be considered to be in flight at any point (the default quota limit is 120,000 messages). Netflix found that because the Spark jobs would take a little while to process the files, they were regularly ending up with 120,000 or more messages in flight. As a result, they came up with an innovative way to work around this by using two different SQS queues.

Working around Amazon SQS quota limits

The re-architected Netflix solution reads the SQS queue containing the S3 events and runs a process to create a mouthful of files (evaluating each file's size to create a batch that is the optimal size for their Spark jobs). This process can complete very quickly as it does not need to read or process the files, just read the metadata contained in the SQS messages to group a mouthful of files to be processed by a Spark job.

The output of the first job writes a message to a second SQS queue, and each message contains the list of files in a single mouthful. While the blog does not provide any indication of how many files may usually be contained in a mouthful of files, if we assumed it was, on average, around 10 files, it would reduce the number of messages on the second SQS queue by 90%. If a mouthful of files was, on average, 100 files, then the number of messages written to the secondary SQS queue would be reduced by 99%.

The Netflix blog does not provide enough details to be able to describe the exact architecture of the solution, but the following diagram shows an example of a potential architecture for this solution (this may not be the architecture that Netflix implemented):

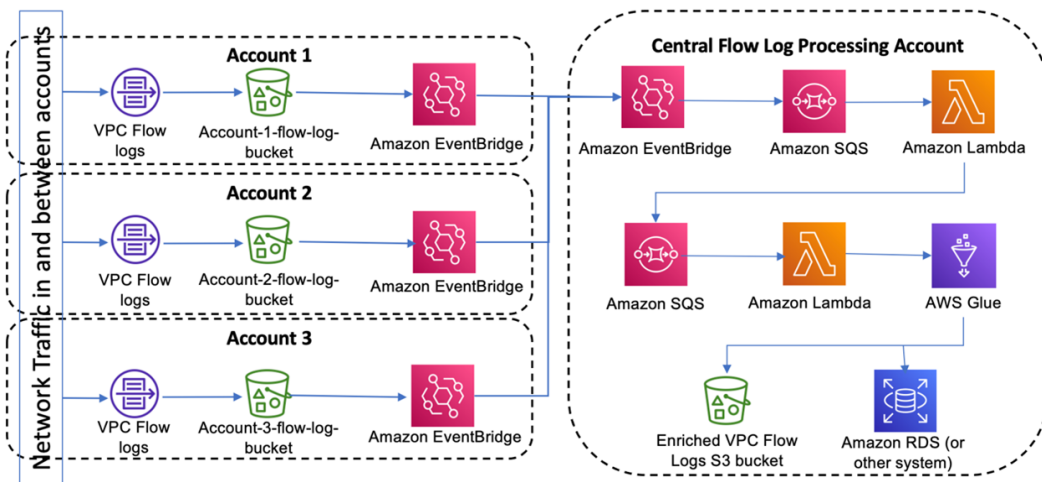


Figure 14.2 – A potential architecture for VPC Flow Logs processing and enriching

In the preceding diagram, we have VPC Flow Logs configured to write to an Amazon S3 bucket in each account where we want to monitor network activity. We have also configured an EventBridge rule in each account that analyzes CloudTrail log files to pick up S3 write events from the VPC flow log bucket. When a write event is detected in the CloudTrail log file, an action is taken to write the S3 event message to an EventBridge bus in a central account.

In the central account, an EventBridge rule detects the S3 events from the source accounts and takes an action to write each S3 event message to an Amazon SQS queue. A Lambda function has been configured that reads messages from the Amazon SQS queue and uses the file size metadata contained in the message to create a batch of files of an optimal size (called a mouthful of files by Netflix). The list of files in the batch is written to a separate SQS queue as a single message. This Lambda function can complete and remove messages from the first SQS queue very quickly as it is only processing metadata in the SQS messages, not reading/writing S3 files, and running a Glue job to enrich the files.

A separate Lambda function processes the much smaller number of messages in the secondary SQS queue by reading the list of files in the mouthful. The list of files is passed to a Glue job that runs Spark code to enrich the VPC Flow Logs files in this mouthful with data from other sources. Enriched files are written to S3 and/or a database system such as RDS.

The key takeaways that we can learn from this example are as follows:

- It is important to know what the AWS quotas/limits are for the services that you use. Some limits can be raised by contacting AWS support, but some limits are hard limits that cannot be increased.
- It is important to stay up to date with *what's new* announcements from AWS. AWS regularly launches new services, as well as major new features, for existing services.

As shown in this blog post, sometimes, new features from AWS can help you significantly simplify existing architectures and reduce costs (based on this blog post, it would seem that Netflix may no longer need their 1,000-shard Kinesis Data Streams cluster to process VPC FlowLogs).

In the next section, we will look at upcoming trends and what the future may hold for data engineers.

Imagining the future – a look at emerging trends

Technology seems to progress at an increasing velocity. For decades, relational databases from vendors such as **Oracle** were the primary technology for managing all data. Today, there is a wide range of different database types that can be used, depending on the use case (such as graph databases for highly connected datasets, or NoSQL databases for low-latency reading and writing for very large tables).

It was also not all that long ago that **Hadoop MapReduce** was the state-of-the-art technology for processing very large datasets, but today, most new projects would choose **Apache Spark** over a MapReduce implementation. And even Apache Spark itself has progressed from its initial release, with Spark 3.0 being released in June 2020. We have also seen the introduction of **Spark Streaming**, **Spark ML**, and **Spark GraphX** for different use cases.

No one can tell for certain what the next big thing will be, but in this section, we will look at a few emerging concepts and technologies, as well as expected trends, that are likely to be of relevance to data engineers.

ACID transactions directly on data lake data

A trend that is developing currently is the **atomicity, consistency, isolation, durability (ACID)** properties for data lake tables, which provide consistency for dataset transactions (concurrent reads and writes). In addition, a lot of these new technologies incorporate the ability to update or delete individual records from a table in the data lake. Before these new technologies were introduced, the lack of ACID transactions and the ability to update and delete records in data lakes was a significant challenge, and each implementation of a data lake would need to create approaches to work around this challenge.

We discussed these emerging technologies in more detail in *Chapter 7, Transforming Data to Optimize for Analytics*. Refer to the *Modern approaches – the transactional data lake* section for more information on these new technologies, including **Databricks Delta Lake**, **Amazon Lake Formation Governed Tables**, and **Apache Hudi**.

More data and more streaming ingestion

A trend that is not new, but that is expected to continue over the next few years, is that of the increasing generation of new data. Not all newly generated data will be stored for long periods, but forecasts do indicate continued growth in stored data.

It is expected that more and more organizations will also adopt data lakes, in addition to existing data warehouse solutions. And with drivers such as increased ML and AI projects, you can expect the amount of data to be ingested, cleansed, and processed to continue increasing significantly.

Another trend we are seeing with data lakes is more and more data ingestion sources becoming **streaming-based**, rather than **batch-based**. Batch-based ingestion and processing are unlikely to go away anytime soon, but over time, streaming data is likely to become a larger percentage of ingested data compared to batch ingestion.

Some of the drivers of this increase in streaming data sources include the following:

- **IoT data**, such as data from sensors and wearable devices
- **Point-of-Sale (PoS)** devices that deliver real-time transaction data to data lakes
- **Event-based workflows**, such as the Netflix example of a stream of recorded changes to IP addresses
- **Ingestion of real-time internet-based data**, such as social media feeds, product reviews, weather forecasts, website scraping, and other sources

Refer to *Chapter 6, Ingesting Batch and Streaming Data*, for more information on AWS services for data ingestion. Also, consider doing a deeper dive into popular streaming technologies such as Amazon Kinesis, Spark Streaming, Apache Kafka, and Apache Flink.

