**Class: 20 DATE: 05-08-2024**

**Agenda:**

* **Spark Intro**
* **Spark Evolution**
* **Spark Sample Execution**
* **Why Spark is faster?**
* **Spark Lazy**

**2013 🡺** There are more requirements and less possibility.

**HIVE RESTRICTIONS:**

1. Hive is only restricted to HDFS. It cannot create tables on data in Edge node. Its environment dependant, only to HDFS only.
2. No streaming support.
3. No ML support.
4. No entire ETL. It cannot bring data to HDFS, we have to get data using sqoop then hive is used for processing.
5. No Much customization (not good for bad data)
6. Less Cloud support

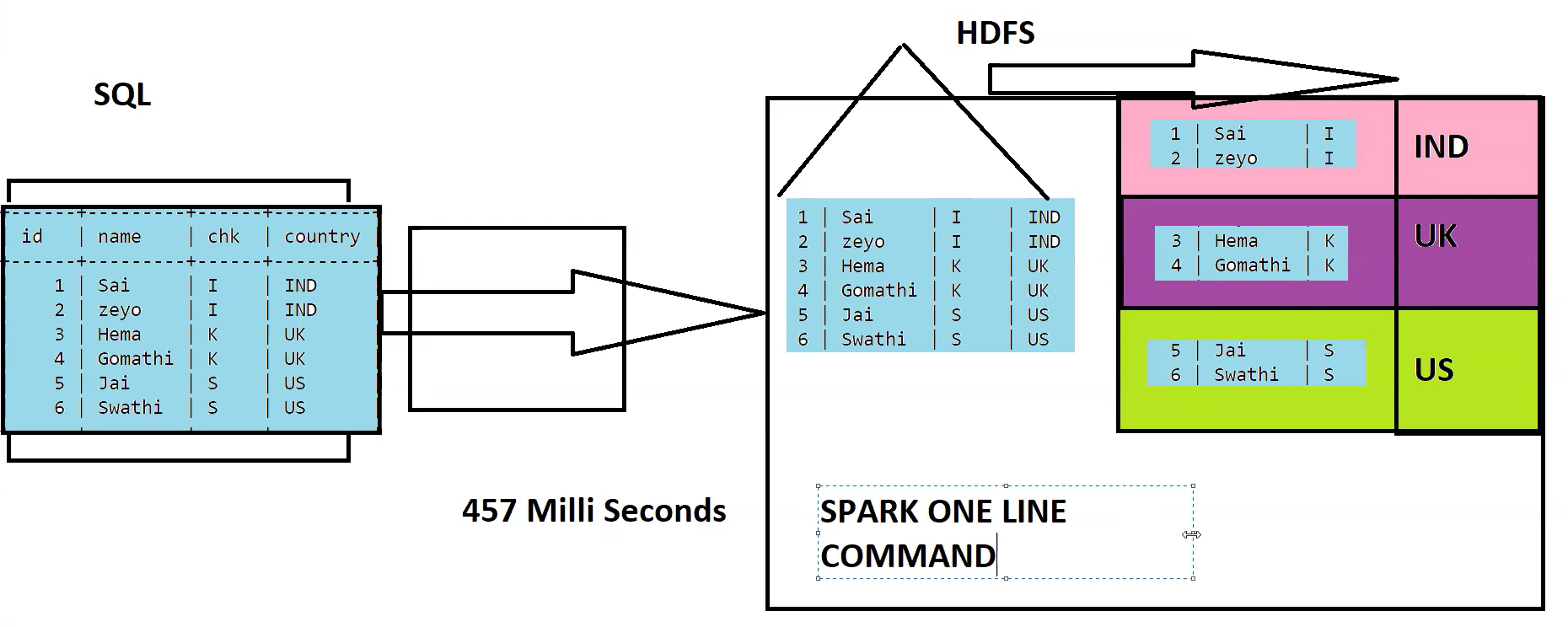
In **2013**, **MATEI ZAHARIA** from **UC Berkley University**, stated that he has a tool(**SPARK**) which is son of MapReduce and capable of:

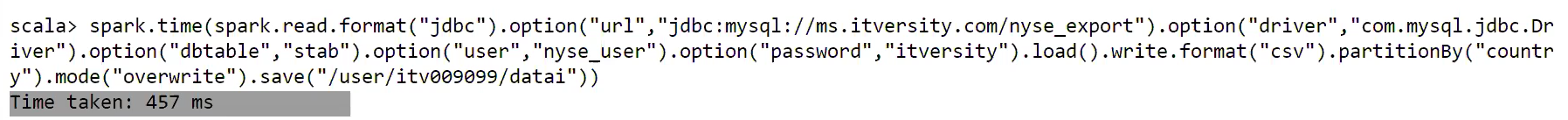
1. Much Simpler
2. Extremely Faster
3. Environment Independent (Linux, HDFS, WINDOWS, MAC, CLOUD STORAGE)
4. Very Good ML Support
5. Very good Streaming Support.
6. Very good SQL support
7. Can do of entire ETL
8. Supports very good customizations.
9. Very good cloud support.
10. Very easy to learn.
11. Good integration

Tag line of spark is **Lightening fast cluster computing**.



**Spark Sample Execution**





**Why Spark is Faster?**

**Why MR is slow and why SPARK is faster?**

We will understand why MR is slow and SPARK is faster using **WORD COUNT** example:

Suppose we have below string:

**“””Spark hadoop**

**Hive spark**

**(new line)**

**Hadoop hive**

**Hadoop sqoop”””**

Both Mapreduce and spark solves wordcount prob in 5 steps:

1. Both will read the data.
2. Both will flatten the data with space.
3. Suffix 1 to each split.
4. GroupBy sum of value.
5. Print the result.

**WORD COUNT PROCESSING STEPS:**

**Read the data:**

**Spark hadoop**

**Hive spark**

**(new line)**

**Hadoop hive**

**Hadoop sqoop**

**Flatten the data:**

**Spark**

**hadoop**

**Hive**

**spark**

**(new line)**

**Hadoop**

**hive**

**Hadoop**

**sqoop**

**Suffix 1 to each split.**

**Spark,1**

**Hadoop,1**

**Hive ,1**

**spark**

**(new line)**

**Hadoop,1**

**Hive,1**

**Hadoop,1**

**Sqoop,1**

**GroupBY words sum of 1’s:**

**Spark,2**

**Hadoop,3**

**Hive, 2**

**Sqoop,1**

**Print the results:**

**Spark,2**

**Hadoop,3**

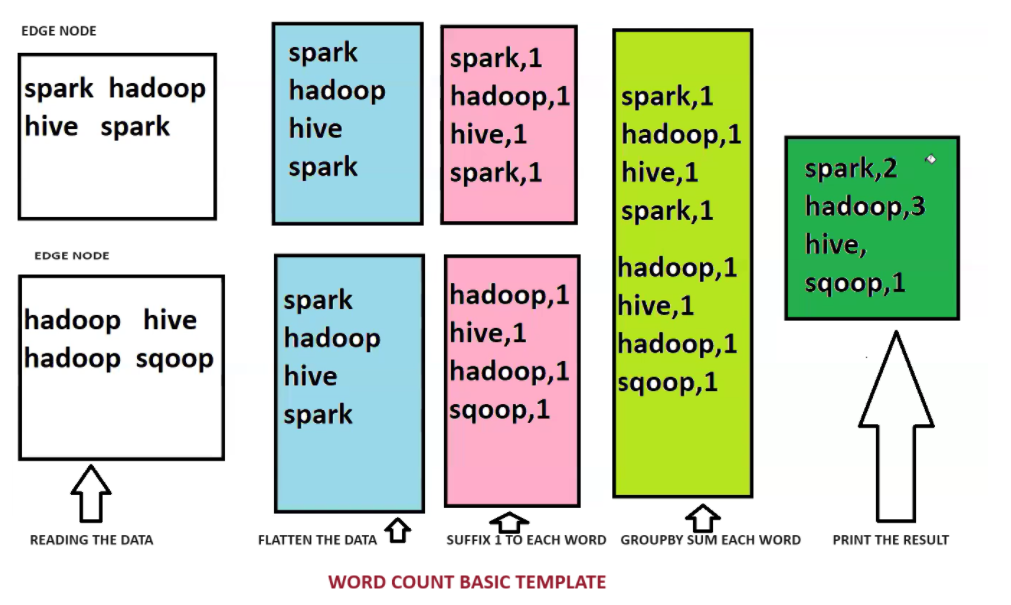
**Hive, 2**

**Sqoop,1**

Both MapReduce and Spark follow the same processing steps. But the difference lies in execution. Zaharia finds a problem in MapReduce which cannot be solved. He just fined tuned MapReduce and created SPARK which is 10x faster.

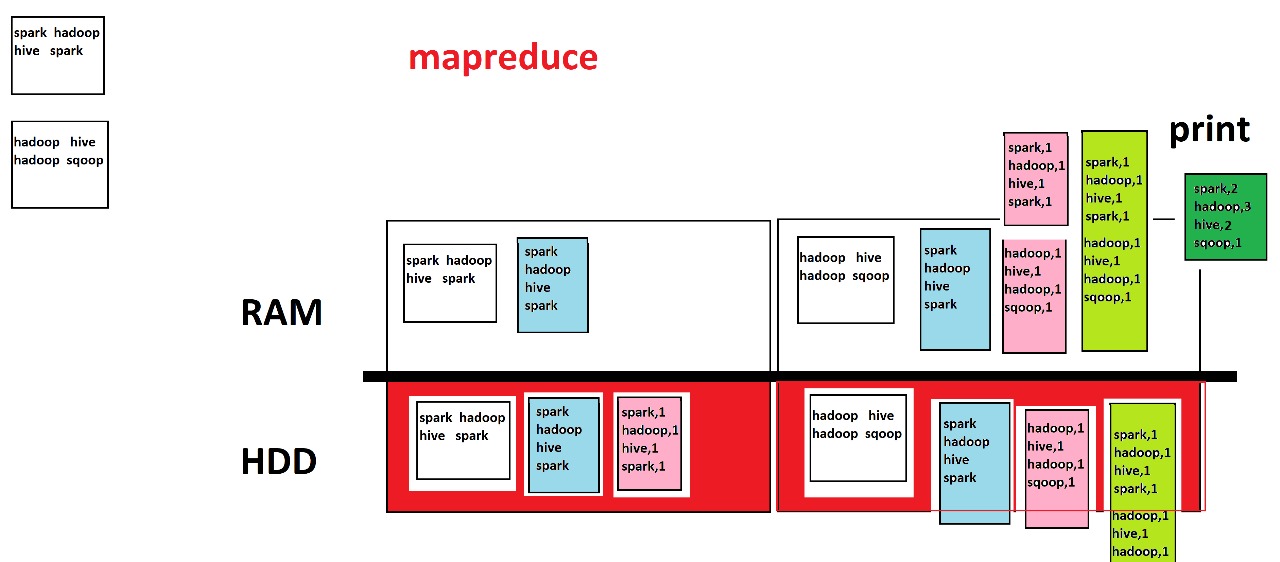
**Execution in MAPREDUCE:**

Word count execution basic template:



When word count program executed in MapReduce,

Initially data will be sitting in HDD, once we read the data. MR will bring data to RAM from HDD. When we flatten the data, a copy of this flattened intermittent data will be stored in HDD. Then suffix 1 gets executed, that suffixed intermittent data will be stored in HDD, then when we group and sum the words that intermittent data will be stored again in HDD. Then Prints the data. Then it refreshes everything and intermittent data will be wiped off.



The issue with Mapreduce is after every step or transformation**, it hits the HDD to store intermittent data**. It is hitting HDD for every transformation **because of its nature**.

