DE235 Soumyadeep Sinha
WEEK 3 - DAY 2 - 13-11-2024

PySPARK

BASIC TERMINOLOGIES

Hadoop vs. Spark Data Processing:

- Hadoop reads and processes data from disk, which is relatively slow.
- Spark uses RAM to store data, enabling faster processing.

Spark Architecture:

- Spark operates on a master-slave architecture.
- It distributes data across a cluster and processes it in parallel over the nodes.

Driver Program:

- Initiates the execution of the main program.
- Responsible for creating the Spark Context.

Spark Context and Spark Session:

- **Spark Context**: Entry point for Spark that helps in creating RDDs (Resilient Distributed Datasets).
- **Spark Session**: Includes Spark Context, SQL Context, Streaming Context, and Hive Context.

Cluster Manager:

- Responsible for acquiring resources on the cluster.
- Receives resource requests from the Driver Program.

Executors:

- Executors are Java processes launched on worker nodes.
- They handle tasks (units or chunks of data) sent from the Driver Program.

Job:

- A job represents a process of parallel computation within Spark.

Important Methods and Setup

```
# Import SparkContext from PySpark
from pyspark.context import SparkContext
```

```
# Create a SparkContext with an application name
sc = SparkContext(appName="Spark-RDD")
```

from pyspark.context import SparkContext: This line imports the **`SparkContext`** class from PySpark, which is required to interact with the Spark environment.


```
# Create a SparkContext with an application name
sc = SparkContext(appName="Spark-RDD")
```


- **SparkContext(appName="Spark-RDD")**: Here, a **`SparkContext`** instance named **`sc`** is created.
    - The **`appName`** parameter specifies a name for the application ("Spark-RDD" in this case), which helps identify the job when it's running in a Spark cluster or in logs.
    - This context (`sc`) allows us to create and interact with Resilient Distributed Datasets (RDDs), perform actions, and run transformations across the Spark cluster.

Purpose of `SparkContext`

The **`SparkContext`** is essential because it:

1. Acts as the **connection to the Spark execution environment**.
2. **Manages configurations** and resource allocation.
3. Allows the creation of RDDs, which are fundamental data structures for distributed processing in Spark.

Creating the **`SparkContext`** is typically the first step in any Spark application, and it enables us to work with RDDs and Spark's distributed data processing features.

```
# Display the version of Spark and Python
sc.version,sc.pythonVer
```

sc.version: This returns the version of Spark installed in our environment. It's useful for verifying the Spark version, especially if we're working in a multi-version environment or testing compatibility.

sc.pythonVer: This returns the version of Python that Spark is using. Since PySpark requires a compatible Python version, this helps ensure that the environment is correctly configured.

```
# Display the application name and application ID
sc.appName,sc.applicationId
```

sc.appName: This returns the name of the application that was set when the **`SparkContext`** was created (e.g., **`"Spark-RDD"`** in our previous example). This is useful for identifying the specific job when multiple applications are running.

sc.applicationId: This returns a unique identifier for the current Spark application. The **`applicationId`** is generated by the cluster manager (e.g., YARN or Spark Standalone) and is useful for tracking, monitoring, and debugging the application in cluster logs or the Spark UI.

```
# Check the default number of parallel tasks (partitions)
sc.defaultParallelism
```

sc.defaultParallelism: This property returns the default number of partitions that Spark will use for distributed operations. It determines how data and tasks are divided across the cluster, influencing the efficiency of parallel processing.

Determining Factor: The default parallelism level is generally based on the number of cores available in the cluster. For example, in local mode (e.g., `master="local[4]"`), `defaultParallelism` will be equal to the number of cores (4 in this case). In a cluster setup, it might depend on the configuration of the cluster manager and the resources available.

```
# Create an RDD with integers 1 through 6, divided into two partitions
rdd1=sc.parallelize([1,2,3,4,5,6])
```

sc.parallelize([1, 2, 3, 4, 5, 6]): The `parallelize` method creates an RDD from the provided list of integers `[1, 2, 3, 4, 5, 6]`.

- By default, Spark will divide this RDD into partitions based on the `defaultParallelism` setting or based on the cluster's capabilities.
- we can specify the number of partitions explicitly by passing a second argument, e.g., `sc.parallelize([1, 2, 3, 4, 5, 6], 2)` for two partitions.

RDD: This RDD (`rdd1`) can now be used to perform distributed computations across the cluster, with the data divided into partitions. Each partition can be processed in parallel, making operations on this RDD faster and more efficient.

```
rdd1.collect()
```

rdd1.collect(): This command returns all the elements in `rdd1`, the RDD we created, as a single list on the driver node. In this case, the output will be:

```
[1, 2, 3, 4, 5, 6]
```

When to Use `collect()`: `collect()` is useful for small datasets where retrieving all elements to the driver is manageable. However, for large datasets, avoid using `collect()` as it can overwhelm the driver's memory, leading to potential memory errors.

```
rdd1.getNumPartitions()
```

rdd1.getNumPartitions(): This command outputs the number of partitions that **rdd1** is divided into.

Why Partitions Matter: Partitions determine how the data is distributed across the cluster and processed in parallel. More partitions can improve parallelism but may add overhead if there are too many small tasks, while fewer partitions can limit parallelism and potentially slow down execution.

```
# Print min, max, sum, and mean of values in rdd1
print(f' Min value :{rdd1.min()}\n Max Value : {rdd1.max()} \n sum of values :{rdd1.sum()}\n
```

```
mean of value :{rdd1.mean()}')
```

`rdd1.min()`: Returns the minimum value in `rdd1`.
`rdd1.max()`: Returns the maximum value in `rdd1`.
`rdd1.sum()`: Calculates the sum of all values in `rdd1`.
`rdd1.mean()`: Calculates the mean (average) of the values in `rdd1`.

```
# Define a lambda function to square each element
square =lambda x:(x,x**2)

# Apply map transformation to rdd1 to create a new RDD with tuples (value, value^2)
rdd2=rdd1.map(square)

rdd2.collect()
```

Output :

[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36)]

```
rdd3 = rdd2.flatMap(lambda x: x)
```

rdd2: This RDD presumably contains tuples or lists of values, for example, **`[(1, 2), (3, 4), (5, 6)]`**.

flatMap(lambda x: x): The `flatMap` transformation is similar to `map`, but it flattens the results. Here, `lambda x: x` simply returns each element within the tuple or list in `rdd2` as an individual element in `rdd3`.

- For example, if `rdd2` contains `[(1, 2), (3, 4), (5, 6)]`, applying `flatMap(lambda x: x)` would produce `rdd3` with elements `[1, 2, 3, 4, 5, 6]`.

Result in `rdd3`: After this transformation, `rdd3` will be a flattened RDD where each value from the tuples in `rdd2` is now a standalone element.

```
# Get distinct values from rdd3
rdd3.distinct().collect()
```

FILTER

```
even_value=lambda x:x%2==0
# Filter rdd3 to keep only even numbers
rdd3.filter(even_value).collect()
```

rdd3.filter(lambda x: x % 2 == 0): The **`filter`** transformation takes a function that returns **`True`** for values to keep. Here, **`lambda x: x % 2 == 0`** checks if each element **`x`** in **`rdd3`** is even.
collect(): After filtering, **`collect()`** gathers all the filtered results back to the driver as a list.

```
# Use glom() to group data in each partition into a list
rdd3.glom().collect()
```

rdd3.glom(): **`glom()`** transforms each partition in **`rdd3`** into a list of its elements. So, if **`rdd3`** has multiple partitions, the result will be an RDD where each element is a list of items in a specific partition.

collect(): This action gathers the grouped lists from each partition to the driver as a single list of lists.

```
indian_cars = [
    "Maruti Suzuki Alto", "Tata Tiago", "Hyundai i10", "Renault Kwid", "Maruti Suzuki Swift",
    "Honda City", "Maruti Suzuki Ciaz", "Hyundai Verna", "Tata Tigor", "Skoda Slavia",
    "Mahindra Scorpio", "Tata Harrier", "Maruti Suzuki Brezza", "Hyundai Creta", "Kia Seltos"
]
```

```
# Parallelize the car list into an RDD with 4 partitions
indian_cars_rdd=sc.parallelize(indian_cars,4)
```

```
# Use glom() to view the data in each partition
for i in indian_cars_rdd.glom().collect():
    print(i)
```

```
# Repartition the RDD into 6 partitions
indian_cars_rdd1=indian_cars_rdd.repartition(6)
```

indian_cars_rdd.repartition(6): The **`repartition`** method reshuffles the data in the RDD and evenly distributes it across the specified number of partitions (6 in this case). This results in a new RDD, **`indian_cars_rdd1`**, with 6 partitions.
Why Repartition?:

- Repartitioning can improve performance for operations that benefit from a larger or smaller number of partitions.
- Increasing partitions can improve parallelism, especially for large datasets.
- Decreasing partitions (using **`coalesce`** instead) is useful if you want to reduce overhead from having too many small tasks.

```
# Perform a reduce operation on rdd3 to sum all the elements
result = rdd3.reduce(lambda a, b: a + b)
print(type(result),result)
```

rdd3.reduce(lambda a, b: a + b): The **`reduce`** function applies a specified binary operation (in this case, **`lambda a, b: a + b`**) across all elements in the RDD. This operation reduces the RDD to a single result by repeatedly combining elements.

- Here, **`lambda a, b: a + b`** adds two elements at a time until all elements in **`rdd3`** are summed up.

print(type(result), result): This line prints the type and the value of the result. Since `reduce` returns a single aggregated value, `result` will be an integer (or float, if any element is a float) representing the sum of all elements in `rdd3`.

```
# Creating a key-value RDD
pairs_rdd = sc.parallelize([("Maruti Suzuki Alto", 3), ("Hyundai i10", 1), ("Hyundai i10", 2),
("Tata Harrier", 2),("Maruti Suzuki Alto", 10),("Hyundai i10", 15)])
```

sc.parallelize(...): The `parallelize` method creates an RDD from the list of tuples.
Each element in this RDD is a **tuple** where:

- The first item is the **key** (the car model, e.g., `"Maruti Suzuki Alto"`).
- The second item is the **value** associated with that key (e.g., `3, 1, 2`).

```
# Reduce by key to sum values for each key
reduceByKey = pairs_rdd.reduceByKey(lambda a, b: a + b)
print(type(reduceByKey),reduceByKey.collect())
```

pairs_rdd.reduceByKey(lambda a, b: a + b): `reduceByKey` applies a specified binary operation (in this case, `lambda a, b: a + b`) to values grouped by each unique key.

- For each key (e.g., `"Maruti Suzuki Alto"`), Spark will sum all associated values (e.g., `3 + 10`).
- This operation is done in parallel across partitions, making it efficient for large datasets.

print(type(reduceByKey), reduceByKey.collect()): This line prints the type of `reduceByKey` (which is an RDD) and uses `collect()` to gather the results to the driver as a list of key-value pairs.

```
# Group by key
grouped = pairs_rdd.groupByKey().mapValues(list)
print(type(grouped),grouped.collect())
```

pairs_rdd.groupByKey(): `groupByKey` groups all values associated with each key into an iterable collection for that key.

- For example, all values associated with `"Maruti Suzuki Alto"` will be grouped together.

