Introduction to Apache Spark

An open-source, distributed processing system used for big data workloads.	
Utilizes in-memory caching, and optimized query execution for fast analytic queries against data of any size.	
Provides:	
Development APIs. Batch processing, interactive queries, real-time analytics, machine learning, and graph processing	
Apache Spark vs. Apache Hadoop	
Hadoop MapReduce is a programming model for processing big data sets with a parallel, distributed algorithm.	
With each step, MapReduce reads data from the cluster, performs operations, and writes the results back to HDFS. Because each step requires a disk read, and write, MapReduce jobs are slower due to the latency of disk I/O.	Because each step requires a disk read, and write, MapReduce jobs are slower due to the latency of disk I/O.
Spark was created to address the limitations to MapReduce	
Spark does processing in-memory, reducing the number of steps in a job, and by reusing data across multiple parallel operations.	With Spark, only one-step is needed where data is read into memory, operations performed, and the results written back

DDL

Data Definition Language	
Resilient Distributed Dataset (RDD) is the fundamental data structure of Spark	
immutable (and therefore fault-tolerant) Distributed collections of objects of any type.	
Each Dataset in Spark RDD is divided into logical partitions across the cluster	thus can be operated in parallel, on different nodes of the cluster.
RDD features	
Lazy Evaluation	Transformation do not compute the results as and when stated
In-Memory Computation	Data is kept in RAM (random access memory) instead of the slower disk drives
Fault Tolerance	Tracks data lineage information to allow for rebuilding lost data automatically on failure
Immutability	Immutability simply rules out lots of potential problems due to various updates from varying threads at once.
	Having Immutable data is safer to share across processes

DDL (cont)

Partitioning	Each node in a spark cluster contains one or more partitions.
Two ways to apply operations on RDDs	
1, Transformation − These are the operations, which are applied on a RDD to create a new RDD. Filter, groupBy and map are the examples of transformations.	Narrow Transformations: In this type, all the elements which are required to compute the records in a single partition live in that single partition.
	Wide Transformations: Here, all elements required to compute the records in that single partition may live in many of the partitions of the parent RDD. These use groupbyKey() and reducebyKey().
2, Action − These are the operations that are applied on RDD, which instructs Spark to perform computation and send the result back to the driver.	count(), collect(), take(n), top(), count value(), reduce(), fold(), aggregate(), foreach().
Create Dataframees	

DDL (cont)

via CSV	df=spark.read.option("header",True) \ .csv("/tmp/resources/simple-zipcodes.csv")
	If you have a header with column names on your input file, you need to explicitly specify True
	df = spark.read.csv("path1,path2,path3") ; df = spark.read.csv("Folder path")
	Using the read.csv() method you can also read multiple csv files, just pass all file names by separating comma as a path
	Using nullValues option you can specify the string in a CSV to consider as null. For example, if you want to consider a date column with a value "1900-01-01" set null on DataFrame.

DDL (cont)

Parition	Used to partition the large dataset (DataFrame) into smaller files based on one or multiple columns while writing to disk
	df.write.option("header",True) \ .partitionBy("state") \ .mode("overwrite") \ .csv("/tmp/zipcodes-state")
	PySpark splits the records based on the partition column and stores each partition data into a sub-directory., If we have a total of 6 different states hence, it creates 6 directories
	df.write.option("header",True) \ .partitionBy("state","city") \ .mode("overwrite") \ .csv("/tmp/zipcodes-state")

DDL (cont)

t creates a folder hierarchy for each partition; we have mentioned the first partition as state followed by city hence, it creates a city folder inside the state folder (one folder for each city in a state).

Queries

```
from pyspark.sql import functions as F
# Select Columns
df.select("firstName").show()
df.select("firstName","lastName") \
 .show()
# split multiple array column data into rows
df2 = df.select(df.name,explode(df.subjectandID))
# Show all entries where age >24
df.select(df['age'] > 24).show()
# Show name and 0 or 1 depending on age > or < than 30
df.select("Name",
 F.when(df.age > 30, 1)
 .otherwise(0)) \
 .show()
# Show firstName if in the given options
df[df.firstName.isin("Jane","Boris")].collect()
# Show firstName, and lastName if lastName is Smith.
df.select("firstName",
 df.lastName.like("Smith"))
 .show()
```

Queries (cont)

```
# Like also excepts wildcard
matches.
df.select("firstName",
 df.lastName.like("%Sm"))
 .show()
# Show firstName, and TRUE if
 df.lastName \ lastName starts
with Sm
Startswith - Endswith
df.select("firstName
 .startswith("Sm")) \
 .show()
# Show last names ending in th
df.select(df.lastName.endswith-
("th"))\
 .show()
# Return substrings of firstName
 Substring
df.select(df.firstName.substr(1,
3) \
 .alias("name")) \
 .collect()
 Between
# Show values where age is
between 22 and 24
df.select(df.age.between(22,
24)) \
 .show()
# Show all entries in firstName
and age + 1
df.select(df["firstName"],df["-
age"]+ 1), .show()
```

DML

Dealing with nulls

DML (cont)

df= df.na.drop(how = 'any', thresh = 2)	To drop null values we use the na function with the drop() attribute.	how: ‘any’ or ‘all’. If ‘any’, drop a row if it contains any nulls. If ‘all’, drop a row only if all its values are null
		thresh: default None If specified, drop rows that have less than thresh non-null values. This overwrites the how parameter.
		subset: optional optional list of column names to consider.
To fill nulls	df.na.fill(50)	

union() method of the DataFrame is used to merge two DataFrame’s of the same structure/schema.

DML (cont)

unionDF = df.union(df2)	returns the new DataFrame with all rows from two Dataframes regardless of duplicate data.
use the use the distinct() function to return just one record when duplicate exists.() function to return just one record when duplicate exists.	use the distinct() function to return just one record when duplicate exists.

Creating a Session

```
import pyspark # importing the
module

 # importing the SparkSession
module
from pyspark.sql import SparkS-
ession
 # creating a session
session = SparkSession.build-
er.appName('First App')
.getOrCreate()
 # calling the session variable
session
```

Creating delta tables

```
# Define the input and output
formats and paths and the table
name.
read_format = 'delta'
write_format = 'delta'
load_path = '/databricks-datas-
ets/learning-spark-v2/people/p-
eople-10m.delta'
save_path = '/tmp/delta/people-
10m'
table_name = 'default.people10m'
# Load the data from its source.
people = spark \
  .read \
  .format(read_format) \
  .load(load_path)
# Write the data to its target.
people.write \
  .format(write_format) \
  .save(save_path)
# Create the table.
spark.sql("CREATE TABLE " +
table_name + " USING DELTA
LOCATION '" + save_path + "'")
session
```

Data preprocessing

```
To select one or multiple
columns the select() function
works
dataframe.select(column_name) #
selecting one column
dataframe.select(column_1,
column_2, .., column_N)
# selecting many columns
dataframe.withColumn()
To add a column the datafr-
ame.withColumn() function
takes two parameters
New column name to add
```

Data preprocessing (cont)

```
Existing column name to use for
(not necessary if the
new column has nothing to do
with the existing column)
# adding columns in dataframe
data = data.withColumn('Age_aft-
er_3_y', data['Age']+3)
to change data type
You would also need cast() along
with withColumn().
 The below statement changes the
datatype from String
 to Integer for the salary
column.
df.withColumn("salary",col("sa-
lary").cast("Integer")).show()
Change a value
Pass an existing column name as
a first argument
and a column as the value to be
assigned as a second argument
df.withColumn("salary",col("sa-
lary")*100).show()
Drop
df.drop("salary") \
  .show()
withColumnRenamed()
rename an existing column
df.withColumnRenamed("gen-
der","sex") \
  .show(truncate=False)
```

Adding columns - df.withColumn('newCol', newVal)
Changing data types - df.withColumn("n-ewCol",col("OldCol").cast("NewD-T")).show()
Changing Values - df.withColumn('oldcol', col("oldcol") operation)

Dropping = withColumnRenamed

Renaming = withColumnRenamed

Sorting and Grouping

df.sort("col", ascending = false)	Default sorting technique used by order by is ASC

df.groupby("col").agg() / df.groupby("a-ge").counr()

Spark SQL

spark.sql(select * from tablename)