

LAB1



**MALMÖ
UNIVERSITY**

Big Data Analytics on Cloud Computing Infrastructures

Azad Shokrollahi, Mahtab Jamali

In this lab, first we submit job on spark and then focus on the PySpark, Spark SQL, and DataFrames

1)How to submit job on Spark

1. Based on installation file, go to **spark/spin folder** and active one master and one worker.
2. After that go to **bin** folder in **spark folder**, and submit the Spark job according to following instructions:

- i. **Create python file using nano**
- ii. **Name of file is example1**

```
from pyspark.sql import SparkSession

if __name__ == "__main__":
    # Initialize a Spark session
    spark = SparkSession.builder \
        .appName("Simple Spark DataFrame Example") \
        .getOrCreate()

    # Create a DataFrame from a list of numbers
    numbers_list = [1, 2, 3, 4, 5]
    numbers_df = spark.createDataFrame([(t,) for t in numbers_list],
["number"])

    # Show the DataFrame
    numbers_df.show()

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

- iii. **Copy the code**
- iv. **use spark-submit to submit your Spark job.**


```
./spark-submit --master spark://<master-ip>:<master-port> <application-python-file> [application-arguments]
```

```

spark@spark-Virtual-Machine:/opt/spark/bin$ ./spark-submit --master spark://spark-Virtual-Machine:7077 example1.py
23/09/06 09:45:51 WARN Utils: Your hostname, spark-Virtual-Machine resolves to a loopback address: 127.0.1.1; using 172.26.83.47 instead (on interface eth0)
23/09/06 09:45:51 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
23/09/06 09:45:51 INFO SparkContext: Running Spark version 3.4.1
23/09/06 09:45:51 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
23/09/06 09:45:51 INFO ResourceUtils: =====
23/09/06 09:45:51 INFO ResourceUtils: No custom resources configured for spark.driver.
23/09/06 09:45:51 INFO ResourceUtils: =====
23/09/06 09:45:51 INFO SparkContext: Submitted application: Simple Spark DataFrame Example
23/09/06 09:45:51 INFO ResourceProfile: Default ResourceProfile created, executor resources: Map(memory -> name: memory, amount: 1024, script: , vendor: , offHeap: , task resources: Map(cpu -> name: cpus, amount: 1.0)
23/09/06 09:45:51 INFO ResourceProfile: Limiting resource is cpu
23/09/06 09:45:51 INFO ResourceProfileManager: Added ResourceProfile id: 0
23/09/06 09:45:51 INFO SecurityManager: Changing view acls to: spark
23/09/06 09:45:51 INFO SecurityManager: Changing modify acls to: spark
23/09/06 09:45:51 INFO SecurityManager: Changing view acls groups to:
23/09/06 09:45:51 INFO SecurityManager: Changing modify acls groups to:
23/09/06 09:45:51 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: spark; groups with view permissions:
23/09/06 09:45:51 INFO Utils: Successfully started service 'sparkDriver' on port 41673.
23/09/06 09:45:51 INFO SparkEnv: Registering MapOutputTracker
23/09/06 09:45:51 INFO SparkEnv: Registering BlockManagerMaster
23/09/06 09:45:51 INFO BlockManagerMasterEndpoint: Using org.apache.spark.storage.DefaultTopologyMapper for getting topology information
23/09/06 09:45:51 INFO BlockManagerMasterEndpoint: BlockManagerMasterEndpoint up

```

J) After running code you can see one application based on bellow pictures:

 **Spark Master at spark://spark-Virtual-Machine:7077**

URL: spark://spark-Virtual-Machine:7077
 Alive Workers: 1
 Cores in use: 10 Total, 0 Used
 Memory in use: 9.9 GiB Total, 0.0 B Used
 Resources in use:
 Applications: 0 Running, 1 Completed
 Drivers: 0 Running, 0 Completed
 Status: ALIVE

Workers (1)

Worker id	Address	State	Cores	Memory
worker-20230906091226-172.26.83.47-42293	172.26.83.47:42293	ALIVE	10 (0 Used)	9.9 GiB (0.0 B Used)

Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time
----------------	------	-------	---------------------	------------------------	----------------

Completed Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time
app-20230906094552-0000	Simple Spark DataFrame Example	10	1024.0 MiB		2023/09/06 09:45:52

I) Changing amount of CPU and RAM

