SQL TO PYSPARK

CENTRICA TRAINING MATERIAL

ELEANOR HILTON HAMZAH JAVAID

ACKNOWLEDGEMENTS

First and foremost, I would like to extend my gratitude to Eleanor Hilton - without her, this book would not have been possible.

I would also like to extend my appreciation to my wider team for their support and encouragement throughout the writing process. Their knowledge and experience have been invaluable in ensuring that this book is accurate and informative.

I hope that this book will prove to be of great benefit to my team and others in their professional endeavours, and that it will serve as a valuable resource for anyone looking to expand their knowledge of SQL to PySpark.

Thank you for taking the time to read this book.

Hamzah

INTRODUCTION

What is PySpark

PySpark is a tour-de-force in the world of big data processing. As an interface for Apache Spark and the Python programming language, it provides a seamless and energetic way to tackle massive datasets with élan.

Its roots can be traced back to the early 2000s, when researchers at UC Berkeley's AMPLab developed Spark as a response to the limitations of Hadoop's MapReduce framework for big data processing. Since its inception, Spark has rapidly grown in popularity, and PySpark has emerged as a key tool for big data processing and analysis.

This makes it an ideal solution for organizations that are looking to process and analyze vast amounts of data, regardless of the source or format. The ease of use of the PySpark API, combined with its scalability and versatility, makes it a powerful tool for data engineers, data scientists, and big data analysts. Whether it's for data preprocessing, data exploration, or building machine learning models, PySpark provides a comprehensive and efficient solution for big data processing and analysis.

In terms of its technical advantages, PySpark offers a highly flexible and scalable architecture that enables organizations to adapt to changing requirements and handle increasing volumes of data. This scalability is a result of the Resilient Distributed Datasets (RDD) data structure, which allows for parallel processing and distribution across multiple nodes. Additionally, PySpark's compatibility with a wide range of data sources, including Hadoop HDFS and Apache Cassandra, makes it a versatile tool for big data processing and analysis.

PySpark also boasts a rich set of libraries and APIs for big data processing, including support for SQL, machine learning, and graph processing. This has led to its widespread use in the development of a wide range of big data applications, from simple data analysis and reporting to complex machine learning models and real-time data processing pipelines. The integration of PySpark with popular machine learning libraries such as Scikit-learn and TensorFlow has also made it possible for organizations to build and deploy state-of-the-art models on massive datasets.

In terms of its community and development, PySpark is an open-source platform that is actively maintained and developed by a large and vibrant community of contributors. This community has been instrumental in the development of the platform and has provided a wealth of resources and expertise to organizations looking to leverage PySpark for big data processing and analysis.

With PySpark, one can unleash the full potential of Apache Spark on large-scale datasets, thanks to the Resilient Distributed Datasets (RDD) data structure that enables parallel processing and distribution across multiple nodes. PySpark also boasts compatibility with a diverse array of data sources, from Hadoop HDFS to Apache Cassandra and beyond.

A brief history of Big Data

In the beginning, processing vast amounts of data was a monumental task, limited by the computing power and storage capacity of traditional systems. That all changed with the advent of Big Data and the technologies developed to harness its power.

In 2003, Doug Cutting, then at Yahoo, created the open-source project known as Hadoop. Inspired by a paper by Google on the concept of distributed file systems and data processing, Cutting sought to build a scalable, reliable, and cost-effective solution for managing Big Data. As he stated, "The idea behind Hadoop is to enable distributed processing of large data sets across clusters of computers using simple programming models."

Hadoop quickly gained popularity and set the stage for the development of other Big Data technologies, including MapReduce. The MapReduce programming model, first introduced by Google in 2004, provided a framework for processing large amounts of data in a parallel and distributed manner. As Jeff Dean, one of the original MapReduce developers at Google, put it, "MapReduce is a way to take a large amount of data, spread it across a large number of computers, and process it in parallel to get a result."

While Hadoop and MapReduce were groundbreaking developments, they were not without their challenges. The programming model was complex and difficult to learn, and the systems were not well suited for iterative or interactive processing, which is often necessary for advanced data analysis.

Enter Apache Spark. Created in response to the limitations of Hadoop and MapReduce, Spark was designed to be a fast, in-memory data processing engine that could handle both batch and streaming data. As Matei Zaharia, one of the original developers of Spark, stated, "The goal of Spark was to make it easier to build end-to-end big data applications, and to provide a more flexible and high-level programming model than MapReduce."

Since its creation in 2010, Spark has grown to become one of the most widely adopted Big Data processing technologies, with a thriving ecosystem of tools and libraries. Spark's performance and ease of use have made it a favorite among data scientists and engineers, and its ability to handle batch, streaming, and interactive data processing has made it a key component of many Big Data architectures.

The evolution of Big Data processing has been driven by a need for scalable, flexible, and powerful technologies. From the early days of Hadoop and MapReduce to the present-day dominance of Spark, each step in this journey has brought us closer to realizing the full potential of Big Data. The future of Big Data processing is bright, and we can be sure that the next chapter in this story will be just as exciting and impactful as the ones that came before.

Chapter 1: The Battle of the Big Data Giants: Hadoop vs Spark

Since the inception of Big Data, two heavyweights have dominated the arena of data processing: Apache Hadoop and Apache Spark. While both have made significant contributions to the field of Big Data, the debate over which is the superior technology continues to rage on.

Hadoop, created in 2003 by Doug Cutting, was the first open-source framework for distributed data processing. Its key feature is the Hadoop Distributed File System (HDFS), which enables large amounts of data to be stored across a cluster of commodity servers. As Cutting stated, "The idea behind Hadoop is to enable distributed processing of large data sets across clusters of computers using simple programming models."

MapReduce, a programming model introduced by Google in 2004, became the de facto method of processing data in Hadoop. MapReduce allowed developers to write code that could be parallelized and executed across a cluster of machines, making it possible to process massive amounts of data in a scalable and efficient manner.

However, despite its pioneering role in the world of Big Data, Hadoop faced significant limitations. The MapReduce programming model was complex and difficult to use, and Hadoop was not well suited for iterative or interactive processing, which is often required for advanced data analysis.

Enter Apache Spark, created in response to the limitations of Hadoop and MapReduce. Spark was designed to be a fast, in-memory data processing engine that could handle both batch and streaming data. As Matei Zaharia, one of the original developers of Spark, stated, "The goal of Spark was to make it easier to build end-to-end big data applications, and to provide a more flexible and high-level programming model than MapReduce."

Since its creation, Spark has grown to become one of the most widely adopted Big Data processing technologies, with a thriving ecosystem of tools and libraries. Spark's ease of use, performance, and ability to handle batch, streaming, and interactive data processing have made it a favorite among data scientists and engineers.

The battle of Hadoop vs Spark continues to rage on, with each technology having its own strengths and weaknesses. While Hadoop was the first to bring distributed data processing to the masses, Spark has since taken the lead with its in-memory processing capabilities and ease of use. Ultimately, the choice between Hadoop and Spark comes down to the specific needs and requirements of each organization, and the ability to choose the right tool for the job will continue to be a critical aspect of successful Big Data processing.

Differences between Hadoop and Spark

Hadoop and Spark are both popular technologies used for big data processing and analysis. However, there are several key differences between them that make them suitable for different use cases.

