

- ❖ Setup the JDBC configuration and run the Spark JDBC Connectivity program

```

192.168.56.100 - PuTTY
hadoop@mainserver1:~$ mysql -u airflow -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.0.35-0ubuntu0.20.04.1 (Ubuntu)

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create database harshadb
-> ;
Query OK, 1 row affected (0.03 sec)

mysql> use harshadb;
Database changed
mysql> create table animals ( animalid int not null primary key,
-> aniname varchar(20) not null,
-> anitype varchar(20) not null )
-> ;
Query OK, 0 rows affected (0.24 sec)

mysql> select * from animals;
Empty set (0.03 sec)

mysql> insert into animals ( 1001, 'lion', 'carnivorous');
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL
server version for the right syntax to use near '1001, 'lion', 'carnivorous'' at line 1
mysql> insert into animals values ( 1001, 'lion', 'carnivorous');
Query OK, 1 row affected (0.03 sec)

mysql> insert into animals values ( 1002, 'tiger', 'carnivorous');
Query OK, 1 row affected (0.02 sec)

mysql> insert into animals values ( 1003, 'cow', 'herbivorous');
Query OK, 1 row affected (0.01 sec)

mysql> insert into animals values ( 1004, 'bufflow', 'herbivorous');
Query OK, 1 row affected (0.01 sec)

mysql> select * from animals;
+-----+-----+-----+
| animalid | aniname | anitype |
+-----+-----+-----+
| 1001 | lion | carnivorous |
| 1002 | tiger | carnivorous |
| 1003 | cow | herbivorous |
| 1004 | bufflow | herbivorous |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql>

```

```
hadoop@mainserver1:~$ sudo dpkg -i ./mysql-connector-j_8.1.0-1ubuntu22.04_all.deb
[sudo] password for hadoop:
Selecting previously unselected package mysql-connector-j.
(Reading database ... 82343 files and directories currently installed.)
Preparing to unpack .../mysql-connector-j_8.1.0-1ubuntu22.04_all.deb ...
Unpacking mysql-connector-j (8.1.0-1ubuntu22.04) ...
Setting up mysql-connector-j (8.1.0-1ubuntu22.04) ...
hadoop@mainserver1:~$ find /usr -name mysql-connector*.jar
/usr/share/java/mysql-connector-java-8.1.0.jar
/usr/share/java/mysql-connector-j-8.1.0.jar
hadoop@mainserver1:~$
```

\$ vim myread_jdbc.py

```
# Sample spark program to fetch the data from the mysql database
from pyspark.sql import SparkSession

# Set up the Spark session
spark = SparkSession.builder.appName("JDBCExample").getOrCreate()

# JDBC connection properties for MySQL
jdbc_url = "jdbc:mysql://localhost:3306/harshadb"
jdbc_properties = {
    "user": "airflow",
    "password": "airflow",
    "driver": "com.mysql.cj.jdbc.Driver"
}

# read data from the MySQL database
df = spark.read \
    .format("jdbc") \
    .option("url", jdbc_url) \
    .option("dbtable", "animals") \
    .option("user", jdbc_properties["user"]) \
    .option("password", jdbc_properties["password"]) \
    .option("driver", jdbc_properties["driver"]) \
    .load()

# show the retrieved data
df.show()

# stop the spark session
spark.stop()

<save and exit>
```

\$ spark-submit --jars /usr/share/java/mysql-connector-java-8.1.0.jar myread_jdbc.py

```

192.168.56.100 - PuTTY
for this job
24/01/10 05:06:06 INFO TaskSchedulerImpl: Killing all running tasks in stage 0: Stage finished
24/01/10 05:06:06 INFO DAGScheduler: Job 0 finished: showString at NativeMethodAccessorImpl.java:0, took
1.809106 s
24/01/10 05:06:06 INFO CodeGenerator: Code generated in 107.986186 ms
+-----+-----+-----+
|animalid|aniname|  anitype|
+-----+-----+-----+
|   1001|   lion|carnivorous|
|   1002|  tiger|carnivorous|
|   1003|   cow|herbivorous|
|   1004|bufflow|herbivorous|
+-----+-----+-----+

24/01/10 05:06:07 INFO SparkUI: Stopped Spark web UI at http://10.0.2.15:4040
24/01/10 05:06:07 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
24/01/10 05:06:07 INFO MemoryStore: MemoryStore cleared
24/01/10 05:06:07 INFO BlockManager: BlockManager stopped
24/01/10 05:06:07 INFO BlockManagerMaster: BlockManagerMaster stopped
24/01/10 05:06:07 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator s
topped!
24/01/10 05:06:07 INFO SparkContext: Successfully stopped SparkContext
24/01/10 05:06:07 INFO ShutdownHookManager: Shutdown hook called
24/01/10 05:06:07 INFO ShutdownHookManager: Deleting directory /tmp/spark-0ee3aeac-c1c3-4b3b-ba30-bfe693e
96e4e
24/01/10 05:06:07 INFO ShutdownHookManager: Deleting directory /tmp/spark-f20430aa-c948-4da7-a81e-50777ec
a43a5
24/01/10 05:06:07 INFO ShutdownHookManager: Deleting directory /tmp/spark-f20430aa-c948-4da7-a81e-50777ec
a43a5/pyspark-f37d9dea-671d-4e30-8375-baf31f3bc516

