# HDFS

16/8/2023

Hadoop Distributed File System.

1) YARN - Yet Another Resource ~~Navigator~~ Negotiator
   - Operating System for HDFS

Commands :-

1) jps - To check which are the process running
2) start-dfs.sh && start-yarn.sh
   - 2 sessions to be started. dfs and yarn
3) Create directory in HDFS
   hdfs dfs -mkdir /user/test_dir.
4) Check if directory created
   hdfs dfs -ls /user/
5) Put files to HDFS
   hdfs dfs -put /home/sois/lab2.txt /user/test_dir.
6) hdfs dfs -ls /user/
7) Put multiple files
   hdfs dfs -put t1.txt t2.txt /user/test_dir
8) Copying files from HDFS
   hdfs dfs -get /user/test_dir/lab2.txt ~~/user/~~ /home/sois/Desktop
9) hdfs dfs -cat /user/test_dir/lab2.txt
   To view content
10) To view in web
    → localhost:9780/
    → Go to utilities
    → Browse file System
11) To copy files within HDFS
    hdfs dfs -cp /user/test_dir/lab2.txt /user/test_dir/lab3.txt
12) Move files within HDFS
    hdfs dfs -mv /user/test_dir/lab2.txt /user/testdir/lab4.txt
13) Delete in HDFS
    hdfs dfs -rm /user/test_dir/lab3.txt

23/08/2023 Sqoop :

1) Transfering structured data from RDBMS to HDFS

1) sudo mysql -u root -p
2) use retail_db;
3) show tables;

Step 1:-

*. Grant permission:-

GRANT ALL PRIVILEGES ON *.* TO 'msis'@'localhost' IDENTIFIED By 'bda23'

Step 2:-

Start Hadoop.

Step 3 :-

# List the DBs

sqoop list-databases --connect jdbc:mysql://localhost?useSSL=false --username msis --password bda23

# List the tables

sqoop list-tables --connect jdbc:mysql://localhost/retail_db?useSSL=false --username msis --password bda23

Step 4:-

# Data transfer from local to HDFS

target-dir:- It will not create a folder. Directly dumps all data together

sqoop import --connect jdbc:mysql://localhost/retail_db?useSSL=false --username msis --password bda23 --table orders --target-dir '/user/sqoop1' → folder should not be present

30/08/2023

Warehouse

sqoop import --connect jdbc:mysql://localhost/retail_db?
useSSL=false --username msis --password bda23
--table orders --warehouse-dir 'user/warehouse'

Warehouse copies and stores the data creating folder of name of the table in HDFS.

#importing partial data

sqoop import --connect jdbc:mysql://localhost/retail_db?
useSSL=false --username msis --password bda23 --
table orders --target-dir 'user/partial'

⇒ Primary key should be there to copy, or else it will fail

#copying selected columns

sqoop import --connect jdbc:mysql://localhost/retail_db?
useSSL=false --username msis --password bda23
--columns product_id, product_name, product_price --table
products --target-dir "user/selCol"

#copying all tables excluding few:- Works only for 'Warehouse-dir

sqoop import-all-tables --connect jdbc:mysql://localhost/retail_db?
useSSL=false --username msis --password bda23
--warehouse-dir 'test/tables' --exclude-tables products,
customers

#export data/table from HDFS to mysql:-

sqoop export --connect jdbc:mysql://localhost/new_db?usessl
=false --username msis --password bda23 -- table orders
--export-dir '/user/warehouse/orders'

HDFS folder

newly created table (mysql)

11/09/2023

# HIVE

Hive is a tool used to perform queries on the data stored in HDFS. In HDFS data is stored as simple texts on which queries can not be operated directly. So hive can be used for this.

