PySpark Training Course

Saeed Shirazi

January 2025

Course Outline

General Concepts and Introduction to PySpark

- General Concepts
- What is PySpark?
- Spark Architecture
- Installing PySpark on a Local OS
- Data in Spark
- Cluster Types

PySpark Basics

- Introduction to Resilient Distributed Datasets (RDDs)
- RDD Transformations
- RDD Actions
- Key-Value RDD Operations
- Partitioning Basics
- Hands-On Practice
- Broadcast and Accumulators (Intro)

PySpark DataFrames Basics

- Introduction to DataFrames
- Creating DataFrames
- Basic DataFrame Operations
- Understanding Schemas

Advanced DataFrame Operation

- Working with SQL in PySpark
- User-Defined Functions (UDFs)
- Joining DataFrames
- Window Functions
- Optimization Insights

PySpark Streaming

- Introduction to Spark Streaming
- Creating a Streaming DataFrame
- Processing Streaming Data
- Stateful Streaming
- Checkpointing
- Writing Processed Data

Optimization and Deployment

- Spark UI Overview
- PySpark Performance Optimization
- Parameter Tuning
- Writing Data to External Storage
- Running PySpark Applications on a Cluster
- Real-World Deployment
- Extra Materials!

General Concepts and Introduction to PySpark

Introduction to Big Data and Distributed Computing

- **Big Data:**
  - Extremely large datasets that are challenging to process using traditional methods.
- **Distributed Computing:**
  - Splitting data and computations across multiple machines.
  - Benefits:
    - Scalability
    - Fault tolerance
    - Faster processing

Challenges in Traditional Data Processing Systems

- **Scalability:** Unable to handle growing data volumes.
- **Fault Tolerance:** High risk of failure without recovery mechanisms.
- **Performance:** Sequential processing leads to high latency.
- **Flexibility:** Limited support for unstructured or semi-structured data.

What is Apache Spark?

- **Spark History:**
  - Apache Hadoop (2006): MapReduce for batch processing.
  - Apache Spark (2009): Overcame Hadoop’s limitations.
- **Advantages of Spark Over Traditional Frameworks:**
  - **Speed:** In-memory processing for faster execution.
  - **Ease of Use:** Intuitive APIs in Python, Scala, Java, and R.
  - **Flexibility:** Handles diverse workloads (batch, streaming, ML).
  - **Fault Tolerance:** Recovers automatically using lineage graphs.

.

Spark’s Multi-Language Support

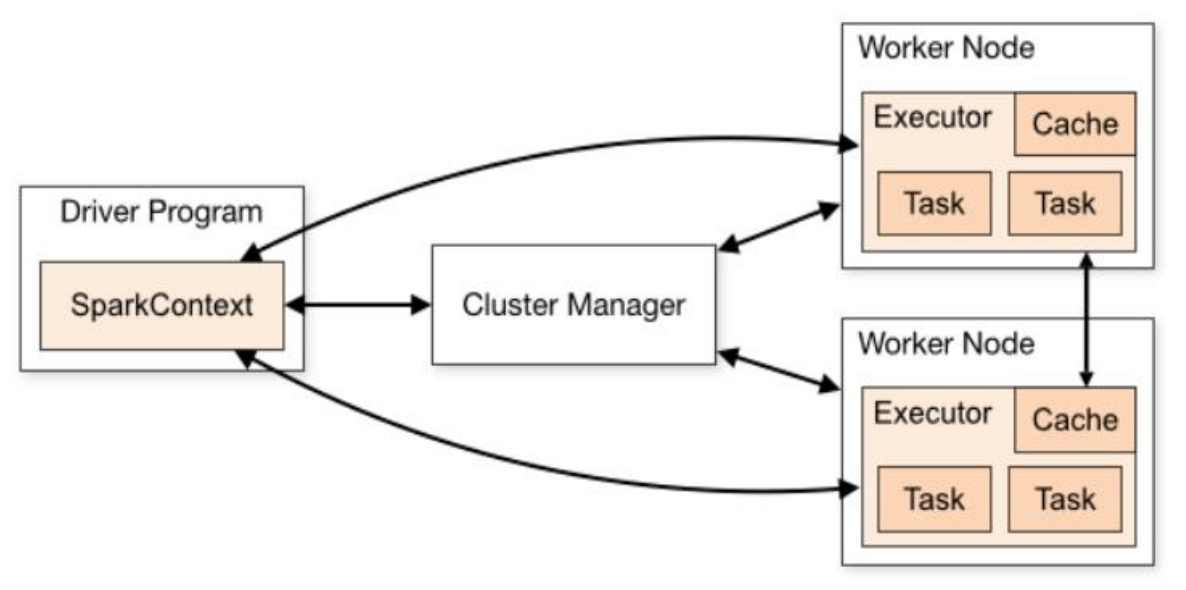
- Although Spark supports multiple languages (e.g., Python via PySpark, R, SQL, and Java), the underlying execution engine is still JVM-based. Here’s how Spark integrates other languages:
- **PySpark**:
  - PySpark uses the JVM to execute Spark's Scala-based core engine.
  - It communicates with the JVM using a bridge called Py4J, which translates Python commands into JVM instructions.
- **SparkR**:
  - Similar to PySpark, SparkR also relies on the JVM for execution.
- **SQL**:
  - Spark SQL queries are translated into operations on the JVM engine.

What is PySpark?

- PySpark is the Python API for Apache Spark. It allows Python developers to harness the power of Spark for distributed data processing. Key features include:
  - High-level abstraction for working with Resilient Distributed Datasets (RDDs) and DataFrames.
  - Compatibility with Python libraries, enabling integration with tools like Pandas and NumPy.
  - Access to Spark’s machine learning and graph processing libraries.

Spark Architecture Overview

- **Driver:** Coordinates the application.
- **Executors:** Perform computations and store results.
- **RDD (Resilient Distributed Dataset):** Immutable distributed data collection.
- **DAG (Directed Acyclic Graph):** Execution plan for computations.
- **Stages and Tasks:**
  - Stages: Group of tasks without shuffles.
  - Tasks: Smallest unit of execution.