1. Architecture: Hadoop is a batch processing framework, which uses MapReduce to process large amounts of data in parallel. Spark, on the other hand, is a real-time data processing engine that can handle both batch processing and real-time processing using its in-memory capabilities. Spark can process data up to 100x faster than Hadoop MapReduce in memory, and up to 10x faster on disk.
2. In-Memory Processing: One of the key differences between Hadoop and Spark is the way they process data. Hadoop processes data on disk, while Spark processes data in memory. This allows Spark to process data much faster, making it suitable for real-time processing and interactive queries.
3. Cost: Hadoop and Spark also differ in terms of cost. Hadoop requires a large amount of hardware resources, which can be expensive. Spark, on the other hand, can run on a smaller cluster, reducing the cost of hardware resources. Additionally, Spark's in-memory processing capabilities also reduce the cost of I/O operations, making it a more cost-effective solution for big data processing.
4. Ease of Use: Hadoop can be more difficult to use, as it requires knowledge of Java programming and the MapReduce framework. Spark, on the other hand, provides a more user-friendly API that can be used with several programming languages, including Python, Scala, and R.

In conclusion, both Hadoop and Spark have their own strengths and weaknesses. Hadoop is a cost-effective solution for batch processing large amounts of data, while Spark is a more powerful and versatile solution for real-time data processing and analysis.

Chapter 2: In-memory processing

In-memory processing is a fundamental aspect of the Apache Spark computing framework that differentiates it from traditional MapReduce-based big data processing. It is a key factor that enables Spark to handle massive datasets with exceptional speed and efficiency.

To understand in-memory processing, we first need to understand the limitations of traditional big data processing frameworks such as Hadoop's MapReduce. In MapReduce, data is processed by writing it to disk between each processing step. This approach has several drawbacks, including the overhead of reading and writing large amounts of data to disk, slow I/O speeds, and limited memory capacity.

In contrast, Spark utilizes an in-memory processing model where intermediate data is stored in memory and not written to disk between processing steps. This approach dramatically reduces I/O overhead and enables Spark to perform much faster processing on large datasets. Spark uses a data structure known as Resilient Distributed Datasets (RDDs) to store intermediate data in memory. RDDs are partitioned across multiple nodes in a cluster, allowing for parallel processing and distribution.

The in-memory processing model of Spark is particularly advantageous for iterative algorithms, where intermediate data is processed repeatedly. In traditional MapReduce, such algorithms would require data to be written to disk and then read back into memory for each iteration, resulting in slow processing times. In Spark, the intermediate data is stored in memory, allowing for much faster processing times.

In-memory processing is also beneficial for machine learning algorithms, where data is often processed repeatedly to update model parameters. With Spark's in-memory processing, these algorithms can be executed much faster, enabling organizations to build and deploy state-of-the-art models on massive datasets.

In addition to the performance benefits of in-memory processing, Spark also provides a more flexible and expressive programming model than traditional MapReduce. With Spark, developers can write code in popular programming languages such as Python, Scala, and R, and leverage a rich set of libraries and APIs for big data processing, including support for SQL, machine learning, and graph processing.

In-memory processing is a key factor that sets Spark apart from traditional MapReduce-based big data processing frameworks. By storing intermediate data in memory, Spark enables faster and more efficient processing of massive datasets, making it a highly attractive platform for big data processing and analysis.

Chapter 3: The Great Debate: PySpark vs Python

In the world of Big Data, the conversation surrounding the best tools and technologies to use is never-ending. Among the most hotly contested debates is the comparison between PySpark and Python. While both have their own unique advantages, it can be challenging to determine which is the right choice for a given project.

Python has been a staple in the programming world for over two decades, and its popularity has only grown with the rise of Big Data. Python is a high-level, interpreted language that is renowned for its simplicity and ease of use. It is often referred to as the "Swiss Army Knife" of programming languages due to its versatility and the vast array of libraries and tools available for use.

Pyspark, on the other hand, is a data processing engine that is built on top of Apache Spark. Spark is a fast and general-purpose cluster computing system that can handle batch and streaming data processing. Pyspark extends the functionality of Spark by providing an interface for writing Spark applications in Python, making it easier for Python users to take advantage of the power of Spark.

One of the key cognitive differences between Python and Pyspark is the focus on speed and performance. While Python is a highly readable and user-friendly language, its performance can be limited when processing large amounts of data. Pyspark, on the other hand, is designed specifically for large-scale data processing and is optimized for performance. With Pyspark, data processing tasks can be parallelized across multiple machines, making it possible to process large amounts of data much faster than with Python alone.

Another important difference between these two technologies is the ease of use. Python is a general-purpose language that is designed to be easy to use for a wide range of programming tasks. Pyspark, on the other hand, is designed specifically for data processing and can be more challenging to use for those who are not familiar with Spark or distributed computing. However, for those who are familiar with Python and Spark, Pyspark provides a more intuitive and flexible interface for data processing than Spark alone.

When it comes to data processing and manipulation, Python and Pyspark both have their strengths and weaknesses. Python provides a high-level, user-friendly interface for data analysis and manipulation, while Pyspark provides a fast and efficient engine for large-scale data processing. Ultimately, the choice between Python and Pyspark will depend on the specific needs and requirements of each organization, and the ability to choose the right tool for the job will be critical to the success of any data processing project.

Whether you are a data scientist, engineer, or simply someone who is interested in processing and manipulating large amounts of data, understanding the benefits and limitations of both Python and Pyspark is essential to making the right choice for your specific needs.

Chapter 4: PySpark vs Spark: Unraveling the Connection

The world of Big Data processing has been greatly impacted by the arrival of Apache Spark and its corresponding Python API, PySpark. While both Spark and PySpark offer powerful and efficient data processing capabilities, the question of which is the superior technology continues to be a topic of debate among the data science community.

Apache Spark was first introduced in 2010 as a fast, in-memory data processing engine that could handle both batch and streaming data. Spark quickly gained popularity among data scientists and engineers due to its ease of use, performance, and ability to handle various types of data processing.

PySpark, on the other hand, is the Python API for Apache Spark. It allows data scientists and engineers to use Python, a widely used programming language in the data science community, to process data with Spark. PySpark enables users to write Spark applications in Python, making it easier for them to utilize the power of Spark for data processing and analysis.

While Spark and PySpark share many similarities, there are key differences between the two technologies that can impact the choice of which to use. For instance, Spark is written in Scala, a statically typed programming language that is known for its performance and efficiency. On the other hand, PySpark, as a Python API, allows for greater flexibility and ease of use, as Python is a dynamically typed language that is popular for its readability and simplicity.

In terms of performance, Spark, being a statically typed language, has a slight advantage over PySpark. However, PySpark compensates for this by offering more flexibility and ease of use, making it a popular choice among data scientists and engineers who are more comfortable with Python.

The choice between Spark and PySpark ultimately comes down to the specific needs and requirements of each organization. While Spark offers improved performance, PySpark provides greater ease of use and flexibility, making it a popular choice for data science projects. Regardless of the technology chosen, both Spark and PySpark have proven to be powerful and efficient tools for processing and analyzing Big Data.

Understanding RDDs, DataFrames, and Datasets in Apache Spark

Apache Spark provides three main APIs for working with data: RDDs (Resilient Distributed Datasets), DataFrames, and Datasets.

Each of these APIs provides a different level of abstraction and a different set of features, making them suitable for different use cases. In this section, we will take a closer look at each of these APIs and compare them to help you decide which one is right for your use case.

