**Lesson Proper for Week 7**

Completion requirements

**SPARK PROGRAMMING WITH PYTHON**

**1. Introduction to PySpark**

* **Background and Importance**:
  + Apache Spark was originally developed to overcome limitations of the traditional MapReduce computing model in Hadoop. It provides in-memory processing capabilities and supports real-time data processing, making it much faster than traditional batch processing frameworks.
  + PySpark, the Python API for Spark, allows data engineers and data scientists to write Spark applications using Python. It leverages Spark’s high-performance engine to handle tasks such as data analysis, data transformation, and machine learning at scale.
* **PySpark Ecosystem Overview**:
  + **Spark Core**: The underlying engine that provides functionalities like task scheduling, memory management, fault recovery, and interacting with storage systems.
  + **Spark SQL**: Supports querying of structured data with SQL-like syntax, allowing for data manipulation through DataFrames and Datasets.
  + **MLlib**: Spark's machine learning library for creating scalable ML pipelines.
  + **GraphX**: For graph processing.
  + **Spark Streaming**: For real-time data processing.
* **Benefits of Using PySpark**:
  + Unified big data processing framework.
  + Supports a wide range of tasks such as ETL, machine learning, and real-time stream processing.
  + Seamlessly integrates with Hadoop, HDFS, Hive, and other big data technologies.

**2. Understanding Spark Context**

* **What is SparkContext?**
  + The SparkContext object is the entry point for all PySpark applications. It is responsible for establishing a connection with the cluster, creating RDDs, and managing resources for job execution.
* **Spark Architecture Basics**:
  + **Driver Program**: Where the SparkContext is initialized. It coordinates tasks and collects results.
  + **Cluster Manager**: Allocates resources across applications. Spark supports several cluster managers, including standalone, YARN, and Mesos.
  + **Workers**: Execute tasks on data partitions and report results back to the driver.
* **Creating a SparkContext**:
  + When creating a SparkContext, you specify configurations such as the master URL (local or cluster), application name, memory settings, etc.

from pyspark import SparkContext

sc = SparkContext(master="local[\*]", appName="MyApp")

**3. Exploring SQL Context in Spark**

* **SQLContext vs. SparkSession**:
  + SQLContext was the primary entry point for working with structured data in earlier Spark versions. In more recent versions, SparkSession has replaced it as the new entry point that combines the functionalities of both SQLContext and HiveContext.
* **DataFrames and Spark SQL Integration**:
  + DataFrames represent structured data and are organized into named columns. They are similar to tables in a relational database and are built on top of RDDs.
  + SQLContext allows running SQL queries on DataFrames, joining tables, and performing aggregations.
* **Example: Reading JSON and Running SQL Queries**:

from pyspark.sql import SQLContext

sqlContext = SQLContext(sc)

df = sqlContext.read.json("path/to/data.json")

df.createOrReplaceTempView("people")

result = sqlContext.sql("SELECT \* FROM people WHERE age > 30")

result.show()

**Lesson Proper for Week 8**

Completion requirements

**PYSPARK - RDD**

**1. Introduction to PySpark RDDs**

* **RDDs (Resilient Distributed Datasets)**:
  + The foundational building block in Spark, RDDs are immutable, distributed collections of objects that can be operated on in parallel.
* **Types of RDD Operations**:
  + **Transformations** (e.g., map, filter, flatMap, join, etc.): Create a new RDD from an existing RDD. Transformations are lazily evaluated, meaning they are not executed until an action is called.
  + **Actions** (e.g., collect, count, reduce, saveAsTextFile): Trigger the actual execution of the transformations to produce a result.

**2. Key Features of RDDs**

* **Immutability**:
  + RDDs are read-only, meaning any transformation applied to an RDD will generate a new RDD. This immutability ensures fault tolerance and simplifies parallel computing.
* **Lazy Evaluation**:
  + Spark optimizes RDD execution through a concept known as lazy evaluation, where transformations are not executed until an action triggers computation.
* **Partitioning**:
  + RDDs can be divided across partitions, allowing data to be processed in parallel across the cluster.
* **Lineage Graph**:
  + Spark maintains a lineage graph of RDD transformations, enabling automatic recovery from node failures.