```

./spark-submit --master spark://spark-Virtual-Machine:7077 --name "example1" --executor-memory 2g --total-executor-cores 4 example1.py

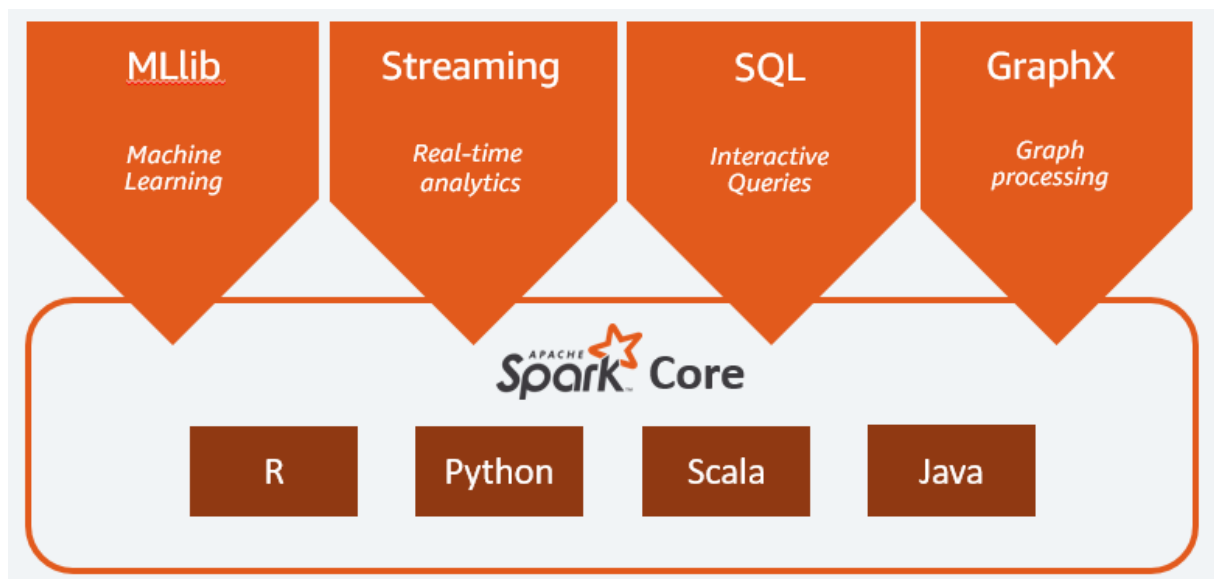
```

2)PySpark, Spark SQL, and DataFrames

2.1- Programming Guides:

- RDD Programming Guide:** overview of Spark basics - RDDs (core but old API), accumulators, and broadcast variables
- Spark SQL, Datasets, and DataFrames:** processing structured data with relational queries (newer API than RDDs)

- **Structured Streaming**: processing structured data streams with relation queries (using Datasets and DataFrames, newer API than DStreams)
- **Spark Streaming**: processing data streams using DStreams (old API)
- **MLlib**: applying machine learning algorithms
- **GraphX**: processing graphs
- **SparkR**: processing data with Spark in R



2.2 - Comparing RDD Programming and Spark SQL

Feature	RDD Programming	Spark SQL
Abstraction Level	Low-level	High-level
Ease of Use	Requires more boilerplate code	Simplified syntax, similar to SQL
Data Schema	Schema-less	Has a schema
Optimization	Manual optimization needed	Automatic optimization using Catalyst
Data Formats	Needs custom parsers for different data formats	Supports various formats like JSON, Parquet, Avro, etc. out-of-the-box
Data Sources	File-based data sources typically	Supports various data sources like Hive, JDBC, etc.

Functionality	Core transformations like map , filter , reduce	Supports SQL queries, joins, and aggregations
Type Safety	Type-safe	Less type-safe compared to RDD
Debugging	Easier to debug due to type safety	Errors often caught at runtime
Performance	Usually slower for analytical queries	Optimized for performance
API Languages	Scala, Java, Python, R	SQL, DataFrame API in Scala, Java, Python, R
Use Case	ETL tasks, stream processing, complex algorithms	Analytical queries, reporting, data exploration
Interoperability	Can be converted to DataFrames but not always optimal	Easily integrated with DataFrames and Datasets

2.3 PySpark.sql

The `pyspark.sql` module is part of PySpark, which is the Python library for Apache Spark. This module provides programming interfaces to work with structured data using Spark. It primarily contains classes and methods for Spark's DataFrame and SQL functionalities.

SparkSession.builder sets up the environment for your Spark application, letting you adjust settings like the app name and where the code should run (locally, on a cluster, etc.). Then, **getOrCreate()** launches that environment.

Local Mode

- **"local"**: Run Spark locally with one worker thread (i.e., no parallelism).
- **"local[N]"**: Run Spark locally with N worker threads (ideally, set N to the number of cores on your machine).
- **"local[*]"**: Run Spark locally with as many worker threads as logical cores on your machine.

Cluster Mode

- **"yarn"**: Connect to a YARN cluster in client or cluster mode depending on the value of **--deploy-mode**. The cluster location will be found based on the `HADOOP_CONF_DIR` or `YARN_CONF_DIR` variable.

- **"mesos://HOST:PORT"**: Connect to a Mesos cluster at the given host and port.
- **"spark://HOST:PORT"**: Connect to a Spark standalone cluster at the given host and port.

2.4 Testing spark