Spark Master and Cluster Manager

- **Spark Master:**
  - Manages resources and schedules tasks.
- **Cluster Manager:**
  - Allocates resources and communicates with the Master.
  - Types:
    - Standalone
    - YARN (Hadoop)
    - Mesos
    - Kubernetes

Clone Course Materials

- Install git on local os:
  - *sudo apt update*
  - *Sudo apt install git –y*

- Clone the course materials
  - *cd /opt*
  - *sudo mkdir pyspark*
  - *sudo chown -R YOUR_USERNAME pyspark*
  - *git clone https://github.com/SaeedShirazi/pyspark_training_codes*

Installing PySpark on a Local OS

- **Install Prerequisites:**
  - **Java:** Install Java Development Kit (JDK 8 or later).
    - *sudo apt install openjdk-8-jdk*
  - **Python:** Install Python 3.6 or later.
    - *sudo apt install python3.10 python3-pip*
  - **Spark:** Download and extract Spark.
    - *wget https://downloads.apache.org/spark/spark-3.5.1/spark-3.5.1-bin-hadoop3.tgz*
    - *tar -xvf spark-3.5.0-bin-hadoop3.tgz.*
- **Configure Environment Variables:**
- Add paths to .bashrc
  - `export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64`
  - `export SPARK_HOME=~/spark-3.5.1-bin-hadoop3`
  - `export PATH=$SPARK_HOME/bin:$PATH`
  - `source ~/.bashrc.`

Setting Up and Running PySpark

- **Setting Up a PySpark Environment:**
  - Verify installation: *java -version, python3 --version, spark-submit --version.*
  - Install PySpark Python library: *pip install pyspark.*
- **Starting PySpark Shell:** *pyspark*
  - Example:

```
from pyspark import SparkContext
sc = SparkContext("local", "IntroApp")
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)
result = rdd.map(lambda x: x * 2).collect()
print(result)
sc.stop()
```

- **Running PySpark Scripts with spark-submit:**
- Create a Python script (example1.py)

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Example").getOrCreate()
data = [("Alice", 29), ("Bob", 35)]
df = spark.createDataFrame(data, ["Name", "Age"])
df.show()
```

Execution Modes in PySpark

- **Local[*] Mode:**
  - Runs Spark locally on all available cores.
  - Suitable for development and testing.
  - Command: `pyspark --master local[*]`.
- **Cluster Mode:**
  - Runs Spark on a cluster with multiple nodes.
  - Requires a cluster manager (YARN, Mesos, Kubernetes).
  - Command example: `spark-submit --master yarn example.py`.
- **Client Mode**

Data in Spark

- **RDDs (Resilient Distributed Datasets):**
  - Immutable distributed data collections.
  - Supports transformations (e.g., map, filter) and actions (e.g., collect, count).
- **DataFrames:**
  - Distributed table-like data structure.
  - Optimized execution via Catalyst optimizer.
  - Provides higher-level APIs compared to RDDs.
- **Practical Example:**
  - **Data Read, Process, Write Pipeline: (example2.py)**

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Pipeline").getOrCreate()
df = spark.read.csv("/opt/pyspark/pyspark_training_codes/sample_data.csv", header=True, inferSchema=True)
processed_df = df.filter(df["age"] > 30)
processed_df.write.mode("overwrite").option("header", "true").csv("output_data")
```

Cluster Types

- **YARN (Yet Another Resource Negotiator):**
  - Resource manager for Hadoop ecosystems.
  - Integrates with Spark for distributed task scheduling.
- **Mesos:**
  - General-purpose cluster manager.
  - Allows multiple frameworks to share resources.
- **Kubernetes:**
  - Container orchestration system.
  - Efficient for containerized Spark deployments.

Cluster Types - Summary

Deployment Mode	Cluster Manager	Driver Location	Use Case	Advantages	Disadvantages
Standalone Mode	Spark (built-in)	Client or Cluster mode	Small to medium-sized clusters, simple setup	Easy to set up, low overhead	Limited resource management, less scalable
Cluster Mode	YARN, Mesos, Kubernetes	Cluster	Large clusters, production jobs	Fault tolerance, better resource management	Complex setup, overhead
YARN Mode	YARN	Cluster	Hadoop ecosystem, multi-tenancy	Resource management, integrates with Hadoop	Complexity of YARN setup
Mesos Mode	Mesos	Cluster	Multi-framework environments, elastic scaling	Efficient resource sharing, scalability	Requires Mesos setup
Kubernetes Mode	Kubernetes	Cluster	Cloud-native, containerized environments	Seamless integration with containerized apps	Requires Kubernetes setup

PySpark Basics

Introduction to Resilient Distributed Datasets (RDDs)

- **Definition:** Resilient Distributed Datasets (RDDs) are the core abstraction in Spark.
- **Key Characteristics:**
  - Immutable distributed collection of objects.
  - Fault-tolerant.
  - Lazy evaluation for transformations.

Lazy Evaluation and Fault Tolerance

- **Lazy Evaluation:**
  - Transformations are not executed immediately.
  - Actions trigger the execution of transformations.
- **Fault Tolerance:**
  - Data is stored in partitions.
  - Lineage information allows recomputation of lost data.