```

\$ vim mywrite_jdbc.py

Sampele spark program to fetch the data from the mysql database
from pyspark.sql import SparkSession

Set up the Spark session
spark = SparkSession.builder.appName("JDBCExample").getOrCreate()

JDBC connection properties for MySQL
jdbc_url = "jdbc:mysql://localhost:3306/harshadb"
jdbc_properties = {
 "user": "airflow",
 "password": "airflow",
 "driver": "com.mysql.cj.jdbc.Driver"
}

sample data to be inserted in the MySQL table
data = [(1005,'donkey','herbivorous'),(1006,'cheeta','carnivorous'),(1007,'elephant','herbivorous')]

Define the schema for the data frame
schema = ["animalid" , "aniname" , "anitype"]

Create a DataFrame fromthe sample data
df = spark.createDataFrame(data, schema=schema)

write data from the MySQL database
df.write \
 .format("jdbc") \
 .option("url", jdbc_url) \
 .option("dbtable","animals") \
 .option("user",jdbc_properties["user"]) \
 .option("password",jdbc_properties["password"]) \
 .option("driver",jdbc_properties["driver"]) \
 .mode("append") \
 .save()

read data from the MySQL database
df = spark.read \
 .format("jdbc") \
 .option("url", jdbc_url) \
 .option("dbtable","animals") \
 .option("user",jdbc_properties["user"]) \
 .option("password",jdbc_properties["password"]) \
 .option("driver",jdbc_properties["driver"]) \
 .load()

