File formats :

Data types : int , float , double , date , timestamp, varchar ,char ,number

Complex data types : Array , struct , map (k,v)

**Databricks : bucketBy, explode\_outer**

Compression codecs( code /decode)

Gzip

Snappy

LZO

Deflate

Exercise 11: Broadcast Variables

Task 1: Basic Broadcast

python

Copy code

# Explanation:

# Broadcast variables are read-only shared variables that can be cached on each machine.

# They are used to efficiently distribute large read-only data structures to all tasks.

# Code:

from pyspark.sql import SparkSession

def broadcast\_example(spark):

data\_to\_broadcast = [1, 2, 3, 4, 5]

broadcast\_data = spark.sparkContext.broadcast(data\_to\_broadcast)

def process\_data(x):

# Access the broadcast variable within the transformation

return x \* broadcast\_data.value[0]

rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])

result = rdd.map(process\_data).collect()

print("Result:", result)

if \_\_name\_\_ == "\_\_main\_\_":

spark\_session = SparkSession.builder.appName("BroadcastExample").getOrCreate()

broadcast\_example(spark\_session)

spark\_session.stop()

Exercise 12: Accumulators

Task 1: Basic Accumulator

python

Copy code

# Explanation:

# Accumulators are variables that can be used to aggregate values across multiple tasks in a parallel manner.

# They are typically used for counters and sums.

# Code:

from pyspark.sql import SparkSession

def accumulator\_example(spark):

accumulator\_var = spark.sparkContext.accumulator(0)

def process\_data(x):

# Increment the accumulator within the transformation

accumulator\_var.add(x)

return x

rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])

result = rdd.map(process\_data).collect()

print("Result:", result)

print("Accumulator Value:", accumulator\_var.value)

if \_\_name\_\_ == "\_\_main\_\_":

spark\_session = SparkSession.builder.appName("AccumulatorExample").getOrCreate()

accumulator\_example(spark\_session)

spark\_session.stop()

Exercise 13: Perpartitions Operations

Task 1: MapPartitions

python

Copy code

# Explanation:

# mapPartitions is a transformation that applies a function to each partition of the RDD.

# Code:

from pyspark.sql import SparkSession

def map\_partitions\_example(spark):

def process\_partition(iter):

# Multiply each element in the partition by 10

return map(lambda x: x \* 10, iter)

rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5], 2)

result = rdd.mapPartitions(process\_partition).collect()

print("Result:", result)

if \_\_name\_\_ == "\_\_main\_\_":

spark\_session = SparkSession.builder.appName("MapPartitionsExample").getOrCreate()

map\_partitions\_example(spark\_session)

spark\_session.stop()

Task 2: Piping Operations

python

Copy code

# Explanation:

# pipe is a transformation that applies an external command to each partition of the RDD.

# Code:

from pyspark.sql import SparkSession

def piping\_operations\_example(spark):

def process\_partition(iter):

# Use an external command to process each element in the partition

command = "awk '{print $1 \* 2}'"

pipe = Popen(command, shell=True, stdin=PIPE, stdout=PIPE, stderr=PIPE)

output, \_ = pipe.communicate(input="\n".join(map(str, iter)).encode())

return map(int, output.decode().split("\n")[:-1])

rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5], 2)

result = rdd.pipe(process\_partition).collect()

print("Result:", result)

if \_\_name\_\_ == "\_\_main\_\_":

spark\_session = SparkSession.builder.appName("PipingOperationsExample").getOrCreate()

piping\_operations\_example(spark\_session)

spark\_session.stop()

Task 3: UDFs (User Defined Functions)

python

Copy code

# Explanation:

# UDFs allow you to define custom functions and apply them to each element in an RDD.

# Code:

from pyspark.sql import SparkSession

from pyspark.sql.functions import udf

from pyspark.sql.types import IntegerType

def udf\_example(spark):

data = [(1,), (2,), (3,), (4,), (5,)]

df = spark.createDataFrame(data, ["value"])

# Define a simple UDF to multiply a value by 10

multiply\_by\_10\_udf = udf(lambda x: x \* 10, IntegerType())