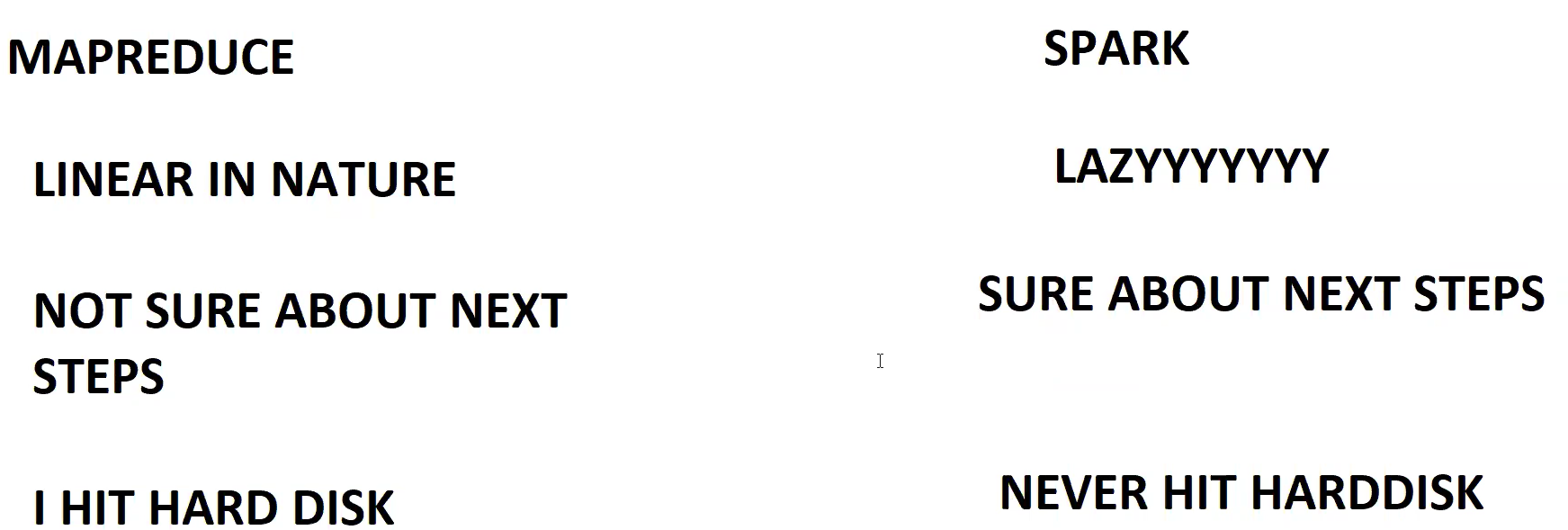
**If the data is traversing data between RAM and HDD its called IO operation, that’s the issue with MapReduce.**

**MAPREDUCE IS LINEAR IN NATURE.**

**NOT SURE ABOUT NEXT STEPS**

**HITS HDD EVERYTIME.**

Zaharia recognized this issue then tuned nature of **SPARK to LAZY, SURE ABOUT NEXT STEPS, NEVER HITS HDD.**

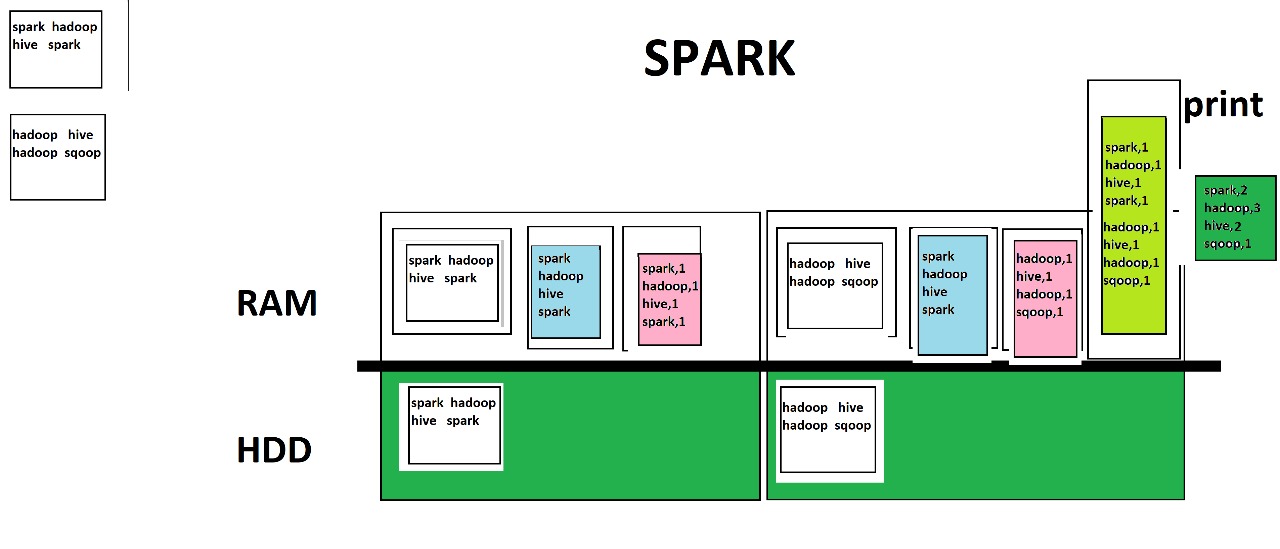


**Execution in SPARK:**

When word count program executed in MapReduce,

Initially data will be sitting in HDD, once we read the data. Spark won’t read the data it just do the planning, when we ask spark to flatten the data spark again do the planning, when we ask spark to suffix 1 to the data spark again do the planning, when we ask spark to GroupBY the data spark again do the planning, When final action gets executed that is **Print statement** , now the execution starts, plan is readily available. It executes read, flatten, suffix 1, groupby then print the data. All the steps till before print are transformations. Spark during its execution it wont hit HDD for any intermittent data storage. There are no IO operations in spark.

**SPARK LAZY Evaluation : Till Action is triggered , No of the Transformations takes place, everything takes place in memory.**



**Class: 21 DATE: 04-08-2024**

**Agenda:**

* **Recap on Spark**
* **Spark Lazy Evaluation hands on**
* **Spark performance run check (MR vs SPARK)**
* **Spark Theory completion**
* **Spark Deployment**

**Spark Lazy evaluation Hands on:**

Suppose we have below data in **data.txt**:

**Sai, Gymnastics**

**Zeyo, Sports,**

**Ravi, Gymnastics**

**Rani, Cricket**

For this we have 3 step process:

1. Read the data. 🡺 Transformation

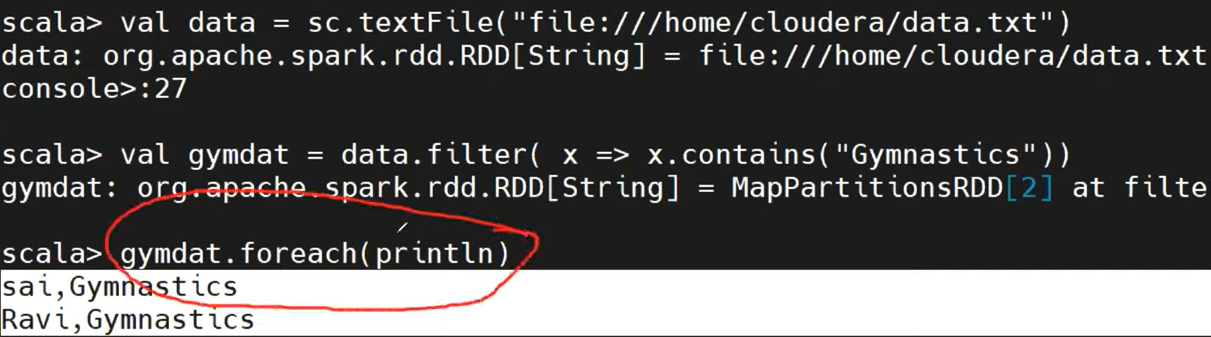
Val data = sc.textFile(‘file:///home/cloudera/data.txt’)

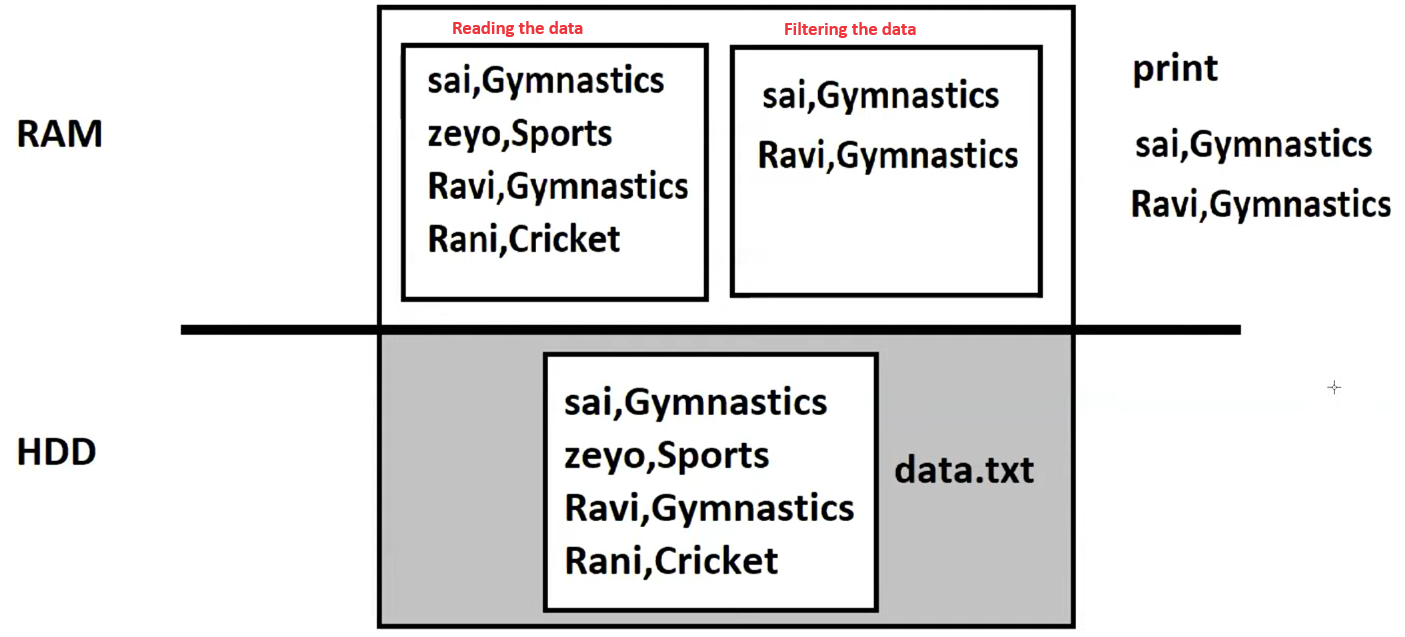
1. Filter rows contains Gymnastics. 🡺 Transformation

Val gymdata = data.filter(x => x.contains(‘Gymanstics’)

1. Print it. 🡺 Spark

Gymdata.foreach(println)