```
# Import SparkSession
from pyspark.sql import SparkSession
# Create a Spark Session
spark = SparkSession.builder.master("local[*]").getOrCreate()
# Check Spark Session Information
spark
```

SparkSession - in-memory

SparkContext

Spark UI

Version

v3.4.1

Master

local[*]

AppName

pyspark-shell

2.5 Types

The **pyspark.sql.types** module contains classes that define the schema of a DataFrame. For example, **StringType**, **IntegerType**, **ArrayType**, **MapType**, etc.

1. **ByteType**: Represents a byte column.
2. **ShortType**: Represents a short column.
3. **IntegerType**: Represents a 4-byte signed integer column.
4. **LongType**: Represents an 8-byte signed integer column.
5. **FloatType**: Represents a single precision floating-point number column.
6. **DoubleType**: Represents a double precision floating-point number column.
7. **StringType**: Represents a string column.
8. **BooleanType**: Represents a boolean column.

pyspark.sql.types

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, IntegerType,
StringType
```

```

spark =
SparkSession.builder.master("local[1]").appName('SparkApp').getOrCreate()

# Define schema
schema = StructType([
    StructField('id', IntegerType(), True),
    StructField('name', StringType(), True)
])

# Create DataFrame
df = spark.createDataFrame([(1, 'John'), (2, 'Mike'), (3, 'Sara')],
schema=schema)

df.show()

```

Name: It's a string that defines the column name.

DataType: Spark provides various pre-defined data types like IntegerType, StringType, FloatType, etc. You can also define complex types like ArrayType, MapType, and StructType.

Nullable: This is a boolean flag to indicate if the column can have null values. If set to True, then the column can have null values. If set to False, then the column cannot have null values.

2.6- different ways to create a Spark DataFrame

Source	Method	Schema Required ?	Example Code
List of Tuples or Lists	<code>spark.createDataFrame()</code>	Optional	<code>spark.createDataFrame([(1, 'John'), (2, 'Mike')])</code>
RDD	<code>spark.createDataFrame()</code>	Optional	<code>spark.createDataFrame(rdd, schema=schema)</code>
Pandas DataFrame	<code>spark.createDataFrame()</code>	No (Inferred)	<code>spark.createDataFrame(pandas_df)</code>
NumPy Array	Convert to Pandas DataFrame, then use <code>spark.createDataFrame()</code>	No (Inferred)	<code>spark.createDataFrame(pd.DataFrame(numpy_array))</code>
JSON File	<code>spark.read.json()</code>	No (Inferred)	<code>spark.read.json("path/to/json")</code>
Parquet File	<code>spark.read.parquet()</code>	No (Inferred)	<code>spark.read.parquet("path/to/parquet")</code>
Database via JDBC	<code>spark.read.format("jdbc").option(...).load()</code>	No (Inferred)	<code>spark.read.format("jdbc").option("url", "jdbc:...").load()</code>