# Apply the UDF to the DataFrame

result\_df = df.withColumn("result", multiply\_by\_10\_udf("value"))

result = result\_df.select("result").collect()

print("Result:", result)

if \_\_name\_\_ == "\_\_main\_\_":

spark\_session = SparkSession.builder.appName("UDFExample").getOrCreate()

udf\_example(spark\_session)

spark\_session.stop()

Exercise 1: Read and Write Different File Formats

python

Copy code

from pyspark.sql import SparkSession

# Create a Spark session

spark = SparkSession.builder.appName("FileFormatsExercise").getOrCreate()

# Read CSV file

csv\_df = spark.read.csv("path/to/csv/file.csv", header=True, inferSchema=True)

# Write to Parquet and JSON

csv\_df.write.parquet("path/to/parquet/output")

csv\_df.write.json("path/to/json/output")

# Read Parquet file

parquet\_df = spark.read.parquet("path/to/parquet/output")

parquet\_df.show()

Explanation:

We create a Spark session using SparkSession.builder.appName("FileFormatsExercise").getOrCreate().

Reading a CSV file into a DataFrame is done with spark.read.csv("path/to/csv/file.csv", header=True, inferSchema=True).

Writing DataFrames to Parquet and JSON formats is accomplished using write.parquet("path/to/parquet/output") and write.json("path/to/json/output").

Reading a Parquet file is done with spark.read.parquet("path/to/parquet/output").

Exercise 2: Working with Different File Systems

python

Copy code

# Read a text file from the local file system

text\_df = spark.read.text("file:///path/to/textfile.txt")

# Write to HDFS

text\_df.write.text("hdfs:///user/spark/text\_output")

# Read Sequence file from HDFS

sequence\_df = spark.read.format("sequence").load("hdfs:///user/spark/sequence\_file")

# Load data from AWS S3

s3\_df = spark.read.csv("s3a://your-bucket/your-file.csv")

Explanation:

Reading a text file from the local file system is done with spark.read.text("file:///path/to/textfile.txt").

Writing to HDFS is achieved with write.text("hdfs:///user/spark/text\_output").

Reading a Sequence file from HDFS is done using spark.read.format("sequence").load("hdfs:///user/spark/sequence\_file").

Loading data from AWS S3 involves using the appropriate path, such as "s3a://your-bucket/your-file.csv".

Exercise 3: Working with Structured Abstract Layers

python

Copy code

# Create a Hive table from a DataFrame

csv\_df.createOrReplaceTempView("my\_table")

spark.sql("CREATE TABLE hive\_table AS SELECT \* FROM my\_table")

# Execute a query on the Hive table

hive\_result = spark.sql("SELECT \* FROM hive\_table")

hive\_result.show()

# Create an Impala table from a Parquet file

parquet\_df.write.format("parquet").mode("overwrite").saveAsTable("impala\_table")

# Perform operations on the Impala table using PySpark

impala\_df = spark.read.table("impala\_table")

impala\_df.show()

Explanation:

Creating a Hive table involves creating a temporary view (createOrReplaceTempView) and executing a SQL statement (CREATE TABLE ... AS SELECT ...).

Executing a query on the Hive table is done using Spark SQL.

Creating an Impala table from a Parquet file involves using the write.format("parquet").mode("overwrite").saveAsTable("impala\_table") method.

Reading and performing operations on the Impala table use standard PySpark DataFrame methods.

Exercise 4: Connecting to Databases

python

Copy code

from pyspark.sql import SparkSession

# Create a Spark session

spark = SparkSession.builder.appName("DatabaseExercise").getOrCreate()

# Connect to Cassandra

spark.conf.set("spark.cassandra.connection.host", "your-cassandra-host")

cassandra\_df = spark.read.format("org.apache.spark.sql.cassandra").option("table", "your\_table").load()

# Read data from Cassandra table

cassandra\_df.show()

# Write data from DataFrame to HBase table

# Note: Use HBase Spark connector or HBase APIs for this task

Explanation:

Connecting to Cassandra involves setting configuration properties using spark.conf.set and reading data using the Cassandra connector.