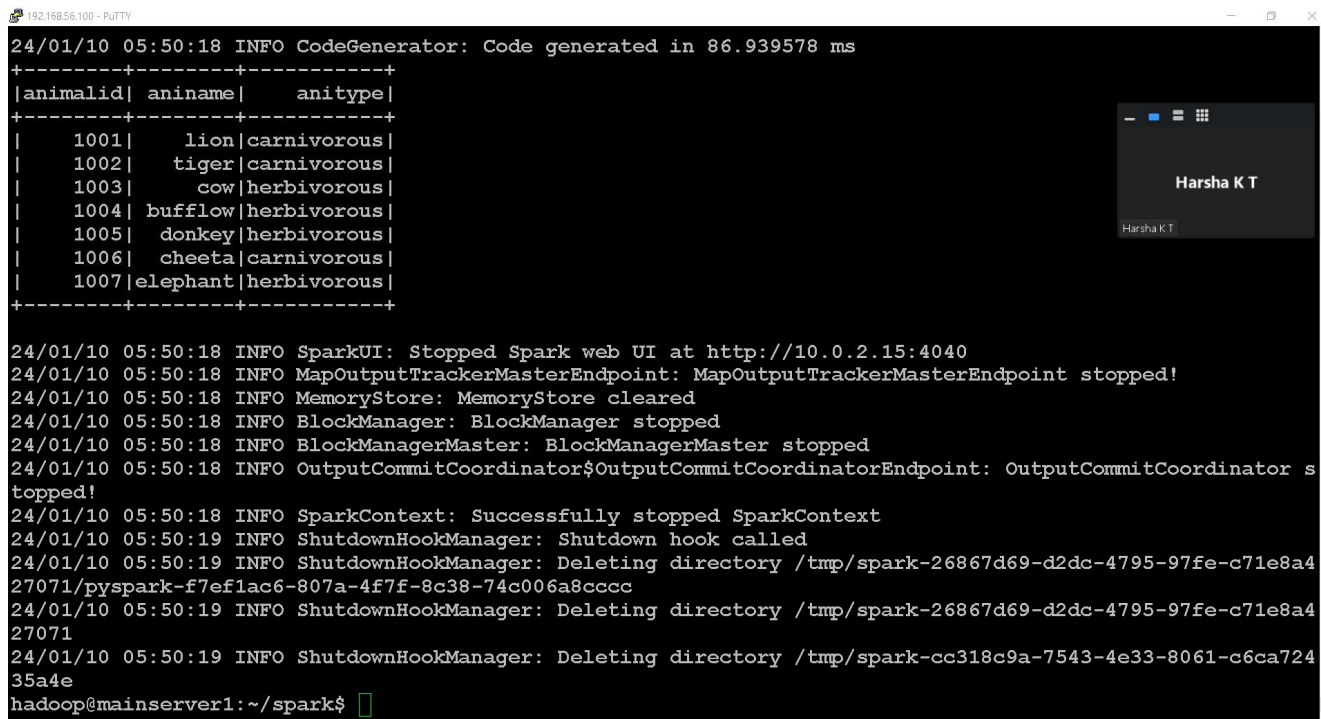
```
.load()

# show the retried data
df.show()

# stop the spark session
spark.stop()

<save and exit>
```

```
$ spark-submit --jars /usr/share/java/mysql-connector-java-8.1.0.jar mywrite_jdbc.py
```



```
192.168.56.100 - PuTTY
24/01/10 05:50:18 INFO CodeGenerator: Code generated in 86.939578 ms
+-----+-----+-----+
|animalid|  aniname|  anitype|
+-----+-----+-----+
|    1001|    lion|carnivorous|
|    1002|   tiger|carnivorous|
|    1003|    cow|herbivorous|
|    1004|bufflow|herbivorous|
|    1005| donkey|herbivorous|
|    1006| cheeta|carnivorous|
|    1007|elephant|herbivorous|
+-----+-----+-----+

24/01/10 05:50:18 INFO SparkUI: Stopped Spark web UI at http://10.0.2.15:4040
24/01/10 05:50:18 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
24/01/10 05:50:18 INFO MemoryStore: MemoryStore cleared
24/01/10 05:50:18 INFO BlockManager: BlockManager stopped
24/01/10 05:50:18 INFO BlockManagerMaster: BlockManagerMaster stopped
24/01/10 05:50:18 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator s
topped!
24/01/10 05:50:18 INFO SparkContext: Successfully stopped SparkContext
24/01/10 05:50:19 INFO ShutdownHookManager: Shutdown hook called
24/01/10 05:50:19 INFO ShutdownHookManager: Deleting directory /tmp/spark-26867d69-d2dc-4795-97fe-c71e8a4
27071/pyspark-f7ef1ac6-807a-4f7f-8c38-74c006a8cccc
24/01/10 05:50:19 INFO ShutdownHookManager: Deleting directory /tmp/spark-26867d69-d2dc-4795-97fe-c71e8a4
27071
24/01/10 05:50:19 INFO ShutdownHookManager: Deleting directory /tmp/spark-cc318c9a-7543-4e33-8061-c6ca724
35a4e
hadoop@mainserver1:~/spark$
```

Apache Spark and MapReduce

Apache Spark is an open-source, distributed computing system that provides a fast and general-purpose cluster computing framework for big data processing.

Spark includes support for MapReduce-style programming, making it compatible with the Hadoop Distributed File System (HDFS) and allowing users to leverage existing Hadoop MapReduce code.

Basic example of using Spark to perform a simple MapReduce operation in Python using PySpark