Multi-cloud

While this book focuses on data engineering using AWS services, there is a trend for many larger companies to adopt a **multi-cloud strategy**, where they use more than one cloud provider for services.

Having a multi-cloud strategy can introduce numerous challenges across **information technology (IT)** teams, including challenges for data engineering teams that need to work with data stored with different cloud providers. Another challenge for IT teams and data engineers is the need to learn the different service implementations for each cloud provider (for example, AWS, Azure, and Google Cloud each offer a managed Apache Spark environment, but the implementation details are different for each provider).

There are many different reasons for organizations wanting to adopt a multi-cloud strategy, but the pros and cons need to be carefully thought through. However, in many cases, data engineers will have no option but to take up the challenge of becoming comfortable with working in multiple different cloud provider environments.

Decentralized data engineering teams, data platforms, and a data mesh architecture

Since the dawn of computer departments in companies, there has been a constant back and forth between centralizing processing and decentralizing processing.

At the start, mainframes were a good example of centralized IT systems, where all processing was done by a central team. Then, in the '90s, there was a move to have departmental servers and systems, with decision-making done at the department level. This led to siloed systems and databases, which led to the introduction of data warehouses (and later, data lakes), to bring data back into a central place.

In many cases, data engineers would work for a centralized team, and they would be responsible for ingesting data from all the departments into the central data warehouse, as well as any processing that was required of the data. They could then load a subset of the processed data back into a data mart, which data analysts in the departmental teams could then work with to analyze the data.

But, once again, there is a noticeable trend to move this centralized control of data back to a decentralized model, although this time with a twist.

In May 2019, Zhamak Dehghani, a principal technology consultant at *ThoughtWorks*, wrote a blog post (<https://martinfowler.com/articles/data-monolith-to-mesh.html>) that got a lot of people rethinking the approach of a centralized data engineering team. In this blog post, Dehghani introduces the concept of a **dash mesh architecture**.

We don't have the space to do a deep dive into this architecture shift in this book, so you are encouraged to read the original blog post mentioned previously, as well as subsequent blog posts by Dehghani, on this topic. However, we will outline the basic concepts here.

Domain-orientated data decomposition and ownership

In the original blog post, Dehghani argues that instead of data flowing from business domains into a centrally owned data lake, individual business domains should host and serve analytic information related to their domain to others in the business.

When referring to a business domain here, we are talking about the team in the business that owns the relevant operational data. For example, in a real-estate business, you may have a team that is responsible for all property listings. They gather the details for each property and make this operational data available to other parts of the business through an API (such as a `getListingPrice` API call, which returns the listing price of a property).

Traditionally, those teams may have then made the database that contains all the listings available as a source for a centralized data engineering team to ingest from. This centralized data engineering team would then be responsible for ingesting from the database, ensuring data quality, creating daily snapshots of current listings, aggregating listing data, and more.

However, Dehghani makes the point that the team that owns the operational data should also be the owner of the analytics data related to that domain. This could involve, for example, making a daily snapshot of all the listings available to other teams, or creating a stream of change events related to listings (new listing, removed listing, modified listing, and so on).

Data and product thinking convergence

For the data mesh model to be successful, Dehghani proposes that domain data teams need to apply product thinking to the datasets they provide. That is, they should look at the analytic data they create for others in the business to consume, as a product they are offering. They need to learn about what their consumers want out of the data they generate (much like a product manager would solicit feedback and requirements from customers to develop their product roadmap).

These domain data teams also need to ensure that they make their data discoverable by other data consumers in the business, and accessible in a way that meets organizational standards. The domain data team should also provide metadata, such as schema information, to best enable their data consumers to work with the provided data assets.

To achieve this, data domain teams will need new roles, such as data product owners and data engineers, on the domain team (rather than just having centralized data engineers that do not have specific domain knowledge).

Data and self-serve platform design convergence

While this approach uses a decentralized design for the ownership of domain data, this does not mean that a centralized data processing platform cannot be used with a data mesh architecture.

You don't want each domain team creating infrastructure for data engineering processing, data storage, orchestration, and so on. Therefore, you may still have a centralized team that builds *data infrastructure as a platform*. However, this platform should be data domain agnostic – that is, it should not include any domain-specific logic. This platform should also provide data services to domain teams in a way that they can self-serve, so they should not need the help of the data platform team to create a new data engineering pipeline.

The centralized data platform should do things such as the following:

- Providing big data processing systems, such as a managed Spark environment that domain data teams can easily access
- Providing a central catalog where domain data teams can publish their available datasets
- Implementing corporate governance standards and controls (such as how to identify and manage PII data, tokenize data, and so on)
- Providing an access control system that allows other domain teams to request access to a specific dataset, get approval for access from the data owner, and then grant access to the domain data

This overview of data mesh concepts does not cover all the aspects of each concept, so it is strongly encouraged that you read the original blog post, as well as other articles and resources, related to the data mesh architecture.

Implementations of the data mesh architecture

As with most new concepts or approaches that are proposed, the actual implementation of the concept in the real world may vary greatly. As organizations look to implement or migrate to a data mesh architecture, some organizations may focus on specific aspects of the architecture initially or may implement a simpler version of the architecture.

For example, an organization may use a centralized team to create a data platform that very much resembles a traditional data lake. However, instead of centralizing a team of data engineers, they will encourage each business unit that wants to use the platform to employ its own team of data engineers.

The central platform team will provide Amazon S3 storage buckets, as well as their associated access controls, for each business unit team. They will also provide a framework for ingesting data into S3 using DMS, and for processing and orchestrating data pipelines using Lambda, Glue, and Step Functions. They will provide access to tools such as Amazon Athena for ad hoc data exploration and AWS Lake Formation for centralized cataloging.

Each business unit will be responsible for processing raw data and transforming and enriching the data. If a different business unit wants access to that enriched data, the centralized team will have created forms in **ServiceNow** (a software solution for managing business workflows) that can be used to request access to the data. ServiceNow will route the request for access to the business unit that owns the data, and when the request is approved, the centralized team will have an automated process to grant access. Each business unit may use a separate AWS account, so the automated process for granting access may leverage AWS Lake Formation cross-account access functionality to grant access to the target business unit account.

While this solution may not reflect a data mesh architecture in the same way that Dehghani envisioned it, it still employs concepts of a data mesh architecture. Primarily, it still achieves one of the important goals of a data mesh – moving data ownership and processed domain data out of centralized teams and into domain teams, while creating a domain-agnostic centralized data platform.

Having looked at some practical implementations of real-world data engineering – including DataOps for pipeline deployment and management, examples of real-world data pipelines, and emerging trends and concepts – we will now move on to our final hands-on section of this book.

Hands-on – cleaning up your AWS account

In the hands-on section of *Chapter 1, An Introduction to Data Engineering*, we went through how to create a new AWS account. If you created a new account at that point, and have used that account to work through the exercises in this book, you may want to delete that account, now that you have reached the final chapter of this book. We'll include instructions on how to do that here.

However, if this was your first AWS account, you may decide that you want to keep the account open so that you can continue to explore and learn more about AWS using other resources. If that is you, we'll include some instructions on how to check your account billing to detect which resources are still being charged for.