RDD transformations

- Definition: Operations that produce a new RDD from an existing one.
- Types:
  - Narrow Transformations (e.g., map, filter).
  - Wide Transformations (e.g., reduceByKey, groupByKey)
  - Common transformations: (example1.py)

```
#map
#Applies a function to each element.
rdd = sc.parallelize([1, 2, 3])
mapped_rdd = rdd.map(lambda x: x * 2)  # [2, 4, 6]
print(mapped_rdd)
#filter
#Filters elements based on a condition.
filtered_rdd = rdd.filter(lambda x: x > 2)  # [3]
print(filtered_rdd)
#flatMap
#Flattens results into a single list
flat_mapped_rdd = rdd.flatMap(lambda x: [x, x*2])  # [1, 2, 2, 4, 3, 6]
print(flat_mapped_rdd)
```

RDD Transformations

- More transformations: (example1.py)

```
#reduceByKey
#Aggregates values for each key.
pairs = sc.parallelize([("a", 1), ("a", 2)])
reduced = pairs.reduceByKey(lambda x, y: x + y)  # [("a", 3)]
print(reduced)
#distinct
#Removes duplicates.
distinct_rdd = rdd.distinct()  # [1, 2, 3]
#union
#Combines two RDDs.
union_rdd = rdd.union(sc.parallelize([4, 5]))  # [1, 2, 3, 4, 5]
```

RDD Actions

- **Definition:** Operations that trigger computation and return a value to the driver program.
  - **Common Actions:**

```
#Actions

#collect
#Retrieves all elements from the RDD.
print(rdd.collect())  # [1, 2, 3]

#count
#Returns the number of elements.
print(rdd.count())  # 3

#take
#Retrieves the first n elements.
print(rdd.take(2))  # [1, 2]

#reduce
#Aggregates elements using a function.
result = rdd.reduce(lambda x, y: x + y)  # 6

#first
#Returns the first element.
print(rdd.first())  # 1
```

Transformations and actions comparison

Aspect	Transformations	Actions
Definition	Create a new RDD/DataFrame by applying a function.	Trigger computation and return results.
Lazy Evaluation	Yes, they are lazy. Spark builds a DAG but doesn't execute it.	No, they trigger the execution of the DAG.
Result	A new RDD/DataFrame (transformation is not executed immediately).	Materialized result (e.g., a value or data).
Examples	map(), filter(), flatMap(), groupByKey()	collect(), count(), saveAsTextFile()

Hands-on practice

- **Objective:** Perform transformations and actions on a real dataset.
- **Steps**:
  - Load the CSV file.
  - Filter rows containing "Marketing“.
  - Extract unique names.
  - Count total rows.

Key-Value RDD Operations

- Key-Value RDDs are a special type of RDD that hold key-value pairs. They are commonly used for aggregations and grouping operations in PySpark.
- **reduceByKey (example2.py)**
  - **Definition:** Combines values for each key using an associative reduce function.
  - **Characteristics:**
    - Performs local aggregation on each partition before sending data across the network, reducing data shuffling.
    - More efficient for large datasets.

```
pairs = sc.parallelize([("a", 1), ("b", 2), ("a", 2)])
result = pairs.reduceByKey(lambda x, y: x + y)  # [("a", 3), ("b", 2)]
print(result.collect())
```

Key-Value RDD Operations

groupByKey

- **Definition:** Groups all values with the same key into a single iterator.
  - **Characteristics:**
    - Sends all values for a key across the network.
    - Can be less efficient than reduceByKey due to higher data transfer

```
grouped = pairs.groupByKey()
print([(k, list(v)) for k, v in grouped.collect()])  # [("a", [1, 2]), ("b", [2])]
```

Feature	reduceByKey	groupByKey
Local Aggregation	Yes	No
Network Transfer	Minimal	Higher
Use Case	Aggregations (e.g., sums)	Full access to all values

Narrow and Wide Transformations

- **Narrow Transformations**
  - Data dependencies are within a single partition
  - Examples: map, filter, flatMap
- **Wide Transformations**
  - Data dependencies span multiple partitions.
  - Examples: reduceByKey, groupByKey, join
  - Requires data shuffling across the network.

Narrow and Wide Transformations

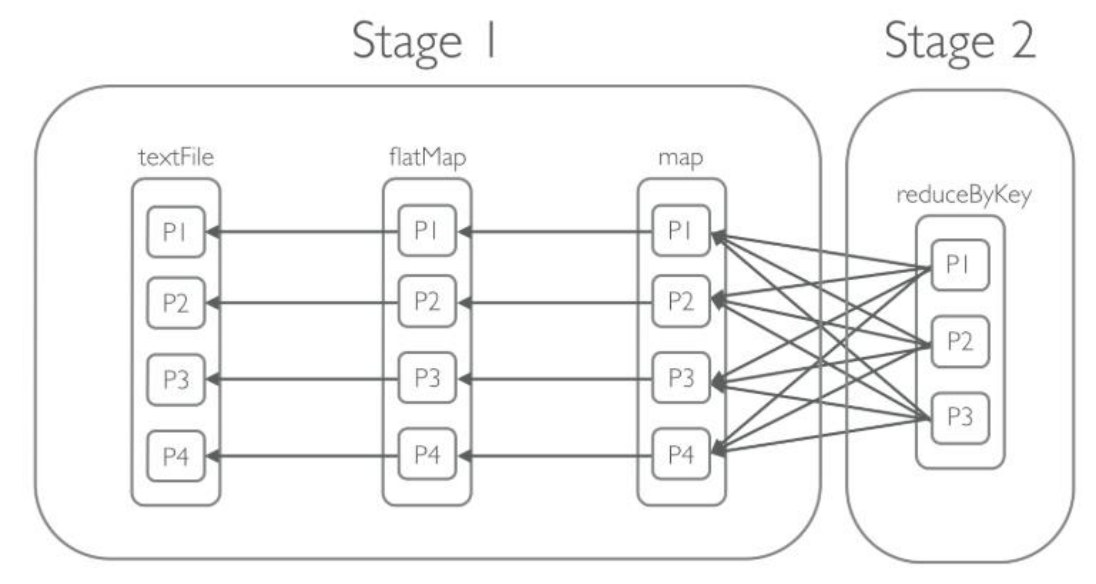
Aspect	Narrow Transformations	Wide Transformations
Data Movement	No data movement between partitions.	Data must be shuffled across partitions.
Shuffling	No shuffle required.	Shuffle is required, leading to higher overhead.
Parallelism	Can be executed in parallel without data movement.	Cannot be fully parallelized due to data movement.
Efficiency	More efficient (lower overhead).	Less efficient (higher overhead).
Examples	map(), filter(), union(), sample()	groupByKey(), reduceByKey(), join(), distinct()