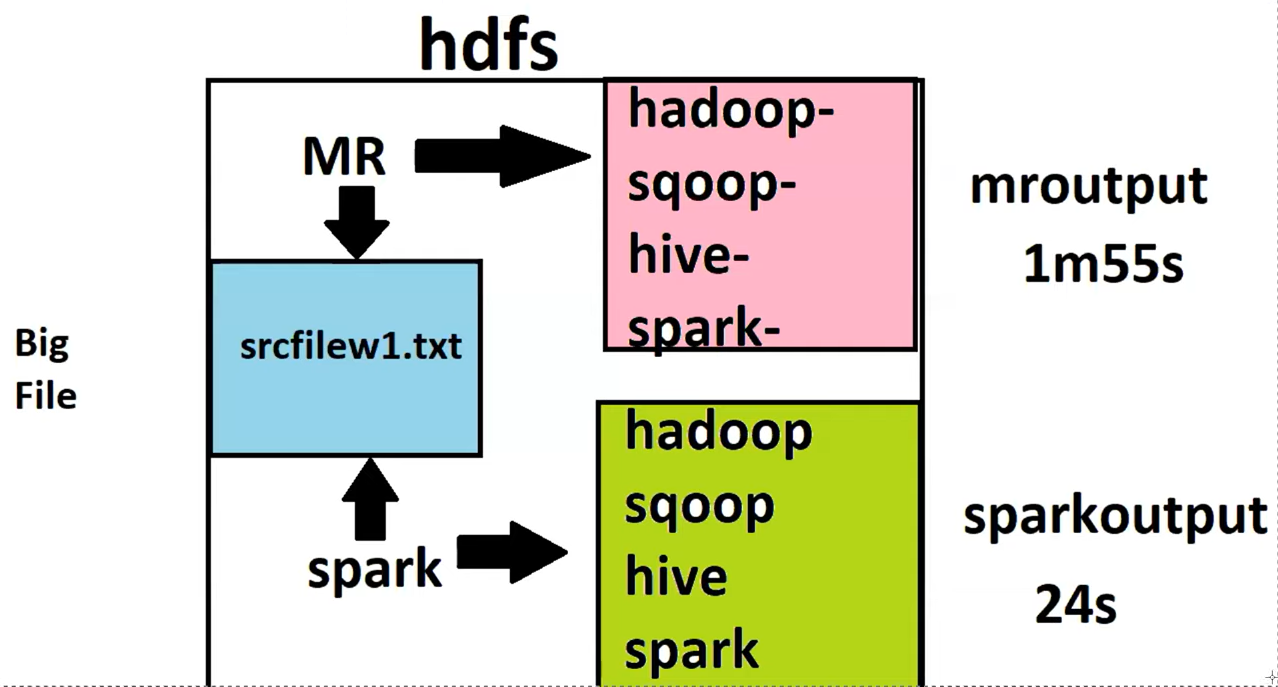


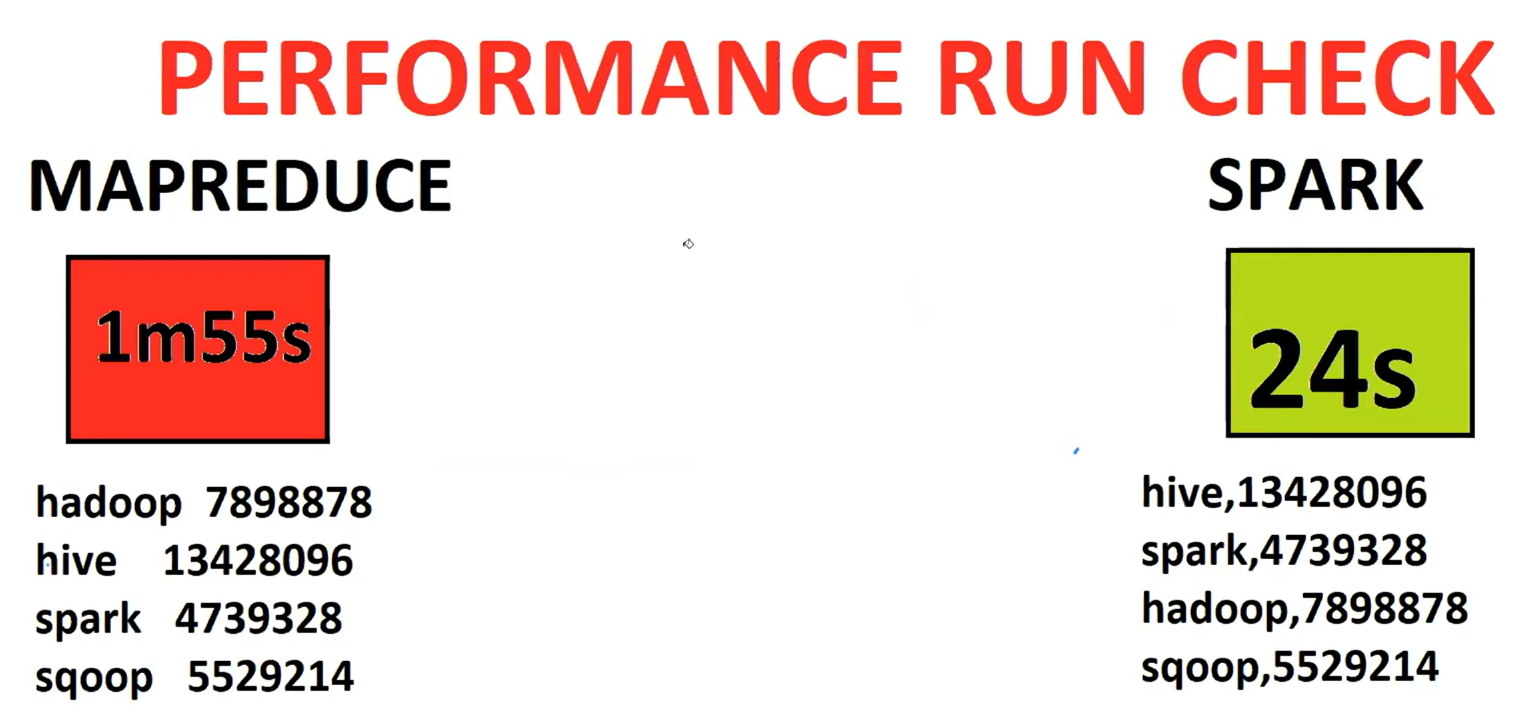
**Spark performance run check (MR vs SPARK)**

To do performance run check/testing we take below scenario:

**Scenario**: Inside HDFS we have file called **srcfilew1.txt** file(large in size, which has words like hadoop, hive, sqoop etc), now we going to do wordcount and mapreduce will write output to **mroutput**(word and its count), spark will write output to **sparkoutput**(word and its count).

Now we will execute and test the above scenario on both Spark and MapReduce.





**DEPLOYMENTS**

Boss Story:

Boss 🡺 mailed Sai, we got some work to do. We have production cluster available. We have **data(dt.txt)** in production cluster. Production cluster name is **zeyobronanalytics2692** . login to this cluster with hostname, username, password.

Sai 🡺 logged in with above details via putty. Checked files.

Boss 🡺 we have to read dt.txt file and filter rows that contains Gymnastics, write the data to gymdata.

Sai 🡺 Boss, that’s fine – that’s simple. I’ll login to production cluster, open spark shell, read the data, filter the data, write the result.

Boss 🡺 Sai, what rubbish your are talking. It’s a production cluster. You will not have access to open sparks shell. You have to run only **spark .py file** to achieve it.

Sai 🡺 Boss, Then where will I develop my code, where will I test my code, where will I test the results.

Boss 🡺 Sai, to test the code, to develop the code, test the result, you should not use production cluster, you should use development cluster. Please check below the credentials of dev cluster, hostname, username, password.

Sai 🡺 Boss, Do I have the same data in dev cluster.

Boss 🡺 we have the data in dev cluster. Please check.

Sai 🡺 Let me start my development.

Open pyspark shell in development cluster. Read the dt.txt file, filter the data then write filtered data to hdfs.

Sai 🡺 I have used dev cluster. I have developed my code, test my code, tested the results also.

Boss 🡺 then what are you waiting for, generate Pyfile with that code. When u generate the Pyfile, please make sure that the path is production paths.

Sai 🡺 Boss, I have code with me but how to generate Pyfile?

Boss 🡺 To generate pyfile you need INTELLIJ or PYCHARM.

* Install Intellij or pycharm in local machine.
* Include all Pyspark Packages
* Put code and Generate Pyfile
* Take it to production (change production path)
* Run it.



**For those who is Not having a set up**

**INTELLIJ/PYCHARM ALTERNATIVE 🡺 Colab**

**https://colab.research.google.com/#create=true**

[**https://tinyurl.com/44jzwkty**](https://tinyurl.com/44jzwkty) **🡺 code**

**Class: 22 DATE: 10-08-2024**

**Agenda:**

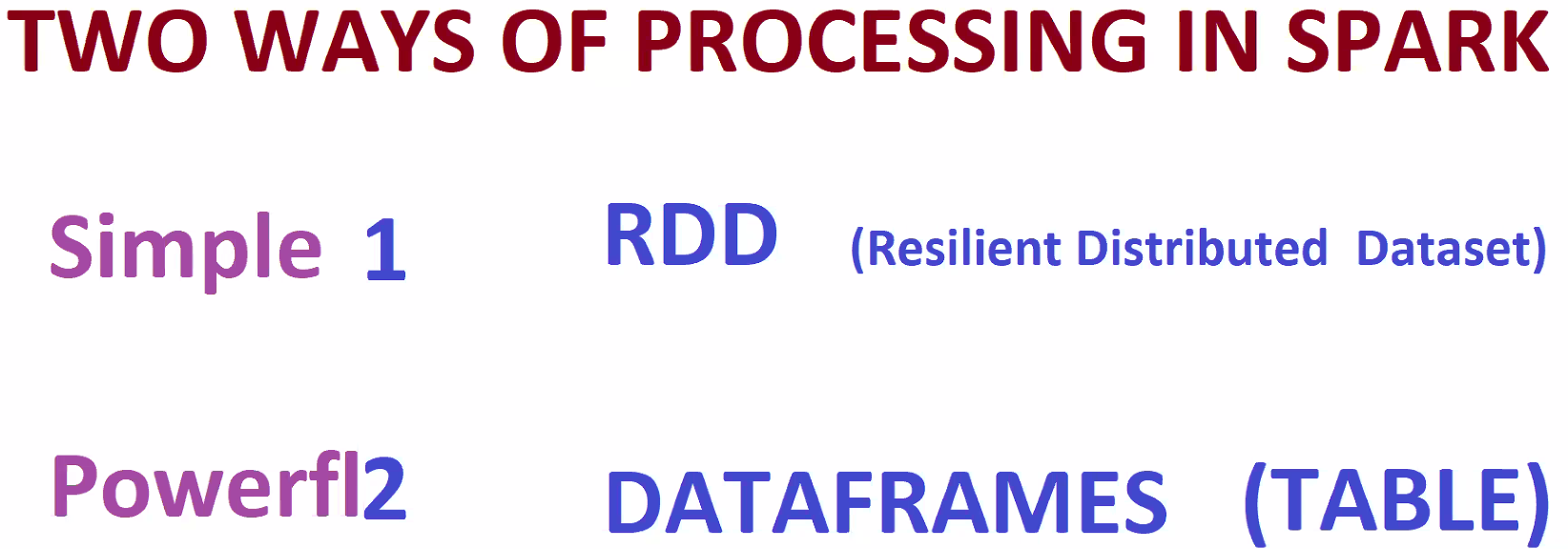
* **Recap**
* **Deployment**
* **Types of Data Processing in Spark**
* **Files Read Scenario – Very important**
* **Start of Handson**

**Types of Data Processing in Spark**

There are only two types of data processing in spark.

1. RDD (Resilient Distributed Dataset) 🡺 very simple and less used in industry. These are programmatical representation
2. DataFrames 🡺 Much Powerful and Widely used. These are table in structure

Both RDD and Data Frame have pros and cons. There is one more processing type that is Dataset, which is combination of DataFrame and RDD but not much used. Dataset is part of Data Frame.



**Files Read Scenario**

Boss story:

Boss 🡺 Sai, you have a file in your computer, path is "D:\Hemanth\spark\_Deploy\pysparkdeploy\data\cust.txt". Read this file and Print it using spark.

Sai 🡺 Zaharia, I wan to read the file and print the content.

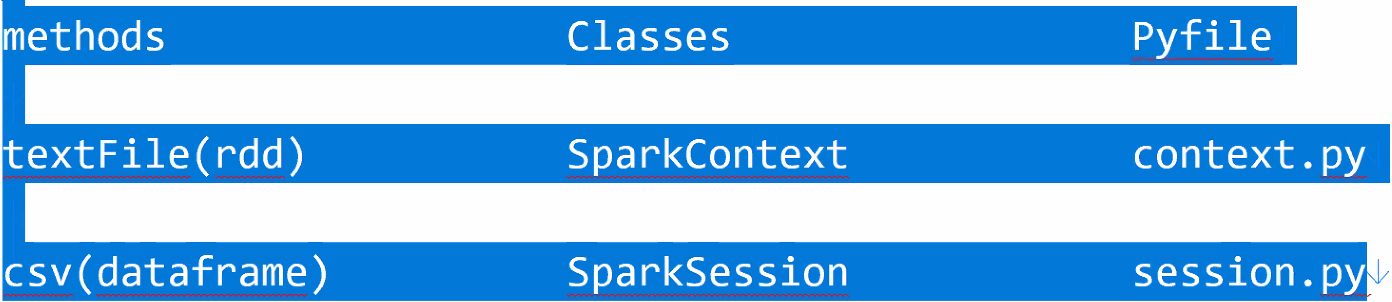
Zaharia 🡺 sai its very simple, it can be achieved **using 2 methods**. They are **textFile** and **csv methods**.

Sai 🡺 what these textFile and csv methods mean and from where are these coming from ?

Zaharia 🡺 these methods would come from **classes. textFile comes from SparkContext class** and **csv method comes from SparkSession class**

Sai 🡺 Zaharia, what are classes mean, where can I find them, where can I see them.

Zaharia 🡺 sai, those classes are coming from **pyfiles**. **SparkContext is coming from context.py** and **SparkSession is coming from session.py**



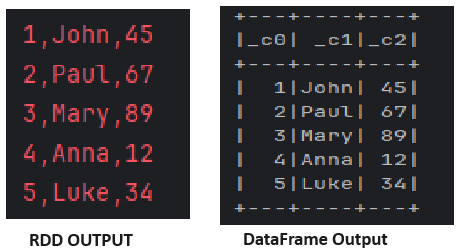
Sai 🡺 Where are those pyfiles, where can I see it, where to use it.

Zaharia 🡺

1. sai, you are going to use intellij/pycharam, open it and create a new project.
2. Install pyspark 🡺 pip install pyspark
3. Once pyspark gets installed, there is a folder called **External Libraries**, open it.
4. Open **site-packages** folder🡺 this is the place where everything starts.
5. Open **pyspark** folder, this is place where we find **context.py**
6. Inside **context.py** we have **SparkContext class**, inside **SparkContext** class we have **textFile** method.
7. Open **External libraries 🡺 site-packages 🡺 pyspark 🡺sql** folder, here we find **session.py**
8. Inside **session.py** we have **SparkSession** class, inside **SparkSession** class we have **read** method, click on **DataStreamReader**, inside DataStreamReader we have **csv** method.
9. We can use above methods with 4 step process as mentioned below:

* Create your own pyfile.
* Call the Classes out (SparkContext, SparkSession)
* Assign a variable to classes (giving powers to variables)
* Call the methods from Classes (textFile and csv) using those variables.
* Use those methods to read and print the data.