**3. Methods to Create RDDs in PySpark**

* **From Parallelized Collections**:
  + Useful for testing and small datasets. Creates an RDD from an existing Python collection.

data = [1, 2, 3, 4, 5]

rdd = sc.parallelize(data)

* **From External Data Sources**:
  + Load data from external sources such as HDFS, S3, or the local file system.

rdd = sc.textFile("hdfs://path/to/data.txt")

* **From Transformations on Existing RDDs**:
  + RDDs can be derived from other RDDs by applying transformations.

**Lesson Proper for Week 9**

Completion requirements

**LOADING AND STORING DATA**

***1. Common File Formats in Big Data***

* **Text Formats (CSV, TSV):**
  + Text files store data as plain text, with columns separated by a delimiter. Though simple, they are less efficient in terms of space and processing speed.
* **JSON:**
  + Stores data in a nested, semi-structured format. It is flexible and ly used in web applications but can be slower to parse.
* **Columnar Formats (Parquet, ORC):**
  + Designed for efficient storage and read-heavy operations. They store data column-wise, which reduces I/O and speeds up queries for selected columns.
* **Avro:**
  + A row-based storage format that supports schema evolution. It's used for data serialization and communication between systems.

***2. Hadoop Input and Output Formats***

* **Hadoop Input Formats:**
  + Spark can read data from Hadoop-based storage systems (HDFS, Amazon S3, etc.) and supports different file formats like SequenceFiles, Avro, and Parquet.
* **Hadoop Output Formats:**
  + Spark allows data to be written back to Hadoop-based storage systems, often in the same formats as those supported for input.

***3. Managing Structured Data with Spark SQL***

* **DataFrame Operations:**
  + With DataFrames, you can apply various transformations such as filtering, grouping, joining, and aggregating data, similar to traditional SQL.
* **Example: Filtering and Aggregating Data in Spark SQL:**

python

Copy code

df.filter(df.age > 30).groupBy("occupation").count().show() 

***4. Introduction to Apache Hive***

* What is Hive?
  + Hive provides a data warehouse solution that allows querying and managing large datasets residing in distributed storage using a SQL-like language (HiveQL).
* Hive Integration with Spark:
  + Spark can read from and write to Hive tables, allowing users to leverage their existing Hive data warehouse infrastructure.

Top of Form

**Lesson Proper for Week 10**

Completion requirements

**ADVANCED SPARK PROGRAMMING**

**1. Understanding Accumulators**

* **Accumulators Overview**:
  + Special variables used for aggregating values across the cluster. They are useful for counters and sums, allowing Spark to collect global statistics.
* **Accumulator Fault Tolerance**:
  + Spark’s task retry mechanism can affect accumulators. If a task fails and is retried, the accumulator will be updated again, potentially leading to double counting.

**2. Creating Custom Accumulators**

* **Steps for Creating Custom Accumulators**:
  + Define a new accumulator type.
  + Implement custom accumulation logic.

from pyspark import AccumulatorParam

**3. Using Broadcast Variables in Spark**

* **When to Use Broadcast Variables**:
  + Broadcast variables are used to cache a read-only dataset in memory on all worker nodes, reducing the need for repeated data transfer across the network.
* **Example**:

broadcastVar = sc.broadcast([1, 2, 3, 4, 5])

**ADVANCED SPARK PROGRAMMING 2**

***1. Optimizing the Use of Broadcast Variables***

* Benefits of Broadcast Variables:
  + Reduce network I/O by distributing data only once.
  + Efficiently share read-only data across tasks.

***2. Piping Data to External Programs***

* Integration with External Tools:
  + Allows Spark to pipe data through command-line programs.

rdd.pipe("wc -l").collect() 

***3. Performing Numeric Operations on RDDs***

* Aggregation Operations:
  + Supports operations like sum(), average(), min(), max(), etc.