This example counts the occurrences of each word in a text file:

```
$ vim SimpleMapReduce.py
```

```
from pyspark import SparkContext, SparkConf

# Create a Spark configuration
conf = SparkConf().setAppName("SimpleMapReduce")
sc = SparkContext(conf=conf)

# Read input data from a text file
input_data = sc.textFile("hdfs://localhost:9000/cdadir/mylocal.txt")

# Map step: Split each line into words
mapped_data = input_data.flatMap(lambda line: line.split(" "))

# Reduce step: Count occurrences of each word
word_counts = mapped_data.map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)

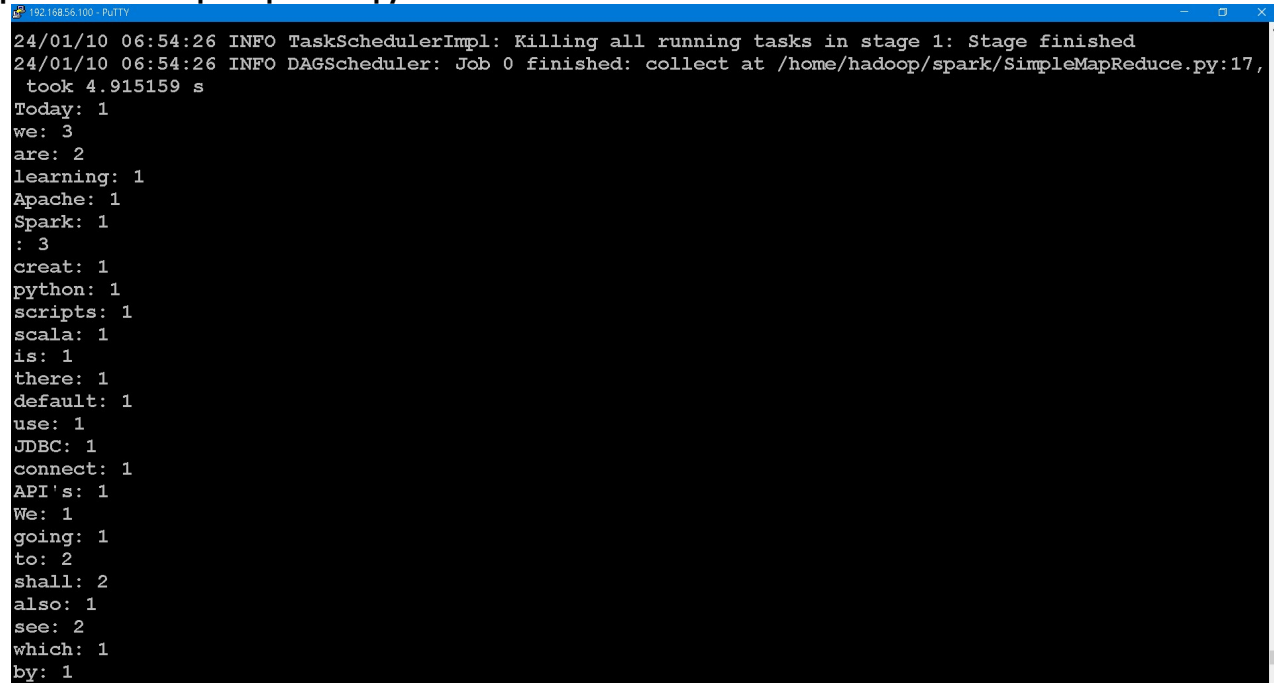
# Show the result
result = word_counts.collect()
```

```
for (word, count) in result:
    print(f"{word}: {count}")
```

```
# Stop the SparkContext
sc.stop()
```

<save and exit>

\$ spark-submit SimpleMapReduce.py



```
24/01/10 06:54:26 INFO TaskSchedulerImpl: Killing all running tasks in stage 1: Stage finished
24/01/10 06:54:26 INFO DAGScheduler: Job 0 finished: collect at /home/hadoop/spark/SimpleMapReduce.py:17,
    took 4.915159 s
Today: 1
we: 3
are: 2
learning: 1
Apache: 1
Spark: 1
: 3
creat: 1
python: 1
scripts: 1
scala: 1
is: 1
there: 1
default: 1
use: 1
JDBC: 1
connect: 1
API's: 1
We: 1
going: 1
to: 2
shall: 2
also: 1
see: 2
which: 1
by: 1
```

RDD - Resilient Distributed Dataset - Persistence Example - SampleScriptRDD.py

In Apache Spark we use RDD persistence to cache or persist on RDD in memory or on disk to avoid precomputations.

\$ vim SampleScriptRDD.py

```
from pyspark import SparkContext

# Create a SparkContext
sc = SparkContext("local", "RDDPersistenceExample")

# Create an RDD from a list of numbers
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)

# Cache the RDD in memory
rdd.persist()

# Action 1: Perform a transformation and an action on the RDD
squared_rdd = rdd.map(lambda x: x * x)
print("Squared RDD:", squared_rdd.collect())

# Action 2: Perform another transformation and an action on the persisted RDD
cubed_rdd = rdd.map(lambda x: x * x * x)
print("Cubed RDD:", cubed_rdd.collect())

# Unpersist the RDD (remove it from the cache)
rdd.unpersist()

# Action 3: Attempting to use the unpersisted RDD will trigger recomputation
doubled_rdd = rdd.map(lambda x: x * 2)
print("Doubled RDD (recomputed):", doubled_rdd.collect())
```

```
# Stop the SparkContext
sc.stop()
```

