669FD03CAE074

1ST EDITION

Databricks Certified Associate Developer for Apache Spark Using Python

The ultimate guide to getting certified in Apache Spark using practical examples with Python

SABA SHAH

Foreword by Rod Waltermann, Distinguished Engineer, Chief Architect, Cloud and AI Software, Lenovo

Databricks Certified Associate Developer for Apache Spark Using Python

The ultimate guide to getting certified in Apache Spark using practical examples with Python

Saba Shah

‹packt›

Databricks Certified Associate Developer for Apache Spark Using Python

Group Product Manager: Kaustubh Manglurkar
Publishing Product Manager: Chayan Majumdar
Book Project Manager: Hemangi Lotlikar
Senior Editor: Shrishti Pandey
Technical Editor: Kavyashree K S
Copy Editor: Safis Editing
Proofreader: Shrishti Pandey
Indexer: Pratik Shirodkar
Production Designer: Ponraj Dhandapani
Senior DevRel Marketing Coordinator: Nivedita Singh

First published: May 2024

Production reference: 1160524

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN: 978-1-80461-978-0

www.packtpub.com

To my parents, Neelam Khalid (late) and Syed Khalid Mahmood, for their sacrifices and for exemplifying the power of patience and determination. To my loving husband, Arslan Shah, for being by my side through the ups and downs of life and being my support through it all. To Amna Shah for being my sister and friend. To Mariam Wasim for reminding me what true friendship looks like.

– Saba Shah

Foreword

I have known and worked with Saba Shah for several years. Saba's journey with Apache Spark began about 10 years ago. In this book, she will guide readers through the experiences she has gained on her journey.

In today's dynamic data landscape, proficiency in Spark has become indispensable for data engineers, analysts, and scientists alike. This guide, meticulously crafted by seasoned experts, is your key to mastering Apache Spark and achieving certification success.

The journey begins with an insightful overview of the certification guide and exam, providing invaluable insights into what to expect and how to prepare effectively. From there, Saba delves deep into the core concepts of Spark, exploring its architecture, transformations, and the myriad of applications it enables.

As you progress through the chapters, you'll gain a comprehensive understanding of Spark DataFrames and their operations, paving the way for advanced techniques and optimization strategies. From adaptive query execution to structured streaming, each topic is meticulously dissected, ensuring you gain a thorough grasp of Spark's capabilities.

Machine learning enthusiasts will find a dedicated section on Spark ML, empowering them to harness the power of Spark for predictive analytics and model development. Additionally, two mock tests serve as the ultimate litmus test, allowing you to gauge your readiness and identify areas for improvement.

Whether you're embarking on your Spark journey or seeking to validate your expertise with certification, this guide equips you with the knowledge, tools, and confidence needed to excel. Let this book be your trusted companion as you navigate the complexities of Apache Spark and embark on a journey of continuous learning and growth.

With Saba's words, step-by-step instructions, screenshots, source code snippets, examples, and links to additional sources of information, you will learn how to continuously enhance your skills and be well-equipped to be a certified Apache Spark developer.

Best wishes on your certification journey!

Rod Waltermann

Distinguished Engineer

Chief Architect Cloud and AI Software

Lenovo

Contributors

About the author

Saba Shah is a data and AI architect and evangelist, with a wide technical breadth and deep understanding of big data and machine learning technologies. She has experience leading data science and data engineering teams at Fortune 500 firms as well as start-ups. She started her career as a software engineer but soon transitioned to big data. She is currently a solutions architect at Databricks and works with enterprises, building their data strategy and helping them create a vision for the future with machine learning and predictive analytics. She currently resides in Research Triangle Park, North Carolina. In this book, she shares her expertise to empower you in the dynamic world of Spark.

About the reviewers

Aviral Bhardwaj is a professional with six years of experience in the big data domain, showcasing expertise in technologies such as AWS and Databricks. Aviral has collaborated with companies including Knowledge Lens, ZS Associates, Amgen Inc., AstraZeneca, Lovelytics, and FanDuel as a contractor, and he currently works with GMG Inc. Furthermore, Aviral holds certifications as a Databricks Certified Spark Associate, Data Engineer Associate, and Data Engineering Professional, demonstrating a deep understanding of Databricks.

Rakesh Dey is a seasoned data engineer with eight years of experience in total and six years of experience in different big data technologies, such as Spark, Hive, and Impala. He has extensive knowledge of the Databricks platform to build any end-to-end ETL project implementation. He has worked on different projects with new technologies and helped customers to achieve performance and cost optimization relative to on-premises solutions. He has different Databricks certifications from the intermediate to professional levels. He currently works at Deloitte.

Table of Contents

5

Advanced Operations and Optimizations in Spark 79

Part 5: Mock Papers

9

10

Preface

Welcome to the comprehensive guide for aspiring developers seeking certification in Apache Spark with Python through Databricks.

In this book, *Databricks Certified Associate Developer for Apache Spark Using Python*, I have distilled years of expertise and practical wisdom into a comprehensive guide to navigate the complexities of data science, AI, and cloud technologies and help you prepare for Spark certification. Through insightful anecdotes, actionable insights, and proven strategies, I will equip you with the tools and knowledge needed to thrive in an ever-evolving technological landscape of big data and artificial intelligence.

Apache Spark has emerged as the go-to framework to process large-scale data, enabling organizations to extract valuable insights and drive informed decision-making. With its robust capabilities and versatility, Spark has become a cornerstone in the toolkit of data engineers, analysts, and scientists worldwide. This book is designed to be your comprehensive companion on the journey to mastering Apache Spark with Python, providing a structured approach to understanding the core concepts, advanced techniques, and best practices for leveraging Spark's full potential.

This book is meticulously crafted to guide you on the journey to becoming a certified Apache Spark developer. With a focus on certification preparation, I offer a structured approach to mastering Apache Spark with Python, ensuring that you're well-equipped to ace the certification exam and validate your expertise.

Who this book is for

This book is tailored for individuals aspiring to become certified developers in Apache Spark using Python. Whether you're a seasoned data professional looking to validate your expertise or a newcomer eager to delve into the world of big data analytics, this guide caters to all skill levels. From beginners seeking a solid foundation in Spark to experienced practitioners aiming to fine-tune their skills and prepare for certification, this book serves as a valuable resource for anyone passionate about harnessing the power of Apache Spark.

Whether you're aiming to enhance your career prospects, validate your skills, or secure new opportunities in the data engineering landscape, this guide is tailored to meet your certification goals. With a focus on exam preparation, we provide targeted resources and practical insights to ensure your success in the certification journey.

The book provides prescriptive guidance and associated methodologies to make your mark in big data space with working knowledge of Spark and help you pass your Spark certification exam. This book expects you to have a working knowledge of Python, but it does not expect any prior Spark knowledge, although having a working knowledge of PySpark would be very beneficial.

What this book covers

In the following chapters, we will cover the following topics.

Chapter 1, *Overview of the Certification Guide and Exam*, introduces the basics of the certification exam in PySpark and how to prepare for it.

Chapter 2, *Understanding Apache Spark and Its Applications*, delves into the fundamentals of Apache Spark, exploring its core functionalities, ecosystem, and real-world applications. It introduces Spark's versatility in handling diverse data processing tasks, such as batch processing, real-time analytics, machine learning, and graph processing. Practical examples illustrate how Spark is utilized across industries and its evolving role in modern data architectures.

Chapter 3, *Spark Architecture and Transformations*, deep-dives into the architecture of Apache Spark, elucidating the RDD (Resilient Distributed Dataset) abstraction, Spark's execution model, and the significance of transformations and actions. It explores the concepts of narrow and wide transformations, their impact on performance, and how Spark's execution plan optimizes distributed computations. Practical examples elucidate these concepts for better comprehension.

Chapter 4, *Spark DataFrames and their Operations*, focuses on Spark's DataFrame API and explores its role in structured data processing and analytics. It covers DataFrame creation, manipulation, and various operations, such as filtering, aggregations, joins, and groupings. Illustrative examples demonstrate the ease of use and advantages of the DataFrame API in handling structured data.

Chapter 5, *Advanced Operations and Optimizations in Spark and Optimization*, expands on your foundational knowledge and delves into advanced Spark operations, including broadcast variables, accumulators, custom partitioning, and working with external libraries. It explores techniques to handle complex data types, optimize memory usage, and leverage Spark's extensibility for advanced data processing tasks.

This chapter also delves into performance optimization strategies in Spark, emphasizing the significance of adaptive query execution. It explores techniques for optimizing Spark jobs dynamically, including runtime query planning, adaptive joins, and data skew handling. Practical tips and best practices are provided to fine-tune Spark jobs for enhanced performance.

Chapter 6, *SQL Queries in Spark*, focuses on Spark's SQL module and explores the SQL-like querying capabilities within Spark. It covers the DataFrame API's interoperability with SQL, enabling users to run SQL queries on distributed datasets. Examples showcase how to express complex data manipulations and analytics using SQL queries in Spark.

Chapter 7, *Structured Streaming in Spark*, focuses on real-time data processing and introduces Structured Streaming, Spark's API for handling continuous data streams. It covers concepts such as event time processing, watermarking, triggers, and output modes. Practical examples demonstrate how to build and deploy streaming applications using Structured Streaming.

This chapter is not included in the Spark certification exam, but it is beneficial to understand streaming concepts, since they are a core concept in the modern data engineering world.

Chapter 8, *Machine Learning with Spark ML*, explores Spark's machine learning library, Spark ML, diving into supervised and unsupervised machine learning techniques. It covers model building, evaluation, and hyperparameter tuning for various algorithms. Practical examples illustrate the application of Spark ML in real-world machine learning tasks.

This chapter is not included in the Spark certification exam, but it is beneficial to understand machine learning concepts in Spark, since they are a core concept in the modern data science world.

Chapter 9, *Mock Test 1*, provides you with the first mock test to prepare for the actual certification exam.

Chapter 10, *Mock Test 2*, provides you with the second mock test to prepare for the actual certification exam.

To get the most out of this book

Before diving into the chapters, it's essential to have a basic understanding of Python programming and familiarity with fundamental data processing concepts. Additionally, a grasp of distributed computing principles and experience with data manipulation and analysis will be beneficial. Throughout the book, we'll assume a working knowledge of Python and foundational concepts in data engineering and analytics. With these prerequisites in place, you'll be well-equipped to embark on your journey to becoming a certified Apache Spark developer.

Software/hardware covered in the book	Operating system requirements
Python	Windows, macOS, or Linux
Spark	

The code will work best if you sign up for the community edition of Databricks and import the python files into your account.

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.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Databricks-Certified-Associate-Developer-for-Apache-Spark-Using-Python`. 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!

Conventions used

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

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The `createOrReplaceTempView()` method allows us to save the processed data as a view in Spark SQL."

A block of code is set as follows:

```
# Perform an aggregation to calculate the average salary
average_salary = spark.sql("SELECT AVG(Salary) AS average_salary FROM
employees")
```

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: "The exam consists of **60 questions**. The time you're given to attempt these questions is **120 minutes**."

> **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 `customercare@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

Now you've finished *Databricks Certified Associate Developer for Apache Spark using Python*, 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.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781804619780`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Exam Overview

This part will show the basics of the certification exam for PySpark and the rules that need to be kept in mind. It will show the various types of questions asked in the exam and how to prepare for them.

This part has the following chapter:

- *Chapter 1*, *Overview of the Certification Guide and Exam*

1

Overview of the Certification Guide and Exam

Preparing for any task initially involves comprehending the problem at hand thoroughly and, subsequently, devising a strategy to tackle the challenge. Creating a step-by-step methodology for addressing each aspect of the challenge is an effective approach within this planning phase. This method enables smaller tasks to be handled individually, aiding in a systematic progression through the challenges without the need to feel overwhelmed.

This chapter intends to demonstrate this step-by-step approach to working through your Spark certification exam. In this chapter, we will cover the following topics:

- Overview of the certification exam
- Different types of questions to expect in the exam
- Overview of the rest of the chapters in this book

We'll start by providing an overview of the certification exam.

Overview of the certification exam

The exam consists of **60 questions**. The time you're given to attempt these questions is **120 minutes**. This gives you about **2 minutes per question**.

To pass the exam, you need to have a **score of 70%**, which means that you need to **answer 42 questions correctly** out of 60 for you to pass.

If you are well prepared, this time should be enough for you to answer the questions and also review them before the time finishes.

Next, we will see how the questions are distributed throughout the exam.

Distribution of questions

Exam questions are distributed into the following broad categories. The following table provides a breakdown of questions based on different categories:

Topic	Percentage of Exam	Number of Questions
Spark Architecture: Understanding of Concepts	17%	10
Spark Architecture: Understanding of Applications	11%	7
Spark DataFrame API Applications	72%	43

Table 1.1: Exam breakdown

Looking at this distribution, you would want to focus on the Spark DataFrame API a lot more in your exam preparation since this section covers around 72% of the exam (about 43 questions). If you can answer these questions correctly, passing the exam will become easier.

But this doesn't mean that you shouldn't focus on the Spark architecture areas. Spark architecture questions have varied difficulty, and they can sometimes be confusing. At the same time, they allow you to score easy points as architecture questions are generally straightforward.

Let's look at some of the other resources available that can help you prepare for this exam.

Resources to prepare for the exam

When you start planning to take the certification exam, the first thing you must do is master Spark concepts. This book will help you with these concepts. Once you've done this, it would be useful to do mock exams. There are two mock exams available in this book for you to take advantage of.

In addition, Databricks provides a practice exam, which is very useful for exam preparation. You can find it here: `https://files.training.databricks.com/assessments/practice-exams/PracticeExam-DCADAS3-Python.pdf`.

Resources available during the exam

During the exam, you will be given access to the Spark documentation. This is done via **Webassessor** and its interface is a little different than the regular Spark documentation you'll find on the internet. It would be good for you to familiarize yourself with this interface. You can find the interface at `https://www.webassessor.com/zz/DATABRICKS/Python_v2.html`. I recommend going through it and trying to find different packages and functions of Spark via this documentation to make yourself comfortable navigating it during the exam.

Next, we will look at how we can register for the exam.

Registering for your exam

Databricks is the company that has prepared these exams and certifications. Here is the link to register for the exam: `https://www.databricks.com/learn/certification/apache-spark-developer-associate`.

Next, we will look at some of the prerequisites for the exam.

Prerequisites for the exam

Some prerequisites are needed before you can take the exam so that you can be successful in passing the certification. Some of the major ones are as follows:

- Grasp the fundamentals of Spark architecture, encompassing the principles of Adaptive Query Execution.
- Utilize the Spark DataFrame API proficiently for various data manipulation tasks, such as the following:
  - Performing column operations, such as selection, renaming, and manipulation
  - Executing row operations, including filtering, dropping, sorting, and aggregating data
  - Conducting DataFrame-related tasks, such as joining, reading, writing, and implementing partitioning strategies
  - Demonstrating proficiency in working with **user-defined functions** (**UDFs**) and Spark SQL functions
- While not explicitly tested, a functional understanding of either Python or Scala is expected. The examination is available in both programming languages.

Hopefully, by the end of this book, you will be able to fully grasp all these concepts and have done enough practice on your own to be prepared for the exam with full confidence.

Now, let's discuss what to expect during the online proctored exam.

Online proctored exam

The Spark certification exam is an online proctored exam. What this means is that you will be taking the exam from the comfort of your home, but someone will be proctoring the exam online. I encourage you to understand the procedures and rules of the proctored exam in advance. This will save you a lot of trouble and anxiety at the time of the exam.

To give you an overview, throughout the exam session, the following procedures will be in place:

- Webcam monitoring will be conducted by a Webassessor proctor to ensure exam integrity
- You will need to present a valid form of identification with a photo
- You will need to conduct the exam alone
- Your desk needs to be decluttered and there should be no other electronic devices in the room except the laptop that you'll need for the exam
- There should not be any posters or charts on the walls of the room that may aid you in the exam
- The proctor will be listening to you during the exam as well, so you'll want to make sure that you're sitting in a quiet and comfortable environment
- It is recommended to **not** use your work laptop for this exam as it requires software to be installed and your antivirus and firewall to be disabled

The proctor's responsibilities are as follows:

- Overseeing your exam session to maintain exam integrity
- Addressing any queries related to the exam delivery process
- Offering technical assistance if needed
- It's important to note that the proctor will not offer any form of assistance regarding the exam content

I recommend that you take sufficient time before the exam to set up the environment where you'll be taking the exam. This will ensure a smooth online exam procedure where you can focus on the questions and not worry about anything else.

Now, let's talk about the different types of questions that may appear in the exam.

Types of questions

There are different categories of questions that you will find in the exam. They can be broadly divided into theoretical and code questions. We will look at both categories and their respective subcategories in this section.

Theoretical questions

Theoretical questions are the questions where you will be asked about the conceptual understanding of certain topics. Theoretical questions can be subdivided further into different categories. Let's look at some of these categories, along with example questions taken from previous exams that fall into them.

Explanation questions

Explanation questions are ones where you need to define and explain something. It can also include how something works and what it does. Let's look at an example.

Which of the following describes a worker node?

A. Worker nodes are the nodes of a cluster that perform computations.
B. Worker nodes are synonymous with executors.
C. Worker nodes always have a one-to-one relationship with executors.
D. Worker nodes are the most granular level of execution in the Spark execution hierarchy.
E. Worker nodes are the coarsest level of execution in the Spark execution hierarchy.

Connection questions

Connections questions are such questions where you need to define how different things are related to each other or how they differ from each other. Let's look at an example to demonstrate this.

Which of the following describes the relationship between worker nodes and executors?

A. An executor is a **Java Virtual Machine** (**JVM**) running on a worker node.
B. A worker node is a JVM running on an executor.
C. There are always more worker nodes than executors.
D. There are always the same number of executors and worker nodes.
E. Executors and worker nodes are not related.

Scenario question

Scenario questions involve defining how things work in different if-else scenarios – for example, "If _____ occurs, then _____ happens." Moreover, it also includes questions where a statement is incorrect about a scenario. Let's look at an example to demonstrate this.

If Spark is running in cluster mode, which of the following statements about nodes is incorrect?

A. There is a single worker node that contains the Spark driver and the executors.
B. The Spark driver runs in its own non-worker node without any executors.
C. Each executor is a running JVM inside a worker node.
D. There is always more than one node.
E. There might be more executors than total nodes or more total nodes than executors.

Categorization questions

Categorization questions are such questions where you need to describe categories that something belongs to. Let's look at an example to demonstrate this.

Which of the following statements accurately describes stages?

A. Tasks within a stage can be simultaneously executed by multiple machines.

B. Various stages within a job can run concurrently.

C. Stages comprise one or more jobs.

D. Stages temporarily store transactions before committing them through actions.

Configuration questions

Configuration questions are such questions where you need to outline how things will behave based on different cluster configurations. Let's look at an example to demonstrate this.

Which of the following statements accurately describes Spark's cluster execution mode?

A. Cluster mode runs executor processes on gateway nodes.

B. Cluster mode involves the driver being hosted on a gateway machine.

C. In cluster mode, the Spark driver and the cluster manager are not co-located.

D. The driver in cluster mode is located on a worker node.

Next, we'll look at the code-based questions and their subcategories.

Code-based questions

The next category is code-based questions. A large number of Spark API-based questions lie in this category. Code-based questions are the questions where you will be given a code snippet, and you will be asked questions about it. Code-based questions can be subdivided further into different categories. Let's look at some of these categories, along with example questions taken from previous exams that fall into these different subcategories.

Function identification questions

Function identification questions are such questions where you need to define which function does something. It is important to know the different functions that are available in Spark for data manipulation, along with their syntax. Let's look at an example to demonstrate this.

Which of the following code blocks returns a copy of the `df` DataFrame, where the `column` salary has been renamed `employeeSalary`?

A. `df.withColumn(["salary", "employeeSalary"])`

B. `df.withColumnRenamed("salary").alias("employeeSalary ")`

C. `df.withColumnRenamed("salary", " employeeSalary ")`

D. `df.withColumn("salary", " employeeSalary ")`

Fill-in-the-blank questions

Fill-in-the-blank questions are such questions where you need to complete the code block by filling in the blanks. Let's look at an example to demonstrate this.

The following code block should return a DataFrame with the `employeeId`, `salary`, `bonus`, and `department` columns from the `transactionsDf` DataFrame. Choose the answer that correctly fills the blanks to accomplish this.

```
df.__1__(__2__)
```

A.

i. `drop`

ii. `"employeeId", "salary", "bonus", "department"`

B.

i. `filter`

ii. `"employeeId, salary, bonus, department"`

C.

i. `select`

ii. `["employeeId", "salary", "bonus", "department"]`

D.

i. `select`

ii. `col(["employeeId", "salary", "bonus", "department"])`

Order-lines-of-code questions

Order-lines-of-code questions are such questions where you need to place the lines of code in a certain order so that you can execute an operation correctly. Let's look at an example to demonstrate this.

Which of the following code blocks creates a DataFrame that shows the mean of the `salary` column of the `salaryDf` DataFrame based on the `department` and `state` columns, where `age` is greater than 35?

i. `salaryDf.filter(col("age") > 35)`
ii. `.filter(col("employeeID")`
iii. `.filter(col("employeeID").isNotNull())`
iv. `.groupBy("department")`
v. `.groupBy("department", "state")`
vi. `.agg(avg("salary").alias("mean_salary"))`
vii. `.agg(average("salary").alias("mean_salary"))`

A. i, ii, v, vi
B. i, iii, v, vi
C. i, iii, vi, vii
D. i, ii, iv, vi

Summary

This chapter provided an overview of the certification exam. At this point, you know what to expect in the exam and how to best prepare for it. To do so, we covered different types of questions that you will encounter.

Going forward, each chapter of this book will equip you with practical knowledge and hands-on examples so that you can harness the power of Apache Spark for various data processing and analytics tasks.

Part 2: Introducing Spark

This part will offer you a comprehensive understanding of Spark's capabilities and operational principles. It will cover what Spark is, why it's important, and some of the applications Spark is most useful in. It will tell you about the different types of users who can benefit from Spark. It will also cover the basics of Spark architecture and how applications are navigated through in Spark. It will detail narrow and wide Spark transformations and discuss lazy evaluations in Spark. It's important to have this understanding because Spark works differently than other traditional frameworks.

This part has the following chapters:

- *Chapter 2, Understanding Apache Spark and Its Applications*
- *Chapter 3, Spark Architecture and Transformations*

2

Understanding Apache Spark and Its Applications

With the advent of machine learning and data science, the world is seeing a paradigm shift. A tremendous amount of data is being collected every second, and it's hard for computing power to keep up with this pace of rapid data growth. To make use of all this data, Spark has become a de facto standard for big data processing. Migrating data processing to Spark is not only a question of saving resources that will allow you to focus on your business; it's also a means of modernizing your workloads to leverage the capabilities of Spark and the modern technology stack to create new business opportunities.

In this chapter, we will cover the following topics:

- What is Apache Spark?
- Why choose Apache Spark?
- Different components of Spark
- What are the Spark use cases?
- Who are the Spark users?

What is Apache Spark?

Apache Spark is an open-source big data framework that is used for multiple big data applications. The strength of Spark lies in its superior parallel processing capabilities that makes it a leader in its domain.

According to its website (`https://spark.apache.org/`), "*The most widely-used engine for scalable computing.*"

The history of Apache Spark

Apache Spark started as a research project at the UC Berkeley AMPLab in 2009 and moved to an open source license in 2010. Later, in 2013, it came under the Apache Software Foundation (`https://spark.apache.org/`). It gained popularity after 2013, and today, it serves as a backbone for a large number of big data products across various Fortune 500 companies and has thousands of developers actively working on it.

Spark came into being because of limitations in the Hadoop MapReduce framework. MapReduce's main premise was to read data from disk, distribute that data for parallel processing, apply map functions to the data, and then reduce those functions and save them back to disk. This back-and-forth reading and saving to disk becomes time-consuming and costly very quickly.

To overcome this limitation, Spark introduced the concept of in-memory computation. On top of that, Spark has several capabilities that came as a result of different research initiatives. You will read more about them in the next section.

Understanding Spark differentiators

Spark's foundation lies in its major capabilities such as in-memory computation, lazy evaluation, fault tolerance, and support for multiple languages such as Python, SQL, Scala, and R. We will discuss each one of them in detail in the following section.

Let's start with in-memory computation.

In-memory computation

The first major differentiator technology that Spark's foundation is built on is that it utilizes in-memory computations. Remember when we discussed Hadoop MapReduce technology? One of its major limitations is to write back to disk at each step. Spark saw this as an opportunity for improvement and introduced the concept of in-memory computation. The main idea is that the data remains in memory as long as it is worked on. If we can work with the size of data that can be stored in the memory at once, we can eliminate the need to write to disk at each step. As a result, the complete computation cycle can be done in memory if we can work with all computations on that amount of data. Now, the thing to note here is that with the advent of big data, it's hard to contain all the data in memory. Even if we look at heavyweight servers and clusters in the cloud computing world, memory remains finite. This is where Spark's internal framework of parallel processing comes into play. Spark framework utilizes the underlying hardware resources in the most efficient manner. It distributes the computations across multiple cores and utilizes the hardware capabilities to the maximum.

This tremendously reduces the computation time, since the overhead of writing to disk and reading it back for the subsequent step is minimized as long as the data can be fit in the memory of Spark compute.

Lazy evaluation

Generally, when we work with programming frameworks, the backend compilers look at each statement and execute it. While this works great for programming paradigms, with big data and parallel processing, we need to shift to a look-ahead kind of model. Spark is well known for its parallel processing capabilities. To achieve even better performance, Spark doesn't execute code as it reads it, but once the code is there and we submit a Spark statement to execute, the first step is that Spark builds a logical map of the queries. Once that map is built, then it plans what the best path of execution is. You will read more about its intricacies in the Spark architecture chapters. Once the plan is established, only then will the execution begin. Once the execution begins, even then, Spark holds off executing all statements until it hits an "action" statement. There are two types of statements in Spark:

- Transformations
- Actions

You will learn more about the different types of Spark statements in detail in *Chapter 3*, where we discuss Spark architecture. Here are a few advantages of lazy evaluation:

- Efficiency
- Code manageability
- Query and resource optimization
- Reduced complexities

Resilient datasets/fault tolerance

Spark's foundation is built on **resilient distributed datasets** (**RDDs**). It is an immutable distributed collection of objects that represent a set of records. RDDs are distributed across a number of servers, and they are computed in parallel across multiple cluster nodes. RDDs can be generated with code. When we read data from an external storage location into Spark, RDDs hold that data. This data can be shared across multiple clusters and can be computed in parallel, thus giving Spark a very efficient way of running computations on RDD data. RDDs are loaded in memory for processing; therefore, loading to and from memory computations is not required, unlike Hadoop.

RDDs are fault-tolerant. This means that if there are failures, RDDs have the ability to self-recover. Spark achieves that by distributing these RDDs to different worker nodes while keeping in view what task is performed by which worker node. This handling of worker nodes is done by the Spark driver. We will discuss this in detail in upcoming chapters.

RDDs give a lot of power to Spark in terms of resilience and fault-tolerance. This capability, along with other features, makes Spark the tool of choice for any production-grade applications.

Multiple language support

Spark supports multiple languages for development such as Java, R, Scala, and Python. This gives users the flexibility to use any language of choice to build applications in Spark.

The components of Spark

Let's talk about the different components Spark has. As you can see in *Figure 1.1*, Spark Core is the backbone of operations in Spark and spans across all the other components that Spark has. Other components that we're going to discuss in this section are Spark SQL, Spark Streaming, Spark MLlib, and GraphX.

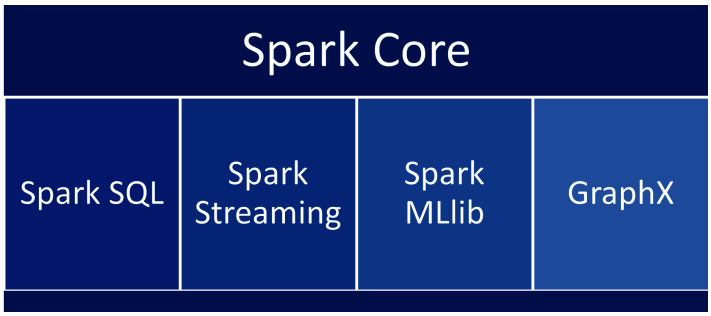


Figure 2.1: Spark components

Let's look at the first component of Spark.

Spark Core

Spark Core is central to all the other components of Spark. It provides functionalities and core features for all the different components. Spark SQL, Spark Streaming, Spark MLlib, and GraphX all make use of Spark Core as their base. All the functionality and features of Spark are controlled by Spark Core. It provides in-memory computing capabilities to deliver speed, a generalized execution model to support a wide variety of applications, and Java, Scala, and Python APIs for ease of development.

In all of these different components, you can write queries in supported languages. Spark will then convert these queries to **directed acyclic graphs** (**DAGs**), and Spark Core has the responsibility of executing them.

The key responsibilities of Spark Core are as follows:

- Interacting with storage systems
- Memory management
- Task distribution

- Task scheduling
- Task monitoring
- In-memory computation
- Fault tolerance
- Optimization

Spark Core contains an API for RDDs which are an integral part of Spark. It also provides different APIs to interact and work with RDDs. All the components of Spark work with underlying RDDs for data manipulation and processing. RDDs make it possible for Spark to have a lineage for data, since they are immutable. This means that every time an operation is run on an RDD that requires changes in it, Spark will create a new RDD for it. Hence, it maintains the lineage information of RDDs and their corresponding operations.

Spark SQL

SQL is the most popular language for database and data warehouse applications. Analysts use this language for all their exploratory data analysis on relational databases and their counterparts in traditional data warehouses. Spark SQL adds this advantage to the Spark ecosystem. Spark SQL is used to query structured data in SQL using the DataFrame API.

As its name represents, Spark SQL gives SQL support to Spark. This means we can query the data present in RDDs and other external sources, such as Parquet files. This is a powerful capability of Spark, since it gives developers the flexibility to use a relational table structure on top of RDDs and other file formats and write SQL queries on top of it. This also adds the capabilities of using SQL where necessary and unifies it with analytics applications and use cases, thus providing unification of the platforms.

With Spark SQL, developers are able to do the following with ease:

- They can read data from different file formats and different sources into RDDs and DataFrames
- They can run SQL queries on top of the data present in DataFrames, thus giving flexibility to the developers to use programming languages or SQL to process data
- Once they're done with the processing of the data, they have the capability to write RDDs and DataFrames to external sources

Spark SQL consists of a cost-based optimizer that optimizes queries, keeping in view the resources; it also has the capability to generate code for these optimizations, which makes these queries very fast and efficient. To support even faster query times, it can scale to multiple nodes with the help of Spark Core and also provides features such as fault tolerance and resiliency. This is known as the Catalyst optimizer. We will read more about it in *Chapter 5*.

The most noticeable features of Sparks SQL are as follows:

- It provides an engine for high-level structured APIs
- Reads/writes data to and from a large number of file formats such as Avro, Delta, **Comma-Separated Values** (**CSV**), and Parquet
- Provides **Open Database Connectivity** (**ODBC**)/**Java Database Connectivity** (**JDBC**) connectors to **business intelligence** (**BI**) tools such as PowerBI and Tableau, as well as popular **relational databases** (**RDBMs**)
- Provides a way to query structured data in files as tables and views
- It supports ANSI SQL:2003-compliant commands and HiveQL

Now that we have covered SparkSQL, let's discuss the Spark Streaming component.

Spark Streaming

We have talked about the rapid growth of data in today's times. If we were to divide this data into groups, there are two types of datasets in practice, batch and streaming:

- **Batch data** is when there's a chunk of data present that you have to ingest and then transform all at once. Think of when you want to get a sales report of all the sales in a month. You would have the monthly data available as a batch and process it all at once.
- **Streaming data** is when you need output of that data in real time. To serve this requirement, you would have to ingest and process that data in real time. This means every data point can be ingested as a single data element, and we would not wait for it to be ingested after a block of data is collected. Think of when self-driving cars need to make decisions in real time based on the data they collect. All the data needs to be ingested and processed in real time for the car to make effective decisions in a given moment.

There are a large number of industries generating streaming data. To make use of this data, you need real-time ingestion, processing, and management of this data. It has become essential for organizations to use streaming data as it arrives for real-time analytics and other use cases. This gives them an edge over their competitors, as this allows them to make decisions in real time.

Spark Streaming enables organizations to make use of streaming data. One of the most important factors of Spark Streaming is its ease of use alongside batch data processing. You can combine batch and stream data within one framework and use it to augment your analytics applications. Spark Streaming also inherits Spark Core's features of resilience and fault tolerance, giving it a dominant position in the industry. It integrates with a large number of streaming data sources such as HDFS, Kafka, and Flume.

The beauty of Spark Streaming is that batch data can be processed as streams to take advantage of built-in paradigms of streaming data and look-back capabilities. There are certain factors that need to be taken into consideration when we work with real-time data. When we work with real-time data streams,

there's a chance that some of the data may get missed due to system hiccups or failures altogether. Spark Streaming takes care of this in a seamless way. To cater to these requirements, it has a built-in mechanism called **checkpoints**. The purpose of these checkpoints is to keep track of the incoming data, knowing what was processed downstream and which data is still left to be processed in the next cycle. We will learn more about this in *Chapter 7* when we discuss Spark Streaming in more detail.

This makes Spark resilient to failures. If there are any failures, you need minimal work to reprocess old data. You can also define mechanisms and algorithms for missing data or late processed data. This gives a lot of flexibility to the data pipelines and makes them easier to maintain in large production environments.

Spark MLlib

Spark provides a framework for distributed and scalable machine learning. It distributes the computations across different nodes, thus resulting in better performance for model training. It also distributes hyperparameter tuning. You will learn more about hyperparameter tuning in *Chapter 8*, where we talk about machine learning. Because Spark can scale to large datasets, it is the framework of choice for machine learning production pipelines. When you build products, execution and computation speed matter a lot. Spark gives you the ability to work with large amounts of data and build state-of-the-art machine learning models that can run very efficiently. Instead of working with models that take days to train, Spark reduces that time to hours. In addition, working with more data results in better-performing models in most cases.

Most of the commonly used machine learning algorithms are part of Spark's libraries. There are two machine learning packages available in Spark:

- Spark MLlib
- Spark ML

The major difference between these two is the type of data they work with. Spark MLlib is built on top of RDDs while Spark ML works with DataFrames. Spark MLlib is the older library and has now entered maintenance mode. The more up-to-date library is Spark ML. You should also note that Spark ML is not the official name of the library itself, but it is commonly used to refer to the DataFrame-based API in Spark. The official name is still Spark MLlib. However, it's important to know the differences.

Spark MLlib contains the most commonly used machine learning libraries for **classification**, **regression**, **clustering**, and **recommendation systems**. It also has some support for frequent pattern mining algorithms.

When there is a need to serve these models to millions and billions of users, Spark is also helpful. You can distribute and parallelize both data processing (**Extract, Transform, Load** (**ETL**)) and model scoring with Spark.

GraphX

GraphX is Spark's API for graphs and graph-parallel computation. GraphX extends Spark's RDD to work with graphs and allows you to run parallel computations with graph objects. This speeds up the computations significantly.

Here's a network graph that represents what a graph looks like.

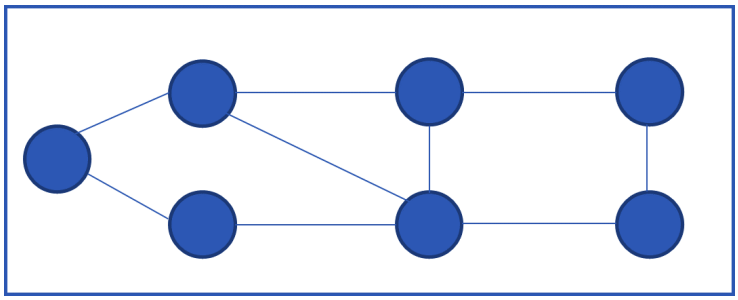


Figure 2.2: A network graph

A graph is an object with vertices and edges. Properties are attached to each vertex and edge. There are primary graph operations that Spark supports, such as `subgraph` and `joinVertices`.

The main premise is that you can use GraphX for exploratory analysis and ETL and transform and join graphs with RDDs efficiently. There are two types of operator— `Graph` and `GraphOps`. On top of that, graph aggregation operators are also available. Spark also includes a number of graph algorithms that are used in common use cases. Some of the most popular algorithms are as follows:

- PageRank
- Connected components
- Label propagation
- SVD++
- Strongly connected components
- Triangle count

Now, let's discuss why we want to use Spark in our applications and what some of the features it provides are.

Why choose Apache Spark?

In this section, we will discuss the applications of Apache Spark and its features, such as speed, reusability, in-memory computations, and how Spark is a unified platform.

Speed

Apache Spark is one of the fastest processing frameworks for data available today. It beats Hadoop MapReduce by a large margin. The main reason is its in-memory computation capabilities and lazy evaluation. We will learn more about this when we discuss Spark architecture in the next chapter.

Reusability

Reusability is a very important consideration for large organizations making use of modern platforms. Spark can join batch and stream data seamlessly. Moreover, you can augment datasets with historical data to serve your use cases better. This gives a large historical view of data to run queries or build modern analytical systems.

In-memory computation

With in-memory computation, all the overhead of reading and writing to disks is eliminated. The data is cached, and at each step, the required data is already present in memory. At the end of the processing, results are aggregated and sent back to the driver for further steps.

All of this is facilitated by the process of DAG creation that Spark performs inherently. Before execution, Spark creates a DAG of the necessary steps and prioritizes them based on its internal algorithms. We will learn more about this in the next chapter. These capabilities support in-memory computation, resulting in fast processing speeds.

A unified platform

Spark provides a unified platform for data engineering, data science, machine learning, analytics, streaming, and graph processing. All of these components are integrated with Spark Core. The core engine is very high-speed and generalizes the commonly needed tasks for its other components. This gives Spark an advantage over other platforms because of the unification of its different components. These components can work in conjunction with each other, providing a unified experience for software applications. In modern applications, this unification makes it easy to use, and different parts of the application can make use of the core capabilities of these components without compromising on features.

Now that you understand the benefits of using Spark, let's talk about the different use cases of Spark in the industry.

What are the Spark use cases?

In this section, we will learn about how Spark is used in the industry. There are various use cases of Spark prevalent today, some of which include big data processing, machine learning applications, near-real-time and real-time streaming, and using graph analytics.

Big data processing

One of the most popular use cases for Spark is big data processing. You might be wondering what big data is, so let's take a look at the components that mark data as big data.

The first component of big data is the **volume of data**. By volume, we mean that the data is very large in size, often amounting to terabytes, petabytes, and beyond in some cases. Organizations have collected a large amount of data over the years. This data can be used for analysis. However, the first step in this activity is to process these large amounts of data. Also, it's only recently that computing power has grown to now be able to process such vast volumes of data.

The second component of big data is the **velocity of data**. The velocity of data refers to the speed of its generation, ingestion, and distribution. This means that the speed with which this data is generated has increased manifold in recent years. Take, for example, data generated by your smart appliance that sends data every second to a server. In this process, the server also needs to keep up with the ingestion of this data, and then distributing this across different sources might be the next step.

The third component of big data is the **variety of data**. The variety of data refers to the different sources that generate the data. It also refers to different types of data that are generated. Gone are the days when data was only generated in structured formats that could be saved as tables in databases. Currently, data can be structured, semi-structured, or unstructured. The systems now have to work with all these different data types, and tools should be able to manipulate these different data types. Think of images that need to be processed or audio and video files that can be analyzed with advanced analytics.

Some other components can be added to the original three Vs as well, such as veracity and value. However, these components are out of the scope of our discussion.

Big data is too large for regular machines to process it. That's why it's called big data. Big data that is high-volume, high-velocity, and high-variety needs to be processed with advanced analytical tools such as Spark, which can distribute the workload across different machines or clusters and does processing in parallel to make use of all the resources available on a machine. So, instead of using only a single machine and loading all the data into one node, Spark gives us the ability to divide the data up into different parts and process them in parallel and across different machines. This massively speeds up the whole process and makes use of all the available resources.

For all of the aforementioned reasons, Spark is one of the most widely used big data processing technologies. Large organizations make use of Spark to analyze and manipulate their big data stacks. Spark serves as a backbone for big data processing in complex analytical use cases.

Some examples of big data use cases are as follows:

- Business intelligence for reporting and dashboarding
- Data warehousing for complex applications
- Operational analytics for application monitoring

It is important to note here that working with Spark requires a mindset shift from single-node processing to big data-processing paradigms. You now have to start thinking about how to best utilize and optimize the use of large clusters for processing and what some of the best practices around parallel processing are.

Machine learning applications

As data grows, so does the need for machine learning models to make use of more and more data. The general understanding in the machine learning community today is that the more data is provided to the models, the better the models will be. This resulted in the need for massive amounts of data to be given to a model for predictive analytics. When we deal with massive amounts of data, the challenges for training machine learning models become more complex than data processing. The reason is that machine learning models crunch data and run statistical estimations to come to a minimum error point. To get to that minimum error, the model must do complex mathematical operations such as matrix multiplication. These computations require large amounts of data to be available in memory and then computations to run on it. This serves as the case for parallel processing in machine learning.

Machine learning adds an element of prediction to products. Instead of reacting to the changes that have already taken place, we can proactively look for ways to improve our products and services based on historical data and trends. Every aspect of an organization can make use of machine learning for predictive analytics. Machine learning can be applied to a number of industries, from hospitals to retail stores to manufacturing organizations. All of us have encountered some kind of machine learning algorithms when we do tasks on the internet, such as online buying and selling, browsing and searching for websites, and using social media platforms. Machine learning has become a major part of our lives knowingly or unknowingly.

Although there's a large number of use cases that an organization can make use of in terms of machine learning, I'm highlighting only a few here:

- Personalized shopping
- Website searches and ranking
- Fraud detection for banking and insurance
- Customer sentiment analysis
- Customer segmentation
- Recommendation engines
- Price optimization
- Predictive maintenance and support
- Text and video analytics
- Customer/patient 360

Let's move on to cover real-time streaming next.

Real-time streaming

Real-time streaming is one of the use cases where Spark really shines. There are very few competing frameworks that offer the flexibility that Spark Streaming has.

Spark Streaming provides a mechanism to ingest data from multiple streaming data sources, such as Kafka and Amazon Kinesis. Once the data is ingested, it can be processed in real time with very efficient Spark Streaming processing.

There are a large number of real-time use cases that can make use of Spark Streaming. Some of them are as follows:

- Self-driving cars
- Real-time reporting and analysis
- Providing updates on stock market data
- Internet of Things (IoT) data ingestion and processing
- Real-time news data processing
- Real-time analytics for optimization of inventory and operations
- Real-time fraud detection systems for credit cards
- Real-time event detection
- Real-time recommendations

Large global organizations make use of Spark Streaming to process billion and trillions of data rows in real time. We see some of this in action in our everyday life. Your credit card blocking a transaction while you're out shopping is one such example of real-time fraud detection in action. Netflix and YouTube use real-time interactions, with the video platforms recommending users what to watch next.

As we move to a world of every device sending data back to its server for analysis, there's an increased need for streaming and real-time analysis. One of the main advantages of using Spark Streaming for this kind of data is the built-in capabilities it has, for look-back and late processing of data. We discussed the usefulness of this approach earlier as well, and a lot of manual pipeline processing work is removed due to these capabilities. We will learn more about this when we discuss Spark Streaming in *Chapter 7*.

Graph analytics

Graph analytics provides a unique way of looking at data by analyzing relationships between different entities. The vertices of a graph represent the entities and the edges of a graph represent the relationship between two entities. Think of your social network on Facebook or Instagram. You represent one

entity, and the people you are connected to represent another entity. The relationship (connection) between you and your friends is the edge. Similarly, your interests on your social media could all be different edges. Then, there can be a location category, for which all people who belong to one location would have an edge (a relationship) with that location, and so on. Therefore, connections can be made with any different type of entities. The more connected you are, the higher the chance that you are connected to like-minded people or interests. This is one method of measuring relationships between different entities. There are several uses for these kinds of graphs. The beauty of Spark is the distributed processing of these graphs to find these relationships very quickly. There can be millions and billions of connections for billions of entities. Spark has the capability to distribute these workloads and compute complex algorithms very fast.

The following are some use cases of graph analytics:

- Social network analysis
- Fraud detection
- Page ranking based on relevance
- Weather prediction
- Search engine optimization
- Supply chain analysis
- Finding influencers on social media
- Money laundering and fraud detection

With a growing number of use cases of graph analytics, this proves to be a critical use case in the industry today where we need to analyze networks of relationships among entities.

In the next section, we're going to discuss who the Spark users are and what their typical role is within an organization.

Who are the Spark users?

As the world moves toward data-driven decision-making approaches, the role of data and the different types of users who can leverage it for critical business decisions has become paramount. There are different types of users in data who can leverage Spark for different purposes. I will introduce some of those different users in this section. This is not an exhaustive list, but it should give you an idea of the different roles that exist in data-driven organizations today. However, as the industry grows, many more new roles are coming up that are similar to the ones present in the following sections, although each may have its own separate role.

We'll start with the role of data analysts.