**Lesson Proper for Week 13**

Completion requirements

**SPARK SQL**

**1. Introduction to Apache Spark SQL**

* **Capabilities of Spark SQL**:
  + Spark SQL provides a powerful engine for structured data processing, allowing users to perform data transformations using both SQL queries and DataFrame APIs.
  + It enables integration with Hive for accessing existing data stored in Hive tables.
  + Supports various data sources such as JSON, Parquet, and ORC, allowing users to read data in multiple formats.
  + Allows optimization of queries through Catalyst, Spark's built-in query optimizer.

**2. Interfaces in Apache Spark SQL**

* **SQLContext and HiveContext**:
  + As mentioned earlier, SQLContext allows users to perform SQL-like operations on DataFrames, while HiveContext provides support for Hive-compatible queries.
* **SparkSession**:
  + SparkSession is now the standard entry point for working with Spark SQL and DataFrames. It provides a unified interface for interacting with Spark.

from pyspark.sql import SparkSession

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

df = spark.read.json("data.json")

df.createOrReplaceTempView("people")

result = spark.sql("SELECT name FROM people WHERE age > 30")

**3. Key Features of Apache Spark SQL**

* **Support for Standard SQL Operations**:
  + Spark SQL allows querying structured data using familiar SQL syntax.
* **Interoperability with RDDs**:
  + It is possible to convert RDDs to DataFrames and vice versa, enabling seamless integration between different APIs.
* **Optimized Execution Plans**:
  + The Catalyst optimizer helps in generating efficient query execution plans, reducing the processing time for large datasets.

**4. Understanding Spark SQL DataFrames and Datasets**

* **DataFrames vs. Datasets**:
  + A **DataFrame** is a collection of data organized into named columns, similar to a table in a relational database.
  + A **Dataset** is a type-safe version of a DataFrame and allows for compile-time type checking. It is more common in Scala/Java but also accessible in Python.

**5. Features of Spark DataFrames and Datasets**

* **Lazy Execution**:
  + Like RDDs, DataFrames use lazy evaluation to optimize query execution.
* **Optimized for Performance**:
  + Supports columnar storage formats (e.g., Parquet), making Spark SQL queries highly efficient.
* **Ease of Use**:
  + DataFrames come with built-in functions for common data manipulation tasks, such as filtering, aggregations, and joins.

**6. Creating DataFrames and Datasets in Spark**

* **Creating DataFrames**:

df = spark.read.json("data.json")

* **Creating Datasets (Scala/Java)**:
  + In Scala, a Dataset is created using as[T] where T is a type.

**Lesson Proper for Week 14**

Completion requirements

**TEMP TABLES / VIEWS - EASY QUERYING**

**1. Creating Temporary Tables in Spark SQL**

* **Temporary Tables**:
  + Temporary tables allow users to register a DataFrame as a table, enabling the use of SQL-like queries without creating permanent tables.
* **Creating Temporary Tables Example**:

df.createOrReplaceTempView("temp\_people")

spark.sql("SELECT \* FROM temp\_people WHERE age > 25").show()

* **Difference Between Temp Views and Global Temp Views**:
  + A **Temp View** is session-scoped, meaning it is only available in the current session.
  + A **Global Temp View** is accessible across multiple Spark sessions.

**2. Building a DataFrame in Spark**

* **Creating DataFrames from Different Sources**:
  + DataFrames can be created from files, RDDs, or other DataFrames, allowing flexible data integration.

# From a CSV file

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

**3. Creating DataSource Tables**

* **Registering External Data Sources**:
  + Spark SQL allows registering external data sources as tables, providing a way to access various data formats using SQL queries.

spark.sql("CREATE TABLE people USING parquet OPTIONS (path 'hdfs://path/to/parquet')")

**4. Creating Views in Spark SQL**

* **Using Views for Query Abstraction**:
  + Views can be created to encapsulate complex queries, making them easier to use and maintain.

spark.sql("CREATE VIEW young\_people AS SELECT \* FROM people WHERE age < 30")

**Lesson Proper for Week 15**

Completion requirements

**SPARK STREAMING**

**1. Introduction to Spark Streaming**

* **Real-Time Data Processing**:
  + Spark Streaming is a scalable and fault-tolerant stream processing system that processes live data streams.
  + It breaks the data stream into small batches (micro-batching) and processes each batch sequentially.
* **Use Cases**:
  + Real-time data analytics.
  + Monitoring and alerting systems.
  + ETL operations on streaming data.

