Università degli Studi di Messina

Data Science

Big Data Acquisition
Spark

Prof. Daniele Ravi

dravi@unime.it

Recap from Last Lecture

- MapReduce -> new distributed computing framework suitable for working with large scale datasets

- Useful in all those situations where data need to be accessed **sequentially**

- May be hard to program and does not support well multiple map-reduce rounds

Data-Flow Systems

- MapReduce uses 2 "ranks" of tasks: one for **map** the other for **reduce**
- Data flows from the first rank to the second
- Generalized Data-Flow Systems abstract from this in two ways:
  - Allow any number of "ranks"/tasks
  - Allow functions other than just map and reduce
- As long as data goes in one direction only, recovery at intermediate rank is possible

MapReduce: Criticisms

- **3 major limitations** of MapReduce:
  - Hard to program directly
    - Many problems are not easily described as map-reduce
  - Relies heavily on disk I/O communication which can become a bottleneck for performance.
    - **Latency**: MapReduce is optimized for batch processing and may not be suitable for real-time applications.
    - Persistence to disk slower than in-memory computation
  - MapReduce is not suitable for more complex operations composed of several map-reduce steps or iterative algorithms (e.g., machine learning)

Apache Spark Overview

- Apache Spark is an open-source, distributed computing system designed for big data processing.
- It provides an advanced data analytics engine that can handle large-scale data with speed and scalability.
- Spark was developed in response to **the limitations of Hadoop's MapReduce** model, focusing on making data processing faster and more efficient.
- It's particularly suited for large-scale data processing tasks like batch processing, streaming, machine learning, and graph computation.

Spark: Introduction

- Originally developed at UC Berkeley in 2009 and later donated to the Apache Software Foundation (open-source)
- Implemented in **Scala** (running on top of the Java Virtual Machine)
- Unified **computing engine** (**Spark Core**)
- Set of **high-level APIs** for data analysis:
  - **Spark SQL** (structured data), **MLlib** (machine learning), **GraphX** (graph analytics), **Spark Streaming** (stream data processing)

Spark: Introduction

- Unlike Hadoop, Spark does not come with a storage system

- In fact, it provides interfaces for many local and distributed storage systems:
  - HDFS, Amazon S3, Cassandra, Hive Metastore, or classical RDBMS

- Additionally, Spark's APIs are available for many programming languages: Scala, Java, Python, and R

- This flexibility is the key of its success in the Big Data landscape

Spark: Most Popular Data-Flow System

- Computing framework not limited to map-reduce model

- In addition to MapReduce, Spark provides:
  - Fast data sharing (no intermediate saving to local disks + caching)
  - General execution graphs (DAGs)
  - Richer functions than just map and reduce

- Compatible with Hadoop

Key Features of Spark

1. **Speed**: Spark uses in-memory computation (keeping data in RAM rather than reading and writing from disk), which makes it faster than Hadoop MapReduce for many tasks.
    - It can also spill data to disk when memory is insufficient.

2. **Ease of Use**: Spark offers APIs in several programming languages

3. **Unified Engine**: Spark integrates several libraries into one unified framework allow for ease of implementation in various domains: **Spark SQL, MLlib** , **GraphX, Spark Streaming**

4. **Fault Tolerance**: Like Hadoop, Spark is resilient to failures. It uses Directed Acyclic Graphs (DAGs) to represent job execution plans, and its **lineage-based fault** tolerance allows lost data to be recomputed in case of node failure.

5. **Scalability**: Spark can scale from a single machine to thousands of nodes in a cluster, handling petabytes of data.

Lineage-based fault tolerance

- Is a key concept in frameworks like **Apache Spark**.

- It refers to the ability of the system to **recompute lost data** by keeping track of the **lineage** of data transformations, instead of replicating data across nodes.

- **Here's how it works:**
  - **Lineage**: Each dataset is generated by applying a sequence of transformations to an initial dataset (e.g., map, filter, reduce).
  - This sequence of transformations is referred to as the **lineage** of the dataset.
  - Spark keeps track of this lineage of operations, so it knows how a dataset was derived from its original source.

Lineage-based fault tolerance

- **Fault Tolerance via Lineage**:
  - Rather than restarting the entire computation or replicating data in advance, Spark can **recompute** only the lost part of the data by **re-executing** the transformations based on the lineage information.
  - This is more efficient especially in cases of intermittent failure.

- **Directed Acyclic Graph (DAG)**:
  - Spark represents the lineage of transformations as a **DAG**.
  - Each node in this graph represents an RDD (Resilient Distributed Dataset), and the edges represent transformations applied to those RDDs.
  - If any part of the computation fails, Spark can trace the lineage back through the DAG and recompute only the necessary parts.

How Apache Spark Works

- **RDD (Resilient Distributed Dataset)**
  - It is a distributed collection of objects across the nodes in a cluster, allowing for parallel processing.
  - RDDs are **immutable**, meaning once created, they cannot be altered.
  - However, transformations can be applied to them to create new RDDs.
  - They are fault-tolerant, and Spark can reconstruct lost data by tracking the transformations that led to each RDD.