\$ spark-submit SampleScriptRDD.py

You should find the following information in your output when you run the above command
Sample data

```
24/01/10 07:21:15 INFO TaskSchedulerImpl: Killing all running tasks in stage 0: Stage finished
24/01/10 07:21:15 INFO DAGScheduler: Job 0 finished: collect at /home/hadoop/spark/SampleScriptRDD.py:15,
took 3.621666 s
Squared RDD: [1, 4, 9, 16, 25]
24/01/10 07:21:15 INFO SparkContext: Starting job: collect at /home/hadoop/spark/SampleScriptRDD.py:19
24/01/10 07:21:15 INFO DAGScheduler: Got job 1 (collect at /home/hadoop/spark/SampleScriptRDD.py:19) with
1 output partitions
24/01/10 07:21:15 INFO DAGScheduler: Final stage: ResultStage 1 (collect at /home/hadoop/spark/SampleScriptRDD.py:19)
```

```
24/01/10 07:21:16 INFO TaskSchedulerImpl: Killing all running tasks in stage 1: Stage finished
24/01/10 07:21:16 INFO DAGScheduler: Job 1 finished: collect at /home/hadoop/spark/SampleScriptRDD.py:19,
took 0.226953 s
Cubed RDD: [1, 8, 27, 64, 125]
24/01/10 07:21:16 INFO ParallelCollectionRDD: Removing RDD 0 from persistence list
24/01/10 07:21:16 INFO BlockManager: Removing RDD 0
24/01/10 07:21:16 INFO BlockManagerInfo: Removed broadcast_1_piece0 on 10.0.2.15:35739 in memory (size: 3
.5 KiB, free: 434.4 MiB)
24/01/10 07:21:16 INFO SparkContext: Starting job: collect at /home/hadoop/spark/SampleScriptRDD.py:26
24/01/10 07:21:16 INFO DAGScheduler: Got job 2 (collect at /home/hadoop/spark/SampleScriptRDD.py:26) with
1 output partitions
24/01/10 07:21:16 INFO DAGScheduler: Final stage: ResultStage 2 (collect at /home/hadoop/spark/SampleScriptRDD.py:26)
```

```
24/01/10 07:21:16 INFO TaskSchedulerImpl: Killing all running tasks in stage 2: Stage finished
24/01/10 07:21:16 INFO DAGScheduler: Job 2 finished: collect at /home/hadoop/spark/SampleScriptRDD.py:26,
took 0.286910 s
Doubled RDD (recomputed): [2, 4, 6, 8, 10]
24/01/10 07:21:16 INFO SparkUI: Stopped Spark web UI at http://10.0.2.15:4040
24/01/10 07:21:16 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
24/01/10 07:21:16 INFO MemoryStore: MemoryStore cleared
24/01/10 07:21:16 INFO BlockManager: BlockManager stopped
24/01/10 07:21:16 INFO BlockManagerMaster: BlockManagerMaster stopped
24/01/10 07:21:16 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator s
topped!
24/01/10 07:21:16 INFO SparkContext: Successfully stopped SparkContext
24/01/10 07:21:17 INFO ShutdownHookManager: Shutdown hook called
24/01/10 07:21:17 INFO ShutdownHookManager: Deleting directory /tmp/spark-86cab064-5110-4ecc-8395-a6703fb
ce489
24/01/10 07:21:17 INFO ShutdownHookManager: Deleting directory /tmp/spark-0bf64685-a553-4e04-b416-7bb1a97
b5f31
24/01/10 07:21:17 INFO ShutdownHookManager: Deleting directory /tmp/spark-86cab064-5110-4ecc-8395-a6703fb
ce489/pyspark-d37ed12e-b288-4735-bf58-49ada4878564
hadoop@mainserver1:~/spark$
```

Passing Functions to Spark and working with key-value pairs

In Apache Spark we can pass functions to spark in the form of closures. Closures are functions (or objects) that can be sent to Spark workers to be executed. Additionally working with Key-Value pairs is a common pattern in spark - when dealing with operations like groupByKey, reduceByKey and joins.

1. Passing functions to spark (closure):

- When we pass a function to spark, it is serialized and sent to the worker nodes to be executed in parallel. However there are considerations for what can and cannot be captured in a closure.