Reading data from a Cassandra table is done with standard PySpark DataFrame methods like show().

Writing data from a DataFrame to an HBase table depends on the specific HBase connector or APIs being used. Adjust the code accordingly.

These code snippets provide a practical understanding of reading and writing different file formats, working with various file systems, using structured abstract layers, and connecting to databases with PySpark. Feel free to adapt them based on your environment and specific requirements.

User

"• Spark runtime architecture: Cluster manager, Driver, executors, submitting applications and launching applications

Exercise 5: Understanding Spark Runtime Architecture

Task 1: Describe Spark Components

Briefly describe the roles of the Cluster Manager, Driver, and Executors in Spark.

Explain how tasks are distributed and executed across the cluster.

Task 2: Application Lifecycle

Outline the lifecycle of a Spark application from submission to execution.

Discuss the stages involved in the execution of a Spark job.

Exercise 6: Using spark-submit

Task 1: Basic spark-submit

Write a simple PySpark application that prints "Hello, Spark!" to the console.

Submit the application using spark-submit.

Task 2: Various spark-submit Options

Explore various options that can be used with spark-submit (e.g., --master, --deploy-mode, --executor-memory).

Submit the application with different configurations and observe the changes.

Exercise 7: Packaging Code

Task 1: Package a Simple Application

Organize a simple PySpark application into a Python package.

Create a setup.py file for packaging.

Task 2: Build and Submit

Build the package using a build tool like setuptools.

Submit the packaged application using spark-submit.

Exercise 8: Dependency Conflicts

Task 1: Introduce a Dependency Conflict

Add conflicting versions of a library in your PySpark application's dependencies.

Attempt to run the application and observe the conflict error.

Task 2: Resolve Dependency Conflict

Use dependency management tools like Maven or sbt to resolve the conflict.

Ensure that the correct version of the library is used in the Spark application.

Exercise 9: Packaging Python Libraries for Spark Submit

Task 1: Package a Python Library

Create a Python library that contains a custom utility or function.

Package the library using setuptools.

Task 2: Use the Library in a Spark Application

Modify your PySpark application to use the utility or function from the packaged library.

Submit the application with the packaged library using spark-submit.

Exercise 10: Cluster Managers - YARN

Task 1: Set Up YARN Cluster

Configure a YARN cluster for Spark. Ensure YARN is properly installed and configured.

Verify the YARN configuration in the Spark application.

Task 2: Submit Application to YARN

Submit your PySpark application to the YARN cluster using spark-submit.

Monitor the application's progress in the YARN ResourceManager UI.

These exercises cover various aspects of Spark runtime architecture, spark-submit, packaging code, resolving dependency conflicts, packaging Python libraries for Spark, and using YARN as a cluster manager. They aim to provide a hands-on experience and understanding of these key concepts in a practical setting. Adjust the tasks based on your specific environment and needs.

User

need code with explanation

ChatGPT

Certainly! Let's go through the exercises with code snippets and explanations.

Exercise 5: Understanding Spark Runtime Architecture

Task 1: Describe Spark Components

python

Copy code

# Explanation:

# Spark components include the Cluster Manager, Driver, and Executors.

# The Driver is the main program that runs the Spark application.

# Executors are distributed processes responsible for executing tasks.

# The Cluster Manager allocates resources and manages the execution of tasks.

# Code:

# No specific code for this task, as it involves explaining concepts.

Task 2: Application Lifecycle

python

Copy code

# Explanation:

# The application lifecycle involves submission, initialization, execution, and completion stages.

# Spark jobs consist of stages, tasks, and transformations.

# Stages are divided into tasks, and tasks are executed on Executors.

# Code:

# No specific code for this task, as it involves explaining concepts.

Exercise 6: Using spark-submit

Task 1: Basic spark-submit

python

Copy code

# Explanation:

# A simple PySpark application that prints "Hello, Spark!" to the console.

# Code:

# Create a Python file, e.g., hello\_spark.py

from pyspark.sql import SparkSession

def main():

spark = SparkSession.builder.appName("HelloSpark").getOrCreate()

print("Hello, Spark!")

spark.stop()

if \_\_name\_\_ == "\_\_main\_\_":

main()