RDDs

RDDs are the basic building blocks of Apache Spark and the first API that was introduced. RDDs provide a low-level, distributed data structure that can be used to process large amounts of data in parallel. RDDs are immutable and partitioned, meaning that data can be processed in parallel across multiple nodes in a cluster.

RDDs are designed to be fast and efficient, but they can be difficult to work with, especially for more complex data manipulations. RDDs do not have a schema or a defined structure, so they require manual processing and data transformation to make the data usable.

The main use case for RDDs is for processing large amounts of data in parallel, especially when you need to perform custom data transformations that are not easily achievable using other APIs.

DataFrames

DataFrames were introduced in Spark 1.3 as a new API for working with structured data. DataFrames provide a higher-level abstraction than RDDs and are optimized for working with structured data. DataFrames have a schema, meaning that the data is organized into columns with specific data types. This makes it easier to work with the data and perform operations like filtering, aggregating, and joining data.

DataFrames are also designed to be fast and efficient, and they use the Spark SQL engine to perform optimizations for common operations like filtering and aggregating data. This means that DataFrames can be significantly faster than RDDs for many use cases, especially when working with structured data.

The main use case for DataFrames is for working with structured data, especially when you need to perform operations like filtering, aggregating, and joining data.

Datasets

Datasets were introduced in Spark 1.6 as a new API for working with typed, structured data. Datasets provide a higher-level abstraction than DataFrames and are designed to be more type-safe and performant. Datasets have a schema and are strongly typed, meaning that the data is organized into columns with specific data types and the type of each column is known at compile time.

Datasets provide the best of both worlds, combining the speed and efficiency of DataFrames with the type-safety and structure of RDDs. Datasets are optimized for working with typed, structured data, and they use the Spark SQL engine to perform optimizations for common operations like filtering and aggregating data.

The main use case for Datasets is for working with typed, structured data, especially when you need to perform operations like filtering, aggregating, and joining data and you want to take advantage of the type-safety and structure provided by Datasets.

RDDs, DataFrames, and Datasets provide different levels of abstraction and different sets of features, making them suitable for different use cases. RDDs are designed for processing large

DataFrame as the preferred option

DataFrames have become the preferred option for tabular data in PySpark for several reasons:

- Ease of Use: DataFrames have a user-friendly API that is easy to use and similar to other data analysis tools like Pandas, making them accessible to a wider range of users.
- Structured Data: DataFrames are designed for working with structured data, which is the most common type of data in data analysis. The structure of DataFrames allows for more efficient processing of data compared to RDDs, which are designed for unstructured data.
- Performance: DataFrames take advantage of optimizations like predicate pushdown, column pruning, and efficient encoding, making them faster than RDDs and more performant for tabular data.
- Compatibility with Spark SQL: DataFrames can be queried using Spark SQL, which is a powerful SQL engine that supports complex queries, aggregations, and joins. This compatibility makes it easy to integrate with other Spark tools and workflows.
- Support for Multiple Data Sources: DataFrames can be created from a variety of data sources, including Parquet, Avro, JSON, and CSV, making it easy to work with a wide range of data formats

DataFrames are the preferred option for tabular data in PySpark because they provide a balance of ease of use, performance, and functionality, making them the most versatile and powerful choice for data analysis in Spark.

Chapter 5: The Inner Mechanics of Apache Spark: A Comprehensive Guide

Apache Spark is a powerful, open-source, in-memory data processing engine that has become one of the most popular technologies for Big Data processing. But what exactly is Spark, and how does it work? To understand the inner workings of Spark, it's necessary to dive deep into its architecture and design principles.

At its core, Spark is a distributed data processing framework that enables users to perform complex data operations across large datasets, spanning multiple machines. As Sean Owen, Director of Data Science at Cloudera, stated, "Spark is designed to be fast, flexible, and user-friendly, making it a go-to choice for data engineers, data scientists, and developers alike."

Spark's architecture is built upon the idea of a Resilient Distributed Dataset (RDD), a fundamental data structure that allows data to be distributed across a cluster of machines. RDDs are the backbone of Spark's distributed processing capabilities, and they enable Spark to perform fast, in-memory computations on large datasets.

At the heart of Spark's architecture is its Master-Worker model, where the Master node coordinates the distribution of tasks across a cluster of Worker nodes. This architecture enables Spark to scale horizontally and process large amounts of data in parallel, making it possible to perform complex data operations in a fraction of the time it would take with traditional data processing techniques.

One of the key benefits of Spark is its ability to perform both batch and stream processing, making it a versatile solution for a wide range of data processing needs. With its built-in support for popular data sources such as Hadoop HDFS, Cassandra, and Kafka, Spark provides a unified data processing platform that can handle large amounts of structured and unstructured data, including batch, real-time, and historical data.

In addition to its powerful data processing capabilities, Spark also includes a rich set of libraries and APIs, including MLlib for machine learning, Spark SQL for structured data processing, and Spark Streaming for real-time data processing. These libraries and APIs make it possible for developers to build complex data processing applications with ease, and they have contributed to Spark's widespread adoption in the Big Data community.

The inner mechanics of Spark are a testament to its versatility and power as a data processing framework. With its scalable architecture, rich set of libraries and APIs, and ability to handle batch and stream processing, Spark is the go-to choice for organizations looking to perform complex data operations at scale. As Reynold Xin, Apache Spark Project Management Committee member, stated, "Spark's architecture and design principles are a perfect example of how technology can be both powerful and user-friendly, making it a top choice for organizations of all sizes."

Chapter 6: Spark Best Practice

Spark performance optimization is a crucial aspect of large-scale data processing and analysis. Here are three of the best practices that are important to keep in mind when using Spark for data processing:

1. Minimize data shuffling:

Partitioning of data in Spark is crucial to ensure optimal performance. When data is partitioned into smaller chunks, it allows for parallel processing of data, leading to faster processing times. It is important to choose an appropriate data partitioning strategy that balances the trade-off between network overhead and computational efficiency. A common approach is to partition the data based on the primary key or the grouping column, which can lead to evenly distributed data and reduce the risk of data skew.

2. Caching

Caching is an important mechanism to improve the performance of Spark applications. It involves storing the intermediate results of computation in memory, so that they can be reused in subsequent stages of the computation. This significantly reduces the I/O overhead and improves the performance of the Spark application. Caching can be used to persist the data frames, RDDs, and datasets that are used frequently in the computation, ensuring that the data is readily available for processing.

3. Executor memory configuration

Spark executors are responsible for processing the data in parallel, and each executor has its own memory configuration. It is important to configure the executor memory in such a way that it balances the trade-off between computation and memory overhead. If the executor memory is not configured correctly, it can lead to out of memory errors, and the Spark application may fail.

Additionally, it is important to allocate enough memory for caching, as well as for storing the intermediate results of the computation. The executor memory should be configured based on the size of the data being processed and the requirements of the Spark application.

These are just a few of the best practices that are important to keep in mind when using Spark for data processing.

Lesson 1: Getting started with PySpark DataFrames

Spark session and application name

The Spark session is the entry point to the Spark API and acts as the foundation for your Spark application. It allows you to create DataFrames and Datasets and manages the creation of the SparkContext. The SparkContext coordinates tasks across the cluster and connects to the cluster manager to allocate resources.

Providing a unique application name makes it easier to monitor the application and helps with logging, tracing, debugging and performance analysis.