    - **Transformations**: Operations like map(), filter(), flatmap() are lazily evaluated and do **not execute immediately**. They generate a new RDD and build up a DAG.

    - **Actions**: Operations like reduce(), collect(), and save() trigger the execution of the transformations and bring results back to the driver program or write them to external storage.

How Apache Spark Works

- **Directed Acyclic Graph (DAG)**:
  - When a Spark job is submitted, Spark constructs a DAG to represent the series of transformations applied to the data.
  - It optimizes the execution plan before running the job, minimizing data shuffling and communication between nodes.

- **In-Memory Computation**:
  - Spark performs operations in-memory, avoiding the overhead of disk I/O seen in traditional MapReduce systems.
  - This leads to much faster execution times, especially for iterative tasks like machine learning algorithms or data analytics.

How Apache Spark Works

- **Job Execution**:
  - A **driver** program coordinates the tasks and transformations.
  - **Workers** execute the tasks in parallel on their respective partitions of the data.
  - Tasks are distributed across nodes in a cluster, and intermediate data can be cached in memory for faster access.

- **Cluster Manager**:
  - Spark can work with different cluster managers for resource allocation:
    - **Standalone mode**: Spark's built-in manager.
    - **Apache YARN**: Used in Hadoop ecosystems.
    - **Mesos**: General-purpose cluster manager.
    - **Kubernetes**: Container orchestration platform.

Summary: Spark Features

- Fault-tolerant system

- In-memory caching which enables efficient execution of multi-round algorithms (i.e., multiple sequential tasks)
  - performance improvement w.r.t. Hadoop

- Spark can run:
  - on a single machine -> local mode
  - on a cluster managed by a cluster manager (e.g., Spark Standalone, YARN, Mesos)

Spark: Features

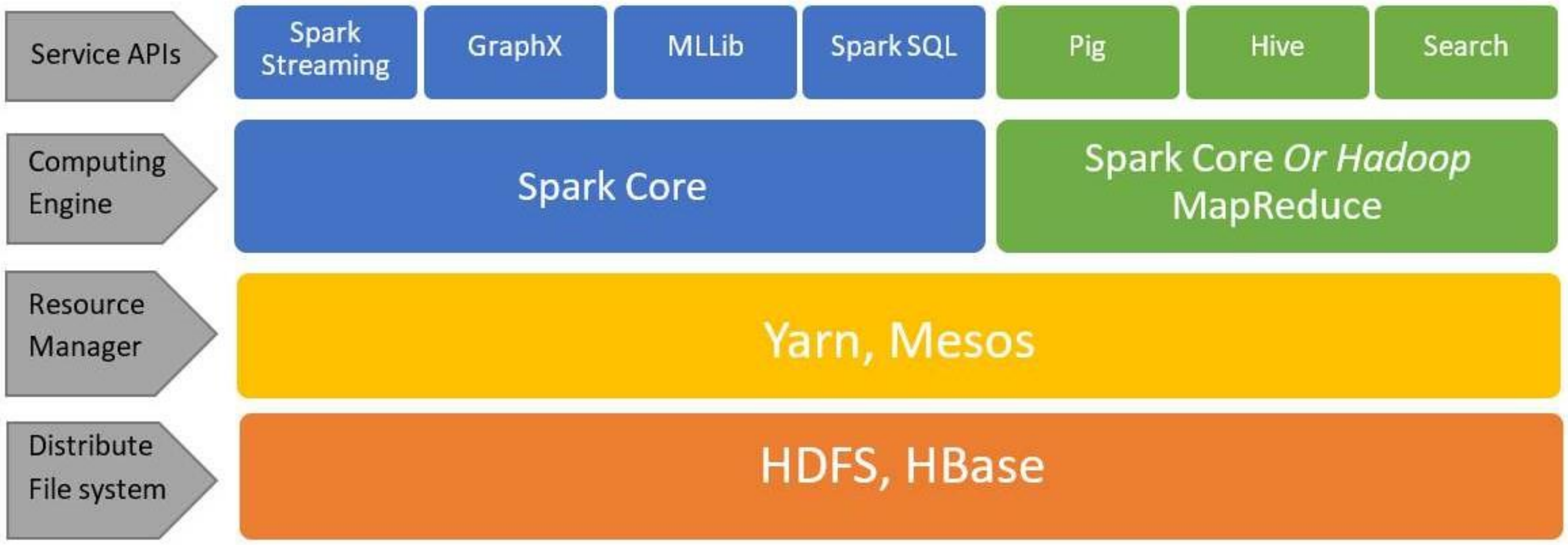


Figure 1 - Spark Context

Spark Application: Driver

- The **driver process** (a.k.a. **master** in MapReduce terminology) runs the application's entry point from a node in the cluster

- The driver is responsible for:
  - Maintaining information about the application
  - Responding to a user program or input
  - Analyzing, distributing, and scheduling work across executors