**2. Key Components of Streaming Architecture**

* **DStreams (Discretized Streams)**:
  + A DStream is a sequence of RDDs representing data received in batches.
* **Receivers**:
  + Components that receive data from a source and store it in Spark's memory for processing.
* **Window Operations**:
  + Allow data to be aggregated over a sliding time window, useful for computing trends.

**3. Benefits of Discretized Stream Processing**

* **Fault Tolerance**:
  + If a node fails, Spark can recompute lost data using the lineage information from previous RDDs.
* **Scalability**:
  + Spark Streaming can scale to handle high throughput and large datasets.

**4. Fundamentals of Spark Streaming**

* **Creating a Spark Streaming Context**:
  + To use Spark Streaming, a StreamingContext must be created, specifying the batch interval.

from pyspark.streaming import StreamingContext

ssc = StreamingContext(sc, 5)  # 5-second batch interval

**Lesson Proper for Week 16**

Completion requirements

**MACHINE LEARNING**

**1. Introduction to Machine Learning**

* **What is Machine Learning?**
  + Machine learning involves building algorithms that can learn from and make predictions on data.
* **Types of Machine Learning**:
  + **Supervised Learning**: Training models on labeled data (e.g., classification, regression).
  + **Unsupervised Learning**: Finding patterns in unlabeled data (e.g., clustering, anomaly detection).

**2. Common Machine Learning Techniques**

* **Regression Analysis**: Predicting continuous values (e.g., house prices).
* **Classification**: Predicting categorical labels (e.g., spam detection).
* **Clustering**: Grouping similar data points (e.g., customer segmentation).

**3. Implementing Machine Learning with Spark**

* **Using MLlib for Model Training**:
  + MLlib is Spark's built-in library for scalable machine learning.
* **Example**:

from pyspark.ml.classification import LogisticRegression

lr = LogisticRegression(featuresCol='features', labelCol='label')

model = lr.fit(trainingData)

**Lesson Proper for Week 17**

Completion requirements

**MACHINE LEARNING 2**

**1. Building a Machine Learning App with Apache Spark MLlib**

* **Combining Data Preprocessing, Model Training, and Evaluation**:
  + MLlib enables the construction of end-to-end machine learning pipelines, including data transformation, model training, and evaluation.

**2. Understanding Classification and Logistic Regression**

* **Logistic Regression Overview**:
  + A statistical method for binary classification that models the probability of a binary outcome.
* **Training a Logistic Regression Model**:

from pyspark.ml.classification import LogisticRegression

lr = LogisticRegression(featuresCol='features', labelCol='label')

model = lr.fit(trainingData)

**3. Creating a Machine Learning Model in Apache Spark**

* **Steps to Build a Model**:
  + Data Preprocessing (e.g., feature scaling).
  + Model Training (e.g., fitting a classifier).
  + Evaluation (e.g., calculating accuracy).

**4. Constructing the Input DataFrame for Modeling**

* **Preparing Data for ML Algorithms**:
  + Transform raw data into a format suitable for machine learning, typically involving converting features into a numerical vector.

**from** pyspark.ml.feature **import** VectorAssembler

assembler = VectorAssembler(inputCols=[**"col1"**, **"col2"**], outputCol=**"features"**)

**transformed\_data = assembler.transform(raw\_data)**

**By understanding these topics in depth, you'll gain a comprehensive foundation in PySpark and Spark's ecosystem, which will enable you to handle a variety of data processing, streaming, and machine learning tasks effectively.**