Data analysts

The more traditional role in data today is a data analyst. The data analyst is typically the first-tier role in data. What this means is that data analysts are at the core of decision making in organizations. This role spans across different business units in an organization, and oftentimes, data analysts have to interact with multiple business stakeholders to put across their requirements. This requires knowledge of the business domain as well as its processes. When an analyst has an understanding of the business and its goals, only then can they perform their duties best. Moreover, a lot of times, the requirement is to make current processes more efficient, which results in a better bottom line for the business. Therefore, having an understanding of not just the business goals but also how it all works together is one of the main requirements for this role.

A typical job role for a data analyst may look as follows:

1. When data analysts are given a project in an organization, the first step in the project is to gather requirements from multiple stakeholders. Let's work with an example here. Say you joined an organization as a data analyst. This organization makes and sells computer hardware. You are given the task of reporting on the revenue each month for the last 10 years. The first step for you would be to gather all requirements. It is possible that some stakeholders want to know how many units of certain products are sold each month, while others may want to know whether the revenues are consistently growing or not. Remember, the end users of your reports might work in different business units of the organization.
2. Once you have all the requirements gathered from all the concerned stakeholders, then you move on to the next step, which is to look for the relevant data sources to answer the questions that you are tasked with. You may need to talk with database administrators in the organization or platform architects to know where the different data sources reside that have relevant information for you to extract.
3. Once you have all the relevant sources, then you want to connect with those sources programmatically (in most cases) and clean and join some data together to come up with relevant statistics, based on your requirements. This is where Spark would help you connect to these different data sources and also read and manipulate the data most efficiently. You also want to slice and dice the data based on your business requirements. Once the data is clean and statistics are generated, you want to generate some reports based on these statistics. There are different tools in the market to generate reports, such as Qlik and Tableau, that you can work with. Once the reports are generated, you may want to share your results with the stakeholders. You could present your results to them or share the reports with them, depending on what the preferred medium is. This will help stakeholders make informed business-critical decisions that are data-driven in nature.

Collaboration across different roles also plays an important role for data analysts. Since organizations have been collecting data for a long time, the most important thing is working with all the data that has been collected over the years and making sense of it, helping businesses with critical decision making. Helping with data-driven decision making is the key to being a successful data analyst.

Here's a summary of the steps taken in a project, as discussed in the previous paragraphs:

1. Gather requirements from stakeholders.
2. Identify the relevant data sources.
3. Collaborate with subject matter experts (SMEs).
4. Slice and dice data.
5. Generate reports.
6. Share the results.

Let's look at data engineers next. This role is gaining a lot of traction in the industry today.

Data engineers

The next role that is getting more and more prevalent in the industry is a data engineer. This is a relatively new role but has gained immense popularity in recent times. The reason for this is that data is growing at tremendous levels. We have more data being generated per second now than in a whole month a few years ago. Working with all this data requires specialized skills. The data can no longer be contained in the modest memory of most computers, so we have to make use of the massive scale of cloud computing to serve this purpose. As data needs are becoming a lot more complex, we need complex architectures to process and use this data for business decision making. This is where the role of the data engineer comes into play. The main job of the data engineer is to prepare data for ingestion for different purposes. The downstream systems that leverage this prepared data could be dashboards that run reports based on this data, or it could be a predictive analytics solution that works with advanced machine learning algorithms to make proactive decisions based on the data.

More broadly, data engineers are responsible for creating, maintaining, optimizing, and monitoring data pipelines that serve different use cases in an organization. These pipelines are typically known as Extract, Transform, Load (ETL) pipelines. The major differentiator is the sheer scale of data that data engineers have to work with. When there are downstream needs for data for BI reporting, advanced analytics, and/or machine learning, that is where data pipelines come into play for large projects.

A typical job role for a data engineer in an organization may look as follows. When data engineers are given a task to create a data pipeline for a project, the first thing they need to consider is the overall architecture of an application. There might be data architects in some organizations to help with some of the architecture requirements, but that might not always be the case. So, a data engineer would ask questions such as the following:

- What are the different sources of data?
- What is the size of the data?
- Where does the data reside today?
- Do we need to migrate the data between different tools?

- How do we connect to the data?
- What kind of transformations are required for the data?
- How often does the data get updated?
- Should we expect a schema change in the new data?
- How do we monitor the pipelines if there are failures?
- Do we need to create a notification system for failures?
- Do we need to add a retry mechanism for failures?
- What is the timeout strategy for failures?
- How do we run back-dated pipelines if there are failures?
- How do we deal with bad data?
- What strategy we should follow – ETL or ELT?
- How can we save the costs of computation?

Once they have answers to these questions, then they start working on a resilient architecture to build data pipelines. Once those pipelines are run and tested, the next step is to maintain these pipelines and make the processing more efficient and visible for failure detection. The goal is to build these pipelines so that once everything is run, the end state of data is consistent for different downstream use cases. Too often, data engineers have to collaborate with data analysts and data scientists to come up with correct data transformation requirements, based on the required use cases.

Let's talk about data scientists now, a job that has been advertised as "*the sexiest job of the 21st century*" on multiple forums.

Data scientists

Traditionally, data has been used for decision making based on what has happened in the past. This means that organizations have been reactive, based on the data. Now, there's been a paradigm shift in advanced and predictive analytics. This means instead of being reactive, organizations can be proactive in their decision making. They achieve this with the help of all the data that is available to organizations now. To make effective use of this data, data scientists play a major part. They take analytics to the next level, where instead of just looking at what has happened in the past, they have sophisticated machine learning algorithms to predict what could take place in the future as well. All this is based on the huge amounts of data that is available to them.

A typical job role for a data scientist in an organization may look as follows.

The data scientist is given a problem to solve or a question to answer. The first task is to see what kind of data is available to them that would help them answer this question. They would create a few hypotheses to test with the given data. If the results are positive and the data is able to answer some of the problem statements, then they move on to experimenting with the data and seek ways to more effectively answer the questions at hand. For this purpose, they would join different datasets together, and they would also transform the data to make it ready for some machine learning algorithms to consume. At this stage, they would also need to decide what kind of machine learning problem they aim to solve.

There are three major types of machine learning techniques that they can use:

- Regression
- Classification
- Clustering

Based on the technique decided and data transformations, they would then move to prototype with a few machine learning algorithms to create a baseline model. A baseline model is a very basic model that serves to answer the original question. Based on this baseline model, other models can be created that would be able to answer the question better. In some cases, some predefined rules can also serve as a baseline model. What this means is that the business might already be operating on some predefined rules that can serve as a baseline to compare the machine learning model. Once the initial prototyping is done, then the data scientist moves on to more advanced optimizations in terms of models. They can work with different hyperparameters of the model or experiment with different data transformations and sample sizes. All of this can be done in Spark or other tools and languages, depending on their preference. Spark has the edge to run these algorithms in a parallel fashion, making the whole process very efficient. Once the data scientist is happy with the model results based on different metrics, they would then move that model to a production environment where these models can be served to customers solving specific problems. At this point, they would hand over these models to machine learning engineers to start incorporating them into the pipelines.

Here's a summary of the steps taken in a project, as discussed in the previous paragraph:

1. Create and test a hypothesis.
2. Transform the data.
3. Decide on a machine learning algorithm.
4. Prototype with different machine learning models.
5. Create a baseline model.
6. Tune the model.
7. Tune the data.
8. Transition models to production.

Let's discuss the role of machine learning engineers next.

Machine learning engineers

Like data engineers, machine learning engineers also build pipelines, but these pipelines are primarily built for machine learning model deployment. Machine learning engineers typically take prototyped models created by data scientists and build machine learning pipelines around them. We will discuss what machine learning pipelines are and what some of the questions that need to be answered to build these pipelines are.

Machine learning models are built to solve complex problems and provide advanced analytic methods to serve a business. After prototyping, these models need to run in the production environments of the organizations and be deployed to serve customers. For deployment, there are several considerations that need to be taken into account:

- How much data is there for model training?
- How many customers do we plan to serve concurrently?
- How often do we need to retrain the models?
- How often do we expect the data to change?
- How do we scale the pipeline up and down based on demand?
- How do we monitor failures in model training?
- Do we need notifications for failures?
- Do we need to add a retry mechanism for failures?
- What is the timeout strategy for failures?
- How do we measure model performance in production?
- How do we tackle data drift?
- How do we tackle model drift?

Once these questions are answered, the next step is to build a pipeline around these models. The main purpose of the pipeline would be such that when new data comes in, the pre-trained models are able to answer questions based on new datasets.

Let's use an example to better understand these pipelines. We'll continue with the first example of an organization selling computer hardware:

1. Suppose the organization wants to build a recommender system on its website that recommends to users which products to buy.
2. The data scientists have built a prototype model that works well with test data. Now, they want to deploy it to production.

3. To deploy this model, the machine learning engineers would have to see how they can incorporate this model on the website.
4. They would start by getting the data ingested from the website to get the user information.
5. Once they have the information, they pass it through the data pipeline to clean and join the data.
6. They might also want to add some precomputed features to the model, such as the time of the year, to get a better idea of whether it's a holiday season and some special deals are going on.
7. Then, they would need a REST API endpoint to get the latest recommendations for each user on the website.
8. After that, the website needs to be connected to the REST endpoint to serve the actual customers.
9. Once these models are deployed on live systems (the website, in our example), there needs to be a monitoring system for any errors and changes in either the model or the data. This is known as **model drift** and **data drift**, respectively.

Data drift

Data may change over time. In our example, people's preferences may change with time, or with seasonality, data may be different. For example, during a holiday season, people's preferences might slightly change because they are looking to get presents for their friends and family, so recommending relevant products based on these preferences is of paramount importance for a business. Monitoring these trends and changes in the data would result in better models over time and would ultimately benefit the business.

Model drift

Similar to data drift, we also have the concept of model drift. This means that the model changes over time, and the old model that was initially built is not the most performant in terms of recommending items to website visitors. With changing data, the model also needs to be updated from time to time. To get a sense of when a model needs to be updated, we need to have monitoring in place for models as well. This monitoring would constantly compare old model results with the new data and see whether model performance is degrading. If that's the case, it's time to update the model.

This whole life cycle of model deployment is typically the responsibility of machine learning engineers. Note that the process would slightly vary for different problems, but the overall idea remains the same.

Summary

In this chapter, we learned about the basics of Apache Spark and why Spark is becoming a lot more prevalent in the industry for big data applications. We also learned about the different components of Spark and how these components are helpful in terms of application development. Then, we discussed the different roles that are present in the industry today and who can make use of Spark's capabilities. Finally, we discussed the modern-day uses of Spark in different industry use cases.

Sample questions

Although these questions are not part of the Spark certification, it's good to answer these to assess your understanding of the basics of Spark:

1. What are the core components of Spark?
2. When do we want to use Spark Streaming?
3. What is a look-back mechanism in Spark Streaming?
4. What are some good use cases for Spark?
5. Which roles in an organization should use Spark?

3
Spark Architecture and Transformations

Spark approaches data processing differently than traditional tools and technologies. To understand Spark's unique approach, we will have to understand its basic architecture. A deep dive into Spark's architecture and its components will give you an idea of how Spark achieves its ground-breaking processing speeds for big data analytics.

In this chapter, you will learn about the following broader topics:

- Spark architecture and execution hierarchy
- Different Spark components
- The roles of the Spark driver and Spark executor
- Different deployment modes in Spark
- Transformations and actions as Spark operations

By the end of this chapter, you will have valuable insights into Spark's inner workings and know how to apply this knowledge effectively for your certification test.

Spark architecture

In the previous chapters, we discussed that Apache Spark is an open source, distributed computing framework designed for big data processing and analytics. Its architecture is built to handle various workloads efficiently, offering speed, scalability, and fault tolerance. Understanding the architecture of Spark is crucial for comprehending its capabilities in processing large volumes of data.

The components of Spark architecture work in collaboration to process data efficiently. The following major components are involved:

- Spark driver
- SparkContext
- Cluster manager
- Worker node
- Spark executor
- Task

Before we talk about any of these components, it's important to understand their execution hierarchy to know how each component interacts when a Spark program starts.

Execution hierarchy

Let's look at the execution flow of a Spark application with the help of the architecture depicted in *Figure 3.1*:

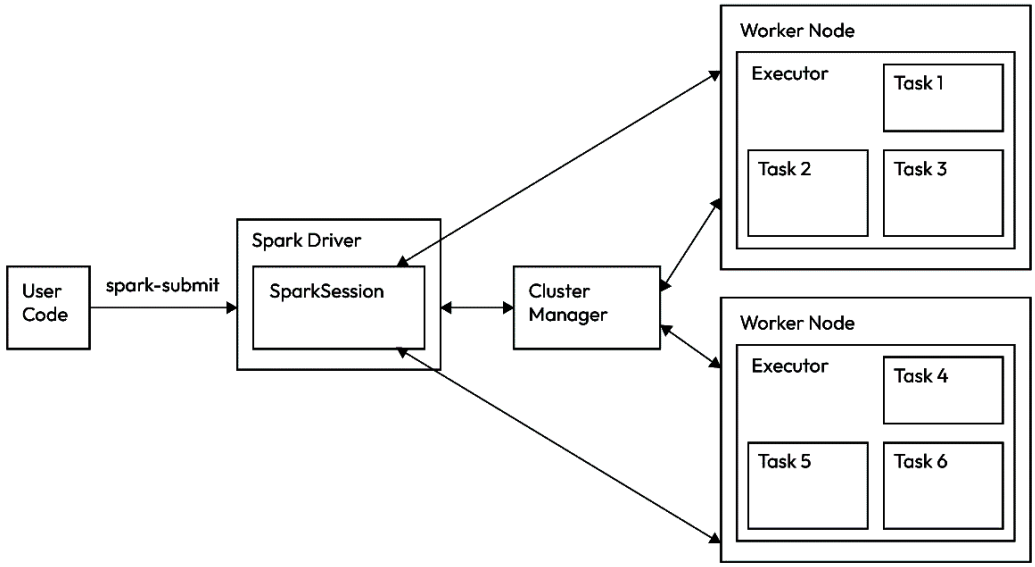


Figure 3.1: Spark architecture

These steps outline the flow from submitting a Spark job to freeing up resources when the job is completed:

1. Spark executions start with a user submitting a `spark-submit` request to the Spark engine. This will create a Spark application. Once an action is performed, it will result in a **job** being created.
2. This request will initiate communication with the cluster manager. In turn, the cluster manager initializes the Spark driver to execute the `main()` method of the Spark application. To execute this method, `SparkSession` is created.
3. The driver starts communicating with the cluster manager and asks for resources to start planning for execution.
4. The cluster manager then starts the executors, which can communicate with the driver directly.
5. The driver creates a logical plan, known as a **directed acyclic graph** (**DAG**), and physical plan for execution based on the total number of tasks required to be executed.
6. The driver also divides data to be run on each executor, along with tasks.
7. Once each task finishes running, the driver gets the results.
8. When the program finishes running, the `main()` method exits and Spark frees all executors and driver resources.

Now that you understand the execution hierarchy, let's discuss each of Spark's components in detail.

Spark components

Let's dive into the inner workings of each Spark component to understand how each of them plays a crucial role in empowering efficient distributed data processing.

Spark driver

The Spark driver is the core of the intelligent and efficient computations in Spark. Spark follows an architecture that is commonly known as the **master-worker architecture** in network topology. Consider the Spark driver as a master and Spark executors as slaves. The driver has control and knowledge of all the executors at any given time. It is the responsibility of the driver to know how many executors are present and if any executor has failed so that it can fall back on its alternative. The Spark driver also maintains communication with executors all the time. The driver runs on the master node of a machine or cluster. When a Spark application starts running, the driver keeps up with all the required information that is needed to run the application successfully.

As shown in *Figure 3.1*, the driver node contains `SparkSession`, which is the entry point of the Spark application. Previously, this was known as the `SparkContext` object, but in Spark 2.0, `SparkSession` handles all contexts to start execution. The application's main method runs on the driver to coordinate the whole application. It runs on its own **Java Virtual Machine** (**JVM**). Spark driver can run as an independent process or it can run on one of the worker nodes, depending on the architecture.

The Spark driver is responsible for dividing the application into smaller entities for execution. These entities are known as **tasks**. You will learn more about tasks in the upcoming sections of this chapter. The Spark driver also decides what data the executor will work on and what tasks are run on which executor. These tasks are scheduled to run on the executor nodes with the help of the cluster manager. This information that is driven by the driver enables fault tolerance. Since the driver has all the information about the number of available workers and the tasks that are running on each of them alongside data in case a worker fails, that task can be reassigned to a different cluster. Even if a task is taking too long to run, it can be assigned to another executor if that gets free. In that case, whichever executor returns the task earlier would prevail. The Spark driver also maintains metadata about the **Resilient Distributed Dataset** (**RDD**) and its partitions.

It is the responsibility of the Spark driver to design the complete execution map. It determines which tasks run on which executors, as well as how the data is distributed across these executors. This is done by creating RDDs internally. Based on this distribution of data, the operations that are required are determined, such as transformations and actions that are defined in the program. A DAG is created based on these decisions. The Spark driver optimizes the logical plan (DAG) and finds the best possible execution strategy for the DAG, in addition to determining the most optimal location for the execution of a particular task. These executions are done in parallel. The executors simply follow these commands without doing any optimization on their end.

For performance considerations, it is optimal to have the Spark driver work close to the executor. This reduces the latency by a great deal. This means that there would be less delay in the response time of the processes. Another point to note here is that this is true for the data as well. The executor reading the data close to it would have better performance than otherwise. Ideally, the driver and worker nodes should be run in the same **local area network** (**LAN**) for the best performance.

The Spark driver also creates a web UI for the execution details. This UI is very helpful in determining the performance of the application. In cases where troubleshooting is required and some bottlenecks need to be identified in the Spark process, this UI is very helpful.

SparkSession

`SparkSession` is the main point of entry and interaction with Spark. As discussed earlier, in the previous versions of Spark, `SparkContext` used to play this role, but in Spark 2.0, `SparkSession` can be created for this purpose. The Spark driver creates a `SparkSession` object to interact with the cluster manager and get resource allocation through it.

In the lifetime of the application, `SparkSession` is also used to interact with all the underlying Spark APIs. We talked about different Spark APIs in *Chapter 2* namely, SparkSQL, Spark Streaming, MLlib, and GraphX. All of these APIs use `SparkSession` from its core to interact with the Spark application.

`SparkSession` keeps track of Spark executors throughout the application's execution.

Cluster manager

Spark is a distributed framework, which requires it to have access to computing resources. This access is governed and controlled by a process known as the cluster manager. It is the responsibility of the cluster manager to allocate computing resources for the Spark application when the application execution starts. These resources become available at the request of the application master. In the Apache Spark ecosystem, the **application master** plays a crucial role in managing and coordinating the execution of Spark applications within a distributed cluster environment. It's an essential component that's responsible for negotiating resources, scheduling tasks, and monitoring the application's execution.

Once the resources are available, the driver is made aware of those resources. It's the responsibility of the driver to manage these resources based on tasks that need to be executed by the Spark application. Once the application has finished execution, these resources are released back to the cluster manager.

Applications have their dedicated executor processes that parallelize how tasks are run. The advantage is that each application is independent of the other and runs on its own schedule. Data also becomes independent for each of these applications, so data sharing can only take place by writing data to disk so that it can be shared across applications.

Cluster modes

Cluster modes define how Spark applications utilize cluster resources, manage task execution, and interact with cluster managers for resource allocation.

If there is more than one user sharing resources on the cluster, be it Spark applications or other applications that need cluster resources, they have to be managed based on different modes. There are two types of modes available for cluster managers – standalone client mode and cluster mode. The following table highlights some of the differences between the two:

Client Mode	**Cluster Mode**
In client mode, the driver program runs on the machine where the Spark application is submitted.	In cluster mode, the driver program runs within the cluster, on one of the worker nodes.
The driver program is responsible for orchestrating the execution of the Spark application, including creating `SparkContext` and coordinating tasks.	The cluster manager is responsible for launching the driver program and allocating resources for execution.
The client machine interacts directly with the cluster manager to request resources and launch executors on worker nodes.	Once the driver program is launched, it coordinates with the cluster manager to request resources and distribute tasks to worker nodes.
It may not be suitable for production deployments with large-scale applications.	It is commonly used for production deployments as it allows for better resource utilization and scalability. It also ensures fault tolerance.

Table 3.1: Client mode versus cluster mode

Now, we will talk about different deployment modes and their corresponding managers in Spark:

- **Built-in standalone mode** (**Spark's native manager**): A simple cluster manager bundled with Spark that's suitable for small to medium-scale deployments without external dependencies.
- **Apache YARN** (**Hadoop's resource manager**): Integrated with Spark, YARN enables Spark applications to share Hadoop's cluster resources efficiently.
- **Apache Mesos** (**resource sharing platform**): Mesos offers efficient resource sharing across multiple applications, allowing Spark to run alongside other frameworks.

We will talk more about deployment modes later in this chapter.

Spark executors

Spark executors are the processes that run on the worker node and execute tasks sent by the driver. The data is stored in memory primarily but can also be written to disk storage closest to them. Driver launches the executors based on the DAG that Spark generates for its execution. Once the tasks have finished executing, executors send the results back to the driver.

Since the driver is the main controller of the Spark application, if an executor fails or takes too long to execute a task, the driver can choose to send that task over to other available executors. This ensures reliability and fault tolerance in Spark. We will read more about this later in this chapter.

It is the responsibility of the executor to read data from external sources that are needed to run the tasks. It can also write its partitioned data to the disk as needed. All processing for a task is done by the executor.

The key functions of an executor are as follows:

- **Task execution**: Executors run tasks assigned by the Spark application, processing data stored in RDDs or DataFrames
- **Resource allocation**: Each Spark application has a set of executors allocated by the cluster manager for managing resources such as CPU cores and memory

In Apache Spark, the concepts of job, stage, and task form the fundamental building blocks of its distributed computing framework. Understanding these components is essential to grasp the core workings of Spark's parallel processing and task execution. See *Figure 3.2* to understand the relationship between these concepts while we discuss them in detail:

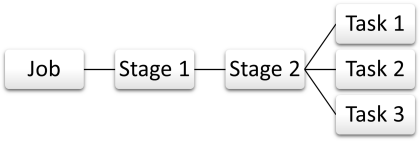


Figure 3.2: Interaction between jobs, stages, and tasks

Let's take a closer look:

- **Job**: The Spark application will initiate multiple jobs when the application starts running. These jobs can be executed in parallel, wherein each job can consist of multiple tasks. A job gets initiated when a Spark action is called (such as `collect`). We will learn more about actions later. When an action (such as `collect` or `count`) is invoked on a dataset, it triggers the execution of one or more jobs.

  A job consists of several stages, each containing tasks that execute a set of transformations on data partitions.

- **Stage**: Each job is divided into stages that may depend on other stages. Stages act as transformation boundaries – they are created at the boundaries of wide transformations that require data shuffling across partitions. If a stage is dependent on outputs from a previous stage, then this stage would not begin execution until the previous dependent stages have finished execution.

  Each stage is divided into a set of tasks to be executed on the cluster nodes, processing data in parallel.

- **Task**: A task is the smallest unit of execution in Spark. It is the smallest object compiled and run by Spark to perform a group of operations. It is executed on a Spark executor. Tasks are essentially a series of operations such as filter, groupBy, and others.

  Tasks run in parallel across executors. They can be run on multiple nodes and are independent of each other. This is done with the help of slots. Each task processes a portion of the data partition. Occasionally, a group of these tasks has to finish execution to begin the next task's execution.

Now that we understand these concepts, let's see why they are significant in Spark:

- **Parallel processing**: Executors, jobs, stages, and tasks collaborate to enable parallel execution of computations, optimizing performance by leveraging distributed computing
- **Task granularity and efficiency**: Tasks divide computations into smaller units, facilitating efficient resource utilization and parallelism across cluster nodes

Next, we will move on to discuss a significant concept that enhances efficiency in computation.

Partitioning in Spark

In Apache Spark, partitioning is a critical concept that's used to divide data across multiple nodes in a cluster for parallel processing. Partitioning improves data locality, enhances performance, and enables efficient computation by distributing data in a structured manner. Spark supports both static and dynamic partitioning strategies to organize data across the cluster nodes:

- **Static partitioning of resources**: Static partitioning is available on all cluster managers. With static partitioning, maximum resources are allocated to each application and these resources remain dedicated to these applications during their lifetime.

- **Dynamic sharing of resources**: Dynamic partitioning is only available on Mesos. When dynamically sharing resources, the Spark application gets fixed and independent memory allocation, such as static partitioning. The major difference is that when the tasks are not being run by an application, these cores can be used by other applications as well.

Let's discuss why partitioning is significant:

- **Performance optimization**: Effective partitioning strategies, whether static or dynamic, significantly impact Spark's performance by improving data locality and reducing data shuffle
- **Adaptability and flexibility**: Dynamic partitioning provides adaptability to varying data sizes or distribution patterns without manual intervention
- **Control and predictability**: Static partitioning offers control and predictability over data distribution, which can be advantageous in specific use cases

In summary, partitioning strategies – whether static or dynamic – in Spark play a crucial role in optimizing data distribution across cluster nodes, improving performance, and ensuring efficient parallel processing of data.

Apache Spark offers different cluster and deployment modes to run applications across distributed computing environments. We'll take a look at them in the next section.

Deployment modes

There are different deployment modes available in Spark. These deployment modes define how Spark applications are launched, executed, and managed in diverse computing infrastructures. Based on these different deployment modes, it gets decided where the Spark driver, executor, and cluster manager will run.

The different deployment modes that are available in Spark are as follows:

- **Local**: In local mode, the Spark driver and executor run on a single JVM and the cluster manager runs on the same host as the driver and executor.
- **Standalone**: In standalone mode, the driver can run on any node of the cluster and the executor will launch its own independent JVM. The cluster manager can remain on any of the hosts in the cluster.
- **YARN (client)**: In this mode, the Spark driver runs on the client and YARN's resource manager allocates containers for executors on NodeManagers.
- **YARN (cluster)**: In this mode, the Spark driver runs with the YARN application master while YARN's resource manager allocates containers for executors on NodeManagers.
- **Kubernetes**: In this mode, the driver runs in Kubernetes pods. Executors have their own pods.

Let's look at some points of significance regarding the different deployment modes:

- **Resource utilization**: Different deployment modes optimize resource utilization by determining where the driver program runs and how resources are allocated between the client and the cluster.
- **Accessibility and control**: Client mode offers easy accessibility to driver logs and outputs, facilitating development and debugging, while cluster mode utilizes cluster resources more efficiently for production workloads.
- **Integration with container orchestration**: Kubernetes deployment mode enables seamless integration with containerized environments, leveraging Kubernetes' orchestration capabilities for efficient resource management.

There are some considerations to keep in mind while choosing deployment modes:

- **Development versus production**: Client mode is suitable for development and debugging, while cluster mode is ideal for production workloads
- **Resource management**: Evaluate the allocation of resources between client and cluster nodes based on the application's requirements
- **Containerization needs**: Consider Kubernetes deployment for containerized environments, leveraging Kubernetes features for efficient container management

In summary, deployment modes in Apache Spark provide flexibility in how Spark applications are launched and executed, catering to different development, production, and containerized deployment scenarios.

Next, we will look at RDDs, which serve as foundational data abstractions in Apache Spark, enabling distributed processing, fault tolerance, and flexibility in handling large-scale data operations. While RDDs continue to be a fundamental concept, Spark's DataFrame and Dataset APIs offer advancements in structured data processing and performance optimization.

RDDs

Apache Spark's RDD stands as a foundational abstraction that underpins the distributed computing capabilities within the Spark framework. RDDs serve as the core data structure in Spark, enabling fault-tolerant and parallel operations on large-scale distributed datasets and they are immutable. This means that they cannot be changed over time. For any operations, a new RDD has to be generated from the existing RDD. When a new RDD originates from the original RDD, the new RDD has a pointer to the RDD it is generated from. This is the way Spark documents the lineage for all the transformations taking place on an RDD. This lineage enables **lazy evaluation** in Spark, which generates DAGs for different operations.

This immutability and lineage gives Spark the ability to reproduce any DataFrame in case of failure and it makes fault-tolerant by design. Since RDD is the lowest level of abstraction in Spark, all other datasets built on top of RDDs share these properties. The high-level DataFrame API is built on top of the low-level RDD API as well, so DataFrames also share the same properties.

RDDs are also partitioned by Spark and each partition is distributed to multiple nodes in the cluster.

Here are some of the key characteristics of Spark RDDs:

- **Immutable nature**: RDDs are immutable, ensuring that once created, they cannot be altered, allowing for a lineage of transformations.
- **Resilience through lineage**: RDDs store lineage information, enabling reconstruction of lost partitions in case of failures. Spark is designed to be fault-tolerant. Therefore, if an executor on a worker node fails while calculating an RDD, that RDD can be recomputed by another executor using the lineage that Spark has created.
- **Partitioned data**: RDDs divide data into partitions, distributed across multiple nodes in a cluster for parallel processing.
- **Parallel execution**: Spark executes operations on RDDs in parallel across distributed partitions, enhancing performance.

Let's discuss some more characteristics in detail.

Lazy computation

RDDs support lazy evaluation, deferring execution of transformations until an action is invoked. The way Spark achieves its efficiency in processing and fault tolerance is through lazy evaluation. Code execution in Spark is delayed. Unless an action is called an operation, Spark does not start code execution. This helps Spark achieve optimization as well. For all the transformations and actions, Spark keeps track of the steps in the code that need to be executed by creating a DAG for these operations. Because Spark creates the query plan before execution, it can make smart decisions about the hierarchy of execution as well. To achieve this, one of the features Spark uses is called **predicate pushdown**.

Predicate pushdown means that Spark can prioritize the operations to make them the most efficient. One example can be a filter operation. A filter operation would generally reduce the amount of data that the subsequent operations have to work with if the filter operation can be applied before other transformations. This is exactly how Spark operates. It will execute filters as early in the process as possible, thus making the next operations more performant.

This also implies that Spark jobs would fail only at execution time. Since Spark uses lazy evaluation, until an action is called, the code is not executed and certain errors can be missed. To catch these errors, Spark code would need to have an action for execution and hence error handling.

Transformations

Transformations create new RDDs by applying functions to existing RDDs (for example, `map`, `filter`, and `reduce`). Transformations are operations that do not result in any code execution. These statements result in Spark creating a DAG for execution. Once that DAG is created, Spark would need an action operation in the end to run the code. Due to this, when certain developers try to time the code from Spark, they see that certain operations' runtime is very fast. The reason could be that the code is only comprised of transformations until that point. Since no action is present, the code doesn't run. To accurately measure the runtime of each operation, actions have to be called to force Spark to execute those statements.

Here are some of the operations that can be classified as transformations:

- `orderBy()`
- `groupBy()`
- `filter()`
- `select()`
- `join()`

When these commands are executed, they are evaluated lazily. This means all these operations on DataFrames result in a new DataFrame, but they are not executed until an action is followed by them. This would return a DataFrame or RDD when it is triggered by an action.

Actions and computation execution

Actions (for example, `collect`, `count`, and `saveAsTextFile`) prompt the execution of transformations on RDDs. Execution is triggered by actions only, not by transformations. When an action is called, this is when Spark starts execution on the DAG it created during the analysis phase of code. With the DAG created, Spark creates multiple query plans based on its internal optimizations. Then, it executes the plan that is the most efficient and cost-effective. We will discuss query plans later in this book.

Here are some of the operations that can be classified as actions:

- `show()`
- `take()`
- `count()`
- `collect()`
- `save()`
- `foreach()`
- `first()`

All of these operations would result in Spark triggering code execution and thus operations are run.

Let's take a look at the following code to understand these concepts better:

```
# Python
>>> df = spark.read.text("{path_to_data_file}")
>>> names_df = df.select(col("firstname"),col("lastname"))
>>> names_df.show()
```

In the preceding code, until line 2, nothing would be executed. On line 3, an action is triggered and thus it triggers the whole code execution. Therefore, if you give the wrong data path in line 1 or the wrong column names in line 2, Spark will not detect this until it runs line 3. This is a different paradigm than most other programming paradigms. This is what we call lazy evaluation in Spark.

Actions bring about computation and collect results to be sent to the driver program.

Now that we've covered the basics of transformations and actions in Spark, let's move on to understanding the two types of transformations it offers.

Types of transformations

Apache Spark's transformations are broadly categorized into narrow and wide transformations, each serving distinct purposes in the context of distributed data processing.

Narrow transformations

Narrow transformations, also known as local transformations, operate on individual partitions of data without shuffling or redistributing data across partitions. These transformations enable Spark to process data within a single partition independently. In narrow transformations, Spark will work with a single input partition and a single output partition. This means that these types of transformations would result in an operation that can be performed on a single partition. The data doesn't have to be taken from multiple partitions or written back to multiple partitions. This results in operations that don't require shuffle.

Here are some of their characteristics:

- **Partition-level operation**: Narrow transformations process data at the partition level, performing computations within each partition
- **Independence and local processing**: They do not require data movement or communication across partitions, allowing parallel execution within partitions
- **Examples**: Operations such as `map`, `filter`, and `flatMap` are typical examples of narrow transformations

Now, let's look at their significance:

- **Efficiency and speed**: Narrow transformations are efficient as they involve local processing within partitions, reducing communication overhead
- **Parallelism**: They facilitate maximum parallelism by operating on partitions independently, optimizing performance

Wide transformations

Wide transformations, also termed global or shuffle-dependent transformations, involve operations that require data shuffling and redistribution across partitions. These transformations involve dependencies between partitions, necessitating data exchange. With wide transformations, Spark will use the data present on multiple partitions and it could also write back the results to multiple partitions. These transformations would force a shuffle operation, so they are also referred to as shuffle transformations.

Wide transformations are complex operations. They would need to write the results out in between operations if needed and they also have to aggregate data across different machines in certain cases.

Here are some of their characteristics:

- **Data shuffling**: Wide transformations reorganize data across partitions by reshuffling or aggregating data from multiple partitions
- **Dependency on multiple partitions**: They depend on data from various partitions, leading to the exchange and reorganization of data across the cluster
- **Examples**: Operations such as `groupBy`, `join`, and `sortByKey` are typical examples of wide transformations

Now, let's look at their significance:

- **Network and disk overhead**: Wide transformations introduce network and disk overhead due to data shuffling, impacting performance
- **Stage boundary creation**: They define stage boundaries within a Spark job, resulting in distinct stages during job execution

The following are the differences between narrow and wide transformations:

- **Data movement**: Narrow transformations process data within partitions locally, minimizing data movement, while wide transformations involve data shuffling and movement across partitions
- **Performance impact**: Narrow transformations typically offer higher performance due to reduced data movement, whereas wide transformations involve additional overhead due to data shuffling
- **Parallelism scope**: Narrow transformations enable maximum parallelism within partitions, while wide transformations might limit parallelism due to dependency on multiple partitions

In Apache Spark, understanding the distinction between narrow and wide transformations is crucial. Narrow transformations excel in local processing within partitions, optimizing performance, while wide transformations, although necessary for certain operations, introduce overhead due to data shuffling and global reorganization across partitions.

Let's look at the significance of Spark RDDs:

- **Distributed data processing**: RDDs enable distributed processing of large-scale data across a cluster of machines, promoting parallelism and scalability
- **Fault tolerance and reliability**: Their immutability and lineage-based recovery ensure fault tolerance and reliability in distributed environments
- **Flexibility in operations**: RDDs support a wide array of transformations and actions, allowing diverse data manipulations and processing operations

Evolution and alternatives

While RDDs remain fundamental, Spark's DataFrame and Dataset APIs offer optimized, higher-level abstractions suitable for structured data processing and optimization.

Spark RDDs serve as the bedrock of distributed data processing within the Apache Spark framework, providing immutability, fault tolerance, and the foundational structure for performing parallel operations on distributed datasets. Although RDDs are fundamental, Spark's DataFrame and Dataset APIs offer advancements in performance and structured data processing, catering to various use cases and preferences within the Spark ecosystem.

Summary

In this chapter, we learned about Spark's architecture and its inner workings. This exploration of Spark's distributed computing landscape covered different Spark components, such as the Spark driver and `SparkSession`. We also talked about the different types of cluster managers available in Spark. Then, we touched on different types of partitioning regarding Spark and its deployment modes.

Next, we discussed Spark executors, jobs, stages, and tasks and highlighted the differences between them before learning about RDDs and their transformation types, learning more about narrow and wide transformations.

These concepts form the foundation for harnessing Spark's immense capabilities in distributed data processing and analytics.

In the next chapter, we will discuss Spark DataFrames and their corresponding operations.

Sample questions

Question 1:

What's true about Spark's execution hierarchy?

A. In Spark's execution hierarchy, a job may reach multiple stage boundaries.
B. In Spark's execution hierarchy, manifests are one layer above jobs.
C. In Spark's execution hierarchy, a stage comprises multiple jobs.
D. In Spark's execution hierarchy, executors are the smallest unit.
E. In Spark's execution hierarchy, tasks are one layer above slots.

Question 2:

What do executors do?

A. Executors host the Spark driver on a worker-node basis.
B. Executors are responsible for carrying out work that they get assigned by the driver.
C. After the start of the Spark application, executors are launched on a per-task basis.
D. Executors are located in slots inside worker nodes.
E. The executors' storage is ephemeral and as such it defers the task of caching data directly to the worker node thread.

Answers

1. A
2. B

669FD03CAE074

Part 3: Spark Operations

In this part, we will cover Spark DataFrames and their operations, emphasizing their role in structured data processing and analytics. This will include DataFrame creation, manipulation, and various operations such as filtering, aggregations, joins, and groupings, demonstrated through illustrative examples. Then, we will discuss advanced operations and optimization techniques, including broadcast variables, accumulators, and custom partitioning. This part also talks about performance optimization strategies, highlighting the significance of adaptive query execution and offering practical tips for enhancing Spark job performance. Furthermore, we will explore SQL queries in Spark, focusing on its SQL-like querying capabilities and interoperability with the DataFrame API. Examples will illustrate complex data manipulations and analytics through SQL queries in Spark.

This part has the following chapters:

- *Chapter 4*, *Spark DataFrames and their Operations*
- *Chapter 5*, *Advanced Operations and Optimizations in Spark*
- *Chapter 6, SQL Queries in Spark*

4

Spark DataFrames and their Operations

In this chapter, we will learn about a few different APIs in Spark and talk about their features. We will also get started with Spark's DataFrame operations and look at different data viewing and manipulation techniques such as filtering, adding, renaming, and dropping columns available in Spark.

We will cover these concepts under the following topics:

- The Spark DataFrame API
- Creating DataFrames
- Viewing DataFrames
- Manipulating DataFrames
- Aggregating DataFrames

By the end of this chapter, you will know how to work with PySpark DataFrames. You'll also discover various data manipulation techniques and see how you can view data after manipulating it.

Getting Started in PySpark

In the previous chapters, we discussed that Spark primarily uses four languages, which are Scala, Python, R, and SQL. When any of these languages are used, the underlying execution engine is the same. This provides the necessary unification we talked about in *Chapter 2*. This means that developers can use any language of their choice and can also switch between different APIs in applications.

For the context of this book, we're going to focus on Python as the primary language. Spark used with Python is called **PySpark**.

Let's get started with the installation of Spark.

Installing Spark

To get started with Spark, you would have to first install it on your computer. There are a few ways to install Spark. We will focus on just one in this section.

PySpark provides **pip** installation from **PyPI**. You can install it as follows:

```
pip install pyspark
```

Once Spark is installed, you will need to create a Spark session.