.mapValues(list): Since `groupByKey` returns an iterable for each key, `mapValues(list)` converts each iterable into a list, making the output more readable and easier to work with.
print(type(grouped), grouped.collect()): This line prints the type of `grouped` (an RDD) and uses `collect()` to gather the results to the driver as a list of key-value pairs.

```
states = {"NY":"New york", "CA":"California", "FL":"Florida"}
print(states.items())
print(states.keys())
```

```
# Parallelize the dictionary into an RDD
```

```
states_rdd = sc.parallelize(states.items())
print(states_rdd.collect(),type(states_rdd.collect()))
```

```
data = [("James","Smith","USA","CA"),
    ("Michael","Rose","USA","NY"),
    ("Robert","Williams","USA","CA"),
    ("Maria","Jones","USA","FL")
  ]
data_rdd = sc.parallelize(data)
```

```
data_rdd.collect()
```

```
# Map the RDD to create a key-value RDD where the state code is the key
rdd_keyed = data_rdd.map(lambda x: (x[3], x))
rdd_keyed.collect()
```

Output :

```
[('CA', ('James', 'Smith', 'USA', 'CA')),
 ('NY', ('Michael', 'Rose', 'USA', 'NY')),
 ('CA', ('Robert', 'Williams', 'USA', 'CA')),
 ('FL', ('Maria', 'Jones', 'USA', 'FL'))]
```

```
# Perform a join operation with states_rdd to add state names
joined_rdd = rdd_keyed.join(states_rdd)
```

rdd_keyed.join(states_rdd): The **`join`** transformation combines two key-value RDDs based on matching keys.

- For each key that exists in both **`rdd_keyed`** and **`states_rdd`**, the join operation will pair the values from both RDDs.
- The result is an RDD where each element is a tuple with the format **`(key, (value_from_rdd_keyed, value_from_states_rdd))`**.

```
joined_rdd.collect()
```

```
[('CA', (('James', 'Smith', 'USA', 'CA'), 'California')),
 ('CA', (('Robert', 'Williams', 'USA', 'CA'), 'California')),
 ('NY', (('Michael', 'Rose', 'USA', 'NY'), 'New york')),
 ('FL', (('Maria', 'Jones', 'USA', 'FL'), 'Florida'))]
```

```
joined_rdd.map(lambda x: (*x[1][0][:-1],x[1][1])).collect()
```

lambda x: This lambda function is applied to each element of **`joined_rdd`**, where each element is a tuple in the form **`(key, (value_from_rdd_keyed, value_from_states_rdd))`**.

x[1][0][:-1]: This part extracts the value from **`rdd_keyed (x[1][0])`** and then uses **`[:-1]`** to remove the last character of this string or list, depending on its structure.

- If `x[1][0]` is a string, `[:-1]` removes the last character.
- If `x[1][0]` is a list or tuple, `[:-1]` removes the last element.

(*x[1][0][:-1], x[1][1]): The `*` unpacks the modified value from `x[1][0][:-1]` into individual elements, and `x[1][1]` appends the value from `states_rdd` as the last element of the resulting tuple.

```
# Broadcast the states dictionary to optimize lookup during transformations
broadcastStates = sc.broadcast(states)

# Create a function that retrieves the full state name based on the state code
def state_convert(code):
    return broadcastStates.value[code]

# Map the data_rdd to replace the state code with the full state name
result = data_rdd.map(lambda x: (*x[:2],state_convert(x[3])))
```

Broadcasting the States Dictionary:

- **broadcastStates = sc.broadcast(states)**: This line broadcasts the `states` dictionary, which makes it available as read-only data on each executor. Broadcasting helps optimize large datasets by avoiding sending the dictionary repeatedly with each task.

Define the `state_convert` Function:

- **def state_convert(code)**: This function accepts a `code` (e.g., a state code) and retrieves the corresponding state name by accessing `broadcastStates.value`, the broadcasted dictionary.

Map Transformation to Replace State Codes:

- **data_rdd.map(lambda x: (*x[:2], state_convert(x[3])))**:
  - The `map` transformation applies a function to each element of `data_rdd`.
  - For each element `x, (*x[:2], state_convert(x[3]))` unpacks the first two elements in `x` as they are (`*x[:2]`), and replaces the state code at position `x[3]` with the full state name using `state_convert(x[3])`.
- The result is a transformed RDD with the original data structure, except with the state code replaced by the full state name.

CONVERTING RDD INTO DATAFRAME

```
# To convert the RDD into a DataFrame, we first need to initialize a SparkSession
from pyspark.sql import SparkSession
spark= SparkSession.builder.getOrCreate()
```

- **SparkSession**:
  - The `SparkSession` is the entry point to programming with DataFrames in Spark. It allows you to interact with Spark's various data processing capabilities (e.g., SQL queries, DataFrame APIs, etc.).

    - Before Spark 2.x, you needed to use `SQLContext` or `HiveContext` to work with DataFrames. Since Spark 2.0, `SparkSession` provides a unified entry point for all Spark functionalities, including DataFrames, SQL, streaming, and more.
- `SparkSession.builder.getOrCreate()`:
    - This method either retrieves the existing `SparkSession` if it's already created, or creates a new one if none exists.
    - `builder` is a configuration builder that allows you to configure settings for the session, such as enabling Hive support, setting app names, and more.
    - `getOrCreate()` ensures that only one `SparkSession` is created per application.

Why SparkSession?

A `SparkSession` encapsulates the functionality provided by `SparkContext` and `SQLContext` into one object. It simplifies the setup process when working with DataFrames and is the preferred way to interact with Spark's high-level API.

```
# Convert the RDD to a DataFrame
result.toDF().collect()
```

`result.toDF()`:

- This method converts the RDD (`result` in this case) into a **DataFrame**.
- For the conversion to work properly, your RDD should contain data that can be converted into a structured format (i.e., rows of columns).
- If the RDD consists of key-value pairs (i.e., tuples), you can convert them into a DataFrame. For example, if your RDD contains `(column_name, value)` pairs, Spark will infer or allow you to define the column names explicitly.

`.collect()`:

- This collects the DataFrame data from all partitions and brings it to the driver as a list of rows.
- It's important to use `.collect()` cautiously, especially when working with large datasets, as it can overwhelm the driver if the data is too large.

```
result.toDF().show()
```

`.show()`:

- This method is used to display the top 20 rows of the DataFrame in a tabular format on the console (by default).
- It provides a quick view of the data in a DataFrame, useful for checking the output or verifying transformations.
- You can specify the number of rows to display by passing an argument to `show()`. For example, `show(10)` will display the top 10 rows.

```
rdd_df=result.toDF(['first_name','last_name','state'])
```

`result.toDF(['first_name', 'last_name', 'state'])`:

- This method converts the RDD (`result`) into a DataFrame.
- The list `['first_name', 'last_name', 'state']` provides the column names for the resulting DataFrame.
- This ensures that each column in the DataFrame is labeled as `first_name`, `last_name`, and `state`, instead of default column names like `_1, _2`, etc.

```
rdd_df.printSchema()
```

`printSchema()`:

- This method prints the **schema** of a DataFrame, which includes the names of the columns and the types of data they hold (e.g., `StringType, IntegerType`, etc.).
- It helps you understand the structure of the data, which is particularly useful when you're working with large datasets or complex transformations.

The schema gives information about:

- **Column Names**: The names of the fields in the DataFrame.
- **Data Types**: The data types of each column (e.g., `StringType, IntegerType`, etc.).

```
rdd_df.dtypes
```

OUTPUT

```
[('first_name', 'string'), ('last_name', 'string'), ('state', 'string')]
```

```
data = [
   (10, "Aarav"),
   (11, "Vivaan"),
   (12, "Aditya"),
   (13, "Diya"),
   (14, "Anaya"),
   (15, "Ishaan"),
   (16, "Meera"),
   (17, "Rohan"),
   (18, "Lakshmi"),
   (19, "Neha")
]
```

```
df1=spark.createDataFrame(data,'id string , name string')
```

```
df1.show()
```

```
+---+-------+
| id|   name|
+---+-------+
| 10|  Aarav|
| 11| Vivaan|
| 12| Aditya|
| 13|   Diya|
| 14|  Anaya|
| 15| Ishaan|
| 16|  Meera|
| 17|  Rohan|
| 18|Lakshmi|
| 19|   Neha|
+---+-------+
```

```
data_dict=[
   {'user_id': 10, 'name': 'Aarav'},
   {'user_id': 11, 'name': 'Vivaan'},
   {'user_id': 12, 'name': 'Aditya'},
   {'user_id': 13, 'name': 'Diya'},
   {'user_id': 14, 'name': 'Anaya'},
   {'user_id': 15, 'name': 'Ishaan'},
   {'user_id': 16, 'name': 'Meera'},
   {'user_id': 17, 'name': 'Rohan'},
   {'user_id': 18, 'name': 'Lakshmi'},
   {'user_id': 19, 'name': 'Neha'}
]
```

```
spark.createDataFrame(
   [tuple(i.values()) for i in data_dict]
   ,'user_id string,user_name string'
   ).show()
```

`[tuple(i.values()) for i in data_dict]`:

- This is a **list comprehension** that converts each dictionary in **`data_dict`** to a **tuple** of its values.
- **`i.values()`** returns the values of the dictionary `i` (i.e., the user IDs and names). The **`tuple()`** function is used to convert the dictionary values into tuples.

`spark.createDataFrame([...], 'user_id string, user_name string')`:

- This creates a **DataFrame** from the list of tuples. The column names and types are explicitly defined in the second argument as **`'user_id string, user_name string'`**.
- This means the DataFrame will have two columns: **`user_id`** of type **`string`** and **`user_name`** of type **`string`**.

PYSPARK SQL ROW

```
from pyspark.sql import Row
```

`Row`:

- The **`Row`** class is part of the **`pyspark.sql`** module, and it is used to create **Row objects**.
- A **`Row`** object represents a **single record** in a DataFrame. You can think of it as an unordered collection of named fields (i.e., a row with columns).
- You can create a **`Row`** by passing **column names** and **values** as keyword arguments or as a list.

```
Row((1,2))
```

`Row((1, 2))`:

- This creates a **`Row`** object, but with the values provided inside a **single tuple**.
- The tuple **`(1, 2)`** becomes a single field within the **`Row`** object.
- This approach means the row will be treated as a single column with a tuple as its value.

```
spark.createDataFrame([Row(*i.values()) for i in data_dict],'id int , name string').show()
```

`*` operator is used to unpack the values into separate arguments for **`Row`**.

```
[Row(**i) for i in data_dict]
```

`Row(**i)` creates a **`Row`** object where the keys of the dictionary become the **column names** and the values become the **data values**.

SUMMARY

Apache Spark Overview: We learned that Spark uses RAM for data storage and processing, making it faster than Hadoop, which relies on disk storage.

Spark Architecture: We explored Spark's master-slave architecture, where the **driver program** coordinates tasks and the **cluster manager** allocates resources.

RDD Basics: We worked with **Resilient Distributed Datasets (RDDs)**, the fundamental data structure in Spark, and applied transformations like **`map()`**, **`filter()`**, and **`flatMap()`**.

Actions on RDDs: We learned about actions like **`min()`**, **`max()`**, **`sum()`**, and **`collect()`**, which trigger computation on the RDD and return results.

Parallelism and Partitions: We explored how Spark distributes data across multiple **partitions** and used operations like **`repartition()`** to control parallelism.

Aggregation with RDDs: We used operations like **`reduceByKey()`**, **`groupByKey()`**, and **`join()`** to aggregate and combine data in RDDs.

Broadcast Variables: We optimized lookups by using **broadcast variables**, which store data on all worker nodes to avoid repeated computation.

DataFrames in Spark: We created DataFrames from RDDs and manually defined schemas, converting RDDs into a more structured form for efficient querying.

Row Objects: We worked with **Row()** objects, converting dictionaries into rows, and used `**` to unpack dictionary keys and values into named columns.

SQL Operations: We performed SQL-like operations on DataFrames, such as `join()` and `show()`, to manipulate and display structured data efficiently.