Example of passing a simple functions

```
$ vim simple_function.py
```

```
from pyspark import SparkContext

# Create a SparkContext
sc = SparkContext("local", "FunctionPassingExample")
```



```
# Define a function
def square(x):
    return x * x

# Parallelize a list and apply the function using map
data = [1, 2, 3, 4, 5]
result = sc.parallelize(data).map(square).collect()

# Display the result
print("Squared values:", result)

# Stop the SparkContext
sc.stop()

<save and exit>
```

\$ spark-submit ./simple_function.py

Sample output:

```
24/01/10 08:47:16 INFO TaskSchedulerImpl: Killing all running tasks in stage 0: Stage finished
24/01/10 08:47:16 INFO DAGScheduler: Job 0 finished: collect at /home/hadoop/spark/simple_functions.py:12
, took 4.534724 s
Squared values: [1, 4, 9, 16, 25]
24/01/10 08:47:16 INFO SparkUI: Stopped Spark web UI at http://10.0.2.15:4040
24/01/10 08:47:16 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
24/01/10 08:47:16 INFO MemoryStore: MemoryStore cleared
24/01/10 08:47:16 INFO BlockManager: BlockManager stopped
24/01/10 08:47:16 INFO BlockManagerMaster: BlockManagerMaster stopped
24/01/10 08:47:16 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator s
topped!
24/01/10 08:47:16 INFO SparkContext: Successfully stopped SparkContext
24/01/10 08:47:17 INFO ShutdownHookManager: Shutdown hook called
24/01/10 08:47:17 INFO ShutdownHookManager: Deleting directory /tmp/spark-658423b7-0b03-4dd4-8bd5-d63df62
67410
24/01/10 08:47:17 INFO ShutdownHookManager: Deleting directory /tmp/spark-2f4b5339-01b9-478d-944d-054d301
9f6fe/pyspark-4ad906cb-51e2-4a2b-85f9-be68c0601570
24/01/10 08:47:17 INFO ShutdownHookManager: Deleting directory /tmp/spark-2f4b5339-01b9-478d-944d-054d301
9f6fe
hadoop@mainserver1:~/spark$
```

Example: Capturing variables in a closure:

\$ vim capture_variable.py

```
from pyspark import SparkContext

# Create a SparkContext
sc = SparkContext("local", "ClosureExample")

# Define a variable outside the function
base_value = 3

# Define a function that captures the external variable
def add_base(x):
    return x + base_value

# Parallelize a list and apply the function using map
data = [1, 2, 3, 4, 5]
result = sc.parallelize(data).map(add_base).collect()

# Display the result
print("Values with base added:", result)

# Stop the SparkContext
sc.stop()
```

<save and exit>

\$ spark-submit capture_variable.py

Sample output:

```

24/01/10 09:00:35 INFO TaskSchedulerImpl: Killing all running tasks in stage 0: Stage finished
24/01/10 09:00:35 INFO DAGScheduler: Job 0 finished: collect at /home/hadoop/spark/capture_variable.py:15
, took 8.190830 s
Values with base added: [4, 5, 6, 7, 8]
24/01/10 09:00:36 INFO SparkUI: Stopped Spark web UI at http://10.0.2.15:4040
24/01/10 09:00:36 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
24/01/10 09:00:36 INFO MemoryStore: MemoryStore cleared
24/01/10 09:00:36 INFO BlockManager: BlockManager stopped
24/01/10 09:00:36 INFO BlockManagerMaster: BlockManagerMaster stopped
24/01/10 09:00:36 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator s
topped!
24/01/10 09:00:36 INFO SparkContext: Successfully stopped SparkContext
24/01/10 09:00:37 INFO ShutdownHookManager: Shutdown hook called
24/01/10 09:00:37 INFO ShutdownHookManager: Deleting directory /tmp/spark-9a728986-cd97-4eeb-a495-cce5144
3b3c1
24/01/10 09:00:37 INFO ShutdownHookManager: Deleting directory /tmp/spark-a10f78f8-6997-4365-a3bb-0d56e5d
1c3b8/pyspark-7d0e68c5-0987-4395-8e11-ac470bc6e87d
24/01/10 09:00:37 INFO ShutdownHookManager: Deleting directory /tmp/spark-a10f78f8-6997-4365-a3bb-0d56e5d
1c3b8
hadoop@mainserver1: ~/spark$

```

Working with Key-Value Pairs:

Key-Value pairs are a fundamental concept in Spark, and many operations in Spark work with RDDs of key-value pairs. Key-Value pairs can be created using the map transformation.