Creating a Spark session

Once you have installed Spark on your system, you can get started with creating a Spark session. A Spark session is the entry point of any Spark application. To create a Spark session, you will initialize it in the following way:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
```

When you are running your code in the Spark shell, the Spark session is automatically created for you so you don't have to manually execute this code to create a Spark session. This session is usually created in a variable called `spark`.

It is important to note that we can only create a single spark session at any given time. Duplicating a Spark session is not possible in Spark.

Now, let's take a look at different data APIs in Spark DataFrames.

Dataset API

Dataset is a newer interface added to *Spark 1.6*. It is a distributed collection of data. The Dataset API is available in Java and Scala, but not in Python and R. The Dataset API uses **Resilient Distributed Datasets** (**RDDs**) and hence provides additional features of RDDs, such as fixed typing. It also uses Spark SQL's optimized engine for faster queries.

Since a lot of the data engineering and data science community is already familiar with Python and uses it extensively for data architectures in production, PySpark also provides an equivalent API for DataFrames for this purpose. Let's take a look at it in the next section.

DataFrame API

The motivation of Spark DataFrames comes from Pandas DataFrames in Python. A DataFrame is essentially a set of rows and columns. You can think of it like a table where you have table headers as column names and below these headers are data arranged accordingly. This table-like format has been part of computations for a long time in tools such as relational databases and comma-separated files.

Spark's DataFrame API is built on top of RDDs. The underlying structures to store the data are still RDDs but DataFrames create an abstraction on top of the RDDs to hide its complexity. Just as RDDs are lazily evaluated and are immutable, DataFrames are also evaluated lazily and are immutable. If you can remember from previous chapters, lazy evaluation gives Spark performance gains and optimization by running the computations only when needed. This also gives Spark a large number of optimizations in its DataFrames by planning how to best compute the operations. The computations start when an action is called on a DataFrame. There are a lot of different ways to create Spark DataFrames. We will learn about some of those in this chapter.

Let's take a look at what a DataFrame is in Spark.

Creating DataFrame operations

As we have already discussed, DataFrames are the main building blocks of Spark data. They consist of rows and column data structures.

DataFrames in PySpark are created using the `pyspark.sql.SparkSession.createDataFrame` function. You can use lists, lists of lists, tuples, dictionaries, Pandas DataFrames, RDDs, and `pyspark.sql.Rows` to create DataFrames.

Spark DataFrames also has an argument named **schema** that specifies the schema of the DataFrame. You can either choose to specify the schema explicitly or let Spark infer the schema from the DataFrame itself. If you don't specify this argument in the code, Spark will infer the schema on its own.

There are different ways to create DataFrames in Spark. Some of them are explained in the following sections.

Using a list of rows

The first way to create DataFrames we see is by using rows of data. You can think of rows of data as lists. They would share common header values for each of the values in the list.

Here's the code to use when creating a new DataFrame using rows of data:

```
import pandas as pd
from datetime import datetime, date
from pyspark.sql import Row

data_df = spark.createDataFrame([
    Row(col_1=100, col_2=200., col_3='string_test_1', col_4=date(2023,
1, 1), col_5=datetime(2023, 1, 1, 12, 0)),
    Row(col_1=200, col_2=300., col_3='string_test_2', col_4=date(2023,
2, 1), col_5=datetime(2023, 1, 2, 12, 0)),
    Row(col_1=400, col_2=500., col_3='string_test_3', col_4=date(2023,
3, 1), col_5=datetime(2023, 1, 3, 12, 0))
])
```

As a result, you will see a DataFrame with our specified columns and their data types:

```
DataFrame[col_1: bigint, col_2: double, col_3: string, col_4: date,
col_5: timestamp]
```

Now, we'll see how we can specify the schema for a Spark DataFrame explicitly.

Using a list of rows with schema

The schema of a DataFrame defines what would be the different data types present in each of the rows and columns of a DataFrame. Explicitly defining schema helps in cases where we want to enforce certain data types to our datasets.

Now, we will explicitly tell Spark which schema to use for the DataFrame that we're creating. Notice that the majority of the code remains the same—we're simply adding another argument named `schema` in the code for creating the DataFrame to explicitly tell which columns would have what kind of datatypes:

```
import pandas as pd
from datetime import datetime, date
from pyspark.sql import Row

data_df = spark.createDataFrame([
    Row(col_1=100, col_2=200., col_3='string_test_1', col_4=date(2023,
1, 1), col_5=datetime(2023, 1, 1, 12, 0)),
    Row(col_1=200, col_2=300., col_3='string_test_2', col_4=date(2023,
2, 1), col_5=datetime(2023, 1, 2, 12, 0)),
    Row(col_1=400, col_2=500., col_3='string_test_3', col_4=date(2023,
3, 1), col_5=datetime(2023, 1, 3, 12, 0))
], schema=' col_1 long, col_2 double, col_3 string, col_4 date, col_5
timestamp')
```

As a result, you will see a DataFrame with our specified columns and their data types:

```
data_df
DataFrame[col_1: bigint, col_2: double, col_3: string, col_4: date,
col_5: timestamp]
```

Now, let's take a look at how we can create DataFrames using Pandas DataFrames.

Using Pandas DataFrames

DataFrames can also be created using Pandas DataFrames. To achieve this, you would need to create a DataFrame in Pandas first. Once that is created, you would then convert that DataFrame to a PySpark DataFrame. The following code demonstrates this process:

```
from datetime import datetime, date
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

rdd = spark.sparkContext.parallelize([
    (100, 200., 'string_test_1', date(2023, 1, 1), datetime(2023, 1,
1, 12, 0)),
    (200, 300., 'string_test_2', date(2023, 2, 1), datetime(2023, 1,
2, 12, 0)),
    (300, 400., 'string_test_3', date(2023, 3, 1), datetime(2023, 1,
3, 12, 0))
])
data_df = spark.createDataFrame(rdd, schema=['col_1', 'col_2',
'col_3', 'col_4', 'col_5'])
```

As a result, you will see a DataFrame with our specified columns and their data types:

```
DataFrame[col_1: bigint, col_2: double, col_3: string, col_4: date,
col_5: timestamp]
```

Now, let's take a look at how we can create DataFrames using tuples.

Using tuples

Another way to create DataFrames is through tuples. This means that we can create a tuple as a row and add each tuple as a separate row in the DataFrame. Each tuple would contain the data for each of the columns of the DataFrame. The following code demonstrates this:

```
import pandas as pd
from datetime import datetime, date
from pyspark.sql import Row

rdd = spark.sparkContext.parallelize([
    (100, 200., 'string_test_1', date(2023, 1, 1), datetime(2023, 1,
1, 12, 0)),
    (200, 300., 'string_test_2', date(2023, 2, 1), datetime(2023, 1,
2, 12, 0)),
    (300, 400., 'string_test_3', date(2023, 3, 1), datetime(2023, 1,
3, 12, 0))
])
data_df = spark.createDataFrame(rdd, schema=['col_1', 'col_2',
'col_3', 'col_4', 'col_5'])
```

As a result, you will see a DataFrame with our specified columns and their data types:

```
DataFrame[col_1: bigint, col_2: double, col_3: string, col_4: date,
col_5: timestamp]
```

Now, let's take a look at different ways we can view the DataFrames in Spark and see the results of the DataFrames that we just created.

How to view the DataFrames

There are different statements in Spark to view data. The DataFrames that we created in the previous section through different methods all yield the same result as the DataFrame. Let's look at a few different ways to view DataFrames.

Viewing DataFrames

The first way to show a DataFrame is through the `DataFrame.show()` statement. Here's an example:

```
data_df.show()
```

As a result, you will see a DataFrame with our specified columns and the data inside this DataFrame:

```
+-----+-----+-------------+----------+-------------------+
|col_1|col_2|   col_3     |  col_4   |       col_5       |
+-----+-----+-------------+----------+-------------------+
|  100|200.0|string_test_1|2023-01-01|2023-01-01 12:00:00|
|  200|300.0|string_test_2|2023-02-01|2023-01-02 12:00:00|
|  300|400.0|string_test_3|2023-03-01|2023-01-03 12:00:00|
+-----+-----+-------------+----------+-------------------+
```

We can also select the total rows that can be viewed in a single statement. Let's see how we can do that in the next topic.

Viewing top n rows

We can also be selective in the number of rows that can be viewed in a single statement. We can control that using a parameter in `DataFrame.show()`. Here's an example of looking at only the top two rows of the DataFrame.

If you specify *n* to be a specific number, then only those sets of rows would be shown. Here's an example:

```
data_df.show(2)
```

As a result, you will see a DataFrame with its top two rows:

```
+-----+-----+-------------+----------+-------------------+
|col_1|col_2|    col_3    |   col_4  |        col_5      |
+------+------+-----------+----------+-------------------+
|  100|200.0|string_test_1|2023-01-01|2023-01-01 12:00:00|
|  200|300.0|string_test_2|2023-02-01|2023-01-02 12:00:00|
------+-----+-------------+----------+-------------------+
only showing top 2 rows.
```

Viewing DataFrame schema

We can also choose to see the schema of the DataFrame using the `printSchema()` function:

```
data_df.printSchema()
```

As a result, you will see the schema of a DataFrame with our specified columns and their data types:

```
root
 |-- col_1: long (nullable = true)
 |-- col_2: double (nullable = true)
 |-- col_3: string (nullable = true)
 |-- col_4: date (nullable = true)
 |-- col_5: timestamp (nullable = true)
```

Viewing data vertically

When the data becomes too long to fit into the screen, it's sometimes useful to see the data in a vertical format instead of a horizontal table view. Here's an example of how you can view the data in a vertical format:

```
data_df.show(1, vertical=True)
```

As a result, you will see a DataFrame with our specified columns and their data but in a vertical format:

```
-RECORD 0------------------
 col_1   | 100
 col_2   | 200.0
 col_3   | string_test_1
 col_4   | 2023-01-01
 col_5   | 2023-01-01 12:00:00
only showing top 1 row
```

Viewing columns of data

When we just need to view the columns that exist in a DataFrame, we would use the following:

```
data_df.columns
```

As a result, you will see a list of the columns in the DataFrame:

```
['col_1', 'col_2', 'col_3', 'col_4', 'col_5']
```

Viewing summary statistics

Now, let's take a look at how we can view the summary statistics of a DataFrame:

```
Show the summary of the DataFrame
data_df.select('col_1', 'col_2', 'col_3').describe().show()
```

As a result, you will see a DataFrame with its summary statistics for each column defined:

```
+-------+-------+-------+-------------+
|summary| col_1 | col_2 |    col_3    |
+-------+-------+-------+-------------+
|  count|   3   |   3   |            3|
|   mean| 200.0 | 300.0 |         null|
| stddev| 100.0 | 100.0 |         null|
|    min| 100   | 200.0 |string_test_1|
|    max| 300   | 400.0 |string_test_3|
+-------+-------+-------+-------------+
```

Now, let's take a look at the collect statement.

Collecting the data

A collect statement is used when we want to get all the data that is being processed in different clusters back to the driver. When using a collect statement, we need to make sure that the driver has enough memory to hold the processed data. If the driver doesn't have enough memory to hold the data, we will get out-of-memory errors.

This is how you show the collect statement:

```
data_df.collect()
```

This statement will then show result as follows:

```
[Row(col_1=100, col_2=200.0, col_3='string_test_1', col_4=datetime.
date(2023, 1, 1), col_5=datetime.datetime(2023, 1, 1, 12, 0)),
```

```
 Row(col_1=200, col_2=300.0, col_3='string_test_2', col_4=datetime.
date(2023, 2, 1), col_5=datetime.datetime(2023, 1, 2, 12, 0)),
 Row(col_1=300, col_2=400.0, col_3='string_test_3', col_4=datetime.
date(2023, 3, 1), col_5=datetime.datetime(2023, 1, 3, 12, 0))]
```

There are a few ways to avoid out-of-memory errors. We will explore some of the options that avoid out-of-memory errors such as take, tail, and head statements. These statements return only a subset of the data and not all of the data in a DataFrame, therefore, they are very useful to inspect the data without having to lead all the data in driver memory.

Now, let's take a look at the take statement.

Using take

A take statement takes an argument for a number of elements to return from the top of a DataFrame. We will see how it is used in the following code example:

```
data_df.take(1)
```

As a result, you will see a DataFrame with its top row:

```
[Row(col_1=100, col_2=200.0, col_3='string_test_1', col_4=datetime.
date(2023, 1, 1), col_5=datetime.datetime(2023, 1, 1, 12, 0))]
```

In this example, we're only returning the first element of the DataFrame by giving `1` as an argument value to the `take()` function. Therefore, only one row is returned in the result.

Now, let's take a look at the tail statement.

Using tail

The tail statement takes an argument for a number of elements to return from the bottom of a DataFrame. We will see how it is used in the following code example:

```
data_df.tail(1)
```

As a result, you will see a DataFrame with its last row of data:

```
[Row(col_1=300, col_2=400.0, col_3='string_test_3', col_4=datetime.
date(2023, 3, 1), col_5=datetime.datetime(2023, 1, 3, 12, 0))]
```

In this example, we're only returning the last element of the DataFrame by giving `1` as an argument value to the `tail()` function. Therefore, only one row is returned in the result.

Now, let's take a look at the head statement.

Using head

The head statement takes an argument for a number of elements to return from the top of a DataFrame. We will see how it is used in the following code example:

```
data_df.head(1)
```

As a result, you will see a DataFrame with its top row of data:

```
[Row(col_1=100, col_2=200.0, col_3='string_test_1', col_4=datetime.
date(2023, 1, 1), col_5=datetime.datetime(2023, 1, 1, 12, 0))]
```

In this example, we're only returning the first element of the DataFrame by giving `1` as an argument value to the `head()` function. Therefore, only one row is returned in the result.

Now, let's take a look at how we can count the number of rows in a DataFrame.

Counting the number of rows of data

When we just need to count the number of rows in a DataFrame, we would use the following:

```
data_df.count()
```

As a result, you will see the total count of rows in a DataFrame:

```
3
```

In PySpark, several methods are available for retrieving data from a DataFrame or RDD, each with its own characteristics and use cases. Here's a summary of the major differences between take, collect, show, head, and tail that we used earlier in this section for data retrieval.

take(n)

This function returns an array containing the first *n* elements from the DataFrame or RDD

- It is useful for quickly inspecting a small subset of the data
- It performs a lazy evaluation, meaning it only computes the required number of elements

collect()

This function retrieves all elements from the DataFrame or RDD and returns them as a list

- It should be used with caution as it brings all data to the driver node, which can lead to out-of-memory errors for large datasets
- It is suitable for small datasets or when working with aggregated results that fit into memory

show(n)

This function displays the first *n* rows of the DataFrame in a tabular format

- It is primarily used for visual inspection of data during **exploratory data analysis** (**EDA**) or debugging
- It provides a user-friendly display of data with column headers and formatting

head(n)

This function returns the first *n* rows of the DataFrame as a list of `Row` objects

- It is similar to `take(n)` but it returns `Row` objects instead of simple values
- It is often used when you need access to specific column values while working with structured data

tail(n)

This function returns the last *n* rows of the DataFrame

- It is useful for examining the end of the dataset, especially in cases where data is sorted in descending order
- It performs a more expensive operation compared to `head(n)` as it may involve scanning the entire dataset

In summary, `take` and `collect` are used to retrieve data elements, with `take` being more suitable for small subsets and `collect` for retrieving all data (with caution). `show` is used for visual inspection, `head` retrieves the first rows as `Row` objects, and `tail` retrieves the last rows of the dataset. Each method serves different purposes and should be chosen based on the specific requirements of the data analysis task.

When working with data in PySpark, sometimes, you will need to use some Python functions on the DataFrames. To achieve that, you will have to convert PySpark DataFrames to Pandas DataFrames. Now, let's take a look at how we can convert a Pyspark DataFrame to a Pandas DataFrame.

Converting a PySpark DataFrame to a Pandas DataFrame

At various times in your workflow, you will want to switch from a Pyspark DataFrame to a Pandas DataFrame. There are options to convert a PySpark DataFrame to a Pandas DataFrame. This option is `toPandas()`.

One thing to note here is that Python inherently is not distributed. Therefore, when a PySpark DataFrame is converted to Pandas, the driver would need to collect all the data in its memory. We need to make sure that the driver's memory is able to collect the data in itself. If the data is not able to fit in the driver's memory, it will cause an out-of-memory error.

Here's an example to see how we can convert a PySpark DataFrame to a Pandas DataFrame:

```
data_df.toPandas()
```

As a result, you will see a DataFrame with our specified columns and their data types:

	col_1	col_2	col_3	col_4	col_5
0	100	200.0	String_test_1	2023-01-01	2023-01-01 12:00:00
1	200	300.0	String_test_2	2023-02-01	2023-01-02 12:00:00
2	300	400.0	String_test_3	2023-03-01	2023-01-03 12:00:00

Table 4.1: DataFrame with the columns and data types specified by us

In the next section, we will learn about different data manipulation techniques. You will need to filter, slice, and dice the data based on different criteria for different purposes. Therefore, data manipulation is essential in working with data.

How to manipulate data on rows and columns

In this section, we will learn how to do different data manipulation operations on Spark DataFrames rows and columns.

We will start by looking at how we can select columns in a Spark DataFrame.

Selecting columns

We can use column functions for data manipulation at the column level in a Spark DataFrame. To select a column in a DataFrame, we would use the `select()` function like so:

```
from pyspark.sql import Column
data_df.select(data_df.col_3).show()
```

As a result, you will see only one column of the DataFrame with its data:

```
+-------------+
|    col_3    |
+-------------+
|string_test_1|
|string_test_2|
|string_test_3|
+-------------+
```

> The important thing to note here is that the resulting DataFrame with one column is a new DataFrame. Recalling what we discussed in *Chapter 3*, RDDs are immutable. The underlying data structure for DataFrames is RDDs, therefore, DataFrames are also immutable. This means every time you make a change to a DataFrame, a new DataFrame would be created out of it. You would either have to assign the resultant DataFrame to a new DataFrame or overwrite the original DataFrame.

There are some other ways to achieve the same result in PySpark as well. Some of those are demonstrated here:

```
data_df.select('col_3').show()
data_df.select(data_df['col_3']).show()
```

Once we select the required columns, there will be instances where you will need to add new columns to a DataFrame. We will now take a look at how we can create columns in a Spark DataFrame.

Creating columns

We can use a `withColumn()` function to create a new column in a DataFrame. To create a new column, we would need to pass the column name and column values to fill the column with. In the following example, we're creating a new column named `col_6` and putting a constant literal `A` in this column:

```
from pyspark.sql import functions as F
data_df = data_df.withColumn("col_6", F.lit("A"))
data_df.show()
```

As a result, you will see a DataFrame with an additional column named `col_6` filled with multiple `A` instances:

```
+-----+-----+-------------+----------+-------------------+-----+
|col_1|col_2|        col_3|     col_4|              col_5|col_6|
+-----+-----+-------------+----------+-------------------+-----+
|  100|200.0|string_test_1|2023-01-01|2023-01-01 12:00:00|    A|
|  200|300.0|string_test_2|2023-02-01|2023-01-02 12:00:00|    A|
|  300|400.0|string_test_3|2023-03-01|2023-01-03 12:00:00|    A|
+-----+-----+-------------+----------+-------------------+-----+
```

The `lit()` function is used to fill constant values in a column.

You can also delete columns that are no longer needed in a DataFrame. We will now take a look at how we can drop columns in a Spark DataFrame.

Dropping columns

If we need to drop a column from a Spark DataFrame, we would use the `drop()` function. We need to provide the name of the column to be dropped from the DataFrame. Here's an example of how to use this function:

```
data_df = data_df.drop("col_5")
data_df.show()
```

As a result, you will see that `col_5` is dropped from the DataFrame:

```
+-----+-----+-------------+----------+------+
|col_1|col_2|    col_3    |  col_4   | col_6|
+-----+-----+-------------+----------+------+
|  100|200.0|string_test_1|2023-01-01|     A|
|  200|300.0|string_test_2|2023-02-01|     A|
|  300|400.0|string_test_3|2023-03-01|     A|
+-----+-----+-------------+----------+------+
```

We have successfully dropped `col_5` from this DataFrame.

You can also drop multiple columns in the same drop statement as well:

```
data_df = data_df.drop("col_4", "col_5")
```

Also note that if we drop a column that does not exist in the DataFrame, it will not result in any errors. The resulting DataFrame would remain as it is.

Just like dropping columns, you can also update columns in Spark DataFrames. Now, we will take a look at how we can update columns in a Spark DataFrame.

Updating columns

Updating columns can also be done with the help of the `withColumn()` function in Spark. We need to provide the name of the column to be updated along with the updated value. Notice that we can also use this function to calculate some new values for the columns. Here's an example:

```
data_df.withColumn("col_2", F.col("col_2") / 100).show()
```

This will give us the following updated frame:

```
+-----+-----+-------------+----------+------+
|col_1|col_2|    col_3    |   col_4  | col_6|
+-----+-----+-------------+----------+------+
|  100|200.0|string_test_1|2023-01-01|     A|
|  200|300.0|string_test_2|2023-02-01|     A|
|  300|400.0|string_test_3|2023-03-01|     A|
+-----+-----+-------------+----------+------+
```

One thing to note here is the use of the `col` function when updating the column. This function is used for column-wise operators. If we don't use this function, our code will return an error.

You don't always have to update a column in a DataFrame if you only need to rename a column. Now, we will see how we can rename columns in a Spark DataFrame.

Renaming columns

For changing the name of a column, we would use the `withColumnRenamed()` function in Spark. We would need to provide the column name that needs to be changed along with the new column name. Here's the code to illustrate this:

```
data_df = data_df.withColumnRenamed("col_3", "string_col")
data_df.show()
```

As a result, we'll see the following change:

```
+-----+-----+-------------+----------+------+
|col_1|col_2|  string_col |   col_4  | col_6|
+-----+-----+-------------+----------+------+
|  100|200.0|string_test_1|2023-01-01|   A  |
|  200|300.0|string_test_2|2023-02-01|   A  |
|  300|400.0|string_test_3|2023-03-01|   A  |
+-----+-----+-------------+----------+------+
```

Notice that `col_3` is now called `string_col` after making the change.

Now, let's shift our focus to some data manipulation techniques in Spark DataFrames. You can have search-like functionality in Spark DataFrames for finding different values in a column. Now, let's take a look at how we can find unique values in a column of a Spark DataFrame.

Finding unique values in a column

Finding unique values is a very useful function that would give us distinct values in a column. For this purpose, we can use the `distinct()` function of the Spark DataFrame like so:

```
data_df.select("col_6").distinct().show()
```

Here is the result:

```
+------+
|col_6 |
+------+
|   A  |
+------+
```

We applied a `distinct` function on `col_6` to get all the unique values in this column. In our case, the column just had one distinct value, `A`, so that was shown.

We can also use it to find the count of distinct values in a given column. Here's an example of how to use this function:

```
data_df.select(F.countDistinct("col_6").alias("Total_Unique")).show()
```

Here is the result:

```
+------+
|col_6 |
+------+
|  1   |
+------+
```

In this example, we can see the total count of distinct values in `col_6`. Currently, it is the only type of distinct value present in this column, therefore, it returned `1`.

One other useful function in Spark data manipulation is changing the case of a column. Now, let's take a look at how we can change the case of a column in a Spark DataFrame.

Changing the case of a column

There is also a function that exists in Spark to change the case of a column. We don't need to specify each value of the column separately to make use of the function. Once applied, the whole column's values would change case. One such example is as follows:

```
from pyspark.sql.functions import upper
data_df.withColumn('upper_string_col', upper(data_df.string_col)).
show()
```

Here is the result:

```
+-----+-----+-------------+----------+-----+---------------+
|col_1|col_2|   string_col|     col_4|col_6|upper_string_col|
+-----+-----+-------------+----------+-----+---------------+
|  100|200.0|string_test_1|2023-01-01|    A|   STRING_TEST_1|
|  200|300.0|string_test_2|2023-02-01|    A|   STRING_TEST_2|
|  300|400.0|string_test_3|2023-03-01|    A|   STRING_TEST_3|
+-----+-----+-------------+----------+-----+---------------+
```

In this example, we change the case of `string_col` to all caps. We need to assign this to a new column, so, we create a column called `upper_string_col` to store these upper-case values. Also, note that this column is not added to the original `data_df` because we did not save the results back in `data_df`.

A lot of times in data manipulation, we would need functions to filter DataFrames. We will now take a look at how we can filter data in a Spark DataFrame.

Filtering a DataFrame

Filtering a DataFrame means that we can get a subset of rows or columns from a DataFrame. There are different ways of filtering a Spark DataFrame. We will take a look at one example here:

```
data_df.filter(data_df.col_1 == 100).show()
```

Here is the result:

```
+-----+-----+-------------+----------+------+
|col_1|col_2| string_col  |   col_4  |col_6 |
+-----+-----+-------------+----------+------+
|  100|200.0|string_test_1|2023-01-01|   A  |
+-----+-----+-------------+----------+------+
```

In this example, we filter `data_df` to only include rows where the column value of `col_1` is equal to `100`. This criterion is met by only one row, therefore, a single row is returned in the resulting DataFrame.

You can use this function to slice and dice your data in a number of different ways based on the requirements.

Since we are talking about data filtering, we can also filter data based on logical operators as well. We will now take a look at how we can use logical operators in DataFrames to filter data.

Logical operators in a DataFrame

Another important set of operators that we can combine with filter operations in a DataFrame are the logical operators. These consist of AND and OR operators, amongst others. These are used to filter DataFrames based on complex conditions. Let's take a look at how we use the AND operator here:

```
data_df.filter((data_df.col_1 == 100)
                  & (data_df.col_6 == 'A')).show()
```

Here is the result:

```
+-----+------+-------------+----------+------+
|col_1| col_2|  string_col |   col_4  |col_6 |
+-----+------+-------------+----------+------+
|  100| 200.0|string_test_1|2023-01-01|   A  |
+-----+------+-------------+----------+------+
```

In this example, we're trying to get the rows where the value of `col_1` is equal to `100` and the value of `col_6` is `A`. Currently, only one row fulfills this condition, therefore, one row is returned as a result.

Now, let's see how we can use the OR operator to combine conditions:

```
data_df.filter((data_df.col_1 == 100)
                  | (data_df.col_2 == 300.00)).show()
```

This statement will give the following result:

```
+-----+------+-------------+----------+------+
|col_1| col_2|  string_col |   col_4  | col_6|
+-----+------+-------------+----------+------+
|  100| 200.0|string_test_1|2023-01-01|   A  |
|  200| 300.0|string_test_2|2023-02-01|   A  |
+-----+------+-------------+----------+------+
```

In this example, we're trying to get the rows where the value of `col_1` is equal to `100` or the value of `col_2` is equal to `300.0`. Currently, two rows fulfill this condition, therefore, they are returned as a result.

In data filtering, there is another important function to find values in a list. Now, we will see how you use the `isin()` function in PySpark.

Using isin()

The `isin()` function is used to find values in a DataFrame column that exist in a list. To do this, we would create a list with some values in it. Once we have the list, then we would use the `isin()` function to see whether some of the values that are in the list exist in the DataFrame. Here's an example to demonstrate this:

```
list = [100, 200]
data_df.filter(data_df.col_1.isin(list)).show()
```

Here is the result:

```
+-----+-----+-------------+----------+------+
|col_1|col_2|  string_col |   col_4  |col_6 |
+-----+-----+-------------+----------+------+
|  100|200.0|string_test_1|2023-01-01|   A  |
|  200|300.0|string_test_2|2023-02-01|   A  |
+-----+-----+-------------+----------+------+
```

In this example, we see that the values `100` and `200` are present in the `data_df` DataFrame in two of its rows, therefore, both rows are returned.

We can also convert data types for different columns in Spark DataFrames. Now, let's look at how we can convert different data types in Spark DataFrames.

Datatype conversions

In this section, we'll see different ways of converting data types in Spark DataFrame columns.

We will start by using the `cast` function in Spark. The following code illustrates this:

```
from pyspark.sql.functions import col
from pyspark.sql.types import
StringType,BooleanType,DateType,IntegerType


data_df_2 = data_df.withColumn("col_4",col("col_4").
cast(StringType())) \
    .withColumn("col_1",col("col_1").cast(IntegerType()))
data_df_2.printSchema()
data_df.show()
```

Here is the result:

```
root
 |-- col_1: integer (nullable = true)
 |-- col_2: double (nullable = true)
 |-- string_col: string (nullable = true)
 |-- col_4: string (nullable = true)
 |-- col_6: string (nullable = false)

+-----+-----+-------------+----------+-----+
|col_1|col_2|   string_col|     col_4|col_6|
+-----+-----+-------------+----------+-----+
|  100|200.0|string_test_1|2023-01-01|    A|
|  200|300.0|string_test_2|2023-02-01|    A|
|  300|400.0|string_test_3|2023-03-01|    A|
+-----+-----+-------------+----------+-----+
```

In the preceding code, we see that we're changing the data types of two columns, namely `col_4` and `col_1`. First, we change `col_4` to string type. This column was previously a date column. Then, we change `col_1` to integer type from `long`.

Here is the schema of `data_df` for reference:

```
root
 |-- col_1: long (nullable = true)
 |-- col_2: double (nullable = true)
 |-- string_col: string (nullable = true)
 |-- col_4: date (nullable = true)
 |-- col_5: timestamp (nullable = true)
 |-- col_6: string (nullable = false)
```

We see that `col_1` and `col_4` were different data types.

The next example of changing the data types of the columns is by using the `selectExpr()` function. The following code illustrates this:

```
data_df_3 = data_df_2.selectExpr("cast(col_4 as date) col_4",
    "cast(col_1 as long) col_1")
data_df_3.printSchema()
data_df_3.show(truncate=False)
```

Here is the result:

```
root
 |-- col_4: date (nullable = true)
 |-- col_1: long (nullable = true)

+----------+-----+
|  col_4   |col_1|
+----------+-----+
|2023-01-01|  100|
|2023-02-01|  200|
|2023-03-01|  300|
+----------+-----+
```

In the preceding code, we see that we're changing the data types of two columns, namely `col_4` and `col_1`. First, we change `col_4` back to the `date` type. Then, we change `col_1` to the `long` type.

The next example of changing the data types of the columns is by using SQL. The following code illustrates this:

```
data_df_3.createOrReplaceTempView("CastExample")
data_df_4 = spark.sql("SELECT DOUBLE(col_1), DATE(col_4) from
CastExample")
data_df_4.printSchema()
data_df_4.show(truncate=False)
```

Here is the result:

```
root
 |-- col_1: double (nullable = true)
 |-- col_4: date (nullable = true)
+-----+----------+
|col_1|  col_4   |
+-----+----------+
|100.0|2023-01-01|
|200.0|2023-02-01|
|300.0|2023-03-01|
+-----+----------+
```

In the preceding code, we see that we're changing the data types of two columns, namely `col_4` and `col_1`. First, we use `createOrReplaceTempView()` to create a table named `CastExample`. Then, we use this table to change `col_4` back to the `date` type. Then, we change `col_1` to the `double` type.

In the data analysis world, working with null values is very valuable. Now, let's take a look at how we can drop null values from a DataFrame.

Dropping null values from a DataFrame

Sometimes, in the data, there exist null values that can make clean data messy. Dropping nulls is an essential exercise that a lot of data analysts and data engineers need to do. Pyspark provides us with relevant functions to do this.

Let's create another DataFrame called `salary_data` to show some of the next operations:

```
salary_data = [("John", "Field-eng", 3500),
    ("Michael", "Field-eng", 4500),
    ("Robert", None, 4000),
    ("Maria", "Finance", 3500),
    ("John", "Sales", 3000),
    ("Kelly", "Finance", 3500),
    ("Kate", "Finance", 3000),
    ("Martin", None, 3500),
    ("Kiran", "Sales", 2200),
    ("Michael", "Field-eng", 4500)
  ]
columns= ["Employee", "Department", "Salary"]
salary_data = spark.createDataFrame(data = salary_data, schema =
columns)
salary_data.printSchema()
salary_data.show()
```

Here is the result:

```
root
 |-- Employee: string (nullable = true)
 |-- Department: string (nullable = true)
 |-- Salary: long (nullable = true)
```

```
+--------+----------+------+
|Employee|Department|Salary|
+--------+----------+------+
|    John| Field-eng| 3500 |
| Michael| Field-eng| 4500 |
|  Robert|      null| 4000 |
|   Maria|   Finance| 3500 |
|    John|     Sales| 3000 |
|   Kelly|   Finance| 3500 |
|    Kate|   Finance| 3000 |
|  Martin|      null| 3500 |
|   Kiran|     Sales| 2200 |
| Michael| Field-eng| 4500 |
+--------+----------+------+
```

Now, let's take a look at the `dropna()` function; this will help us drop null values from our DataFrame:

```
salary_data.dropna().show()
```

Here is the result:

```
+--------+----------+------+
|Employee|Department|Salary|
+--------+----------+------+
| John   | Field-eng| 3500 |
| Michael| Field-eng| 4500 |
| Maria  |   Finance| 3500 |
| John   |     Sales| 3000 |
| Kelly  |   Finance| 3500 |
| Kate   |   Finance| 3000 |
| Kiran  |     Sales| 2200 |
| Michael| Field-eng| 4500 |
+--------+----------+------+
```

We see in the resulting DataFrame that rows with `Robert` and `Martin` are deleted from the new DataFrame when we use the `dropna()` function.

Deduplicating data is another useful technique that is often required in data analysis tasks. Now, let's take a look at how we can drop duplicate values from a DataFrame.

Dropping duplicates from a DataFrame

Sometimes, in the data, there are redundant values present that would make clean data messy. Dropping these values might be needed in a lot of use cases. PySpark provides us with the `dropDuplicates()` function to do this. Here's the code to illustrate this:

```
new_salary_data = salary_data.dropDuplicates().show()
```

Here is the result:

```
+--------+----------+------+
|Employee|Department|Salary|
+--------+----------+------+
|    John| Field-eng|  3500|
| Michael| Field-eng|  4500|
|   Maria|   Finance|  3500|
|    John|     Sales|  3000|
|   Kelly|   Finance|  3500|
|    Kate|   Finance|  3000|
|   Kiran|     Sales|  2200|
+--------+----------+------+
```

We see in this example that employees named Michael are only shown once in the resulting DataFrame after we apply the `dropDuplicates()` function to the original DataFrame. This name and its corresponding values exist in the original DataFrame twice.

Now that we have learned about different data filtering techniques, we will now see how we can aggregate data in Pyspark DataFrames.

Using aggregates in a DataFrame

Some of the methods available in Spark for aggregating data are as follows:

- `agg`
- `avg`
- `count`
- `max`
- `mean`
- `min`
- `pivot`
- `sum`

We will see some of them in action in the following code examples.

Average (avg)

In the following example, we see how to use aggregate functions in Spark. We will start by calculating the average of all the values in a column:

```
from pyspark.sql.functions import countDistinct, avg
salary_data.select(avg('Salary')).show()
```

Here is the result:

```
+-----------+
|avg(Salary)|
+-----------+
|     3520.0|
+-----------+
```

This example calculates the average of the salary column of the `salary_data` DataFrame. We have passed the `Salary` column to the `avg` function and it has calculated the average of that column for us.

Now, let's take a look at how to count different elements in a PySpark DataFrame.

Count

In the following code example, we can see how you use aggregate functions in Spark:

```
salary_data.agg({'Salary':'count'}).show()
```

Here is the result:

```
+-------------+
|count(Salary)|
+-------------+
|           10|
+-------------+
```

This example calculates the total count of the values in the `Salary` column of the `salary_data` DataFrame. We have passed the `Salary` column to the `agg` function with `count` as its other parameter, and it has calculated the count of that column for us.

Now, let's take a look at how to count distinct elements in a PySpark DataFrame.

Count distinct values

In the following example, we will look at how to count distinct elements in a PySpark DataFrame:

```
salary_data.select(countDistinct("Salary").alias("Distinct Salary")).
show()
```

Here is the result:

```
+---------------+
|Distinct Salary|
+---------------+
|              5|
+---------------+
```

This example calculates total distinct values in the salary column of the `salary_data` DataFrame. We have passed the `Salary` column to the `countDistinct` function and it has calculated the count of that column for us.

Now, let's take a look at how to find maximum values in a PySpark DataFrame.

Finding maximums (max)

In the following code example, we will take a look at how to find maximum values in a column of a PySpark DataFrame:

```
salary_data.agg({'Salary':'max'}).show()
```

Here is the result:

```
+-----------+
|max(Salary)|
+-----------+
|       4500|
+-----------+
```

This example calculates the maximum value out of all the values in the `Salary` column of the `salary_data` DataFrame. We have passed the `Salary` column to the `agg` function with `max` as its other parameter, and it has calculated the maximum of that column for us.

Now, let's take a look at how to get the sum of all elements in a PySpark DataFrame.

Sum

In the following code example, we will look at how to sum all values in a PySpark DataFrames:

```
salary_data.agg({'Salary':'sum'}).show()
```

Here is the result:

```
+-----------+
|sum(Salary)|
+-----------+
|      35200|
+-----------+
```

This example calculates the sum of all the values in the `Salary` column of the `salary_data` DataFrame. We have passed the `Salary` column to the `agg` function with `sum` as its other parameter, and it has calculated the sum of that column for us.

Now, let's take a look at how to sort data in a PySpark DataFrame.

Sort data with OrderBy

In the following code example, we will look at how can we sort data in ascending order in a PySpark DataFrame:

```
salary_data.orderBy("Salary").show()
```

Here is the result:

```
+--------+----------+------+
|Employee|Department|Salary|
+--------+----------+------+
| Kiran  | Sales    | 2200 |
| Kate   | Finance  | 3000 |
| John   | Sales    | 3000 |
| John   | Field-eng| 3500 |
| Martin | null     | 3500 |
| Kelly  | Finance  | 3500 |
| Maria  | Finance  | 3500 |
| Robert | null     | 4000 |
| Michael| Field-eng| 4500 |
| Michael| Field-eng| 4500 |
+--------+----------+------+
```

This example sorts the full DataFrame based on the values in the `Salary` column of the `salary_data` DataFrame. We have passed the `Salary` column to the `orderBy` function and it has sorted the DataFrame based on this column.

We can also sort the data in descending format by adding another function, `desc()`, to the original `orderBy` function. The following example illustrates this:

```
salary_data.orderBy(salary_data["Salary"].desc()).show()
```

Here is the result:

```
+--------+----------+------+
|Employee|Department|Salary|
+--------+----------+------+
| Michael| Field-eng| 4500 |
| Michael| Field-eng| 4500 |
| Robert | null     | 4000 |
| Martin | null     | 3500 |
| Kelly  | Finance  | 3500 |
| Maria  | Finance  | 3500 |
| John   | Field-eng| 3500 |
| John   | Sales    | 3000 |
| Kate   | Finance  | 3000 |
| Kiran  | Sales    | 2200 |
+--------+----------+------+
```

This example is sorting the full DataFrame in descending order based on the values in the `Salary` column of the `salary_data` DataFrame. We have passed the `Salary` column to the `orderBy` function with `desc()` as an additional function call and it has sorted the DataFrame in descending order based on this column.

Summary

Over the course of this chapter, we have learned how to manipulate data in Spark DataFrames.

We talked about the Spark DataFrame API and what different data types are in Spark. We also learned how to create DataFrames in Spark and how we can view these DataFrames once they've been created. Finally, we learned about different data manipulation and data aggregation functions.

In the next chapter, we will cover some advanced operations in Spark with respect to data manipulation.

Sample question

1. Which of the following operations will trigger evaluation?

    A. `DataFrame.filter()`
    B. `DataFrame.distinct()`
    C. `DataFrame.intersect()`
    D. `DataFrame.join()`
    E. `DataFrame.count()`

Answer

1. E

5

Advanced Operations and Optimizations in Spark

In this chapter, we will delve into the advanced capabilities of Apache Spark, equipping you with the knowledge and techniques necessary to optimize your data processing workflows. From the inner workings of the Catalyst optimizer to the intricacies of different types of joins, we will explore advanced Spark operations that empower you to harness the full potential of this powerful framework.

The chapter will cover the following topics:

- Different options to group data in Spark DataFrames.
- Various types of joins in Spark, including inner join, left join, right join, outer join, cross join, broadcast join, and shuffle join, each with its unique use cases and implications
- Shuffle and broadcast joins, with a focus on broadcast hash joins and shuffle sort-merge joins, along with their applications and optimization strategies
- Reading and writing data to disk in Spark using different data formats, such as CSV, Parquet, and others.
- Using Spark SQL for different operations
- The Catalyst optimizer, a pivotal component in Spark's query execution engine that employs rule-based and cost-based optimizations to enhance query performance
- The distinction between narrow and wide transformations in Spark and when to use each type to achieve optimal parallelism and resource efficiency
- Data persistence and caching techniques to reduce recomputation and expedite data processing, with best practices for efficient memory management
- Data partitioning through repartition and coalesce, and how to use these operations to balance workloads and optimize data distribution

- **User-defined functions** (**UDFs**) and custom functions, which allow you to implement specialized data processing logic, as well as when and how to leverage them effectively
- Performing advanced optimizations in Spark using the Catalyst optimizer and **Adaptive Query Execution** (**AQE**)
- Data-based optimization techniques and their benefits

Each section will provide in-depth insights, practical examples, and best practices, ensuring you are well-equipped to handle complex data processing challenges in Apache Spark. By the end of this chapter, you will possess the knowledge and skills needed to harness the advanced capabilities of Spark and unlock its full potential for your data-driven endeavors.

Grouping data in Spark and different Spark joins

We will start with one of the most important data manipulation techniques: grouping and joining data. When we are doing data exploration, grouping data based on different criteria becomes essential to data analysis. We will look at how we can group different data using `groupBy`.

Using groupBy in a DataFrame

We can group data in a DataFrame based on different criteria – for example, we can group data based on different columns in a DataFrame. We can also apply different aggregations, such as `sum` or `average`, to this grouped data to get a holistic view of data slices.

For this purpose, in Spark, we have the `groupBy` operation. The `groupBy` operation is similar to `groupBy` in SQL in that we can do group-wise operations on these grouped datasets. Moreover, we can specify multiple `groupBy` criteria in a single `groupBy` statement. The following example shows how to use `groupBy` in PySpark. We will use the DataFrame salary data we created in the previous chapter.

In the following `groupBy` statement, we are grouping the salary data based on the `Department` column:

```
salary_data.groupby('Department')
```

As a result, this operation returns a grouped data object that has been grouped by the `Department` column:

```
<pyspark.sql.group.GroupedData at 0x7fc8495a3c10>
```

This can be assigned to a separate DataFrame and more operations can be done on this data. All the aggregate operations can also be used for different groups of a DataFrame.

We will use the following statement to get the average salary across different departments in our `salary_data` DataFrame:

```
salary_data.groupby('Department').avg().show()
```

Here's the result:

```
+----------+------------------+
|Department| avg(Salary)      |
+----------+------------------+
| null     |            3750.0|
| Sales    |            2600.0|
| Field-eng| 4166.666666666667|
| Finance  |3333.3333333333335|
+----------+------------------+
```

In this example, we can see that each department's average salary is calculated based on the `salary` column of the `salary_data` DataFrame. All four departments, including `null` (since we had null values in our DataFrame), are included in the resulting DataFrame.

Now, let's take a look at how we can apply complex `groupBy` operations to data in PySpark DataFrames.

A complex groupBy statement

`groupBy` can be used in complex data operations, such as multiple aggregations within a single `groupBy` statement.

In the following code snippet, we are going to use `groupBy` by taking a sum of the salary column for each department. Then, we will round off the `sum(Salary)` column that we just created to two digits after a decimal. After, we will rename the `sum(Salary)` column back to `Salary`. All of these operations are being done in a single `groupBy` statement:

```
from pyspark.sql.functions import col, round
salary_data.groupBy('Department')\
  .sum('Salary')\
  .withColumn('sum(Salary)',round(col('sum(Salary)'), 2))\
  .withColumnRenamed('sum(Salary)', 'Salary')\
  .orderBy('Department')\
  .show()
```

As a result, we will see the following DataFrame showing the aggregated sum of the `Salary` column based on each department:

```
+----------+------------------+
|Department|    sum(Salary)   |
+----------+------------------+
| null     |              7500|
| Field-eng|             12500|
| Finance  |             10000|
| Sales    |              5200|
+----------+------------------+
```

In this example, we can see that each department's total salary is calculated in a new column named `sum(Salary)`, after which we round this total up to two decimal places. In the next statement, we rename the `sum(Salary)` column back to `Salary` and then sort this resulting DataFrame based on `Department`. In the resulting DataFrame, we can see that each department's sum of salaries is calculated in the new column.

Now that we know how to group data using different aggregations, let's take a look at how we can join two DataFrames together in Spark.

Joining DataFrames in Spark

Join operations are fundamental in data processing tasks and are a core component of Apache Spark. Spark provides several types of joins to combine data from different DataFrames or datasets. In this section, we will explore different Spark join operations and when to use each type.

Join operations are used to combine data from two or more DataFrames based on a common column. These operations are essential for tasks such as merging datasets, aggregating information, and performing relational operations.

In Spark, the primary syntax for performing joins is using the `.join()` method, which takes the following parameters:

- `other`: The other DataFrame to join with
- `on`: The column(s) on which to join the DataFrames
- `how`: The type of join to perform (inner, outer, left, or right)
- `suffixes`: Suffixes to add to columns with the same name in both DataFrames

These parameters are used in the main syntax of the join operation, as follows:

```
Dataframe1.join(Dataframe2, on, how)
```

Here, `Dataframe1` would be on the left-hand side of the join and `Dataframe2` would be on the right-hand side of the join.

DataFrames or datasets can be joined based on common columns within a DataFrame, and the result of a join query is a new DataFrame.

We will demonstrate the join operation on two new DataFrames. First, let's create these DataFrames. The first DataFrame is called `salary_data_with_id`:

```
salary_data_with_id = [(1, "John", "Field-eng", 3500), \
    (2, "Robert", "Sales", 4000), \
    (3, "Maria", "Finance", 3500), \
```

```
    (4, "Michael", "Sales", 3000), \
    (5, "Kelly", "Finance", 3500), \
    (6, "Kate", "Finance", 3000), \
    (7, "Martin", "Finance", 3500), \
    (8, "Kiran", "Sales", 2200), \
  ]
columns= ["ID", "Employee", "Department", "Salary"]
salary_data_with_id = spark.createDataFrame(data = salary_data_with_
id, schema = columns)
salary_data_with_id.show()
```

The resulting DataFrame, named `salary_data_with_id`, looks like this:

```
+---+--------+----------+------+
|ID |Employee|Department|Salary|
+---+--------+----------+------+
| 1 | John   | Field-eng|  3500|
| 2 | Robert | Sales    |  4000|
| 3 | Maria  | Finance  |  3500|
| 4 | Michael| Sales    |  3000|
| 5 | Kelly  | Finance  |  3500|
| 6 | Kate   | Finance  |  3000|
| 7 | Martin | Finance  |  3500|
| 8 | Kiran  | Sales    |  2200|
+---+--------+----------+------+
```

Now, we'll create another DataFrame named `employee_data`:

```
employee_data = [(1, "NY", "M"), \
    (2, "NC", "M"), \
    (3, "NY", "F"), \
    (4, "TX", "M"), \
    (5, "NY", "F"), \
    (6, "AZ", "F") \
  ]
columns= ["ID", "State", "Gender"]
employee_data = spark.createDataFrame(data = employee_data, schema =
columns)
employee_data.show()
```

The resulting DataFrame, named `employee_data`, looks like this:

```
+---+-----+------+
| ID|State|Gender|
+---+-----+------+
|  1|   NY|     M|
|  2|   NC|     M|
|  3|   NY|     F|
|  4|   TX|     M|
|  5|   NY|     F|
|  6|   AZ|     F|
+---+-----+------+
```

Now, let's suppose we want to join these two DataFrames together based on the `ID` column.

As we mentioned earlier, Spark offers different types of join operations. We will explore some of them in this chapter. Let's start with inner joins.

Inner joins

An **inner join** is used when we want to join two DataFrames based on values that are common in both DataFrames. Any value that doesn't exist in any one of the DataFrames would not be part of the resulting DataFrame. By default, the join type is an inner join in Spark.

Use case

Inner joins are useful for merging data when you are interested in common elements in both DataFrames – for example, joining sales data with customer data to see which customers made a purchase.

The following code illustrates how we can use an inner join with the DataFrames we created earlier:

```
salary_data_with_id.join(employee_data,salary_data_with_id.ID
==  employee_data.ID,"inner").show()
```

The resulting DataFrame now contains all the columns of both DataFrames – `salary_data_with_id` and `employee_data` – joined together in a single DataFrame. It only includes rows that are common in both DataFrames. Here's what it looks like:

```
+---+--------+----------+------+---+-----+------+
| ID|Employee|Department|Salary| ID|State|Gender|
+---+--------+----------+------+---+-----+------+
|  1|    John| Field-eng|  3500|  1|   NY|     M|
|  2|  Robert|     Sales|  4000|  2|   NC|     M|
|  3|   Maria|   Finance|  3500|  3|   NY|     F|
|  4| Michael|     Sales|  3000|  4|   TX|     M|
|  5|   Kelly|   Finance|  3500|  5|   NY|     F|
|  6|    Kate|   Finance|  3000|  6|   AZ|     F|
+---+--------+----------+------+---+-----+------+
```

You will notice that the `how` parameter defines the type of join that is being done in this statement. Currently, it says `inner` because we wanted the DataFrames to join based on an inner join. We can also see that IDs `7` and `8` are missing. The reason is that the `employee_data` DataFrame did not contain IDs `7` and `8`. Since we're using an inner join, it only joins data based on common data elements in both DataFrames. Any data that is not present in either one of the DataFrames will not be part of the resulting DataFrame.

Next, we will explore outer joins.

Outer joins

An **outer join**, also known as a **full outer join**, returns all the rows from both DataFrames, filling in missing values with `null`.

We should use an outer join when we want to join two DataFrames based on values that exist in both DataFrames, regardless of whether they don't exist in the other DataFrame. Any values that exist in any one of the DataFrames would be part of the resulting DataFrame.

Use case

Outer joins are suitable for situations where you want to include all records from both DataFrames while accommodating unmatched values – for example, when merging employee data with project data to see which employees are assigned to which projects, including those not currently assigned to any.

The following code illustrates how we use an outer join with the DataFrames we created earlier:

```
salary_data_with_id.join(employee_data,salary_data_with_id.ID
==  employee_data.ID,"outer").show()
```

The resulting DataFrame contains data for all the employees in the `salary_data_with_id` and `employee_data` DataFrames. Here's what it looks like:

```
+---+--------+----------+------+----+-----+------+
| ID|Employee|Department|Salary|  ID|State|Gender|
+---+--------+----------+------+----+-----+------+
|  1|    John| Field-eng|  3500|   1|   NY|     M|
|  2|  Robert|     Sales|  4000|   2|   NC|     M|
|  3|   Maria|   Finance|  3500|   3|   NY|     F|
|  4| Michael|     Sales|  3000|   4|   TX|     M|
|  5|   Kelly|   Finance|  3500|   5|   NY|     F|
|  6|    Kate|   Finance|  3000|   6|   AZ|     F|
|  7|  Martin|   Finance|  3500|null| null|  null|
|  8|   Kiran|     Sales|  2200|null| null|  null|
+---+--------+----------+------+----+-----+------+
```

You will notice that the `how` parameter has changed and says `outer`. In the resulting DataFrame, IDs `7` and `8` are now present. However, also notice that the `ID`, `State`, and `Gender` columns for IDs `7`

and 8 are null. The reason is that the employee_data DataFrame did not contain IDs 7 and 8. Any data not present in either of the DataFrames would be part of the resulting DataFrame, but the corresponding columns would be null for the DataFrame that this was not present, as shown in the case of employee IDs 7 and 8.

Next, we will explore left joins.

Left joins

A left join returns all the rows from the left DataFrame and the matched rows from the right DataFrame. If there is no match in the right DataFrame, the result will contain null values.

Use case

Left joins are handy when you want to keep all records from the left DataFrame and only the matching records from the right DataFrame – for instance, when merging customer data with transaction data to see which customers have made a purchase.

The following code illustrates how we can use a left join with the DataFrames we created earlier:

```
salary_data_with_id.join(employee_data,salary_data_with_id.ID
==  employee_data.ID,"left").show()
```

The resulting DataFrame contains all the data from the left DataFrame – that is, salary_data_with_id. It looks like this:

```
+---+--------+----------+------+----+-----+------+
| ID|Employee|Department|Salary|  ID|State|Gender|
+---+--------+----------+------+----+-----+------+
|  1|    John| Field-eng|  3500|   1|   NY|     M|
|  2|  Robert|     Sales|  4000|   2|   NC|     M|
|  3|   Maria|   Finance|  3500|   3|   NY|     F|
|  4| Michael|     Sales|  3000|   4|   TX|     M|
|  5|   Kelly|   Finance|  3500|   5|   NY|     F|
|  6|    Kate|   Finance|  3000|   6|   AZ|     F|
|  7|  Martin|   Finance|  3500|null| null|  null|
|  8|   Kiran|     Sales|  2200|null| null|  null|
+---+--------+----------+------+----+-----+------+
```

Note that the how parameter has changed and says left. Now, IDs 7 and 8 are present. However, also notice that the ID, State, and Gender columns for IDs 7 and 8 are null. The reason is that the employee_data DataFrame did not contain IDs 7 and 8. Since salary_data_with_id is the left DataFrame in the join statement, its values take priority in the join.

All records from the left DataFrame are present in the resulting DataFrame, and matching records from the right DataFrame are included. Non-matching entries in the right DataFrame are filled with null values in the result.

Next, we will explore right joins.

Right joins

A right join is similar to a left join, but it returns all the rows from the right DataFrame and the matched rows from the left DataFrame. Non-matching rows from the left DataFrame contain null values.

Use case

Right joins are the opposite of left joins and are used when you want to keep all records from the right DataFrame while including matching records from the left DataFrame.

The following code illustrates how to use a right join with the DataFrames we created earlier:

```
salary_data_with_id.join(employee_data,salary_data_with_id.ID
==  employee_data.ID,"right").show()
```

The resulting DataFrame contains all the data from the right-hand DataFrame – that is, `employee_data`. It looks like this:

```
+---+--------+----------+------+---+-----+------+
| ID|Employee|Department|Salary| ID|State|Gender|
+---+--------+----------+------+---+-----+------+
|  1|    John| Field-eng|  3500|  1|   NY|     M|
|  2|  Robert|     Sales|  4000|  2|   NC|     M|
|  3|   Maria|   Finance|  3500|  3|   NY|     F|
|  4| Michael|     Sales|  3000|  4|   TX|     M|
|  5|   Kelly|   Finance|  3500|  5|   NY|     F|
|  6|    Kate|   Finance|  3000|  6|   AZ|     F|
+---+--------+----------+------+---+-----+------+
```

Notice the `how` parameter has changed and now says `right`. The resulting DataFrame shows that IDs `7` and `8` are not present. The reason is that the `employee_data` DataFrame does not contain IDs `7` and `8`. Since `employee_data` is the right DataFrame in the join statement, its values take priority in the join.

All records from the right DataFrame are present in the resulting DataFrame, and matching records from the left DataFrame are included. Non-matching entries in the left DataFrame are filled with `null` values in the result.

Next, we will explore cross joins.

Cross joins

A **cross join**, also known as a **Cartesian join**, combines each row from the left DataFrame with every row from the right DataFrame. This results in a large, Cartesian product DataFrame.

Use case

Cross joins should be used with caution due to their potential for generating massive datasets. They are typically used when you want to explore all possible combinations of data, such as when generating test data.

Next, we will explore the union option to join two DataFrames.

Union

Union is used to join two DataFrames that have a similar schema. To illustrate this, we will create another DataFrame called `salary_data_with_id_2` that contains some more values. The schema of this DataFrame is the same as the one for `salary_data_with_id`:

```
salary_data_with_id_2 = [(1, "John", "Field-eng", 3500), \
    (2, "Robert", "Sales", 4000), \
    (3, "Aliya", "Finance", 3500), \
    (4, "Nate", "Sales", 3000), \
  ]
columns2= ["ID", "Employee", "Department", "Salary"]

salary_data_with_id_2 = spark.createDataFrame(data = salary_data_with_
id_2, schema = columns2)

salary_data_with_id_2.printSchema()
salary_data_with_id_2.show(truncate=False)
```

As a result, you will see the schema of the DataFrame first, after which you will see the actual DataFrame and its values:

```
root
 |-- ID: long (nullable = true)
 |-- Employee: string (nullable = true)
 |-- Department: string (nullable = true)
 |-- Salary: long (nullable = true)

+---+--------+----------+------+
|ID |Employee|Department|Salary|
+---+--------+----------+------+
|1  |John    |Field-eng |3500  |
|2  |Robert  |Sales     |4000  |
|3  |Aliya   |Finance   |3500  |
|4  |Nate    |Sales     |3000  |
+---+--------+----------+------+
```

Once we have this DataFrame, we can use the `union()` function to join the `salary_data_with_id` and `salary_data_with_id_2` DataFrames together. The following example illustrates this:

```
unionDF = salary_data_with_id.union(salary_data_with_id_2)
unionDF.show(truncate=False)
```

The resulting DataFrame, named `unionDF`, looks like this:

```
+---+--------+----------+------+
|ID |Employee|Department|Salary|
+---+--------+----------+------+
|1  |John    |Field-eng |3500  |
|2  |Robert  |Sales     |4000  |
|3  |Maria   |Finance   |3500  |
|4  |Michael |Sales     |3000  |
|5  |Kelly   |Finance   |3500  |
|6  |Kate    |Finance   |3000  |
|7  |Martin  |Finance   |3500  |
|8  |Kiran   |Sales     |2200  |
|1  |John    |Field-eng |3500  |
|2  |Robert  |Sales     |4000  |
|3  |Aliya   |Finance   |3500  |
|4  |Nate    |Sales     |3000  |
+---+--------+----------+------+
```

As you can see, both DataFrames are joined together and as a result, new rows are added to the resulting DataFrame. The last four rows are from `salary_data_with_id_2` and were added to the rows of `salary_data_with_id`. This is another way to join two DataFrames together.

In this section, we explored different types of Spark joins and their appropriate use cases. Choosing the right join type is crucial to ensure efficient data processing in Spark, and understanding the implications of each type will help you make informed decisions in your data analysis and processing tasks.

Now, let's look at how we can read and write data in Spark.

Reading and writing data

When we work with Spark and do all the operations in Spark for data manipulation, one of the most important things that we need to do is read and write data to disk. Remember, Spark is an in-memory framework, which means that all the operations take place in the memory of the compute or cluster. Once these operations are completed, we'll want to write that data to disk. Similarly, before we manipulate any data, we'll likely need to read data from disk as well.

There are several data formats that Spark supports for reading and writing different types of data files. We will discuss the following formats in this chapter.

- **Comma Separated Values** (**CSV**)
- **Parquet**
- **Optimized Row Columnar** (**ORC**)

Please note that these are not the only formats that Spark supports but this is a very popular subset of formats. A lot of other formats are also supported by Spark, such as Avro, text, JDBC, Delta, and others.

In the next section, we will discuss the CSV file format and how to read and write CSV format data files.

Reading and writing CSV files

In this section, we will discuss how to read and write data from the CSV file format. In this file format, data is separated by commas. This is a very popular data format because of its ease of use and simplicity.

Let's look at how to write CSV files with Spark by running the following code:

```
salary_data_with_id.write.csv('salary_data.csv', mode='overwrite',
header=True)
spark.read.csv('/salary_data.csv', header=True).show()
```

The resulting DataFrame, named `salary_data_with_id`, looks like this:

```
+---+--------+----------+------+
|ID |Employee|Department|Salary|
+---+--------+----------+------+
| 1 | John   | Field-eng|  3500|
| 2 | Robert | Sales    |  4000|
| 3 | Maria  | Finance  |  3500|
| 4 | Michael| Sales    |  3000|
| 5 | Kelly  | Finance  |  3500|
| 6 | Kate   | Finance  |  3000|
| 7 | Martin | Finance  |  3500|
| 8 | Kiran  | Sales    |  2200|
+---+--------+----------+------+
```

There are certain parameters in the `dataframe.write.csv()` function that we can see here. The first parameter is the dataframe name that we need to write to disk. The second parameter, `header`, specifies whether the file that we need to write should be written with a header row or not.

There are certain parameters in the `dataframe.read.csv()` function that we should discuss. The first parameter is the `path/name` value of the file that we need to read. The second parameter, `header`, specifies whether the file has a header row to be read.

In the first statement, we're writing the `salary_data` DataFrame to a CSV file named `salary_data.csv`.

In the next statement, we're reading back the same file that we wrote to see its contents. We can see that the resulting file contains the same data that we wrote.

Let's look at another function that can be used to read CSV files with Spark:

```
from pyspark.sql.types import *
filePath = '/salary_data.csv'
columns= ["ID", "State", "Gender"]
schema = StructType([
      StructField("ID", IntegerType(),True),
  StructField("State",  StringType(),True),
  StructField("Gender",  StringType(),True)
])
read_data = spark.read.format("csv").option("header","true").
schema(schema).load(filePath)
read_data.show()
```

The resulting DataFrame, named `read_data`, looks like this:

```
+---+--------+----------+------+
|ID |Employee|Department|Salary|
+---+--------+----------+------+
| 1 | John   | Field-eng|  3500|
| 2 | Robert | Sales    |  4000|
| 3 | Maria  | Finance  |  3500|
| 4 | Michael| Sales    |  3000|
| 5 | Kelly  | Finance  |  3500|
| 6 | Kate   | Finance  |  3000|
| 7 | Martin | Finance  |  3500|
| 8 | Kiran  | Sales    |  2200|
+---+--------+----------+------+
```

There are certain parameters in the `spark.read.format()` function. First, we specify the format of the file that needs to be read. Then, we can perform different function calls for different options. In the next call, we specify that the file has a header, so the DataFrame expects to have a header. Then, we specify that we need to have a schema for this data, which is defined in the `schema` variable. Finally, in the `load` function, we define the path of the file to be loaded.

Next, we will learn how to read and write Parquet files with Spark.

Reading and writing Parquet files

In this section, we will discuss the Parquet file format. Parquet is a columnar file format that makes data reading and writing very efficient. It is also a compact file format that facilitates faster reads and writes.

Let's learn how to write Parquet files with Spark by running the following code:

```
salary_data_with_id.write.parquet('salary_data.parquet',
mode='overwrite')
spark.read.parquet(' /salary_data.parquet').show()
```

The resulting DataFrame, named `salary_data_with_id`, looks like this:

```
+---+--------+----------+------+
|ID |Employee|Department|Salary|
+---+--------+----------+------+
| 1 | John   | Field-eng|  3500|
| 2 | Robert | Sales    |  4000|
| 3 | Maria  | Finance  |  3500|
| 4 | Michael| Sales    |  3000|
| 5 | Kelly  | Finance  |  3500|
| 6 | Kate   | Finance  |  3000|
| 7 | Martin | Finance  |  3500|
| 8 | Kiran  | Sales    |  2200|
+---+--------+----------+------+
```

There are certain parameters in the `dataframe.write()` function that we can see here. The first call is to the `parquet` function to define the file type. Then, as the next parameter, we specify the path where this Parquet file needs to be written.

In the next statement, we're reading the same file that we wrote, to see its contents. We can see that the resulting file contains the data we wrote.

Next, we will look at how we can read and write ORC files with Spark.

Reading and writing ORC files

In this section, we will discuss the ORC file format. Like Parquet, ORC is also a columnar and compact file format that makes data reading and writing very efficient.

Let's learn how to write ORC files with Spark by running the following code:

```
salary_data_with_id.write.orc('salary_data.orc', mode='overwrite')
spark.read.orc(' /salary_data.orc').show()
```

The resulting DataFrame, named `salary_data_with_id`, looks like this:

```
+---+--------+----------+------+
|ID |Employee|Department|Salary|
+---+--------+----------+------+
| 1 | John   | Field-eng|  3500|
| 2 | Robert | Sales    |  4000|
| 3 | Maria  | Finance  |  3500|
| 4 | Michael| Sales    |  3000|
| 5 | Kelly  | Finance  |  3500|
| 6 | Kate   | Finance  |  3000|
| 7 | Martin | Finance  |  3500|
| 8 | Kiran  | Sales    |  2200|
+---+--------+----------+------+
```

There are certain parameters in the `dataframe.write()` function that we can see. The first call is to the `orc` function to define the file type. Then, as the next parameter, we specify the path where this Parquet file needs to be written.

In the next statement, we're reading back the same file that we wrote to see its contents. We can see that the resulting file contains the same data that we wrote.

Next, we will look at how we can read and write Delta files with Spark.

Reading and writing Delta files

The Delta file format is an open format that is more optimized than Parquet and other columnar formats. When the data is stored in Delta format, you will notice that the underlying files are in Parquet. The Delta format adds a transactional log on top of Parquet files to make data reads and writes a lot more efficient.

Let's learn how to read and write Delta files with Spark by running the following code:

```
salary_data_with_id.write.format("delta").save("/FileStore/tables/
salary_data_with_id", mode='overwrite')
df = spark.read.load("/FileStore/tables/salary_data_with_id")
df.show()
```

The resulting DataFrame, named `salary_data_with_id`, looks like this:

```
+---+--------+----------+------+
| ID|Employee|Department|Salary|
+---+--------+----------+------+
|  7|  Martin|   Finance|  3500|
|  4| Michael|     Sales|  3000|
|  6|    Kate|   Finance|  3000|
|  2|  Robert|     Sales|  4000|
|  1|    John| Field-eng|  3500|
|  5|   Kelly|   Finance|  3500|
|  3|   Maria|   Finance|  3500|
|  8|   Kiran|     Sales|  2200|
+---+--------+----------+------+
```

In this example, we're writing `salary_data_with_id` to a Delta file. We added the `delta` parameter to the `format` function, after which we saved the file to a location.

In the next statement, we are reading the same Delta file we wrote into a DataFrame called `df`. The contents of the file remain the same as the DataFrame we used to write it with.

Now that we know how to manipulate and join data with advanced operations in Spark, we will look at how we can use SQL with Spark DataFrames interchangeably to switch between Python and SQL as languages. This gives a lot of power to Spark users since this allows them to use multiple languages, depending on the use case and their knowledge of different languages.

Using SQL in Spark

In *Chapter 2*, we talked about Spark Core and how it's shared across different components of Spark. DataFrames and Spark SQL can also be used interchangeably. We can also use data stored in DataFrames with Spark SQL queries.

The following code illustrates how we can make use of this feature:

```
salary_data_with_id.createOrReplaceTempView("SalaryTable")
spark.sql("SELECT count(*) from SalaryTable").show()
```

The resulting DataFrame looks like this:

```
+--------+
|count(1)|
+--------+
|       8|
+--------+
```

The `createOrReplaceTempView` function is used to convert a DataFrame into a table named `SalaryTable`. Once this conversion is made, we can run regular SQL queries on top of this table. We are running a `count *` query to count the total number of elements in a table.

In the next section, we will see what a UDF is and how we use that in Spark.

UDFs in Apache Spark

UDFs are a powerful feature in Apache Spark that allows you to extend the functionality of Spark by defining custom functions. UDFs are essential for transforming and manipulating data in ways not directly supported by built-in Spark functions. In this section, we'll delve into the concepts, implementation, and best practices for using UDFs in Spark.

What are UDFs?

UDFs are custom functions that are created by users to perform specific operations on data within Spark. UDFs extend the range of transformations and operations you can apply to your data, making Spark more versatile for diverse use cases.

Here are some of the key characteristics of UDFs:

- **User-customized logic**: UDFs allow you to apply user-specific logic or custom algorithms to your data
- **Support for various languages**: Spark supports UDFs written in various programming languages, including Scala, Python, Java, and R
- **Compatibility with DataFrames and resilient distributed datasets (RDDs)**: UDFs can be used with both DataFrames and RDDs
- **Leverage external libraries**: You can use external libraries within your UDFs to perform advanced operations

Let's see how UDFs are created.

Creating and registering UDFs

To use UDFs in Spark, you need to create and register them. The process involves defining a function and registering it with Spark. You can define UDFs for both SQL and DataFrame operations. In this section, you will see the basic syntax of defining a UDF in Spark and then registering that UDF with Spark. You can write any custom Python code in your UDF for your application's logic. The first example is in Python; the next example is in Scala.

Creating UDFs in Python

We can use the following code to create a UDF in Python:

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType
# Define a UDF in Python
def my_udf_function(input_param):
# Your custom logic here
return processed_value
# Register the UDF with Spark
my_udf = udf(my_udf_function, IntegerType())
# Using the UDF in a DataFrame operation
df = df.withColumn("new_column", my_udf(df["input_column"]))
```

Creating UDFs in Scala

We can use the following code to create a UDF in Scala:

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.types._
// Define a UDF in Scala
val myUDF: UserDefinedFunction = udf((inputParam: InputType) => {
// Your custom logic here
processedValue }, OutputType)
// Using the UDF in a DataFrame operation
val df = df.withColumn("newColumn", myUDF(col("inputColumn")))
```

Use cases for UDFs

UDFs are versatile and can be used in a wide range of scenarios, including, but not limited to, the following:

- **Data transformation**: Applying custom logic to transform data, such as data enrichment, cleansing, and feature engineering
- **Complex calculations**: Implementing complex mathematical or statistical operations not available in Spark's standard functions
- **String manipulation**: Parsing and formatting strings, regular expressions, and text processing
- **Machine learning**: Creating custom functions for feature extraction, preprocessing, or post-processing in machine learning workflows
- **Domain-specific logic**: Implementing specific domain-related logic that is unique to your use case

Best practices for using UDFs

When working with UDFs in Spark, consider the following best practices:

- **Avoid performance bottlenecks**: UDFs can impact performance, especially when used with large datasets. Profile and monitor your application to identify performance bottlenecks.
- **Minimize UDF complexity**: Keep UDFs simple and efficient to avoid slowing down your Spark application. Complex operations can lead to longer execution times.
- **Check for data type compatibility**: Ensure that the UDF's output data type matches the column data type to avoid errors and data type mismatches.
- **Optimize data processing**: Consider using built-in Spark functions whenever possible as they are highly optimized for distributed data processing.
- **Use vectorized UDFs**: In some Spark versions, vectorized UDFs are available, which can significantly improve UDF performance by processing multiple values at once.
- **Test and validate**: Test your UDFs thoroughly on small subsets of data before applying them to the entire dataset. Ensure they produce the desired results.
- **Document UDFs**: Document your UDFs with comments and descriptions to make your code more maintainable and understandable to others.

In this section, we explored the concept of UDFs in Apache Spark. UDFs are powerful tools for extending Spark's capabilities and performing custom data transformations and operations. When used judiciously and efficiently, UDFs can help you address a wide range of data processing challenges in Spark.

Now that we've covered the advanced operations in Spark, we will dive into the concept of Spark optimization.

Optimizations in Apache Spark

Apache Spark, renowned for its distributed computing capabilities, offers a suite of advanced optimization techniques that are crucial for maximizing performance, improving resource utilization, and enhancing the efficiency of data processing jobs. These techniques go beyond basic optimizations, allowing users to fine-tune and optimize Spark applications for optimal execution.

Understanding optimization in Spark

Optimization in Spark aims to fine-tune the execution of jobs to improve speed, resource utilization, and overall performance.

Apache Spark is well-known for its powerful optimization capabilities, which significantly enhance the performance of distributed data processing tasks. At the heart of this optimization framework lies the Catalyst optimizer, an integral component that plays a pivotal role in enhancing query execution efficiency. This is achieved before the query is executed.

The Catalyst optimizer works primarily on static optimization plans that are generated during query compilation. However, AQE, which was introduced in Spark 3.0, is a dynamic and adaptive approach to optimizing query plans at runtime based on the actual data characteristics and execution environment. We will learn more about both these paradigms in the next section.

Catalyst optimizer

The Catalyst optimizer is an essential part of Apache Spark's query execution engine. It is a powerful tool that uses advanced techniques to optimize query plans, thus improving the performance of Spark applications. The term "*catalyst*" refers to its ability to spark transformations in the query plan and make it more efficient.

Let's look at some of the key characteristics of the Catalyst optimizer:

- **Rule-based optimization**: The Catalyst optimizer employs a set of rules and optimizations to transform and enhance query plans. These rules cover a wide range of query optimization scenarios.
- **Logical and physical query plans**: It works with both logical and physical query plans. The logical plan represents the abstract structure of a query, while the physical plan outlines how to execute it.
- **Extensibility**: Users can define custom rules and optimizations. This extensibility allows you to tailor the optimizer to your specific use case.
- **Cost-based optimization**: The Catalyst optimizer can evaluate the cost of different query plans and choose the most efficient one based on cost estimates. This is particularly useful when dealing with complex queries.

Let's take a look at the different components that make up the Catalyst optimizer.

Catalyst optimizer components

To gain a deeper understanding of the Catalyst optimizer, it's essential to examine its core components.

Logical query plan

The logical query plan represents the high-level, abstract structure of a query. It defines what you want to accomplish without specifying how to achieve it. Spark's Catalyst optimizer works with this logical plan to determine the optimal physical plan.

Rule-based optimization

Rule-based optimization is the backbone of the Catalyst optimizer. It comprises a set of rules that transform the logical query plan into a more efficient version. Each rule focuses on a specific aspect of optimization, such as predicate pushdown, constant folding, or column pruning.

Physical query plan

The physical query plan defines how to execute the query. Once the logical plan is optimized using rule-based techniques, it's converted into a physical plan, taking into account the available resources and the execution environment. This phase ensures that the plan is executable in a distributed and parallel manner.

Cost-based optimization

In addition to rule-based optimization, the Catalyst optimizer can use cost-based optimization. It estimates the cost of different execution plans, taking into account factors such as data distribution, join strategies, and available resources. This approach helps Spark choose the most efficient plan based on actual execution characteristics.

Catalyst optimizer in action

To witness the Catalyst optimizer in action, let's consider a practical example using Spark's SQL API.

In this code example, we're loading data from a CSV file, applying a selection operation to pick specific columns, and filtering rows based on a condition. By calling `explain()` on the resulting DataFrame, we can see the optimized query plan that was generated by the Catalyst optimizer. The output provides insights into the physical execution steps Spark will perform:

```
# SparkSession setup
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("CatalystOptimizerExample").
getOrCreate()
# Load data
df = spark.read.csv("/salary_data.csv", header=True, inferSchema=True)
# Query with Catalyst Optimizer
result_df = df.select("employee", "department").filter(df["salary"] >
3500)
# Explain the optimized query plan
result_df.explain()
```

This explanation from the `explain()` method often includes details about the physical execution plan, the use of specific optimizations, and the chosen strategies for query execution.

By examining the query plan and understanding how the Catalyst optimizer enhances it, you can gain valuable insights into the inner workings of Spark's optimization engine.

This section provided a solid introduction to the Catalyst optimizer, its components, and a practical example. You can expand on this foundation by delving deeper into rule-based and cost-based optimization techniques, as well as discussing real-world scenarios where the Catalyst optimizer can have a substantial impact on query performance.

Next, we will see how AQE takes optimizations to the next level in Spark.

Adaptive Query Execution (AQE)

Apache Spark, a powerful distributed computing framework, offers a multitude of optimization techniques to enhance the performance of data processing jobs. One such advanced optimization feature is AQE, a dynamic approach that significantly improves query processing efficiency.

AQE dynamically adjusts execution plans during runtime based on actual data statistics and hardware conditions. It collects and utilizes runtime statistics to optimize join strategies, partitioning methods, and broadcast operations.

Let's look at its key components:

- **Runtime statistics collection**: AQE collects runtime statistics, such as data size, skewness, and partitioning, during query execution
- **Adaptive optimization rules**: It utilizes collected statistics to adjust and optimize join strategies, partitioning methods, and broadcast operations dynamically

Now, let's consider its benefits and significance:

- **Improved performance**: AQE significantly enhances performance by optimizing execution plans dynamically, leading to better resource utilization and reduced execution time
- **Handling variability**: It efficiently handles variations in data sizes, skewed data distributions, and changing hardware conditions during query execution
- **Efficient resource utilization**: It optimizes query plans in real time, leading to better resource utilization and reduced execution time

AQE workflow

Let's look at how AQE optimizes workflows in Spark 3.0:

- **Runtime statistics collection**: During query execution, Spark collects statistics related to data distribution, partition sizes, and join keys' cardinality
- **Adaptive optimization**: Utilizing the collected statistics, Spark dynamically adjusts the query execution plan, optimizing join strategies, partitioning methods, and data redistribution techniques
- **Enhanced performance**: The adaptive optimization ensures that Spark adapts to changing data and runtime conditions, resulting in improved query performance and resource utilization

AQE in Apache Spark represents a significant advancement in query optimization, moving beyond static planning to adapt to runtime conditions and data characteristics. By dynamically adjusting execution plans based on real-time statistics, it optimizes query performance, ensuring efficient and scalable processing of large-scale datasets.

Next, we will see how Spark does cost-based optimizations.

Cost-based optimization

Spark estimates the cost of executing different query plans based on factors such as data size, join operations, and shuffle stages. It utilizes cost estimates to select the most efficient query execution plan.

Here are the benefits:

- **Optimal plan selection**: Cost-based optimization chooses the most cost-effective execution plan while considering factors such as join strategies and data distribution
- **Performance improvement**: Minimizing unnecessary shuffling and computations improves query performance

Next, we will see how Spark utilizes memory management and tuning for optimizations.

Memory management and tuning

Spark also applies efficient memory allocation strategies, including storage and execution memory, to avoid unnecessary spills and improve processing. It fine-tunes garbage collection settings to minimize interruptions and improve overall job performance.

Here are its benefits:

- **Reduced overheads**: Optimized memory usage minimizes unnecessary spills to disk, reducing overheads and improving job performance
- **Stability and reliability**: Tuned garbage collection settings enhance stability and reduce pauses, ensuring more consistent job execution

Advanced Spark optimization techniques, including AQE, cost-based optimization, the Catalyst optimizer, and memory management, play a vital role in improving Spark job performance, resource utilization, and overall efficiency. By leveraging these techniques, users can optimize Spark applications to meet varying data processing demands and enhance their scalability and performance.

So far, we have seen how Spark optimizes its query plans internally. However, there are other optimizations that users can implement to make Spark's performance even better. We will discuss some of these optimizations next.

Data-based optimizations in Apache Spark

In addition to Spark's inner optimizations, there are certain things we can take care of in terms of implementation to make Spark more efficient. These are user-controlled optimizations. If we are aware of these challenges and how to handle them in real-world data applications, we can utilize Spark's distributed architecture to its fullest.

We'll start by looking at a very common occurrence in distributed frameworks called the small file problem.

Addressing the small file problem in Apache Spark

The small file problem poses a significant challenge in distributed computing frameworks such as Apache Spark as it impacts performance and efficiency. It arises when data is stored in numerous small files rather than consolidated in larger files, leading to increased overhead and suboptimal resource utilization. In this section, we'll delve into the implications of the small file problem in Spark and explore effective solutions to mitigate its effects.

The key challenges associated with the small file problem are as follows:

- **Increased metadata overhead**: Storing data in numerous small files leads to higher metadata overhead as each file occupies a separate block and incurs additional I/O operations for file handling
- **Reduced throughput**: Processing numerous small files is less efficient as it involves a high level of overhead for opening, reading, and closing files, resulting in reduced throughput
- **Inefficient resource utilization**: Spark's parallelism relies on data partitioning, and small files can lead to inadequate partitioning, underutilizing resources, and hindering parallel processing

Now that we've discussed the key challenges, let's discuss some solutions to mitigate the small file problem:

- **File concatenation or merging**: Consolidating small files into larger files can significantly alleviate the small file problem. Techniques such as file concatenation or merging, either manually or through automated processes, help reduce the number of individual files.
- **File compaction or coalescing**: Tools or processes that compact or coalesce small files into fewer, more substantial files can streamline data storage. This consolidation reduces metadata overhead and enhances data access efficiency.
- **File format optimization**: Choosing efficient file formats such as Parquet or ORC, which support columnar storage and compression, can reduce the impact of small files. These formats facilitate efficient data access and reduce storage space.
- **Partitioning strategies**: Applying appropriate partitioning strategies during data ingestion or processing in Spark can mitigate the effects of the small file problem. It involves organizing data into larger partitions to improve parallelism.

- **Data prefetching or caching**: Prefetching or caching small files into memory before processing can minimize I/O overhead. Techniques such as caching or loading data into memory using Spark's capabilities can improve performance.
- **AQE**: Leveraging Spark's AQE features helps optimize query plans based on runtime statistics. This can mitigate the impact of small files during query execution.
- **Data lake architectural changes**: Reevaluating the data lake architecture and adopting data ingestion strategies that minimize the creation of small files can prevent the problem at its source.

Let's look at the best practices for handling small files:

- **Regular monitoring and cleanup**: Implement regular monitoring and cleanup processes to identify and merge small files that are generated over time
- **Optimize the storage layout**: Design data storage layouts that minimize the creation of small files while considering factors such as block size and filesystem settings
- **Automated processes**: Use automated processes or tools to consolidate and manage small files efficiently, reducing manual effort
- **Educate data producers**: Educate data producers on the impact of small files and encourage practices that generate larger files or optimize file creation

By adopting these strategies and best practices, organizations can effectively mitigate the small file problem in Apache Spark, ensuring improved performance, enhanced resource utilization, and efficient data processing capabilities. These approaches empower users to overcome the challenges posed by the small file problem and optimize their Spark workflows for optimal performance and scalability.

Next, we will see how data skew affects performance in Spark.

Tackling data skew in Apache Spark

Data skew presents a significant challenge in distributed data processing frameworks such as Apache Spark, causing uneven workload distribution and hindering parallelism.

Data skew occurs when certain keys or partitions hold significantly more data than others. This imbalance leads to unequal processing times for different partitions, causing stragglers. Skewed data distribution can result in certain worker nodes being overloaded while others remain underutilized, leading to inefficient resource allocation. Tasks that deal with skewed data partitions take longer to complete, causing delays in job execution and affecting overall performance.

Here are some of the solutions we can use to address data skew:

- **Partitioning techniques**:
  - **Salting**: Introduce randomness by adding a salt to keys to distribute data more evenly across partitions. This helps prevent hotspots and balances the workload.
  - **Custom partitioning**: Implement custom partitioning logic to redistribute skewed data by grouping keys differently, ensuring a more balanced distribution across partitions.
- **Skew-aware algorithms**: Utilize techniques such as skew join optimization, which handles skewed keys separately from regular joins, redistributing and processing them more efficiently
- **Replicate small-skewed data**: Replicate small skewed partitions across multiple nodes to parallelize processing and alleviate the load on individual nodes
- **AQE**: Leverage Spark's AQE capabilities to dynamically adjust execution plans based on runtime statistics, mitigating the impact of data skew
- **Sampling and filtering**: Apply sampling and filtering techniques to identify skewed data partitions beforehand, allowing for proactive handling of skewed keys during processing
- **Dynamic resource allocation**: Implement dynamic resource allocation to allocate additional resources to tasks dealing with skewed data partitions, optimizing resource utilization

Let's discuss the best practices for handling data skew:

- **Regular profiling**: Continuously profile and monitor data distribution to identify and address skew issues early in the processing pipeline
- **Optimized partitioning**: Choose appropriate partitioning strategies based on data characteristics to prevent or mitigate data skew
- **Distributed processing**: Leverage distributed processing frameworks to distribute skewed data across multiple nodes for parallel execution
- **Task retry mechanisms**: Implement retry mechanisms for tasks dealing with skewed data to accommodate potential delays and avoid job failures
- **Data preprocessing**: Apply preprocessing techniques to mitigate skew before data processing, ensuring a more balanced workload

By employing these strategies and best practices, organizations can effectively combat data skew in Apache Spark, ensuring more balanced workloads, improved resource utilization, and enhanced overall performance in distributed data processing workflows. These approaches empower users to overcome the challenges posed by data skew and optimize Spark applications for efficient and scalable data processing.

Addressing data skew in Apache Spark is critical for optimizing performance and ensuring efficient resource utilization in distributed computing environments. By understanding the causes and impacts of data skew and employing mitigation strategies users can significantly improve the efficiency and reliability of Spark jobs, mitigating the adverse effects of data skew.

In the next section, we will talk about data spills in Spark and how to manage them.

Managing data spills in Apache Spark

Data spill, something that's often encountered in distributed processing frameworks such as Apache Spark, occurs when the data being processed exceeds the available memory capacity, leading to data being written to disk. This phenomenon can significantly impact performance and overall efficiency. In this section, we'll delve into the implications of data spill in Spark and effective strategies to mitigate its effects for optimized data processing.

Data spill occurs when Spark's memory capacity is exceeded, resulting in excessive data write operations to disk, which are significantly slower than in-memory operations. Writing data to disk incurs high I/O overhead, leading to a substantial degradation in processing performance due to increased latency. Data spillage can cause resource contention as disk operations compete with other computing tasks, leading to inefficient resource utilization.

Here are some of the solutions we can implement to address data spill:

- **Memory management techniques**:
  - **Increase executor memory**: Allocating more memory to Spark executors can help reduce the likelihood of data spill by accommodating larger datasets in memory
  - **Tune memory configuration**: Optimize Spark's memory configurations, such as adjusting memory fractions for storage and execution, to better utilize available memory
- **Partitioning and caching strategies**:
  - **Repartitioning**: Repartitioning data into an optimal number of partitions can help manage memory usage and minimize data spills by ensuring better data distribution across nodes
  - **Caching intermediate results**: Caching or persisting intermediate datasets in memory can prevent recomputation and reduce the chances of data spill during subsequent operations
- **Advanced optimization techniques**:
  - **Shuffle tuning**: Tune shuffle operations by adjusting parameters such as shuffle partitions and buffer sizes to reduce the likelihood of data spill during shuffle phases
  - **Data compression**: Utilize data compression techniques when storing intermediate data in memory or on disk to reduce the storage footprint and alleviate memory pressure

- **AQE**: Leverage Spark's AQE capabilities to dynamically adjust execution plans based on runtime statistics, optimizing memory usage and reducing spillage.
- **Task and data skew handling**: Apply techniques to mitigate task and data skew. Skewed data can exacerbate memory pressure and increase the chances of data spill.

Here are the best practices for handling data spills:

- **Resource monitoring**: Regularly monitor memory usage and resource allocation to identify and preempt potential data spillage issues
- **Optimized data structures**: Utilize optimized data structures and formats (such as Parquet or ORC) to reduce memory overhead and storage requirements
- **Efficient caching strategies**: Strategically cache or persist intermediate results to minimize recomputation and reduce the probability of data spill
- **Incremental processing**: Employ incremental processing techniques to handle large datasets in manageable chunks, reducing memory pressure

By adopting these strategies and best practices, organizations can effectively manage data spillage in Apache Spark, ensuring efficient memory utilization, optimized processing performance, and enhanced overall scalability in distributed data processing workflows. These approaches empower users to proactively address data spillage challenges and optimize Spark applications for improved efficiency and performance.

In the next section, we will talk about what data shuffle is and how to handle it to optimize performance.

Managing data shuffle in Apache Spark

Data shuffle, a fundamental operation in distributed processing frameworks such as Apache Spark, involves moving data across nodes in the cluster. While shuffle operations are essential for various transformations, such as joins and aggregations, they can also introduce performance bottlenecks and resource overhead. In this section, we'll explore the implications of data shuffle in Spark and effective strategies to optimize and mitigate its impact for efficient data processing.

Data shuffle involves extensive network and disk I/O operations, leading to increased latency and resource utilization. Shuffling large amounts of data across nodes can introduce performance bottlenecks due to excessive data movement and processing. Intensive shuffle operations can cause resource contention among nodes, impacting overall cluster performance.

Let's discuss the solutions for optimizing data shuffle:

- **Data partitioning techniques**: Implement optimized data partitioning strategies to reduce shuffle overhead, ensuring a more balanced workload distribution
- **Skew handling**: Mitigate data skew by employing techniques such as salting or custom partitioning to prevent hotspots and balance data distribution

- **Shuffle partitions adjustment**: Tune the number of shuffle partitions based on data characteristics and job requirements to optimize shuffle performance and reduce overhead
- **Memory management**: Optimize memory allocation for shuffle operations to minimize spills to disk and improve overall shuffle performance
- **Data filtering and pruning**: Apply filtering or pruning techniques to reduce the amount of data shuffled across nodes, focusing only on relevant subsets of data
- **Join optimization**:
  - **Broadcast joins**: Utilize broadcast joins for smaller datasets to replicate them across nodes, minimizing data shuffling and improving join performance
  - **Sort-merge joins**: Employ sort-merge join algorithms for large datasets to minimize data movement during join operations
- **AQE**: Leverage Spark's AQE capabilities to dynamically optimize shuffle operations based on runtime statistics and data distribution

The best practices for managing data shuffle are as follows:

- **Profile and monitor**: Continuously profile and monitor shuffle operations to identify bottlenecks and optimize configurations
- **Optimized partition sizes**: Determine optimal partition sizes based on data characteristics and adjust shuffle partitioning accordingly
- **Caching and persistence**: Cache or persist intermediate shuffle results to reduce recomputation and mitigate shuffle overhead
- **Regular tuning**: Regularly tune Spark configurations related to shuffle operations based on workload requirements and cluster resources

By implementing these strategies and best practices, organizations can effectively optimize data shuffle operations in Apache Spark, ensuring improved performance, reduced resource contention, and enhanced overall efficiency in distributed data processing workflows. These approaches empower users to proactively manage and optimize shuffle operations for streamlined data processing and improved cluster performance.

Despite all the data-related challenges that users need to be aware of, there are certain types of joins that Spark has available in its internal working that we can utilize for better performance. We'll take a look at these next.

Shuffle and broadcast joins

Apache Spark offers two fundamental approaches for performing join operations: shuffle joins and broadcast joins. Each method has its advantages and use cases, and understanding when to use them is crucial for optimizing your Spark applications. Note that these joins are done by Spark automatically to join different datasets together. You can enforce some of the join types in your code but Spark takes care of the execution.

Shuffle joins

Shuffle joins are a common method for joining large datasets in distributed computing environments. These joins redistribute data across partitions, ensuring that matching keys end up on the same worker nodes. Spark performs shuffle joins efficiently thanks to its underlying execution engine.

Here are some of the key characteristics of shuffle joins:

- **Data redistribution**: Shuffle joins redistribute data to ensure that rows with matching keys are co-located on the same worker nodes. This process may require substantial network and disk I/O.
- **Suitable for large datasets**: Shuffle joins are well-suited for joining large DataFrames with comparable sizes.
- **Replicating data**: During a shuffle join, data may be temporarily replicated on worker nodes to facilitate efficient joins.
- **Costly in terms of network and disk I/O**: Shuffle joins can be resource-intensive due to data shuffling, making them slower compared to other join techniques for smaller datasets.
- **Examples**: Inner join, left join, right join, and full outer join are often implemented as shuffle joins.

Use case

Shuffle joins are typically used when joining two large DataFrames with no significant size difference.

Shuffle sort-merge joins

A shuffle sort-merge join is a type of shuffle join that leverages a combination of sorting and merging techniques to perform the join operation. It sorts both DataFrames based on the join key and then merges them efficiently.

Here are some of the key features of shuffle sort-merge joins:

- **Data sorting**: Shuffle sort-merge joins sort the data on both sides to ensure efficient merging
- **Suitable for large datasets**: They are efficient for joining large DataFrames with skewed data distribution

- **Complexity**: This type of shuffle join is more complex than a simple shuffle join as it involves sorting operations

Use case

Shuffle sort-merge joins are effective for large-scale joins, especially when the data distribution is skewed, and a balanced distribution of data across partitions is essential.

Let's look at broadcast joins next.

Broadcast joins

Broadcast joins are a highly efficient technique for joining a small DataFrame with a larger one. In this approach, the smaller DataFrame is broadcast to all worker nodes, eliminating the need for shuffling data across the network. A broadcast join is a specific optimization technique that can be applied when one of the DataFrames is small enough to fit in memory. In this case, the small DataFrame is broadcast to all worker nodes, avoiding costly shuffling.

Let's look at some of the key characteristics of broadcast joins:

- **Small DataFrame broadcast**: The smaller DataFrame is broadcast to all worker nodes, ensuring that it is available locally
- **Reduced network overhead**: Broadcast joins significantly reduce network and disk I/O because they avoid data shuffling
- **Ideal for dimension tables**: Broadcast joins are commonly used when joining a fact table with smaller dimension tables, such as in data warehousing scenarios
- **Efficient for small-to-large joins**: They are efficient for joins where one DataFrame is significantly smaller than the other

Use case

Broadcast joins are useful when you're joining a large DataFrame with a much smaller one, such as joining a fact table with dimension tables in a data warehouse.

Broadcast hash joins

A specific type of broadcast join is the broadcast hash join. In this variant, the smaller DataFrame is broadcast as a hash table to all worker nodes, which allows for efficient lookups in the larger DataFrame.

Use case

Broadcast hash joins are suitable for scenarios where one DataFrame is small enough to be broadcast, and you need to perform equality-based joins.

In this section, we discussed two fundamental join techniques in Spark – shuffle joins and broadcast joins – including specific variants, such as the broadcast hash join and the shuffle sort-merge join. Choosing the right join method depends on the size of your DataFrames, data distribution, and network considerations, and it's essential to make informed decisions to optimize your Spark applications. In the next section, we will cover different types of transformations that exist in Spark.

Narrow and wide transformations in Apache Spark

As discussed in *Chapter 3*, transformations are the core operations for processing data. Transformations are categorized into two main types: narrow transformations and wide transformations. Understanding the distinction between these two types of transformations is essential for optimizing the performance of your Spark applications.

Narrow transformations

Narrow transformations are operations that do not require data shuffling or extensive data movement across partitions. They can be executed on a single partition without the need to communicate with other partitions. This inherent locality makes narrow transformations highly efficient and faster to execute.

The following are some of the key characteristics of narrow transformations:

- **Single-partition processing**: Narrow transformations operate on a single partition of the data independently, which minimizes communication overhead.
- **Speed and efficiency**: Due to their partition-wise nature, narrow transformations are fast and efficient.

`map()`, `filter()`, `union()`, and `groupBy()` are typical examples of narrow transformations.

Wide transformations

Wide transformations, in contrast, involve data shuffling, which necessitates the exchange of data between partitions. These transformations require communication between multiple partitions and can be resource-intensive. As a result, they tend to be slower and more costly in terms of computation.

Here are a few of the key characteristics of wide transformations:

- **Data shuffling**: Wide transformations involve the reorganization of data across partitions, requiring data exchange between different workers.
- **Slower execution**: Due to the need for shuffling, wide transformations are relatively slower and resource-intensive compared to narrow transformations.

`groupByKey()`, `reduceByKey()`, and `join()` are common examples of wide transformations.

Let's discuss which transformation works best, depending on the operation.

Choosing between narrow and wide transformations

Selecting the appropriate type of transformation depends on the specific use case and the data at hand. Here are some considerations for choosing between narrow and wide transformations:

- **Data size**: If your data is small enough to fit comfortably within a single partition, it's preferable to use narrow transformations. This minimizes the overhead associated with shuffling.
- **Data distribution**: If your data is distributed unevenly across partitions, wide transformations might be necessary to reorganize and balance the data.
- **Performance**: Narrow transformations are typically faster and more efficient, so if performance is a critical concern, they are preferred.
- **Complex operations**: Some operations, such as joining large DataFrames, often require wide transformations. In such cases, the performance trade-off is inevitable.
- **Cluster resources**: Consider the available cluster resources. Resource-intensive wide transformations may lead to resource contention in a shared cluster.

Next, we'll learn how to optimize wide transformations in cases where it is necessary to implement them.

Optimizing wide transformations

While wide transformations are necessary for certain operations, it's crucial to optimize them to reduce their impact on performance. Here are some strategies for optimizing wide transformations:

- **Minimize data shuffling**: Whenever possible, use techniques to minimize data shuffling. For example, consider using broadcast joins for small DataFrames.
- **Partitioning**: Carefully choose the number of partitions and partitioning keys to ensure even data distribution, reducing the need for extensive shuffling.
- **Caching and persistence**: Caching frequently used DataFrames can help reduce the need for recomputation and shuffling in subsequent stages.
- **Tuning cluster resources**: Adjust cluster configurations, such as the number of executors and memory allocation, to meet the demands of wide transformations.
- **Profiling and monitoring**: Regularly profile and monitor your Spark applications to identify performance bottlenecks, especially in the case of wide transformations.

In this section, we explored the concepts of narrow and wide transformations in Apache Spark. Understanding when and how to use these transformations is critical for optimizing the performance of your Spark applications, especially when dealing with large datasets and complex operations.

In the next section, we will cover the persist and cache operations in Spark.

Persisting and caching in Apache Spark

In Apache Spark, optimizing the performance of your data processing operations is essential, especially when working with large datasets and complex workflows. Caching and persistence are techniques that allow you to store intermediate or frequently used data in memory or on disk, reducing the need for recomputation and enhancing overall performance. This section explores the concepts of persisting and caching in Spark.

Understanding data persistence

Data persistence is the process of storing the intermediate or final results of Spark transformations in memory or on disk. By persisting data, you reduce the need to recompute it from the source data, thereby improving query performance.

The following key concepts are related to data persistence:

- **Storage levels**: Spark offers multiple storage levels for data, ranging from memory-only to disk, depending on your needs. Each storage level comes with its trade-offs in terms of speed and durability.
- **Lazy evaluation**: Spark follows a lazy evaluation model, meaning transformations are not executed until an action is called. Data persistence ensures that the intermediate results are available for reuse without recomputation.
- **Caching versus persistence**: Caching is a specific form of data persistence that stores data in memory, while persistence encompasses both in-memory and on-disk storage.

Caching data

Caching is a form of data persistence that stores DataFrames, RDDs, or datasets in memory for fast access. It is an essential optimization technique that improves the performance of Spark applications, particularly when dealing with iterative algorithms or repeated computations.

To cache a DataFrame or an RDD, you can use the `.cache()` or `.persist()` method while specifying the storage level:

- **Memory-only**: This option stores data in memory but does not replicate it for fault tolerance. Use `.cache()` or `.persist(StorageLevel.MEMORY_ONLY)`.
- **Memory-only, serialized**: This option stores data in memory in a serialized form, reducing memory usage. Use `.persist(StorageLevel.MEMORY_ONLY_SER)`.
- **Memory and disk**: This option stores data in memory and spills excess data to disk when memory is full. Use `.persist(StorageLevel.MEMORY_AND_DISK)`.

- **Disk-only**: This option stores data only on disk, avoiding memory usage. Use `.persist(StorageLevel.DISK_ONLY)`.

Caching is particularly beneficial in the following scenarios:

- **Iterative algorithms**: Caching is vital for iterative algorithms such as machine learning, graph processing, and optimization problems, where the same data is used repeatedly
- **Multiple actions**: When a DataFrame is used for multiple actions, caching it after the first action can improve performance
- **Avoiding recomputation**: Caching helps avoid recomputing the same data when multiple transformations depend on it
- **Interactive queries**: In interactive data exploration or querying, caching frequently used intermediate results can speed up ad hoc analysis

Unpersisting data

Caching consumes memory, and in a cluster environment, it's essential to manage memory efficiently. You can release cached data from memory using the `.unpersist()` method. This method allows you to specify whether to release the data immediately or only when it is no longer needed.

Here's an example of unpersisting data:

```
# Cache a DataFrame
df.cache()
# Unpersist the cached DataFrame
df.unpersist()
```

Best practices

To use caching and persistence effectively in your Spark applications, consider the following best practices:

- **Cache only what's necessary**: Caching consumes memory, so cache only the data that is frequently used or costly to compute
- **Monitor memory usage**: Regularly monitor memory usage to avoid running out of memory or excessive disk spills
- **Automate unpersistence**: If you have limited memory resources, automate the unpersistence of less frequently used data to free up memory for more critical operations
- **Consider serialization**: Depending on your use case, consider using serialized storage levels to reduce memory overhead

In this section, we explored the concepts of persistence and caching in Apache Spark. Caching and persistence are powerful techniques for optimizing performance in Spark applications, particularly when dealing with iterative algorithms or scenarios where the same data is used repeatedly. Understanding when and how to use these techniques can significantly improve the efficiency of your data processing workflows.

In the next section, we'll learn how repartition and coalesce work in Spark.

Repartitioning and coalescing in Apache Spark

Efficient data partitioning plays a crucial role in optimizing data processing workflows in Apache Spark. Repartitioning and coalescing are operations that allow you to control the distribution of data across partitions. In this section, we'll explore the concepts of repartitioning and coalescing and their significance in Spark applications.

Understanding data partitioning

Data partitioning in Apache Spark involves dividing a dataset into smaller, manageable units called partitions. Each partition contains a subset of the data and is processed independently by different worker nodes in a distributed cluster. Proper data partitioning can significantly impact the efficiency and performance of Spark applications.

Repartitioning data

Repartitioning is the process of redistributing data across a different number of partitions. This operation can help balance data distribution, improve parallelism, and optimize data processing. You can use the `.repartition()` method to specify the number of desired partitions.

Here are some key points related to repartitioning data:

- **Increasing or decreasing partitions**: Repartitioning allows you to increase or decrease the number of partitions to suit your processing needs.
- **Data shuffling**: Repartitioning often involves data shuffling, which can be resource-intensive. Therefore, it should be used judiciously.
- **Even data distribution**: Repartitioning is useful when the original data is unevenly distributed across partitions, causing skewed workloads.
- **Optimizing for joins**: Repartitioning can be beneficial when performing joins to minimize data shuffling.

Here's an example of repartitioning data:

```
# Repartition a DataFrame into 8 partitions
df.repartition(8)
```

Coalescing data

Coalescing is the process of reducing the number of partitions while preserving data locality. It is a more efficient operation than repartitioning because it avoids unnecessary data shuffling whenever possible. You can use the `.coalesce()` method to specify the target number of partitions.

Here are some key points related to coalescing data:

- **Decreasing partitions**: Coalescing is used when you want to decrease the number of partitions to optimize data processing
- **Minimizing data movement**: Unlike repartitioning, coalescing minimizes data shuffling by merging partitions locally whenever possible
- **Efficient for data reduction**: Coalescing is efficient when you need to reduce the number of partitions without incurring the full cost of data shuffling

Here's an example of coalescing data:

```
# Coalesce a DataFrame to 4 partitions
df.coalesce(4)
```

Use cases for repartitioning and coalescing

Understanding when to repartition and coalesce is critical for optimizing your Spark applications.

The following are some use cases for repartitioning:

- **Data skew**: When data is skewed across partitions, repartitioning can balance the workload
- **Join optimization**: For optimizing join operations by ensuring that the joining keys are collocated
- **Parallelism control**: Adjusting the level of parallelism to optimize resource utilization

Now, let's look at some use cases for coalescing:

- **Reducing data**: When you need to reduce the number of partitions to save memory and reduce overhead
- **Minimizing shuffling**: To avoid unnecessary data shuffling and minimize network communication
- **Post-filtering**: After applying a filter or transformation that significantly reduces the dataset size

Best practices

To repartition and coalesce effectively in your Spark applications, consider these best practices:

- **Profile and monitor**: Profile your application to identify performance bottlenecks related to data partitioning. Use Spark's UI and monitoring tools to track data shuffling.

- **Consider data size**: Consider the size of your dataset and the available cluster resources when deciding on the number of partitions.
- **Balance workloads**: Aim for a balanced workload distribution across partitions to optimize parallelism.
- **Coalesce where possible**: When reducing the number of partitions, prefer coalescing over repartitioning to minimize data shuffling.
- **Plan for joins**: When performing joins, plan for the optimal number of partitions to minimize shuffle overhead.

In this section, we explored the concepts of repartitioning and coalescing in Apache Spark. Understanding how to efficiently control data partitioning can significantly impact the performance of your Spark applications, especially when you're working with large datasets and complex operations.

Summary

In this chapter, we delved into advanced data processing capabilities in Apache Spark, enhancing your understanding of key concepts and techniques. We explored the intricacies of Spark's Catalyst optimizer, the power of different types of Spark joins, the importance of data persistence and caching, the significance of narrow and wide transformations, and the role of data partitioning using repartition and coalesce. Additionally, we discovered the versatility and utility of UDFs.

As you advance in your journey with Apache Spark, these advanced capabilities will prove invaluable for optimizing and customizing your data processing workflows. By harnessing the potential of the Catalyst optimizer, you can fine-tune query execution for improved performance. Understanding the nuances of Spark joins empowers you to make informed decisions on which type of join to employ for specific use cases. Data persistence and caching become indispensable when you seek to reduce recomputation and expedite iterative processes.

Narrow and wide transformations play a pivotal role in achieving the desired parallelism and resource efficiency in Spark applications. Proper data partitioning through repartition and coalesce ensures balanced workloads and optimal data distribution.

UDFs open the door to limitless possibilities, enabling you to implement custom data processing logic, from data cleansing and feature engineering to complex calculations and domain-specific operations. However, it is crucial to use UDFs judiciously, optimizing them for performance and adhering to best practices.

With this chapter's knowledge, you are better equipped to tackle complex data processing challenges in Apache Spark, enabling you to extract valuable insights from your data efficiently and effectively. These advanced capabilities empower you to leverage the full potential of Spark and achieve optimal performance in your data-driven endeavors.

In the next chapter, we will be introduced to SparkSQL and will learn how to create and manipulate SQL queries in Spark.

Sample questions

Question 1:

Which of the following code blocks returns a DataFrame showing the mean of the `salary` column of the `df` DataFrame, grouped by the `department` column?

A. `df.groupBy("department").agg(avg("salary"))`

B. `df.groupBy(col(department).avg())`

C. `df.groupBy("department").avg(col("salary"))`

D. `df.groupBy("department").agg(average("salary"))`

Question 2:

Which of the following code blocks returns unique values across all values in the `state` and `department` columns in `df`?

A. `df.select(state).join(transactionsDf.select('department'), col(state)==col('department'), 'outer').show()`

B. `df.select(col('state'), col('department')).agg({'*': 'count'}).show()`

C. `df.select('state', 'department').distinct().show()`

D. `df.select('state').union(df.select('department')).distinct().show()`

Answers

1. A
2. D

669FD03CAE074

6
SQL Queries in Spark

In this chapter, we will explore the vast capabilities of Spark SQL for structured data processing. We will dive into loading and manipulating data, executing SQL queries, performing advanced analytics, and integrating Spark SQL with external systems. By the end of this chapter, you will have a solid understanding of Spark SQL's features and be equipped with the knowledge to leverage its power in your data processing tasks.

We will cover the following topics:

- What is Spark SQL?
- Getting Started with Spark SQL
- Advanced Spark SQL operations

What is Spark SQL?

Spark SQL is a powerful module within the Apache Spark ecosystem that allows for the efficient processing and analysis of structured data. It provides a higher-level interface for working with structured data compared to the traditional RDD-based API of Apache Spark. Spark SQL combines the benefits of both relational and procedural processing, enabling users to seamlessly integrate SQL queries with complex analytics. By leveraging Spark's distributed computing capabilities, Spark SQL enables scalable and high-performance data processing.

It provides a programming interface to work with structured data using SQL queries, DataFrame API, and Datasets API.

It allows users to query data using SQL-like syntax and provides a powerful engine for executing SQL queries on large datasets. Spark SQL also supports reading and writing data from various structured sources such as Hive tables, Parquet files, and JDBC databases.

Advantages of Spark SQL

Spark SQL offers several key advantages that make it a popular choice for structured data processing:

Unified data processing with Spark SQL

With Spark SQL, users can process both structured and unstructured data using a single engine. This means that users can use the same programming interface to query data stored in different formats such as JSON, CSV, and Parquet.

Users can seamlessly switch between SQL queries, DataFrame transformations, and Spark's machine learning APIs. This unified data processing approach allows for the easier integration of different data processing tasks within a single application, reducing development complexity.

Performance and scalability

Spark SQL leverages the distributed computing capabilities of Apache Spark, enabling the processing of large-scale datasets across a cluster of machines. It utilizes advanced query optimization techniques, such a the Catalyst optimizer (discussed in detail in *Chapter 5*), to optimize and accelerate query execution. Additionally, Spark SQL supports data partitioning and caching mechanisms, further enhancing performance and scalability.

Spark SQL uses an optimized execution engine that can process queries much faster than traditional SQL engines. It achieves this by using in-memory caching and optimized query execution plans.

Spark SQL is designed to scale horizontally across a cluster of machines. It can handle large datasets by partitioning them across multiple machines and processing them in parallel.

Seamless integration with existing infrastructure

Spark SQL integrates seamlessly with existing Apache Spark infrastructure and tools. It provides interoperability with other Spark components, such as Spark Streaming for real-time data processing and Spark MLlib for machine learning tasks. Furthermore, Spark SQL integrates with popular storage systems and data formats, including Parquet, Avro, ORC, and Hive, making it compatible with a wide range of data sources.

Advanced analytics capabilities

Spark SQL extends traditional SQL capabilities by using advanced analytics features. It supports window functions, which enable users to perform complex analytical operations, such as ranking, aggregation over sliding windows, and cumulative aggregations. The integration with machine learning libraries in Spark allows for the seamless integration of predictive analytics and data science workflows.

Ease of use

Spark SQL provides a simple programming interface that allows users to query data using SQL-like syntax. This makes it easy for users who are familiar with SQL to get started with Spark SQL.

Integration with Apache Spark

Spark SQL is an integral part of the Apache Spark framework and works seamlessly with other Spark components. It leverages Spark's core functionalities, such as fault tolerance, data parallelism, and distributed computing, to provide scalable and efficient data processing. Spark SQL can read data from a variety of sources, including distributed file systems (such as HDFS), object stores (like Amazon S3), and relational databases (via JDBC). It also integrates with external systems such as Hive, allowing users to leverage existing Hive metadata and queries.

Now let's take a look at some basic constructs of Spark SQL.

Key concepts – DataFrames and datasets

Spark SQL introduces two fundamental abstractions for working with structured data: DataFrames and Datasets.

DataFrames

DataFrames represent distributed collections of data organized into named columns. They provide a higher-level interface for working with structured data and offer rich APIs for data manipulation, filtering, aggregation, and querying. DataFrames are immutable and lazily evaluated, enabling optimized execution plans through Spark's Catalyst optimizer. They can be created from various data sources, including structured files (CSV, JSON, and Parquet), Hive tables, and existing RDDs.

Datasets

Datasets are an extension of DataFrames and provide a type-safe, object-oriented programming interface. Datasets combine the benefits of Spark's RDDs (strong typing and user-defined functions) with the performance optimizations of DataFrames. Datasets enable compile-time type checking and can be seamlessly converted to DataFrames, allowing for flexible and efficient data processing.

Now that we know what DataFrames and Datasets are, we'll see how to apply different Spark SQL operations to these constructs in the next section.

Getting started with Spark SQL

To get started with Spark SQL operations, we would first need to load data into a DataFrame. We'll see how to do that next. Then, we will see how we can switch between PySpark and Spark SQL data and apply different transformations to it.

Loading and saving data

In this section, we will explore various techniques for loading data into Spark SQL from different sources and saving this as a table. We will delve into Python code examples that demonstrate how to effectively load data into Spark SQL, perform the necessary transformations, and save the processed data as a table for further analysis.

Executing SQL queries in Spark SQL allows us to leverage the familiar SQL syntax and take advantage of its expressive power. Let's take a look at the syntax and an example of executing an SQL query using Spark SQL:

To execute an SQL query in Spark SQL, we use the `spark.sql()` method as follows:

```
results = spark.sql("SELECT * FROM tableName")
```

- The `spark.sql()` method is used to execute SQL queries in Spark SQL
- Inside the method, we provide the SQL query as a string argument. In this example, we select all columns from the `tableName` table
- The results of the query are stored in the `results` variable, which can be further processed or displayed as desired

To start with code examples in this chapter, we will use the DataFrame we created in *Chapter 4*:

```
salary_data_with_id = [(1, "John", "Field-eng", 3500, 40), \
    (2, "Robert", "Sales", 4000, 38), \
    (3, "Maria", "Finance", 3500, 28), \
    (4, "Michael", "Sales", 3000, 20), \
    (5, "Kelly", "Finance", 3500, 35), \
    (6, "Kate", "Finance", 3000, 45), \
    (7, "Martin", "Finance", 3500, 26), \
    (8, "Kiran", "Sales", 2200, 35), \
  ]
columns= ["ID", "Employee", "Department", "Salary", "Age"]
salary_data_with_id = spark.createDataFrame(data = salary_data_with_
id, schema = columns)
salary_data_with_id.show()
```

The output will be the following:

```
+---+--------+----------+------+---+
| ID|Employee|Department|Salary|Age|
+---+--------+----------+------+---+
|  1|    John| Field-eng|  3500| 40|
|  2|  Robert|     Sales|  4000| 38|
|  3|   Maria|   Finance|  3500| 28|
|  4| Michael|     Sales|  3000| 20|
|  5|   Kelly|   Finance|  3500| 35|
|  6|    Kate|   Finance|  3000| 45|
|  7|  Martin|   Finance|  3500| 26|
|  8|   Kiran|     Sales|  2200| 35|
+---+--------+----------+------+---+
```

I have added an `Age` column in this DataFrame for further processing.

Remember, in *Chapter 4*, we saved this DataFrame as a CSV file. The code snippet we used to read CSV files can be seen in the following code.

As you might recall, we write CSV files with Spark using this line of code:

```
salary_data_with_id.write.format("csv").mode("overwrite").
option("header", "true").save("salary_data.csv")
```

The output will be the following:

```
+---+--------+----------+------+---+
| ID|Employee|Department|Salary|Age|
+---+--------+----------+------+---+
|  1|    John| Field-eng|  3500| 40|
|  2|  Robert|     Sales|  4000| 38|
|  3|   Maria|   Finance|  3500| 28|
|  4| Michael|     Sales|  3000| 20|
|  5|   Kelly|   Finance|  3500| 35|
|  6|    Kate|   Finance|  3000| 45|
|  7|  Martin|   Finance|  3500| 26|
|  8|   Kiran|     Sales|  2200| 35|
+---+--------+----------+------+---+
```

Now that we have the DataFrame, we can use SQL operations on it:

```
# Perform transformations on the loaded data
processed_data = csv_data.filter(csv_data["Salary"] > 3000)
# Save the processed data as a table
processed_data.createOrReplaceTempView("high_salary_employees")
# Perform SQL queries on the saved table
results = spark.sql("SELECT * FROM high_salary_employees ")
results.show()
```

The output will be the following:

```
+---+--------+----------+------+---+
| ID|Employee|Department|Salary|Age|
+---+--------+----------+------+---+
|  1|    John| Field-eng|  3500| 40|
|  2|  Robert|     Sales|  4000| 38|
|  3|   Maria|   Finance|  3500| 28|
|  5|   Kelly|   Finance|  3500| 35|
|  7|  Martin|   Finance|  3500| 26|
+---+--------+----------+------+---+
```

The preceding code snippet shows how to perform a transformation on the loaded data. In this case, we filter the data to only include rows where the `Salary` column is greater than 3,000.

By using the `filter()` function, we can apply specific conditions to select the desired subset of data.

The transformed data will be stored in the results variable and ready for further analysis.

Saving transformed data as a view

Once we have performed the necessary transformations, it is often useful to save the processed data as a view for easier access and future analysis. Let's see how we can accomplish this in Spark SQL:

The `createOrReplaceTempView()` method allows us to save the processed data as a view in Spark SQL. We provide a name for the view, in this case, `high_salary_employees`.

By giving the table a meaningful name, we can easily refer to it in subsequent operations and queries. The saved table acts as a structured representation of the processed data, facilitating further analysis and exploration.

With the transformed data saved as a table, we can leverage the power of SQL queries to gain insights and extract valuable information.

By using the `spark.sql()` method, we can execute SQL queries on the saved view `high_salary_employees`.

In the preceding example, we perform a simple query to select all columns from the view based on a filter condition.

The `show()` function displays the results of the SQL query, allowing us to examine the desired information extracted from the dataset.

Another method to create a view in Spark SQL is `createTempView()`. The difference between this method and `createOrReplaceTempView()` method is that `createTempView()` would only try to create a view. If that view name already exists in a catalog, then it would throw a `TempTableAlreadyExistsException` exception.

Utilizing Spark SQL to filter and select data based on specific criteria

In this section, we will explore the syntax and practical examples of executing SQL queries and applying transformations using Spark SQL.

Let's consider a practical example where we execute an SQL query to filter and select specific data from a table:

```
# Save the processed data as a view
salary_data_with_id.createOrReplaceTempView("employees")
#Apply filtering on data
filtered_data = spark.sql("SELECT Employee, Department, Salary, Age
FROM employees WHERE age > 30")
# Display the results
filtered_data.show()
```

The output will be the following:

```
+--------+----------+------+---+
|Employee|Department|Salary|Age|
+--------+----------+------+---+
|    John| Field-eng|  3500| 40|
|  Robert|     Sales|  4000| 38|
|   Kelly|   Finance|  3500| 35|
|    Kate|   Finance|  3000| 45|
|   Kiran|     Sales|  2200| 35|
+--------+----------+------+---+
```

In this example, we create a temp view with the names of `employees` and execute an SQL query using Spark SQL to filter and select specific columns from the `employees` table.

The query selects the `employee`, `department`, `salary`, and `age` columns from the table where `age` is greater than 30. The results of the query are stored in the `filtered_data` variable.

Finally, we call the `show()` method to display the filtered data.

Exploring sorting and aggregation operations using Spark SQL

Spark SQL provides a rich set of transformation functions that can be applied to manipulate and transform data. Let's explore some of the practical examples of transformations in Spark SQL:

Aggregation

In this example, we perform an aggregation operation using Spark SQL to calculate the average salary from the `employees` table.

```
# Perform an aggregation to calculate the average salary
average_salary = spark.sql("SELECT AVG(Salary) AS average_salary FROM
employees")
# Display the average salary
average_salary.show()
```

The output will be the following:

```
+--------------+
|average_salary|
+--------------+
|        3275.0|
+--------------+
```

The `AVG()` function calculates the average of the `salary` column. We alias the result as `average_salary` using the AS keyword.

The results are stored in the `average_salary` variable and displayed using the `show()` method:

Sorting

In this example, we apply a sorting transformation to the `employees` table using Spark SQL.

```
# Sort the data based on the salary column in descending order
sorted_data = spark.sql("SELECT * FROM employees ORDER BY Salary
DESC")
# Display the sorted data
sorted_data.show()
```

The output will be the following:

```
+---+--------+----------+------+---+
| ID|Employee|Department|Salary|Age|
+---+--------+----------+------+---+
|  2|  Robert|     Sales|  4000| 38|
|  1|    John| Field-eng|  3500| 40|
|  5|   Kelly|   Finance|  3500| 35|
|  3|   Maria|   Finance|  3500| 28|
|  7|  Martin|   Finance|  3500| 26|
|  6|    Kate|   Finance|  3000| 45|
|  4| Michael|     Sales|  3000| 20|
|  8|   Kiran|     Sales|  2200| 35|
+---+--------+----------+------+---+
```

The `ORDER BY` clause is used to specify the sorting criteria, in this case, the `salary` column in descending order.

The sorted data are stored in the `sorted_data` variable and displayed using the `show()` method.

Combining aggregations

We can also combine different aggregations in one SQL command, such as in the following code example:

```
# Sort the data based on the salary column in descending order
filtered_data = spark.sql("SELECT Employee, Department, Salary, Age
FROM employees WHERE age > 30 AND Salary > 3000 ORDER BY Salary DESC")
# Display the results
filtered_data.show()
```

The output will be the following:

```
+--------+----------+------+---+
|Employee|Department|Salary|Age|
+--------+----------+------+---+
|  Robert|     Sales|  4000| 38|
|   Kelly|   Finance|  3500| 35|
|    John| Field-eng|  3500| 40|
+--------+----------+------+---+
```

In this example, we combine different transformations to the `employees` table using Spark SQL. First, we select those employees whose age is greater than 30 and who have a salary greater than 3,000. The `ORDER BY` clause is used to specify the sorting criteria; in this case, the `salary` column in descending order.

The resulting data are stored in the "`filtered_data`" variable and displayed using the `show()` method.

In this section, we explored the process of executing SQL queries and applying transformations using Spark SQL. We learned about the syntax for executing SQL queries and demonstrated practical examples of executing queries, filtering data, performing aggregations, and sorting data. By leveraging the expressive power of SQL and the flexibility of Spark SQL, you can efficiently analyze and manipulate structured data for a wide range of data analysis tasks.

Grouping and aggregating data – grouping data based on specific columns and performing aggregate functions

In Spark SQL, grouping and aggregating data are common operations that are performed to gain insights and summarize information from large datasets. This section will explore how to group data based on specific columns and perform various aggregate functions using Spark SQL. We will walk through code examples that demonstrate the capabilities of Spark SQL in this regard.

Grouping data

When we want to group data based on specific columns, we can utilize the `GROUP BY` clause in SQL queries. Let's consider an example where we have a DataFrame of employees with the columns `department` and `salary`. We want to calculate the average salary for each department:

```
# Group the data based on the Department column and take average
salary for each department
grouped_data = spark.sql("SELECT Department, avg(Salary) FROM
employees GROUP BY Department")
# Display the results
grouped_data.show()
```

The output will be the following:

```
+----------+------------------+
|Department|       avg(Salary)|
+----------+------------------+
| Field-eng|            3500.0|
|     Sales|3066.6666666666665|
|   Finance|            3375.0|
+----------+------------------+
```

In this example, we group the data based on different transformations to the `employees` table using Spark SQL. First, we group employees based on the `Department` column. We take the average salary of each department from the `employees` table.

The resulting data are stored in the `grouped_data` variable and displayed using the `show()` method.

Aggregating data

Spark SQL provides a wide range of aggregate functions to calculate summary statistics on grouped data. Let's consider another example where we want to calculate the total salary and the maximum salary for each department:

```
# Perform grouping and multiple aggregations
aggregated_data = spark.sql("SELECT Department, sum(Salary) AS total_
salary, max(Salary) AS max_salary FROM employees GROUP BY Department")

# Display the results
aggregated_data.show()
```

The output will be the following:

```
+----------+-----------+-----------+
|Department|sum(Salary)|max(Salary)|
+----------+-----------+-----------+
| Field-eng|       3500|       3500|
|     Sales|       9200|       4000|
|   Finance|      13500|       3500|
+----------+-----------+-----------+
```

In this example, we combine and group the data based on different transformations to the `employees` table using Spark SQL. First, we group the employees based on `Department` column. We take the total salary and maximum salary of each department from the employees table. We also use an alias for these aggregated columns.

The resulting data are stored in the `aggregated_data` variable and displayed using the `show()` method.

In this section, we have explored the capabilities of Spark SQL in grouping and aggregating data. We have seen examples of how to group data based on specific columns and perform various aggregate functions. Spark SQL provides a wide range of aggregate functions and allows for the creation of custom aggregate functions to suit specific requirements. With these capabilities, you can efficiently summarize and gain insights from large datasets using Spark SQL.

In the next section, we will look at advanced Spark SQL functions for complex data manipulation operations.

Advanced Spark SQL operations

Let's explore the key capabilities of Apache Spark's advanced operations.

Leveraging window functions to perform advanced analytical operations on DataFrames

In this section, we will explore the powerful capabilities of window functions in Spark SQL for performing advanced analytical operations on DataFrames. Window functions provide a way to perform calculations across a set of rows within a partition, allowing us to derive insights and perform complex computations efficiently. In this section, we will dive into the topic of window functions and showcase code examples that demonstrate their usage in Spark SQL queries.

Understanding window functions

Window functions in Spark SQL enable advanced analytical operations by dividing a dataset into groups or partitions based on specified criteria. These functions operate on a sliding window of rows within each partition, performing calculations or aggregations.

The general syntax for using window functions in Spark SQL is as follows:

```
function().over(Window.partitionBy("column1", "column2").
orderBy("column3").rowsBetween(start, end))
```

The `function()` represents the window function that you want to apply, such as `sum`, `avg`, `row_number`, or custom-defined functions. The `over()` clause defines the window to which the function is applied. `Window.partitionBy()` specifies the columns used to divide the dataset into partitions. It also determines the order of rows within each partition. `rowsBetween(start, end)` specifies the range of rows included in the window. It can be unbounded or defined relative to the current row.

Calculating cumulative sum using window functions

Let's explore a practical example that demonstrates the usage of window functions to calculate a cumulative sum of a column in a DataFrame:

```
from pyspark.sql.window import Window
from pyspark.sql.functions import col, sum

# Define the window specification
window_spec = Window.partitionBy("Department").orderBy("Age")

# Calculate the cumulative sum using window function
df_with_cumulative_sum = salary_data_with_id.withColumn("cumulative_
sum", sum(col("Salary")).over(window_spec))

# Display the result
df_with_cumulative_sum.show()
```

The output will be the following:

```
+---+--------+----------+------+---+--------------+
| ID|Employee|Department|Salary|Age|cumulative_sum|
+---+--------+----------+------+---+--------------+
|  1|    John| Field-eng|  3500| 40|          3500|
|  7|  Martin|   Finance|  3500| 26|          3500|
|  3|   Maria|   Finance|  3500| 28|          7000|
|  5|   Kelly|   Finance|  3500| 35|         10500|
|  6|    Kate|   Finance|  3000| 45|         13500|
|  4| Michael|     Sales|  3000| 20|          3000|
|  8|   Kiran|     Sales|  2200| 35|          5200|
|  2|  Robert|     Sales|  4000| 38|          9200|
+---+--------+----------+------+---+--------------+
```

In this example, we start by importing the necessary libraries. We use the same DataFrame as our previous examples: `salary_data_with_id`.

Next, we define a window specification using `Window.partitionBy("Department").orderBy("Age")`, which partitions the data according to the `Department` column and orders the rows within each partition according to the `Age` column.

We then use the `sum()` function as a window function, applied over the defined window specification, to calculate the cumulative sum of the `Salary` column. The result is stored in a new column called `cumulative_sum`.

Finally, we call the `show()` method to display the DataFrame with the added cumulative sum column. By leveraging window functions, we can efficiently calculate cumulative sums, running totals, rolling averages, and other complex analytical calculations over defined windows in Spark SQL.

In this section, we explored the powerful capabilities of window functions in Spark SQL for advanced analytics. We discussed the syntax and usage of window functions, allowing us to perform complex calculations and aggregations within defined partitions and windows. By incorporating window functions into Spark SQL queries, you can derive valuable insights and gain a deeper understanding of your data for advanced analytical operations.

In the next section, we will explore Spark user-defined functions.

User-defined functions

In this section, we will delve into the topic of **user-defined functions** (**UDFs**) in Spark SQL. UDFs allow us to extend the functionality of Spark SQL by defining our custom functions that can be applied to DataFrames or SQL queries. In this section, we will explore the concept of UDFs and provide code examples to demonstrate their usage and benefits in Spark SQL.

UDFs in Spark SQL enable us to create custom functions to perform transformations or computations on columns in a DataFrame or in SQL queries. UDFs are particularly useful when Spark's built-in functions do not meet our specific requirements.

To define a UDF in Spark SQL, we use the `udf()` function from the `pyspark.sql.functions` module. The general syntax is as follows:

```
from pyspark.sql.functions import udf
udf_name = udf(lambda_function, return_type)
```

First, we import the `udf()` function from the `pyspark.sql.functions` module. Next, we define the UDF by providing a lambda function or a regular Python function as the `lambda_function` argument. This function encapsulates the custom logic we want to apply.

We also specify `return_type` for the UDF, which represents the data type that the UDF will return.

Applying a UDF to a DataFrame

Let's explore a practical example that demonstrates the usage of UDFs in Spark SQL by applying a custom function to a DataFrame:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

# Define a UDF to capitalize a string
capitalize_udf = udf(lambda x: x.upper(), StringType())

# Apply the UDF to a column
df_with_capitalized_names = salary_data_with_
id.withColumn("capitalized_name", capitalize_udf("Employee"))

# Display the result
df_with_capitalized_names.show()
```

The output will be the following:

```
+---+--------+----------+------+---+----------------+
| ID|Employee|Department|Salary|Age|capitalized_name|
+---+--------+----------+------+---+----------------+
|  1|    John| Field-eng|  3500| 40|            JOHN|
|  2|  Robert|     Sales|  4000| 38|          ROBERT|
|  3|   Maria|   Finance|  3500| 28|           MARIA|
|  4| Michael|     Sales|  3000| 20|         MICHAEL|
|  5|   Kelly|   Finance|  3500| 35|           KELLY|
|  6|    Kate|   Finance|  3000| 45|            KATE|
|  7|  Martin|   Finance|  3500| 26|          MARTIN|
|  8|   Kiran|     Sales|  2200| 35|           KIRAN|
+---+--------+----------+------+---+----------------+
```

In this example, we start by defining a UDF called `capitalize_udf` using the `udf()` function. It applies a lambda function that changes the input string to upper case. We use the `withColumn()` method to apply the UDF `capitalize_udf` to the `name` column, creating a new column called `capitalized_name` in the resulting DataFrame.

Finally, we call the `show()` method to display the DataFrame with the transformed column.

UDFs allow us to apply custom logic and transformations to columns in DataFrames, enabling us to handle complex computations, perform string manipulations, or apply domain-specific operations that are not available in Spark's built-in functions.

In this section, we explored the concept of UDFs in Spark SQL. We discussed the syntax for defining UDFs and demonstrated their usage through a code example. UDFs provide a powerful mechanism to extend Spark SQL's functionality by allowing us to apply custom transformations and computations to DataFrames or SQL queries. By incorporating UDFs into your Spark SQL workflows, you can handle complex data operations and tailor your data processing pipelines to meet specific requirements or domain-specific needs.

Applying a function

PySpark also supports various UDFs and APIs to allow users to execute native Python functions. For instance, the following example allows users to directly use the APIs in a pandas series within the Python native function:

```
import pandas as pd
from pyspark.sql.functions import pandas_udf

@pandas_udf('long')
def pandas_plus_one(series: pd.Series) -> pd.Series:
    # Simply plus one by using pandas Series.
```

```
    return series + 1

salary_data_with_id.select(pandas_plus_one(salary_data_with_
id.Salary)).show()
```

The output will be the following:

```
+-----------------------+
|pandas_plus_one(Salary)|
+-----------------------+
|                   3501|
|                   4001|
|                   3501|
|                   3001|
|                   3501|
|                   3001|
|                   3501|
|                   2201|
+-----------------------+
```

In this example, we start by defining a pandas UDF called `pandas_plus_one` using the `@pandas_udf()` function. We define this function so that it adds 1 to a pandas series. We use the already created DataFrame named `salary_data_with_id` and call the pandas UDF to apply this function to the `salary` column of the DataFrame.

Finally, we call the `show()` method in the same statement, to display the DataFrame with the transformed column.

In addition, UDFs can be registered and invoked in SQL out of the box. The following is an example of how we can achieve this:

```
@pandas_udf("integer")
def add_one(s: pd.Series) -> pd.Series:
    return s + 1

spark.udf.register("add_one", add_one)
spark.sql("SELECT add_one(Salary) FROM employees").show()
```

The output will be the following:

```
+---------------+
|add_one(Salary)|
+---------------+
|           3501|
|           4001|
|           3501|
|           3001|
|           3501|
|           3001|
|           3501|
|           2201|
+---------------+
```

In this example, we start by defining a pandas UDF called `add_one` by using the `@pandas_udf()` function. We define this function so that it adds 1 to a pandas series. We then register this UDF for use in SQL functions. We use the already created employees table and call the pandas UDF to apply this function to the `salary` column of the table.

Finally, we call the `show()` method in the same statement to display the results.

In this section, we explored the powerful capabilities of UDFs and how we can use them in calculating aggregations.

In the next section, we will explore pivot and unpivot functions.

Working with complex data types – pivot and unpivot

Pivot and unpivot operations are used to transform data from a row-based format to a column-based format and vice versa. In Spark SQL, these operations can be performed using the pivot and unpivot functions.

The pivot function is used to transform rows into columns. It takes three arguments: the column to use as the new column headers, the column to use as the new row headers and the columns to use as the values in the new table. The resulting table will have one row for each unique value in the row header column, and one column for each unique value in the column header column.

The unpivot function is used to transform columns into rows. It takes two arguments: the columns to use as the new row headers, and the column to use as the values in the new table. The resulting table will have one row for each unique combination of values in the row header columns and one column for the values.

Some use cases for pivot and unpivot operations include the following:

- Converting data from a wide format to a long format or vice versa
- Aggregating data by multiple dimensions
- Creating summary tables or reports
- Preparing data for visualization or analysis

Overall, pivot and unpivot operations are useful tools for transforming data in Spark SQL.

Summary

In this chapter, we explored the process of transforming and analyzing data in Spark SQL. We learned how to filter and manipulate loaded data, save the transformed data as a table, and execute SQL queries to extract meaningful insights. By following the Python code examples provided, you can apply these techniques to your own datasets, unlocking the potential of Spark SQL for data analysis and exploration.

After covering those topics, we explored the powerful capabilities of window functions in Spark SQL for advanced analytics. We discussed the syntax and usage of window functions, allowing us to perform complex calculations and aggregations within defined partitions and windows. By incorporating window functions into Spark SQL queries, you can derive valuable insights and gain a deeper understanding of your data for advanced analytical operations.

We then discussed some ways to use UDFs in Spark and how they can be useful in complex aggregations over multiple rows and columns of a DataFrame.

Finally, we covered some of the ways to use pivot and unpivot in Spark SQL.

Sample questions

Question 1:

Which of the following code snippets creates a view in Spark SQL that will replace existing views if already present?

A. `dataFrame.createOrReplaceTempView()`
B. `dataFrame.createTempView()`
C. `dataFrame.createTableView()`
D. `dataFrame.createOrReplaceTableView()`
E. `dataDF.write.path(filePath)`

Question 2:

Which function do we use to join two DataFrames together?

A. `DataFrame.filter()`
B. `DataFrame.distinct()`
C. `DataFrame.intersect()`
D. `DataFrame.join()`
E. `DataFrame.count()`

Answers

1. A
2. D

669FD03CAE074

Part 4: Spark Applications

In this part, we will cover Spark's Structured Streaming, focusing on real-time data processing with concepts such as event time processing, watermarking, triggers, and output modes. Practical examples will illustrate building and deploying streaming applications using Structured Streaming. Additionally, we will delve into Spark ML, Spark's machine learning library, exploring supervised and unsupervised techniques, model building, evaluation, and hyperparameter tuning across various algorithms. Practical examples will demonstrate Spark ML's application in real-world machine learning tasks, crucial in contemporary data science. While not included in the Spark certification exam, understanding these concepts is essential in modern data engineering.

This part has the following chapters:

- *Chapter 7*, *Structured Streaming in Spark*
- *Chapter 8*, *Machine Learning with Spark ML*

7

Structured Streaming in Spark

The world of data processing has evolved rapidly as data volume and data velocity increase every day. With that, the need to analyze and derive insights from real-time data is becoming increasingly crucial. Structured Streaming, a component of Apache Spark, has emerged as a powerful framework to process and analyze data streams in real time. This chapter delves into the realm of Structured Streaming, exploring its capabilities, features, and real-world applications.

In this chapter, we will cover the following topics:

- Real-time data processing
- The fundamentals of streaming
- Streaming architectures
- Spark Streaming
- Structured Streaming
- Streaming sources and sinks
- Advanced topics in Structured Streaming
- Joins in Structured Streaming

By the end of this chapter, you will understand Spark Streaming and the power of real-time data insights.

We will start by looking at what real-time data processing means.

Real-time data processing

Real-time data processing has become increasingly critical in today's fast-paced and data-driven world. Organizations need to analyze and derive insights from data as it arrives, enabling them to make timely decisions and take immediate action. Spark Streaming, a powerful component of Apache Spark, addresses this need by providing a scalable and fault-tolerant framework to process real-time data streams.

Real-time data processing has gained immense importance in various industries, ranging from finance and e-commerce to the Internet of Things (IoT) and social media. Traditional batch processing approaches, while suitable for many scenarios, fall short when immediate insights and actions are required. Real-time data processing fills this gap by enabling the analysis and processing of data as it arrives, allowing organizations to make timely decisions and respond quickly to changing conditions.

Real-time data processing involves the continuous ingestion, processing, and analysis of streaming data. Unlike **batch processing**, which operates on static datasets, real-time data processing systems handle data that is generated and updated in real time. This data can be sourced from various channels, including sensors, logs, social media feeds, and financial transactions.

The key characteristics of real-time data processing are as follows:

- **Low latency**: Real-time data processing aims to minimize the time delay between data generation and processing. It requires fast and efficient processing capabilities to provide near-instantaneous insights and responses.
- **Scalability**: Real-time data processing systems must be able to handle high-volume and high-velocity data streams. The ability to scale horizontally and distribute processing across multiple nodes is essential to accommodate the increasing data load.
- **Fault tolerance**: Given the continuous nature of streaming data, real-time processing systems need to be resilient to failures. They should have mechanisms in place to recover from failures and ensure uninterrupted processing.
- **A streaming data model**: Real-time data processing systems operate on **streaming data**, which is an unbounded sequence of events or records. Streaming data models are designed to handle the continuous flow of data and provide mechanisms for event-time and window-based computations.

These characteristics of real-time data processing lead to several advantages, including the following:

- **A rapid response**: Real-time processing enables organizations to respond quickly to changing conditions, events, or opportunities. It allows for timely actions, such as fraud detection, anomaly detection, real-time monitoring, and alerting.
- **Personalization**: Real-time processing enables personalized experiences by analyzing and acting on user behavior in real time. It powers real-time recommendations, dynamic pricing, targeted advertising, and content personalization.
- **Operational efficiency**: Real-time processing provides insights into operational processes, allowing organizations to optimize their operations, identify bottlenecks, and improve efficiency in real time. It facilitates predictive maintenance, supply chain optimization, and real-time resource allocation.
- **Situational awareness**: Real-time data processing helps organizations gain situational awareness by continuously analyzing and aggregating data from various sources. It enables real-time analytics, monitoring, and decision making in domains such as cybersecurity, financial markets, and emergency response systems.

In summary, real-time streaming involves the continuous transmission and processing of data, enabling immediate insights and rapid decision-making. It has a wide range of applications across different industries and utilizes various technologies to facilitate efficient and reliable streaming.

In the next section, we will explore the basics of streaming and understand how streaming is useful for real-time operations.

What is streaming?

Streaming refers to the continuous and real-time processing of data as it is generated or received. Unlike batch processing, where data is processed in chunks or batches at fixed intervals, streaming enables the processing of data continuously and incrementally. It allows applications to ingest, process, and analyze data in real time, enabling timely decision-making and immediate responses to events.

Different types of streaming architectures are available to handle streaming data. We will look at them next.

Streaming architectures

Streaming architectures are designed to handle the continuous and high-velocity nature of streaming data. They typically consist of three key components:

- **Streaming sources**: These are the origins of the streaming data, such as IoT devices, sensors, logs, social media feeds, or messaging systems. Streaming sources continuously produce and emit data in real time.
- **A streaming processing engine**: The streaming processing engine is responsible for ingesting, processing, and analyzing streaming data. It provides the necessary infrastructure and computational capabilities to handle the continuous and incremental nature of streaming data.
- **Streaming sinks**: Streaming sinks are destinations where the processed data is stored, visualized, or acted upon. They can be databases, data warehouses, dashboards, or external systems that consume the processed data.

There are various streaming architectures, including the following:

- **Event-driven architecture**: In event-driven architecture, events are generated by sources and then captured and processed by the engine, leading to immediate reactions and triggering actions or updates in real time. This framework facilitates real-time event processing, supports the development of event-driven microservices, and contributes to the creation of reactive systems.

  Event-driven architecture's advantages lie in its ability to provide responsiveness, scalability, and flexibility. This allows for prompt reactions to events as they unfold, fostering agility in system responses.

- **Lambda architecture**: The lambda architecture seamlessly integrates batch and stream processing to effectively manage both historical and real-time data. This involves parallel processing of data streams to enable real-time analysis, coupled with offline batch processing for in-depth and comprehensive analytics.

  This approach is particularly well-suited for applications that require a balance of real-time insights and thorough historical analysis. The lambda architecture's strengths lie in its provision of fault tolerance, scalability, and capacity to handle substantial data volumes. This is achieved by harnessing the combined power of both the batch and stream processing techniques.

- **Unified streaming architecture**: Unified streaming architectures, such as Apache Spark's Structured Streaming, aim to provide a unified API and processing model for both batch and stream processing. They simplify the development and deployment of real-time applications by abstracting away the complexities of managing separate batch and stream processing systems.