Task 2: Various spark-submit Options

bash

Copy code

# Explanation:

# Explore various spark-submit options such as --master, --deploy-mode, --executor-memory.

# Code:

# Submit the application with different configurations

# Example:

# spark-submit --master yarn --deploy-mode client --executor-memory 2g hello\_spark.py

Exercise 7: Packaging Code

Task 1: Package a Simple Application

bash

Copy code

# Explanation:

# Package the PySpark application into a Python package using setuptools.

# Code:

# Create a directory structure:

# my\_spark\_app/

# ├── my\_app/

# │ ├── \_\_init\_\_.py

# │ └── hello\_spark.py

# └── setup.py

# setup.py content:

from setuptools import setup, find\_packages

setup(

name="my\_spark\_app",

version="0.1",

packages=find\_packages(),

install\_requires=[

"pyspark",

],

)

Task 2: Build and Submit

bash

Copy code

# Explanation:

# Build the package using setuptools and submit the Spark application.

# Code:

# Build the package

# cd my\_spark\_app

# python setup.py sdist

# Submit the application

# spark-submit --master yarn --deploy-mode client --executor-memory 2g dist/my\_spark\_app-0.1.tar.gz

Exercise 8: Dependency Conflicts

Task 1: Introduce a Dependency Conflict

bash

Copy code

# Explanation:

# Add conflicting versions of a library in your PySpark application's dependencies.

# Code:

# Modify setup.py to include a conflicting library version

# Example:

# install\_requires=[

# "pyspark",

# "conflict-lib==1.2.3",

# ],

Task 2: Resolve Dependency Conflict

bash

Copy code

# Explanation:

# Use dependency management tools like Maven or sbt to resolve the conflict.

# Code:

# Modify setup.py to include the correct version

# Example:

# install\_requires=[

# "pyspark",

# "conflict-lib==1.0.1",

# ],

# Rebuild the package and submit the application.

Exercise 9: Packaging Python Libraries for Spark Submit

Task 1: Package a Python Library

bash

Copy code

# Explanation:

# Create a Python library with a custom utility.

# Code:

# Create a directory structure:

# my\_library/

# ├── my\_utils/

# │ ├── \_\_init\_\_.py

# │ └── custom\_util.py

# └── setup.py

# custom\_util.py content:

def print\_message(message):

print(message)

# setup.py content:

from setuptools import setup, find\_packages

setup(

name="my\_library",

version="0.1",

packages=find\_packages(),

)

Task 2: Use the Library in a Spark Application

python

Copy code

# Explanation:

# Modify the PySpark application to use the utility from the packaged library.

# Code:

# Modify hello\_spark.py to use the library

# from my\_utils.custom\_util import print\_message

# Update setup.py in my\_spark\_app to include my\_library as a dependency

# Rebuild the package and submit the application.

Exercise 10: Cluster Managers - YARN

Task 1: Set Up YARN Cluster

bash

Copy code

# Explanation:

# Set up YARN for Spark. Ensure YARN is properly installed and configured.

# Code:

# No specific code for this task, as it involves YARN setup.

Task 2: Submit Application to YARN

bash

Copy code

# Explanation:

# Submit the PySpark application to the YARN cluster using spark-submit.

# Code:

# spark-submit --master yarn --deploy-mode client --executor-memory 2g dist/my\_spark\_app-0.1.tar.gz

These code snippets and explanations cover the exercises related to Spark runtime architecture,

1. SparkConf class and Various Options

The SparkConf class in PySpark is used to configure various settings and parameters for a Spark application. It allows you to set properties such as application name, master URL, and other Spark-specific configurations.

Example Code:

python

Copy code

from pyspark.conf import SparkConf

from pyspark.sql import SparkSession

# Creating a SparkConf object

spark\_conf = SparkConf()

# Setting application name and master URL

spark\_conf.setAppName("MySparkApplication")

spark\_conf.setMaster("local[2]") # Using local mode with 2 cores

# Additional configurations (e.g., adjusting log levels)

spark\_conf.set("spark.logConf", "true")

# Creating a SparkSession with the configured SparkConf

spark\_session = SparkSession.builder.config(conf=spark\_conf).getOrCreate()

# Your Spark application logic goes here...

# Stopping the Spark session

spark\_session.stop()

2. Parallelism and Serialization

Parallelism:

Parallelism in Spark refers to the ability to perform multiple tasks simultaneously. It can be controlled by adjusting the number of partitions in RDDs or DataFrames.

