Introduction to Data Science
CA3 - Report
1403/01/30

Amirreza Akbari | 810899045
Reza Baghestani | 810899046
Hananeh Jamali | 810899053

Purpose:

The purpose of this assignment is to familiarize ourselves with PySpark, a powerful tool for large-scale data processing, and apply its functionalities to analyze real-world datasets. Through a series of warm-up exercises and a main task focusing on Spotify song data, we aim to demonstrate our proficiency in using PySpark for data manipulation, aggregation, and analysis.

Summary:

This report presents the results of our exploration with PySpark, covering both warm-up exercises and the main task involving Spotify song data. We begin by installing and setting up PySpark, followed by a series of warm-up exercises analyzing a stocks dataset. Subsequently, we delve into the main task, where we analyze the Spotify dataset to extract useful statistics and insights.

Datasets Used:

1. **Stocks Dataset (stocks.csv):** This dataset contains information about stock market performance, including opening, closing, highest, and lowest prices, along with volume traded for each day. It serves as the basis for our warm-up exercises, helping us familiarize ourselves with PySpark's functionalities.

2. **Spotify Dataset (spotify.parquet):** The Spotify dataset comprises information about songs streamed on Spotify, including attributes like album, artist, musical characteristics, and release date. This dataset forms the core of our main task, allowing us to extract meaningful insights about song characteristics and trends.

Warm-Up:

Step 1: Reading the CSV file: We utilize SparkSession's read.csv() method to read the stocks dataset from the CSV file into a DataFrame. This step is crucial for accessing the dataset and initiating the analysis process.

Step 2: Understanding the Schema: We print the schema of the DataFrame using printSchema() to understand the structure and data types of the dataset. This helps in gaining insights into the columns present in the dataset.

Steps 3 and 4: Filtering Data: We filter the DataFrame to select records based on specific conditions using filter() and select() operations. This enables us to focus on subsets of data that meet predefined criteria, such as closing prices less than 500 or opening prices greater than 200 and closing prices less than 200.

Step 5: Extracting Year from Date: We use the withColumn() function to create a new column named 'Year' by extracting the year from the 'Date' column. This transformation facilitates analysis on an annual basis, aiding in trend identification.

Amirreza Akbari | 810899045
Reza Baghestani | 810899046
Hananeh Jamali | 810899053

Step 6: Aggregating Data: We group the DataFrame by the 'Year' column and compute the minimum volume traded for each year using groupBy() and min() functions. This aggregation provides insights into the lowest trading volumes recorded annually.

Step 7: Further Aggregation: We extend the aggregation to include both 'Year' and 'Month' columns and calculate the highest low price for each combination. This additional analysis offers granularity in identifying lowest prices within specific time periods.

Step 8: Summary Statistics: We compute the mean and standard deviation of the 'High' price over the entire dataset using select(), mean(), and stddev() functions. These summary statistics offer insights into the central tendency and dispersion of high prices.

Main Task:

Step 1: Checking the Schema: We start by creating a SparkSession and reading the provided Spotify dataset in Parquet format into a DataFrame. Printing the schema allows us to understand the structure and data types of the dataset, providing insights into the available columns and their formats.

Step 2: Preprocessing Columns: In this step, we preprocess the 'release_date' column by converting it to the date type using the withColumn() function. This conversion ensures that the 'release_date' column is in a suitable format for further analysis.

Step 3: Aggregation, Filtering, and Transformation: We perform aggregation, filtering, and transformation operations to extract useful statistics from the dataset:

- Aggregating statistics for danceability and energy features by year enables us to understand the average and variation of these characteristics over time.
- Filtering the dataset to exclude songs with explicit content ensures that the analysis focuses on non-explicit songs only.
- Converting the duration from milliseconds to minutes and creating a binary feature 'long_song' based on a specified threshold (e.g., 15 minutes) enhances the dataset with additional insights about song length.
- Calculating the number of long songs in the dataset provides information about the prevalence of longer-duration songs.

Step 4: Dealing with Array Columns: We handle array columns, such as 'artists', by splitting the string and creating an array to separate each artist. We then explode the array to obtain each artist in a separate row, facilitating analysis at the artist level. Finally, we drop the intermediate column to maintain data integrity.