- The driver is represented by an object called **Spark Context**

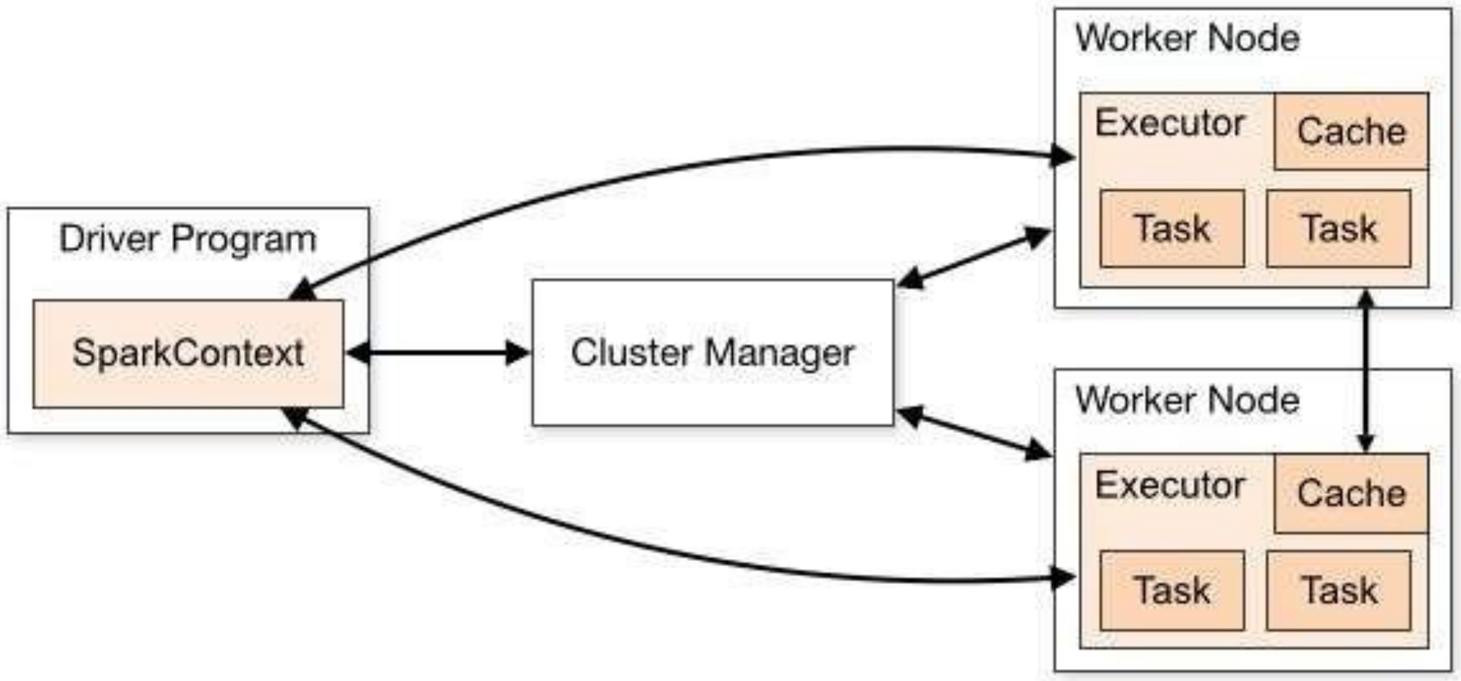
Spark Application: Executor(s) and Cluster Manager

- **Executor** processes (a.k.a. **workers** in Hadoop terminology) actually compute the tasks assigned by the driver

- Each executor is responsible for:
  - Running the code assigned to it by the driver
  - Reporting the state of the computation back to the driver

- The **cluster manager** controls physical machines and allocates resources to applications

Spark Application



Spark Application: Considerations

- **Driver and executors** are processes which can live on the same machines or on different nodes

- When Spark is running in local mode, both the driver and executors are running as separate threads on the same machine

- Executors mostly run Scala code

- Driver can be governed by different languages using Spark's APIs

Resilient Distributed Dataset (RDD)

- Fundamental **abstraction** of Spark to indicate a collection of elements of the same type
  - Generalization of MapReduce's key-value pairs

- RDDs are **partitioned** and possibly spread across multiple nodes of the cluster

- Best suited for applications that apply the same operation across all the elements of the dataset

RDD: Partitions

- Each RDD is split into chunks called partitions distributed across nodes

- A program can specify the number of partitions for an RDD (otherwise Spark will choose one)

- Programmer can also decide whether to use the default Hash Partitioner or a custom one

- A typical number of partitions is 2 or 3 times the number of **total cores in the cluster**

RDD: Partitions

- Partitioning enables the following:
  - **Data reuse - >** data is kept in executors' main memory so as to avoid expensive access to external disks
  - **Parallelism - >** Some data transformations are applied independently to each partition thereby avoiding expensive data transfers

RDD: Characteristics

- RDDs are **immutable** (i.e., read-only)
- Can be created either from data stored on distributed file system (e.g., HDFS) or as a result of transformations of other RDDs
- RDDs do not need to be always materialized
  - Each RDD maintains a sort of "trace" of transformations (lineage) that led to the current status
  - This way, RDD can always be re-created even upon a failure