Serialization:

Serialization is the process of converting data structures or objects into a format that can be easily transported or stored. Spark uses serialization to transmit data between nodes in a cluster.

Example Code:

python

Copy code

from pyspark.sql import SparkSession

# Creating a SparkSession with custom parallelism and serialization settings

spark\_session = SparkSession.builder \

.appName("ParallelismSerializationExample") \

.config("spark.default.parallelism", 4) # Set default parallelism to 4

.config("spark.serializer", "org.apache.spark.serializer.KryoSerializer") # Use Kryo serializer

.getOrCreate()

# Your Spark application logic goes here...

# Stopping the Spark session

spark\_session.stop()

3. Memory Management

Spark provides various configurations to manage memory efficiently. Key configurations include spark.memory.fraction and spark.memory.storageFraction that control the amount of memory allocated for execution and storage.

Example Code:

python

Copy code

from pyspark.sql import SparkSession

# Creating a SparkSession with custom memory management settings

spark\_session = SparkSession.builder \

.appName("MemoryManagementExample") \

.config("spark.memory.fraction", 0.8) # 80% of the executor memory for execution

.config("spark.memory.storageFraction", 0.2) # 20% of the executor memory for storage

.getOrCreate()

# Your Spark application logic goes here...

# Stopping the Spark session

spark\_session.stop()

4. Deciding Executor Memory, Executor Cores, Driver Memory, Driver Cores

The amount of memory and the number of cores allocated to executors and the driver significantly impact Spark job performance. These configurations need careful consideration based on the available resources.

Example Code:

python

Copy code

from pyspark.sql import SparkSession

# Creating a SparkSession with custom resource allocation settings

spark\_session = SparkSession.builder \

.appName("ResourceAllocationExample") \

.config("spark.executor.memory", "2g") # Set executor memory to 2 gigabytes

.config("spark.executor.cores", 2) # Allocate 2 cores per executor

.config("spark.driver.memory", "1g") # Set driver memory to 1 gigabyte

.config("spark.driver.cores", 1) # Allocate 1 core for the driver

.getOrCreate()

# Your Spark application logic goes here...

# Stopping the Spark session

spark\_session.stop()

=========================================

spark.conf.get("spark.sql.autoBroadcastJoinThreshold")

spark.conf.set("spark.sql.autoBroadcastJoinThreshold","100")

order\_itemdf.join(broadcast(productdf),order\_itemdf.order\_item\_product\_id == productdf.product\_id,"inner").show()

spark.conf.set("spark.sql.join.preferSortMergeJoin", "false")

**Spark-submit**

Num-executors

Total-num-executor-cores

Driver-memory

Executor-memory

**--configuring cluster resource to spark application**

--master local , yarn , mesos , kubernetes(k8s)

--deploy-mode client and cluster mode

userdf=spark.read.format("parquet").load("file:///usr/local/spark/examples/src/main/resources/users.parquet")

>>> userdf=spark.read.load("file:///usr/local/spark/examples/src/main/resources/users.parquet")

>

>> userdf=spark.read.parquet("file:///usr/local/spark/examples/src/main/resources/users.parquet")

>>> userdf.printSchema()

root

|-- name: string (nullable = true)

|-- favorite\_color: string (nullable = true)

|-- favorite\_numbers: array (nullable = true)

| |-- element: integer (containsNull = true)

>>> userdf.show()

+------+--------------+----------------+

| name|favorite\_color|favorite\_numbers|

+------+--------------+----------------+

|Alyssa| null| [3, 9, 15, 20]|

| Ben| red| []|

+------+--------------+----------------+

salesdf.write.option("compression","gzip").format("csv").save("file:///home/hduser/target/sales\_gz")