Step 5: Top-K Records: To identify the top songs based on a specific feature (e.g., valence), we use the orderBy() function to sort the DataFrame in descending order and the limit() function to retrieve the top K records. This analysis provides insights into the most positively or negatively perceived songs.

Amirreza Akbari | 810899045
Reza Baghestani | 810899046
Hananeh Jamali | 810899053

Questions:

Question 1) How Hadoop & Sparks work and Lazy Evaluation

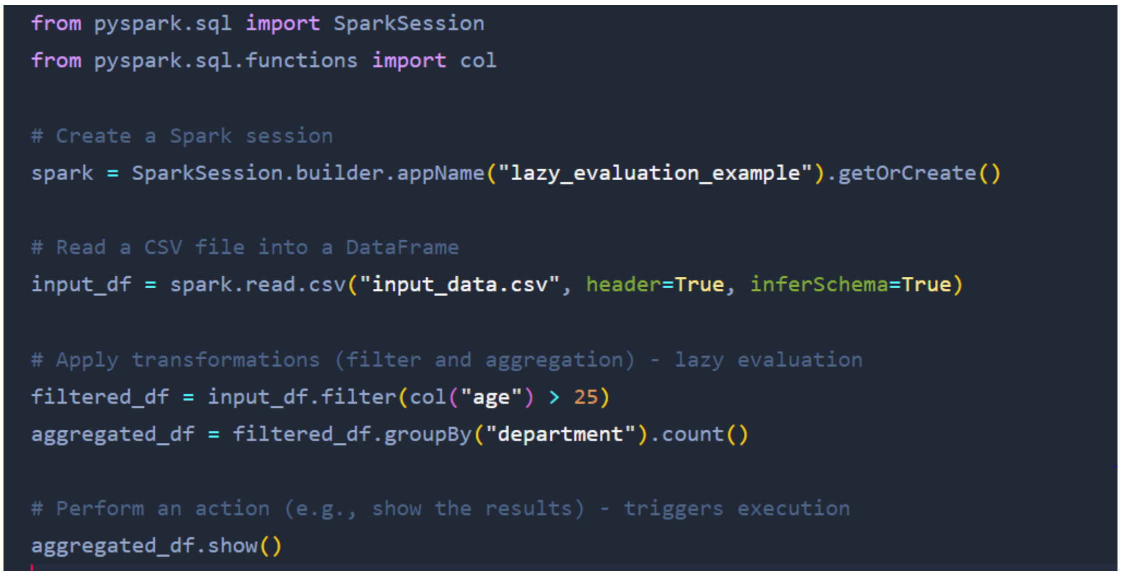
They are both popular big data processing frameworks, but they have different architectures and use cases. Hadoop is well-suited for batch processing and long-running jobs, while Spark is preferred for interactive, iterative, and real-time processing tasks

1. Hadoop is designed for distributed storage and processing of large datasets across clusters of computers. It consists of **Hadoop Distributed File System (HDFS)** that stores data across multiple nodes in a Hadoop cluster. It provides high availability and fault tolerance by replicating data blocks across nodes. **Yet Another Resource Negotiator (YARN)** which is the resource management layer in Hadoop that manages resources and schedules tasks across the cluster. It allows different data processing engines to run on the same Hadoop cluster. **MapReduce** is a programming model for processing and generating large datasets in parallel across a Hadoop cluster. It consists of two main phases: Map phase for processing input data and generating key-value pairs, and Reduce phase for aggregating and summarizing the intermediate data.

2. Spark is a fast and general-purpose cluster computing system that provides in-memory processing capabilities for big data workloads. It includes **Spark Core** that provides distributed task scheduling, fault recovery, and data sharing among different parallel computations. **Spark SQL**: a module provides support for querying data using SQL, DataFrame API, and integration with external data sources. **Spark Streaming** enables real-time processing of streaming data by dividing data streams into micro-batches and processing them using Spark's batch processing capabilities. Spark uses a directed acyclic graph (DAG) execution engine to optimize data processing workflows.

lazy evaluation" refers to a computational model where Instead of executing an operation immediately when it is defined, lazy evaluation defers the computation until the result is required by another operation to optimize performance and resource utilization. It allows these frameworks to build a directed acyclic graph (DAG) of transformations and only execute the computations when an action is triggered.