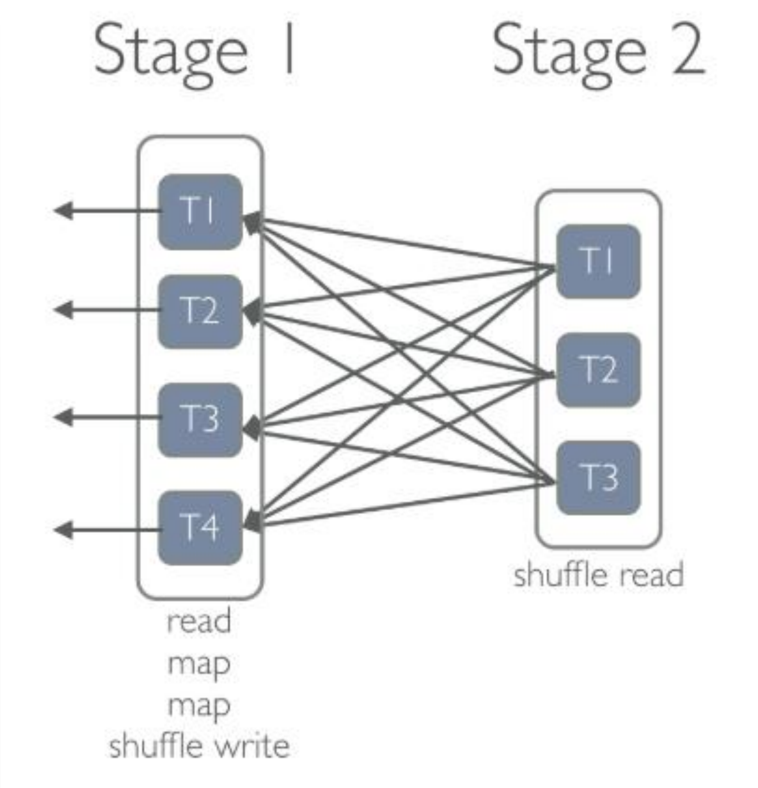
Spark Shuffle

- **Definition:** The process of redistributing data across partitions, often triggered by wide transformations.
- **Stages of Shuffle:**
  - **Map Stage:** Prepares data for transfer.
  - **Shuffle Write:** Writes intermediate data to disk.
  - **Shuffle Read:** Reads data by other nodes.
- **Impact:**
  - Increases execution time due to disk I/O and network transfer.
  - Optimization techniques like partitioning and avoiding skew are crucial.

Job flow example

```
rdd = sc.textFile("input.txt")\
.flatMap(lambda line: line.split())\
.map(lambda word: (word, 1))\
.reduceByKey(lambda x, y: x + y, 3)\
.collect()
```





How RDDs are Partitioned

- Data in RDDs is divided into logical partitions, which are processed in parallel.
- Default partitioning is based on the number of cores available. (example3.py)

```
rdd = sc.parallelize([1,2,3,4,5], numSlices=2)
print(rdd.getNumPartitions()) #Output: 2
```

- Proper Partitioning: Balances workload across nodes and minimizes shuffling.
- Partitioning Strategies:
  - Use partitionBy for key-value RDDs to ensure keys are grouped.
  - Repartition large datasets for better parallelism.

```
pairs = sc.parallelize([("a",1), ("b", 2), ("a", 2)])
partiotioned = pairs.partitionBy(2)
print(partiotioned.getNumPartitions()) #Output: 2
```

Broadcast and Accumulators (Intro)

- **Broadcast Variables**
  - **Definition:** Shared, read-only variables that are cached on each machine in the cluster.
  - **Use Case:** Efficiently distribute large datasets (e.g., lookup tables).
  - Example: (example4.py)

```
lookup_table = {"a": 1, "b": 2, "c": 3}
broadcast_var = sc.broadcast(lookup_table)

rdd = sc.parallelize(["a", "b", "c", "d"])
result = rdd.map(lambda x: broadcast_var.value.get(x, 0))
print(result.collect())  # Output: [1, 2, 3, 0]
```

Broadcast and Accumulators (Intro)

- **Accumulators**
  - **Definition:** Shared variables for aggregating information (e.g., counters, sums) across tasks.
  - **Use Case:** Monitor or debug jobs by tracking counts or metrics.
- Example:

```python
accum = sc.accumulator(0)

def add_to_accum(x):
    global accum
    accum += x

rdd = sc.parallelize([1, 2, 3, 4])
rdd.foreach(add_to_accum)
print(accum.value)  # Output: 10
```

PySpark DataFrames Basics

How DataFrames Differ from RDDs

- **Structure**: DataFrames are distributed collections of data organized into named columns, similar to a table in a relational database.
- **Ease of Use**: Provide a higher-level abstraction than RDDs, making them easier to use for SQL-like operations.
- **Optimizations**: DataFrames leverage Spark's Catalyst optimizer for optimized query execution, whereas RDDs require manual optimization.
- **Benefits of DataFrames**
  - **Optimized Execution**: Catalyst optimizer and Tungsten execution engine improve performance.
  - **Familiar Syntax**: SQL-like operations make DataFrames user-friendly.
  - **Interoperability**: Easily integrates with various file formats (CSV, JSON, Parquet).

Difference between RDD and Dataframe

Aspect	RDD (Resilient Distributed Dataset)	DataFrame
Abstraction	Low-level abstraction for distributed data.	High-level abstraction with schema information.
Schema	Does not have a schema (unstructured).	Has a schema (structured).
Ease of Use	Requires more code to perform operations.	Provides optimized APIs for easier usage.
Performance	Less optimized (no Catalyst or Tungsten).	Highly optimized using Catalyst and Tungsten.
Interoperability	Not easily compatible with SQL or BI tools.	Easily integrates with Spark SQL and B tools.
Use Case	Ideal for complex, low-level transformations.	Best for structured data and SQL-like queries.