RDD Operations

- Let A be an RDD, the following **3 operations** are possible:
  - **Transformations - >** generate a new RDD B from the data in A
  - **Actions - >** launch a computation on the data in A, which returns a value to the application
  - **Persistence - >** save the RDD in memory for later actions

RDD Operations: Transformations

- **Narrow:** each partition of A contributes at most to one partition of B (e.g., **`map`**)
  - No need to transfer data across nodes
- **Wide:** each partition of A may contribute to multiple partitions of B (e.g., **`groupBy`**)
  - Possible need to transfer data across nodes
- **Lazy evaluation:** nothing is computed unless required by an action

Narrow Transformations

In Apache Spark, transformations are classified as either **narrow** or **wide** depending on how data moves between partitions during execution.

Narrow Transformations

- Are operations where each partition of the input RDD is used by at most one partition of the output RDD.

- **Characteristics**:
  - Data dependencies are localized within partitions, which makes processing efficient.
  - Examples include transformations like map(), filter(), and mapPartitions().

- **Data Flow**: There is no need for data to be transferred across the cluster, so narrow transformations are generally faster.

Wide Transformations

- Are operations where each partition of the input RDD may be used by multiple partitions of the output

- **Characteristics**:
  - These transformations require data to be reshuffled across partitions, which involves network communication, making them more time-consuming.
  - Examples include transformations like reduceByKey(), groupByKey(), and join().

- **Data Flow**: Data needs to be moved across the nodes in the cluster, which involves combining and reorganizing data.

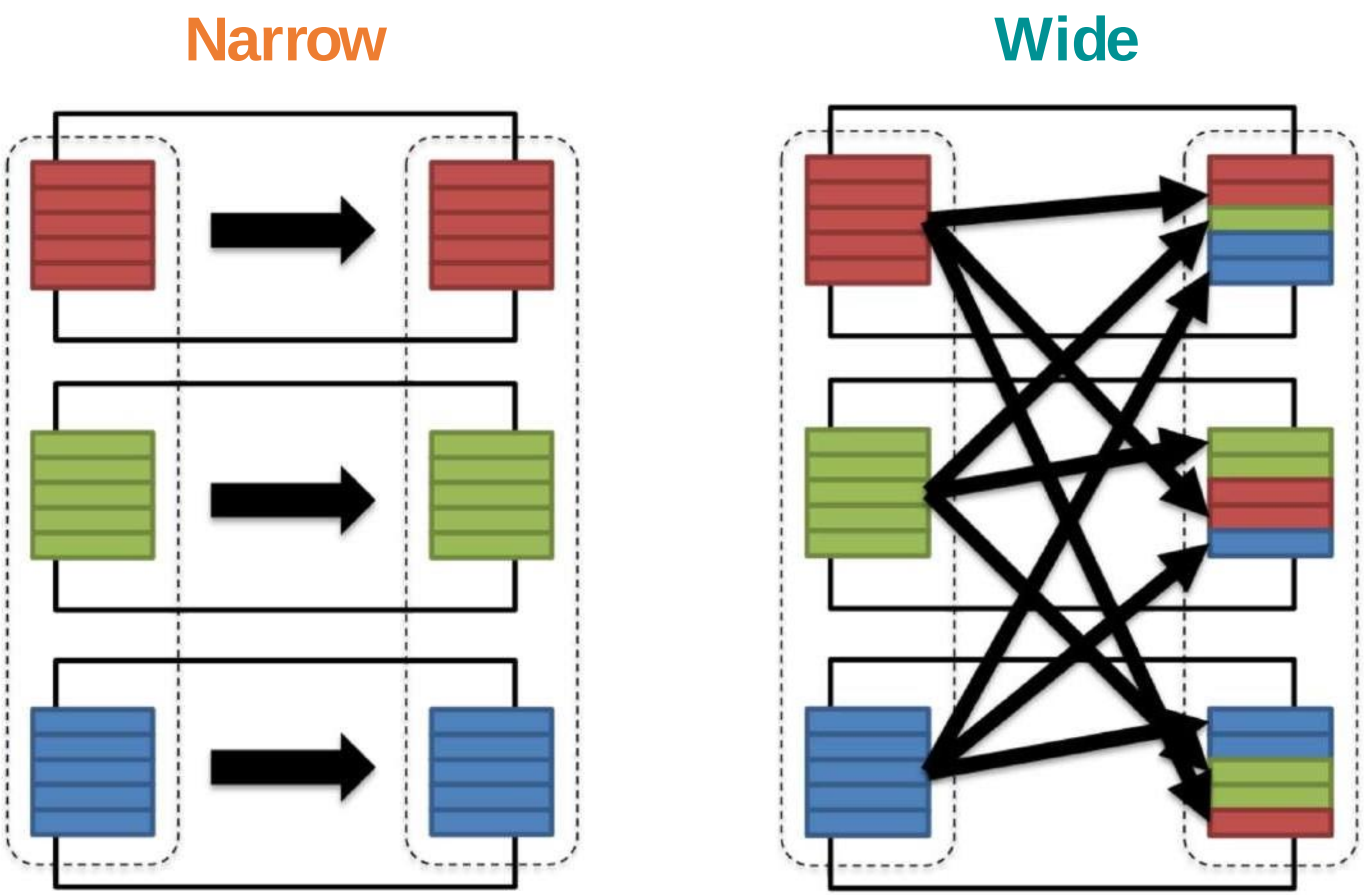
- **Use Cases**: Situations where data from different partitions must be brought together, for example, to perform aggregations or joins.

- **Summary**
  - Involve data shuffling and network I/O, making them more resource-intensive.
  - Efficient use of transformations in Spark involves minimizing wide transformations, as they typically represent a major bottleneck due to the need for shuffling data across the cluster.

Narrow vs. Wide Transformations

Input and output stay on the same partition

No data transfers



Input from other partitions may be needed

Data shuffling across nodes

RDD Operations: Actions

- **Example:** the count method returns the number of elements of the RDD
- When the action is called the RDD is actually materialized (lazy evaluation)

Spark DataFrame and Dataset APIs

- RDDs are the most basic data model used by Spark
  - low-level and schema-less
- On top of RDD API, **Spark SQL** module provides **2 interfaces** to operate on structured data like tables in relational databases:
  - **DataFrame API**
  - **Dataset API**

Spark: DataFrame

- Distributed collection of data organized into **named columns**

- Allows higher level abstraction than plain vanilla RDDs

- Similar to Pandas DataFrame unless few differences

- Dataset API is available only for Scala and Java as it extends DataFrame API with type-safe, object-oriented programming interface

Spark DataFrame vs. Pandas DataFrame

- Spark DataFrames are **immutable**: once created they cannot be modified
- As for RDDs, Spark may apply 2 kinds of operations on DataFrames: **transformations** and **actions**
- Lazy evaluation allows to queue transformations applied to elements of a DataFrame until an action is called
- DataFrame (and Dataset aswell) can be turned back to RDD

Spark vs. Hadoop MapReduce

- **Performance:** Spark is usually faster
  - In-memory data processing vs. data persistencing to disk after any map/reduce step
  - Spark requires lots of memory to run fast, otherwise its performance deteriorates
  - MapReduce integrates better with other services running
- **Ease of use:** Spark provides a higher-level API which is easier to program
- **Data processing:** Spark is more flexible and general

Useful Extra References

- Spark official https://spark.apache.org/
- What is Spark? [https://www.databricks.com/spark/about]
- Getting Started with Spark [https://www.databricks.com/spark/getting-started-with-apache-spark]
- Spark tutorial [https://data-flair.training/blogs/spark-tutorial/]
- PySpark Tutorial (video) [https://www.youtube.com/watch?v=_C8kWso4ne4]
- Docker con spark [https://hub.docker.com/_/spark]

Take-Home Message of Today

- Spark is a general-purpose distributed data processing engine which overcomes many of the Hadoop's limitations
- Spark provides a rich ecosystem of services to work on (big) data through APIs accessible via multiple programming languages
- Lazy execution avoids frequent and costly disk operations
- Spark's **DataFrame** as the main abstraction for playing with data

CODING with SPARK

Overview of some essential code in Spark

1. Setting Up a SparkSession

`SparkSession` is the entry point for data processing.

```
from pyspark.sql import SparkSession

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

- `appName("BasicSparkExample")`: Sets the name for the application.
- `getOrCreate()`: Either creates a new `SparkSession` or returns an existing one.

Overview of some essential code in Spark

2. Creating an RDD (Resilient Distributed Dataset)

RDDs are the core data structure in Spark.

```
# Create an RDD from a Python list
data = [1, 2, 3, 4, 5]
rdd = spark.sparkContext.parallelize(data)

# Show the contents of the RDD
print(rdd.collect())
```

- `spark.sparkContext.parallelize(data)`: Converts a Python list to an RDD.
- `rdd.collect()`: Collects all the elements of the RDD to the driver program.

- **Note**: For large datasets, avoid `collect()` as it might cause memory issues.

Overview of some essential code in Spark

- **3. Performing Transformations on an RDD**

Transformations create a new RDD from an existing one.

```
# Map transformation - multiply each element by 2
mapped_rdd = rdd.map(lambda x: x * 2)

# Filter transformation - keep only even numbers
filtered_rdd = mapped_rdd.filter(lambda x: x % 2 == 0)

# Collect the results
print(filtered_rdd.collect())  # Output: [4, 8]
```

- `map(lambda x: x * 2)`: Applies the given function to each element.
- `filter(lambda x: x % 2 == 0)`: Filters the elements based on the given condition.

Overview of some essential code in Spark

- **4. Loading Data Using `SparkSession`**

- Spark is often used to process large files, like CSVs or JSON.

```
# Load a CSV file
csv_path = "/path/to/your/data.csv"  # Replace with the path to your CSV file
df = spark.read.option("header", True).csv(csv_path)

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

- `.read.option("header", True).csv(csv_path)`: Reads a CSV file into a DataFrame with headers.
- `.show(5)`: Displays the first 5 rows of the DataFrame.

Overview of some essential code in Spark

- **5. Performing Transformations on DataFrames**

- Transformations can also be applied to DataFrames. Here are examples of selecting columns and filtering rows:

```
# Select specific columns
selected_df = df.select("name", "age")

# Filter rows where age is greater than 30
filtered_df = selected_df.filter(df.age > 30)

# Show the result
filtered_df.show()
```

- `.select("name", "age")`: Selects specific columns from the DataFrame.
- `.filter(df.age > 30)`: Filters rows based on the condition.

Overview of some essential code in Spark

- **6. Aggregating Data with GroupBy**

We can group data by a column and calculate aggregate metrics:

```
# Group by the "age" column and count the number of records for each age
grouped_df = df.groupBy("age").count()

# Show the result
grouped_df.show()
```

- `.groupBy("age").count()`: Groups the data by the `age` column and counts the occurrences for each unique age.

Overview of some essential code in Spark

- **7. Actions on DataFrames**

Actions trigger the actual computation in Spark.

```
# Count the number of rows in the DataFrame
row_count = df.count()
print(f"Number of rows: {row_count}")

# Collect data to the driver
collected_data = df.collect()
print(collected_data)
```

- `.count()`: Returns the number of rows in the DataFrame.
- `.collect()`: Collects the DataFrame rows to the driver as a list.
- Note: Be cautious when using `.collect()` for large datasets.

Overview of some essential code in Spark

- **8. Writing Data to Output**

You can save your DataFrame to various formats such as CSV :

```
# Save the filtered DataFrame as a CSV file
output_path = "/path/to/output/folder"
filtered_df.write.option("header", True).csv(output_path)
```

- `.write.option("header", True).csv(output_path)`: Saves the DataFrame as a CSV file to the specified path.

Summary

- SparkSession is the entry point for working with Spark.
- RDDs: Low-level data structure that you can create using `sparkContext` and apply transformations like `map` and `filter`.
- DataFrames: Higher-level abstraction for data, similar to tables. You can load data from files, perform column operations, group by values, and write results.
- Transformations (like `map()`, `filter()`, `groupBy()`) are lazy operations used to manipulate data.
- Actions (like `count()`, `collect()`) trigger actual computation.

Common Transformations in Spark

- In Apache Spark, transformations are operations that create a new dataset from an existing one.

- They are lazy, meaning that they don’t immediately compute their results.

- Instead, transformations are recorded and executed only when an action (like a count or collect) is called, which triggers the computation.

Common Transformations in Spark

1. map(func)

- Applies a function to each element of the RDD/DataFrame and returns a new RDD/DataFrame.

  - **Example**: Converting a list of numbers to their squares.

```
rdd = sc.parallelize([1, 2, 3, 4])
squared = rdd.map(lambda x: x ** 2)
        # Result: [1, 4, 9, 16]
```

2. flatMap(func)

- Similar to `map()`, but the output is flattened. Each input item can be mapped to 0 or more items, and the results are combined into a single RDD.

  - **Example**: Splitting lines of text into words.

```
rdd = sc.parallelize(["Hello world", "Spark is great"])
words = rdd.flatMap(lambda line: line.split(" "))
    # Result: ["Hello", "world", "Spark", "is", "great"]
```

Common Transformations in Spark

3. filter(func)

- Returns a new RDD/DataFrame containing only the elements that satisfy the given condition
- **Example**: Filtering even numbers.

```
rdd = sc.parallelize([1, 2, 3, 4, 5, 6])
even = rdd.filter(lambda x: x % 2 == 0)
    # Result: [2, 4, 6]
```

4. distinct()

- Returns a new RDD/DataFrame with duplicate elements removed.
- **Example**: Removing duplicates from a list.

```
rdd = sc.parallelize([1, 2, 2, 3, 4, 4, 5])
unique = rdd.distinct()
    # Result: [1, 2, 3, 4, 5]
```

Common Transformations in Spark

5. union(otherDataset)

Returns a new RDD/DataFrame that contains the union of the elements from the two datasets.

- **Example:** Combining two lists.

```
rdd1 = sc.parallelize([1, 2, 3])
rdd2 = sc.parallelize([4, 5, 6])
combined = rdd1.union(rdd2)
    # Result: [1, 2, 3, 4, 5, 6]
```

6. intersection(otherDataset)

- Returns a new RDD/DataFrame containing only the elements present in both datasets (intersection).
- **Example:** Finding common elements between two lists.

```
rdd1 = sc.parallelize([1, 2, 3, 4])
rdd2 = sc.parallelize([3, 4, 5, 6])
common = rdd1.intersection(rdd2)
    # Result: [3, 4]
```

Common Transformations in Spark

7. groupByKey()

- Groups all values associated with each key in an RDD of key-value pairs.
- **Example:** Grouping numbers by key.

```
rdd = sc.parallelize([("a", 1), ("b", 2), ("a", 3)])
grouped = rdd.groupByKey()
    # Result: [("a", [1, 3]), ("b", [2])]
```

8. reduceByKey(func)

- Combines values with the same key using the specified function.
- **Example:** Adding values for the same key.

```
rdd = sc.parallelize([("a", 1), ("b", 2), ("a", 3)])
summed = rdd.reduceByKey(lambda x, y: x + y)
    # Result: [("a", 4), ("b", 2)]
```

Common Transformations in Spark

9. join(otherDataset)

- Joins two RDDs/DataFrames by their keys.
- **Example:** Joining two datasets by common key.

```
rdd1 = sc.parallelize([("a", 1), ("b", 2)])
rdd2 = sc.parallelize([("a", 3), ("b", 4)])
joined = rdd1.join(rdd2)
    # Result: [("a", (1, 3)), ("b", (2, 4))]
```

10. cartesian(otherDataset)

- Returns the Cartesian product of two datasets.
- **Example:** Generating all possible pairs.

```
rdd1 = sc.parallelize([1, 2])
rdd2 = sc.parallelize([3, 4])
product = rdd1.cartesian(rdd2)
    # Result: [(1, 3), (1, 4), (2, 3), (2, 4)]
```

Common Transformations in Spark

11. sortBy(func, ascending=True)

- Sorts the RDD by the given function.
- **Example:** Sorting numbers in ascending order.

```
rdd = sc.parallelize([4, 2, 7, 1])
sorted_rdd = rdd.sortBy(lambda x: x)
        # Result: [1, 2, 4, 7]
```

12. sample(withReplacement, fraction)

- **Description**: Returns a sampled subset of the dataset.
- **Example:** Sampling elements from an RDD.

```
rdd = sc.parallelize(range(10))
sampled_rdd = rdd.sample(False, 0.4)
        # Sample about 40% of the elements
```

Scenario 1

- We have a dataset containing customer transactions (transactions.csv), and we want to:
  1. Load the dataset.
  2. Filter transactions to only include purchases over $100.
  3. Group by customer ID and sum their total purchase amounts.
  4. Output the result.

- **Sample Data (transactions.csv):**
- Each log entry is a space-separated string with the following format:
  transaction_id,customer_id,amount,timestamp

```
1,101,150.75,2023-09-20 10:15:00
2,102,80.50,2023-09-20 11:20:00
3,101,200.00,2023-09-21 09:45:00
4,103,55.00,2023-09-21 12:10:00
5,104,500.30,2023-09-22 15:25:00
```

Generate Random Data on a CSV file

```
import csv
import random
from datetime import datetime, timedelta
import uuid

# Numero di transazioni da generare
num_transactions = 1000

# Lista per memorizzare le transazioni
transactions = []

# Data di inizio per le timestamp (ultimi 30
giorni)
end_date = datetime.now()
start_date = end_date - timedelta(days=30)
```

```
# Generazione delle transazioni
for _ in range(num_transactions):
    transaction = {
        'transaction_id': str(uuid.uuid4()),  # ID univoco
per ogni transazione
        'customer_id': random.randint(1000, 9999),  #
ID cliente casuale
        'amount': round(random.uniform(10.0, 1000.0),
2),  # Importo casuale tra 10 e 1000
        'timestamp': (start_date + timedelta(
            seconds=random.randint(0, int((end_date -
start_date).total_seconds()))
        )).strftime('%Y-%m-%d %H:%M:%S')
    }
    transactions.append(transaction)
```

Generate Random Data on a CSV file

```
# Ordinamento delle transazioni per timestamp
transactions.sort(key=lambda x: x['timestamp'])

# Scrittura su file CSV
filename = 'transactions.csv'
with open(filename, 'w', newline='') as csvfile:
    fieldnames = ['transaction_id', 'customer_id', 'amount', 'timestamp']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

    writer.writeheader()
    for transaction in transactions:
        writer.writerow(transaction)

print(f"File {filename} generato con successo con {num_transactions} transazioni!")
```

Implementation

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F

# Initialize Spark session
spark = SparkSession.builder.appName("CustomerTransactionsAnalysis").getOrCreate()

# Step 1: Load the dataset
df = spark.read.csv("transactions.csv", header=True, inferSchema=True)

# Step 2: Filter transactions where the amount is greater than 100
filtered_df = df.filter(df['amount'] > 100)
```

Implementation

```
# Step 3: Group by customer ID and sum the total amount spent
grouped_df = filtered_df.groupBy("customer_id").agg(F.sum("amount").alias("total_spent"))
    # The .agg() function is used to perform an aggregate operation on the grouped data.
    # Here, F.sum("amount") calculates the total of the "amount" column for each group of customer_id.
    # F is an alias for the pyspark.sql.functions module, which provides a set of functions for manipulating and
    transforming data.
    # alias("total_spent") is used to rename the resulting column to "total_spent".

# Step 4: Show the final result
grouped_df.show()

# (Optional) Save the result to a CSV file
grouped_df.write.csv("customer_totals.csv", header=True)

# Stop the Spark session
spark.stop()
```

Scenario 2 (Machine Learning MLlib)

- You have a dataset of customer information (e.g., customer_data.csv) and you want to predict whether a customer will buy a product (binary outcome: 1 for purchase, 0 for no purchase) based on features like age, income, and gender

- **Sample customer_data.csv:**
  - Each log entry is a come-separated string with the following format:

```
age,income,gender,label

25,50000,0,0
32,60000,1,1
47,120000,0,1
51,130000,1,0
30,55000,0,0
```

Implementation

```
from pyspark.sql import SparkSession
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator

# Step 1: Initialize Spark session
spark = SparkSession.builder.appName("LogisticRegressionExample").getOrCreate()

# Step 2: Load the data
data = spark.read.csv("customer_data.csv", header=True, inferSchema=True)

# Step 3: Assemble features into a single vector
assembler = VectorAssembler(inputCols=["age", "income", "gender"], outputCol="features")
data_with_features = assembler.transform(data)
final_data = data_with_features.select("features", "label")
```

Implementation

```
# Step 4: Split the data into training and test sets
train_data, test_data = final_data.randomSplit([0.8, 0.2])

# Step 5: Train a Logistic Regression model
logistic_regression = LogisticRegression(featuresCol='features', labelCol='label')
lr_model = logistic_regression.fit(train_data)

# Step 6: Make predictions on the test set
test_results = lr_model.transform(test_data)

# Step 7: Evaluate the model performance
evaluator = BinaryClassificationEvaluator(labelCol="label", rawPredictionCol="rawPrediction")
accuracy = evaluator.evaluate(test_results)
print(f"Test Accuracy = {accuracy}")

# (Optional) Save the model
lr_model.save("logistic_regression_model")

# Stop the Spark session
spark.stop()
```

Scenario 3: Word Count with Spark Streaming

- To create a word count program using Apache Spark Streaming in Python, you'll first need to set up a real-time data stream and then count words as they appear.

- In this example we make use of the socketTextStream() method, which listens to data on a socket.

- We can send text data to that socket through a terminal using **nc -lk <port>**

- In Spark Streaming, the SparkContext is typically used instead of SparkSession.

Scenario 3: Word Count with Spark Streaming

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

# Create a local StreamingContext with two working threads and batch interval of 1 second
sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1)

# Connect to hostname:port, e.g., localhost:9999
lines = ssc.socketTextStream("localhost", 9999)

# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))

# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
word_counts = pairs.reduceByKey(lambda x, y: x + y)

# Print the word counts
word_counts.pprint()
# Start computation and wait for the streaming to finish
ssc.start()
ssc.awaitTermination()
```

Scenario 3: **Word Count with Spark Streaming**

Step-by-Step Explanation:

1. **Imports:**
   1. Import the necessary Spark components (SparkContext, StreamingContext).
2. **Setup Contexts:**
   1. Create a SparkContext and a StreamingContext.
   2. The StreamingContext has a batch interval of 1 second, meaning it processes data in 1-second chunks.
3. **Create the Stream:**
   1. ssc.socketTextStream("localhost", 9999) sets up the stream to receive data on port 9999.
4. **Process the Data:**
   1. flatMap(lambda line: line.split(" ")) splits each line into individual words.
   2. map(lambda word: (word, 1)) pairs each word with the value 1.
   3. reduceByKey(lambda x, y: x + y) counts the occurrences of each word.
5. **Output:**
   1. pprint() prints the word counts.
6. **Start Streaming:**
   1. ssc.start() starts the computation.
   2. ssc.awaitTermination() keeps the program running until you manually stop it.

Scenario 3: **Word Count with Spark Streaming**

Running the Application

Start a socket listener using Netcat:

```
nc -lk 9999
```

This command will open a terminal where you can type messages that Spark will process.

Run the Spark script using:

```
spark-submit word_count.py
```

Scenario 4: Spark SQL

This is a simple example of using Spark SQL in Python to create a DataFrame, register it as a temporary table, and then query it using SQL.

```
from pyspark.sql import SparkSession
# Initialize SparkSession
spark = SparkSession.builder .appName("Spark SQL Example") .getOrCreate()

# Create a DataFrame with sample data
data = [
    ("Alice", 34),
    ("Bob", 45),
    ("Cathy", 29),
    ("David", 40)
]
```

Scenario 4: Spark SQL

```
columns = ["Name", "Age"]

df = spark.createDataFrame(data, columns)

# Register the DataFrame as a temporary SQL table
df.createOrReplaceTempView("people")

# Use Spark SQL to select people with age greater than 30
result = spark.sql("SELECT Name, Age FROM people WHERE Age > 30")

# Show the result
result.show()

# Stop the Spark session
spark.stop()
```

Scenario 4: **Spark SQL**

Explanation

1. **Initialize SparkSession**: We create a SparkSession which allows us to interact with Spark.

2. **Create DataFrame**: A DataFrame is created with sample data (data and columns).

3. **Register Temporary Table**: The DataFrame is registered as a temporary SQL table using createOrReplaceTempView.

4. **Query with Spark SQL**: We run an SQL query using the spark.sql method to select rows where Age is greater than 30.

5. **Show the result**: The result.show() function prints the output.