  This architecture abstracts complexities by offering a simplified approach to developing and deploying real-time applications. This is ideal for scenarios where simplicity and ease of development are crucial, allowing developers to focus more on business logic than intricate technicalities.

  The advantages are that it simplifies development, reduces operational overhead, and ensures consistency across batch and stream processing.

These architectures cater to different needs based on the specific requirements of a given application. Event-driven is ideal for real-time reactions, lambda for a balance between real-time and historical data, and unified streaming for a streamlined, unified approach to both batch and stream processing. Each approach has its strengths and trade-offs, making them suitable for various scenarios based on the specific needs of a system.

In the following sections, we will delve into the specifics of Structured Streaming, its key concepts, and how it compares to Spark Streaming. We will also explore stateless and stateful streaming, streaming sources, and sinks, providing code examples and practical illustrations to enhance understanding.

Introducing Spark Streaming

As you've seen so far, Spark Streaming is a powerful real-time data processing framework built on Apache Spark. It extends the capabilities of the Spark engine to support high-throughput, fault-tolerant, and scalable stream processing. Spark Streaming enables developers to process real-time data streams using the same programming model as batch processing, making it easy to transition from batch to streaming workloads.

At its core, Spark Streaming divides the real-time data stream into small batches or micro-batches, which are then processed using Spark's distributed computing capabilities. Each micro-batch is treated as a **Resilient Distributed Dataset** (**RDD**), Spark's fundamental abstraction for distributed data processing. This approach allows developers to leverage Spark's extensive ecosystem of libraries, such as Spark SQL, MLlib, and GraphX, for real-time analytics and machine learning tasks.

Exploring the architecture of Spark Streaming

Spark Streaming follows a **master-worker architecture**, where the driver program serves as the master and the worker nodes process the data. The high-level architecture consists of the following components:

- **The driver program**: The driver program runs the main application and manages the overall execution of the Spark Streaming application. It divides the data stream into batches, schedules tasks on the worker nodes, and coordinates the processing.
- **Receivers**: Receivers are responsible for connecting to the streaming data sources and receiving the data. They run on worker nodes and pull the data from sources such as Kafka, Flume, or TCP sockets. The received data is then stored in the memory of the worker nodes.
- **Discretized Stream (DStream)**: DStream is the basic abstraction in Spark Streaming. It represents a continuous stream of data divided into small, discrete RDDs. DStream provides a high-level API to perform transformations and actions on the streaming data.
- **Transformations and actions**: Spark Streaming supports a wide range of transformations and actions, similar to batch processing. Transformations, such as `map`, `filter`, and `reduceByKey`, are applied to each RDD in the DStream. Actions, such as `count`, `saveAsTextFiles`, and `foreachRDD`, trigger the execution of the streaming computation and produce results.
- **Output operations**: Output operations allow the processed data to be written to external systems or storage. Spark Streaming supports various output operations, such as writing to files, databases, or sending to dashboards for visualization.

Key concepts

To effectively use Spark Streaming, it is important to understand some key concepts:

- **DStreams**: As mentioned earlier, DStreams represent the continuous stream of data in Spark Streaming. They are a sequence of RDDs, where each RDD contains data from a specific time interval. DStreams support various transformations and actions, enabling complex computations on the stream.
- **Window operations**: Window operations allow you to apply transformations on a sliding window of data in the stream. It enables computations over a fixed window size or based on time durations, enabling tasks such as windowed aggregations or time-based joins.
- **Stateful operations**: Spark Streaming gives you the ability to maintain stateful information across batches. It enables operations that require maintaining and updating state, such as cumulative counts.
- **Checkpointing**: Checkpointing is a crucial mechanism in Spark Streaming to ensure fault tolerance and recovery. It periodically saves the metadata about the streaming application, including the configuration, DStream operations, and the processed data. It enables the recovery of the application if there are failures.

Advantages

Now, we will see the different advantages of using Spark Streaming in real-time operations:

- **A unified processing model**: One of the significant advantages of Spark Streaming is its integration with the larger Spark ecosystem. It leverages the same programming model as batch processing, allowing users to seamlessly transition between batch and real-time processing. This unified processing model simplifies development and reduces the learning curve for users familiar with Spark.
- **High-level abstractions**: Spark Streaming provides high-level abstractions such as DStreams to represent streaming data. DStreams are designed to handle continuous data streams and enable easy integration with existing Spark APIs, libraries, and data sources. These abstractions provide a familiar and expressive programming interface to process real-time data.
- **Fault tolerance and scalability**: Spark Streaming offers fault-tolerant processing by leveraging Spark's RDD abstraction. It automatically recovers from failures by recomputing lost data, ensuring the processing pipeline remains resilient and robust. Additionally, Spark Streaming can scale horizontally by distributing a workload across a cluster of machines, allowing it to handle large-scale data streams effectively.
- **Windowed computations**: Spark Streaming supports windowed computations, which enable time-based analysis over sliding or tumbling windows of data. Window operations provide flexibility in performing aggregations, time-series analysis, and window-level transformations. This capability is particularly useful when analyzing streaming data based on temporal characteristics or patterns.
- **A wide range of data sources**: Spark Streaming seamlessly integrates with various data sources, including Kafka, Flume, Hadoop Distributed File System (HDFS), and Amazon S3. This broad range of data sources allows users to ingest data from multiple streams and integrate it with existing data pipelines. Spark Streaming also supports custom data sources, enabling integration with proprietary or specialized streaming platforms.

While Spark Streaming offers powerful real-time data processing capabilities, there are certain challenges to consider.

Challenges

Building streaming architectures for applications comes with its set of challenges, such as the following:

- **End-to-end latency**: Latency is introduced when Spark Streaming processes data in micro-batches. The end-to-end latency can vary based on factors such as the batch interval, the data source, and the complexity of the computations.
- **Fault tolerance**: Spark Streaming provides fault tolerance through RDD lineage and checkpointing. However, failures in receivers or the driver program can still disrupt the stream processing.

Handling and recovering from failures is an important consideration to ensure the reliability of Spark Streaming applications.

- **Scalability**: Scaling Spark Streaming applications to handle large volumes of data and meet high throughput requirements can be a challenge. Proper resource allocation, tuning, and cluster management are crucial to achieve scalability.
- **Data ordering**: Spark Streaming processes data in parallel across multiple worker nodes, which can affect the order of events. Ensuring the correctness of the order of events becomes important in certain use cases, and developers need to consider this when designing their applications.

In summary, Spark Streaming brings the power of Apache Spark to real-time data processing. Its integration with the Spark ecosystem, high-level abstractions, fault tolerance, scalability, and support for windowed computations make it a compelling choice for processing streaming data. By harnessing the advantages of Spark Streaming, organizations can unlock valuable insights and make informed decisions in real time.

In the next section, we will explore Structured Streaming, a newer and more expressive streaming API in Apache Spark that overcomes some of the limitations and challenges of Spark Streaming. We will discuss its core concepts, the differences from Spark Streaming, and its benefits for real-time data processing.

Introducing Structured Streaming

Structured Streaming is a revolutionary addition to Apache Spark that brings a new paradigm for real-time data processing. It introduces a high-level API that seamlessly integrates batch and streaming processing, providing a unified programming model. Structured Streaming treats streaming data as an unbounded table or DataFrame, enabling developers to express complex computations using familiar SQL-like queries and transformations.

Unlike the micro-batch processing model of Spark Streaming, Structured Streaming follows a continuous processing model. It processes data incrementally as it arrives, providing low-latency and near-real-time results. This shift toward continuous processing opens up new possibilities for interactive analytics, dynamic visualizations, and real-time decision-making.

Key features and advantages

Structured Streaming offers several key features and advantages over traditional stream processing frameworks:

- **An expressive API**: Structured Streaming provides a declarative API that allows developers to express complex streaming computations using SQL queries, DataFrame operations, and Spark SQL functions. This enables developers with SQL or DataFrame expertise to easily transition to real-time data processing.

- **Fault tolerance and exactly-once semantics**: Structured Streaming guarantees end-to-end fault tolerance and exactly-once semantics by maintaining the necessary metadata and state information. It handles failures gracefully and ensures that data is processed exactly once, even in the presence of failures or retries.
- **Scalability**: Structured Streaming leverages the scalability of the Spark engine, enabling horizontal scaling by adding more worker nodes to the cluster. It can handle high-throughput data streams and scale seamlessly as the data volume increases.
- **Unified batch and streaming**: With Structured Streaming, developers can use the same API and programming model for both batch and streaming processing. This unification simplifies the development and maintenance of applications, as there is no need to manage separate batch and stream processing code bases.
- **Ecosystem integration**: Structured Streaming seamlessly integrates with the broader Spark ecosystem, enabling the use of libraries such as Spark SQL, MLlib, and GraphX for real-time analytics, machine learning, and graph processing on streaming data.

Now, let's take a look at some of the differences between Structured Streaming and Spark Streaming.

Structured Streaming versus Spark Streaming

Structured Streaming differs from Spark Streaming in several fundamental ways:

- **The processing model**: Spark Streaming processes data in micro-batches, where each batch is treated as a discrete RDD. In contrast, Structured Streaming processes data incrementally in a continuous manner, treating the stream as an unbounded table or DataFrame.
- **The API and query language**: Spark Streaming primarily offers a low-level API based on RDD transformations and actions. Structured Streaming, on the other hand, provides a higher-level API with SQL-like queries, DataFrame operations, and Spark SQL functions. This makes it easier to express complex computations and leverage the power of SQL for real-time analytics.
- **Fault tolerance**: Both Spark Streaming and Structured Streaming provide fault tolerance. However, Structured Streaming's fault tolerance is achieved by maintaining the necessary metadata and state information, whereas Spark Streaming relies on RDD lineage and checkpointing for fault recovery.
- **Data processing guarantees**: Spark Streaming provides at-least-once processing guarantees by default, where some duplicates may be processed if there are failures. Structured Streaming, on the other hand, provides exactly-once processing semantics, ensuring that each event is processed exactly once, even in the presence of failures or retries.

Limitations and considerations

While Structured Streaming offers significant advantages, there are certain limitations and considerations to keep in mind:

- **Event time handling**: Proper handling of event time, such as timestamp extraction, watermarking, and late data handling, is essential in Structured Streaming. Care should be taken to ensure the correct processing and handling of out-of-order events.
- **State management**: Structured Streaming allows you to maintain stateful information across batches, which can introduce challenges related to state management and scalability. Monitoring memory usage and configuring appropriate state retention policies are crucial for optimal performance.
- **Ecosystem compatibility**: While Structured Streaming integrates well with the Spark ecosystem, certain libraries and features might not be fully compatible with real-time streaming use cases. It is important to evaluate the compatibility of specific libraries and functionalities before using them in a Structured Streaming application.
- **Performance considerations**: Structured Streaming's continuous processing model introduces different performance considerations compared to micro-batch processing. Factors such as the event rate, processing time, and resource allocation need to be carefully monitored and optimized for efficient real-time data processing.

In the next section, we will delve deeper into the concepts of stateless and stateful streaming, exploring their differences and use cases in the context of Structured Streaming.

Streaming fundamentals

Let's start by looking at some of the fundamental concepts in streaming that will help us get familiarized with different paradigms in streaming.

Stateless streaming – processing one event at a time

Stateless streaming refers to the processing of each event in isolation, without considering any context or history from previous events. In this approach, each event is treated independently, and the processing logic does not rely on any accumulated state or information from past events.

Stateless streaming is well-suited for scenarios where each event can be processed independently, and the output is solely determined by the content of the event itself. This approach is often used for simple filtering, transformation, or enrichment operations that do not require you to maintain any contextual information across events.

Stateful streaming – maintaining stateful information

Stateful streaming involves maintaining and utilizing contextual information or state across multiple events during processing. The processing logic considers the history of events and uses accumulated information to make decisions or perform computations. Stateful streaming enables more sophisticated analysis and complex computations that rely on context or accumulated knowledge.

Stateful streaming requires you to maintain and update state information as new events arrive. The state can be as simple as a running count or more complex, involving aggregations, windowed computations, or maintaining session information. Proper management of state is essential to ensure correctness, scalability, and fault tolerance in stateful streaming applications.

Let's understand the differences between stateless and stateful streaming.

The differences between stateless and stateful streaming

The main differences between stateless and stateful streaming can be summarized as follows:

- Stateless streaming processes events independently, while stateful streaming maintains and uses accumulated state information across events.
- Stateless streaming is suitable for simple operations that don't rely on past events, while stateful streaming enables complex computations that require context or accumulated knowledge.
- Stateless streaming is generally simpler to implement and reason about, while stateful streaming introduces additional challenges in managing state, fault tolerance, and scalability.
- Stateless streaming is often used for real-time filtering, transformation, or basic aggregations, while stateful streaming is necessary for windowed computations, sessionization, and stateful joins.

Understanding the distinction between stateless and stateful streaming is crucial when designing real-time data processing systems, as it helps determine the appropriate processing model and requirements for a given use case.

Now, let's take a look at some of the fundamentals of Structured Streaming.

Structured Streaming concepts

To understand Structured Streaming, it's important for us to understand the different operations that take place in a near-real-time scenario when data arrives. We will understand them in the following section.

Event time and processing time

In Structured Streaming, there are two important notions of time – event time and processing time:

- **Event time**: Event time refers to the time when an event occurred or was generated. It is typically embedded within the data itself, representing the timestamp or a field indicating when the event

occurred in the real world. Event time is crucial for analyzing data based on its temporal order or performing window-based computations.

- **Processing time**: Processing time, on the other hand, refers to the time when an event is processed by the streaming application. It is determined by the system clock or the time at which the event is ingested by the processing engine. Processing time is useful for tasks that require low latency or an immediate response but may not accurately reflect the actual event order.

Based on these different time concepts, we can determine which one works best for a given use case. It's important to understand the difference between the two. Based on that, the strategy for data processing can be determined.

Watermarking and late data handling

Now, we will discuss how to handle data that doesn't arrive at the defined time in real-time applications. There are different ways to handle that situation. Structured Streaming has a built-in mechanism to handle this type of data. These mechanisms include:

- **Watermarking**: Watermarking is a mechanism in Structured Streaming used to deal with event time and handle delayed or late-arriving data. A watermark is a threshold timestamp that indicates the maximum event time seen by a system up to a certain point. It allows the system to track the progress of event time and determine when it is safe to emit results for a specific window.
- **Late data handling**: Late-arriving data refers to events that have timestamps beyond the watermark threshold. Structured Streaming provides options to handle late data, such as discarding it, updating existing results, or storing it separately for further analysis.

These built-in mechanisms save users a lot of time and efficiently handle late-arriving data.

Next, we will see, once the data arrives, how we start the operations on it in streaming.

Triggers and output modes

Triggers determine when a streaming application should emit results or trigger the execution of the computation. Structured Streaming supports different types of triggers:

- **Event time triggers**: Event time triggers operate based on the arrival of new events or when a watermark advances beyond a certain threshold. They enable more accurate and efficient processing, based on event time semantics.
- **Processing time triggers**: These triggers operate based on processing time, allowing you to specify time intervals or durations at which the computation should be executed.

Structured Streaming also offers different output modes. The output modes determine how data is updated in the sink. A sink is where we would write the output after the streaming operation:

- **Complete mode**: In this mode, the entire updated result, including all the rows in the output, is written to the sink. This mode provides the most comprehensive view of data but can be memory-intensive for large result sets.
- **Append mode**: In append mode, only the new rows appended to the result table since the last trigger are written to the sink. This mode is suitable for cases where the result is an append-only stream.
- **Update mode**: Update mode only writes the changed rows to the sink, preserving the existing rows that haven't changed since the last trigger. This mode is useful for cases where the result table is updated incrementally.

Now, let's take a look at the different types of aggregate operations we can do on streaming data.

Windowing operations

Windowing operations in Structured Streaming allow you to group and aggregate data over specific time windows. Windows for these operations can be defined based on either event time or processing time, and they provide a way to perform computations over a subset of events within a given time range.

The common types of windowing operations include the following:

- **Tumbling windows**: Tumbling windows divide a stream into non-overlapping fixed-size windows. Each event falls into exactly one window, and computations are performed independently for each window.
- **Sliding windows**: Sliding windows create overlapping windows that slide or move over a stream at regular intervals. Each event can contribute to multiple windows, and computations can be performed on the overlapping parts.
- **Session windows**: Session windows group events that are close in time or belong to the same session, based on a specified session timeout. A session is defined as a series of events within a certain time threshold of each other.

The next operation that we frequently use in streaming is the join operation. Now, we will see how we can use joins with streaming data.

Joins and aggregations

Structured Streaming supports joins and aggregations on streaming data, enabling complex analytics and data transformations:

- **Joins**: Streaming joins allow you to combine two or more streams or a stream with static/reference data, based on a common key or condition. The join operation can be performed

using event time or processing time, and it supports different join types such as inner join, outer join, and left/right join.

- **Aggregations**: Streaming aggregations enable you to calculate summary statistics or perform aggregations on streaming data. Aggregations can be performed on individual streams or in combination with joins. Common operations include `count`, `sum`, `average`, `min`, and `max`.

Structured Streaming's flexible and expressive API for handling event time, triggers, output modes, windowing operations, joins, and aggregations allows developers to perform comprehensive real-time analytics and computations on streaming data. By understanding these concepts, developers can build sophisticated streaming applications with ease and precision.

In the next section, we will explore how to read and write data with streaming sources and sinks.

Streaming sources and sinks

Streaming sources and sinks are essential components in a streaming system that enable the ingestion of data from external systems and the output of processed data to external destinations. They form the connectors between the streaming application and the data sources or sinks.

Streaming sources retrieve data from various input systems, such as message queues, filesystems, databases, or external APIs, and make it available for processing in a streaming application. On the other hand, streaming sinks receive processed data from the application and write it to external storage, databases, filesystems, or other systems for further analysis or consumption.

There are different types of streaming sources and sinks. We will explore some of them next.

Built-in streaming sources

Structured Streaming provides built-in support for a variety of streaming sources, making it easy to integrate with popular data systems. Some of the commonly used built-in streaming sources include the following:

- **The file source**: The file source allows you to read data from files in a directory or a file stream from a file-based system, such as HDFS or Amazon S3.
- **KafkaSource**: KafkaSource enables the consumption of data from Apache Kafka, a distributed streaming platform. It provides fault-tolerant, scalable, and high-throughput ingestion of data streams.
- **The socket source**: The socket source allows a streaming application to read data from a **Transmission Control Protocol** (**TCP**) socket. It is useful for scenarios where data is sent through network connections, such as log streaming or data sent by external systems.

- **The Structured Streaming source**: The Structured Streaming source allows developers to define their own streaming sources by extending the built-in source interfaces. It provides the flexibility to integrate with custom or proprietary data sources.

Custom streaming sources

In addition to the built-in streaming sources, Structured Streaming allows developers to create custom streaming sources to ingest data from any system that can be accessed programmatically. Custom streaming sources can be implemented by extending the `Source` interface provided by the Structured Streaming API.

When implementing a custom streaming source, developers need to consider aspects such as data ingestion, event-time management, fault tolerance, and scalability. They must define how data is fetched, how it is partitioned and distributed among workers, and how to handle late-arriving data and schema evolution.

Similar to different streaming sources, we have streaming sinks as well. Let's explore them next.

Built-in streaming sinks

Structured Streaming provides built-in support for various streaming sinks, enabling the output of processed data to different systems. Some of the commonly used built-in streaming sinks include the following:

- **The console sink**: The console sink writes the output data to the console or standard output. It is useful for debugging and quick prototyping but not suitable for production use.
- **The file sink**: The file sink writes the output data to files in a directory or a file-based system such as HDFS or Amazon S3. It allows data to be stored and consumed later for batch processing or archival purposes.
- **The Kafka sink**: The Kafka sink enables you to write data to Apache Kafka topics. It provides fault-tolerant, scalable, and high-throughput output to Kafka for consumption by other systems.
- **The foreach sink**: The foreach sink allows developers to define their own custom sink logic by implementing the `ForeachWriter` interface. It provides the flexibility to write data to external systems or perform custom operations on the output data.

Custom streaming sinks

Similar to custom streaming sources, developers can implement custom streaming sinks in Structured Streaming by extending the `Sink` interface. There are instances when you would need to write the data back to a system that might not support streaming. It could be a database or a file-based storage system. Custom streaming sinks enable integration with external systems or databases that are not supported by the built-in sinks.

When implementing a custom streaming sink, developers need to define how output data is written or processed by the external system. This may involve establishing connections, handling batching or buffering, and ensuring fault tolerance and exactly-once semantics.

In the next section, we will talk about advanced techniques in Structured Streaming.

Advanced techniques in Structured Streaming

There are certain built-in capabilities of Structured Streaming that makes it the default choice for even some batch operations. Instead of architecting things yourself, Structured Streaming handles these properties for you. Some of them are as follows.

Handling fault tolerance

Fault tolerance is crucial in streaming systems to ensure data integrity and reliability. Structured Streaming provides built-in fault tolerance mechanisms to handle failures in both streaming sources and sinks:

- **Source fault tolerance**: Structured Streaming ensures end-to-end fault tolerance in sources, by tracking the progress of event time using watermarks and checkpointing the metadata related to the stream. If there are failures, the system can recover and resume processing from the last consistent state.
- **Sink fault tolerance**: Fault tolerance in sinks depends on the guarantees provided by the specific sink implementation. Some sinks may inherently provide exactly-once semantics, while others may rely on idempotent writes or deduplication techniques to achieve at-least-once semantics. Sink implementations should be carefully chosen to ensure data consistency and reliability.

Developers should consider the fault tolerance characteristics of the streaming sources and sinks they use and configure appropriate checkpointing intervals, retention policies, and recovery mechanisms to ensure the reliability of their streaming applications.

Structured Streaming has built-in support for schema evolution as well. Let's explore that in the next section.

Handling schema evolution

Structured Streaming provides support for handling schema evolution in streaming data sources. Schema evolution refers to changes in the structure or schema of incoming data over time.

Structured Streaming can handle schema evolution by applying the concept of schema inference or schema merging. When reading from streaming sources, the initial schema is inferred from the incoming data. As the data evolves, subsequent DataFrames are merged with the initial schema, accommodating any new or changed fields.

The following code snippet demonstrates handling schema evolution in Structured Streaming:

```
stream = spark.readStream \
  .format("csv") \
  .option("header", "true") \
  .schema(initialSchema) \
  .load("data/input")

mergedStream = stream \
  .selectExpr("col1", "col2", "new_col AS col3")
```

In this example, the initial schema is provided explicitly using the `schema` method. As new data arrives with additional fields, such as `new_col`, it can be selected and merged into the stream using the `selectExpr` method.

Handling schema evolution is crucial to ensure compatibility and flexibility in streaming applications where the data schema may change or evolve over time.

Different joins in Structured Streaming

One of the key features of Structured Streaming is its ability to join different types of data streams together in one sink.

Stream-stream joins

Stream-stream joins, also known as **stream-stream co-grouping** or **stream-stream correlation**, involve joining two or more streaming data sources based on a common key or condition. In this type of join, each incoming event from the streams is matched with events from other streams that share the same key or satisfy the specified condition.

Stream-stream joins enable real-time data correlation and enrichment, making it possible to combine multiple streams of data to gain deeper insights and perform complex analytics. However, stream-stream joins present unique challenges compared to batch or stream-static joins, due to the unbounded nature of streaming data and potential event-time skew.

One common approach to stream-stream joins is the use of windowing operations. By defining overlapping or tumbling windows on the streams, events within the same window can be joined based on their keys. Careful consideration of window size, watermarking, and event time characteristics is necessary to ensure accurate and meaningful joins.

Here's an example of a stream-stream join using Structured Streaming:

```
stream1 = spark.readStream.format("kafka")
.option("kafka.bootstrap.servers", "localhost:9092")
.option("subscribe", "topic1") .load()
```

```
stream2 = spark.readStream.format("kafka").option("kafka.bootstrap.
servers","localhost:9092") .option("subscribe", "topic2") .load()

 joinedStream =stream1.join(stream2, "common_key")
```

In this example, two Kafka streams, `stream1` and `stream2`, are read from different topics. The `join` method is then applied to perform the join operation, based on the `common_key` field shared by both streams.

Stream-static joins

Stream-static joins, also known as **stream-batch joins**, involve joining a streaming data source to a static or reference dataset. The static dataset typically represents reference data, such as configuration data or dimension tables, that remains constant over time.

Stream-static joins are useful for enriching streaming data with additional information or attributes from the static dataset. For example, you might join a stream of user activity events with a static user profile table to enrich each event with user-related details.

To perform a stream-static join in Structured Streaming, you can load the static dataset as a static DataFrame and then use the join method to perform the join with the streaming DataFrame. Since the static dataset does not change, the join operation can be performed using the default "right outer join" mode.

Here's an example of a stream-static join in Structured Streaming:

```
stream =spark.readStream.format("kafka")
.option("kafka.bootstrap.servers", "localhost:9092")
.option("subscribe", "topic") .load()

  staticData = spark.read.format("csv") .option("header", "true")
.load("data/static_data.csv")

 enrichedStream = stream.join(staticData,"common_key")
```

In this example, the streaming data is read from a Kafka source, and the static dataset is loaded from a CSV file. The join method is then used to perform the stream-static join based on the "`common_key`" field.

Both stream-stream and stream-static joins provide powerful capabilities for real-time data analysis and enrichment. When using these join operations, it is essential to carefully manage event time characteristics, windowing options, and data consistency to ensure accurate and reliable results. Additionally, performance considerations should be considered to handle large volumes of data and meet low-latency requirements in real-time streaming applications.

Final thoughts and future developments

Structured Streaming has emerged as a powerful framework for real-time data processing in Apache Spark. Its unified programming model, fault tolerance, and seamless integration with the Spark ecosystem make it an attractive choice for building scalable and robust streaming applications.

As Structured Streaming continues to evolve, there are several areas that hold promise for future developments. These include the following:

- **Enhanced support for streaming sources and sinks**: Providing more built-in connectors for popular streaming systems and databases, as well as improving the integration and compatibility with custom sources and sinks.
- **Advanced event time handling**: Introducing more advanced features for event time handling, including support for event-time skew detection and handling, event deduplication, and watermark optimizations.
- **Performance optimization**: Continuously improving the performance of Structured Streaming, especially in scenarios with high data volumes and complex computations. This could involve optimizations in memory management, query planning, and query optimization techniques.
- **Integration with AI and machine learning**: Further integrating Structured Streaming with AI and machine learning libraries in Spark, such as MLlib and TensorFlow, to enable real-time machine learning and predictive analytics on streaming data.
- **Seamless integration with streaming data warehouses**: Providing better integration with streaming data warehouses or data lakes, such as Apache Iceberg or Delta Lake, to enable scalable and efficient storage and the querying of streaming data.

In conclusion, Structured Streaming offers a modern and expressive approach to real-time data processing in Apache Spark. Its ease of use, scalability, fault tolerance, and integration with the Spark ecosystem make it a valuable tool for building robust and scalable streaming applications. By leveraging the concepts and techniques covered in this chapter, developers can unlock the full potential of real-time data processing with Structured Streaming.

Summary

Throughout this chapter, we have explored the fundamental concepts and advanced techniques in Structured Streaming.

We started by understanding the fundamentals of Structured Streaming, its advantages, and the core concepts that underpin its operation. Then, we talked about Spark Streaming and what it has to offer.

After that, we dived into the core functionalities of Structured Streaming. Then, we further delved into advanced topics, such as windowed operations in Structured Streaming. We explored sliding and tumbling windows, which enable us to perform aggregations and computations over a specified time window, allowing for time-based analysis of the streaming data. Additionally, we explored stateful streaming processing, which involves maintaining and updating state in streaming applications and integrating external libraries and APIs to enhance the capabilities of Structured Streaming.

Finally, we explored emerging trends in real-time data processing and concluded the chapter by summarizing the key takeaways and insights gained.

In the next chapter, we will look into machine learning techniques and how to use Spark with machine learning.

669FD03CAE074

8

Machine Learning with Spark ML

Machine learning has gained popularity in recent times. In this chapter, we will do a comprehensive exploration of Spark **Machine Learning** (**ML**), a powerful framework for scalable ML on Apache Spark. We will delve into the foundational concepts of ML and how Spark ML leverages these principles to enable efficient and scalable data-driven insights.

We will cover the following topics:

- Key concepts in ML
- Different types of ML
- ML with Spark
- Considering the ML life cycle with the help of a real-world example
- Different case studies for ML
- Future trends in Spark ML and distributed ML

ML encompasses diverse methodologies tailored to different data scenarios. We will start by learning about different key concepts in ML.

Introduction to ML

ML is a field of study that focuses on the development of algorithms and models that enable computer systems to learn and make predictions or decisions without being explicitly programmed. It is a subset of **artificial intelligence** (**AI**) that aims to provide systems with the ability to automatically learn and improve from data and experience.

In today's world, where vast amounts of data are being generated at an unprecedented rate, ML plays a critical role in extracting meaningful insights, making accurate predictions, and automating decision-making processes. As data grows, machines can learn the patterns better, thus making it even easier to gain insights from this data. It finds applications in various domains, including finance, healthcare, marketing, image and speech recognition, recommendation systems, and many more.

The key concepts of ML

To understand ML, it is important to grasp the fundamental concepts that underpin its methodology.

Data

Data is the foundation of any ML process. It can be structured, semi-structured, or unstructured and encompasses various types, such as numerical, categorical, text, images, and more. ML algorithms require high-quality, relevant, and representative data to learn patterns and make accurate predictions. When dealing with ML problems, it is imperative to have data that can answer the question that we're trying to solve. The quality of data used in any analysis or model-building process significantly impacts the outcomes and decisions derived from it. Bad or poor-quality data can lead to inaccurate, unreliable, or misleading results, ultimately affecting the overall performance and credibility of any analysis or model.

An ML model trained on bad data is likely to make inaccurate predictions or classifications. For instance, a model trained on incomplete or biased data might incorrectly identify loyal customers as potential churners or vice versa.

Decision-makers relying on flawed or biased analysis derived from bad data might implement strategies based on inaccurate insights. For instance, marketing campaigns targeting the wrong customer segments due to flawed churn predictions can lead to wasted resources and missed opportunities.

Therefore, we need to make sure that the data we're using for ML problems is representative of the population that we want to build the models for. The other thing to note is that data might have some inherent biases in it. It is our responsibility to look for them and be aware of them when using this data to build ML models.

Features

Features are the measurable properties or characteristics of the data that the ML algorithm uses to make predictions or decisions. They are the variables or attributes that capture the relevant information from the data. Out of the vast amounts of data that are present, we want to understand which features of this data would be useful for solving a particular problem. Relevant features would generate better models.

Feature engineering, the process of selecting, extracting, and transforming features, plays a crucial role in improving the performance of ML models.

Labels and targets

Labels or targets are the desired outputs or outcomes that the ML model aims to predict or classify. In supervised learning, where the model learns from labeled data, the labels represent the correct answers or class labels associated with the input data. In unsupervised learning, the model identifies patterns or clusters in the data without any explicit labels.

Training and testing

In ML, models are trained using a subset of the available data, which is called the **training set**. The training process involves feeding the input data and corresponding labels to the model, which learns from this data to make predictions. Once the model is trained, its performance is evaluated using a separate subset of data called the testing set. This evaluation helps assess the model's ability to generalize and make accurate predictions on unseen data.

Algorithms and models

ML algorithms are mathematical or statistical procedures that learn patterns and relationships in the data and make predictions or decisions. They can be categorized into various types, including regression, classification, clustering, dimensionality reduction, and reinforcement learning. These algorithms, when trained on data, generate models that capture the learned patterns and can be used to make predictions on new, unseen data.

Discussing different ML algorithms in depth is beyond the scope of this book. We will talk about different types of ML problems in the next section.

Types of ML

ML problems can be broadly categorized into two distinct categories. In this section, we will explore both of them.

Supervised learning

Supervised learning is a type of ML where the algorithm learns from labeled training data to make predictions or decisions. In supervised learning, the training data consists of input features and corresponding output labels or target values. The goal is to learn a mapping function that can accurately predict the output for new, unseen inputs.

The process of supervised learning involves the following steps:

1. **Data preparation**: The first step is to collect and preprocess the training data. This includes cleaning the data, handling missing values, and transforming the data into a suitable format for the learning algorithm. The data should be split into features (input variables) and labels (output variables).

2. **Model training**: Once the data has been prepared, the supervised learning algorithm is trained on the labeled training data. The algorithm learns the patterns and relationships between the input features and the corresponding output labels. The goal is to find a model that can generalize well to unseen data and make accurate predictions.
3. **Model evaluation**: After training the model, it needs to be evaluated to assess its performance. This is done using a separate set of data called the testing or validation set. The model's predictions are compared with the actual labels in the testing set, and various evaluation metrics such as accuracy, precision, recall, or mean squared error are calculated.
4. **Model deployment and prediction**: Once the model is trained and evaluated, it can be deployed to make predictions on new, unseen data. The trained model takes the input features of the new data and produces predictions or decisions based on what it has learned during the training phase.

Examples of supervised learning algorithms include linear regression, logistic regression, **support vector machines** (**SVM**), decision trees, random forests, gradient boosting, and neural networks. Again, going in-depth on these algorithms is beyond the scope of this book. You can read more about them here: `https://spark.apache.org/docs/latest/ml-classification-regression.html`.

Unsupervised Learning

Unsupervised learning is a type of ML where the algorithm learns patterns and relationships in the data without any labeled output. In unsupervised learning, the training data consists only of input features, and the goal is to discover hidden patterns, structures, or clusters within the data.

The process of unsupervised learning involves the following steps:

1. **Data preparation**: Similar to supervised learning, the first step is to collect and preprocess the data. However, in unsupervised learning, there are no labeled output values or target variables. The data is transformed and prepared in a way that it's suitable for the specific unsupervised learning algorithm.
2. **Model training**: In unsupervised learning, the algorithm is trained on the input features without any specific target variable. The algorithm explores the data and identifies patterns or clusters based on statistical properties or similarity measures. The goal is to extract meaningful information from the data without any predefined labels.
3. **Model evaluation** (optional): Unlike supervised learning, unsupervised learning does not have a direct evaluation metric based on known labels. Evaluation in unsupervised learning is often subjective and depends on the specific task or problem domain. It is also a more manual process than it is in supervised learning. Evaluation can involve visualizing the discovered clusters, assessing the quality of dimensionality reduction, or using domain knowledge to validate the results.

4. **Pattern discovery and insights**: The primary objective of unsupervised learning is to discover hidden patterns, structures, or clusters in the data. Unsupervised learning algorithms can reveal insights about the data, identify anomalies or outliers, perform dimensionality reduction, or generate recommendations.

Examples of unsupervised learning algorithms include K-means clustering, hierarchical clustering, **principal component analysis** (**PCA**), association rule mining, and **self-organizing maps** (**SOM**).

In conclusion, supervised learning and unsupervised learning are two key types of ML. Supervised learning relies on labeled data to learn patterns and make predictions, while unsupervised learning discovers patterns and structures in unlabeled data. Both types have their own set of algorithms and techniques, as well as different choices. Discussing unsupervised learning in depth is beyond the scope of this book.

In the next section, we will explore supervised ML, a cornerstone in the realm of AI and data science that represents a powerful approach to building predictive models and making data-driven decisions.

Types of supervised learning

As we know, supervised learning is a branch of ML where algorithms learn patterns and relationships from labeled training data. It involves teaching or supervising the model by presenting input data along with corresponding output labels, allowing the algorithm to learn the mapping between the input and output variables. We'll explore three key types of supervised learning – classification, regression, and time series.

Classification

Classification is a type of ML task where the goal is to categorize or classify data into predefined classes or categories based on its features. The algorithm learns from labeled training data to build a model that can predict the class label of new, unseen data instances.

In classification, the output is discrete and represents class labels. Some common algorithms that are used for classification tasks include logistic regression, decision trees, random forests, SVM, and naive Bayes.

For example, consider a spam email classification task, where the goal is to predict whether an incoming email is spam or not. The algorithm is trained on a dataset of labeled emails, where each email is associated with a class label indicating whether it is spam or not. The trained model can then classify new emails as spam or non-spam based on their features, such as the content, subject, or sender.

Regression

Regression is another type of ML task that focuses on predicting continuous or numerical values based on input features. In regression, the algorithm learns from labeled training data to build a model that can estimate or forecast the numerical value of a target variable given a set of input features.

Regression models are used when the output is a continuous value, such as predicting house prices, stock market trends, or predicting the sales of a product based on historical data. Some commonly used regression algorithms include linear regression, decision trees, random forests, gradient boosting, and neural networks.

For example, consider a case where you want to predict the price of a house based on its various features, such as the area, number of bedrooms, location, and so on. In this case, the algorithm is trained on a dataset of labeled house data, where each house is associated with its corresponding price. The trained regression model can then predict the price of a new house based on its features.

Time series

Time series analysis is a specialized area of ML that deals with data collected over time, where the order of observations is important. In time series analysis, the goal is to understand and forecast the patterns, trends, and dependencies within the data.

Time series models are used to predict future values based on historical data points. They are widely used in fields such as finance, stock market prediction, weather forecasting, and demand forecasting. Some popular time series algorithms include **Autoregressive Integrated Moving Average** (**ARIMA**), exponential smoothing methods, and **long short-term memory** (**LSTM**) networks.

For example, suppose you have historical stock market data for a particular company, including the date and the corresponding stock prices. The time series algorithm can analyze the patterns and trends in the data and make predictions about future stock prices based on historical price fluctuations.

In conclusion, supervised learning encompasses various types, including classification, regression, and time series analysis. Each type addresses specific learning tasks and requires different algorithms and techniques. Understanding these types helps in choosing the appropriate algorithms and approaches for specific data analysis and prediction tasks.

Next, we will explore how to leverage Spark for ML tasks.

ML with Spark

Spark provides a powerful and scalable platform for performing large-scale ML tasks. Spark's **ML library**, also known as **MLlib**, offers a wide range of algorithms and tools for building and deploying ML models.

The advantages of using Spark for ML include its distributed computing capabilities, efficient data processing, scalability, and integration with other Spark components, such as Spark SQL and Spark Streaming. Spark's MLlib supports both batch and streaming data processing, enabling the development of real-time ML applications.

ML is a transformative field that enables computers to learn from data and make predictions or decisions. By understanding the key concepts and leveraging tools such as Spark's MLlib, we can harness the power of ML to gain insights, automate processes, and drive innovation across various domains.

Now, let's take a look at the benefits of using Spark for ML tasks.

Advantages of Apache Spark for large-scale ML

By leveraging Spark's distributed computing capabilities and rich ecosystem, data scientists and engineers can effectively tackle complex ML challenges on massive datasets. It offers various advantages due to its distributed computing capabilities, some of which are as follows:

- **Speed and performance**: One of the key advantages of Apache Spark is its ability to handle large-scale data processing with exceptional speed. Spark leverages in-memory computing and optimized data processing techniques, such as **data parallelism** and **task pipelining**, to accelerate computations. This makes it highly efficient for iterative algorithms often used in ML, reducing the overall processing time significantly.
- **Distributed computing**: Spark's distributed computing model allows it to distribute data and computations across multiple nodes in a cluster, enabling parallel processing. This distributed nature enables Spark to scale horizontally, leveraging the computing power of multiple machines and processing data in parallel. This makes it well-suited for large-scale ML tasks that require processing massive volumes of data.
- **Fault tolerance**: Another advantage of Apache Spark is its built-in fault tolerance mechanism. Spark automatically tracks the lineage of **Resilient Distributed Datasets** (**RDDs**), which are the fundamental data abstraction in Spark, allowing it to recover from failures and rerun failed tasks. This ensures the reliability and resilience of Spark applications, making it a robust platform for handling large-scale ML workloads.
- **Versatility and flexibility**: Spark provides a wide range of APIs and libraries that facilitate various data processing and analytics tasks, including ML. Spark's MLlib library offers a rich set of distributed ML algorithms and utilities, making it easy to develop and deploy scalable ML models. Additionally, Spark integrates well with other popular data processing frameworks and tools, enabling seamless integration into existing data pipelines and ecosystems.
- **Real-time and streaming capabilities**: As we discussed in the previous chapter, Spark extends its capabilities beyond batch processing with its streaming component called Spark Streaming. This is particularly valuable in scenarios where immediate insights or decisions are required based on continuously arriving data, such as real-time fraud detection, sensor data analysis, or sentiment analysis on social media streams.

- **Ecosystem and community support**: Apache Spark has a vibrant and active community of developers and contributors, ensuring continuous development, improvement, and support. Spark benefits from a rich ecosystem of tools and extensions, providing additional functionality and integration options. The community-driven nature of Spark ensures a wealth of resources, documentation, tutorials, and online forums for learning and troubleshooting.

Therefore, Apache Spark offers significant advantages for large-scale ML tasks. Its speed, scalability, fault tolerance, versatility, and real-time capabilities make it a powerful framework for processing big data and developing scalable ML models.

Now let's take a look at different libraries that Spark provides to make use of ML capabilities in the distributed framework.

Spark MLlib versus Spark ML

Apache Spark provides two libraries for ML: Spark MLlib and Spark ML. Although they share a similar name, there are some key differences between the two libraries in terms of their design, APIs, and functionality. Let's compare Spark MLlib and Spark ML to understand their characteristics and use cases.

Spark MLlib

Spark MLlib is the original ML library in Apache Spark. It was introduced in earlier versions of Spark and provides a rich set of distributed ML algorithms and utilities. MLlib is built on top of the RDD API, which is the core data abstraction in Spark.

Spark MLlib has a few key features that set it apart from other non-distributed ML libraries such as `scikit-learn`. Let's look at a few of them:

- **RDD-based API**: MLlib leverages the RDD abstraction for distributed data processing, making it suitable for batch processing and iterative algorithms. The RDD API allows for efficient distributed computing but can be low-level and complex for some use cases.
- **Diverse algorithms**: MLlib offers a wide range of distributed ML algorithms, including classification, regression, clustering, collaborative filtering, dimensionality reduction, and more. These algorithms are implemented to work with large-scale data and can handle various tasks in the ML pipeline.
- **Feature engineering**: MLlib provides utilities for feature extraction, transformation, and selection. It includes methods for handling categorical and numerical features, text processing, and feature scaling.
- **Model persistence**: MLlib supports model persistence, allowing trained models to be saved to disk and loaded later for deployment or further analysis.

In the next section, we will explore the Spark ML library. This is the newer library that also provides ML capabilities.

Spark ML

Spark ML, introduced in Spark 2.0, is the newer ML library in Apache Spark. It is designed to be more user-friendly, with a higher-level API and a focus on DataFrames, which are a structured and optimized distributed data collection introduced in Spark SQL.

The key features of Spark ML are as follows:

- **DataFrame-based API**: Spark ML leverages the DataFrame API, which provides a more intuitive and higher-level interface compared to the RDD API. DataFrames offer a structured and tabular data representation, making it easier to work with structured data and integrate with Spark SQL.
- **Pipelines**: Spark ML introduces the concept of pipelines, which provides a higher-level abstraction for constructing ML workflows. Pipelines enable the chaining of multiple data transformations and model training stages into a single pipeline, simplifying the development and deployment of complex ML pipelines.
- **Integrated feature transformers**: Spark ML includes a rich set of feature transformers, such as StringIndexer, OneHotEncoder, VectorAssembler, and more. These transformers seamlessly integrate with DataFrames and simplify the feature engineering process.
- **Unified API**: Spark ML unifies the APIs for different ML tasks, such as classification, regression, clustering, and recommendation. This provides a consistent and cohesive programming interface across different algorithms and simplifies the learning curve.

Now that we know the key features of both Spark MLlib and Spark ML, let's explore when to use each of them.

You would benefit from using Spark MLlib in the following scenarios:

- You are working with older versions of Spark that do not support Spark ML
- You require low-level control and need to work directly with RDDs
- You need access to a specific algorithm or functionality that is not available in Spark ML

You should prefer to use Spark ML in the following scenarios:

- You are using Spark 2.0 or later versions
- You prefer a higher-level API and want to leverage DataFrames and Spark SQL capabilities
- You need to build end-to-end ML pipelines with integrated feature transformers and pipelines

Both Spark MLlib and Spark ML provide powerful ML capabilities in Apache Spark. As we've seen, Spark MLlib is the original library with a rich set of distributed algorithms, while Spark ML is a newer library with a more user-friendly API and integration with DataFrames. The choice between the two depends on your Spark version, preference for API style, and specific requirements of your ML tasks.