Hive Table	<code>spark.sql()</code>	No (Inferred)	<code>spark.sql("SELECT * FROM hive_table")</code>
------------	--------------------------	---------------	--

1. Create DataFrame from a List of Tuples or Lists

```
from pyspark.sql import SparkSession

spark =
SparkSession.builder.master("local[*]").appName("FromList").getOrCreate()

df = spark.createDataFrame([(1, 'John'), (2, 'Mike'), (3, 'Sara')])
df.show()

spark.stop()
```

```
+---+-----+
|_1|_2|
+---+-----+
| 1|John|
| 2|Mike|
| 3|Sara|
+---+-----+
```

2. Create DataFrame from an RDD

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

spark =
SparkSession.builder.master("local[*]").appName("FromRDD").getOrCreate()

schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True)
])

rdd = spark.sparkContext.parallelize([(1, 'John'), (2, 'Mike'), (3, 'Sara')])
df = spark.createDataFrame(rdd, schema=schema)
df.show()

spark.stop()
```


spark.sparkContext.parallelize is a method in Spark that is used to create a Resilient Distributed Dataset (RDD) from a local collection (like a list or an array). RDDs are the fundamental data structures in Spark, and they represent an immutable, distributed collection of objects that can be processed in parallel.

3. Create DataFrame from a Pandas DataFrame

```
from pyspark.sql import SparkSession
import pandas as pd

spark =
SparkSession.builder.master("local[*]").appName("FromPandas").getOrCreate()

pandas_df = pd.DataFrame({
    'id': [1, 2, 3],
    'name': ['John', 'Mike', 'Sara']
})
df = spark.createDataFrame(pandas_df)
df.show()

spark.stop()
```

4. Create DataFrame from a NumPy Array

```
from pyspark.sql import SparkSession
import numpy as np
import pandas as pd

spark =
SparkSession.builder.master("local[*]").appName("FromNumpy").getOrCreate()

numpy_array = np.array([(1, 'John'), (2, 'Mike'), (3, 'Sara')])
pandas_df = pd.DataFrame(numpy_array, columns=['id', 'name'])
df = spark.createDataFrame(pandas_df)
df.show()

spark.stop()
```

5. Create DataFrame from a JSON File

```
import json
```

```

from pyspark.sql import SparkSession

# Step 1: Create a JSON File with Sample Data

# Define the sample data
data = [
    {"id": 1, "name": "John"},
    {"id": 2, "name": "Mike"},
    {"id": 3, "name": "Sara"}
]