SQL

```
#  Creating a SparkSession is not necessary in SQL, as you can directly issue
SQL commands to Spark

#  Read a CSV file into a Spark DataFrame
CREATE TEMPORARY TABLE data USING csv OPTIONS (path "data.csv", header "true"
, inferSchema "true");

#  Show the first 5 rows of the DataFrame
SELECT * FROM data LIMIT 5;

#  Print the schema of the DataFrame
DESCRIBE data;

#  Get the number of rows in the DataFrame
SELECT COUNT(*) FROM data;

#  Get the number of columns in the DataFrame. NB: you can directly look at t
he output of DESCRIBE

#  Describe the summary statistics of the DataFrames columns
SELECT MIN(age), MAX(age), AVG(age), SUM(age) FROM data;

#  Check if the DataFrame has any null values
SELECT COUNT(*) FROM data WHERE age IS NULL;
```

PySpark

```python
# Create a SparkSession
spark = SparkSession.builder.appName("SQLtoPySpark").getOrCreate()

# Read a CSV file into a PySpark DataFrame
df = spark.read.csv("data.csv", header=True, inferSchema=True)

# Show the first 5 rows of the DataFrame
df.show(5)

# Print the schema of the DataFrame
df.printSchema()

# Get the number of rows in the DataFrame
df.count()

# Get the number of columns in the DataFrame
len(df.columns)

# Describe the summary statistics of the DataFrame's columns
df.describe().show()

# Check if the DataFrame has any null values
df.filter(df.column.isNull()).count()
```

Lesson 2: Filtering and selecting data in PySpark

The process of selecting and filtering data in SQL and PySpark is similar in many ways.

In PySpark, you can perform the same operations using DataFrames and the select and filter methods – with the added advantage of being able to perform more complex operations, such as aggregation and group by, and to write custom functions in Python that can be applied to the data. Additionally, PySpark also provides support for multiple data sources, including Parquet.

SQL

```
#  Filter the DataFrame where the column "age" is greater than 30
SELECT * FROM data WHERE age > 30;

#  Select only the "name" and "age" columns from the DataFrame
SELECT name, age FROM data;

#  Select only the rows where the "income" column is greater than 100000
SELECT name, age, income FROM data WHERE income > 100000;

# Filter the DataFrame where the column "age" is greater than 30
SELECT * FROM data WHERE age > 30;

#  Select only the "name" and "age" columns from the DataFrame
SELECT name, age FROM data;

#  Select only the rows where the "income" column is greater than 100000
SELECT name, age, income FROM data WHERE income > 100000;
```

PySpark

```
# Filter the DataFrame where the column "age" is greater than 30
df_filtered = df.filter(df.age > 30)

# Show the first 5 rows of the filtered DataFrame
df_filtered.show(5)

# Select only the "name" and "age" columns from the DataFrame
df_selected = df.select(["name", "age"])

# Show the first 5 rows of the selected DataFrame
df_selected.show(5)

# Select only the rows where the "income" column is greater than 100000
df_selected = df.filter(df.income > 100000).select(["name", "age", "income"])

# Show the first 5 rows of the selected DataFrame
df_selected.show(5)
```

Lesson 3: Grouping and aggregating data in PySpark

When grouping and aggregating data in PySpark, the process is similar to that in SQL. However, PySpark provides more flexibility in terms of the functions and transformations that you can apply to your data. You can use the built-in functions or write your own custom functions in Python, which can be applied to your data using PySpark's powerful data processing capabilities.

Additionally, PySpark also supports the use of window functions, which can be used for advanced grouping and aggregating operations, such as calculating running totals or rankings.

SQL

```
# Group the DataFrame by the "age" column and count occurrences of each age
SELECT age, COUNT(*) FROM data GROUP BY age;

# Group the DataFrame by the "income" column and calculate the average age fo
r each income group
SELECT income, AVG(age) FROM data GROUP BY income;

# Group the DataFrame by the "age" column and calculate the total income for
each age group
SELECT age, SUM(income) FROM data GROUP BY age;
```

PySpark

```
# Group the DataFrame by the "age" column and count occurrences of each age
grouped = df.groupBy("age").count()

# Show the first 5 rows of the grouped DataFrame
grouped.show(5)

# Group the DataFrame by the "income" column and calculate the average age fo
r each income group
grouped = df.groupBy("income").agg(avg("age"))

# Show the first 5 rows of the grouped DataFrame
grouped.show(5)

# Group the DataFrame by the "age" column and calculate the total income for
each age group

grouped = df.groupBy("age").agg(sum("income"))

# Show the first 5 rows of the grouped DataFrame

grouped.show(5)
```

Lesson 4: Joining DataFrames in PySpark

When performing joins in PySpark, the process is similar to that in SQL. In both cases, you can join two or more tables based on one or more common columns to create a new table with combined data. The most commonly used join types in both PySpark and SQL are inner join, left join, right join, and full outer join. The syntax for joining tables in SQL typically involves using the JOIN keyword followed by the ON clause, which specifies the conditions for the join.

PySpark also provides advanced join capabilities, such as broadcast joins, which can be used to optimize the performance of joins for large datasets. Unlike SQL, PySpark also supports the use of join types beyond the traditional inner, left, right, and full outer join, such as cartesian product and cross join, which can be useful for specific use cases.

SQL

```
#Join the data table with the customer table on the customer_id column
SELECT data.*, customer.name
FROM data
JOIN customer ON data.customer_id = customer.customer_id;

#Inner join the data table with the customer table on the customer_id column
SELECT data.*, customer.name
FROM data
JOIN customer ON data.customer_id = customer.customer_id
WHERE data.age > 30;

#Left join the data table with the customer table on the customer_id column
SELECT data.*, customer.name
FROM data
LEFT JOIN customer ON data.customer_id = customer.customer_id;

#Right join the data table with the customer table on the customer_id column
SELECT data.*, customer.name
FROM data
RIGHT JOIN customer ON data.customer_id = customer.customer_id;

#Outer join the data table with the customer table on the customer_id column
SELECT data.*, customer.name
FROM data
FULL OUTER JOIN customer ON data.customer_id = customer.customer_id;
```

PySpark

```
# Join the data table with the customer table on the customer_id column
joined = df_data.join(df_customer, df_data["customer_id"] == df_customer["cus
tomer_id"])

# Show the first 5 rows of the joined DataFrame
joined.show(5)

# Inner join the data table with the customer table on the customer_id column
inner_joined = df_data.join(df_customer, df_data["customer_id"] == df_custome
r["customer_id"], "inner").filter(df_data["age"] > 30)

# Show the first 5 rows of the inner joined DataFrame
inner_joined.show(5)

# Left join the data table with the customer table on the customer_id column
left_joined = df_data.join(df_customer, df_data["customer_id"] == df_customer
["customer_id"], "left")

# Show the first 5 rows of the left joined DataFrame
left_joined.show(5)

# Right join the data table with the customer table on the customer_id column
right_joined = df_data.join(df_customer, df_data["customer_id"] == df_custome
r["customer_id"], "right")

# Show the first 5 rows of the right joined DataFrame
right_joined.show(5)

# Outer join the data table with the customer table on the customer_id column
outer_joined = df_data.join(df_customer, df_data["customer_id"] == df_custome
r["customer_id"], "outer")

# Show the first 5 rows of the outer joined DataFrame
outer_joined.show(5)
```

Lesson 5: Advanced Joining in PySpark I

PySpark provides advanced join capabilities that go beyond traditional inner, left, right, and full outer join operations, such as broadcast joins and cross joins. Additionally, PySpark offers a high-level API for joining dataframes that allows for more concise and readable code, as well as the ability to perform join operations using custom Python functions. This can make PySpark a better choice for more complex join operations compared to SQL.