Reviewing AWS Billing to identify the resources being charged for

In this section, we will go through how to review your AWS billing console to determine which resources you are being charged for:

1. Log in to the AWS Billing console using the following link: <https://console.aws.amazon.com/billing/home>.
2. On the right-hand side of the **Billing & Cost Management Dashboard** page, there will be a visual showing **Month-to-Date Spend by Service**:

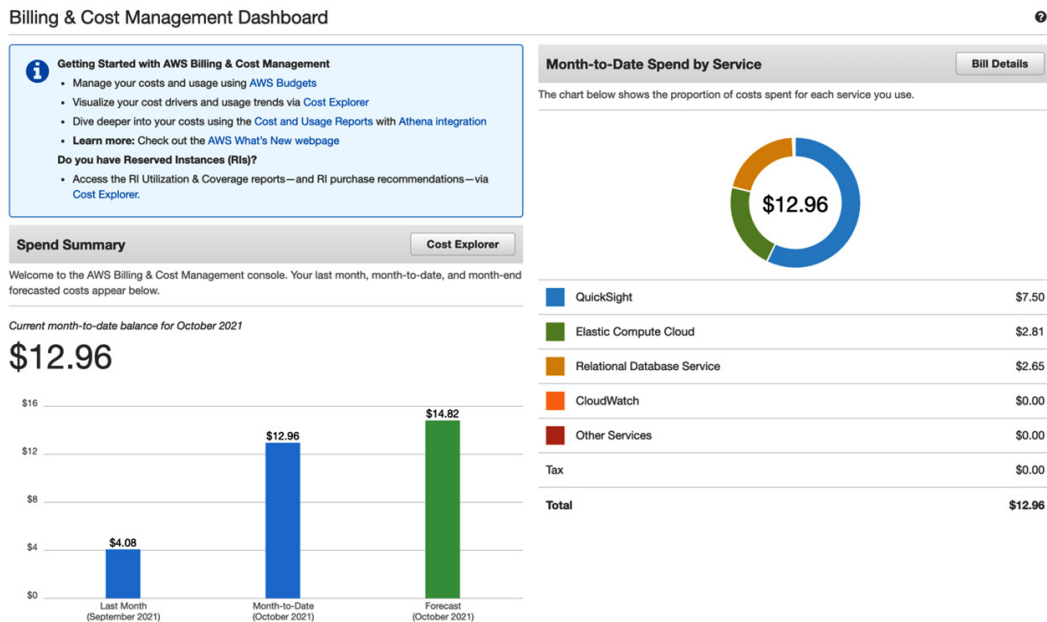


Figure 14.3 – Billing & Cost Management Dashboard

In the preceding screenshot, you can see that I have spent **\$12.96** so far this month, while at the bottom left, you can see that the forecast for the full month is a total of **\$14.82**.

I did not cancel my QuickSight subscription after completing the exercises in *Chapter 12, Visualizing Data with Amazon QuickSight*, and my free 30-day trial ended. If I wanted to cancel my QuickSight subscription now to avoid any future charges after this month, I could follow the instructions in *Canceling your Amazon QuickSight subscription and closing the account* (<https://docs.aws.amazon.com/quicksight/latest/user/closing-account.html>).

3. I can also see charges for **Elastic Compute Cloud** and **Relational Database Service**. I am not sure what these charges relate to, so to investigate this further, I can click on **Bill Details** at the top right, above the pie chart visualization of my spending for this month:

Details			+ Expand All
AWS Service Charges			\$12.96
▶	CloudWatch		\$0.00
▶	Comprehend		\$0.00
▶	Data Transfer		\$0.00
▼	Elastic Compute Cloud		\$2.81
▼	US East (Ohio)		\$2.81
	EBS		\$2.81
	\$0.10 per GB-month of General Purpose SSD (gp2) provisioned storage - US East (Ohio)	28.065 GB-Mo	\$2.81
▶	Glue		\$0.00
▶	Lambda		\$0.00
▶	QuickSight		\$7.50
▶	Redshift		\$0.00
▶	Rekognition		\$0.00
▼	Relational Database Service		\$2.65
▼	US East (Ohio)		\$2.65
	Amazon Relational Database Service Backup Storage		\$2.50
	\$0.095 per RDS additional GB-month of backup storage exceeding free allocation	26.354 GB-Mo	\$2.50
	Amazon Relational Database Service for Aurora MySQL		\$0.15
	USD 0.021 per GB-month of backup storage exceeding free allocation for Aurora MySQL	6.979 GB-Mo	\$0.15

Figure 14.4 – AWS Bill Details view

Using the **Bill Details** view, I can expand the **Elastic Compute Cloud (EC2)** and **Relational Database Service (RDS)** sections for more information.

Here, I can see that the EC2 charges were incurred in the **US-East (Ohio)** region and that the charges relate to **General Purpose SSD provisioned storage**. I can now go to the EC2 console, where I will see that I have four EC2 instances that I have stopped (so they are not incurring any charges) but that each of them has attached disk volumes that continue to incur costs, even when the instances are stopped. If I wanted to ensure that I do not get billed for these in the future, I could terminate the instances, which will permanently delete the volumes.

Looking back at the **Bill Details** screen, I can see that the RDS charges all relate to backups that I have created. Even though I terminated the RDS instances I had previously launched, I chose to create and store a backup copy of the databases on termination, so I will continue to incur costs related to those backups. If I wanted to stop any future billing, I could delete the RDS snapshots if they're no longer needed.

If I canceled my QuickSight subscription, terminated my EC2 instances, and deleted my RDS snapshots, I could continue using my AWS account without incurring additional charges for those items. However, it is strongly recommended that you regularly check the billing console and set billing alarms to alert you of spending above the limit you've set. For more information, see https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/monitor_estimated_charges_with_cloudwatch.html.

Closing your AWS account

If you decide that you want to close your AWS account, you can do so with the following steps.

Before proceeding, make sure that you have read the *Considerations before you close your AWS account* section of the AWS documentation at <https://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/close-account.html>. Now, let's get started:

1. Log into your AWS account as the root user of the account (that is, using the email address and password you registered when you opened the account). Use the following link to log in: <https://console.aws.amazon.com>.

If you're prompted for an IAM username and password, click on the link for **Sign in using root user email**.

2. Enter your root user email address and password when prompted.
3. Open the billing console with the following link: https://console.aws.amazon.com/billing/home#.

- In the top-right corner, select the dropdown next to your account number (or account alias, if set). From this dropdown, select **My account**:

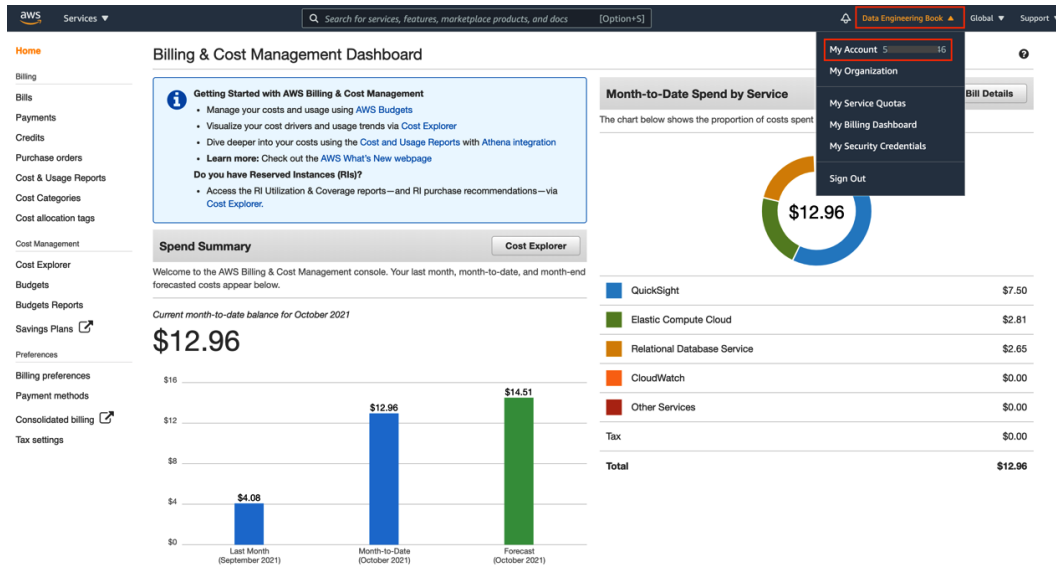


Figure 14.5 – Accessing the My Account screen in the AWS Management Console

- Scroll to the bottom of the **My Account** page. Read and ensure you understand the text next to each checkbox, and if you understand and agree, click the relevant checkboxes. Then, click **Close Account**:

Close Account

I understand that by clicking this checkbox, I am closing my AWS account. The closure of my AWS account serves as notice to AWS that I wish to terminate the AWS Customer Agreement or any other agreement with AWS that governs my AWS account, solely with respect to that AWS account.

Monthly usage of certain AWS services is calculated and billed at the beginning of the following month. If I have used these types of services this month, then at the beginning of next month I will receive a bill for usage that occurred prior to termination of my account. In addition, if I have any active subscriptions (such as a Reserved Instance for which I have elected to pay in monthly installments), then even after my account is closed I may continue to be billed for the subscription until the subscription expires or is sold in accordance with the terms governing the subscription.

I acknowledge that I may reopen my AWS account only within 90 days of my account closure (the "Post-Closure Period"). If I reopen my account during the Post-Closure Period, I may be charged for any AWS services that were not terminated before I closed my account. If I reopen my AWS account, I agree that the same terms will govern my access to and use of AWS services through my reopened AWS account.

If I choose not to reopen my account after the Post-Closure Period, any content remaining in my AWS account will be deleted. For more information, please see the [Amazon Web Services Account Closure page](#).

I understand that after the Post-Closure Period I will no longer be able to reopen my closed account.

I understand that after the Post-Closure Period I will no longer be able to access the Billing Console to download past bills and tax invoices.

If you wish to [download any statements you can do so here](#). Select the month and expand the summary section to download the payment invoices and/or tax documents.

I understand that after the Post-Closure Period I will not be able to create a new AWS account with the email address currently associated with this account.

If you wish to update your e-mail address, [follow the directions here](#).

Close Account

Figure 14.6 – The AWS Close Account confirmation page

- In the pop-up box, click **Close Account** to confirm that you want to close your account.

Subsequently, if you change your mind about closing your account, it may still be possible to reopen your account within 90 days of choosing to close it. To do so, contact AWS support.

Summary

Data engineering is an exciting role to be in and promises to continue to offer interesting challenges, constant learning opportunities, and increasing importance in helping organizations draw out the maximum value that they can from their data assets. And the cloud is an exciting place to build data engineering pipelines.

Also, AWS has a proven track record in listening to their customers and continuing to innovate based on their customer requirements. Things move quickly with AWS services, so hold on tight for the ride.

If you're new to data engineering on AWS, then this book is just the start of what could be a long and interesting journey for you. There is much more to be learned than what could ever be captured in a single book, or even a volume of books. Much of what you will learn will be through practical experience and things you learn on the job, as well as from other data engineers.

But this book, and other books like it, as well as resources such as podcasts, YouTube videos, and blogs, are all useful vehicles along your journey. Let the end of this book be just the end of the first chapter of your learning journey about data engineering with AWS.



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

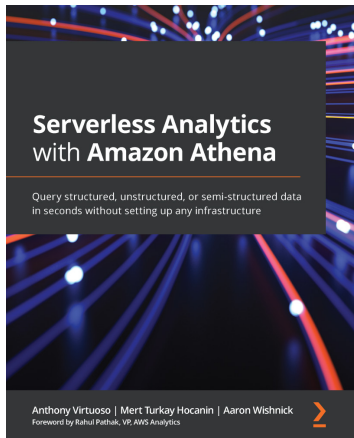
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

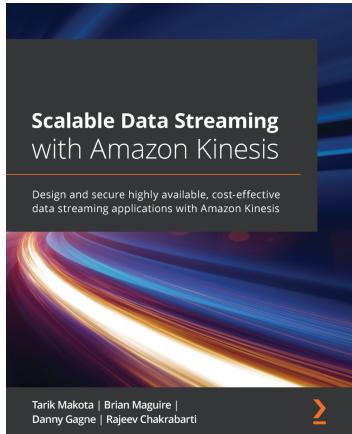


Serverless Analytics with Amazon Athena

Anthony Virtuoso, Mert Turky Hoccanin, Aaron Wishnick

ISBN: 9781800562349

- Secure and manage the cost of querying your data
- Use Athena ML and User Defined Functions (UDFs) to add advanced features to your reports
- Write your own Athena Connector to integrate with a custom data source
- Discover your datasets on S3 using AWS Glue Crawlers
- Integrate Amazon Athena into your applications
- Setup Identity and Access Management (IAM) policies to limit access to tables and databases in Glue Data Catalog
- Add an Amazon SageMaker Notebook to your Athena queries
- Get to grips with using Athena for ETL pipelines



Scalable Data Streaming with Amazon Kinesis

Tarik Makota, Brian Maguire, Danny Gagne, Rajeev Chakrabarti

ISBN: 9781800565401

- Get to grips with data streams, decoupled design, and real-time stream processing
- Understand the properties of KFH that differentiate it from other Kinesis services
- Monitor and scale KDS using CloudWatch metrics
- Secure KDA with identity and access management (IAM)
- Deploy KVS as infrastructure as code (IaC)
- Integrate services such as Redshift, Dynamo Database, and Splunk into Kinesis

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Data Engineering with AWS*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here` to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

- ACID transactions
 - atomicity 212
 - consistency 212
 - durability 212
 - isolation 212
 - on data lake data 429
- Airflow Connections 304
- Airflow Hooks 304
- Airflow Operators 305
- Airflow Sensors 305
- Airflow Tasks 305
- Airplane Health Management (AHM) 171
- Amazon AppFlow
 - about 406
 - data, ingesting from SaaS services 60, 61
- Amazon Athena
 - about 48, 200, 236, 327, 329, 353
 - data, querying with 191
 - file format optimization 330
 - layout optimization 330
 - overview, for SQL queries
 - in data lake 77
 - tips and tricks, to optimize queries 330
- AmazonAthenaFullAccess 328
- Amazon Athena Query Federation
 - queries of external data
 - sources, federating 337
- Amazon Athena Workgroups
 - costs, managing 341
 - creating 344-346
 - data usage controls, enforcing 343
 - governance, managing 341
 - settings, enforcing for
 - group of users 342
- Amazon Comprehend
 - about 208
 - insights, extracting from text 402, 403
 - reviews, reviewing 407
 - solution, testing 413-415
- Amazon Comprehend Medical 403
- Amazon Database Migration Service (DMS) 44, 52-54, 209
- Amazon DataSync
 - overview for ingesting, from
 - on-premise storage 62, 63
- Amazon DynamoDB 78
- Amazon EC2 238
- Amazon Elastic File System (EFS) 395

- Amazon Elastic Map Reduce (EMR)
 - about 69, 168, 197, 273
 - for Hadoop ecosystem processing 69, 70
- Amazon EventBridge
 - configuring 319
- Amazon Forecast 405
- Amazon Fraud Detector 406
- Amazon GuardDuty 112
- Amazon HealthLake 257
- Amazon Kinesis
 - overview, for streaming
 - data ingestion 54
 - versus Amazon Managed Streaming for Kafka (MSK) 171-174
- Amazon Kinesis Agent 55
- Amazon Kinesis Data Analytics 58
- Amazon Kinesis Data Generator (KDG)
 - configuring 187-190
- Amazon Kinesis Data Streams 56, 57
- Amazon Kinesis Firehose 44, 55
- Amazon Kinesis Producer
 - Library (KPL) 55
- Amazon Kinesis services
 - Kinesis Data Analytics 55
 - Kinesis Data Firehose 54
 - Kinesis Data Streams 54
 - Kinesis Video Streams 55
- Amazon Kinesis Video Streams 58
- Amazon Lake Formation
 - Governed Tables 429
- Amazon Lex 389
- Amazon Machine Images (AMIs) 393
- Amazon Macie 112
- Amazon Managed Streaming for Apache Kafka (MSK)
 - about 56
 - overview, for streaming
 - data ingestion 59
- Amazon Managed Workflows for Apache Airflow 75, 76
- Amazon Managed Workflows for Apache Airflow (MWAA)
 - about 299, 303, 309
 - cons 306
 - pros 305
- Amazon Personalize 407
- Amazon QuickSight
 - about 82, 236, 255, 353
 - core concepts 361
 - data visualization 81, 82
 - embedded dashboards 375
 - embedding, for registered QuickSight users 375
 - embedding, for unauthenticated users 375
 - enterprise edition 361, 362
 - overview 233-235
 - standard edition 361
 - visual types 368
- Amazon QuickSight Autonarratives 373
- Amazon QuickSight ML Insights 372
- Amazon Redshift 25, 52, 198, 255
- Amazon Redshift cluster
 - nodes, types 29
- Amazon Redshift Spectrum
 - for data lakehouse architectures 78-81
 - for data warehousing 78-81
- Amazon Rekognition 208, 403, 404
- Amazon Rekognition Video 403
- Amazon Relational Database Service (RDS) 168
- Amazon S3 bucket
 - creating 50, 85
 - reference link 50
- Amazon S3 Bucket Keys
 - reference link 111

- Amazon S3 data events
 - configuring 319, 320
- Amazon S3 data lake 53
- Amazon S3 Glacier (S3 Glacier) 253
- Amazon S3 Glacier Deep Archive 253
- Amazon S3 Intelligent-Tiering 254
- AmazonS3ReadOnlyAccess 328
- Amazon S3 Standard (S3 Standard) 254
- Amazon S3 Standard-
 - Infrequent Access 254
- Amazon SageMaker 240, 390
- Amazon SageMaker Clarify 394
- Amazon SageMaker Data Wrangler 394
- Amazon SageMaker Ground Truth 394
- Amazon Simple Notification
 - Service (SNS) 344
- Amazon Simple Queue Service (SQS) 407
- Amazon Simple Storage
 - Service (Amazon S3)
 - about 10
 - external tables, creating for
 - querying data 282-286
 - sample data, uploading to 276
- Amazon SQS message queue
 - setting up 407, 408
- Amazon SQS quota limits 427, 428
- Amazon States Language (ASL) 73, 306
- Amazon Textract
 - text, extracting from
 - documents 400, 401
- Amazon Transcribe
 - about 208
 - speech, converting into text 399
- Amazon Transcribe Call Analytics 400
- Amazon Transcribe Medical 400
- Amazon Transfer Family
 - about 61
 - for ingestion, with FTP/
 - SFTP protocols 62
- Amazon Translate 399
- Amazon VPC
 - enhancements 425
- American National Standards
 - Institute (ANSI) 329
- analysis
 - creating 379-384
- analytics
 - extending, with data warehouses/
 - data marts 252
- anonymized data 102
- Apache 6
- Apache Airflow
 - about 75
 - as open source orchestration
 - solution 303
- Apache Airflow pipelines
 - creating, core concepts 304
- Apache Flink code 55
- Apache HBase 69
- Apache Hudi 213, 429
- Apache Iceberg 213
- Apache Kafka 8
- approximate aggregate functions
 - using 335
- artificial intelligence (AI)
 - about 208, 388
 - for fraud detection and
 - personalization 406
 - for ML-powered forecasts 405
 - for organizations 389
 - for unstructured speech and text 399
 - metadata, extracting from
 - images and video 403

- Athena Federated Query
 - about 78
 - external data sources, querying 338, 339
- Athena settings
 - configuring 344, 345
- Athena views 211
- authentication 103
- authorization 104
- AutoGraph
 - about 368
 - for automatic graphing 369
- automatic speech recognition (ASR) 400
- AWS
 - about 25
 - data lakehouse, building on 47, 48
 - Python, running in 238
 - R, running in 238
- AWS account
 - accessing 11, 15-18
 - cleaning up 434
 - closing 437, 438
 - creating 11-15
- AWS account root user 113
- AWS AI services 398, 399
- AWS Artifact
 - reference link 100
- AWS Billing
 - reviewing 435-437
- AWS CloudFormation (CFN) service 420
- AWS CloudTrail
 - configuring 319
- AWS CloudWatch 73
- AWS CMDB connector 340
- AWS CodeCommit 421
- AWS Command-Line Interface (CLI)
 - configuring 49
 - installation link 49
 - installing 49
- AWS Database Migration Service (DMS) 166
- AWS Data Exchange service 207
- AWS Data Pipeline
 - ETL, managing between data sources 300
- AWS Data Wrangler library
 - used, for creating Lambda layer 83, 84
- AWS Deep Learning AMIs
 - reference link 393
- AWS Deep Learning Containers
 - reference link 393
- AWS documentation, usage and cost
 - reference link 11
- AWS Glue
 - about 166, 197
 - for serverless Spark processing 65
 - use cases 166, 167
- AWS Glue catalog 108
- AWS Glue console
 - table properties 108
- AWS Glue Crawlers 69
- AWS Glue DataBrew
 - about 237, 238
 - datasets, configuring for 242, 243
 - data transformations, creating 242
- AWS Glue data catalog 66-68, 328
- AWS Glue jobs
 - about 406
 - creating, with AWS Lake Formation 167
- AWS Glue Python Shell 238
- AWS Glue Studio
 - about 198, 199
 - datasets, joining with 214
 - denormalization transform, configuring with 217-222
 - used, for creating transform job 224-227
 - versus AWS Glue DataBrew 242

- AWS Glue Workflows
 - error handling 302
 - event-driven approach 302
 - Glue resources, orchestrating 301
 - job progress, monitoring 302
 - overview, for orchestrating
 - Glue components 71-73
 - triggering 302, 303
 - AWS Identity and Access
 - Management (IAM) 113
 - AWS Key Management
 - Service (KMS) 111
 - AWS Lake Formation
 - about 108
 - fine-grained permissions, managing 123
 - permissions management 117, 118
 - using, to manage data lake access 116
 - AWS Lake Formation console
 - functionalities 109
 - table properties 108
 - AWS Lake Formation governed tables 212
 - AWS Lambda
 - about 238
 - for light transformations 64, 65
 - AWS Lambda function
 - triggering, in S3 bucket 83
 - AWS managed policies 114
 - AWS Management Console
 - URL 214
 - AWS ML services 392, 394
 - AWS Premium Support
 - reference link 15
 - AWS S3 API calls 62
 - AWS SDK 55
 - AWS services
 - for data encryption 110
 - for security monitoring 110
 - identity, managing 113
 - permissions, managing 113
 - AWS services, for big data pipelines
 - orchestration 71
 - AWS services, for data consumption 77
 - AWS services, for data ingestion 52
 - AWS services, for data transformation 64
 - AWS Snowball Edge 63
 - AWS Snowcone 63
 - AWS Snow family of devices
 - for large data transfers 63
 - AWS Snowmobile 63
 - AWS Step Functions
 - about 73, 406
 - cons 309
 - data pipeline, orchestrating 311
 - for complex workflows 73-75
 - pros 308
 - serverless orchestration solution 306
 - AWS tools
 - for business users 233
 - AWS tools, for data analysts
 - about 236
 - Amazon Athena 236
 - AWS Glue DataBrew 237, 238
 - AWS tools, for data scientists
 - about 239
 - SageMaker Clarify 241
 - SageMaker Data Wrangler 240, 241
 - SageMaker Ground Truth 240
 - Azure DevOps (ADO) 421
 - Azure Synapse 25
 - Azure Synapse Analytics 46
- ## B
- bar charts 369
 - batch-based ingestion 429

- big data
 - rising, as corporate asset 4, 5
 - big data architect 10
 - big data processing 24
 - BitBucket 421
 - BI tools 354
 - bookmarks 167
 - bucketing 333
 - business applications 138
 - business catalog 107
 - Business Intelligence (BI) 140
 - business unit (BU) 202
 - business use case transforms
 - about 205
 - data denormalization 205, 206
 - data enriching 207
 - metadata, extracting from
 - unstructured data 208, 209
 - pre-aggregation transform 207
 - business users
 - about 138, 232
 - AWS tools 233
 - needs, meeting with data
 - visualization 232
- ## C
- California Consumer Privacy Act (CCPA) 99
 - California Privacy Rights Act (CPRA) 99
 - cataloging and search layer 43
 - CDC file
 - Delete 209
 - Insert 209
 - Update 209
 - Centralized Operations Center
 - dashboard 389
 - Change Data Capture (CDC)
 - about 53, 168
 - example 209
 - modern approaches 211
 - traditional approaches 210, 211
 - working with 209, 210
 - chatbot 389
 - Chief Information Security Officer (CISO) 100
 - cloud
 - benefits, when building big data
 - analytic solutions 10, 11
 - Cloudera 24, 340
 - cloud object stores
 - adopting 26
 - CloudWatch logs 340
 - CloudWatch metrics connector 340
 - cold data 252, 253
 - Collibra Data Catalog 107
 - columnar data storage 30-32
 - Comma Separated Values (CSV) files 159
 - complex data environments
 - managing, with DataOps 420
 - complex SQL queries
 - running 288-291
 - compound sort key 261
 - Comprehend permissions
 - adding, for IAM role 412
 - consumption layer 45
 - continuous delivery (CD) 421
 - continuous integration (CI) 421
 - core concepts, for creating Apache
 - Airflow pipelines
 - about 304
 - Airflow Connections 304
 - Airflow Hooks 304
 - Airflow Operators 305
 - Airflow Sensors 305

- Airflow Tasks 305
- directed acyclic graph (DAG) 304
- costs
 - managing, with Amazon Athena Workgroups 341
- Create Table As Select (CTAS) 331
- custom connectors 340
- customer-managed policies 114
- customer relationship management (CRM) systems 212
- custom visuals types 370

D

- dashboards 234
- data
 - cataloging, to avoid data swamp 105
 - enriching 207
 - exporting, from Redshift
 - to data lake 274
 - feeding, into warehouse 37-40
 - ingesting 140, 364
 - ingesting, from database 167, 168
 - ingesting, from relational database 165
 - loading, into Amazon Redshift cluster 275
 - loading, into data marts 146
 - moving, between data lake and Redshift 272
 - preparing 364
 - product thinking convergence 432
 - querying, with Amazon Athena 191
 - querying, with external tables
 - in Amazon S3 282-286
 - self-serve platform design
 - convergence 432
 - data analysts
 - about 9, 10, 139, 235
 - AWS tools 236
 - needs, meeting with structured reporting 235, 236
 - data analyst team 151
 - data analytics 418
 - data assets 27
 - database
 - about 23
 - Lake Formation permissions, activating for 124-126
 - Database Migration Service 153
 - Databricks 197
 - Databricks Delta Lake
 - about 46, 213, 429
 - reference link 213
 - data cataloging 144
 - data catalogs
 - about 106
 - business catalog 107
 - technical catalog 106
 - data cleansing 203
 - data compression 30-32
 - data consumers
 - about 196, 229
 - identifying 138-140
 - types 231, 232
 - Data Definition Language (DDL) 329
 - data democratization
 - impact 230, 231
 - data denormalization 143, 205
 - data engineer 7, 8, 10, 229
 - data engineering
 - environments 419
 - teams 419
 - data governance 98
 - data gravity 231

- data infrastructure
 - as source control-managed
 - code 420, 421
- data, ingesting with AWS DMS
 - data, querying with Athena 185
 - demo data, loading with Amazon EC2 instance 177, 178
 - DMS settings, configuring 181-184
 - full load, performing from
 - MySQL to S3 181-184
 - Glue crawler, creating 185
 - IAM policy, creating 179-181
 - IAM role, creating 179-181
 - MySQL database instance, creating 175, 176
 - steps 174
- data ingestion
 - optimizing, in Redshift 272-274
- data ingestion, from database
 - best approach 169, 170
- data lake architecture 26
- data lakehouse
 - about 26
 - building, on AWS 47, 48
 - implementing 46
- data lake logical architecture
 - about 42, 45
 - cataloging and search layer 43
 - consumption layer 45
 - ingestion layer 44
 - processing layer 44
 - storage layer 42, 43
 - storage zones 42, 43
- data lakes
 - about 7, 40
 - access, managing with AWS
 - Lake Formation 116
 - building, to tame variety and volume of big data 40, 41
 - clean/transform zone 43
 - curated/enriched zone 43
 - landing/raw zone 43
- data management
 - evolution, for analytics 22
- Data Manipulation Language (DML) 329
- data marts
 - about 27, 36
 - analytics, extending 252
 - creating, formats 36
- data mesh architecture
 - implementations 433
- DataOps
 - about 420
 - complex data environments, managing 420
- data optimizations
 - identifying 142
- data partitioning
 - about 143
 - optimizing with 202, 203
- data pipeline
 - about 295
 - architecting 149
 - as source control-managed
 - code 420, 421
 - failure retry strategies 299
 - failures of step, handling 298
 - reasons, for failure 298, 299
 - triggering, to run 297
- data pipeline architecture 134
- data pipeline orchestration 295
- data pipeline orchestration tool
 - selecting 309, 310
- data platforms 431