When to use?

Scenario	Preferred API	Reason
Unstructured/complex data	RDD	No schema required; supports custom transformations.
Tabular data (e.g., CSV, JSON)	DataFrame	Schema support; SQL-like operations; performance.
Performance-critical tasks	DataFrame	Optimized execution via Catalyst and Tungsten.
Full control over processing	RDD	Fine-grained control over partitions and data flow.
SQL queries or analytics	DataFrame	Intuitive SQL-style operations.

Creating DataFrames

❖ **From Python Data Structures.** (example1.py)

```
#From Lists:
data = [(1, "Alice", 29), (2, "Bob", 35), (3, "Cathy", 25)]
columns = ["ID", "Name", "Age"]
df = spark.createDataFrame(data, columns)
df.show()

#From Dictionaries:
data = [{"ID": 1, "Name": "Alice", "Age": 29},
        {"ID": 2, "Name": "Bob", "Age": 35},
        {"ID": 3, "Name": "Cathy", "Age": 25}]

df = spark.read.json(sc.parallelize(data))
df.show()
```

Creating DataFrames

- Reading files

```
#CSV:
df = spark.read.csv("/opt/pyspark/pyspark_training_codes/sample_data.csv", header=True, inferSchema=True)
df.show()

#JSON:
df = spark.read.json("data.json")
df.show()

#Parquet:
df = spark.read.parquet("data.parquet")
df.show()
```

Basic DataFrame Operations

(example2.py)

```
#Select and Filter
#Select Columns:
df.select("name", "age").show()
#Filter Rows:
df.filter(df.age > 30).show()
#GroupBy and Aggregate
#GroupBy:
df.groupBy("age").count().show()
#Aggregate:
df.groupBy("age").agg({"id": "count"}).show()
#Handling Missing or Corrupt Data
#Drop Rows with Null Values:
df.na.drop().show()
#Fill Missing Values:
df.na.fill({"age": 0}).show()
```

Advanced Column Operations

```
#Add a New Column:

df = df.withColumn("NewColumn", df.age * 2)
df.show()

#Rename Columns:

df = df.withColumnRenamed("age", "years")
df.show()

#Inspecting Schemas:
df.printSchema()

#Explaining Logical and Physical Plans:
df.select("name", "age").explain()
```

Hands-on practice

- Filter and Clean Data:
  - Remove rows with null or missing values in the salary column.
  - Convert the sign_up column to a proper date format.
- Department Analysis:
  - Calculate the salary for each job_title.
  - Identify the job_title with the highest average salary.
- Joining Trends:
  - Find the year-wise count of employees joining the company.
  - Determine the year with the highest number of joinings.
- Employee Insights:
  - Add a new column ExperienceYears based on the difference between the current year and the JoiningDate.
  - Filter out employees with less than 5 years of experience.
- Write the Result:
  - Save the final DataFrame as a Parquet file for future analysis.

Advanced DataFrame Operations

Working with SQL in PySpark

- PySpark allows you to run SQL queries directly on DataFrames by registering them as temporary tables. This is particularly useful for those familiar with SQL syntax.(example1.py)

```
#Registering DataFrames as Temporary SQL Tables

# Registering a DataFrame as a temporary SQL table
df.createOrReplaceTempView("employee_table")

#Executing SQL Queries

# Querying the table
high_salary_employees = spark.sql("SELECT * FROM employee_table WHERE salary > 900")
high_salary_employees.show()
```

User-Defined Functions (UDFs)

- UDFs allow custom Python functions to be applied to DataFrame columns. However, they can negatively impact performance because they run outside Spark's Catalyst optimizer.
- **Performance Considerations**
- UDFs do not leverage Spark's Catalyst optimizer.
- Use built-in Spark functions whenever possible for better performance.
- Use pandas UDFs for improved speed with Arrow optimization.

User-Defined Functions (UDFs)

- **Creating and Applying UDFs. (example2.py)**

```
# Define a UDF to categorize salary
def categorize_salary(salary):
    if salary > 900:
        return "High"
    elif salary > 700:
        return "Medium"
    else:
        return "Low"

# Register the UDF
categorize_salary_udf = udf(categorize_salary, StringType())

# Apply the UDF
df_with_category = df.withColumn("salary_category", categorize_salary_udf(col("salary")))
df_with_category.show()
```

Joining DataFrames

- Joins are a fundamental operation in PySpark, but they can be resource-intensive.
- **Types of Joins**
  - **Inner Join:** Matches records from both DataFrames.
  - **Left Join:** All records from the left DataFrame and matching records from the right.
  - **Right Join:** All records from the right DataFrame and matching records from the left.
  - **Outer Join:** All records from both DataFrames, with nulls where no match is found.

Joining DataFrames

- Example code(example3.py)

```
df1 = df.select("id", "name", "job_title")
df2 = df.select("id", "salary")

# Inner join
inner_join_df = df1.join(df2, on="id", how="inner")
inner_join_df.show()

# Left join
left_join_df = df1.join(df2, on="id", how="left")
left_join_df.show()

#Avoiding Shuffles in Joins

#Broadcast Joins: Use when one DataFrame is small.
from pyspark.sql.functions import broadcast
broadcast_join_df = df1.join(broadcast(df2), on="id", how="inner")
```

Window Functions

- ❖ Window functions allow calculations across a subset of rows.
- ❖ **Ranking, Cumulative Sums, and More (example4.py)**

```
# Define a window
window_spec = Window.partitionBy("job_title").orderBy(col("salary").desc())

# Rank employees by purchase within their job_title
ranked_df = df.withColumn("Rank", rank().over(window_spec))
ranked_df.show()

# Cumulative sum of salary
cumulative_salary_df = df.withColumn("cumulative_purchase", sum("salary").over(window_spec))
cumulative_salary_df.show()
```

Optimization Insights

- The Catalyst optimizer is a core part of Spark SQL that optimizes query execution plans.
- Optimization Strategies:
  - Use explain() to understand the physical and logical plans.
  - Avoid UDFs when possible; use built-in functions.
  - Optimize joins using broadcast or partitioning
  - Code:
    - # Show logical and physical plans
    - df.explain(True)

Hands-on practice

- Task 1: User-Defined Functions (UDFs)
  - Create and Apply a UDF
  - Write a UDF named bonus_percentage that calculates 10% of the employee's purchase.
- Task 2: Joining DataFrames
  - Perform a Simple Join: Split the sample_data.csv DataFrame into two smaller DataFrames:
    - departments: Columns ID, Name, Job_title.
    - salaries: Columns ID, salary
  - Perform an inner join on these two DataFrames using the ID column.