Step 1: Start HDFS node
start-dfs.sh && start-yarn.sh
Step 2: Use sqoop command to import mysql data from local to HDFS
Step 3: In mysql check for the schema of copied table.
DESC orders;
Step 4: Open new terminal and use the command hive
Step 5: It will prompt the hive prompts.
Step 6: In hive create new database and tables
a) create database Test;
b) create table Test.orders (order_id int, department_id int, order_status varchar(45)) row format delimited fields terminated by ',' stored as textfile;
Step 7: In HDFS create a backup of order table.
a) hdfs dfs -mkdir /user/TempTables
b) hdfs dfs -cp /user/orders/ /user/TempTables
Step 8: Go to hive prompt load data from HDFS
a) load data inpath '/user/orders' into table Test.orders;
This will move the data from HDFS to Hive (This is like cut-copy-paste) This data will not be present in HDFS unless we explicitely create a copy of it as given in step 7.
We can not move data back into HDFS from Hive.
Step 9: All mysql commands except for update works fine in Hive.
a) select * from orders; → To see the data

27/09/2023

## Map-Reduce :-

1) In Eclipse create a new Java project
2) In project create a Java class.
3) Copy word count code to java class
4) Right click on project and select Build Path → Configure Build Path
5) Select Libraries → Add external Libraries (JARs)
6) In hadoop opt → common → hadoop-common-3.2.1.jar (4.1MB)
7) In hadoop → mapreduce → hadoop-mapreduce-client-core-3.2.1.jar
8) Select these 2 jars apply
9) select the project → Right click → Export → JAR file → Next → Select the jar file name & save it in eclipse-workspace → Next → Next → Finish
10) Oper terminal → gedit test.txt # creating an input text file
11) Copy this file to HDFS, create directory & save
    hdfs dfs -mkdir /user/map_reduce/input
    hdfs dfs -put /home/sois/test.txt /user/map_reduce/input
12) Open new terminal
13) cd eclipse-workspace
14) # check if the created jar is present
    ls
15) Executing the map-reduce task
    hadoop jar jar_name.jar packagename.java-classname /user/map_reduce/input/test.text /user/map_reduce/output1
    (/user/map_reduce/input/test.text: input text file; /user/map_reduce/output1: Where the output file is generated with count)

30/09/2023

# Spark:-

```
import pyspark
from pyspark.sql SparkSession
spark = SparkSession.builder.appName('RDD example').
                                        getOrCreate()
#spark session created

#type of spark
type(spark)
```

#first way of creating RDD is parallelize method.
#second way of creating is reading a datasource and putting it in an RDD
#Third way of creating is using transformations.

03/10/2023

## RDD using parallelizing method:-

- parRDD = spark.sparkContext.parallelize(['Hope all is well', 'I'm Nidhi'])
  ⇒ It can have any number of rows, but only one column
  ⇒ Type of data inside RDD will always be String
- parRDD.collect() → To view the data
- parRDD.count() → Returns the total no. of elements in RDD

Count and collect are examples of Actions.

- flatMap() is a transformation
  → Use it whenever we need 1:n mapping
- map() → use it when we need 1:1 mapping

```
parRDD.flatMap(lambda data: data.split(' ')).collect()
['Hope', 'all', 'is', 'well', 'I'm', 'Nidhi']
```

- parRDD.flatMap(lambda data: data.split(' ')).map(lambda d: (d,1)).collect()

```
[('Hope',1), ('all',1), ('is',1), ('well',1)
('I'm',1), ('Nidhi',1)]
```

parRDD.flatMap(lambda data: data.split(' ')).map(lambda d:(d,1)).
reduceByKey(lambda a,b:a+b).collect()

parRDD.flatMap(lambda d:d.split(' ')).map(lambda d:(len(d),1)).
reduceByKey(lambda a,b:a+b).collect()

parRDD.flatMap(lambda d:d.split(' ')).map(lambda d:((len(d),1))
.max()

→ It will give the tuple with largest length

5/10/2023

RDD using datasource method :-

→ numrdd = sparks.sparkContext.textFile("nums.txt")
→ numrdd.collect()
→ numrdd.flatMap(lambda d:d.split(' ')).split()
→ r1 = numrdd.flatMap(lambda d:d.split(' ')).filter(lambda d:
len(d)>0).collect()
→ def clean(data):
    pat = ['0','1','2','3','4','5','6','7','8','9']
    out = ''
    for ele in data:
        if ele in pat:
            out = out + ele
    return out
→ r1.map(lambda d:len(clean(d))).filter(lambda d:len(d)>0).
collect()
→ r1.map(lambda d:clean(d)).filter(lambda d:len(d)>0).
map(lambda d:int(d)).collect()