# Write the data to a JSON file
with open("data.json", "w") as f:
    json.dump(data, f)

# Step 2: Read the JSON File into a DataFrame using PySpark

# Create a Spark Session
spark =
SparkSession.builder.master("local[*]").appName("FromJSON").getOrCreate()

# Read the JSON file into a DataFrame
df = spark.read.json("data.json")

# Show the DataFrame
df.show()

# Stop the Spark session
spark.stop()

```

2.7 Spark DataFrames operations

Spark DataFrames have a rich set of operations that can be categorized into various types. Below is a comprehensive table summarizing the types of operations you can perform on Spark DataFrames

Operation	Method Used	Description	Example
Projection	select	Select specific columns from a DataFrame	df.select("column1", "column2").show()
Filtering	filter or where	Filter rows based on a condition	df.filter(df.column1 > 10).show()
Add Column	withColumn	Add a new column to the DataFrame	df.withColumn("new_column", F.col("column1") * 2)
Drop Column	drop	Remove a column from the DataFrame	df.drop("column_to_drop").show()
Sorting	sort or orderBy	Sort DataFrame based on one or more columns	df.sort("column1").show()
Aggregation	groupBy and agg	Perform aggregate operations like sum, min, max, etc.	df.groupBy("column_to_group").agg(F.sum("column_to_sum"))

Joins	join	Join two DataFrames based on a common column	df1.join(df2, "common_column").show()
Limit Rows	limit	Limit the number of rows returned	df.limit(5).show()
Rename Column	withColumnRenamed	Rename a column	df.withColumnRenamed("old_name", "new_name").show()
SQL Queries	createOrReplaceTempView and spark.sql	Run SQL queries on a DataFrame	spark.sql("SELECT * FROM table_name WHERE some_condition")
Distinct Rows	distinct	Return distinct rows	df.distinct().show()
Remove Duplicates	dropDuplicates	Remove duplicate rows	df.dropDuplicates().show()

You can see a simple example based on above operation:

operations
<pre> from pyspark.sql import SparkSession import pyspark.sql.functions as F # Initialize Spark session spark = SparkSession.builder \ .appName("CommonDataFrameOperations") \ .getOrCreate() # Create an example DataFrame data = [("Alice", 1, 34), ("Bob", 2, 45), ("Catherine", 3, 29), ("Bob", 2, 45)] columns = ["Name", "ID", "Age"] df = spark.createDataFrame(data, columns) df.show() # 1. Projection: Select only the "Name" and "Age" columns print("1. Projection:") df.select("Name", "Age").show() # 2. Filtering: Select rows where Age > 30 print("2. Filtering:") df.filter(df.Age > 30).show() # 3. Adding a new column: Age + 1 print("3. Adding a New Column:") df.withColumn("AgePlusOne", F.col("Age") + 1).show() # 4. Dropping a column: Remove "Age" print("4. Dropping a Column:") df.drop("Age").show() # 5. Sorting: Sort by Age print("5. Sorting:") df.sort("Age").show() </pre>

```

# 6. Aggregation: Group by "Name" and count
print("6. Aggregation:")
df.groupBy("Name").agg(F.count("ID").alias("Count"),
F.avg("Age").alias("AverageAge")).show()

# 7. Joins: Join with another DataFrame
print("7. Joins:")
data2 = [("Alice", "F"), ("Bob", "M"), ("Catherine", "F")]
columns2 = ["Name", "Gender"]
df2 = spark.createDataFrame(data2, columns2)
df.join(df2, "Name").show()

# 8. Limiting rows: Show only 2 rows
print("8. Limiting Rows:")
df.limit(2).show()

# 9. Renaming columns: Rename column "ID" to "Identifier"
print("9. Renaming Columns:")
df.withColumnRenamed("ID", "Identifier").show()

# 10. Running SQL Queries
print("10. Running SQL Queries:")
df.createOrReplaceTempView("people")
spark.sql("SELECT * FROM people WHERE Age > 30").show()

# 11. Distinct rows: Remove duplicate rows
print("11. Distinct Rows:")
df.distinct().show()

# Stop the Spark session
spark.stop()

```

example

```

from pyspark.sql import SparkSession

# Initialize Spark Session
spark = SparkSession.builder \
    .appName('Transformation Operations') \
    .getOrCreate()

# Create a DataFrame
df = spark.createDataFrame([(1, 'Apple', 3.0), (2, 'Banana', 1.0), (3,
'Orange', 2.5), (4, 'Mango', 3.5)], ['id', 'fruit', 'price'])

```

```

# 1. Select Operation
print("1. Select Operation:")
df.select('id', 'fruit').show()

# 2. Filter Operation
print("2. Filter Operation:")
df.filter(df['price'] > 2).show()

# 3. GroupBy Operation
print("3. GroupBy Operation:")
df.groupBy('fruit').count().show()

# 4. OrderBy Operation
print("4. OrderBy Operation:")
df.orderBy('price', ascending=False).show()

# 5. WithColumn Operation
print("5. WithColumn Operation:")
df.withColumn('discounted_price', df['price'] * 0.9).show()

# 6. Drop Operation
print("6. Drop Operation:")
df.drop('price').show()

# Stop the Spark Session
spark.stop()

```

Comparing group by in pandas and pyspark

```

import pandas as pd

# Create a DataFrame
df = pd.DataFrame({
    'Name': ['Alice', 'Alice', 'Bob', 'Bob'],
    'Subject': ['Math', 'Science', 'Math', 'Science'],
    'Score': [90, 85, 70, 80]
})