**Code: Reading a file**

import os  
import sys  
python\_path = sys.executable  
os.environ['PYSPARK\_PYTHON'] = python\_path  
os.environ['JAVA\_HOME'] = r'C:\Users\91709\.jdks\corretto-1.8.0\_422'  
#################################################################  
  
from pyspark import SparkConf  
from pyspark import SparkContext   
from pyspark.sql import SparkSession  
  
conf = SparkConf().setAppName("pyspark").setMaster("local[\*]")

.set("spark.driver.host","localhost")

.set("spark.default.parallelism","1")  
sc = SparkContext(conf=conf)  
  
spark = SparkSession.builder.getOrCreate()  
  
**# Below is the RDD type of reading file**  
sc.textFile("file:///D:/Hemanth/spark\_Deploy/pysparkdeploy/data/cust.txt").foreach(print)  
########################################################################################  
  
**# Below is DataFrame type of reading file**  
spark.read.csv("file:///D:/Hemanth/spark\_Deploy/pysparkdeploy/data/cust.txt").show()

**Class: 23 DATE: 11-08-2024**

**Agenda:**

* **Recap on Spark, Deployment, file reads**
* **Python coding info**
* **Base Prints**
* **List iterations, integer list iterations, string Iterations**
* **Lambda iterations**

**List iterations in Pyspark:**

**Given a list, process a list in such we have to add 2 to each element in pyspark.**

d = [1,2,3,4]  
rdd = sc.parallelize(d) 🡺 **will convert raw list to RDD**  
print()  
print("=====RDD print =====")  
print(rdd.collect()) 🡺 **to print RDD**

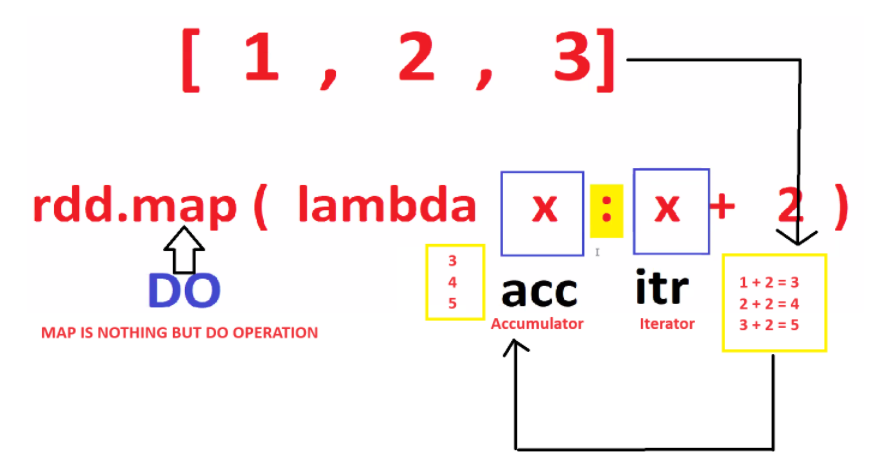
**PySpark parallelize()**is a function in SparkContext and is used to create an RDD from a list collection.

**PySpark rdd.collect() :** Return a list that contains all of the elements in this RDD.

For every rdd operations, we have one powerful support known as **LAMBDA**

**Lambda** is extremely powerful and it need an operation support, that is nothing but **map** . Map means **do**  in simple English.

proclist = rdd.map(lambda x:x+2)  
print(proclist.collect())



**Filter()**

We have a list, iterate the list in such a way to filter elements greater than 2. For these type of operations we use **filter()**

d = [1,1,2,4,5,6,10]  
rdd1 = sc.parallelize(d)  
prcgrater = rdd1.filter(lambda x:x>2)  
print(prcgrater.collect())

for processing all the elements we use **map** Operation, for filtration we use filter()

**Class: 24 DATE: 17-08-2024**

**Agenda:**

* **Recap on Spark**
* **Recap on PySpark Handson**
* **Recap on Lambda Operations**
* **RDD String Processing(map, filter, flatmap)**
* **Usecase 1**
* **File processing**

Dataset Download Code:

import os, urllib.request, ssl;  
ssl\_context = ssl.\_create\_unverified\_context();  
[open(path, 'wb').write(urllib.request.urlopen(url, context=ssl\_context).read()) \  
 for url, path in { "https://github.com/saiadityaus1/test1/raw/main/df.csv": "data/dataset\_17082024/df.csv", \  
 "https://github.com/saiadityaus1/test1/raw/main/df1.csv": "data/dataset\_17082024/df1.csv", \  
 "https://github.com/saiadityaus1/test1/raw/main/dt.txt": "data/dataset\_17082024/dt.txt", \  
 "https://github.com/saiadityaus1/test1/raw/main/file1.txt": "data/dataset\_17082024/file1.txt", \  
 "https://github.com/saiadityaus1/test1/raw/main/file2.txt": "data/dataset\_17082024/file2.txt", \  
 "https://github.com/saiadityaus1/test1/raw/main/file3.txt": "data/dataset\_17082024/file3.txt", \  
 "https://github.com/saiadityaus1/test1/raw/main/file4.json": "data/dataset\_17082024/file4.json", \  
 "https://github.com/saiadityaus1/test1/raw/main/file5.parquet": "data/dataset\_17082024/file5.parquet", \  
 "https://github.com/saiadityaus1/test1/raw/main/file6": "data/dataset\_17082024/file6", \  
 "https://github.com/saiadityaus1/test1/raw/main/prod.csv": "data/dataset\_17082024/prod.csv", \  
 "https://github.com/saiadityaus1/test1/raw/main/state.txt": "data/dataset\_17082024/state.txt", \  
 "https://github.com/saiadityaus1/test1/raw/main/usdata.csv": "data/dataset\_17082024/usdata.csv"}.items()]

<https://gist.githubusercontent.com/saiadityaus1/ce9dc950f6434b5c3bc33716a88c3bbc/raw> 🡺 Generic code

**String Processing:**

Suppose we have list of strings [‘zeyobron’,’zeyo’,’analytics’] we have to concatenate each element with ‘ pvt’ as mentioned below op 🡺 [‘zeyobron pvt’,’zeyo pvt’,’analytics pvt’]

**lst = ['zeyobron','zeyo','analytics']  
rddlst = sc.parallelize((lst))  
  
pvrrdd = rddlst.map(lambda x:x+' pvt')  
print(pvrrdd.collect())**

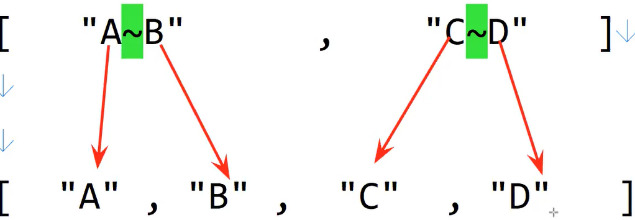
**Replace():** Iterate each element of list and replace ‘zeyo’ with ‘tera’

**reprdd = rddlst.map(lambda x:x.replace('zeyo','tera'))  
print(reprdd.collect())**

Iterate each element of list and filter element that contains ‘zeyo’. [‘zeyobron’,’zeyo’,’analytics’] 🡺 [‘zeyobron’,’zeyo’]

**filstr = rddlst.filter(lambda x: 'zeyo' in x)  
print(filstr.collect())**

**flatMap()**

We have string of lists as **[ ‘A~B’ , ’C~D’ ]** Expected op RDD is **[‘A’,’B’,’C’,’D’]**

Here map, filter operation doesn’t work, we need new operation that is called

as flatMap.

**lst = ['A~B' , 'C~D' ]  
rdd = sc.parallelize(lst) output  
print(rdd.collect())  
  
flatmap = rdd.flatMap(lambda x : x.split('~'))  
print(flatmap.collect())**

**Task:**

Liststr = ['zeyobron','zeyot','analyticszeyo'] 🡺 remove zeyo from each element

**Liststr = ['zeyobron','zeyot','analyticszeyo']  
rdd = sc.parallelize(Liststr)  
  
emptystr = rdd.map(lambda x:x.replace("zeyo",""))  
print(emptystr.collect())**

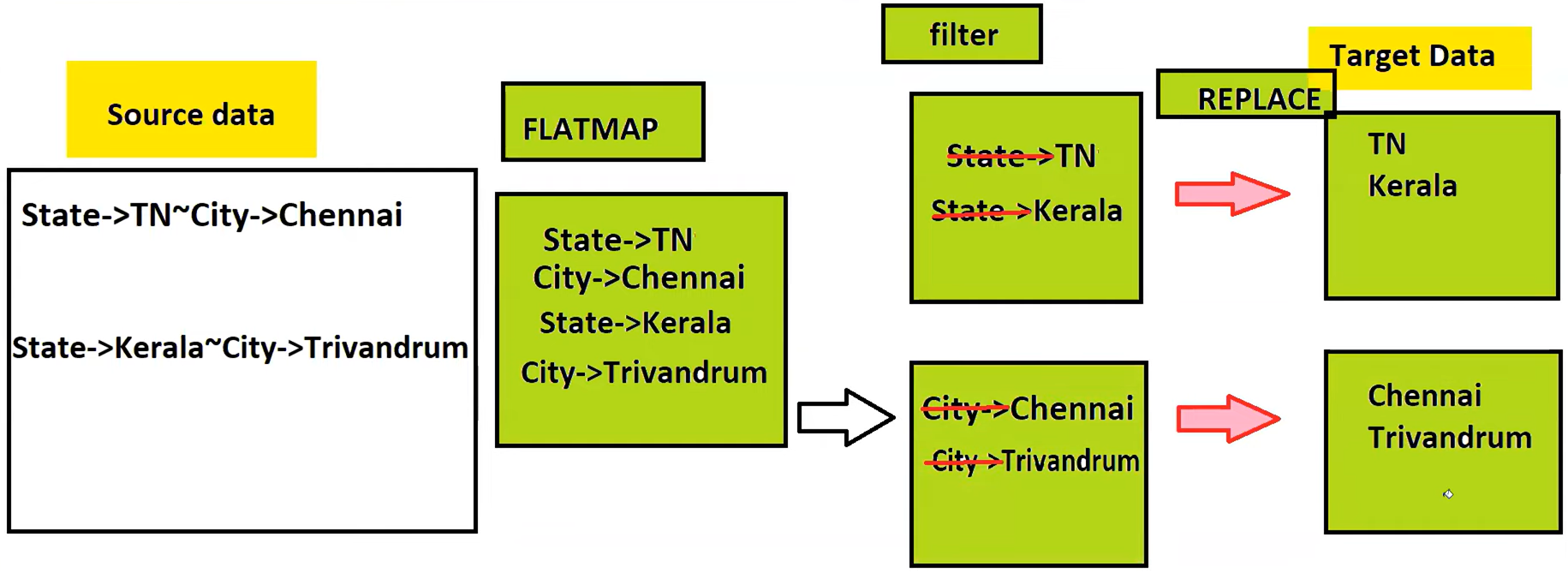
**Class: 25 DATE: 18-08-2024**

**Agenda:**

* **Recap on RDD operations**
* **Usecase**
* **File Reads**
* **Usdata.csv processing**
* **Column based processing**
* **Dataframe show**

**Use case:**

we have list as mentioned 🡺 [“State->TN~City->Chennai”,”State->Kerala~City->Trivandrum”] expected op is 2 lists, one list should contain all cities [“Chennai”,”Trivandrum”] and second list should contain all states [“TN”,“Kerala”].

****

lst = [“State->TN~City->Chennai”,”State->Kerala~City->Trivandrum”]

#list to Rdd conversion

rdd = sc.parallelize(lst)

**#flatmap**

flatdata = rdd.flatMap(lambda x: x.split(“->”)

#**filter state data and city data**

statedata = flatdata.filter(lambda x: ‘state’ in x)

citydata = flatdata.filter(lambda x:’city’ in x)

**#replace ‘state->’ and ‘city->’ with empty**

statedata = statedata.map(lambda x:x.replace(‘state->’,’’)

citydata = citydata.map(lambda x:x.replace(‘city->’,’’)



**File Reads**

**Boss Story:**

Boss 🡺 So far we are executing RDD operation on top of list. To process a list to converted that list to RDD. But now the requirement is different. You have file named **‘state.txt’**.

Sai 🡺 Boss I have checked it could see the same data in that file which I have processed as a list.

Boss 🡺 can you read this file and apply the same transformations.

#File Reads:

data = sc.textFile("data/dataset\_17082024/state.txt")  
print(data.collect())  
print(type(data.collect()))

When u read file using RDD, Rdd reads each row comes as single element in a list.