Example: Word count using key-value pairs:

\$ vim key_value_pair.py

```
from pyspark import SparkContext
```

```
# Create a SparkContext
sc = SparkContext("local", "KeyValuePairsExample")
```

```
# Sample input data (text)
text_data = ["hello world", "world of Spark", "hello Spark", "hello all", "world is for all of us"]
```

```
# Create an RDD from the input data
lines = sc.parallelize(text_data)
```

```
# Transform the RDD into key-value pairs (word, 1)
word_counts = lines.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)
```

```
# Display the result
print("Word counts:")
for (word, count) in word_counts.collect():
    print(f"{word}: {count}")
```

```
# Stop the SparkContext
sc.stop()
```

<save and exit>

\$ spark-submit key_value_pair.py**Sample output:**


```

24/01/10 09:11:56 INFO TaskSchedulerImpl: Killing all running tasks in stage 1: Stage finished
24/01/10 09:11:56 INFO DAGScheduler: Job 0 finished: collect at /home/hadoop/spark/key_value_pair.py:19,
took 12.577972 s
hello: 3
world: 3
of: 2
Spark: 2
all: 2
is: 1
for: 1
us: 1
24/01/10 09:11:56 INFO SparkUI: Stopped Spark web UI at http://10.0.2.15:4040
24/01/10 09:11:57 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
24/01/10 09:11:57 INFO MemoryStore: MemoryStore cleared
24/01/10 09:11:57 INFO BlockManager: BlockManager stopped
24/01/10 09:11:57 INFO BlockManagerMaster: BlockManagerMaster stopped
24/01/10 09:11:57 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator s
topped!
24/01/10 09:11:57 INFO SparkContext: Successfully stopped SparkContext
24/01/10 09:11:58 INFO ShutdownHookManager: Shutdown hook called
24/01/10 09:11:58 INFO ShutdownHookManager: Deleting directory /tmp/spark-8c39b4c8-997b-45c5-a866-faac621
bed87/pyspark-9f488788-346d-4304-b33a-1cbbc896c84a
24/01/10 09:11:58 INFO ShutdownHookManager: Deleting directory /tmp/spark-8c39b4c8-997b-45c5-a866-faac621
bed87
24/01/10 09:11:58 INFO ShutdownHookManager: Deleting directory /tmp/spark-d0e2e523-e712-489c-8bab-cc97cf5
878a6
hadoop@mainserver1:~/spark$

```

In this example, flatMap is used to split each line into words, map is used to create key-value pairs (word, 1), and reduceByKey is used to count the occurrences of each word.

Understanding closures and key-value pairs is crucial for effectively using Spark for distributed data processing tasks. Closures allow you to send custom functions to Spark workers, and key-value pairs enable powerful transformations and aggregations on your data.

Data Preprocessing

This is a crucial step in the data analysis and machine learning pipeling. It involves cleaning and transformaing raw data into a format suitable for analysis or model training. Using Apache spark we can perform various data preprocessing tasks using DataFrame API.

Some of the command data preproceesing tasks:-

1. Handling Missing Values --> dealing with missing values bcs, it sis critical aspect of data preprocessing and spark provides method to drop or fill missing values
2. Handling Categorical Data ---> encode categorial data using one-hot encoding or numerical encoding
3. Feature scaling --> scale numerical features to standardize their range. Common techniques includes Min-Max scaling or Z-score normalization
4. Feature Engineering -->create a new features or transform existing features to enhance the model's performance
5. String manipulation --> perform operations on string columns, such as tokenization or concatenation
6. Removing Duplicates --> remove duplicates rows from the DataFrame
7. Select Relevant Features --> Select a subset of relevant features for analysis or modelling.
8. Handling Date and time ---> extract information form date and time column
9. Joining DataFrames --> combine multiple DataFrames based on common keys.
10. Persisting data ---> persist or cache the DataFrames for iterative operations

There are many such options available but we need to customize the tasks based on the specifics of the dataset and the goals of the analysis or machine learning model

EDA --> Exploratory Data Analysis

--> understanding and summarizing the main characteristics of a dataset.