PySpark Streaming

Introduction to Spark Streaming

- How Spark Processes Streaming Data in Micro-Batches:
  - Spark Streaming breaks the live data stream into small batches (micro-batches).
  - Each batch is processed as an RDD or DataFrame.
  - Micro-batch processing offers near real-time data processing.
- Use Cases for Spark Streaming:
  - Real-time fraud detection.
  - Monitoring log files and metrics.
  - Live dashboard updates.
  - Stream processing for IoT devices.

Creating a Streaming DataFrame

- **Reading Data from File Sources: (**example1.py)
  - Spark can monitor a directory for new files and process them as a stream.
  - Example: Processing log files or CSV files dropped into a directory.

```
# Monitor a directory for new CSV files
stream_df = spark.readStream.format("csv").option("header", True).schema(
    "id INT, name STRING, age INT, city STRING, salary FLOAT, "
    "signup_date STRING, email STRING, job_title STRING, is_active STRING"
).load("/opt/pyspark/pyspark_training_codes/pyspark_streaming/data/")
```

Processing Streaming Data

- **Transformations and Actions**:
  - Similar to static DataFrames but with continuous updates.
  - Example: Filtering and selecting columns.

```
filtered_stream = stream_df.filter("salary > 900").select("name", "job_title", "salary")
```

- **Aggregating Streaming Data**:
  - Example: Grouping data and computing aggregates.

```
aggregated_stream = stream_df.groupBy("job_title").avg("salary")
```

Stateful Streaming

- Stateful Transformations (e.g., updateStateByKey):
  - Maintains state across batches for operations like running totals
  - Example: Tracking cumulative sales by product.
- Structured Streaming's with Watermark:
  - Handles late data by specifying an event time watermark.
  - Ensures efficient state management by removing old state data.

```
# Apply watermark and aggregation
watermarked_stream = stream_df.withWatermark("eventTime", "1 seconds") \
    .groupBy("job_title", "eventTime") \
    .count()
```

Checkpointing and Writing Processed Data

- **Checkpointing**
  - **Ensuring Fault Tolerance**:
    - Spark uses checkpointing to store intermediate states.
    - Required for stateful transformations

```
# Also outputting to the console for testing
console_query = watermarked_stream.writeStream \
    .format("console") \
    .outputMode("update") \
    .option("checkpointLocation", "/opt/pyspark/pyspark_training_codes/pyspark_streaming/checkpoint_console") \
    .start()
```

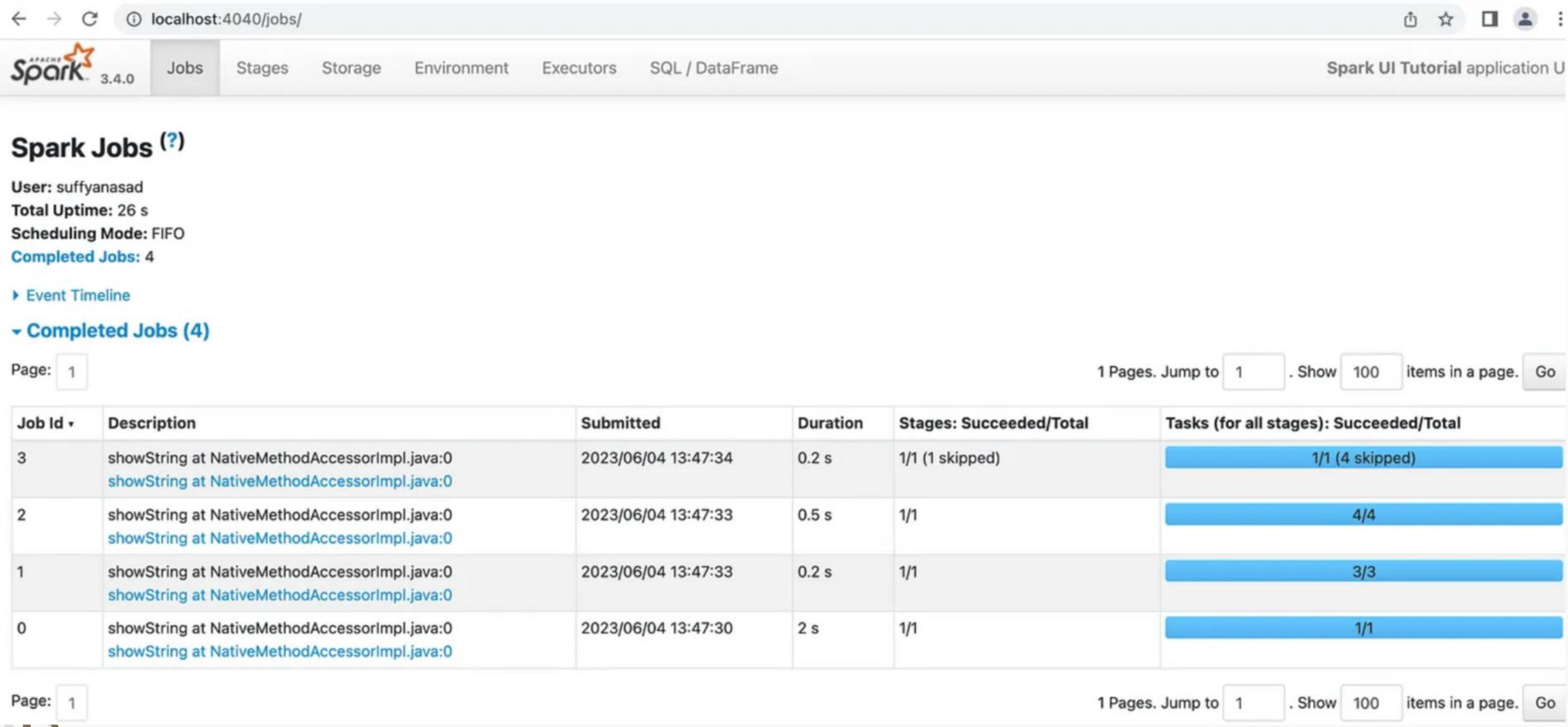
- **Writing Processed Data**
  - **Writing to External Systems or Storage**:
    - Supports sinks like HDFS, Kafka, or a database.

Hands-on practice

- Stream Data Processing:
  - Monitor a directory for new CSV files, as done in the example.
  - Add a column salary_group that categorizes salaries into "Low", "Medium", or "High" based on salary ranges.
- Rolling Average Calculation:
  - Calculate the rolling average of the salary for each job_title. The window for the rolling average should be set to 5 seconds (or another reasonable time window based on incoming data frequency).
- Outlier Detection:
  - Identify outliers in the salary data for each job_title. An outlier is defined as a salary that is greater than 1.5 times the rolling average for that job title.
  - Create a column is_outlier that is True if the salary is an outlier, and False otherwise.
- Output:
  - Write the results to both the console (with outputMode("update")) and a Parquet directory (outputMode("append")).
  - Store the output in separate directories for the console and Parquet output.

Optimization and Deployment

Spark UI Overview


localhost:4040/jobs/

Spark 3.4.0 | Jobs | Stages | Storage | Environment | Executors | SQL / DataFrame | Spark UI Tutorial application U

Spark Jobs (?)

User: suffyanasad
Total Uptime: 26 s
Scheduling Mode: FIFO
Completed Jobs: 4

▸ Event Timeline

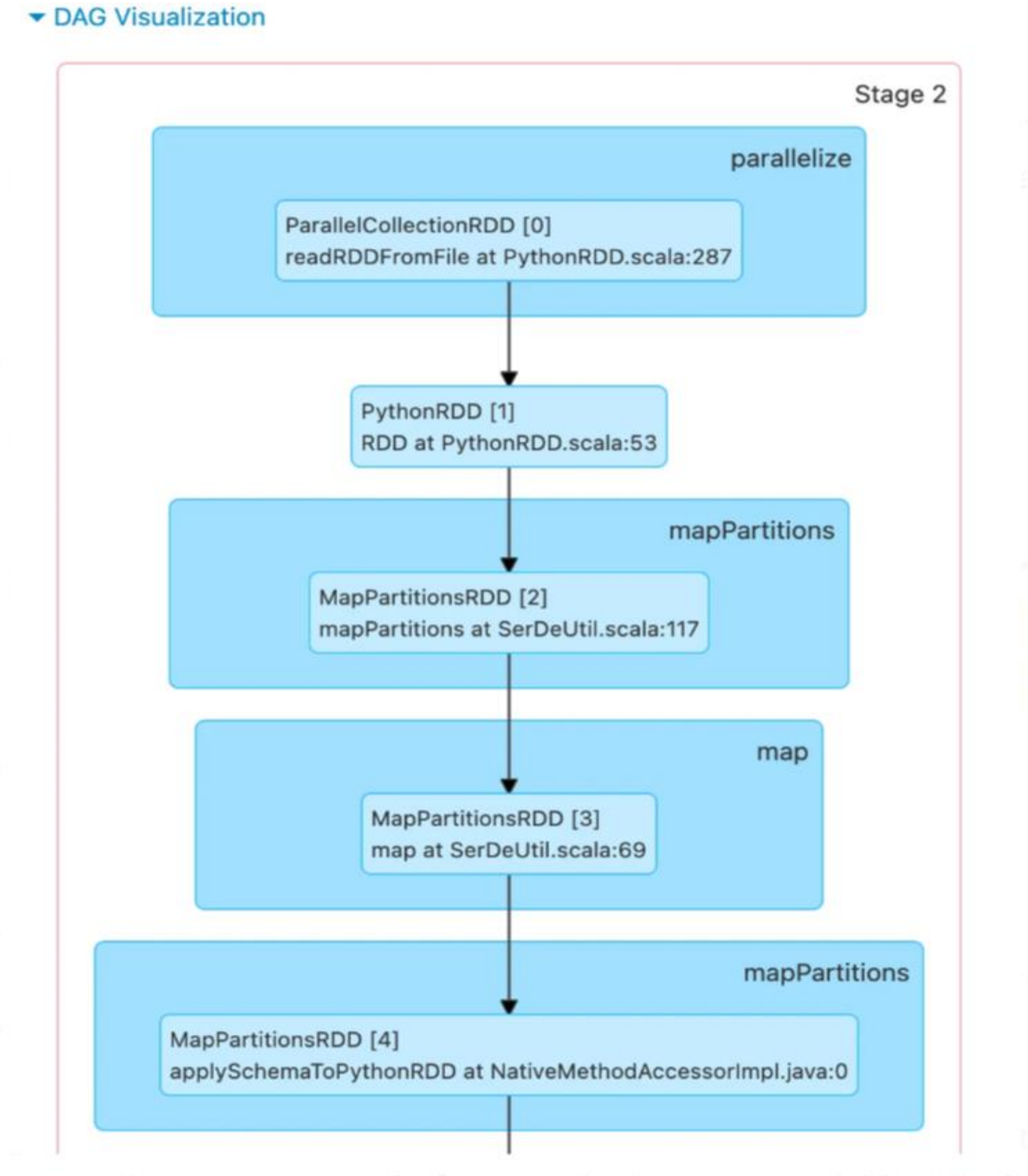
▾ Completed Jobs (4)

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2023/06/04 13:47:34	0.2 s	1/1 (1 skipped)	1/1 (4 skipped)
2	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2023/06/04 13:47:33	0.5 s	1/1	4/4
1	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2023/06/04 13:47:33	0.2 s	1/1	3/3
0	showString at NativeMethodAccessorImpl.java:0 showString at NativeMethodAccessorImpl.java:0	2023/06/04 13:47:30	2 s	1/1	1/1

Page: 1 — 1 Pages. Jump to 1 . Show 100 items in a page. Go

Spark UI Overview



PySpark Performance Optimization

- Partitioning and Repartitioning:
  - Proper partitioning ensures data is distributed evenly across executors.
  - Use .repartition() or .coalesce() to adjust partitions.
  - df = df.repartition(10) # Increase partitions
  - df = df.coalesce(2)   # Decrease partitions
- Caching and Persistence Strategies:
  - Cache data when it is reused multiple times to avoid recomputation.
  - df.cache() # Stores data in memory
  - df.persist(StorageLevel.DISK_ONLY) # Stores data on disk
- Using Broadcast Variables:
  - Broadcast small datasets to avoid repetitive data shuffling.
  - broadcast_var = spark.sparkContext.broadcast([1, 2, 3])

PySpark Performance Optimization

Aspect	repartition()	coalesce()
Purpose	Increases or decreases the number of partitions.	Reduces the number of partitions.
Shuffle	Requires a full shuffle of data.	No shuffle (merges adjacent partitions).
Use Case	Used when increasing or significantly changing the number of partitions.	Used when reducing the number of partitions (e.g., after filtering or aggregation).
Efficiency	Can be expensive due to the shuffle.	More efficient for reducing partitions, as it avoids a full shuffle.
Typical Operations	Used to balance data across many partitions or to optimize parallelism.	Used before writing data out to disk or when a small number of partitions is sufficient.

Parameter Tuning

❖ Important Configuration Parameters:

Configuration	Description	Example
spark.master	Specifies the cluster manager (e.g., yarn, local, k8s)	spark.master=yarn
spark.app.name	Name of the Spark application	spark.app.name=MySparkApp
spark.executor.memory	Amount of memory per executor	spark.executor.memory=4g
spark.driver.memory	Amount of memory for the driver	spark.driver.memory=2g
spark.executor.cores	Number of cores per executor	spark.executor.cores=4
spark.shuffle.partitions	Number of shuffle partitions	spark.shuffle.partitions=200
spark.local.dir	Local directory for shuffle and spill files	spark.local.dir=/tmp
spark.eventLog.enabled	Enable Spark event logging	spark.eventLog.enabled=true
spark.eventLog.dir	Directory for event logs	spark.eventLog.dir=hdfs:///user/
spark.hadoop.fs.defaultFS	Default file system (HDFS, S3, etc.)	spark.hadoop.fs.defaultFS=hdfs
spark.sql.shuffle.partitions	Number of partitions for Spark SQL shuffle	spark.sql.shuffle.partitions=200
spark.yarn.am.memory	Amount of memory for the YARN Application Master	spark.yarn.am.memory=1g
spark.executor.instances	Number of executors for YARN cluster	spark.executor.instances=10

Writing Data to External Storage

- Writing Data to External Storage HDFS, Databases, and File Formats:
  - Save data in efficient formats like Parquet or ORC.
  - df.write.format("parquet").save("hdfs://path/to/save")
- Databases:
  - Writing to relational or NoSQL databases like PostgreSQL, MySQL, or Cassandra.

```
df.write \
  .format("jdbc") \
  .option("url", "jdbc:postgresql://localhost:5432/mydb") \
  .option("dbtable", "orders") \
  .option("user", "username") \
  .option("password", "password") \
  .save()
```

Running PySpark Applications on a Cluster

- Using spark-submit in Cluster Mode:
  - Command to submit a PySpark application:
  - *spark-submit --master yarn --deploy-mode cluster \ --num-executors 4 --executor-memory 4G --executor-cores 2 \ my_script.py*
- **Dynamic Allocation of Resources**:
  - Enables automatic scaling of executors.
    - *--conf spark.dynamicAllocation.enabled=true*

Real-World Deployment

- **Logging, Monitoring, and Debugging**:
- Use log aggregation systems like ELK Stack or Prometheus.
- Enable Spark event logging for detailed insights.
  - *--conf spark.eventLog.enabled=true \ --conf spark.eventLog.dir=hdfs://path/to/logs*

Hands-on with PySpark in Interactive Notebooks

- **Introduction:**
  - PySpark can be run in interactive notebooks such as Jupyter Notebooks or Apache Zeppelin.
  - These notebooks allow for real-time code execution, making them ideal for learning and experimentation.
- **Key Features:**
  - Code Execution: Write and execute PySpark code interactively.
  - Visualization: Display results and plots within the same notebook.
  - Collaboration: Share notebooks with peers or use them for team-based learning.

Delta Lake for ACID Transactions & Time Travel

- **Introduction:**
  - Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark and big data workloads.
  - Provides features like schema evolution, time travel, and versioned data**.**
- **Key Features:**
  - ACID Transactions: Ensures data consistency and integrity during write operations.
  - Time Travel: Allows querying previous versions of the data.
  - Schema Enforcement & Evolution: Enforces schema and allows schema evolution over time.

Spark MLlib for Machine Learning

- **Introduction:**
  - MLlib is Spark's scalable machine learning library for building predictive models.
  - It provides algorithms for classification, regression, clustering, and more.
- **Key Features:**
  - Scalability: Works with large datasets across distributed clusters.
  - Algorithms: Supports common ML algorithms like Logistic Regression, Decision Trees, K-Means, etc.
  - Pipelines: Provides a unified API for building end-to-end ML pipelines.

You're now equipped with the skills to handle big data like a pro. Go forth, conquer, and may your clusters always be efficient. Good luck!