# Group by the 'Name' column and calculate the mean
grouped_df = df.groupby('Name').mean()

print(grouped_df)

```

```

from pyspark.sql import SparkSession

```

```

import pyspark.sql.functions as F

# Initialize a Spark Session
spark = SparkSession.builder.appName("GroupByExample").getOrCreate()

# Create a DataFrame
data = [("Alice", "Math", 90),
        ("Alice", "Science", 85),
        ("Bob", "Math", 70),
        ("Bob", "Science", 80)]
columns = ["Name", "Subject", "Score"]

df = spark.createDataFrame(data, columns)

# Group by 'Name' and calculate the average
grouped_df = df.groupBy("Name").agg(F.mean("Score"))

grouped_df.show()

```

Group by and agg

```

from pyspark.sql import SparkSession

# Initialize Spark Session
spark = SparkSession.builder \
    .appName('GroupBy and Agg Example') \
    .getOrCreate()

# Create a DataFrame
df = spark.createDataFrame([
    ('Alice', 'F', 34, 5000),
    ('Bob', 'M', 45, 4800),
    ('Catherine', 'F', 29, 6000),
    ('David', 'M', 31, 5500),
    ('Emily', 'F', 42, 7000),
], ['name', 'gender', 'age', 'salary'])

# Using groupBy to group by 'gender' and then applying the count()
aggregate function
print("1. Using groupBy to count rows for each gender:")
df.groupBy("gender").count().show()

# Using groupBy to group by 'gender' and then applying the avg()
aggregate function
print("2. Using groupBy to calculate average salary by gender:")

```

```

df.groupBy("gender").avg("salary").show()

# Using agg to find the maximum salary and minimum age in the
DataFrame
print("3. Using agg to find maximum salary and minimum age:")
df.agg({'salary': 'max', 'age': 'min'}).show()

# Using groupBy and agg together for more complex operations
print("4. Using groupBy and agg together to find maximum and minimum
salary by gender:")
df.groupBy("gender").agg({'salary': 'max', 'salary': 'min'}).show()

# Stop the Spark Session
spark.stop()

```

2.8- Running SQL Queries

SQL Instruction Type	Description	Example SQL Syntax
SELECT	Retrieves data from a table	SELECT column1, column2 FROM table
WHERE	Filters rows based on conditions	SELECT * FROM table WHERE condition
GROUP BY	Groups rows based on a column	SELECT column, AGG_FUNCTION(column2) FROM table GROUP BY column
ORDER BY	Sorts rows by a column	SELECT * FROM table ORDER BY column [ASC
JOIN	Combines rows from two or more tables	SELECT * FROM table1 JOIN table2 ON condition
UNION	Combines results of two queries	SELECT column FROM table1 UNION SELECT column FROM table2
INSERT INTO	Inserts new rows into a table	INSERT INTO table (column1, column2) VALUES (value1, value2)
UPDATE	Updates existing rows in a table	UPDATE table SET column=value WHERE condition
DELETE	Removes existing rows from a table	DELETE FROM table WHERE condition
DISTINCT	Returns unique values in a column	SELECT DISTINCT column FROM table
COUNT	Counts number of rows	SELECT COUNT(column) FROM table
SUM	Adds up the values of a column	SELECT SUM(column) FROM table

AVG	Finds the average value of a column	SELECT AVG(column) FROM table
MIN/MAX	Finds the minimum/maximum value of a column	SELECT MIN(column)/MAX(column) FROM table

Running SQL Queries

```

from pyspark.sql import SparkSession

# Step 1: Initialize Spark Session
spark = SparkSession.builder \
    .appName('PySpark SQL Example') \
    .getOrCreate()