ML life cycle

The ML life cycle encompasses the end-to-end process of developing and deploying ML models. It involves several stages, each with its own set of tasks and considerations. Understanding the ML life cycle is crucial for building robust and successful ML solutions. In this section, we will explore the key stages of the ML life cycle:

1. **Problem definition**: The first stage of the ML life cycle is problem definition. It involves clearly defining the problem you want to solve and understanding the goals and objectives of your ML project. This stage requires collaboration between domain experts and data scientists to identify the problem, define success metrics, and establish the scope of the project.
2. **Data acquisition and understanding**: Once the problem has been defined, the next step is to acquire the necessary data for training and evaluation. Data acquisition may involve collecting data from various sources, such as databases, APIs, or external datasets. It is important to ensure data quality, completeness, and relevance to the problem at hand. Additionally, data understanding involves exploring and analyzing the acquired data to gain insights into its structure, distributions, and potential issues.
3. **Data preparation and feature engineering**: Data preparation and feature engineering are crucial steps in the ML life cycle. It involves transforming and preprocessing the data to make it suitable for training ML models. This includes tasks such as cleaning the data, handling missing values, encoding categorical variables, scaling features, and creating new features through feature engineering techniques. Proper data preparation and feature engineering significantly impact the performance and accuracy of ML models.
4. **Model training and evaluation**: In this stage, ML models are trained on the prepared data. Model training involves selecting an appropriate algorithm, defining the model architecture, and optimizing its parameters using training data. The trained model is then evaluated using evaluation metrics and validation techniques to assess its performance. This stage often requires iterating and fine-tuning the model to achieve the desired accuracy and generalization.
5. **Model deployment**: Once the model has been trained and evaluated, it is ready for deployment. Model deployment involves integrating the model into the production environment, making predictions on new data, and monitoring its performance. This may involve setting up APIs, creating batch or real-time inference systems, and ensuring the model's scalability and reliability. Deployment also includes considerations for model versioning, monitoring, and retraining to maintain the model's effectiveness over time.
6. **Model monitoring and maintenance**: Once the model has been deployed, it is important to continuously monitor its performance and maintain its effectiveness. Monitoring involves tracking model predictions, detecting anomalies, and collecting feedback from users or domain experts. It also includes periodic retraining of the model using new data to adapt to changing patterns or concepts. Model maintenance involves addressing model drift, updating dependencies, and managing the model's life cycle in the production environment.

7. **Model iteration and improvement**: The ML life cycle is an iterative process, and models often require improvement over time. Based on user feedback, performance metrics, and changing business requirements, models may need to be updated, retrained, or replaced. Iteration and improvement are essential for keeping the models up-to-date and ensuring they continue to deliver accurate predictions.

The ML life cycle involves problem definition, data acquisition, data preparation, model training, model deployment, model monitoring, and model iteration. Each stage plays a critical role in developing successful ML solutions. By following a well-defined life cycle, organizations can effectively build, deploy, and maintain ML models to solve complex problems and derive valuable insights from their data.

Problem statement

Let's dive into a case study where we'll explore the art of predicting house prices using historical data. Picture this: we have a treasure trove of valuable information about houses, including details such as zoning, lot area, building type, overall condition, year built, and sale price. Our goal is to harness the power of ML to accurately forecast the price of a new house that comes our way.

To accomplish this feat, we'll embark on a journey to construct an ML model exclusively designed for predicting house prices. This model will leverage the existing historical data and incorporate additional features. By carefully analyzing and understanding the relationships between these features and the corresponding sale prices, our model will become a reliable tool for estimating the value of any new house that enters the market.

To achieve this, we will go through some of the steps defined in the previous section, where we talked about the ML life cycle. Since housing prices are continuous, we will use a linear regression model to predict these prices.

We will start by preparing the data to make it usable for an ML model.

Data preparation and feature engineering

As we know, data preparation and feature engineering are crucial steps in the ML process. Proper data preparation and feature engineering techniques can significantly improve the performance and accuracy of models. In this section, we will explore common data preparation and feature engineering tasks with code examples.

Introduction to the dataset

The first step in building a model is to find the relevant data. We are going to use house price data (located at `https://docs.google.com/spreadsheets/d/1caaR9pT24GNmq3rDQpMiIMJrmiTGarbs/edit#gid=1150341366`) for this purpose. This data has 2,920 rows and 13 columns.

This dataset has the following columns:

- `Id`: Unique identifier for each row of the data
- `MSSubClass`: Subclass of the property
- `MSZoning`: Zoning of the property
- `LotArea`: Total area of the lot where the property is situated
- `LotConfig`: Configuration of the lot – for example, if it's a corner lot
- `BldgType`: Type of home – for example, single, family, and so on
- `OverallCond`: General condition of the house
- `YearBuilt`: The year the house was built in
- `YearRemodAdd`: The year any remodeling was done
- `Exterior1st`: Type of exterior – for example, vinyl, siding, and so on
- `BsmtFinSF2`: Total size of finished basement
- `TotalBsmtSF`: Total size of basement
- `SalePrice`: The sale price of the house

We will download this data from the link provided at the beginning of this section.

Now that we know some of the data points that exist in the data, let's learn how to load it.

Loading data

At this point, we already have the data downloaded on our computer and to our Databricks environment as a CSV file. As you may recall from the previous chapters, we learned how to load a dataset into a DataFrame through various techniques. We will use a CSV file here to load the data:

```
housing_data = spark.read.csv("HousePricePrediction.csv")

# Printing first 5 records of the dataset
housing_data.show(5)
```

We can see the result in Figure 8.1. Please note that we can see only part of the result in the image since the dataset is too large to be displayed in full.:

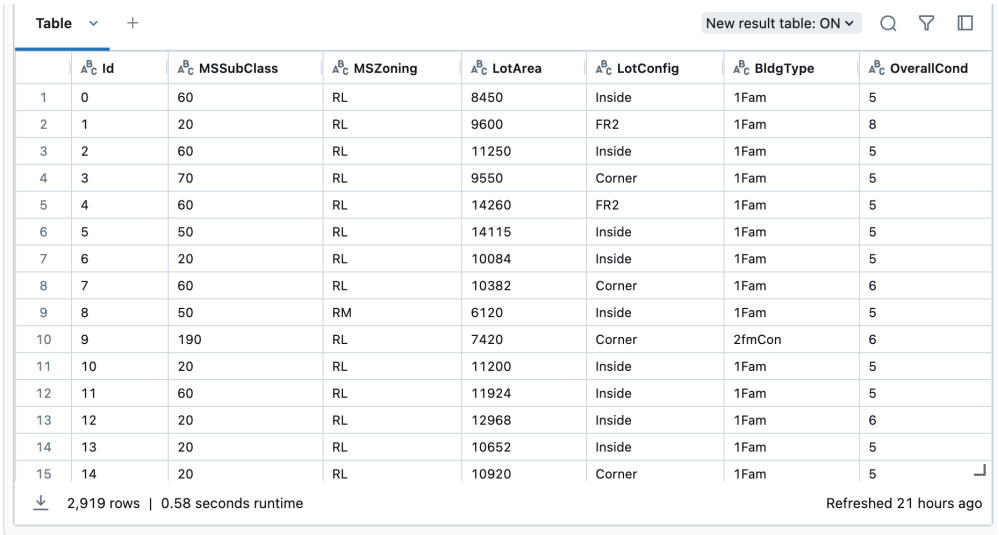


Table + New result table: ON

	Id	MSSubClass	MSZoning	LotArea	LotConfig	BldgType	OverallCond
1	0	60	RL	8450	Inside	1Fam	5
2	1	20	RL	9600	FR2	1Fam	8
3	2	60	RL	11250	Inside	1Fam	5
4	3	70	RL	9550	Corner	1Fam	5
5	4	60	RL	14260	FR2	1Fam	5
6	5	50	RL	14115	Inside	1Fam	5
7	6	20	RL	10084	Inside	1Fam	5
8	7	60	RL	10382	Corner	1Fam	6
9	8	50	RM	6120	Inside	1Fam	5
10	9	190	RL	7420	Corner	2fmCon	6
11	10	20	RL	11200	Inside	1Fam	5
12	11	60	RL	11924	Inside	1Fam	5
13	12	20	RL	12968	Inside	1Fam	6
14	13	20	RL	10652	Inside	1Fam	5
15	14	20	RL	10920	Corner	1Fam	5

2,919 rows | 0.58 seconds runtime Refreshed 21 hours ago

Let's print the schema of this dataset:

```
housing_data.printSchema
```

We will get the following schema:

```
<bound method DataFrame.printSchema of DataFrame[Id: bigint,
MSSubClass: bigint, MSZoning: string, LotArea: bigint, LotConfig:
string, BldgType: string, OverallCond: bigint, YearBuilt: bigint,
YearRemodAdd: bigint, Exterior1st: string, BsmtFinSF2: bigint,
TotalBsmtSF: bigint, SalePrice: bigint]>
```

As you may have noticed, some of the column types are strings. We will clean up this data in the next section.

Cleaning data

Cleaning the data involves handling missing values, outliers, and inconsistent data. Before we clean up the data, we will see how many rows are in the data. We can do this by using the `count()` function:

```
housing_data.count()
```

The result of this statement is shown here:

```
2919
```

This means the data contains 2,919 rows before we apply any cleaning. Now, we will drop missing values from this dataset, like so:

```
# Remove rows with missing values
cleaned_data = housing_data.dropna()
cleaned_data.count()
```

The result of this code is as follows:

```
1460
```

This shows that we have dropped some rows of data and that the data size is smaller now.

In the next section, we will discuss categorical variables and how to handle them, specifically those represented as strings in our example.

Handling categorical variables

In the realm of statistics and data analysis, a categorical variable is a type of variable that represents categories or groups and can take on a limited, fixed number of distinct values or levels. These variables signify qualitative characteristics and do not possess inherent numerical significance or magnitude. Instead, they represent different attributes or labels that classify data into specific groups or classes. Categorical variables need to be encoded to numerical values before training machine learning models.

In our example, we have a few columns that are string types. Those need to be encoded into numerical values so that the model can correctly use them. For this purpose, we'll use Spark's `StringIndexer` library to index the string columns.

The following code shows how to use `StringIndexer`:

```
#import required libraries
from pyspark.ml.feature import StringIndexer

mszoning_indexer = StringIndexer(inputCol="MSZoning",
outputCol="MSZoningIndex")
#Fits a model to the input dataset with optional parameters.
df_mszoning = mszoning_indexer.fit(cleaned_data).transform(cleaned_
data)
df_mszoning.show()
```

In the preceding code, we are taking the `MSZoning` column and converting it into an indexed column. To achieve this, we created a `StringIndexer` value by the name of `mszoning` indexer. We gave it `MSZoning` as the input column to work on. The output column's name is `MSZoningIndex`. We will use this output column in the next step. After that, we'll fit `mszoning_indexer` to `cleaned_data`.

In the resulting DataFrame, you will notice that one additional column was added by the name of `MSZoningIndex`.

Now, we will use a pipeline to transform all the features in the DataFrame.

A **pipeline** brings together a series of essential steps, each contributing to transforming raw data into valuable predictions and analyses. The pipeline serves as a structured pathway, composed of distinct stages or components, arranged in a specific order. Each stage represents a unique operation or transformation that refines the data, molding it into a more suitable format for ML tasks.

At the heart of a pipeline lies its ability to seamlessly connect these stages, forming a well-coordinated flow of transformations. This orchestration ensures that the data flows effortlessly through each stage, with the output of one stage becoming the input for the next. It eradicates the need for manual intervention, automating the entire process and saving us valuable time and effort. We integrate a variety of operations into the pipeline, such as data cleaning, feature engineering, encoding categorical variables, scaling numerical features, and much more. Each operation plays its part in transforming the data, to make it usable for the ML model.

The ML pipeline empowers us to streamline our workflows, experiment with different combinations of transformations, and maintain consistency in our data processing tasks. It provides a structured framework that allows us to effortlessly reproduce and share our work, fostering collaboration and fostering a deeper understanding of the data transformation process.

In ML and data preprocessing, a **one-hot encoder** is a technique that's used to convert categorical variables into a numerical format, allowing algorithms to better understand and process categorical data. It's particularly useful when working with categorical features that lack ordinal relationships or numerical representation.

We will use `StringIndexer` and `OneHotEncoder` in this pipeline. Let's see how we can achieve this:

```
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import OneHotEncoder
from pyspark.ml import Pipeline


mszoning_indexer = StringIndexer(inputCol="MSZoning",
outputCol="MSZoningIndex")
lotconfig_indexer = StringIndexer(inputCol="LotConfig",
outputCol="LotConfigIndex")
bldgtype_indexer = StringIndexer(inputCol="BldgType",
outputCol="BldgTypeIndex")
exterior1st_indexer = StringIndexer(inputCol="Exterior1st",
outputCol="Exterior1stIndex")
onehotencoder_mszoning_vector =
OneHotEncoder(inputCol="MSZoningIndex", outputCol="MSZoningVector")
onehotencoder_lotconfig_vector =
OneHotEncoder(inputCol="LotConfigIndex", outputCol="LotConfigVector")
onehotencoder_bldgtype_vector =
OneHotEncoder(inputCol="BldgTypeIndex", outputCol="BldgTypeVector")
onehotencoder_exterior1st_vector =
OneHotEncoder(inputCol="Exterior1stIndex",
```

```
outputCol="Exterior1stVector")

#Create pipeline and pass all stages
pipeline = Pipeline(stages=[mszoning_indexer,
                            lotconfig_indexer,
                            bldgtype_indexer,
                            exterior1st_indexer,
                            onehotencoder_mszoning_vector,
                            onehotencoder_lotconfig_vector,
                            onehotencoder_bldgtype_vector,
                            onehotencoder_exterior1st_vector])
```

To begin our code, we import the required modules from the PySpark library. The `StringIndexer` and `OneHotEncoder` modules will be used to handle the string columns of the housing dataset.

As we embark on the process of transforming categorical columns into numerical representations that can be understood by ML algorithms, let's take a closer look at the magic happening in our code.

The first step is to create `StringIndexer` instances for each categorical column we wish to transform. Each instance takes an input column, such as `MSZoning` or `LotConfig`, and produces a corresponding output column with a numerical index. For example, the `MSZoningIndex` column captures the transformed index values of the `MSZoning` column.

With the categorical columns successfully indexed, we progress to the next stage. Now, we want to convert these indices into binary vectors. For that, we can use `OneHotEncoder`. The resulting vectors represent each categorical value as a binary array, with a value of 1 indicating the presence of that category and 0 otherwise.

We create `OneHotEncoder` instances for each indexed column, such as `MSZoningIndex` or `LotConfigIndex`, and generate new output columns holding the binary vector representations. These output columns, such as `MSZoningVector` or `LotConfigVector`, are used to capture the encoded information.

As our code progresses, we assemble a pipeline – a sequence of transformations – where each transformation represents a stage. In our case, each stage encompasses the steps of indexing and one-hot encoding for a specific categorical column. We arrange the stages in the pipeline, ensuring the correct order of transformations.

By structuring our pipeline, we orchestrate a seamless flow of operations. The pipeline connects the dots between different stages, making it effortless to apply these transformations to our dataset as a whole. Our pipeline acts as a conductor, leading our data through the transformations, ultimately making it into a format ready for ML.

Now, we will fit this pipeline to our cleaned dataset so that all the columns can be transformed together:

```
df_transformed = pipeline.fit(cleaned_data).transform(cleaned_data)
df_transformed.show(5)
```

The resulting DataFrame will have the additional columns that we created in the pipeline with transformation. We have created index and vector columns for each of the string columns.

Now, we need to remove the unnecessary and redundant columns from our dataset. We will do this in the next section.

Data cleanup

In this step, we will make sure that we are only using the features needed by ML. To achieve this, we will remove different additional columns, such as the identity column, which don't serve the model. Moreover, we will also remove the features that we have already applied transformations to, such as string columns.

The following code shows how to delete the columns:

```
drop_column_list = ["Id", "MSZoning","LotConfig","BldgType",
"Exterior1st"]
df_dropped_cols = df_transformed.select([column for column in df_
transformed.columns if column not in drop_column_list])
df_dropped_cols.columns
```

Here's the result:

```
['MSSubClass',
 'LotArea',
 'OverallCond',
 'YearBuilt',
 'YearRemodAdd',
 'BsmtFinSF2',
 'TotalBsmtSF',
 'SalePrice',
 'MSZoningIndex',
 'LotConfigIndex',
 'BldgTypeIndex',
 'Exterior1stIndex',
 'MSZoningVector',
 'LotConfigVector',
 'BldgTypeVector',
 'Exterior1stVector']
```

As you can see from the resulting column list, the `Id`, `MSZoning`, `LotConfig`, `BldgType`, and `Exterior1st` columns have been deleted from the resulting DataFrame.

The next step in the process is assembling the data.

Assembling the vector

In this step, we will assemble a vector based on the features that we want. This step is necessary for Spark ML to work with data.

The following code captures how we can achieve this:

```
from pyspark.ml.feature import VectorAssembler

#Assembling features
feature_assembly = VectorAssembler(inputCols = ['MSSubClass',
 'LotArea',
 'OverallCond',
 'YearBuilt',
 'YearRemodAdd',
 'BsmtFinSF2',
 'TotalBsmtSF',
 'MSZoningIndex',
 'LotConfigIndex',
 'BldgTypeIndex',
 'Exterior1stIndex',
 'MSZoningVector',
 'LotConfigVector',
 'BldgTypeVector',
 'Exterior1stVector'], outputCol = 'features')
output = feature_assembly.transform(df_dropped_cols)
output.show(3)
```

In the preceding code block, we have created a features column that contains the assembled vector. We will use this column for our model training after scaling it.

Once the vector has been assembled, the next step in the process is to scale the data.

Feature scaling

Feature scaling ensures that all features are on a similar scale, preventing certain features from dominating the learning process.

For this, we can use the following code:

```
#Normalizing the features
from pyspark.ml.feature import StandardScaler

scaler = StandardScaler(inputCol="features",
outputCol="scaledFeatures",withStd=True, withMean=False)

# Compute summary statistics by fitting the StandardScaler
scalerModel = scaler.fit(output)

# Normalize each feature to have unit standard deviation.
scaledOutput = scalerModel.transform(output)
scaledOutput.show(3)
```

The following code selects only the scaled features and the target column – that is, `SalePrice`:

```
#Selecting input and output column from output
df_model_final = scaledOutput.select(['SalePrice', 'scaledFeatures'])
df_model_final.show(3)
```

We'll get the following output:

```
+---------+--------------------+
|SalePrice|   scaledFeatures   |
+---------+--------------------+
|   208500|(37,[0,1,2,3,4,6,...|
|   181500|(37,[0,1,2,3,4,6,...|
|   223500|(37,[0,1,2,3,4,6,...|
+---------+--------------------+
```

As you can see, `df_model_final` now only has two columns. `SalePrice` is the column that we're going to predict so that is our target column. `scaledFeatures` contains all the features that we are going to use to train our ML model.

These examples demonstrate common data preparation and feature engineering tasks using PySpark. However, the specific techniques and methods applied may vary, depending on the dataset and the requirements of the ML task. It is essential to understand the characteristics of the data and choose appropriate techniques to preprocess and engineer features effectively. Proper data preparation and feature engineering lay the foundation for building accurate and robust ML models.

The next step in this process is training and evaluating the ML model.

Model training and evaluation

Model training and evaluation are crucial steps in the ML process. In this section, we will explore how to train ML models and evaluate their performance using various metrics and techniques. We will use PySpark as the framework for model training and evaluation.

Splitting the data

Before training a model, it is important to split the dataset into training and testing sets. The training set is used to train the model, while the testing set is used to evaluate its performance. Here's an example of how to split the data using PySpark:

```
#test train split
df_train, df_test = df_model_final.randomSplit([0.75, 0.25])
```

In the preceding code, we are doing a random split of the data, putting 75% of the data into the training set and 25% of the data into the test set. There are other split techniques as well. You should look at your data carefully and then define the split that works best for your data and model training.

The reason we split the data is that once we train the model, we want to see how the trained model predicts on a dataset that it has never seen. In this case, that is our test dataset. This would help us evaluate the model and determine the quality of the model. Based on this, we can deploy different techniques to improve our model.

The next step is model training.

Model training

Once the data has been split, we can train an ML model on the training data. PySpark provides a wide range of algorithms for various types of ML tasks. For this example, we are going to use linear regression as our model of choice.

Here's an example of training a linear regression model:

```
from pyspark.ml.regression import LinearRegression
# Instantiate the linear regression model
regressor = LinearRegression(featuresCol = 'scaledFeatures', labelCol
= 'SalePrice')
# Fit the model on the training data
regressor = regressor.fit(df_train)
```

In the preceding code, we are using the training data and fitting it into a linear regressor model. We also added a parameter for `labelCol` that tells the model that this is the column that is our target column to predict.

Once the model has been trained, the next step is to determine how good the model is. We'll do this in the next section by evaluating the model.

Model evaluation

After training the model, we need to evaluate its performance on the test data. Evaluation metrics provide insights into how well the model is performing.

Mean squared error (**MSE**) is a fundamental statistical metric that's used to evaluate the performance of regression models by quantifying the average of the squared differences between predicted and actual values.

R-squared, often denoted as **R2**, is a statistical measure that represents the proportion of the variance in the dependent variable that is predictable or explained by the independent variables in a regression model. It serves as an indicator of how well the independent variables explain the variability of the dependent variable.

Here's an example of evaluating a regression model using the MSE and R2 metrics:

```
#MSE for the train data
pred_results = regressor.evaluate(df_train)
print("The train MSE for the model is: %2f"% pred_results.
meanAbsoluteError)
print("The train r2 for the model is: %2f"% pred_results.r2)
```

Here's the result:

```
The MSE for the model is: 32287.153682
The r2 for the model is: 0.614926
```

We can check the test data's performance as depicted here:

```
#Checking test performance
pred_results = regressor.evaluate(df_test)
print("The test MSE for the model is: %2f"% pred_results.
meanAbsoluteError)
print("The test r2 for the model is: %2f"% pred_results.r2)
```

Here's the result:

```
The MSE for the model is: 31668.331218
The r2 for the model is: 0.613300
```

Based on the results of the model, we can tune it further. We'll see some of the techniques to achieve this in the next section.

Cross-validation

Cross-validation is one of the different methods to improve an ML model's performance.

Cross-validation is used to assess the model's performance more robustly by dividing the data into multiple subsets for training and evaluation. So, instead of just using train and test data, we use a validation set as well, where the model never sees that data and is only used for measuring performance.

Cross-validation follows a simple principle: rather than relying on a single train-test split, we divide our dataset into multiple subsets, or **folds**. Each fold acts as a mini train-test split, with a portion of the data used for training and the remainder reserved for testing. By rotating the folds, we ensure that every data point gets an opportunity to be part of the test set, thereby mitigating biases and providing a more representative evaluation.

The most common form of cross-validation is **k-fold cross-validation**. In this method, the dataset is divided into k equal-sized folds. The model is trained and evaluated k times, with each fold serving as the test set once while the remaining folds collectively form the training set. By averaging the performance metrics obtained from each iteration, we obtain a more robust estimation of our model's performance.

Through cross-validation, we gain valuable insights into the generalization capabilities of our model. It allows us to gauge its performance across different subsets of the data, capturing the inherent variations and nuances that exist within our dataset. This technique helps us detect potential issues such as **overfitting**, where the model performs exceptionally well on the training set but fails to generalize to unseen data.

In addition to k-fold cross-validation, there are variations and extensions tailored to specific scenarios. **Stratified cross-validation** ensures that each fold maintains the same class distribution as the original dataset, preserving the representativeness of the splits. **Leave-one-out cross-validation**, on the other hand, treats each data point as a separate fold, providing a stringent evaluation but at the cost of increased computational complexity.

Next, we will learn about hyperparameter tuning.

Hyperparameter tuning

Hyperparameter tuning is the process of optimizing the hyperparameters of an ML algorithm to improve its performance. Hyperparameters are settings or configurations that are external to the model and cannot be learned from the training data directly. Unlike model parameters, which are learned during the training process, hyperparameters need to be specified beforehand and are crucial in determining the behavior and performance of an ML model. We will use hyperparameter tuning to improve model performance. **Hyperparameters** are parameters that are not learned from the data but are set before training. Tuning hyperparameters can significantly impact the model's performance.

Picture this: our model is a complex piece of machinery, composed of various knobs and levers known as hyperparameters. These hyperparameters govern the behavior and characteristics of our model, influencing its ability to learn, generalize, and make accurate predictions. However, finding the optimal configuration for these hyperparameters is no easy feat.

Hyperparameter tuning is the art of systematically searching and selecting the best combination of hyperparameters for our model. It allows us to venture beyond default settings and discover the configurations that align harmoniously with our data, extracting the most meaningful insights and delivering superior performance.

The goal of hyperparameter tuning is to get optimal values. We explore different hyperparameter settings, traversing through a multidimensional landscape of possibilities. This exploration can take various forms, such as grid search, random search, or more advanced techniques such as Bayesian optimization or genetic algorithms.

Grid search, a popular method, involves defining a grid of potential values for each hyperparameter. The model is then trained and evaluated for every possible combination within the grid. By exhaustively searching through the grid, we unearth the configuration that yields the highest performance, providing us with a solid foundation for further refinement.

Random search takes a different approach. It samples hyperparameter values randomly from predefined distributions and evaluates the model's performance for each sampled configuration. This randomized exploration enables us to cover a wider range of possibilities, potentially discovering unconventional yet highly effective configurations.

These examples demonstrate the process of model training and evaluation using PySpark. However, the specific algorithms, evaluation metrics, and techniques applied may vary, depending on the ML task at hand. It is important to understand the problem domain, select appropriate algorithms, and choose relevant evaluation metrics to train and evaluate models effectively.

Model deployment

Model deployment is the process of making trained ML models available for use in production environments. In this section, we will explore various approaches and techniques for deploying ML models effectively:

- **Serialization and persistence**: Once a model has been trained, it needs to be serialized and persisted to disk for later use. Serialization is the process of converting the model object into a format that can be stored, while persistence involves saving the serialized model to a storage system.
- **Model serving**: Model serving involves making the trained model available as an API endpoint or service that can receive input data and return predictions. This allows other applications or systems to integrate and use the model for real-time predictions.

Model monitoring and management

Once a model has been deployed, it is important to monitor its performance and behavior in the production environment and maintain its effectiveness over time. Monitoring can help identify issues such as data drift, model degradation, or anomalies. Additionally, model management involves versioning, tracking, and maintaining multiple versions of the deployed models. These practices ensure that models remain up to date and perform optimally over time. In this section, we will explore the key aspects of model monitoring and maintenance:

- **Scalability and performance**: When deploying ML models, scalability and performance are essential considerations. Models should be designed and deployed in a way that allows for efficient processing of large volumes of data and can handle high throughput requirements. Technologies such as Apache Spark provide distributed computing capabilities that enable scalable and high-performance model deployment.
- **Model updates and retraining**: ML models may need to be updated or retrained periodically to adapt to changing data patterns or improve performance. Deployed models should have mechanisms in place to facilitate updates and retraining without interrupting the serving process. This can involve automated processes, such as monitoring for data drift or retraining triggers based on specific conditions.
- **Performance metrics**: To monitor a deployed model, it is important to define and track relevant performance metrics. These metrics can vary, depending on the type of ML problem and the specific requirements of the application. Some commonly used performance metrics include accuracy, precision, recall, F1 score, and **area under the ROC curve** (**AUC**). By regularly evaluating these metrics, deviations from the expected performance can be identified, indicating the need for further investigation or maintenance actions.
- **Data drift detection**: Data drift refers to the phenomenon where the statistical properties of the input data change over time, leading to a degradation in model performance. Monitoring for data drift is crucial to ensure that the deployed model continues to provide accurate predictions. Techniques such as statistical tests, feature distribution comparison, and outlier detection can be employed to detect data drift. When data drift is detected, it may be necessary to update the model or retrain it using more recent data.
- **Model performance monitoring**: Monitoring the performance of a deployed model involves tracking its predictions and comparing them with ground truth values. This can be done by periodically sampling a subset of the predictions and evaluating them against the actual outcomes. Monitoring can also include analyzing prediction errors, identifying patterns or anomalies, and investigating the root causes of any performance degradation. By regularly monitoring the model's performance, issues can be identified early on and corrective actions can be taken.
- **Model retraining and updates**: Models that are deployed in production may require periodic updates or retraining to maintain their effectiveness. When new data becomes available or significant changes occur in the application domain, retraining the model with fresh data can

help improve its performance. Additionally, bug fixes, feature enhancements, or algorithmic improvements may necessitate updating the deployed model. It is important to have a well-defined process and infrastructure in place to handle model retraining and updates efficiently.

- **Versioning and model governance**: Maintaining proper versioning and governance of deployed models is crucial for tracking changes, maintaining reproducibility, and ensuring regulatory compliance. Version control systems can be used to manage model versions, track changes, and provide a historical record of model updates. Additionally, maintaining documentation related to model changes, dependencies, and associated processes contributes to effective model governance.
- **Collaboration and feedback**: Model monitoring and maintenance often involve collaboration among different stakeholders, including data scientists, engineers, domain experts, and business users. Establishing channels for feedback and communication can facilitate the exchange of insights, identification of issues, and implementation of necessary changes. Regular meetings or feedback loops can help align the model's performance with the evolving requirements of the application.

Overall, model deployment is a critical step in the ML life cycle. It involves serializing and persisting trained models, serving them as APIs or services, monitoring their performance, ensuring scalability and performance, and managing updates and retraining.

By actively monitoring and maintaining deployed models, organizations can ensure that their ML systems continue to provide accurate and reliable predictions. Effective model monitoring techniques, coupled with proactive maintenance strategies, enable timely identification of performance issues and support the necessary actions to keep the models up to date and aligned with business objectives.

Model iteration and improvement

Model iteration and improvement is a crucial phase in the ML life cycle that focuses on enhancing the performance and effectiveness of deployed models. By continuously refining and optimizing models, organizations can achieve better predictions and drive greater value from their ML initiatives. In this section, we will explore the key aspects of model iteration and improvement:

- **Collecting feedback and gathering insights**: The first step in model iteration and improvement is to gather feedback from various stakeholders, including end users, domain experts, and business teams. Feedback can provide valuable insights into the model's performance, areas for improvement, and potential issues encountered in real-world scenarios. This feedback can be collected through surveys, user interviews, or monitoring the model's behavior in the production environment.
- **Analyzing model performance**: To identify areas for improvement, it is important to thoroughly analyze the model's performance. This includes examining performance metrics, evaluating prediction errors, and conducting in-depth analyses of misclassified or poorly predicted

instances. By understanding the strengths and weaknesses of the model, data scientists can focus their efforts on specific areas that require attention.

- **Exploring new features and data**: One way to improve model performance is by incorporating new features or utilizing additional data sources. Exploratory data analysis can help identify potential features that may have a strong impact on predictions. Feature engineering techniques, such as creating interaction terms, scaling, or transforming variables, can also be employed to enhance the representation of the data. Additionally, incorporating new data from different sources can provide fresh insights and improve the model's generalization capabilities.
- **Algorithm selection and hyperparameter tuning**: Experimenting with different algorithms and hyperparameters can lead to significant improvements in model performance. Data scientists can explore alternative algorithms or variations of the existing algorithm to identify the best approach for the given problem. Hyperparameter tuning techniques, such as grid search or Bayesian optimization, can be used to find optimal values for model parameters. This iterative process helps identify the best algorithm and parameter settings that yield superior results.
- **Ensemble methods**: Ensemble methods involve combining multiple models to create a more robust and accurate prediction. Techniques such as bagging, boosting, or stacking can be applied to build an ensemble model from multiple base models. Ensemble methods can often improve model performance by reducing bias, variance, and overfitting. Experimenting with different ensemble strategies and model combinations can lead to further enhancements in prediction accuracy.
- **A/B testing and controlled experiments**: A/B testing or controlled experiments can be conducted to evaluate the impact of model improvements in a controlled setting. By randomly assigning users or data samples to different versions of the model, organizations can measure the performance of the new model against the existing one. This approach provides statistically significant results to determine if the proposed changes lead to desired improvements or not.
- **Continuous monitoring and evaluation**: Once the improved model has been deployed, continuous monitoring and evaluation are essential to ensure its ongoing performance. Monitoring for data drift, analyzing performance metrics, and conducting periodic evaluations help identify potential degradation or the need for further improvements. This feedback loop allows for continuous iteration and refinement of the deployed model.

By embracing a culture of iteration and improvement, organizations can continuously enhance the performance and accuracy of their ML models. Through collecting feedback, analyzing model performance, exploring new features and algorithms, conducting experiments, and continuous monitoring, models can be iteratively refined to achieve better predictions and drive tangible business outcomes.

Case studies and real-world examples

In this section, we will explore two prominent use cases of ML: customer churn prediction and fraud detection. These examples demonstrate the practical applications of ML techniques in addressing real-world challenges and achieving significant business value.

Customer churn prediction

Customer churn refers to the phenomenon where customers discontinue their relationship with a company, typically by canceling a subscription or switching to a competitor. Predicting customer churn is crucial for businesses as it allows them to proactively identify customers who are at risk of leaving and take appropriate actions to retain them. ML models can analyze various customer attributes and behavior patterns to predict churn likelihood. Let's dive into a customer churn prediction case study.

Case study – telecommunications company

A telecommunications company wants to reduce customer churn by predicting which customers are most likely to cancel their subscriptions. The company collects extensive customer data, including demographics, call records, service usage, and customer complaints. By leveraging ML, they aim to identify key indicators of churn and build a predictive model to forecast future churners:

- **Data preparation**: The company gathers and preprocesses the customer data, ensuring it is cleaned, formatted, and ready for analysis. They combine customer profiles with historical churn information to create a labeled dataset.
- **Feature engineering**: To capture meaningful patterns, the company engineers relevant features from the available data. These features may include variables such as average call duration, number of complaints, monthly service usage, and tenure.
- **Model selection and training**: The company selects an appropriate ML algorithm, such as logistic regression, decision trees, or random forests, to build the churn prediction model. They split the dataset into training and testing sets, train the model on the training data, and evaluate its performance on the testing data.
- **Model evaluation**: The model's performance is assessed using evaluation metrics such as accuracy, precision, recall, and F1 score. The company analyzes the model's ability to correctly identify churners and non-churners, striking a balance between false positives and false negatives.
- **Model deployment and monitoring**: Once the model meets the desired performance criteria, it is deployed into the production environment. The model continuously monitors incoming customer data, generates churn predictions, and triggers appropriate retention strategies for at-risk customers.

Fraud detection

Fraud detection is another critical application of ML that aims to identify fraudulent activities and prevent financial losses. ML models can learn patterns of fraudulent behavior from historical data and flag suspicious transactions or activities in real time. Let's explore a fraud detection case study.

Case study – financial institution

A financial institution wants to detect fraudulent transactions in real time to protect its customers and prevent monetary losses. The institution collects transaction data, including transaction amounts, timestamps, merchant information, and customer details. By leveraging ML algorithms, they aim to build a robust fraud detection system:

- **Data preprocessing**: The financial institution processes and cleans the transaction data, ensuring data integrity and consistency. They may also enrich the data by incorporating additional information, such as IP addresses or device identifiers, to enhance fraud detection capabilities.
- **Feature engineering**: Relevant features are extracted from the transaction data to capture potential indicators of fraudulent activity. These features may include transaction amounts, frequency, geographical location, deviation from typical spending patterns, and customer transaction history.
- **Model training**: The financial institution selects suitable ML algorithms, such as anomaly detection techniques or supervised learning methods (for example, logistic regression and gradient boosting), to train the fraud detection model. The model is trained on historical data labeled as fraudulent or non-fraudulent.
- **Real-time monitoring**: Once the model has been trained, it is deployed to analyze incoming transactions in real time. The model assigns a fraud probability score to each transaction, and transactions exceeding a certain threshold are flagged for further investigation or intervention.
- **Continuous improvement**: The financial institution continuously refines the fraud detection model by monitoring its performance and incorporating new data. They periodically evaluate the model's effectiveness, adjust thresholds, and update the model to adapt to evolving fraud patterns and techniques.

By applying ML techniques to customer churn prediction and fraud detection, organizations can enhance their decision-making processes, improve customer retention, and mitigate financial risks. These case studies highlight the practical application of ML in real-world scenarios, demonstrating its value in various industries.

Future trends in Spark ML and distributed ML

As the field of ML continues to evolve, there are several future trends and advancements that we can expect in Spark ML and distributed ML. Here are a few key areas to watch:

- **Deep learning integration**: Spark ML is likely to see deeper integration with deep learning frameworks such as TensorFlow and PyTorch. This will enable users to seamlessly incorporate deep learning models into their Spark ML pipelines, unlocking the power of neural networks for complex tasks such as image recognition and natural language processing.
- **Automated ML**: Automation will play a significant role in simplifying and accelerating the machine learning process. We can anticipate advancements in automated feature engineering, hyperparameter tuning, and model selection techniques within Spark ML. These advancements will make it easier for users to build high-performing models with minimal manual effort.
- **Explainable AI**: As the demand for transparency and interpretability in machine learning models grows, Spark ML is likely to incorporate techniques for model interpretability. This will enable users to understand and explain the predictions made by their models, making them more trustworthy and compliant with regulatory requirements.
- **Generative AI (GenAI):** GenAI is the latest rage. As use cases for GenAI become more in demand, the current platforms may incorporate some of the LLMs that are used in GenAI.
- **Edge computing and IoT**: With the rise of edge computing and the **Internet of Things** (**IoT**), Spark ML is expected to extend its capabilities to support ML inference and training on edge devices. This will enable real-time, low-latency predictions and distributed learning across edge devices, opening up new possibilities for applications in areas like smart cities, autonomous vehicles, and industrial IoT.

And with that we've concluded the learning portion of the book. Let's briefly recap what we've covered.

Summary

In conclusion, Spark ML provides a powerful and scalable framework for distributed ML tasks. Its integration with Apache Spark offers significant advantages in terms of processing large-scale datasets, parallel computing, and fault tolerance. Throughout this chapter, we explored the key concepts, techniques, and real-world examples of Spark ML.

We discussed the ML life cycle, emphasizing the importance of data preparation, model training, evaluation, deployment, monitoring, and continuous improvement. We also compared Spark MLlib and Spark ML, highlighting their respective features and use cases.

Throughout this chapter, we discussed various key concepts and techniques related to Spark ML. We explored different types of ML, such as classification, regression, time series analysis, supervised learning, and unsupervised learning. We highlighted the importance of data preparation and feature engineering in building effective ML pipelines. We also touched upon fault-tolerance and reliability aspects in Spark ML, ensuring robustness and data integrity.

Furthermore, we examined real-world use cases, including customer churn prediction and fraud detection, to demonstrate the practical applications of Spark ML in solving complex business challenges.

These case studies showcased how organizations can leverage Spark ML to enhance decision-making, improve customer retention, and mitigate financial risks.

As you continue your journey in ML with Spark ML, it is important to keep the iterative and dynamic nature of the field in mind. Stay updated with the latest advancements, explore new techniques, and embrace a mindset of continuous learning and improvement.

By harnessing the power of Spark ML, you can unlock valuable insights from your data, build sophisticated ML models, and make informed decisions that drive business success. So, leverage the capabilities of Spark ML, embrace the future trends, and embark on your journey toward mastering distributed ML.

That concludes this chapter. Hopefully, it will help you on your exciting journey in the world of ML models. The next two chapters are mock tests to prepare you for the certification exam.

Part 5: Mock Papers

This part will provide two mock papers to help readers prepare for the certification exam by practicing questions.

This part has the following chapters:

- *Chapter 9, Mock Paper 1*
- *Chapter 10, Mock Paper 2*

9
Mock Test 1

Questions

Try your hand at these practice questions to test your knowledge of Apache Spark:

Question 1:

Which statement does not accurately describe a feature of the Spark driver?

A. The Spark driver serves as the node where the main method of a Spark application runs to co-ordinate the application
B. The Spark driver can be horizontally scaled to enhance overall processing throughput
C. The Spark driver houses the SparkContext object
D. The Spark driver is tasked with scheduling the execution of data by using different worker nodes in cluster mode
E. Optimal performance dictates that the Spark driver should be positioned as close as possible to worker nodes

Question 2:

Which of these statements accurately describes stages?

A. Tasks within a stage can be simultaneously executed by multiple machines
B. Various stages within a job can run concurrently
C. Stages comprise one or more jobs
D. Stages temporarily store transactions before committing them through actions

Question 3:

Which of these statements accurately describes Spark's cluster execution mode?

A. Cluster mode runs executor processes on gateway nodes
B. Cluster mode involves the driver being hosted on a gateway machine
C. In cluster mode, the Spark driver and the cluster manager are not co-located
D. The driver in cluster mode is located on a worker node

Question 4:

Which of these statements accurately describes Spark's client execution mode?

A. Client mode runs executor processes on gateway nodes
B. In client mode, the driver is co-located with the executor
C. In client mode, the Spark driver and the cluster manager are co-located
D. In client mode, the driver is found on an edge node

Question 5:

Which statement accurately describes Spark's standalone deployment mode?

A. Standalone mode utilizes only one executor per worker for each application
B. In standalone mode, the driver is located on a worker node
C. In standalone mode, the cluster does not need the driver
D. In standalone mode, the driver is found on an edge node

Question 6:

What is a task in Spark?

A. The unit of work performed for each data partition within a task is slots
B. Tasks are the second-smallest entity that can be executed within Spark
C. Tasks featuring wide dependencies can be combined into a single task
D. A task is a single unit of work done by a partition within Spark

Question 7:

Which of the following is the highest level in Spark's execution hierarchy?

A. Job
B. Task
C. Executor
D. Stage

Question 8:

How can the concept of slots be accurately described in Spark's context?

A. The creation and termination of slots align with the workload of an executor
B. Spark strategically stores data on disk across various slots to enhance I/O performance
C. Each slot is consistently confined to a solitary core
D. Slots enable the tasks to run in parallel

Question 9:

What is the role of an executor in Spark?

A. The executor's role is to request the transformation of operations into DAG
B. There can only be one executor within a Spark environment
C. The executor processes partitions in an optimized and distributed manner
D. The executor schedules queries for execution
E.

Question 10:

What is the role of shuffle in Spark?

A. Shuffle broadcasts variables to different partitions
B. With shuffle, data is written to the disk
C. The shuffle command transforms data in Spark
D. Shuffles are a narrow transformation

Question 11:

What is the role of actions in Spark?

A. Actions only read data from a disk
B. Actions are used to modify existing RDDs
C. Actions trigger the execution of tasks
D. Actions are used to establish stage boundaries

Question 12:

Which of the following is one of the tasks of the cluster manager in Spark?

A. In the event of an executor failure, the cluster manager will collaborate with the driver to initiate a new executor
B. The cluster manager can coalesce partitions to increase the speed of complex data processing
C. The cluster manager collects runtime statistics of queries
D. The cluster manager creates query plans

Question 13:

Which of the following is one of the tasks of adaptive query execution in Spark?

A. Adaptive query execution can coalesce partitions to increase the speed of complex data processing
B. In the event of an executor failure, the adaptive query execution feature will collaborate with the driver to initiate a new executor
C. Adaptive query execution creates query plans
D. Adaptive query execution is responsible for spawning multiple executors to carry our tasks in Spark

Question 14:

Which of the following operations is considered a transformation?

A. `df.select()`
B. `df.show()`

C. `df.head()`

D. `df.count()`

Question 15:

What is a feature of lazy evaluation in Spark?

A. Spark will fail a job only during execution but not during definition

B. Spark will fail a job only during definition

C. Spark will execute upon receiving a transformation operation

D. Spark will fail upon receiving an action

Question 16:

Which of the following statements about Spark's execution hierarchy is correct?

A. In Spark's execution hierarchy, tasks are above the level of jobs

B. In Spark's execution hierarchy, multiple jobs are contained in a stage

C. In Spark's execution hierarchy, a job can potentially span multiple stage boundaries

D. In Spark's execution hierarchy, slots are the smallest unit

Question 17:

Which of the following is the characteristic of the Spark driver?

A. The worker nodes are responsible for transforming Spark operations into DAGs when the driver sends a command

B. The Spark driver is responsible for executing tasks and returning results to executors

C. Spark driver can be scaled by adding more machines so that the performance of Spark tasks can be improved

D. The Spark driver processes partitions in an optimized and distributed fashion

Question 18:

Which of the following statements about broadcast variables is accurate?

A. Broadcast variables are only present on driver nodes
B. Broadcast variables can only be used for tables that fit into memory
C. Broadcast variables are not immutable, meaning they can be shared across clusters
D. Broadcast variables are not shared across the worker nodes

Question 19:

Which of the following code blocks returns unique values in columns `employee_state` and `employee_salary` in DataFrame `df` for all columns?

A. `Df.select('employee_state').join(df.select('employee_salary'), col('employee_state')==col('employee_salary'), 'left').show()`
B. `df.select(col('employee_state'), col('employee_salary')).agg({'*': 'count'}).show()`
C. `df.select('employee_state', 'employee_salary').distinct().show()`
D. `df.select('employee_state').union(df.select('employee_salary')).distinct().show()`

Question 20:

Which of the following code blocks reads a Parquet file from the `my_fle_path` location, where the file name is `my_file.parquet`, into a DataFrame `df`?

A. `df = spark.mode("parquet").read("my_fle_path/my_file.parquet")`
B. `df = spark.read.path("my_fle_path/my_file.parquet")`
C. `df = spark.read().parquet("my_fle_path/my_file.parquet")`
D. `df = spark.read.parquet("/my_fle_path/my_file.parquet")`

Question 21:

Which of the following code blocks performs an inner join of the `salarydf` and `employeedf` DataFrames for columns `employeeSalaryID` and `employeeID`, respectively?