- data preparation transformations
 - about 200
 - data cleansing 203
 - data partitioning, optimizing
 - with 202, 203
 - file format, optimizing 201
 - PII data, protecting 200
- Data Processing Units (DPUs) 66
- data protection 100
- Data Protection Officer (DPO) 100
- data quality checks 143
- data regulatory
 - requirements 99, 100
- data science team 151, 239
- data scientist
 - about 8, 9, 139
 - AWS tools 239
 - needs, meeting 239
- data security 98
- dataset
 - configuring, for AWS Glue
 - DataBrew 242, 243
 - loading 376-379
 - partitioning 331, 332
- datasets, joining with AWS Glue Studio
 - about 214
 - curated zone, creating 214
 - data lake zone, creating 214
 - denormalization transform,
 - configuring 217-222
 - denormalization transform
 - job, finalizing 222, 224
 - IAM role, creating for Glue job 215-217
 - transform job, creating for joining
 - streams and film data 224-227
- datasets, preparing in QuickSight
 - versus ETL outside of
 - QuickSight 365, 366
- data sources
 - about 158, 196
 - identifying 140
- data standardization 143
- data swamp
 - avoiding 106
- data transformation
 - about 193, 196
 - as part of data pipeline 196
 - baking example 195
 - cooking example 195
 - creating, with AWS Glue DataBrew 242
 - identifying 142
 - raw data, making valuable 194
- data transformation tasks, data cleansing
 - column data type, changing 204
 - consistent column names, ensuring 204
 - duplicate records, removing 204
 - missing values, providing 204
 - standard column format, ensuring 204
- data transformation tools
 - about 196
 - Apache Spark 196
 - GUI-based tools 199
 - Hadoop 197
 - MapReduce 197
 - SQL 198
- data type, for columns
 - Boolean type 267
 - character types 264
 - datetime types 266
 - decimal type 265
 - floating-point types 266
 - HLLSKETCH type 267
 - integer types 265
 - numeric types 265
 - SUPER type 267

- data value 164
- data variety 159
- data velocity 163
- data veracity 164
- data virtualization 338
- data visualization
 - benefits 356
 - for maximum impact 355
 - needs of business users, meeting 232
- data visualization developer 10
- data visualization, uses
 - about 356
 - data, over geographic area 358, 359
 - heat maps, to represent intersection of data 359, 360
 - trends, over time 356, 357
- data volume 163
- data warehouses
 - about 23, 27
 - analytics, extending 252
 - dimensional modeling 32-36
 - using, as data lake 256
 - using, for real-time and record-level use cases 257
 - using, as transactional datastore 256
- DC2 nodes 262
- decentralized data engineering teams 431
- denormalization 206
- denormalization transform
 - configuring, with AWS Glue Studio 218-222
 - finalizing, for writing to S3 222-224
- directed acyclic graph (DAG)
 - about 296, 304
 - example 296
- disk access 261

- disk seek 261
- distributed storage 29, 30
- domain-orientated data
 - decomposition 431
- domain-orientated data ownership 431
- Dredge 425
- Druid 425
- DynamoDB 211
- DynamoDB connector 340

E

- ECS 197
- EKS 197
- encryption 101
- encryption at rest 101
- encryption in transit 101
- Enterprise Data Warehouse (EDW) 27
- event-based pipelines 297
- EventBridge rule
 - creating, to trigger Step Function state machine 320-322
- event-driven data orchestration pipeline testing 322, 323
- ever-growing datasets
 - challenges 5, 6
- EXPLAIN statement 334
- exponential backoff 299
- external data sources
 - querying, with Athena Federated Queries 338, 339
- external tables
 - creating, for querying data in Amazon S3 282-286
- Extract-Load-Transform (ELT) 37, 39
- Extract-Transform-Load (ETL) 37, 38
- ExxonMobil 5

F

Facebook 329
 Federated Query 338
 file-based optimizations 333
 file format optimizations 142, 201
 fine-grained permissions
 managing, with AWS Lake
 Formation 123
 Fivetran 199
 forecasting 391

G

General Data Protection
 Regulation (GDPR) 99
 geospatial charts 358, 369
 GitLab 421
 Glue crawler 330
 Glue DataBrew
 about 199
 job, creating 248, 249
 project, creating 243, 244
 recipe, building 245-247
 Glue Data Catalog
 ingested data, adding to 190, 191
 Glue job
 configuring 215-217
 Glue Python Shell 65
 Google BigQuery 25, 340
 Google BigTable 423
 Google Compute Cloud (GCP) 25
 Google Voice
 URL 12
 governance
 managing, with Amazon Athena
 Workgroups 341

governed tables 212
 GUI-based tools 199

H

Hadoop 6, 24, 197
 Hadoop Distributed File System
 (HDFS) 24, 197
 Hadoop distributions 24
 Hadoop MapReduce 429
 HBase 197
 Health Insurance Portability and
 Accountability Act (HIPAA) 100
 heat maps 369
 high-performance data
 warehouse, designing
 about 262
 data type for columns, selecting 263
 optimal Redshift node type,
 selecting 262
 optimal table distribution style
 and sort key, selecting 263
 optimal table type, selecting 268
 Hive 24, 197
 Hive Metastore 68, 106
 Hive partitioning 202
 horizontal scaling 30
 Hortonworks 24
 hot data 255
 houses
 architecting 135
 hyperparameter tuning 397

I

IAM permissions
 new user, creating 119-123

- IAM policy
 - creating, for Lambda function 86-88
- IAM role
 - about 114
 - Comprehend permissions, adding for 412
 - creating, for Lambda function 86-88
 - for Redshift 277, 279
- IAM User 113
- IAM User Groups 114
- IBM 24, 329
- Identity and Access Management (IAM) 15
- identity-based policies
 - AWS managed policies 114
 - customer-managed policies 114
 - inline policies 114
- image recognition 392
- Informatica 199
- Informatica Enterprise Data Catalog 107
- Infrastructure as Code (IaC) 420
- ingested data
 - adding, to Glue Data Catalog 190, 191
- ingestion layer 44
- inline policies 114
- interleaved sort key 261
- Internet of Things (IoT) 203

J

- JDBC connector 340
- JDBC database connection 78

K

- Key Performance Indicator (KPI) 370
- Kinesis Agent 153
- Kinesis Client Library (KCL) 57

- Kinesis Data Generator 350
- Kinesis Data Streams cluster 425
- Kinesis Firehose
 - configuring, for streaming delivery to Amazon S3 186, 187

L

- Lake Formation 48, 212
- Lake Formation permissions
 - activating, for database 124-126
 - activating, for table 124-126
 - configuring 118
 - granting 127, 128
- lake house architecture 26
- Lambda function
 - adding, as trigger for SQS message queue 412
 - configuring, to trigger by S3 upload 93, 95
 - creating 88-93, 311
 - creating, for calling Amazon Comprehend 408-410
 - IAM policy and role, creating 86-88
 - using, to determine file extension 311, 312
 - using, to randomly generate failures 313
- Lambda layer
 - creating, with AWS Data Wrangler library 83, 84
- line charts 369
- Load-Transform sequence 37
- local Redshift table
 - schema, creating 287

M

- machine learning (ML)
 - about 239, 387
 - for organizations 389
- Managed Workflows for Apache
 - Airflow (MWAA) 75
- manifest 297
- manifest files
 - using, as pipeline triggers 297, 298
- MapR 24
- map-reduce 24
- MapReduce 6, 197
- marketing specialists 150
- Marketo 60
- Massively Parallel Processing
 - (MPP) 29, 30
- materialized view 271
- Matillion 199
- metadata
 - extracting, from unstructured data 208, 209
- Microsoft Azure 25
- ML and AI, use cases
 - about 391
 - forecasting 391
 - image recognition 392
 - natural language processing (NLP) 392
 - personalization 392
- ML models, needs
 - meeting 239
- ML-powered anomaly detection 374
- ML-powered forecasting 374
- modern approaches, CDC data
 - about 211
 - Apache Hudi 213
 - Apache Iceberg 213