Here's a simple example We define two transformations: filtered_df and aggregated_df which are lazy are not executed immediately. and only executed when an action is called
(e.g., show(), collect(), write, etc.).



```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col

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

# Read a CSV file into a DataFrame
input_df = spark.read.csv("input_data.csv", header=True, inferSchema=True)

# Apply transformations (filter and aggregation) - lazy evaluation
filtered_df = input_df.filter(col("age") > 25)
aggregated_df = filtered_df.groupBy("department").count()

# Perform an action (e.g., show the results) - triggers execution
aggregated_df.show()
```

Question 2) Parquet files solve problem with large data

In contrast of CSV files, Parquet files store data in a columnar format, which means that values from the same column are stored together, this approach allows for more efficient compression and encoding techniques to be applied on a per-column basis. It reduces the amount of data that needs to be read from disk. For analytical queries that typically involve scanning only a subset of columns, Parquet files can skip reading irrelevant columns.

Parquet files support various compression algorithms results in better compression ratios compared to compressing the entire file as a whole, as is typically done with CSV files. It reduces storage requirements and can also improve read performance by minimizing the amount of data that needs to be read from disk.

In contrast of a CSV file Parquet files include metadata that describes the schema of the data, making them self-describing. This feature allows for schema evolution, meaning that new columns can be added or existing ones modified without rewriting the entire dataset.

Parquet files support predicate pushdown, enabling query engines to push filters down to the storage layer. This optimization reduces the amount of data that needs to be read from disk by skipping irrelevant data blocks. CSV files do not have built-in support for predicate pushdown, so query engines may have to read the entire file and then filter the data in memory, which can be inefficient for large datasets.

Question 3) Check Points!

Here's how you can enforce Spark to save checkpoints in your Spark application:

To enable checkpointing in Spark, you need to call the checkpoint() method on the RDD or DataFrame that represents an intermediate result that you want to save. This will trigger the actual checkpointing process when the action is executed. You need to specify a checkpoint directory

Amirreza Akbari | 810899045
Reza Baghestani | 810899046
Hananeh Jamali | 810899053

where Spark will save the checkpoint data. This directory should be on a reliable storage system that is accessible to all nodes in the Spark cluster.

Set Checkpoint Directory in Spark Configuration:

```
val spark = SparkSession.builder()
  .appName("CheckpointExample")
  .config("spark.sql.streaming.checkpointLocation", "hdfs://path/to/checkpoint/dir")
  .getOrCreate()
```

Trigger Checkpointing:

```
val intermediateRDD = sourceRDD.map(...)
intermediateRDD.checkpoint()
intermediateRDD.count()  // Trigger action to save checkpoint
```

Question 4) Faster Filtering

By leveraging Spark Structured Streaming with Parquet file format, partitioning by date, and efficient filtering using Spark SQL, you can save and filter streaming data based on specific columns faster and more efficiently than regular filtering methods.

Question 5) Pandas vs PySpark

Data Size: *Pandas* is well-suited for handling medium-sized datasets that can fit into memory. It is optimized for single-node processing and works efficiently with datasets that are a few gigabytes in size. *PySpark* is designed for processing large-scale datasets that exceed the memory capacity of a single machine. It can distribute computations across a cluster of machines, making it suitable for big data processing.

Processing Complexity: *Pandas* is great for exploratory data analysis, data cleaning, and manipulation tasks. It provides a rich set of functions and operations for data wrangling, grouping, filtering, and visualization. *PySpark* is ideal for handling complex data processing tasks, such as large-scale aggregations, joins, and machine learning on big data

User Experience: Pandas offers a user-friendly and intuitive interface that makes it easy for data scientists and analysts to work with tabular data. It provides a DataFrame data structure that closely resembles working with a spreadsheet, making it accessible to users familiar with tools like Excel. While *PySpark* offers scalability and performance benefits for big data processing, its learning curve can be steeper compared to Pandas. Users need to understand distributed computing concepts and Spark's API to leverage its full potential.