SQL

```
# Join the data table with the customer table on multiple columns (customer_i
d and name)
SELECT data.*, customer.age
FROM data
JOIN customer ON data.customer_id = customer.customer_id AND data.name = cust
omer.name;

#| Left join the data table with the customer table on multiple columns (cust
omer_id and name)
SELECT data.*, customer.age
FROM data
LEFT JOIN customer ON data.customer_id = customer.customer_id AND data.name =
customer.name;

#| Right join the data table with the customer table on multiple columns (cus
tomer_id and name)
SELECT data.*, customer.age
FROM data
RIGHT JOIN customer ON data.customer_id = customer.customer_id AND data.name
= customer.name;

#| Outer join the data table with the customer table on multiple columns (cus
tomer_id and name)
SELECT data.*, customer.age
FROM data
FULL OUTER JOIN customer ON data.customer_id = customer.customer_id AND data.
name = customer.name;
```

```
# Join the data table with the customer table on multiple columns (customer_id and name)
joined = df_data.join(df_customer, (df_data["customer_id"] == df_customer["customer_id"]) & (df_data["name"] == df_customer["name"]))

# Show the first 5 rows of the joined DataFrame
joined.show(5)

# Left join the data table with the customer table on multiple columns (customer_id and name)
left_joined = df_data.join(df_customer, (df_data["customer_id"] == df_customer["customer_id"]) & (df_data["name"] == df_customer["name"]), "left")

# Show the first 5 rows of the left joined DataFrame
left_joined.show(5)

# Right join the data table with the customer table on multiple columns (customer_id and name)
right_joined = df_data.join(df_customer, (df_data["customer_id"] == df_customer["customer_id"]) & (df_data["name"] == df_customer["name"]), "right")

# Show the first 5 rows of the right joined DataFrame
right_joined.show(5)

# Outer join the data table with the customer table on multiple columns (customer_id and name)
outer_joined = df_data.join(df_customer, (df_data["customer_id"] == df_customer["customer_id"]) & (df_data["name"] == df_customer["name"]), "outer")

# Show the first 5 rows of the outer joined DataFrame
outer_joined.show(5)
```

Lesson 6: Subqueries in PySpark

In the group by and subquery operations are typically performed using the GROUP BY clause in a SELECT statement, followed by subqueries that are nested within the main query.

In terms of syntax in PySpark, to perform subqueries, you can use the .alias() method to create an alias for a dataframe, which can then be used as an input to another query. PySpark also supports the use of complex data structures, such as arrays and maps, which can be used to store and manipulate grouped data in a more flexible way.

PySpark also provides support for advanced grouping and aggregating operations, such as pivot tables and cube operations, which can be useful for specific use cases.

SQL

```
SELECT departments.department, employees.name, employees.salary
FROM
(SELECT department, AVG(salary) as avg_salary
      FROM employees
      GROUP BY department) departments
JOIN employees
ON departments.department = employees.department
WHERE departments.avg_salary > 75000 AND employees.salary > 80000;
```

PySpark

```
from pyspark.sql.functions import avg

df = spark.read.csv("employees.csv", header=True)

# Compute average salary by department
avg_salary = df.groupBy("department").agg(avg("salary").alias("avg_salary"))

# Filter departments with average salary greater than $75,000
departments = avg_salary.filter(avg_salary["avg_salary"] > 75000)

# Join with employees table and filter employees with salary greater than $80
,000
result = departments.join(df, "department").filter(df["salary"] > 80000)

result.show()
```

Lesson 7 : Advanced Sub-queries I

SQL and PySpark both handle advanced subqueries similarly, with slight syntax differences. In SQL, subqueries are nested within main queries and can perform a variety of operations such as aggregations, filtering, and transformations. PySpark's high-level API for subqueries allows for the same operations using a more concise and readable syntax, with dataframes being aliased for use in another query.

SQL

```
WITH department_salary AS (
  SELECT department, AVG(salary) as avg_salary
  FROM employees
  GROUP BY department
),
high_salary_department AS (
  SELECT department
  FROM department_salary
  WHERE avg_salary > 75000
)
SELECT employees.department, employees.name, employees.salary, department_sal
ary.avg_salary
FROM employees
JOIN high_salary_department
ON employees.department = high_salary_department.department
JOIN department_salary
ON employees.department = department_salary.department
WHERE employees.salary > 80000;
```

PySpark

```
from pyspark.sql.functions import avg
df = spark.read.csv("employees.csv", header=True)
# Compute average salary by department
department_salary = df.groupBy("department").agg(avg("salary").alias("avg_sal
ary"))

# Filter departments with average salary greater than $75,000
high_salary_department = department_salary.filter(department_salary["avg_sala
ry"] > 75000)

# Join with employees table and filter employees with salary greater than $80
,000
result = df.join(high_salary_department, "department").join(department_salary
, "department").filter(df["salary"] > 80000)

result.show()
```

Optimising Joins in PySpark

Broadcast join

When joining a small DataFrame with a large DataFrame, you can use broadcast join to improve performance. Broadcast join broadcasts the smaller DataFrame to each worker node so that it can be used by the join operation without having to be transferred across the network. You can use the broadcast function from the pyspark.sql.functions module to achieve this.

```
# Broadcast join
from pyspark.sql.functions import broadcast

# Use broadcast join when joining a small DataFrame with a large DataFrame
broadcast_joined = df_large.join(broadcast(df_small), "id")

# Show the first 5 rows of the broadcast joined DataFrame
broadcast_joined.show(5)
```

Sort Merge Join

Use sort merge join when joining two large DataFrames on a common key. This is because sort merge join sorts both DataFrames on the join key and then performs the join. Sort merge join is particularly effective when both DataFrames are sorted on the join key.

```
# Sort merge join
sort_merge_joined = df_large1.join(df_large2, "id").sortWithinPartitions("id"
)

# Show the first 5 rows of the sort merge joined DataFrame
sort_merge_joined.show(5)
```

Shuffle Hash Join

Use shuffle hash join when joining two large DataFrames that have a large number of unique join keys. This is because shuffle hash join uses a hash partitioner to distribute the data evenly among worker nodes. Shuffle hash join is effective when the join keys are well distributed among the partitions.

```
# Shuffle hash join
shuffle_hash_joined = df_large1.join(df_large2, "id").repartition(1000, "id")
.sortWithinPartitions("id")

# Show the first 5 rows of the shuffle hash joined DataFrame
shuffle_hash_joined.show(5)
```

Tune Parallelism

When joining large DataFrames, it's important to set the right number of partitions and parallelism level to achieve the best performance. You can use the repartition method to control the number of partitions and the spark.default.parallelism configuration property to control the parallelism level.

```
# Repartition DataFrames
df_large1 = df_large1.repartition(1000, "id")
df_large2 = df_large2.repartition(1000, "id")

# Set parallelism level
spark.conf.set("spark.default.parallelism", 1000)
```

Predicate Pushdown

Predicate pushdown is a technique to improve the performance of joins by reducing the amount of data that needs to be processed.

Predicate pushdown works by pushing the filtering conditions down to the source system to reduce the amount of data that is read into Spark. You can use the filter method in PySpark to apply filtering conditions before the join operation.

```
# Apply filtering conditions before join
filtered_df_large1 = df_large1.filter(df_large1.column_a > 10)
filtered_df_large2 = df_large2.filter(df_large2.column_b < 20)

# Join filtered DataFrames
filtered_joined = filtered_df_large1.join(filtered_df_large2, "id")

# Show the first 5 rows of the filtered joined DataFrame

filtered_joined.show(5)
```

Use Bucketing:

Bucketing is a technique used to optimize joins by partitioning data into smaller groups, or buckets, based on a hash value of one or more columns. When two DataFrames are bucketed on the same columns, Spark can efficiently perform joins between the buckets without having to shuffle the data.

```
# Bucket the DataFrames

bucketed_df_large1 = df_large1.write.bucketBy(1000, "id").sortBy("id").saveAs
Table("bucketed_df_large1")
bucketed_df_large2 = df_large2.write.bucketBy(1000, "id").sortBy("id").saveAs
Table("bucketed_df_large2")

# Load the bucketed DataFrames into Spark

bucketed_df_large1 = spark.table("bucketed_df_large1")
bucketed_df_large2 = spark.table("bucketed_df_large2")

# Join the bucketed DataFrames

bucketed_joined = bucketed_df_large1.join(bucketed_df_large2, "id")

# Show the first 5 rows of the bucketed joined DataFrame

bucketed_joined.show(5)
```

Use Broadcast Variables

Broadcast variables allow you to cache a read-only variable on each node in a cluster, reducing the amount of data that needs to be transferred over the network during a join. When joining a large DataFrame with a small DataFrame, you can use a broadcast variable to cache the small DataFrame on each node, improving the performance of the join.

```
# Convert the small DataFrame to a broadcast variable
broadcast_df_small = sc.broadcast(df_small.collect())

# Join the large DataFrame with the broadcast variable
broadcast_joined = df_large.rdd.map(lambda x: (x.id, x)).leftOuterJoin(broadc
ast_df_small.value, lambda x, y: x).map(lambda x: x[1][0])

# Convert the result back to a DataFrame
broadcast_joined_df = broadcast_joined.toDF(df_large.columns)

# Show the first 5 rows of the broadcast joined DataFrame
broadcast_joined_df.show(5)
```

Use Caching

Caching is a technique used to store the result of a query or computation in memory, allowing you to reuse it multiple times without having to recompute it each time. You can use caching to improve the performance of joins by reducing the amount of data that needs to be read from disk.

```
# Cache the large DataFrames
df_large1.cache()
df_large2.cache()

# Join the cached DataFrames
cached_joined = df_large1.join(df_large2, "id")

# Show the first 5 rows of the cached joined DataFrame
cached_joined.show(5)
```

Lesson 7: Complex PySpark Operations

Example 1

SQL:

```
SELECT c.id, c.name, c.age, c.city, c.country, o.order_id, o.product, o.quant
ity, o.price, o.date
FROM customers c
JOIN orders o ON c.id = o.customer_id
WHERE c.age > 30 AND c.country = 'USA' AND o.price > 100
GROUP BY c.id, o.product
HAVING COUNT(o.product) > 1
ORDER BY o.price DESC
LIMIT 10;
```

PySpark equivalent:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("ComplexSQL").getOrCreate()

customers = spark.read.csv("customers.csv", header=True)
orders = spark.read.csv("orders.csv", header=True)

join_query = customers.join(orders, customers.id == orders.customer_id) \
    .filter((customers.age > 30) & (customers.country == "USA") & (orders.pri
ce > 100)) \
    .groupBy(customers.id, orders.product) \
    .agg({"*": "count"}) \
    .filter("count(1) > 1") \
    .select(customers.id, customers.name, customers.age, customers.city, cust
omers.country,
            orders.order_id, orders.product, orders.quantity, orders.price, o
rders.date) \
    .orderBy(orders.price.desc()) \
    .limit(10)

join_query.show()
```

More optimized PySpark implementation:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("ComplexSQL").getOrCreate()

customers = spark.read.csv("customers.csv", header=True).filter("age > 30").f
ilter("country = 'USA'")
orders = spark.read.csv("orders.csv", header=True).filter("price > 100")

join_query = customers.join(orders, customers.id == orders.customer_id) \
    .groupBy(customers.id, orders.product) \
    .agg({"*": "count"}) \
    .filter("count(1) > 1") \
    .select(customers.id, customers.name, customers.age, customers.city, cust
omers.country,
            orders.order_id, orders.product, orders.quantity, orders.price, o
rders.date) \
    .orderBy(orders.price.desc()) \
    .limit(10)

join_query.show()
```

Example 2 - subqueries

SQL

```
SELECT *
FROM table1 t1
JOIN (
    SELECT col1, MAX(col2) as max_col2
    FROM table2
    GROUP BY col1
) t2
ON t1.col1 = t2.col1
JOIN (
    SELECT col3, SUM(col4) as sum_col4
    FROM table3
    GROUP BY col3
) t3
ON t1.col3 = t3.col3
WHERE t1.col5 = 'some value'
```

PySpark equivalent:

```
from pyspark.sql import functions as F

t1 = spark.table("table1")
t2 = spark.table("table2").groupBy("col1").agg(F.max("col2").alias("max_col2"
))
t3 = spark.table("table3").groupBy("col3").agg(F.sum("col4").alias("sum_col4"
))

t1.join(t2, on="col1", how="inner").join(t3, on="col3", how="inner")\
    .filter(col("col5") == "some value").select("*").show()
```

More optimized PySpark implementation:

```
from pyspark.sql import functions as F

t1 = spark.table("table1")
t2 = spark.table("table2").groupBy("col1").agg(F.max("col2").alias("max_col2"
))
t3 = spark.table("table3").groupBy("col3").agg(F.sum("col4").alias("sum_col4"
))

t1.createOrReplaceTempView("t1")
t2.createOrReplaceTempView("t2")
t3.createOrReplaceTempView("t3")

spark.sql("""
SELECT *
FROM t1
JOIN t2
ON t1.col1 = t2.col1
JOIN t3
ON t1.col3 = t3.col3
WHERE t1.col5 = 'some value'
""").show()
```

Example 3 – subqueries

SQL

```
SELECT
  customer_id,
  order_date,
  order_total,
  -- Subquery to calculate the average order total for each customer
  (SELECT AVG(order_total)
   FROM orders
   WHERE customer_id = o.customer_id) avg_order_total,
  -- Window function to calculate the running total of order_total for each c
ustomer
  SUM(order_total) OVER (PARTITION BY customer_id ORDER BY order_date) runnin
g_total
FROM orders o
```

PySpark

```
#
from pyspark.sql.functions import avg, sum
from pyspark.sql.window import Window

# Define window function
window = Window.partitionBy("customer_id").orderBy("order_date")

# Calculate running total using window function
df_running_total = df_orders.select(
  "customer_id",
  "order_date",
  "order_total",
  sum("order_total").over(window).alias("running_total")
)

# Join with average order total
df_result = df_running_total.join(
  df_orders.groupBy("customer_id").agg(avg("order_total").alias("avg_order_to
tal")),
  on="customer_id"
)
```

More optimized PySpark implementation:

```
from pyspark.sql.functions import avg, sum
from pyspark.sql.window import Window
from pyspark.sql.functions import broadcast

# Calculate average order total using SparkSQL
avg_order_total = df_orders.groupBy("customer_id").agg(avg("order_total").ali
as("avg_order_total"))

# Broadcast average order total data
broadcastAvgOrderTotal = broadcast(avg_order_total)

# Define window function
window = Window.partitionBy("customer_id").orderBy("order_date")

# Calculate running total using window function
df_running_total = df_orders.select(
  "customer_id",
  "order_date",
  "order_total",
  sum("order_total").over(window).alias("running_total")
)

# Join with average order total using broadcast variable
df_result = df_running_total.join(
  broadcastAvgOrderTotal,
  on="customer_id"
)
```

Example 4 - using CTEs

SQL

```
WITH recursive_cte (id, name, parent_id, level) AS (
  SELECT id, name, parent_id, 1
  FROM employees
  WHERE parent_id IS NULL
  UNION ALL
  SELECT e.id, e.name, e.parent_id, r.level + 1
  FROM employees e
  JOIN recursive_cte r ON e.parent_id = r.id
)
SELECT id, name,
  CASE WHEN level = 1 THEN 'Manager'
       WHEN level = 2 THEN 'Team Lead'
       ELSE 'Employee' END AS role
FROM recursive_cte;
```

PySpark

```
from pyspark.sql import functions as F
from pyspark.sql.window import Window

employees_df = spark.createDataFrame([
    (1, 'John', None),
    (2, 'Jane', 1),
    (3, 'Jim', 2),
    (4, 'Joan', 3),
    (5, 'Jake', 4)
], ['id', 'name', 'parent_id'])

window_spec = Window.orderBy('level')

result_df = employees_df.select('id', 'name', 'parent_id') \
    .union(employees_df.filter(F.col('parent_id').isNull()) \
           .withColumn('level', F.lit(1)) \
           .select('id', 'name', 'parent_id', 'level'))

for i in range(2, 20):
    result_df = result_df.union(result_df \
        .filter(F.col('level') == i - 1) \
        .join(employees_df, result_df.id == employees_df.parent_id) \
        .select(employees_df.id, employees_df.name, employees_df.parent_id, F
.lit(i).alias('level')))
```

```
result_df = result_df.withColumn('role', F.when(F.col('level') == 1, 'Manager
') \
                                 .when(F.col('level') == 2, 'Team Lead') \
                                 .otherwise('Employee'))
result_df.show()
```

More optimized PySpark implementation:

```
from pyspark.sql import functions as F

employees_df = spark.createDataFrame([
    (1, 'John', None),
    (2, 'Jane', 1),
    (3, 'Jim', 2),
    (4, 'Joan', 3),
    (5, 'Jake', 4)
], ['id', 'name', 'parent_id'])

result_df = employees_df.filter(F.col('parent_id').isNull()) \
    .withColumn('level', F.lit(1)) \
    .select('id', 'name', 'parent_id', 'level')

for i in range(2, 20):
    result_df = result_df.union
```

Example 5 - using pivots

SQL

```
WITH data_cte AS (
    SELECT
        order_id,
        product,
        order_date,
        CASE
            WHEN product = 'book' THEN 'books'
            WHEN product = 'music' THEN 'music'
            ELSE 'others'
        END AS category
    FROM orders
),
pivot_cte AS (
    SELECT
        order_id,
        order_date,
        SUM(CASE WHEN category = 'books' THEN 1 ELSE 0 END) AS books,
        SUM(CASE WHEN category = 'music' THEN 1 ELSE 0 END) AS music,
        SUM(CASE WHEN category = 'others' THEN 1 ELSE 0 END) AS others
    FROM data_cte
    GROUP BY order_id, order_date
)
SELECT
    order_date,
    SUM(books) AS total_books,
    SUM(music) AS total_music,
    SUM(others) AS total_others
FROM pivot_cte
GROUP BY order_date
ORDER BY order_date DESC;
```

PySpark

```
from pyspark.sql.functions import when, sum
from pyspark.sql.window import Window

data = (
    spark.table("orders")
    .select(
        "order_id",
        "product",
        "order_date",
        when(col("product") == "book", "books").when(col("product") == "music
", "music").otherwise("others").alias("category")
    )
)

pivot = (
    data.groupBy("order_id", "order_date")
    .agg(
        sum(when(col("category") == "books", 1).otherwise(0)).alias("books"),
        sum(when(col("category") == "music", 1).otherwise(0)).alias("music"),
        sum(when(col("category") == "others", 1).otherwise(0)).alias("others"
),
    )
)

result = (
    pivot
    .groupBy("order_date")
    .agg(
        sum(col("books")).alias("total_books"),
        sum(col("music")).alias("total_music"),
        sum(col("others")).alias("total_others"),
    )
    .sort(desc("order_date"))
)
```

Example 6 - using rank() function

SQL

```
WITH ranked_data AS (
  SELECT *, RANK() OVER (PARTITION BY col1 ORDER BY col2 DESC) AS rank_col
  FROM data_table
)
SELECT *
FROM ranked_data
WHERE rank_col <= 5
```

PySpark

```
from pyspark.sql.functions import rank, dense_rank

ranked_data = data_table.select("*", dense_rank().over(Window.partitionBy("co
l1").orderBy(col("col2").desc())).alias("rank_col"))

top_5 = ranked_data.filter(ranked_data["rank_col"] <= 5)
top_5.show()
```

More optimized PySpark implementation:

```
from pyspark.sql.functions import dense_rank

ranked_data = data_table.select("col1", "col2").sort(col("col2").desc()).grou
pBy("col1").agg(dense_rank().alias("rank_col"))

top_5 = ranked_data.filter(ranked_data["rank_col"] <= 5)
top_5.show()
```

Example 7 - using user defined functions (udfs)

SQL code:

```
WITH sales AS (
  SELECT order_id, product_id, quantity, unit_price, order_date
  FROM sales
)
SELECT
  order_date,
  product_id,
  SUM(quantity * unit_price) AS total_sales,
  AVG(quantity * unit_price) AS average_sale
FROM sales
GROUP BY order_date, product_id
```

Pyspark code:

```
from pyspark.sql.functions import udf, sum, avg
from pyspark.sql.types import FloatType

sales = spark.read.format("delta").load("/mnt/delta/sales")

def calculate_total_sales(quantity, unit_price):
  return quantity * unit_price

calculate_total_sales_udf = udf(calculate_total_sales, FloatType())

sales_df = (sales
  .withColumn("total_sales", calculate_total_sales_udf(sales.quantity, sales.
unit_price))
  .groupBy("order_date", "product_id")
  .agg(sum("total_sales").alias("total_sales"), avg("total_sales").alias("ave
rage_sale")))

sales_df.write.format("delta").save("/mnt/delta/grouped_sales")
```

PySpark on Databricks

PySpark is the Python API for Apache Spark, an open-source big data processing framework. It provides a high-level API for distributed data processing and supports various programming languages including Python, Scala, and Java. By using PySpark, data scientists and engineers can leverage the power of Spark without having to write complex code in Scala or Java.

Databricks is a cloud-based platform that allows you to easily run Apache Spark jobs in a collaborative environment. It provides an interactive workspace and a comprehensive set of tools and services that simplify Spark development and management. By using PySpark on Databricks, you can take advantage of the following benefits:

Scalability and Performance:

Spark is known for its ability to process large amounts of data in parallel, making it possible to handle big data challenges. By using Databricks, you can take advantage of its cloud-based architecture to easily scale your Spark clusters and process data at scale.

Ease of Use:

PySpark on Databricks provides a high-level API that makes it easy to write complex data processing logic without having to worry about the underlying infrastructure. It also provides an interactive workspace, where you can easily write, run, and debug your Spark code.

Collaboration:

Databricks provides a collaborative environment for data scientists, engineers, and business analysts to work together on big data projects. It allows you to share code, notebooks, and data with your team, and collaborate in real-time.

Monitoring and Management:

Databricks provides a comprehensive set of tools and services for monitoring and managing Spark jobs. You can easily track the progress of your Spark jobs, view detailed performance metrics, and receive notifications in case of failures.

Delta Lake on Databricks

Delta Lake is an open-source data lake format that brings ACID transactions to data lakes. It provides a unified and transactional storage layer that allows you to manage big data at scale and with high reliability. By using Delta Lake on Databricks, you can take advantage of the following benefits:

Data Reliability: Delta Lake provides ACID transactions, which ensure that your data remains consistent and accurate, even in the face of failures or concurrent updates.

Data Versioning: Delta Lake provides a versioning system that allows you to keep track of changes to your data over time. You can easily revert to previous versions of your data, or compare different versions to understand how your data has evolved.

Schema Evolution: Delta Lake allows you to change the schema of your data lake tables, even after data has been written to them. This makes it easy to accommodate new data sources and evolve your data models over time.

Performance: Delta Lake provides optimized storage and indexing that make it possible to perform fast analytics on large data sets. It also provides batch and streaming integration, making it possible to process data in real-time.

Integration: Delta Lake integrates seamlessly with Spark, making it easy to use within your PySpark workflows. It also provides support for Spark SQL, allowing you to query your data using SQL.

Lesson 8: PySpark with Delta Tables

SQL

```
WITH customers AS (
  SELECT customer_id, name, address, city, state, zip
  FROM customers
),
orders AS (
  SELECT order_id, customer_id, order_date, order_total
  FROM orders
),
order_details AS (
  SELECT order_id, product_id, quantity, unit_price
  FROM order_details
),
products AS (
  SELECT product_id, product_name, category
  FROM products
)
SELECT
  customers.name AS customer_name,
  products.category AS product_category,
  SUM(order_details.quantity * order_details.unit_price) AS total_sales
FROM
  customers
  LEFT JOIN orders ON customers.customer_id = orders.customer_id
  LEFT JOIN order_details ON orders.order_id = order_details.order_id
  LEFT JOIN products ON order_details.product_id = products.product_id
GROUP BY
  customer_name, product_category
```

PySpark

```
from pyspark.sql.functions import sum, col

customers = spark.read.format("delta").load("/mnt/delta/customers")
orders = spark.read.format("delta").load("/mnt/delta/orders")
order_details = spark.read.format("delta").load("/mnt/delta/order_details")
products = spark.read.format("delta").load("/mnt/delta/products")

joined_df = (customers
  .join(orders, customers.customer_id == orders.customer_id, "left")
  .join(order_details, orders.order_id == order_details.order_id, "left")
  .join(products, order_details.product_id == products.product_id, "left")
  .groupBy(customers.name.alias("customer_name"), products.category.alias("pr
oduct_category"))
  .agg(sum(col("quantity") * col("unit_price")).alias("total_sales")))

joined_df.write.format("delta").save("/mnt/delta/joined_tables")
```

PySpark with Delta Tables II

SQL

```
WITH customers AS (
  SELECT customer_id, name, address, city, state, zip
  FROM customers
),
orders AS (
  SELECT order_id, customer_id, order_date, order_total
  FROM orders
),
order_details AS (
  SELECT order_id, product_id, quantity, unit_price
  FROM order_details
),
products AS (
  SELECT product_id, product_name, category
  FROM products
)
SELECT
  customers.name AS customer_name,
  products.category AS product_category,
  SUM(order_details.quantity * order_details.unit_price) AS total_sales,
  MIN(orders.order_date) AS first_order_date,
  MAX(orders.order_date) AS last_order_date
FROM
  customers
  LEFT JOIN orders ON customers.customer_id = orders.customer_id
  LEFT JOIN order_details ON orders.order_id = order_details.order_id
  LEFT JOIN products ON order_details.product_id = products.product_id
GROUP BY
  customer_name, product_category
```

PySpark

```
from pyspark.sql.functions import sum, min, max, col

customers = spark.read.format("delta").load("/mnt/delta/customers")
orders = spark.read.format("delta").load("/mnt/delta/orders")
order_details = spark.read.format("delta").load("/mnt/delta/order_details")
products = spark.read.format("delta").load("/mnt/delta/products")

joined_df = (customers
  .join(orders, customers.customer_id == orders.customer_id, "left")
  .join(order_details, orders.order_id == order_details.order_id, "left")
  .join(products, order_details.product_id == products.product_id, "left")
  .groupBy(customers.name.alias("customer_name"), products.category.alias("pr
oduct_category"))
  .agg(sum(col("quantity") * col("unit_price")).alias("total_sales"),
       min(col("order_date")).alias("first_order_date"),
       max(col("order_date")).alias("last_order_date")))

# Use Delta Lake's time travel feature to view the results as of a specific v
ersion
joined_df.write.format("delta").mode("overwrite").save("/mnt/delta/joined_tab
les")
spark.sql("""
  SELECT *
  FROM delta.`/mnt/delta/joined_tables`
  AS OF TIMESTAMP '2022-12-31 12:00:00'
""").show()

# Use Delta Lake's schema evolution feature to add a new column to the Delta
table
from pyspark.sql.functions import lit
joined_df = joined_df.withColumn("year", lit("2023"))
joined_df.write.format
```

Using the Pandas API with PySpark

The PySpark library provides a powerful and flexible toolset for working with big data, but sometimes it can be difficult to work with when the data is particularly complex or requires advanced data manipulation. In these cases, it can be helpful to use the Pandas API in combination with PySpark to gain more fine-grained control over the data.

The Pandas API provides a number of powerful data manipulation and analysis functions, which are especially useful for working with tabular data. When combined with PySpark, these functions can be used to perform complex data manipulations and analysis on big data, making it possible to perform data manipulations that would be difficult or impossible to perform using only PySpark.

To use the Pandas API with PySpark, you will need to convert the PySpark dataframe into a Pandas dataframe using the toPandas method. Once you have converted the data, you can perform any number of Pandas operations on the data, including filtering, aggregating, grouping, and transforming the data. Once you have finished working with the data, you can convert the Pandas dataframe back into a PySpark dataframe using the createDataFrame method.

Here is a simple example that demonstrates how to use the Pandas API with PySpark:

Pandas API

```
from pyspark.sql import SparkSession
import pandas as pd

# Create a Spark session
spark = SparkSession.builder.appName("PandasExample").getOrCreate()

# Load a PySpark dataframe
df = spark.read.csv("data.csv", header=True, inferSchema=True)

# Convert the PySpark dataframe to a Pandas dataframe
pandas_df = df.toPandas()

# Filter the Pandas dataframe using a Pandas operation
filtered_df = pandas_df[pandas_df['age'] > 30]

# Group the Pandas dataframe using a Pandas operation
grouped_df = filtered_df.groupby(['gender'])['age'].mean()

# Convert the Pandas dataframe back to a PySpark dataframe
spark_df = spark.createDataFrame(grouped_df.reset_index())

# Show the results
spark_df.show()
```