Spark Based EDA :

1. Loading data ---> we can load our data into a spark dataframe. Spark supports various data sources (CSV, JSON, Parquet and ..)
2. Data Summary --> get a quick summary of the dataframe, include the schema, basic statistics and count of null values
3. Column Exploration --> explore specific columns in the dataframe, including distinct values, count and basic statistics
4. Data visualizations ---> spark itself doesn't provide extensive visualization capabilities, but we can use other libraries (matplotlib or seaborn) for visualizing summary statistics.
5. Handle missing data --> deal with missing or null values in the dataset
6. Correlation Analysis --> compute correlation between numerical columns in the DataFrame
7. Handle categorical data --> encode categorical data using one-hot encoding

Depending on specific dataset and analysis goals, we may need to perform additional tasks. The Spark DataFrame API provides a rich set of functions for data manipulation, and we can leverage these functions to perform a thorough EDA tailored to your needs.

Introduction to Apache Kafka

- Introduction to Kafka
- Working with Kafka using Spark
- Spark streaming Architecture
- Spark Streaming APIs
- Building Stream Processing Application with Spark

Lab Assignment

- Execute the spark streaming with Kafka

Lecture

- Setting up Kafka Producer and Consumer
- Kafka Connect API
- Spark SQL

Lab Assignment

- Run the sparkSQL programs using step-runs for each and every code line
- Run all the SparkSQL programs
- Analyse the election data using spark and provide analysis

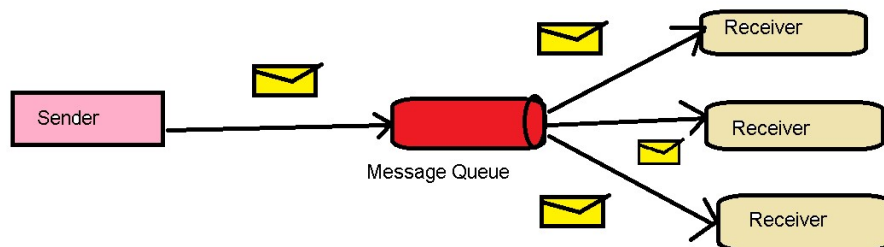
Lecture

- Spark MLlib
- Predictive Analysis

Apache Kafka

An open-source distributed event streaming platform used for building real-time data pipelines and streaming applications.

Developed by Apache foundation but originated at LinkedIn . Kafka is written in Scala and Java. IT is a Publish-subscribe based fault tolerant messaging system.



Event Streaming --> practice of capturing data in real-time from event sources like database, sensors, mobile devices, cloud service, software applications in the form of streams of events, storing these event streams durably for later retrieval, manipulation, processing and reacting to the event streams in real-time as well as any other retrospectively; and routing the event streams to different destination technologies as needed. Hence ensuring a continuous flow and interpretation of data so that the right information is at the right place at the right time.

Event Streaming Use Cases:-

- To process payments and financial transactions in real-time, such as in stock exchanges, banks, and insurances.
- To track and monitor cars, trucks, fleets, and shipments in real-time, such as in logistics and the automotive industry.
- To continuously capture and analyze sensor data from IoT devices or other equipment, such as in factories and wind parks.
- To collect and immediately react to customer interactions and orders, such as in retail, the hotel and travel industry, and mobile applications.

- To monitor patients in hospital care and predict changes in condition to ensure timely treatment in emergencies.
- To connect, store, and make available data produced by different divisions of a company.
- To serve as the foundation for data platforms, event-driven architectures, and microservices.

Kafka has 3 key capabilities for event streaming end-to-end

1. To publish (write) and subscribe to (read) stream of events, including continuous import/export of data from other systems
2. To store streams of events durably and reliably for as long as we require.
3. To process streams of events as they occur

All the functionalities is provided in a distributed, highly scalable, elastic, fault-tolerant and secure manner. We can use bare-metal hardware, virtual machines and containers, and on-premises as well as on the cloud. We can select self-managing kafka environment or we can use a fully managed service offered by a variety of vendors who provide such services.

While reading data (enormous volume of data) we face two main challenges

- How to collect large volume of data
- Analyze the collected data

Now the kafka can help us to address these issues.

Messaging system

Messaging system is responsible for transferring data from one application to another, here we focus on data rather than how to share it. Distributed messaging is based on the concept of reliable message queuing.

Messages are queued asynchronously between client applications and messaging systems.

Messaging system types :

- Point to point messaging systems
- Publisher-subscribe messaging system (pub-sub)

Point - to - point messaging system : --> Messages are persisted in a queue, one or more consumers can consume the message in the queue, but a particular message can be consumed by only one consumer (maximum). Once a consumer reads a message in the queue, it disappears from that queue.

Publisher-subscribe messaging system: --> like in point-to-point consumers can subscribe to one or more topics and consume all the messages in that topic. Message producers are called publishers and message consumers are called subscribers.