05

2/10/2023

# Converting an RDD to dataframe

df = rdd.toDF
df.show()
# showing the structured data as rows & columns

```
#Renaming the column name
df.withColumnRenamed("_1","currency")

#Creating a dataframe with column names all together
column = ["currency","value"]
df1 = spark.createDataFrame(RDD).toDF(*column)
df1.columns() # Gives all column names
df1.printSchema() # Gives the schema

#Directly creating a dataframe if we have a structured dataSource
df2 = spark.createDataFrame(data=inputdata, schema=column)

csvdf= spark.read.csv('',header=True)
textdf =spark.read.text('')
jsondf = spark.read.json('')
exceldf = spark.read.excel('')
```

03

```
#Create dataframe using schema:-
from pyspark.sql.types import *
sch = "Name STRING, RegNo STRING, marks INT"
csv2 = spark.read.csv("path", header=True, schema=sch)

#Infer schema
  #Need not go write DDL statements
  #Use it only for small size dataset (rows are less)
  #Can be used if more number of columns are present
csv3 = spark.read.csv("path", inferschema)

#Built-in Date-Time Operations:-

from pyspark.sql.functions import *

df = spark.range(5).withColumn('Today', current_date()).
                    withColumn('Now', current_timestamp())
df = df.withColumn('New Date', date_add('Today', 5))
```

```
df = df.withColumn('Previous date', date_sub('Today', 5))
df = ?e
df.select(abs(datediff('Prev date', 'New Date'))).show()
    #Getting the days difference

#lit is the literal function
df = df.withColumn('String Date', lit('07-12-2022'))
df.withColumn('Proper Date', to_date('String Date', 'dd-mm-yyyy
# converts string type to date data type

# Changing the default date format from 'yyyy-mm-dd' to
  'dd-mm-yyyy'
df=df.withColumn('Proper Date', date_format(to_date('String Dat
  'dd-mm-yyy'), 'dd-mm-yyyy'))

df.select(quarter(to_date(lit('2022-12-07')))).show()
df.select(year(to_date(lit('2022-12-07')))).show()
df.select(month(to_date(lit('2022-12-07')))).show()
df.select(weeks(to_date(lit('2022-12-07')))).show()
```

17/10/2023

Fire dataset :-

```
import pyspark
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('Fire DF').getOrCreate()

from pyspark.sql.functions import *
from pyspark.sql.types import *

df = spark.read.csv('dataset/Fire.csv', header=True, inferschema
  df.show()                                          =True)
  df.printSchema()
df1 = df.select('Call Type', 'Call Date', 'City', 'Zipcode', 'Neighborhood
  'Delay')
df1.show()
```

17/10/2023

df2 = df1.withColumn('Date', to_date(col

df2 = df1.withColumn('Date', to_date(col('callDate'), 'dd/mm/yyyy'))
drop('callType'))
Date

df3 = df2.withColumn('Year', year(col('Date'))).show()

df3 = df2.withColumn('Year', year(col('Date')))\
.withColumn('Month', month(col('Date')))\
.withColumn('Week', weekofyear(col('Date')))

Any python method should be stored as UDF - User Defined Function

def method(data):
....

newUdf = udf(method, StringType())
df3.withColumn('Season', newUdf(col('month'))).show()

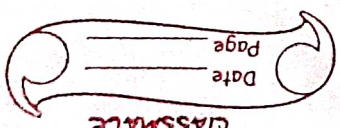
#converting pyspark dataframe to pandas dataframe

import pandas as pd
import matplotlib.pyplot as plt
panda_df = clean_df.select('Year')\
.groupBy('Year').count()\
.orderBy('Year', ascending=True).toPandas()
panda_df.plot.line(x='Year', y='count')
plt.show()

20/10/2023 from pyspark.sql.window import window

df = spark.read.csv('', header=True, inferschema=True)

win = Window.partitionBy('Program').orderBy(col('Marks').desc())



```
programRank = df.withColumn('Rank', rank().over(win))
                                    denseRank()

programRanks.filter(col('Rank')<4).show()

season_count = clean_df.select('callType', 'season')\
              .filter(col('Year')==2014)\
              .groupBy('Season', 'callType')\
              .count()\
              .withColumn('Ranks', dense_rank().over(window.
partitionBy('Season').orderBy(col('count').desc())))\
              .orderBy('Season', 'Rank', ascending=[1,1])
```

Which week in 2018 had most fire calls?

```
clean_df.select('week')\
      .where(col('Year')==2018)\
      .groupBy('Weeks')\
      .count()\
      .orderBy('count', ascending=False).show()
max_month.select('weeks', 'count').filter(col('count')== max_month.
  agg({'count':'max'}).collect()[0][0])
```

28/10/2023

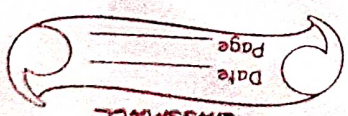
```
clean_df.createOrReplaceTempView('TestView')
spark.sql("SELECT CallType, City From TestView WHERE Year==2014
 and Month==10").show()
```

Streaming data:-

```
import pyspark
from time import sleep
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

spark = SparkSession.builder.appName('Basic streaming').getOrCreate
                                                        ()
```

29/10/2023

```
# Get schema of dataset

static = spark.read.json("D:/Dataset/activity-data")
                          Location of the file (Don't give filename)
static.printSchema()
dataSchema = static.schema

#create streaming dataset
streaming = spark.readStream.schema(dataSchema).option("maxFilesPerTrigger", 1)\
        .json("D:/Dataset/activity-data")

#Get required info from dataset
activityCounts = streaming.groupBy("gt").count()

# Specify the number of shuffle partitions
spark.conf.set("spark.sql.shuffle.partitions", 1)
  Since we have only one system, partition = 1. In clusters we can give any value
#Generate query by specifying output sink using writeschema
activityQuery = activityCounts.writeStream.queryName("activity_count")\
    .format("memory").outputMode("complete")\
    .start()

# Any number of queries (usecases) can be created
~~user_count~~
userCounts = streaming.groupBy("User").count()
userQuery = userCounts.writeStream.queryName("user_count")\
    .format("memory").outputMode("complete")\
    .start()

#Display result
```



```
for x in range(5):
    spark.sql("SELECT * FROM activity_count").show()
    spark.sql("SELECT * FROM user_count").show()
    sleep(10)

# Repeat the proce non stop
activityQuery.awaitTermination()
```

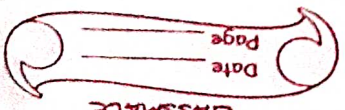
31/10/2023

## Bank Data generator :-

```
count = 301
for x in range (0,100):
    data = {}
    readings = [ ]
    for iter in range(0,10):
        for i in range(0,20):
            data['Acc No'] = acNumbers[random.randInt(0,5)]
            data['Transaction'] = transType[random.randInt(0,1)]
            data['Time stamp'] = str(datetime.now())
            data['Amount'] = random.randInt(1,1000000)
            readings.append(data.copy())
        time.sleep(1)
    fname = 'trans' + str(count) + '.json'
    files = open('./data/'+fname, 'w')
    files.write(json.dumps(readings))
    files.close()
    count += 1
```

## Streaming application :-

```
userSchema = StructType().add('Acc No', 'Integer').add("Transaction",
"string").add("Time Stamp", "string").add("Amount", "integer")
streaming = spark.readStream.schema(userSchema)\
        .option("maxFilesPerTrigger', 1)\
        .json('C:\Users\Data\data')
```



31/10/2023

```
accountDF = streaming.withColumn('Time', to_timestamp(col
    ('Time stamp'))).drop('Time stamp')
spark.conf.set("spark.sql.shuffle.partitions", 1)

accCount = accountDF.groupBy('Acc No').count()
accountQuery = account.writeStream.queryName('count_account')
    .format('console')\
    .outputMode('complete')\
    .start()
accountQuery.awaitTermination()
```

↓ similar to table name

If the dataframe is result of aggregated data, then only we can select 'complete' output mode.
For filter and other cases, select 'update' or 'append' mode.
non-aggregated operation

```
accountQuery = account.writeStream.queryName('min_amount')\
    .trigger(processingTime="20 seconds")
```

It waits for time specified before processing the new data

```
windowedCount = accountDF\
    .withWaterMark("Time", "10 minutes")\
    .groupBy(col('Acc No'), "Time")\
    .count()
```

Watermarks is used when we want the data in out of order

```
query = windowedCount.writeStream)\
    .format("csv")\
    .trigger(------)\
    .option("checkpointLocation", "./checkPoint/")\
    .option("path", "./outputDir/")\
    .outputMode("append")\
    .start()
```