**Code:**

**data = sc.textFile("data/dataset\_17082024/state.txt")  
print(data.collect())  
print(type(data.collect()))  
  
flatdata = data.flatMap(lambda x:x.split(','))  
print(flatdata.collect())  
  
state = flatdata.filter(lambda x: 'State' in x )  
city = flatdata.filter(lambda x : 'City' in x )  
  
state\_data = state.map(lambda x:x.replace('State->',""))  
city\_data = city.map(lambda x:x.replace('City->',""))  
  
state\_data.foreach(print)  
city\_data.foreach(print)**

**==============================================================================================**

**Boss Story:** Sai we have different file to process now that is **usdata.csv.** Perform Below Transformations:

* Read this file first.
* Iterate each row filter rows length greater than 200.
* Flatten it with comma.
* Remove hyphens if any
* Concat zeyo to each element
* Print it

**Code:**

**rawdata = sc.textFile('data/dataset\_17082024/usdata.csv')  
  
#rawdata.foreach(print)  
  
length200 = rawdata.filter(lambda x:len(x)>200)  
#length200.foreach(print)  
#print(len(length200.collect()))  
  
flat = length200.flatMap(lambda x: x.split(','))  
#flat.foreach(print)  
  
removehyphen = flat.map(lambda x:x.replace("-",""))  
#removehyphen.foreach(print)  
  
concatzeyo = removehyphen.map(lambda x : x+',Zeyo')  
concatzeyo.foreach(print)**

Task1:

Use dt.txt and filter rows that contain ‘Gymanstics’

**Code:**

**rawdata = sc.textFile('data/dataset\_17082024/dt.txt')  
  
#rawdata.foreach(print)  
  
gymdata = rawdata.filter(lambda x:'Gymnastics' in x)  
  
gymdata.foreach(print)**

Task 2: Complete Python Youtube videos.

**Class: 26 DATE: 24-08-2024**

**Agenda:**

* **Recap on Spark**
* **Recap on File Processing**
* **Column Based Processing**
* **Dataframe Intro**
* **Dataframe File reads**
* **Sql Practise <HOME WORK>**

Writing to a file using RDD:

Data.saveAsTextFile(“file:///d:/finaldata/)

**Column Based Processing**

**Boss Story**

Bos🡺 Read dt.txt and filter rows that contain ‘Gymnastics’.

Sai 🡺 filtered, totally got 4 rows.

Boss 🡺 check if there is a way to filter rows with 5th column contains ‘gymnastics’ and Write the filtered data as parquet file.

Sai 🡺 Zaharia, can u help me on filtering whose 5th col contains ‘Gymnastics’.

Zaharia 🡺 its possible.., but u have to split the data.

**To achieve column filters it’s a 5 step process:**

1. We have to split the data.
2. Define column schema.
3. Impose the column schema to the split data.
4. Filter specific column.
5. Write the filtered data as parquet file.

**Code:**

data = sc.textFile("dt.txt")  
data.foreach(print)  
print(data.collect())/  
  
##split the data  
splitdata = data.map(lambda x:x.split(','))  
splitdata.foreach(print)  
  
splitdata.map(lambda x:x[0]).foreach(print)  
  
# define column schema, we need namedtuple.  
from collections import namedtuple  
schema = namedtuple('schema',['txnno','txndate','amount','category','product','spendby'])  
#print(schema)  
  
##Impose the schema  
schemardd = splitdata.map(lambda x : schema(x[0],x[1],x[2],x[3],x[4],x[5]))  
schemardd.foreach(print)  
print(schemardd.collect())  
  
## Product Filter:  
productfilter = schemardd.filter(lambda x: 'Gymnastics' in x.product)  
productfilter.foreach(print)  
  
## Convert RDD to Df  
df = productfilter.toDF()  
df.show()  
  
## write it as parquet  
df.write.parquet("file:///D:/parquetwrite")

**Boss story contd..**

Boss 🡺 Wite the final result as parquet and leave.

Sai 🡺 Zaharaia, can u help me.

Zaharia 🡺 it is possible. But Unfortunately RDD does not support write in parquet. For which we need to convert it to dataframe for parquet writes.

**Data Frames:**

Data Frames have capable of doing everything.

Suppose we have prod.csv, file4.json , file5.parquet. Read prod.csv and filter rows that have id>1, read file4.json and filter amount>100, read file5.parquet and filter spendby=’cash’.

We can read only text files with RDD. When our data is in Parequt, ORC, AVRO, RDD have restrictions, it cannot read this type of data formats. This is the place where we use of dataframe. It can make anything file format data into table like structure(dataframe) or it can make data structured.

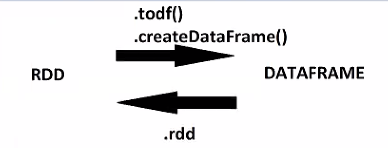
**SINGLE FORMULAE FOR DATA READS:** single formula used to read anytime of data

**CSV Reads:**

Read prod.csv can u find way to filter id greater than 1.

csvdata = spark.read.format('csv').option("header",'true').load("prod.csv")  
csvdata.show()  
  
csvdata.createOrReplaceTempView("chiru")  
  
procdf = spark.sql("select \* from chiru where id>1")  
procdf.show()

we can convert RDD to Dataframe and also dataframe to rdd.



SQL word document task 🡺 completed

**Class: 27 DATE: 25-08-2024**

**Agenda:**

* **Recap on Spark (till column-based processing)**
* **File Read and hands-on**
* **Dataframe DSL (Domain Specific Language) 🡺 Data Transformations**

**File Read and hands on**

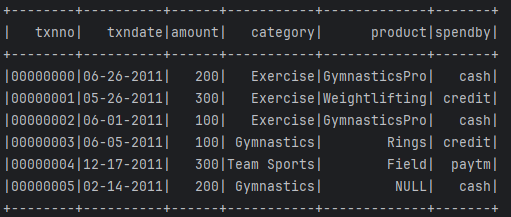
#CSV Reads  
csvdf = spark.read.format("csv").option("header","true").load("prod.csv")  
csvdf.show()  
  
csvdf.createOrReplaceTempView("csvview")  
spark.sql("select \* from csvview").show()  
  
#==================================================================================================  
#json Reads  
jsondf = spark.read.format("json").load("file4.json")  
  
jsondf.createOrReplaceTempView("jdf")  
  
filjdf = spark.sql("select \* from jdf where state='Washington' ")  
filjdf.show()  
#==================================================================================================  
#Parquet reads  
pdf = spark.read.load("file5.parquet")  
  
pdf.createOrReplaceTempView("pdft")  
  
procp = spark.sql("select category,cast(sum(amount) as int) as total\_amount from pdft group by category")  
procp.show()

**DSL(Domain Specific Language)**:

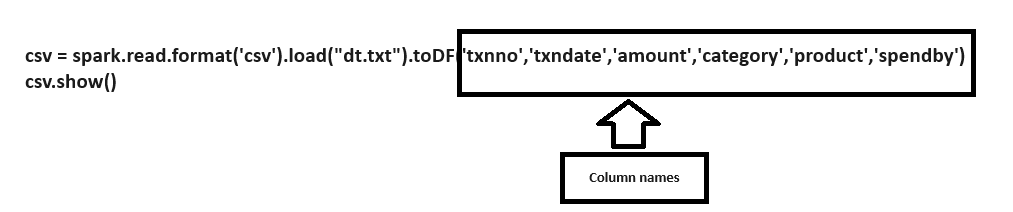
We have Created dataframe using rdd & df reads. But we have processed data using sql by creating a tempviews till now. There is another way of processing data that is **DSL.**



**Processing a Raw DataFrame:**

****

We can give columns while reading a file to .toDF method, as mentioned in below snip.



**Select method:** After reading above file, select only txndate, product columns

**procdf = csv.select("txndate,'product")**

**drop method:** drop txndate and product columns from raw dataframe.

**procdf = csv.drop("txndate","product")**

**Filters:**

* Single column filter
* Multi column filter
* Multi value filter
* In filter
* Like filter
* Null filter
* Not and not null filter

Single column filter: Filter above raw df with category = ‘Exercise’

**onefil = csv.filter(" category = 'Exercise' ")**

multi column filter with and: Filter above raw df with category = ‘Exercise’ and spendby=’cash’

**multifilter = csv.filter(" category='Exercise' and spendby='cash' ")**

multi column filter with or: Filter above raw df with category = ‘Exercise’ or spendby=’cash’

**multifilter \_filter = csv.filter(" category='Exercise’ or spendby='cash' ")**

multi value filter with or: Filter above raw df with category = ‘Exercise’ or ‘Gymnastics’

**multivalue = csv.filter(" category in ( 'Exercise' , 'Gymnastics' ) ")**

like filter : filter data frame product contains Gymnastics

**likefilter = csv.fliter(" product like '%Gymnastics%' ")**

null filter: filter product is null

**nullfilter = csv.filter(" product is null ")**

Not null filter : filter where product is not null

**notnullfilter = csv.filter(" product is not null ")**

multiple methods chained together

**singlefilter = csv.filter(" category='Exercise' ").drop("product").select("txnno","category")**

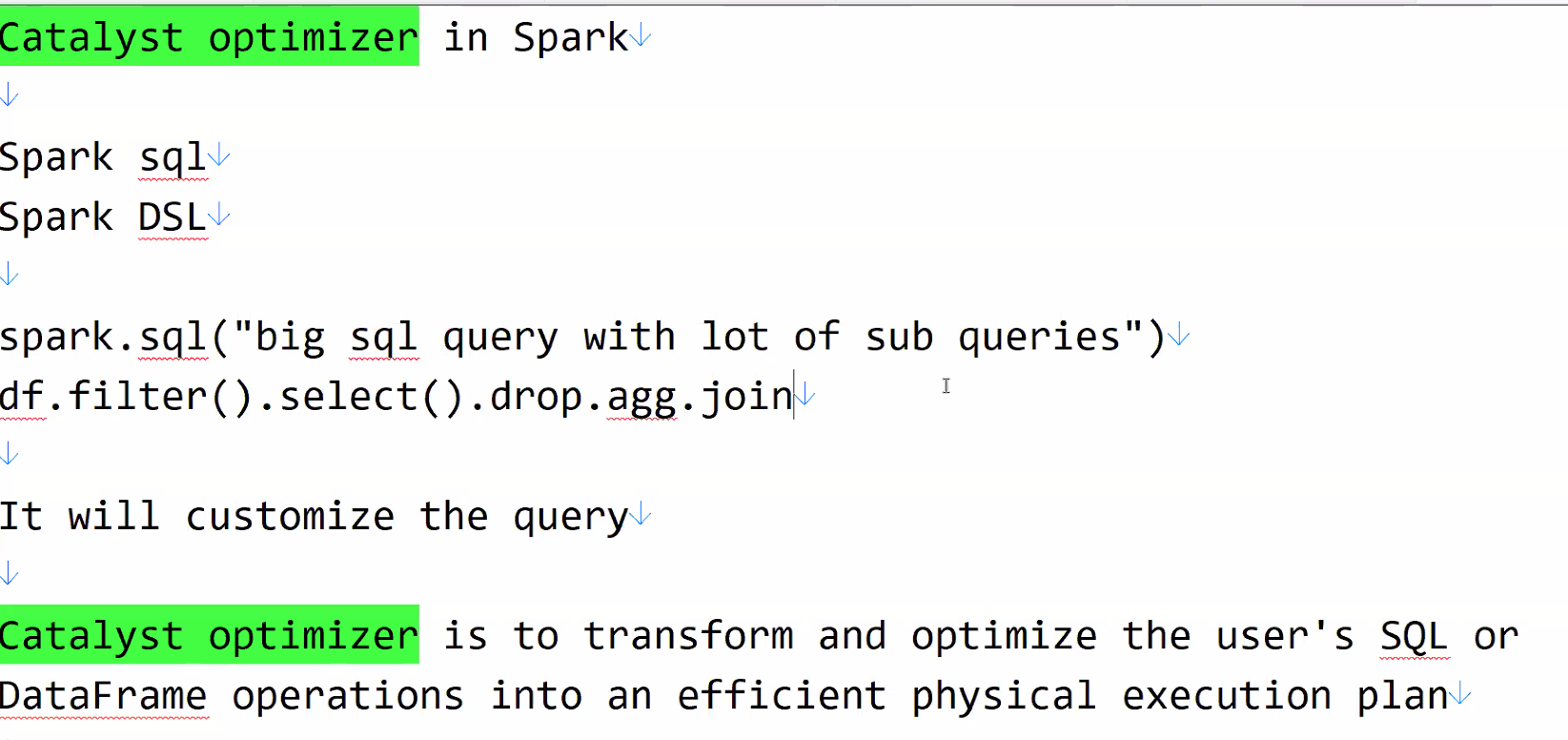
not Filter:

**notfil = csvdf.filter("product != 'Exercise'")**

data = [  
 ("00000000", "06-26-2011", 200, "Exercise", "GymnasticsPro", "cash"),  
 ("00000001", "05-26-2011", 300, "Exercise", "Weightlifting", "credit"),  
 ("00000002", "06-01-2011", 100, "Exercise", "GymnasticsPro", "cash"),  
 ("00000003", "06-05-2011", 100, "Gymnastics", "Rings", "credit"),  
 ("00000004", "12-17-2011", 300, "Team Sports", "Field", "paytm"),  
 ("00000005", "02-14-2011", 200, "Gymnastics", "", "cash")  
]  
  
# Create an RDD from the data  
rdd = spark.sparkContext.parallelize(data)  
  
# Convert RDD to DataFrame using toDF() and specify column names  
columns = ["txnno", "txndate", "amount", "category", "product", "spendby"]  
csvdf = rdd.toDF(columns)  
print()  
print("===== raw dataframe ===")  
print()  
  
csvdf.show()  
  
print()  
print("===== Category not equal to Exercise ===")  
print()  
  
notfilter = csvdf.filter(" category != 'Exercise' ")  
notfilter.show()  
  
print()  
print("===== category=Exercise and spendby=cash ===")  
print()  
  
notfil1 = csvdf.filter(" category='Exercise' and spendby !='cash' ")  
notfil1.show()

Catalyst Optimizer in Spark:

Catalyst optimizer is to transform and optimize the user’s SQL or Dataframe operation into an efficient physical Execution plan(Physical Execution is nothing but Memory optimization).



**Class: 28 DATE: 31-08-2024**

**Agenda:**

* **DSL Recap**
* **Expressions**
* **Column Processing**
* **Joins**
* **Scenario**

**Column Processing:**

**selectExpr**

**#Single columns select expression**

**procdf = csvdf.selectExpr("txnno", "txndate","amount+100 as amount", "category", "product", "spendby")**

Make category column to upper case(Multi column select Expression) **procdf = (**

**csvdf.selectExpr(**

**"txnno", # column select**

**"txndate", # column select**

**"amount+100 as amount", # column Expression**

**"upper(category) as category", # column Expression**

**"lower(product) as product", # column select**

**"spendby")**

**)**

**Multiple Case when**

we have dt.txt data, add extra column as **STATUS,** the status column data will be ifspendby=cash then status is 0 else status 1

**procdf = (**

**csvdf.selectExpr(**

**"txnno", # column select**

**"txndate", # column select**

**"amount+100 as amount", # column Expression**

**"upper(category) as category", # column Expression**

**"lower(product) as product", # column select**

**"spendby",**

**“case when spendby=’cash’ then 0 else 1 end as status”)**

**)**

If Cash =0, paytm=2 else 1:

“””Case when spendby=’cash’ then 0

when spendby=’paytm’ then 2

else 1 end as status”””

**split() and cast()**

rename txndate to year and year column should consist of year. No month and date.

**procdf = (**

**csvdf.selectExpr(**

**"cast(txnno as int) as txnno", # column select**

**"split(txndate,’-‘)[2] as year", # column select**

**"amount+100 as amount", # column Expression**

**"upper(category) as category", # column Expression**

**"lower(product) as product", # column select**

**"spendby",**

**“case when spendby=’cash’ then 0 else 1 end as status”)**

**)**

**To print the schema of DataFrame:**

**Procdf.printSchema()**

Task 1: take category col and concat it with ‘~zeyo’ then create a new col with con name

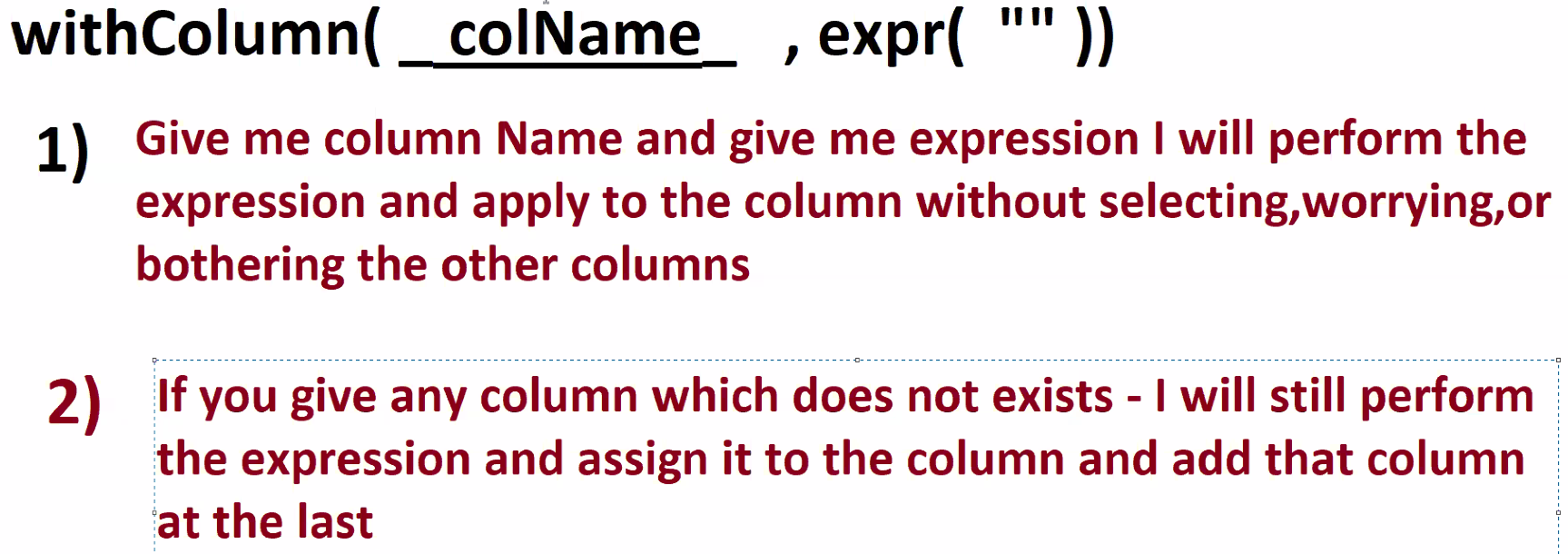
**Class: 28 DATE: 31-08-2024**

**Agenda:**

* **Recap on Expressions**
* **Expressions with withColumn**
* **Joins**
* **Window Scenarios**
* **Scenarios**

**withColumn:**

Just to process single column, we need to select all columns with selectExpr. When we have huge columns say 1000 cols, just to process single column, unnecessarily process all the columns in selectExpr. This can be resolved with **withColumn.**



**procdf = (  
 csvdf.withColumn("category",expr("upper(category)"))  
 .withColumn("amount",expr("amount+100"))  
 .withColumn("txnno",expr("cast(txnno as int)"))  
 .withColumn("txndate",expr("split(txndate,'-')[2]"))  
 .withColumn("product",expr("lower(product)"))  
 .withColumn("spendby",expr("spendby"))  
 .withColumn("status",expr("case when spendby ='cash' then 0 else 1 end as status"))  
)**

**withColumns:**

**procdf = (  
 csvdf.withColumns({**

**“txnno”:expr(“cast(txnno as int”),**

**“year”:expr(”split(txndate,’-‘)[2]”)**

**})**

**Fetch year from txndate ans rename column as year:**

**Procdf = csvdf.withColumn(“txndate”,expr(“split(txndate,’-‘)[2]”))**

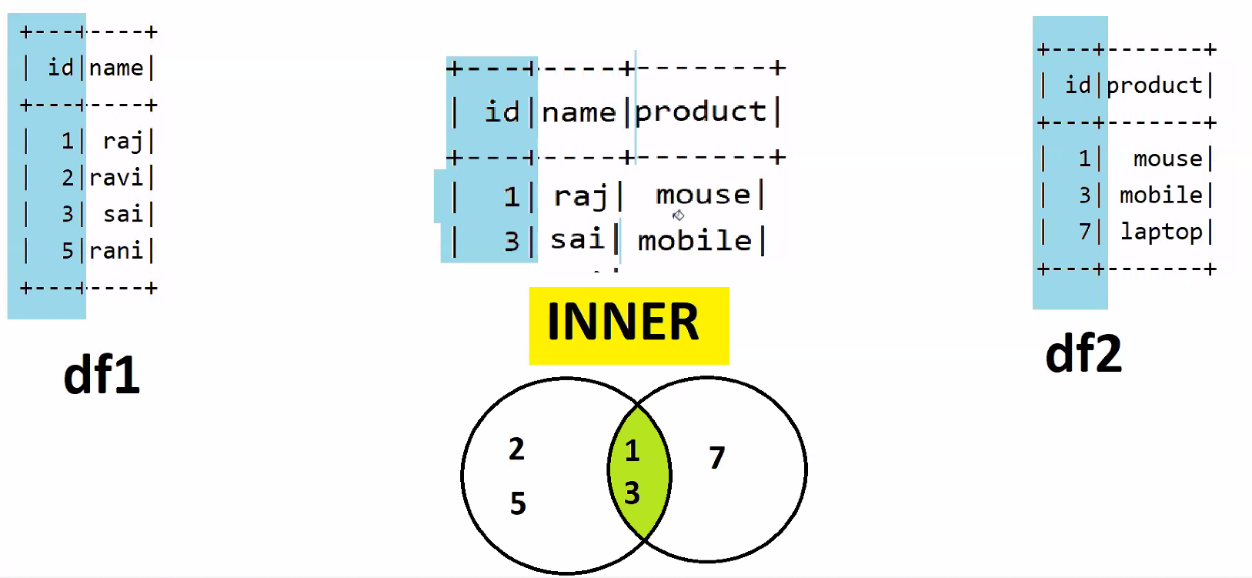
**.withColumnenamed(“txndate”,”year”))**

**Joins**

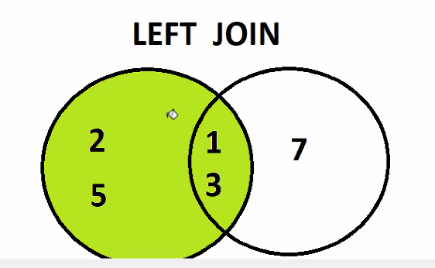
**Inner join:**

**Requirement 1**: join custdf and proddf, get the name from left table and get product from right table for common ids.

**inner = custdf.join(proddf,['id'],'inner')**

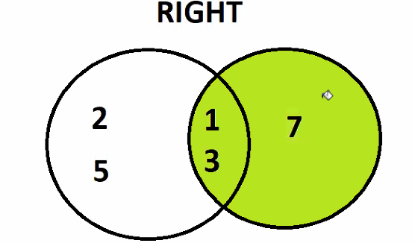


**Left Join:**

**Requirement 2:** Join these tables, all id from left table and its respective products from right table. (NO LOSS IN THE LEFT TABLE)

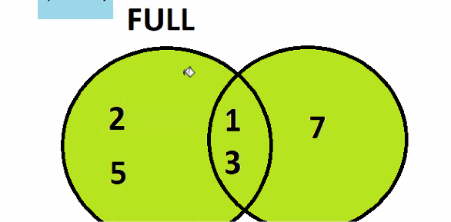
**left = custdf.join(proddf,['id'],'left')**

**Right Join:**

**Requirement 3:** Join these tables, all id from right table and its respective joins from left table. (NO LOSS IN THE RIGHT TABLE)

**right = custdf.join(proddf,['id'],’right’)**

**Full Outer Join:**

**Requirement 4:**  join these two tables, get all the ids from left table and all the ids from right table. No loss from left and right tables.

**full = custdf.join(proddf,['id'],'full')**

suppose, if we have table column with 2 different names and want to use for join condition:

**full = custdf.join( proddf, custdf.id==proddf.custid , 'full').drop(“custid”)**

Task 1: Mapside join in Hive which is equivalent to Broadcast join in Spark. Explore more on this

https://medium.com/@ashwin\_kumar\_/broadcast-join-in-spark-e128a7c21b94#:~:text=Broadcast%20Join%20is%20an%20optimization,ultimately%20saving%20costs%20for%20users.

Task2: Scenario1

**Class: 29 DATE: 07-09-2024**

**Agenda:**

* **Recap on DSL**
* **Joins Recap**
* **Anti Join and Cross join**
* **Aggregations**
* **Scenarios**
* **Revision**

**Anti Join:**

**df1.join(df2 df1.id==df2.custid, “left\_anti”).show()**

Tip : Interview Experience:

Whenever we join any big data project, execute the entire project once end to end from first job to last job. We run totally 55 Spark Jobs. When running entire 55 jobs, there are 5 or 6 jobs which take 2 hours to 6 hours long. Totally 30 to 48 hours for all 55 jobs. In spark UI, we can find how much it takes for each line takes to get executed.

Dig the code for 5 to 6 hours job, each job is of 2000 lines of code. One strange issue I experienced was, my requirement was whatever the ids we have in the right table should be removed from the left table. Teammate has **applied list filtering** that’s why its taking much time to get executed.

**Non optimized Code with list filtering:**

Listvalue = df2.Select(“id”).rdd.flatmap(lambda x:x)

Filterdf = df1.filter( ~ df1[‘id’].isin(listvalue))

This way of problem solving creates a **Data Skewness.** The above data skewness problem can be resolved with the help of Anti joins. After replacing list filtering with Anti joins, the entire job execution time came from 6 hours to 1 hr 40mins.

We don’t have as such right\_anti, if u wan t to perform anything like just replace df1 and df2.

**Cross Join:**

Example: when we have plans that to be applied to all customers then we use cross join.

**Code:**

**rdd1 = spark.sparkContext.parallelize([  
 (1, 'raj'),  
 (2, 'ravi'),  
 (3, 'sai'),  
 (5, 'rani')  
],1)  
  
rdd2 = spark.sparkContext.parallelize([  
 (1, 'mouse'),  
 (3, 'mobile'),  
 (7, 'laptop')  
],1)  
  
df1 = rdd1.toDF(['id', 'name']).coalesce(1)  
df2 = rdd2.toDF(['id', 'product']).coalesce(1)  
  
# Show the DataFrames  
df1.show()  
df2.show()  
listvalue= df2.select("id").rdd.flatMap(lambda x : x).collect()  
  
print(listvalue)  
  
filterdf = df1.filter( ~ df1['id'].isin(listvalue))  
  
filterdf.show()  
  
antijoin=df1.join(df2, ["id"] ,"left\_anti")  
  
antijoin.show()  
  
crossj=df1.crossJoin(df2)  
crossj.show()**

**Task 1:**

**What is difference between Left anti join and left semi join?**

**Left Semi Join:**

* Purpose: Returns rows from the left DataFrame where there is a match in the right DataFrame based on the specified join condition.
* Behavior: It works like a filter, where only the rows from the left DataFrame that have a match in the right DataFrame are returned, but only columns from the left DataFrame are included.

**Left Anti Join:**

* **Purpose:** Returns rows from the left DataFrame where there is **no match** in the right DataFrame based on the specified join condition.
* **Behavior:** It filters out rows from the left DataFrame that have matches in the right DataFrame, returning only the rows that do not match.

**Task 2:**

**Read udata.csv, filter state=’LA’, add column tdate and value should be todays date?**

**df = spark.read.format("csv").option("header","true").load("usdata.csv")  
df.show()  
  
filterla = df.filter( "state == 'LA'" )  
#filterla.show()  
  
filterla.withColumn("tdate",current\_date()).show()**

**Class: 30 DATE: 08-09-2024**

**Agenda:**

* **Recap on Joins**
* **Aggregations**
* **Windowing Functions**
* **Spark Revision**

**Aggregations:**

**rdd1 = spark.sparkContext.parallelize([  
 ("sai", 10),  
 ("zeyo", 20),  
 ("sai", 15),  
 ("zeyo", 10 ),  
 ("sai", 10 ),  
],1)  
df = rdd1.toDF(['name', 'amt']).coalesce(1)  
df.show()  
df.groupBy("name").agg(sum("amt").alias("total"),   
 count("name").alias("cnt"),  
 collect\_list("amt").alias("col\_collect"),  
 collect\_set("amt").alias("col\_set")  
 ).show()**

**Production issue:** use of collect\_set instead of collect\_list, because collect\_list cell can allow maximum of 2GB of data else it throws **bytebuffer** error. (Search online databricks forum)

**Windowing Functions:**

**Find the second highest salary? 🡺 most common interview question asked.**

**Step 1 🡺 partition it.**

**Step 2🡺 Order by Desc.**

**Step 3 🡺Dense Rank**

**Step4🡺 Filter rank =2**

**Step 5🡺 drop rank column**

**data = [("DEPT3", 500),**

**("DEPT3", 200),**

**("DEPT1", 1000),**

**("DEPT1", 700),**

**("DEPT1", 500),**

**("DEPT2", 400),**

**("DEPT2", 200)]**

**columns = ["dept", "salary"]**

**df = spark.createDataFrame(data, columns)**

**df.show()**

**from pyspark.sql.window import Window**

**deptwindow = Window.partitionBy("dept").orderBy(col("salary").desc())**

**dfrank = df.withColumn("drank", dense\_rank().over(deptwindow))**

**dfrank.show()**

**filterdf= dfrank.filter("drank=2")**

**filterdf.show()**

**finaldf = filterdf.drop("drank")**

**finaldf.show()**

Tasks:

Task 1: Revision task

Task 2: python tutorials.

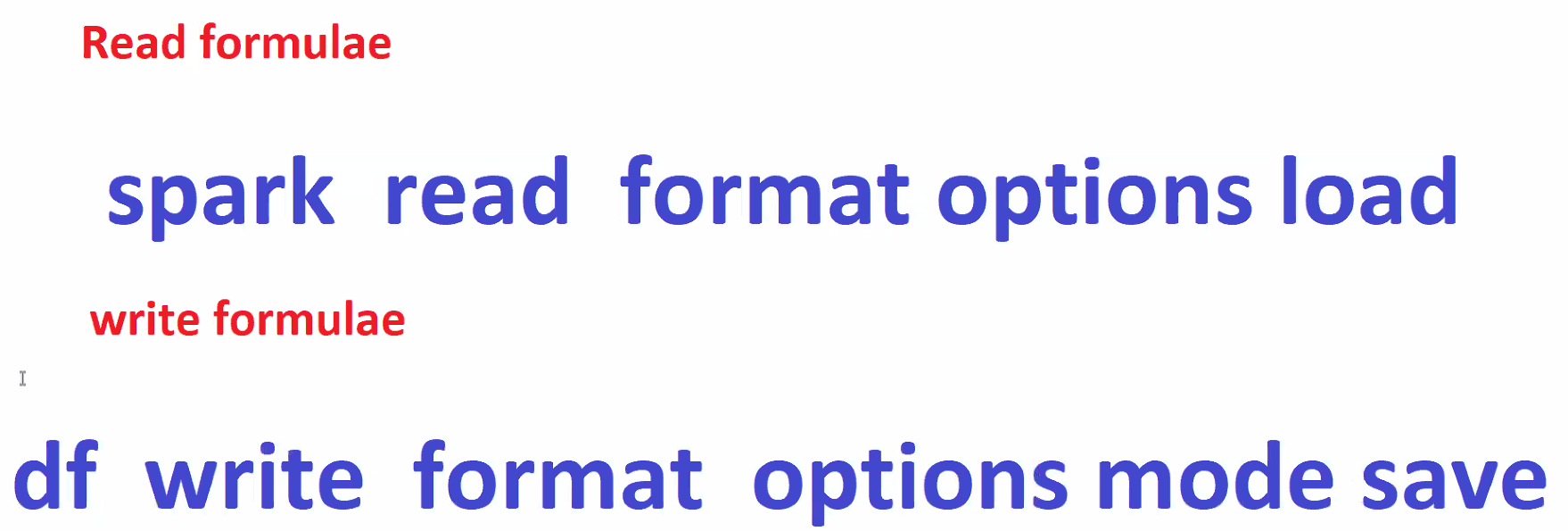
**Class: 31 DATE: 14-09-2024**

**Agenda:**

* **DataFrame Writes**
* **Complex data Processing**

**DataFrame Writes**

**Wite Formulae:**



**Complex data Processing**

Most of the companies these days dealing with complex data.

Features of Complex data:

* 90% of real time projects sue complex data.
* Paraquet, Avro, Json, xml supports complex data.
* Complex data solves many real time problems.
* Complex data helps for easy Design.

Complex data is nothing but Hierarchical data (nested data : data inside the data).

**Possible File formats that supports Complex Big Data:**



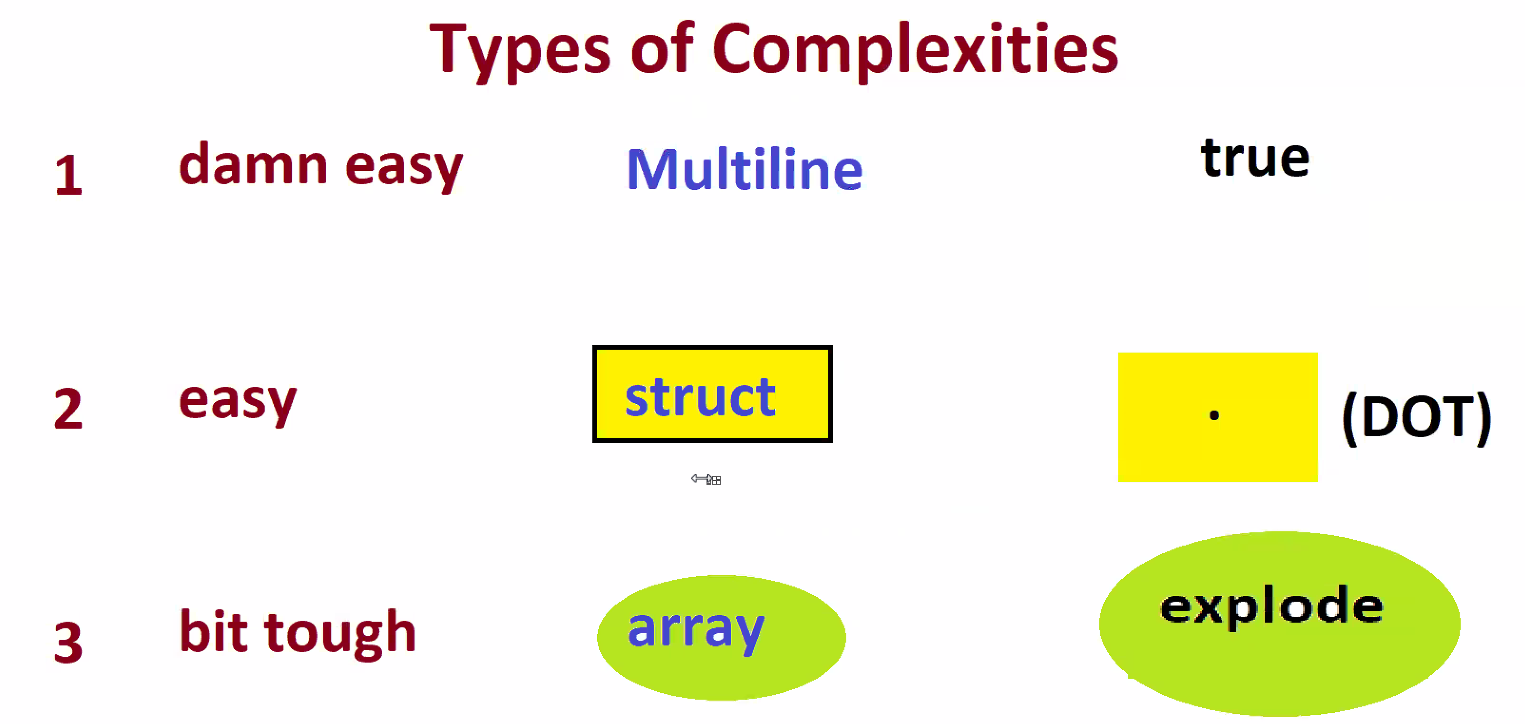
**Class: 32 DATE: 15-09-2024**

**Agenda:**

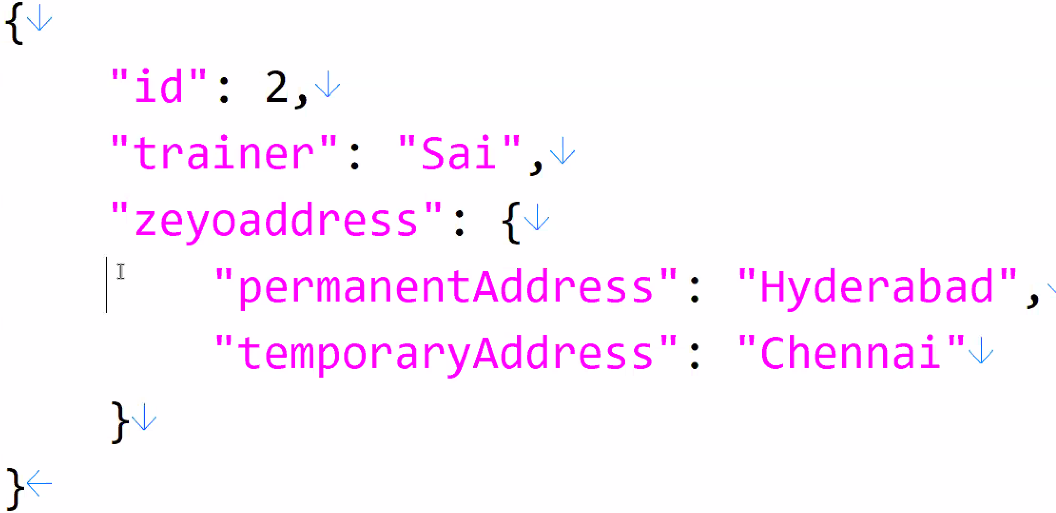
* **Complex data Processing**

Whenever dealing with complex data, print its schema using **printSchema()** method.

**Types of Complexities:**

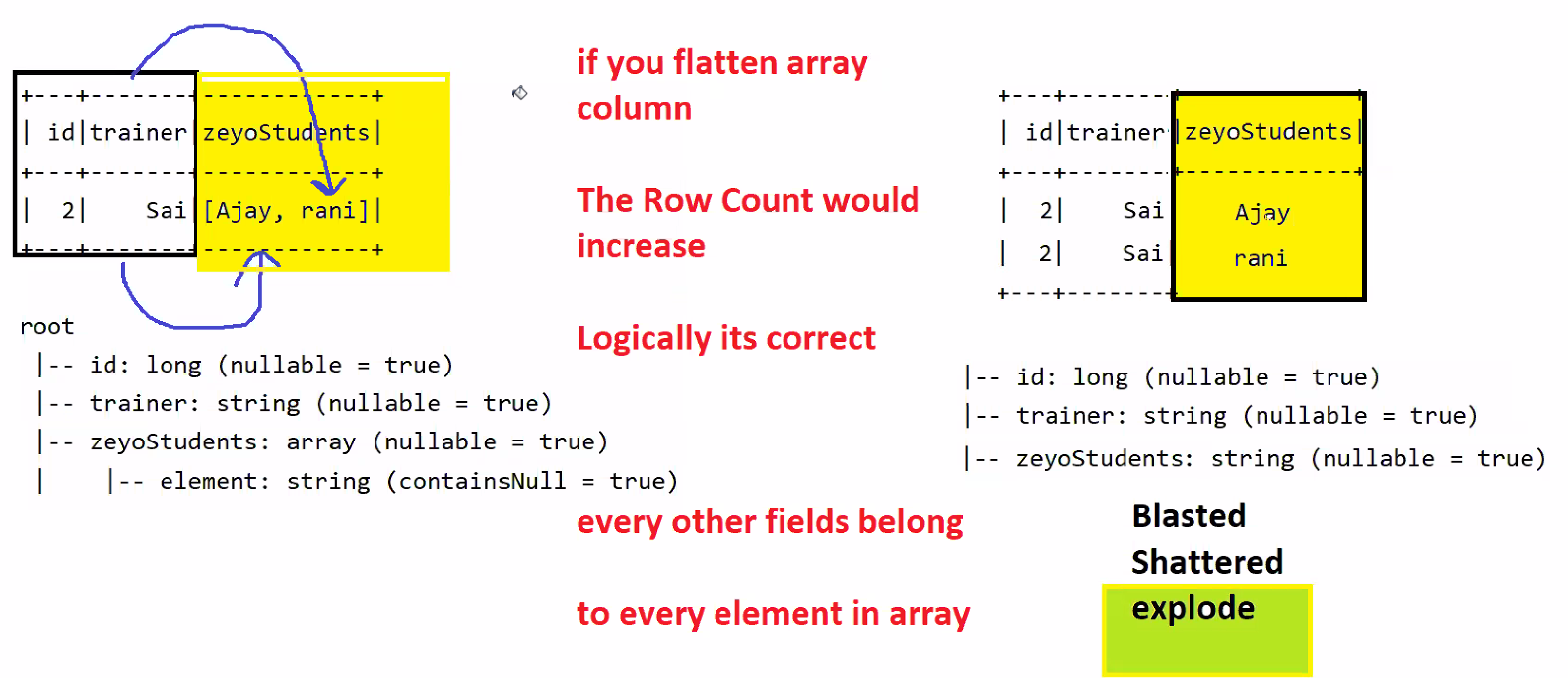
****

1. **Multi-Line (Damn Easy)**
2. **Struct (Easy )**

****

**data= """  
{  
 "id": 2,  
 "trainer": "Sai",  
 "zeyoaddress": {  
 "user":{  
 "permanentAddress": "Hyderabad",  
 "temporaryAddress": "Chennai"  
 }  
 }  
}  
"""  
df = spark.read.json(sc.parallelize([data],1))  
df.show()  
df.printSchema()  
  
falttendf = df.selectExpr(  
 "id","trainer","zeyoaddress.user.permanentAddress","zeyoaddress.user.temporaryAddress"  
)  
falttendf.show()  
falttendf.printSchema()**

1. **Array (Bit tough)**

****

**data= """  
  
{  
 "id": 2,  
 "trainer": "Sai",  
 "zeyoaddress": {  
 "permanentAddress": "Hyderabad",  
 "temporaryAddress": "Chennai"  
 },  
 "zeyoStudents": [  
 "Ajay",  
 "rani"  
 ]  
}  
"""  
df = spark.read.json(sc.parallelize([data],1))  
df.show()  
df.printSchema()  
  
flattendf = df.selectExpr(  
 "id",  
 "trainer",  
 "zeyoaddress.permanentAddress",  
 "zeyoaddress.temporaryAddress",  
 "explode(zeyoStudents) as zeyoStudents"  
)  
  
flattendf.show()  
flattendf.printSchema()**

**Class: 33 DATE: 21-09-2024**

**Agenda:**

* **Recap on Complex data Processing type 2, type 3**
* **Deep Type 3**
* **API Data Reads**
* **AWS cloud Sessions**

**Deep Type 3**

**Actors Select Expr Code**

data= """

{

"country" : "US",

"version" : "0.6",

"Actors": [

{

"name": "Tom Cruise",

"age": 56,

"BornAt": "Syracuse, NY",

"Birthdate": "July 3, 1962",

"photo": "https://jsonformatter.org/img/tom-cruise.jpg",

"wife": null,

"weight": 67.5,

"hasChildren": true,

"hasGreyHair": false,

"picture": {

"large": "https://randomuser.me/api/portraits/men/73.jpg",

"medium": "https://randomuser.me/api/portraits/med/men/73.jpg",

"thumbnail": "https://randomuser.me/api/portraits/thumb/men/73.jpg"

}

},

{

"name": "Robert Downey Jr.",

"age": 53,

"BornAt": "New York City, NY",

"Birthdate": "April 4, 1965",

"photo": "https://jsonformatter.org/img/Robert-Downey-Jr.jpg",

"wife": "Susan Downey",

"weight": 77.1,

"hasChildren": true,

"hasGreyHair": false,

"picture": {

"large": "https://randomuser.me/api/portraits/men/78.jpg",

"medium": "https://randomuser.me/api/portraits/med/men/78.jpg",

"thumbnail": "https://randomuser.me/api/portraits/thumb/men/78.jpg"

}

}

]

}

"""

df = spark.read.json(sc.parallelize([data],1))

df.show()

df.printSchema()

flatten1 = df.selectExpr(

"explode(Actors) as Actors",

"country",

"version"

)

flatten1.show()

flatten1.printSchema()

flatten2=flatten1.selectExpr(

"Actors.Birthdate",

"Actors.BornAt",

"Actors.age",

"Actors.hasChildren",

"Actors.hasGreyHair",

"Actors.name",

"Actors.photo",

"Actors.picture.large",

"Actors.picture.medium",

"Actors.picture.thumbnail",

"Actors.weight",

"Actors.wife",

"country",

"version"

)

flatten2.show()

flatten2.printSchema()

**withColumn**

df = spark.read.json(sc.parallelize([data],1))

df.show()

df.printSchema()

from pyspark.sql.functions import \*

flatten1 = (

df.withColumn(

"Actors",

expr("explode(Actors)")

)

)

flatten1.show()

flatten1.printSchema()

flatten2 = (

flatten1.withColumn("Birthdate",expr("Actors.Birthdate"))

.withColumn("BornAt",expr("Actors.BornAt"))

.withColumn("age",expr("Actors.age"))

.withColumn("hasChildren",expr("Actors.hasChildren"))

.withColumn("hasGreyHair",expr("Actors.hasGreyHair"))

.withColumn("name",expr("Actors.name"))

.withColumn("photo",expr("Actors.photo"))

.withColumn("large",expr("Actors.picture.large"))

.withColumn("medium",expr("Actors.picture.medium"))

.withColumn("thumbnail",expr("Actors.picture.thumbnail"))

.withColumn("weight",expr("Actors.weight"))

.withColumn("wife",expr("Actors.wife"))

).drop("Actors")

flatten2.show()

flatten2.printSchema()

**WithColumn and SelectExpr**

df = spark.read.json(sc.parallelize([data],1))

df.show()

df.printSchema()

from pyspark.sql.functions import \*

flatten1 = (

df.withColumn(

"Actors",

expr("explode(Actors)")

)

)

flatten1.show()

flatten1.printSchema()

flatten2 = flatten1.selectExpr(

"Actors.Birthdate",

"Actors.BornAt",

"Actors.age",

"Actors.hasChildren",

"Actors.hasGreyHair",

"Actors.name",

"Actors.photo",

"Actors.picture.large",

"Actors.picture.medium",

"Actors.picture.thumbnail",

"Actors.weight",

"Actors.wife",

"country",

"version")

flatten2.show()

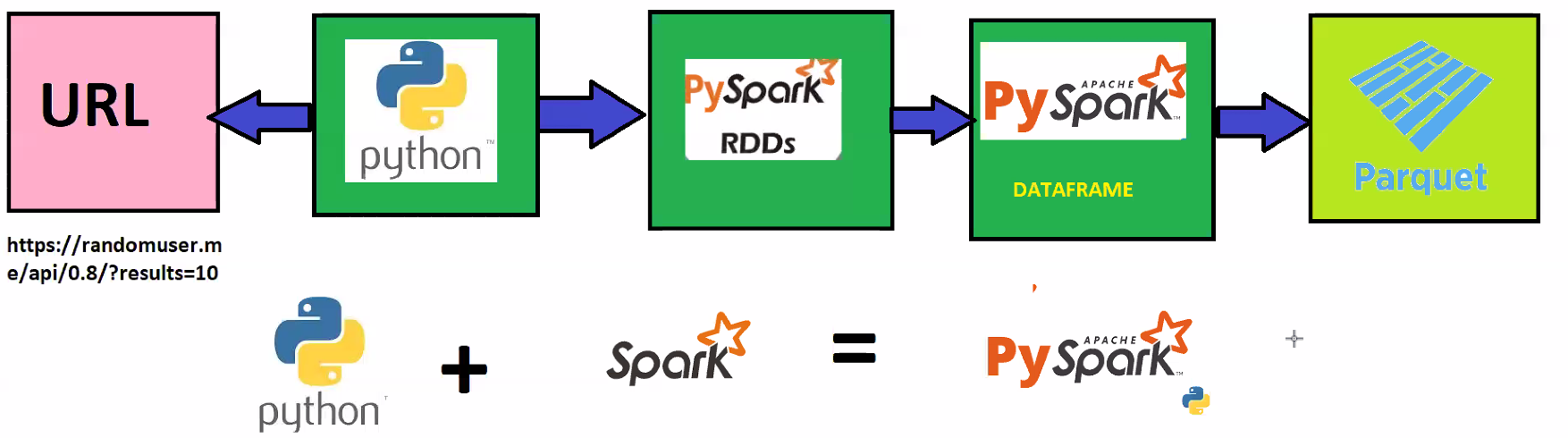
flatten2.printSchema()

**API Data Reads**

Consume APIs, process it and write the data as parquet file.

**Steps to Achieve:**

* Use python to read API Endpoints.
* Convert the data to PySpark RDD.
* Convert RDD to Dataframe.



CODE:

import os

import urllib.request

import ssl

urldata=(

urllib

.request

.urlopen("https://randomuser.me/api/0.8/?results=10",context=ssl.\_create\_unverified\_context())

.read()

.decode('utf-8')

)

print(urldata)

rdd = sc.parallelize([urldata],1)

df = spark.read.json(rdd)

df.show()

df.printSchema()

from pyspark.sql.functions import \*

flatten1= df.withColumn("results",expr("explode(results)"))

flatten1.show()

flatten1.printSchema()

flatten2= flatten1.selectExpr(

"nationality",

"results.user.cell",

"results.user.dob",

"results.user.email",

"results.user.gender",

"results.user.location.city",

"results.user.location.state",

"results.user.location.street",

"results.user.location.zip",

"results.user.md5",

"results.user.name.first",

"results.user.name.last",

"results.user.name.title",

"results.user.password",

"results.user.phone",

"results.user.picture.large",

"results.user.picture.medium",

"results.user.picture.thumbnail",

"results.user.registered",

"results.user.salt",

"results.user.sha1",

"results.user.sha256",

"results.user.username",

"seed",

"version")

flatten2.show()

flatten2.printSchema()