# Step 2: Create DataFrame
data = [("Alice", 34, "Female"), ("Bob", 45, "Male"), ("Catherine",
29, "Female"), ("David", 31, "Male"), ("Eva", 25, "Female")]
columns = ["Name", "Age", "Gender"]
df = spark.createDataFrame(data, columns)

# Step 3: Register DataFrame as Temporary SQL Table
df.createOrReplaceTempView("people")

# Step 4: Execute SQL Queries
result1 = spark.sql("SELECT * FROM people")
print("All rows in the table:")
result1.show()

result2 = spark.sql("SELECT AVG(Age) as average_age FROM people")
print("Average age:")
result2.show()

result3 = spark.sql("SELECT * FROM people WHERE Gender='Female'")
print("All females:")
result3.show()

# Additional SQL operations:

# Count of males and females
result4 = spark.sql("SELECT Gender, COUNT(*) as Count FROM people
GROUP BY Gender")
print("Count of males and females:")
result4.show()

# People older than 30
result5 = spark.sql("SELECT * FROM people WHERE Age > 30")

```



```

print("People older than 30:")
result5.show()

# Distinct genders in the table
result6 = spark.sql("SELECT DISTINCT Gender FROM people")
print("Distinct genders:")
result6.show()

# Sorting people by age in descending order
result7 = spark.sql("SELECT * FROM people ORDER BY Age DESC")
print("People sorted by age in descending order:")
result7.show()

# Close the Spark session
spark.stop()

```

2.9 Applying a Function in spark

1. **Built-in Function:** Utilizes native Spark functions for common data transformations, offering optimized performance. Example: `length("Name")` calculates string lengths.
2. **Python Native Function as UDF:** Converts standard Python functions to User-Defined Functions (UDFs) for use in Spark. May be less efficient but offers custom logic. Example: `plus_one_udf("Number")` adds 1 to each number.
3. **Pandas UDF:** Uses Pandas Series for operations within UDFs, offering a performance-optimized way to use Pythonic operations in Spark. Example: `pandas_plus_one("Number")` adds 1 to each number efficiently using Pandas.

```

from pyspark.sql import SparkSession
from pyspark.sql.functions import length, udf, pandas_udf
from pyspark.sql.types import IntegerType, LongType
import pandas as pd

# Initialize Spark Session
spark = SparkSession.builder \
    .appName("Function Application Examples") \
    .getOrCreate()

# ----- Using Built-in Function -----

# Create DataFrame
data1 = [("Alice",), ("Bob",), ("Catherine",)]
df1 = spark.createDataFrame(data1, ["Name"])

```

```

# Apply built-in function
print("Using Built-in Function:")
df1.withColumn("Length", length("Name")).show()

# ----- Using Python Native Function as UDF -----

# Define Python native function
def plus_one(x):
    return x + 1

# Create UDF from Python function
plus_one_udf = udf(plus_one, IntegerType())

# Create DataFrame
data2 = [(1,), (2,), (3,)]
df2 = spark.createDataFrame(data2, ["Number"])

# Apply UDF
print("Using Python Native Function as UDF:")
df2.withColumn("PlusOne", plus_one_udf("Number")).show()

# ----- Using Pandas UDF -----

# Define Pandas UDF
@pandas_udf(LongType())
def pandas_plus_one(s: pd.Series) -> pd.Series:
    return s + 1

# Apply Pandas UDF
print("Using Pandas UDF:")
df2.withColumn("PandasPlusOne", pandas_plus_one("Number")).show()

# Stop Spark Session
spark.stop()

```

@pandas_udf(LongType())

This line introduces a decorator **@pandas_udf** which is used to define a Pandas UDF (User Defined Function) in PySpark. A Pandas UDF operates on **pandas.Series** and **pandas.DataFrame** objects, allowing for vectorized operations that can be more efficient than row-by-row operations of traditional UDFs.

- **LongType():** This specifies the return type of the UDF. In this case, the UDF will return a column of type **LongType** (which corresponds to Python's **int** type).

Great! Now you are a master in Spark 😊 Now, try to work on sample tasks!