**SUMMARY-DAY13**

**Name:** Podutur Lahari

**Roll no:**DE126

**Date:**26-11-2024

-------------------------------------------------------------------------------------------------------------------------------------

**PySpark ETL Workflow**

**1. What is PySpark?**

* **PySpark** is the Python API for **Apache Spark**, an open-source distributed computing system designed for big data processing. PySpark provides an interface for working with **Spark's core functionalities** using Python, enabling developers to handle large-scale data with ease. It supports a variety of tasks such as ETL (Extract, Transform, Load), machine learning, and real-time data streaming.

**2. Key Reasons to Use PySpark**:

* **Performance**:
  + PySpark leverages **in-memory computing**, which significantly speeds up ETL tasks. Spark stores intermediate results in memory, which reduces the need for writing data to disk, improving the performance of tasks such as filtering, sorting, and aggregation.
* **Ease of Use**:
  + PySpark uses **Python**, a user-friendly language with a simple syntax. Python developers can easily transition to PySpark without needing to learn a new programming language. PySpark’s API is designed to be intuitive and familiar to Python programmers.
* **Scalability**:
  + PySpark works with **distributed computing**. This means it can process vast amounts of data across multiple nodes in a cluster. Whether you are working with terabytes or petabytes of data, Spark can scale seamlessly to meet the demands.
* **Rich Ecosystem**:
  + PySpark integrates with other popular **data tools** such as **Hadoop** (for distributed storage), **Kafka** (for real-time data streaming), **Hive** (for data warehousing), and **MLlib** (for machine learning). This makes PySpark a versatile tool that fits into a wide variety of data processing and analytics workflows.

**3. PySpark ETL Process**:

* The **ETL process** is a fundamental part of data engineering, where data is extracted, transformed, and loaded into a system for analysis or reporting. Here’s a breakdown of the key steps:

**a. Extract**:

* The **Extract** step involves pulling data from various sources like databases, flat files (e.g., CSV, JSON), and external systems or APIs. PySpark provides an easy interface to load data from different formats into a DataFrame.

df = spark.read.csv("source\_path", header=True, schema="cust\_id int, first\_name string, last\_name string")

Here, the data is extracted from a CSV file, and PySpark automatically handles the schema (i.e., data types for columns).

**b. Transform**:

* In the **Transform** step, data is cleaned, manipulated, and aggregated to meet the needs of the analysis. Transformations can include:
  + **Data cleaning** (e.g., removing duplicates, handling missing values).
  + **Feature engineering** (e.g., creating new columns based on existing data).
  + **Data manipulation** (e.g., filtering, grouping, sorting, joining).
  + **Aggregation** (e.g., summing values, calculating averages).
  + **Sorting and filtering** data based on conditions (e.g., age > 30). Example:

# Concatenate first and last names into a full name

df = df.withColumn("full\_name", concat(col("first\_name"), lit(" "), col("last\_name")))

# Add net salary column by calculating 90% of the salary (after tax deduction)

df = df.withColumn("net\_salary", floor(lit(10000) + rand() \* lit(50)))

# Filter records where age is greater than or equal to 30

df = df.filter(col("age") >= 30)

**c. Load**:

* In the **Load** step, the transformed data is saved into a target storage system (e.g., a relational database, data warehouse, or file system like HDFS or S3). PySpark supports various output formats, including CSV, Parquet, and Delta Lake.
* df.write.csv("target\_path", mode="overwrite", header=True)

In this example, the transformed data is written to a CSV file. You can also save it in other formats like Parquet for optimized storage and query performance.

**4. Real-World Applications of PySpark ETL**:

* **a. Data Cleansing and Preparation**:
  + **Scenario**: A financial institution gathers customer transaction data from various channels like online banking, ATMs, and payment gateways. The data might contain errors, missing values, and duplicates that need to be cleaned before analysis.
  + **PySpark ETL Steps**: Extract raw transaction data, clean it (remove duplicates, fill missing values), and transform it into a clean format for reporting and analysis.
  + **Example**:

df\_cleaned = df.dropDuplicates().fillna({'transaction\_amount': 0})

df\_cleaned.write.csv("cleaned\_data\_path", mode="overwrite", header=True)

* **b. Real-Time Data Processing**:
  + **Scenario**: A smart city project collects real-time data from traffic sensors (e.g., vehicle count, speed) to manage traffic flow. This data needs to be processed in real-time.
  + **PySpark ETL Steps**: Use **Structured Streaming** to ingest, process, and analyze real-time data.
  + **Example**:

# Stream data from Kafka or socket

stream\_df = spark.readStream.schema(schema).json("path\_to\_streaming\_data")

# Filter records with speed > 80 km/h

transformed\_stream = stream\_df.filter(col("speed") > 80)

# Output the processed stream to the console

query = transformed\_stream.writeStream.outputMode("append").format("console").start()

query.awaitTermination()

* **c. Data Pipelines for Machine Learning**:
  + **Scenario**: A retail company wants to create a product recommendation system using customer interaction data (e.g., clicks, purchases).
  + **PySpark ETL Steps**: Extract customer data, transform it (e.g., one-hot encode categorical variables), and prepare it for feeding into a machine learning model.
  + **Example**:

from pyspark.ml.feature import StringIndexer, VectorAssembler

# Encode categorical variables

indexer = StringIndexer(inputCol="product\_id", outputCol="product\_id\_index")

indexed\_df = indexer.fit(df).transform(df)

# Prepare features for the ML model

assembler = VectorAssembler(inputCols=["product\_id\_index", "user\_rating"], outputCol="features")

final\_df = assembler.transform(indexed\_df)

final\_df.write.parquet("path\_to\_ml\_data")

* **d. Data Aggregation and Reporting**:
  + **Scenario**: A sales company wants to aggregate sales data by region and product category to generate monthly reports for executives.
  + **PySpark ETL Steps**: Aggregate sales by region, calculate total sales, and store the report in a data warehouse.
  + **Example**:

sales\_df = spark.read.csv("sales\_data.csv", header=True, schema="product\_id int, region string, sales\_amount double, date string")

# Aggregate by region and product category

sales\_report = sales\_df.groupBy("region", "product\_id").agg({"sales\_amount": "sum", "sales\_amount": "avg"})

# Write the report to a target location

sales\_report.write.parquet("sales\_report\_path")

**5. Benefits of PySpark in Real-World Scenarios**:

* **Efficient Big Data Processing**: PySpark’s distributed nature makes it ideal for handling and processing large datasets that cannot fit into a single machine’s memory.
* **Real-Time Analytics**: With PySpark’s streaming capabilities, you can perform real-time data processing, such as monitoring live traffic or stock prices.
* **Machine Learning Pipelines**: PySpark allows data engineers to preprocess data efficiently and prepare it for machine learning models, helping data scientists to build scalable models.
* **Ease of Use**: PySpark’s simple API (especially for Python developers) makes it easy to integrate into existing workflows without requiring a steep learning curve.