- AWS Lake Formation
 - governed tables 212
- Databricks Delta Lake 213
- transactional data lakes 211
- multi-cloud strategy 430
- Multi-Factor Authentication
 - (MFA) 15, 104
- Multi-Listing Service (MLS) 62
- MWAA environment, components
 - meta database 76
 - scheduler 76
 - web server 76
 - worker/ execute tasks 76

N

- natural language processing (NLP) 392
- Network Attached Storage (NAS) 37
- Network File System (NFS) 62

O

- ODBC database connection 78
- Online Analytical Processing (OLAP) 78
- Online Analytics Processing (OLAP) 206
- Online Transaction Processing
 - (OLTP) 30, 206
- optimal table type
 - data caching, with Redshift
 - materialized views 271
 - external tables, for querying data
 - in Amazon S3 269, 270
 - selecting 268
 - storage and compute, coupling 268
 - temporary staging table, for loading data into Redshift 270, 271

- optimized file formats
 - raw source files, transforming
 - to 330, 331

- optimized SQL queries
 - writing 334

- Oracle 340, 428

- organizational policies
 - for metadata capture 107

P

- Panoply 199

- Parquet 201, 295, 330

- Parquet files

 - about 201

 - benefits 201

- partitioning 203

- partition keys 223

- partition projection 333

- Payment Card Industry Data Security Standard (PCI DSS) 100

- permissions management

 - before Lake Formation 117

 - with AWS Lake Formation 117, 118

- per query data usage control 343

- personal data 101

- Personal Data Protection

 - Bill (PDP Bill) 99

- personalization 392

- Personally identifiable information (PII) 100, 200

- Fig 69, 197

- PII data

 - protecting 200

- pipeline orchestration

 - core concepts 294

- pipelines

 - architecting 135

 - designing 137

 - options for orchestrating, in AWS 299

- pipeline triggers

 - manifest files, using as 297, 298

- Polybase 46

- pre-aggregation transform 207, 208

- pre-built connectors 340

- Presto 8, 24, 69

- Presto SQL analytics engine 329

- processing layer 44

- profile

 - creating 49

- proof of concept test 407

- Protection of Personal Information

 - Act (POPIA) 99

- pseudonymized data 102

- public cloud infrastructure

 - adopting, benefits 25

- Python

 - about 65, 236

 - running, in AWS 238

Q

- Query Federation 338

- QuickSight account

 - setting up 376-379

- QuickSight Analyses 367

- QuickSight pricing page

 - reference link 362

R

- R

 - about 236

 - running, in AWS 238

- RA3 nodes 262

- raw data 330

- raw source files
 - transforming, to optimized file formats 330, 331
 - real-world data pipelines
 - examples, examining 422
 - Redis connector 340
 - Redshift
 - data, exporting to data lake 274
 - data ingestion, optimizing 272-274
 - IAM roles 277, 279
 - Zone Maps 261
 - Redshift architecture
 - data distribution, across slices 258-261
 - review 258
 - storage 258
 - Redshift cluster
 - creating 280-282
 - Redshift Managed storage 268
 - Redshift Spectrum 47, 275
 - regular expressions
 - using 336, 337
 - relational database
 - data, ingesting from 165
 - Relational Database Management Systems (RDBMSes) 159
 - replication instances 166
 - retry backoff rate 299
 - root user 15
 - row chunks/groups 31
 - row-oriented physical data layout 30
 - RStudio 238
- S**
- S3 bucket
 - AWS Lambda function, triggering 83
 - SaaS services
 - Amazon AppFlow, overview for ingesting data 60, 61
 - SageMaker
 - in ML build phase 395
 - in ML deployment and management phase 397, 398
 - in ML preparation phase 394
 - in ML training and tuning phase 396, 397
 - SageMaker Autopilot 395
 - SageMaker Clarify 241
 - SageMaker Data Wrangler 240, 241
 - SageMaker Experiments 397
 - SageMaker Ground Truth 240
 - SageMaker JumpStart 396
 - SageMaker Model Monitor 398
 - SageMaker Studio 394
 - SageMaker Studio notebooks 395
 - sample data
 - uploading, to Amazon S3 276
 - sample Step Function state
 - machine 307, 308
 - sandbox 343
 - sandbox account 11
 - schedule-based pipelines 297
 - schema
 - creating, for local Redshift table 287
 - schema evolution 211
 - select * 335
 - semi-structured data 160, 162
 - serverless ETL processing 65, 66
 - Server Message Block (SMB) 62
 - Server Side Encryption - KMS (SSE-KMS) 111
 - Service Control Policy (SCP) 112
 - ServiceNow 434
 - SHA-256 hash 200

- shared-nothing architecture 30
- simple QuickSight visualization
 - creating 376
- Snowflake 25, 198, 340
- snowflake schema 34-36
- SNS topic
 - creating 313, 314
- Sonar 425
- source code repository 421
- Spark 6, 8, 24, 196, 332, 429
- Spark GraphX 197, 429
- Spark ML 197, 429
- Spark SQL 197, 198
- Spark Streaming 66, 197, 429
- specialized ML projects
 - about 389
 - early detection, of diseases 390
 - medical clinical decision
 - support platform 389
 - sports safety 390
- SPICE capacity
 - managing 363, 364
- Spotify Wrapped feature 423
- SQL queries
 - running 347-351
- SQL Workbench 78
- star schema 34
- Step Function state machine
 - creating 314-318
- storage layer 42, 43
- streaming data
 - ingesting 171, 186
- streaming files
 - ingesting and processing, at Netflix scale 424, 425
- streaming ingestion 429
- structured data 159, 160

- Structured Query Language (SQL) 77, 198, 329
- structured reporting
 - needs of data analysts, meeting 235, 236
- Super-fast, Parallel, In-memory, Calculation Engine (SPICE) 362, 363

T

- tables
 - as visuals 371
 - Lake Formation permissions,
 - activating for 124-126
- Tab Separated Values (TSV) files 159
- Tabular 213
- Talend 199
- technical catalog 106
- terabytes (TB) 158
- Terradata 340
- test/quality assurance (QA)
 - environment 419
- Tez 197
- time travel 211
- tokenization system 102
- traditional approaches, CDC
 - data 210, 211
- transactional data lakes 211
- transform job
 - creating, for joining streams and film data 224-227
- Transform-Load sequence 37
- Transport Layer Security (TLS) 101
- Trianz 340

U

- unstructured data
 - about 24, 162, 399
 - storing 257
- upsert 210
- user
 - creating, with IAM permissions 119-123

V

- Virtual Private Cloud (VPC) 424
- visuals
 - creating 367
 - sharing 367
- visual types, Amazon QuickSight
 - about 368, 372
 - AutoGraph for automatic graphing 369
 - bar charts 369
 - custom visual types 370
 - geospatial charts 369
 - heat maps 369
 - Key Performance Indicators (KPIs) 370, 371
 - line charts 369
 - tables 371
- VPC Flow Logs
 - about 424
 - enriching, with application information 425

W

- warehouse
 - data, feeding 37-40
- warm data 253, 254
- WHERE clause 332

- whiteboard architecture
 - for project Bright Light 155
- whiteboarding
 - as information-gathering tool 136
- whiteboarding data ingestion 141, 142
- whiteboarding data sources 141, 142
- whiteboarding data
 - transformation 144, 145
- whiteboarding session
 - conducting 137, 138
 - wrapping up 147, 148
- whiteboard notes
 - for project Bright Light 156
- Workgroup data usage controls 344
- Workgroups
 - switching 347-351

Y

- Yahoo 6
- Yarn 197

Z

- zettabytes (ZB) 158