>>>

>>> salesdf=spark.read.format("csv").load("file:///home/hduser/target/sales\_gz/part-00000-e818eaf5-08ef-4873-a60e-219c514fe113-c000.csv.gz")

--------------------------broadcast variable ---------------------------

# Broadcast variables are read-only shared variables that can be cached on each machine.

# They are used to efficiently distribute large read-only data structures to all tasks.

from pyspark.sql import SparkSession

def broadcast\_example(spark):

data\_to\_broadcast = [1, 2, 3, 4, 5]

broadcast\_data = spark.sparkContext.broadcast(data\_to\_broadcast)

def process\_data(x):

# Access the broadcast variable within the transformation

return x \* broadcast\_data.value[0]

rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])

result = rdd.map(process\_data).collect()

print("Result:", result)

if \_\_name\_\_ == "\_\_main\_\_":

spark\_session = SparkSession.builder.appName("BroadcastExample").getOrCreate()

broadcast\_example(spark\_session)

spark\_session.stop()

-------------------------Accumulator--------------------

# Accumulators are variables that can be used to aggregate values across multiple tasks in a parallel manner.

# They are typically used for counters and sums.

# Code:

from pyspark.sql import SparkSession

def accumulator\_example(spark):

accumulator\_var = spark.sparkContext.accumulator(0)

def process\_data(x):

# Increment the accumulator within the transformation

accumulator\_var.add(x)

return x

rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])

result = rdd.map(process\_data).collect()

print("Result:", result)

print("Accumulator Value:", accumulator\_var.value)

if \_\_name\_\_ == "\_\_main\_\_":

spark\_session = SparkSession.builder.appName("AccumulatorExample").getOrCreate()

accumulator\_example(spark\_session)

spark\_session.stop()

===============Reading table from RDB -Mysql ========================

products\_mysqldf = spark.read.format("jdbc").option("url", "jdbc:mysql://localhost:3306/salesdb?allowPublicKeyRetrieval=true&useSSL=false").option("driver", "com.mysql.jdbc.Driver").option("dbtable", "products").option("user", "root").option("password", "root").load()

order\_items\_mysqldf = spark.read.format("jdbc").option("url", "jdbc:mysql://localhost:3306/salesdb?allowPublicKeyRetrieval=true&useSSL=false").option("driver", "com.mysql.jdbc.Driver").option("dbtable", "order\_items").option("lowerBound", "1").option("upperBound", "172168").option("numPartitions", "10").option("partitionColumn","order\_item\_id").option("user", "root").option("password", "root").load()

q= "(select \* from order\_items where order\_item\_order\_id>100) ord"

order\_items\_mysqldf = spark.read.format("jdbc").option("url", "jdbc:mysql://localhost:3306/salesdb?allowPublicKeyRetrieval=true&useSSL=false").option("driver", "com.mysql.jdbc.Driver").option("dbtable",q).option("user", "root").option("password", "root").load()

=======================complex data types ============================

true|10|100|1000|10000|4.0|20.0|2.2222|1969-12-31 15:59:58.174|1970-01-01 00:00:00|hello|hello|k1:v1,k2:v2|100,200|{10, "foo"}

true|20|200|2000|20000|8.0|40.0|4.2222|1970-12-31 15:59:58.174|1971-01-01 00:00:00|||k3:v3,k4:v4|200,300|{20, "bar"}

spark.sql("""

CREATE TABLE IF NOT EXISTS alltypes (

bo1 BOOLEAN,

ti1 TINYINT,

si1 SMALLINT,

i1 INT,

bi1 BIGINT,

f1 FLOAT,

d1 DOUBLE,

de1 DECIMAL,

ts1 TIMESTAMP,

da1 DATE,

s1 STRING,

vc1 VARCHAR(5),

m1 MAP<STRING, STRING>,

l1 ARRAY<INT>,

st1 STRUCT<c1:INT, c2:STRING>

)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '|'

COLLECTION ITEMS TERMINATED BY ','

MAP KEYS TERMINATED BY ':'

STORED AS TEXTFILE

""")

spark.sql("load data local inpath 'file:///usr/local/hive/examples/files/alltypes.txt' into table alltypes")