A. `salarydf.join(employeedf, salarydf.employeeID == employeedf.employeeSalaryID)`

B.

i. `Salarydf.createOrReplaceTempView(salarydf)`
ii. `employeedf.createOrReplaceTempView('employeedf')`
iii. `spark.sql("SELECT * FROM salarydf CROSS JOIN employeedf ON employeeSalaryID ==employeeID")`

C.

i. `salarydf`
ii. `.join(employeedf, col(employeeID)==col(employeeSalaryID))`

D.

i. `Salarydf.createOrReplaceTempView(salarydf)`
ii. `employeedf.createOrReplaceTempView('employeedf')`
iii. `SELECT * FROM salarydf`
iv. `INNER JOIN employeedf`
v. `ON salarydf.employeeSalaryID == employeedf. employeeID`

Question 22:

Which of the following code blocks returns the `df` DataFrame sorted in descending order by column salary, showing missing values in the end?

A. `df.sort(nulls_last("salary"))`
B. `df.orderBy("salary").nulls_last()`
C. `df.sort("salary", ascending=False)`
D. `df.nulls_last("salary")`

Question 23:

The following code block contains an error. The code block should return a copy of the `df` DataFrame, where the name of the column state is changed to `stateID`. Find the error.

Code block:

```
df.withColumn("stateID", "state")
```

A. The arguments to the method `"stateID"` and `"state"` should be swapped

B. The `withColumn` method should be replaced by the `withColumnRenamed` method

C. The `withColumn` method should be replaced by `withColumnRenamed` method, and the arguments to the method need to be reordered

D. There is no such method whereby the column name can be changed

Question 24:

Which of the following code blocks performs an inner join between the `salarydf` and `employeedf` DataFrames, using the `employeeID` and `salaryEmployeeID` columns as join keys, respectively?

A. `salarydf.join(employeedf, "inner", salarydf.employeedf == employeeID.salaryEmployeeID)`

B. `salarydf.join(employeedf, employeeID == salaryEmployeeID)`

C. `salarydf.join(employeedf, salarydf.salaryEmployeeID == employeedf.employeeID, "inner")`

D. `salarydf.join(employeedf, salarydf.employeeID == employeedf.salaryEmployeeID, "inner")`

Question 25:

The following code block should return a `df` DataFrame, where the `employeeID` column is converted into an integer. Choose the answer that correctly fills the blanks in the code block to accomplish this:

```
df.__1__(__2__.__3__(__4__))
```

A.

i. `select`

ii. `col("employeeID")`

iii. `as`

iv. `IntegerType`

B.

i. `select`

ii. `col("employeeID")`

iii. `as`

iv. `Integer`

C.

i. `cast`
ii. `"employeeID"`
iii. `as`
iv. `IntegerType()`

D.

i. `select`
ii. `col("employeeID")`
iii. `cast`
iv. `IntegerType()`

Question 26:

Find the number of records that are not empty in the column department of the resulting DataFrame when we join the `employeedf` and `salarydf` DataFrames for the `employeeID` and `employeeSalaryID` columns, respectively. Which code blocks (in order) should be executed to achieve this?

1. `.filter(col("department").isNotNull())`
2. `.count()`
3. `employeedf.join(salarydf, employeedf.employeeID == salarydf.employeeSalaryID)`
4. `employeedf.join(salarydf, employeedf.employeeID ==salarydf.employeeSalaryID, how='inner')`
5. `.filter(col(department).isnotnull())`
6. `.sum(col(department))`

A. 3, 1, 6
B. 3, 1, 2
C. 4, 1, 2
D. 3, 5, 2

Question 27:

Which of the following code blocks returns only those rows from the `df` DataFrame in which the values in the column state are unique?

A. `df.dropDuplicates(subset=["state"]).show()`

B. `df.distinct(subset=["state"]).show()`

C. `df.drop_duplicates(subset=["state"]).show()`

D. `df.unique("state").show()`

Question 28:

The following code block contains an error. The code block should return a copy of the `df` DataFrame with an additional column named `squared_number`, which has the square of the column number. Find the error.

Code block:

```
df.withColumnRenamed(col("number"), pow(col("number"), 0.2).
alias("squared_number"))
```

A. The arguments to the `withColumnRenamed` method need to be reordered

B. The `withColumnRenamed` method should be replaced by the `withColumn` method

C. The `withColumnRenamed` method should be replaced by the `select` method, and `0.2` should be replaced with `2`

D. The argument `0.2` should be replaced by `2`

Question 29:

Which of the following code blocks returns a new DataFrame in which column salary is renamed to `new_salary` and employee is renamed to `new_employee` in the `df` DataFrame?

A. `df.withColumnRenamed(salary, new_salary).`
`withColumnRenamed(employee, new_employee)`

B. `df.withColumnRenamed("salary", "new_salary")`

C. `df.withColumnRenamed("employee", "new_employee")`

D. `df.withColumn("salary", "new_salary").withColumn("employee",`
`"new_employee")`

E. `df.withColumnRenamed("salary", "new_salary").withColumnRenamed("employee", "new_employee")`

Question 30:

Which of the following code blocks returns a copy of the `df` DataFrame, where the column salary has been renamed to `employeeSalary`?

A. `df.withColumn(["salary", "employeeSalary"])`

B. `df.withColumnRenamed("salary").alias("employeeSalary ")`

C. `df.withColumnRenamed("salary", "employeeSalary ")`

D. `df.withColumn("salary", "employeeSalary ")`

Question 31:

The following code block contains an error. The code block should save the `df` DataFrame to the `my_file_path` path as a Parquet file, appending to any existing parquet file. Find the error.

```
df.format("parquet").option("mode", "append").save(my_file_path)
```

A. The code is not saved to the correct path

B. The `save()` and `format` functions should be swapped

C. The code block is missing a reference to the `DataFrameWriter`

D. The `option` mode should be overwritten to correctly write the file

Question 32:

How can we reduce the `df` DataFrame from 12 to 6 partitions?

A. `df.repartition(12)`

B. `df.coalesce(6).shuffle()`

C. `df.coalesce(6, shuffle=True)`

D. `df.repartition(6)`

Question 33:

Which of the following code blocks returns a DataFrame where the timestamp column is converted into unix epoch timestamps in a new column named `record_timestamp` with a format of day, month, and year?

A. `df.withColumn("record_timestamp", from_unixtime(unix_timestamp(col("timestamp")), "dd-MM-yyyy"))`

B. `df.withColumnRenamed("record_timestamp", from_unixtime(unix_timestamp(col("timestamp")), "dd-MM-yyyy"))`

C. `df.select ("record_timestamp", from_unixtime(unix_timestamp(col("timestamp")), "dd-MM-yyyy"))`

D. `df.withColumn("record_timestamp", from_unixtime(unix_timestamp(col("timestamp")), "MM-dd-yyyy"))`

Question 34:

Which of the following code blocks creates a new DataFrame by appending the rows of the DataFrame `salaryDf` to the rows of the DataFrame `employeeDf`, regardless of the fact that both DataFrames have different column names?

A. `salaryDf.join(employeeDf)`

B. `salaryDf.union(employeeDf)`

C. `salaryDf.concat(employeeDf)`

D. `salaryDf.unionAll(employeeDf)`

Question 35:

The following code block contains an error. The code block should calculate the total of all salaries in the `employee_salary` column across each department. Find the error.

```
df.agg("department").sum("employee_salary")
```

A. Instead of `avg("value")`, `avg(col("value"))` should be used

B. All column names should be wrapped in `col()` operators

C. `"storeId"` and "`value"` should be swapped

D. `Agg` should be replaced by `groupBy`

Question 36:

The following code block contains an error. The code block is intended to perform a cross-join of the `salarydf` and `employeedf` DataFrames for the `employeeSalaryID` and `employeeID` columns, respectively. Find the error.

```
employeedf.join(salarydf, [salarydf.employeeSalaryID, employeedf.
employeeID], "cross")
```

A. The join type `"cross"` in the argument needs to be replaced with `crossJoin`

B. `[salarydf.employeeSalaryID, employeedf.employeeID]` should be replaced by `salarydf.employeeSalaryID == employeedf.employeeID`

C. The `"cross"` argument should be eliminated since `"cross"` is the default join type

D. The `"cross"` argument should be eliminated from the call and `join` should be replaced by `crossJoin`

Question 37:

The following code block contains an error. The code block should display the schema of the `df` DataFrame. Find the error.

```
df.rdd.printSchema()
```

A. In Spark, we cannot print the schema of a DataFrame

B. `printSchema` is not callable through `df.rdd` and should be called directly from `df`

C. There is no method in Spark named `printSchema()`

D. The `print_schema()` method should be used instead of `printSchema()`

Question 38:

The following code block should write the `df` DataFrame as a Parquet file to the `filePath` path, replacing any existing file. Choose the answer that correctly fills the blanks in the code block to accomplish this:

```
df.__1__.format("parquet").__2__(__3__).__4__(filePath)
```

A.

i. `save`

ii. `mode`

iii. `"ignore"`

iv. `path`

B.

i. `store`
ii. `with`
iii. `"replace"`
iv. `path`

C.

i. `write`
ii. `mode`
iii. `"overwrite"`
iv. `save`

D.

i. `save`
ii. `mode`
iii. `"overwrite"`
iv. `path`

Question 39:

The following code block contains an error. The code block is supposed to sort the `df` DataFrame according to salary in descending order. Then, it should sort based on the bonus column, putting nulls to last. Find the error.

```
df.orderBy ('salary', asc_nulls_first(col('bonus')))
transactionsDf.orderBy('value', asc_nulls_first(col('predError')))
```

A. The `salary` column should be sorted in a descending way. Moreover, it should be wrapped in a `col()` operator
B. The `salary` column should be wrapped by the `col()` operator
C. The `bonus` column should be sorted in a descending way, putting `nulls` last
D. The `bonus` column should be sorted by `desc_nulls_first()` instead

Question 40:

The following code block contains an error. The code block should use the `square_root_method` Python method to find the square root of the `salary` column in the `df` DataFrame and return it in a new column called `sqrt_salary`. Find the error.

```
square_root_method_udf = udf(square_root_method)
df.withColumn("sqrt_salary", square_root_method("salary"))
```

A. There is no return type specified for `square_root_method`

B. In the second line of the code, Spark needs to call `squre_root_method_udf` instead of `square_root_method`

C. `udf` is not registered with Spark

D. A new column needs to be added

Question 41:

The following code block contains an error. The code block should return the `df` DataFrame with `employeeID` renamed to `employeeIdColumn`. Find the error.

```
df.withColumn("employeeIdColumn", "employeeID")
```

A. Instead of `withColumn`, the `withColumnRenamed` method should be used

B. Instead of `withColumn`, the `withColumnRenamed` method should be used and argument `"employeeIdColumn"` should be swapped with argument `"employeeID"`

C. Arguments `"employeeIdColumn"` and `"employeeID"` should be swapped

D. The `withColumn` operator should be replaced with the `withColumnRenamed` operator

Question 42:

Which of the following code blocks will return a new DataFrame with the same columns as DataFrame `df`, except for the `salary` column?

A. `df.drop("salary")`

B. `df.drop(col(salary))`

C. `df.drop(salary)`

D. `df.delete("salary")`

Question 43:

Which of the following code blocks returns a DataFrame showing the mean of the salary column from the `df` DataFrame, grouped by column department?

A. `df.groupBy("department").agg(avg("salary"))`

B. `df.groupBy(col(department).avg())`

C. `df.groupBy("department").avg(col("salary"))`

D. `df.groupBy("department").agg(average("salary"))`

Question 44:

Which of the following code blocks creates a DataFrame that shows the mean of the salary column of the `salaryDf` DataFrame, based on the department and state columns, where age is greater than 35?

1. `salaryDf.filter(col("age") > 35)`
2. `.filter(col("employeeID")`
3. `.filter(col("employeeID").isNotNull())`
4. `.groupBy("department")`
5. `.groupBy("department", "state")`
6. `.agg(avg("salary").alias("mean_salary"))`
7. `.agg(average("salary").alias("mean_salary"))`

A. 1,2,5,6

B. 1,3,5,6

C. 1,3,6,7

D. 1,2,4,6

Question 45:

The following code block contains an error. The code block needs to cache the `df` DataFrame so that this DataFrame is fault-tolerant. Find the error.

```
df.persist(StorageLevel.MEMORY_AND_DISK_3)
```

A. `persist()` is not a function of the API DataFrame

B. `df.write()` should be used in conjunction with `df.persist` to correctly write the DataFrame

C. The storage level is incorrect and should be `MEMORY_AND_DISK_2`

D. `df.cache()` should be used instead of `df.persist()`

Question 46:

Which of the following code blocks concatenates the rows of the `salaryDf` and `employeeDf` DataFrames without any duplicates (assuming the columns of both DataFrames are similar)?

A. `salaryDf.concat(employeeDf).unique()`

B. `spark.union(salaryDf, employeeDf).distinct()`

C. `salaryDf.union(employeeDf).unique()`

D. `salaryDf.union(employeeDf).distinct()`

Question 47:

Which of the following code blocks reads a complete folder of CSV files from `filePath` with column headers?

A. `spark.option("header",True).csv(filePath)`

B. `spark.read.load(filePath)`

C. `spark.read().option("header",True).load(filePath)`

D. `spark.read.format("csv").option("header",True).load(filePath)`

Question 48:

The following code block contains an error. The `df` DataFrame contains columns [`employeeID`, `salary`, and `department`]. The code block should return a DataFrame that contains only the `employeeID` and `salary` columns from DataFrame `df`. Find the error.

```
df.select(col(department))
```

A. All column names from the `df` DataFrame should be specified in the `select` arguments

B. The `select` operator should be replaced by a `drop` operator, and all the column names from the `df` DataFrame should be listed as a list

C. The `select` operator should be replaced by a `drop` operator

D. The column name `department` should be listed like `col("department")`

Question 49:

The following code block contains an error. The code block should write DataFrame `df` as a Parquet file to the `filePath` location, after partitioning it for the `department` column. Find the error.

```
df.write.partition("department").parquet()
```

A. `partitionBy()` method should be used instead of `partition()`.

B. `partitionBy()` method should be used instead of `partition()` and `filePath` should be added to the `parquet` method

C. The `partition()` method should be called before the write method and `filePath` should be added to `parquet` method

D. The `"department"` column should be wrapped in a `col()` operator

Question 50:

Which of the following code blocks removes the cached `df` DataFrame from memory and disk?

A. `df.unpersist()`

B. `drop df`

C. `df.clearCache()`

D. `df.persist()`

Question 51:

The following code block should return a copy of the `df` DataFrame with an additional column: `test_column`, which has a value of `19`. Choose the answer that correctly fills the blanks in the code block to accomplish this:

```
df.__1__(__2__, __3__)
```

A.

i. `withColumn`

ii. `'test_column'`

iii. `19`

B.

i. `withColumnRenamed`

ii. `test_column`

iii. `lit(19)`

C.

i. `withColumn`

ii. `'test_column'`

iii. `lit(19)`

D.

i. `withColumnRenamed`

ii. `test_column`

iii. `19`

Question 52:

The following code block should return a DataFrame with the columns `employeeId`, `salary`, `bonus`, and `department` from `transactionsDf` DataFrame. Choose the answer that correctly fills the blanks to accomplish this:

```
df.__1__(__2__)
```

A.

i. `drop`

ii. `"employeeId", "salary", "bonus", "department"`

B.

i. `filter`

ii. `"employeeId, salary, bonus, department"`

C.

i. `select`

ii. `["employeeId", "salary", "bonus", "department"]`

D.

i. `select`

ii. `col(["employeeId", "salary", "bonus","department"])`

Question 53:

Which of the following code blocks returns a DataFrame with the `salary` column converted into a string in the `df` DataFrame?

A. `df.withColumn("salary", castString("salary", "string"))`
B. `df.withColumn("salary", col("salary").cast("string"))`
C. `df.select(cast("salary", "string"))`
D. `df.withColumn("salary", col("salary").castString("string"))`

Question 54:

The following code block contains an error. The code block should combine data from DataFrames `salaryDf` and `employeeDf`, showing all rows of DataFrame `salaryDf` that have a matching value in column `employeeSalaryID` with a value in column `employeeID` of DataFrame `employeeDf`. Find the error.

```
employeeDf.join(salaryDf, employeeDf.employeeID==employeeSalaryID)
```

A. The `join` statement is missing the right-hand DataFrame, where the column name is `employeeSalaryID`
B. The `union` method should be used instead of `join`
C. Instead of `join`, `innerJoin` should have been used
D. `salaryDf` should come in place of `employeeDf`

Question 55:

Which of the following code blocks reads a JSON file stored at `my_file_path` as a DataFrame?

A. `spark.read.json(my_file_path)`
B. `spark.read(my_file_path, source="json")`
C. `spark.read.path(my_file_path)`
D. `spark.read().json(my_file_path)`

Question 56:

The following code block contains an error. The code block should return a new DataFrame filtered by the rows where `salary` column is greater than 2000 in DataFrame `df`. Find the error.

```
df.where("col(salary) >= 2000")
```

A. Instead of `where()`, `filter()` should be used

B. The argument to the `where` method should be `"col(salary) > 2000"`

C. Instead of >=, the operator > should be used

D. The argument to the `where` method should be `"salary > 2000"`

Question 57:

Which of the following code blocks returns a DataFrame in which the `salary` and `state` columns are dropped from the `df` DataFrame?

A. `df.withColumn ("salary", "state")`

B. `df.drop(["salary", "state"])`

C. `df.drop("salary", "state")`

D. `df.withColumnRenamed ("salary", "state")`

Question 58:

Which of the following code blocks returns a two-column DataFrame that contains counts of each department in the `df` DataFrame?

A. `df.count("department").distinct()`

B. `df.count("department")`

C. `df.groupBy("department").count()`

D. `df.groupBy("department").agg(count("department"))`

Question 59:

Which of the following code blocks prints the schema of a DataFrame and contains both column names and types?

A. `print(df.columns)`

B. `df.printSchema()`

C. `df.rdd.printSchema()`

D. `df.print_schema()`

Question 60:

Which of the following code blocks creates a new DataFrame with three columns: `department`, `age`, and `max_salary` and has the maximum salary for each employee from each department and each age group from the `df` DataFrame?

A. `df.max(salary)`

B. `df.groupBy(["department", "age"]).agg(max("salary").alias("max_salary"))`

C. `df.agg(max(salary).alias(max_salary')`

D. `df.groupby(department).agg(max(salary).alias(max_salary)`

Answers

1. B
2. A
3. D
4. D
5. A
6. D
7. A
8. D
9. C
10. B
11. C
12. A
13. A

14. A
15. A
16. C
17. B
18. B
19. D
20. D
21. D
22. C
23. C
24. D
25. D
26. C
27. A
28. C
29. E
30. C
31. C
32. D
33. A
34. B
35. D
36. B
37. B
38. C
39. A
40. B
41. B
42. A
43. A
44. A
45. C

46. D
47. D
48. C
49. B
50. A
51. C
52. C
53. B
54. A
55. A
56. D
57. C
58. C
59. B
60. B

10
Mock Test 2

Questions

Try your hand at these practice questions to test your knowledge of Apache Spark:

Question 1:

What is a task in Spark?

A. The unit of work performed for each data partition within a task is the slots
B. A task is the second-smallest entity that can be executed within Spark
C. Tasks featuring wide dependencies can be combined into a single task
D. A task is the smallest component that can be executed within Spark

Question 2:

What is the role of an executor in Spark?

A. The executor's role is to request the transformation of operations into a directed acyclic graph (DAG)
B. There can only be one executor within a Spark environment
C. Executors are tasked with executing the assignments provided to them by the driver
D. The executor schedules queries for execution

Question 3:

Which of the following is one of the tasks of Adaptive Query Execution in Spark?

A. Adaptive Query Execution collects runtime statistics during query execution to optimize query plans
B. Adaptive Query Execution is responsible for distributing tasks to executors
C. Adaptive Query Execution is responsible for wide operations in Spark
D. Adaptive Query Execution is responsible for fault tolerance in Spark

Question 4:

Which is the lowest level in Spark's execution hierarchy?

A. Task
B. Slot
C. Job
D. Stage

Question 5:

Which one of these operations is an action?

A. `DataFrame.count()`
B. `DataFrame.filter()`
C. `DataFrame.select()`
D. `DataFrame.groupBy()`

Question 6:

Which of the following describes the characteristics of the DataFrame API?

A. The DataFrame API is based on resilient distributed dataset (RDD) at the backend
B. The DataFrame API is available in Scala, but it is not available in Python
C. The DataFrame API does not have data manipulation functions
D. The DataFrame API is used for distributing tasks in executors

Question 7:

Which of the following statements is accurate about executors?

A. Slots are not a part of an executor
B. Executors are able to run tasks in parallel via slots
C. Executors are always equal to tasks
D. An executor is responsible for distributing tasks for a job

Question 8:

Which of the following statements is accurate about the Spark driver?

A. There are multiple drivers in a Spark application
B. Slots are a part of a driver
C. Drivers execute tasks in parallel
D. It is the responsibility of the Spark driver to transform operations into DAG computations

Question 9:

Which one of these operations is a wide transformation?

A. `DataFrame.show()`
B. `DataFrame.groupBy()`
C. `DataFrame.repartition()`
D. `DataFrame.select()`
E. `DataFrame.filter()`

Question 10:

Which of the following statements is correct about lazy evaluation?

A. Execution is triggered by transformations
B. Execution is triggered by actions
C. Statements are executed as they appear in the code
D. Spark distributes tasks to different executors

Question 11:

Which of the following is true about DAGs in Spark?

A. DAGs are lazily evaluated
B. DAGs can be scaled horizontally in Spark
C. DAGs are responsible for processing partitions in an optimized and distributed fashion
D. DAG is comprised of tasks that can run in parallel

Question 12:

Which of the following statements is true about Spark's fault tolerance mechanism?

A. Spark achieves fault tolerance via DAGs
B. It is the responsibility of the executor to enable fault tolerance in Spark
C. Because of fault tolerance, Spark can recompute any failed RDD
D. Spark builds a fault-tolerant layer on top of the legacy RDD data system, which by itself is not fault tolerant

Question 13:

What is the core of Spark's fault-tolerant mechanism?

A. RDD is at the core of Spark, which is fault tolerant by design
B. Data partitions, since data can be recomputed
C. DataFrame is at the core of Spark since it is immutable
D. Executors ensure that Spark remains fault tolerant

Question 14:

What is accurate about jobs in Spark?

A. Different stages in a job may be executed in parallel
B. Different stages in a job cannot be executed in parallel
C. A task consists of many jobs
D. A stage consists of many jobs

Question 15:

What is accurate about a shuffle in Spark?

A. In a shuffle, data is sent to multiple partitions to be processed
B. In a shuffle, data is sent to a single partition to be processed
C. A shuffle is an action that triggers evaluation in Spark
D. In a shuffle, all data remains in memory to be processed

Question 16:

What is accurate about the cluster manager in Spark?

A. The cluster manager is responsible for managing resources for Spark
B. The cluster manager is responsible for working with executors directly
C. The cluster manager is responsible for creating query plans
D. The cluster manager is responsible for optimizing DAGs

Question 17:

The following code block needs to take the sum and average of the `salary` column for each department in the `df` DataFrame. Then, it should calculate the sum and maximum value for the `bonus` column:

```
df.___1___ ("department").___2___ (sum("salary").alias("sum_salary"),
___3___ ("salary").alias("avg_salary"), sum("bonus").alias("sum_
bonus"), ___4___("bonus").alias("max_bonus") )
```

Choose the answer that correctly fills the blanks in the code block to accomplish this:

A.

i. `groupBy`
ii. `agg`
iii. `avg`
iv. `max`

B.

i. `filter`
ii. `agg`
iii. `avg`
iv. `max`

C.

i. `groupBy`
ii. `avg`
iii. `agg`
iv. `max`

D.

i. `groupBy`
ii. `agg`
iii. `avg`
iv. `avg`

Question 18:

The following code block contains an error. The code block needs to join the `salaryDf` DataFrame with the bigger `employeeDf` DataFrame on the `employeeID` column:

```
salaryDf.join(employeeDf, "employeeID", how="broadcast")
```

Identify the error:

A. Instead of `join`, the code should use `innerJoin`
B. `broadcast` is not a `join` type in Spark for joining two DataFrames
C. `salaryDf` and `employeeDf` should be swapped
D. In the `how` parameter, `crossJoin` should be used instead of `broadcast`

Question 19:

Which of the following code blocks shuffles the `df` DataFrame to have 20 partitions instead of 5 partitions?

A. `df.repartition(5)`

B. `df.repartition(20)`

C. `df.coalesce(20)`

D. `df.coalesce(5)`

Question 20:

Which of the following operations will trigger evaluation?

A. `df.filter()`

B. `df.distinct()`

C. `df.intersect()`

D. `df.join()`

E. `df.count()`

Question 21:

Which of the following code blocks returns unique values for the `age` and `name` columns in the `df` DataFrame in its respective columns where all values are unique in these columns?

A.
```
df.select('age').join(df.select('name'),
col(state)==col('name'), 'inner').show()
```

B. `df.select(col('age'), col('name')).agg({'*': 'count'}).show()`

C. `df.select('age', 'name').distinct().show()`

D. `df.select('age').unionAll(df.select('name')).distinct().show()`

Question 22:

Which of the following code blocks returns the count of the total number of rows in the `df` DataFrame?

A. `df.count()`

B. `df.select(col('state'), col('department')).agg({'*': 'count'}).show()`

C. `df.select('state', 'department').distinct().show()`

D. `df.select('state').union(df.select('department')).distinct().show()`

Question 23:

The following code block contains an error. The code block should save the `df` DataFrame at the `filePath` path as a new parquet file:

```
df.write.mode("append").parquet(filePath)
```

Identify the error:

A. The code block should have `overwrite` instead of `append` as an option

B. The code should be `write.parquet` instead of `write.mode`

C. The `df.write` operation cannot be called directly from the DataFrame

D. The first part of the code should be `df.write.mode(append)`

Question 24:

Which of the following code blocks adds a `salary_squared` column to the `df` DataFrame that is the square of the `salary` column?

A. `df.withColumnRenamed("salary_squared", pow(col("salary"), 2))`

B. `df.withColumn("salary_squared", col("salary"*2))`

C. `df.withColumn("salary_squared", pow(col("salary"), 2))`

D. `df.withColumn("salary_squared", square(col("salary")))`

Question 25:

Which of the following code blocks performs a join in which the small `salaryDf` DataFrame is sent to all executors so that it can be joined with the `employeeDf` DataFrame on the `employeeSalaryID` and `EmployeeID` columns, respectively?

A. `employeeDf.join(salaryDf, "employeeDf.employeeID == salaryDf.employeeSalaryID", "inner")`

B. `employeeDf.join(salaryDf, "employeeDf.employeeID == salaryDf.employeeSalaryID", "broadcast")`

C. `employeeDf.join(broadcast(salaryDf), employeeDf.employeeID == salaryDf.employeeSalaryID)`

D. `salaryDf.join(broadcast(employeeDf), employeeDf.employeeID == salaryDf.employeeSalaryID)`

Question 26:

Which of the following code blocks performs an outer join between the `salarydf` DataFrame and the `employeedf` DataFrame, using the `employeeID` and `salaryEmployeeID` columns as join keys respectively?

A. `Salarydf.join(employeedf, "outer", salarydf.employeedf == employeeID.salaryEmployeeID)`

B. `salarydf.join(employeedf, employeeID == salaryEmployeeID)`

C. `salarydf.join(employeedf, salarydf.salaryEmployeeID == employeedf.employeeID, "outer")`

D. `salarydf.join(employeedf, salarydf.employeeID == employeedf.salaryEmployeeID, "outer")`

Question 27:

Which of the following pieces of code would print the schema of the `df` DataFrame?

A. `df.rdd.printSchema`

B. `df.rdd.printSchema()`

C. `df.printSchema`

D. `df.printSchema()`

Question 28:

Which of the following code blocks performs a left join between the `salarydf` DataFrame and the `employeedf` DataFrame, using the `employeeID` column?

A. `salaryDf.join(employeeDf, salaryDf["employeeID"] == employeeDf["employeeID"], "outer")`

B. `salaryDf.join(employeeDf, salaryDf["employeeID"] == employeeDf["employeeID"], "left")`

C. `salaryDf.join(employeeDf, salaryDf["employeeID"] == employeeDf["employeeID"], "inner")`

D. `salaryDf.join(employeeDf, salaryDf["employeeID"] == employeeDf["employeeID"], "right")`

Question 29:

Which of the following code blocks aggregates the `bonus` column of the `df` DataFrame in ascending order with `nulls` being last?

A. `df.agg(asc_nulls_last("bonus").alias("bonus_agg"))`

B. `df.agg(asc_nulls_first("bonus").alias("bonus_agg"))`

C. `df.agg(asc_nulls_last("bonus", asc).alias("bonus_agg"))`

D. `df.agg(asc_nulls_first("bonus", asc).alias("bonus_agg"))`

Question 30:

The following code block contains an error. The code block should return a DataFrame by joining the `employeeDf` and `salaryDf` DataFrames on the `employeeID` and `employeeSalaryID` columns, respectively, excluding the `bonus` and `department` columns from the `employeeDf` DataFrame and the `salary` column from the `salaryDf` DataFrame in the final DataFrame.

```
employeeDf.groupBy(salaryDf, employeeDf.employeeID == salaryDf.
employeeSalaryID, "inner").delete("bonus", "department", "salary")
```

Identify the error:

A. `groupBy` should be replaced with the `innerJoin` operator

B. `groupBy` should be replaced with a `join` operator and `delete` should be replaced with `drop`

C. `groupBy` should be replaced with the `crossJoin` operator and `delete` should be replaced with `withColumn`

D. `groupBy` should be replaced with a `join` operator and `delete` should be replaced with `withColumnRenamed`

Question 31:

Which of the following code blocks reads a `/loc/example.csv` CSV file as a `df` DataFrame?

A. `df = spark.read.csv("/loc/example.csv")`

B. `df = spark.mode("csv").read("/loc/example.csv")`

C. `df = spark.read.path("/loc/example.csv")`

D. `df = spark.read().csv("/loc/example.csv")`

Question 32:

Which of the following code blocks reads a parquet file at the `my_path` location using a schema file named `my_schema`?

A. `spark.read.schema(my_schema).format("parquet").load(my_path)`

B. `spark.read.schema("my_schema").format("parquet").load(my_path)`

C. `spark.read.schema(my_schema).parquet(my_path)`

D. `spark.read.parquet(my_path).schema(my_schema)`

Question 33:

We want to find the number of records in the resulting DataFrame when we join the `employeedf` and `salarydf` DataFrames on the `employeeID` and `employeeSalaryID` columns respectively. Which code blocks should be executed to achieve this?

1. `.filter(~isnull(col(department)))`
2. `.count()`
3. `employeedf.join(salarydf, col("employeedf.employeeID")==col("salarydf.employeeSalaryID"))`
4. `employeedf.join(salarydf, employeedf. employeeID ==salarydf.employeeSalaryID, how='inner')`
5. `.filter(col(department).isnotnull())`

6. `.sum(col(department))`

   A. 3, 1, 6

   B. 3, 1, 2

   C. 4, 2

   D. 3, 5, 2

Question 34:

Which of the following code blocks returns a copy of the `df` DataFrame where the name of the `state` column has been changed to `stateID`?

A. `df.withColumnRenamed("state", "stateID")`

B. `df.withColumnRenamed("stateID", "state")`

C. `df.withColumn("state", "stateID")`

D. `df.withColumn("stateID", "state")`

Question 35:

Which of the following code blocks returns a copy of the `df` DataFrame where the `salary` column has been converted to `integer`?

A. `df.col("salary").cast("integer"))`

B. `df.withColumn("salary", col("salary").castType("integer"))`

C. `df.withColumn("salary", col("salary").convert("integerType()"))`

D. `df.withColumn("salary", col("salary").cast("integer"))`

Question 36:

Which of the following code blocks splits a `df` DataFrame in half with the exact same values even when the code is run multiple times?

A. `df.randomSplit([0.5, 0.5], seed=123)`

B. `df.split([0.5, 0.5], seed=123)`

C. `df.split([0.5, 0.5])`

D. `df.randomSplit([0.5, 0.5])`

Question 37:

Which of the following code blocks sorts the `df` DataFrame by two columns, `salary` and `department`, where `salary` is in ascending order and `department` is in descending order?

A. `df.sort("salary", asc("department"))`

B. `df.sort("salary", desc(department))`

C. `df.sort(col(salary)).desc(col(department))`

D. `df.sort("salary", desc("department"))`

Question 38:

Which of the following code blocks calculates the average of the `bonus` column from the `salaryDf` DataFrame and adds that in a new column called `average_bonus`?

A. `salaryDf.avg("bonus").alias("average_bonus"))`

B. `salaryDf.agg(avg("bonus").alias("average_bonus"))`

C. `salaryDf.agg(sum("bonus").alias("average_bonus"))`

D. `salaryDf.agg(average("bonus").alias("average_bonus"))`

Question 39:

Which of the following code blocks saves the `df` DataFrame in the `/FileStore/file.csv` location as a CSV file and throws an error if a file already exists in the location?

A. `df.write.mode("error").csv("/FileStore/file.csv")`

B. `df.write.mode.error.csv("/FileStore/file.csv")`

C. `df.write.mode("exception").csv("/FileStore/file.csv")`

D. `df.write.mode("exists").csv("/FileStore/file.csv")`

Question 40:

Which of the following code blocks reads the `my_csv.csv` CSV file located at `/my_path/` into a DataFrame?

A. `spark.read().mode("csv").path("/my_path/my_csv.csv")`

B. `spark.read.format("csv").path("/my_path/my_csv.csv")`

C. `spark.read("csv", "/my_path/my_csv.csv")`

D. `spark.read.csv("/my_path/my_csv.csv")`

Question 41:

Which of the following code blocks displays the top 100 rows of the `df` DataFrame, where the `salary` column is present, in descending order?

A. `df.sort(asc(value)).show(100)`

B. `df.sort(col("value")).show(100)`

C. `df.sort(col("value").desc()).show(100)`

D. `df.sort(col("value").asc()).print(100)`

Question 42:

Which of the following code blocks creates a DataFrame that shows the mean of the `salary` column of the `salaryDf` DataFrame based on the `department` and `state` columns, where `age` is greater than `35` and the returned DataFrame should be sorted in ascending order by the `employeeID` column such that there are no nulls in that column?

1. `salaryDf.filter(col("age") > 35)`
2. `.filter(col("employeeID")`
3. `.filter(col("employeeID").isNotNull())`
4. `.groupBy("department")`
5. `.groupBy("department", "state")`
6. `.agg(avg("salary").alias("mean_salary"))`
7. `.agg(average("salary").alias("mean_salary"))`
8. `.orderBy("employeeID")`

A. 1, 2, 5, 6, 8

B. 1, 3, 5, 6, 8

C. 1, 3, 6, 7, 8

D. 1, 2, 4, 6, 8

Question 43:

The following code block contains an error. The code block should return a new DataFrame without the `employee` and `salary` columns and with an additional `fixed_value` column, which has a value of `100`.

```
df.withColumnRenamed(fixed_value).drop('employee', 'salary')
```

Identify the error:

A. `withcolumnRenamed` should be replaced with `withcolumn` and the `lit()` function should be used to fill the `100` value

B. `withcolumnRenamed` should be replaced with `withcolumn`

C. `employee` and `salary` should be swapped in a `drop` function

D. The `lit()` function call is missing

Question 44:

Which of the following code blocks returns the basic statistics for numeric and string columns of the `df` DataFrame?

A. `df.describe()`

B. `df.detail()`

C. `df.head()`

D. `df.explain()`

Question 45:

Which of the following code blocks returns the top 5 rows of the `df` DataFrame?

A. `df.select(5)`

B. `df.head(5)`

C. `df.top(5)`

D. `df.show()`

Question 46:

Which of the following code blocks creates a new DataFrame with the `department`, `age`, and `salary` columns from the `df` DataFrame?

A. `df.select("department", "age", "salary")`

B. `df.drop("department", "age", "salary")`

C. `df.filter("department", "age", "salary")`

D. `df.where("department", "age", "salary")`

Question 47:

Which of the following code blocks creates a new DataFrame with three columns, `department`, `age`, and `max_salary`, which has the maximum salary for each employee from each department and each age group from the `df` DataFrame?

```
df.___1___ (["department", "age"]).___2___ (___3___ ("salary").
alias("max_salary"))
```

Identify the correct answer:

A.

i. `filter`

ii. `agg`

iii. `max`

B.

i. groupBy

ii. agg

iii. max

C.

i. filter

ii. agg

iii. sum

D.

i. groupBy
ii. agg
iii. sum

Question 48:

The following code block contains an error. The code block should return a new DataFrame, filtered by the rows, where the `salary` column is greater than or equal to `1000` in the `df` DataFrame.

```
df.filter(F(salary) >= 1000)
```

Identify the error:

A. Instead of `filter()`, `where()` should be used
B. The `F(salary)` operation should be replaced with `F.col("salary")`
C. Instead of >=, the > operator should be used
D. The argument to the `where` method should be `"salary > 1000"`

Question 49:

Which of the following code blocks returns a copy of the `df` DataFrame where the `department` column has been renamed `business_unit`?

A. `df.withColumn(["department", "business_unit"])`
B. `itemsDf.withColumn("department").alias("business_unit")`
C. `itemsDf.withColumnRenamed("department", "business_unit")`
D. `itemsDf.withColumnRenamed("business_unit", "department")`

Question 50:

Which of the following code blocks returns a DataFrame with the total count of employees in each department from the `df` DataFrame?

A. `df.groupBy("department").agg(count("*").alias("total_employees"))`

B. `df.filter("department").agg(count("*").alias("total_employees"))`

C. `df.groupBy("department").agg(sum("*").alias("total_employees"))`

D. `df.filter("department").agg(sum("*").alias("total_employees"))`

Question 51:

Which of the following code blocks returns a DataFrame with the `employee` column from the `df` DataFrame case to the `string` type?

A. `df.withColumn("employee", col("employee").cast_type("string"))`

B. `df.withColumn("employee", col("employee").cast("string"))`

C. `df.withColumn("employee", col("employee").cast_type("stringType()"))`

D. `df.withColumnRenamed("employee", col("employee").cast("string"))`

Question 52:

Which of the following code blocks returns a DataFrame with a new `fixed_value` column, which has `Z` in all rows in the df DataFrame?

A. `df.withColumn("fixed_value", F.lit("Z"))`

B. `df.withColumn("fixed_value", F("Z"))`

C. `df.withColumnRenamed("fixed_value", F.lit("Z"))`

D. `df.withColumnRenamed("fixed_value", lit("Z"))`

Question 53:

Which of the following code blocks returns a new DataFrame with a new `upper_string` column, which is the capitalized version of the `employeeName` column in the `df` DataFrame?

A. `df.withColumnRenamed('employeeName', upper(df.upper_string))`

B. `df.withColumnRenamed('upper_string', upper(df.employeeName))`

C. `df.withColumn('upper_string', upper(df.employeeName))`

D. `df.withColumn(' employeeName', upper(df.upper_string))`

Question 54:

The following code block contains an error. The code block is supposed to capitalize the employee names using a udf:

```
capitalize_udf = udf(lambda x: x.upper(), StringType())
df_with_capitalized_names = df.withColumn("capitalized_name",
capitalize("employee"))
```

Identify the error:

A. The `capitalize_udf` function should be called instead of `capitalize`

B. The `udf` function, `capitalize_udf`, is not capitalizing correctly

C. Instead of `StringType()`, `IntegerType()` should be used

D. Instead of `df.withColumn("capitalized_name", capitalize("employee"))`, it should use `df.withColumn("employee", capitalize("capitalized_name"))`

Question 55:

The following code block contains an error. The code block is supposed to sort the `df` DataFrame by salary in ascending order. Then, it should sort based on the `bonus` column, putting `nulls` last.

```
df.orderBy ('salary', asc_nulls_first(col('bonus')))
```

Identify the error:

A. The `salary` column should be sorted in descending order and `desc_nulls_last` should be used instead of `asc_nulls_first`. Moreover, it should be wrapped in a `col()` operator.

B. The `salary` column should be wrapped by the `col()` operator.

C. The `bonus` column should be sorted in a descending way, putting nulls last.

D. The `bonus` column should be sorted by `desc_nulls_first()` instead.

Question 56:

The following code block contains an error. The code block needs to group the `df` DataFrame based on the `department` column and calculate the total salary and average salary for each department.

```
df.filter("department").agg(sum("salary").alias("sum_salary"),
avg("salary").alias("avg_salary"))
```

Identify the error:

A. The `avg` method should also be called through the `agg` function
B. Instead of `filter`, `groupBy` should be used
C. The `agg` method syntax is incorrect
D. Instead of filtering on `department`, the code should filter on `salary`

Question 57:

Which code block will write the `df` DataFrame as a parquet file on the `filePath` path partitioning it on the `department` column?

A. `df.write.partitionBy("department").parquet(filePath)`
B. `df.write.partition("department").parquet(filePath)`
C. `df.write.parquet("department").partition(filePath)`
D. `df.write.coalesce("department").parquet(filePath)`

Question 58:

The `df` DataFrame contains columns `[employeeID, salary, department]`. Which of the following pieces of code would return the `df` DataFrame with only columns `[employeeID, salary]`?

A. `df.drop("department")`
B. `df.select(col(employeeID))`
C. `df.drop("department", "salary")`
D. `df.select("employeeID", "department")`

Question 59:

Which of the following code blocks returns a new DataFrame with the same columns as the `df` DataFrame, except for the `salary` column?

A. `df.drop(col("salary"))`
B. `df.delete(salary)`
C. `df.drop(salary)`
D. `df.delete("salary")`

Question 60:

The following code block contains an error. The code block should return the `df` DataFrame with `employeeID` renamed as `employeeIdColumn`.

```
df.withColumnRenamed("employeeIdColumn", "employeeID")
```

Identify the error:

A. Instead of `withColumnRenamed`, the `withColumn` method should be used

B. Instead of `withColumnRenamed`, the `withColumn` method should be used and the `"employeeIdColumn"` argument should be swapped with the `"employeeID"` argument

C. The `"employeeIdColumn"` and `"employeeID"` arguments should be swapped

D. `withColumnRenamed` is not a method for DataFrames

Answers

1. D
2. C
3. A
4. A
5. A
6. A
7. B
8. D
9. C
10. B
11. C
12. C
13. A
14. B
15. A
16. A
17. A
18. B
19. B

20. E
21. C
22. A
23. A
24. C
25. C
26. D
27. D
28. B
29. A
30. B
31. A
32. A
33. C
34. A
35. D
36. A
37. D
38. B
39. A
40. D
41. C
42. B
43. A
44. A
45. B
46. A
47. B
48. B
49. C
50. A
51. B

52. A
53. C
54. A
55. A
56. B
57. A
58. A
59. A
60. C

669FD03CAE074

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

669FD03CAE074

packtpub.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?

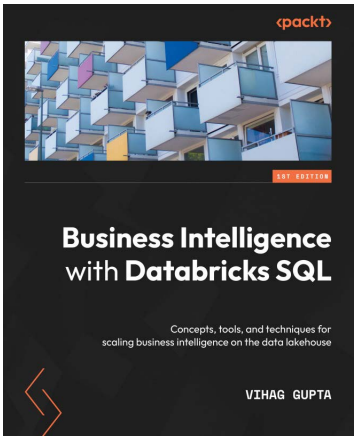
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, 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 packtpub.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.packtpub.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:

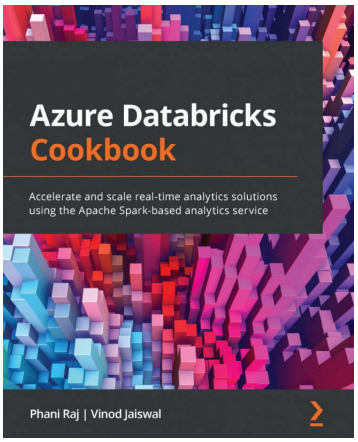


Business Intelligence with Databricks SQL

Vihag Gupta

ISBN: 978-1-80323-533-2

- Understand how Databricks SQL fits into the Databricks Lakehouse Platform
- Perform everyday analytics with Databricks SQL Workbench and business intelligence tools
- Organize and catalog your data assets
- Program the data security model to protect and govern your data
- Tune SQL warehouses (computing clusters) for optimal query experience
- Tune the Delta Lake storage format for maximum query performance
- Deliver extreme performance with the Photon query execution engine
- Implement advanced data ingestion patterns with Databricks SQL



Azure Databricks Cookbook

Phani Raj, Vinod Jaiswal

ISBN: 978-1-78980-971-8

- Read and write data from and to various Azure resources and file formats
- Build a modern data warehouse with Delta Tables and Azure Synapse Analytics
- Explore jobs, stages, and tasks and see how Spark lazy evaluation works
- Handle concurrent transactions and learn performance optimization in Delta tables
- Learn Databricks SQL and create real-time dashboards in Databricks SQL
- Integrate Azure DevOps for version control, deploying, and productionizing solutions with CI/CD pipelines
- Discover how to use RBAC and ACLs to restrict data access
- Build end-to-end data processing pipeline for near real-time data analytics

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

Once you've read *Databricks Certified Associate Developer for Apache Spark using Python*, 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.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781804619780`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly