Spark is a engine which will work on large data sets in a distributed systems. It uses in-memory (RAM) for processing the data and will get o/p fast. It has built-in modules Spark SQL, Streaming, Py-Spark, MLib, Graph X.... For Data Scientists It will be helpful for Exploratory Data Analysis (Visualizing), Feature Extraction (Consolidating Large Data Sets to Minimum Useful Information Data Sets) and Machine Learning.

Creating the connection is as simple as creating an instance of the SparkContext class. The class constructor takes a few optional arguments that allow you to specify the attributes of the cluster you're

connecting to. An object holding all these attributes can be created with the SparkConf() constructor.

**Using DataFrames**

Spark's core data structure is the Resilient Distributed Dataset (RDD). This is a low level object that lets Spark

work its magic by splitting data across multiple nodes in the cluster. However, RDDs are hard to work with

directly, so in this course you'll be using the Spark DataFrame abstraction built on top of RDDs.

The Spark DataFrame was designed to behave a lot like a SQL table (a table with variables in the columns and observations in the rows). Not only are they easier to understand, DataFrames are also more optimized for dcomplicated operations than RDDs.

When you start modifying and combining columns and rows of data, there are many ways to arrive at the same result, but some often take much longer than others. When using RDDs, it's up to the data scientist to figure out the right way to optimize the query, but the DataFrame implementation has much of this optimization built in!

To start working with Spark DataFrames, you first have to create a SparkSession object from your SparkContext. You can think of the SparkContext as your connection to the cluster and the SparkSession as your interface with that connection.

Remember, for the rest of this course you'll have a SparkSession called spark available in your workspace!

Q)Which of the following is an advantage of Spark DataFrames over RDDs?

A)Operations using DataFrames are automatically optimized.

Your SparkSession has an attribute called catalog which lists all the data inside the cluster. Thisattributeas a few methods for extracting different pieces of information.One of the most useful is the .listTables() method, which returns the names of all the tables in your cluster as a list.

spark.catalog.listTables()

* Use the .sql() method to get the first 10 rows of the flights table and save the result to flights10. The variable query contains the appropriate SQL query.
* Use the DataFrame method .show() to print flights10.

# Don't change this query

query ="FROM flights SELECT \* LIMIT 10"

# Get the first 10 rows of f

lights

flights10 = spark.sql(query)

# Show the results

flights10.show()

Sometimes it makes sense to then take that table and work with it locally using a tool like pandas. Spark DataFrames make that easy with the .toPandas() method. Calling this method on a Spark DataFrame returns the corresponding pandas DataFrame. It's as simple as that!

This time the query counts the number of flights to each airport from SEA and PDX.

Remember, there's already a SparkSession called spark in your workspace!

# Don't change this query

query = "SELECT origin, dest, COUNT(\*) as N FROM flights GROUP BY origin, dest"

# Run the query

flight\_counts = spark.sql(query)

# Convert the results to a pandas DataFrame

pd\_counts = flight\_counts.toPandas()

# Print the head of pd\_counts

print(pd\_counts.head())

In the last exercise, you saw how to move data from Spark to pandas. However, maybe you want to go the

other direction, and put a pandas DataFrame into a Spark cluster! The SparkSession class has a method for this as well.

The .createDataFrame() method takes a pandas DataFrame and returns a Spark DataFrame.

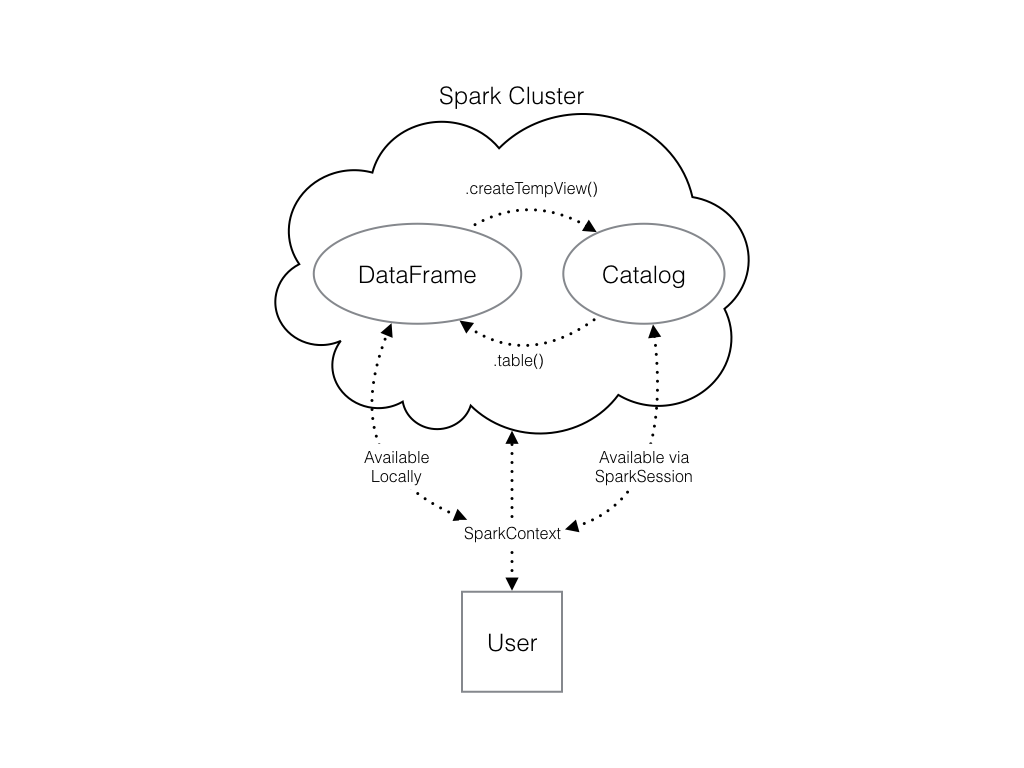
The output of this method is stored locally, not in the SparkSession catalog. This means that you can use all the Spark DataFrame methods on it, but you can't access the data in other contexts.

For example, a SQL query (using the .sql() method) that references your DataFrame will throw an error. To access the data in this way, you have to save it as a temporary table.

You can do this using the .createTempView() Spark DataFrame method, which takes as its only argument the name of the temporary table you'd like to register. This method registers the DataFrame as a table in the catalog, but as this table is temporary, it can only be accessed from the specific SparkSession used to create the Spark DataFrame.

There is also the method .createOrReplaceTempView(). This safely creates a new temporary table if nothing was there before, or updates an existing table if one was already defined. You'll use this method to avoid running into problems with duplicate tables.

Check out the diagram to see all the different ways your Spark data structures interact with each other.



There's already a SparkSession called spark in your workspace, numpy has been imported as np, and pandas as pd.

# Create pd\_temp

pd\_temp = pd.DataFrame(np.random.random(10))

# Create spark\_temp from pd\_temp

spark\_temp = spark.createDataFrame(pd\_temp)

# Examine the tables in the catalog

print(spark.catalog.listTables())

# Add spark\_temp to the catalog

spark\_temp.createOrReplaceTempView('temp')

# Examine the tables in the catalog again

print(spark.catalog.listTables())

Now you know how to put data into Spark via pandas, but you're probably wondering why deal

with pandas at all? Wouldn't it be easier to just read a text file straight into Spark? Of course it would!

Luckily, your SparkSession has a .read attribute which has several methods for reading different data sources into Spark DataFrames. Using these you can create a DataFrame from a .csv file just like with regular pandas DataFrames!

The variable file\_path is a string with the path to the file airports.csv. This file contains information about different airports all over the world.

A SparkSession named spark is available in your workspace.

# Don't change this file path

file\_path = "/usr/local/share/datasets/airports.csv"

# Read in the airports data

airports = spark.read.csv(file\_path,header=True)

# Show the data

airports.show()

------------------------------------------------------------------------------------------------------------------------------------------

Yarn will allocate you resources like ram for execution of the program. So when you submit the program driver will send the program to all blocks which contain data and executers(Resource) in the nodes will execute program on the data and all the o/p will comes to the driver, In the driver we can write the code where we want to store the data. **Driver – App Master.**

**Manipulating Data**

In this chapter, you'll learn how to use the methods defined by Spark's DataFrame class to perform common data operations.

Let's look at performing column-wise operations. In Spark you can do this using the .withColumn() method, which takes two arguments. First, a string with the name of your new column, and second the new column itself.

The new column must be an object of class Column. Creating one of these is as easy as extracting a column from your DataFrame using df.colName.

Updating a Spark DataFrame is somewhat different than working in pandas because the Spark DataFrame is *immutable*. This means that it can't be changed, and so columns can't be updated in place.

Thus, all these methods return a new DataFrame. To overwrite the original DataFrame you must reassign the returned DataFrame using the method like so:

df = df.withColumn("newCol", df.oldCol + 1)

The above code creates a DataFrame with the same columns as df plus a new column, newCol, where every entry is equal to the corresponding entry from oldCol, plus one.

To overwrite an existing column, just pass the name of the column as the first argument!

# Create the DataFrame flights

flights = spark.table("flights")

# Show the head

flights.show()

# Add duration\_hrs

flights = flights.withColumn('duration\_hrs',flights.air\_time/60)

A SQL query returns a table derived from one or more tables contained in a database.

Every SQL query is made up of commands that tell the database what you want to do with the data. The two commands that every query has to contain are SELECT and FROM.

The SELECT command is followed by the columns you want in the resulting table.

The FROM command is followed by the name of the table that contains those columns. The minimal SQL query is:

SELECT \* FROM my\_table;

The \* selects all columns, so this returns the entire table named my\_table.

Similar to .withColumn(), you can do column-wise computations within a SELECT statement. For example,

SELECT origin, dest, air\_time / 60 FROM flights;

returns a table with the origin, destination, and duration in hours for each flight.

Another commonly used command is WHERE. This command filters the rows of the table based on some logical condition you specify. The resulting table contains the rows where your condition is true. For example, if you had a table of students and grades you could do:

SELECT \* FROM students

WHERE grade = 'A';

to select all the columns and the rows containing information about students who got As.

Another common database task is aggregation. That is, reducing your data by breaking it into chunks and summarizing each chunk.

This is done in SQL using the GROUP BY command. This command breaks your data into groups and applies a function from your SELECT statement to each group.

For example, if you wanted to count the number of flights from each of two origin destinations, you could use the query

SELECT COUNT(\*) FROM flights

GROUP BY origin;

GROUP BY origin tells SQL that you want the output to have a row for each unique value of the origin column. The SELECT statement selects the values you want to populate each of the columns.

Here, we want to COUNT() every row in each of the groups.

It's possible to GROUP BY more than one column. When you do this, the resulting table has a row for every combination of the unique values in each column. The following query counts the number of flights from SEA and PDX to every destination airport:

SELECT origin, dest, COUNT(\*) FROM flights

GROUP BY origin, dest;

The output will have a row for every combination of the values in origin and dest (i.e. a row listing each

origin and destination that a flight flew to). There will also be a column with the COUNT() of all the rows in each group.

What information would this query get? Remember the flights table holds information about flights that departed PDX and SEA in 2014 and 2015. Note that AVG() function gets the average value of a column!

SELECT AVG(air\_time) / 60 FROM flights

GROUP BY origin, carrier;

The Spark variant of SQL's SELECT is the .select() method. This method takes multiple arguments - one for each column you want to select. These arguments can either be the column name as a string (one for each column) or a column object (using the df.colName syntax). When you pass a column object, you can perform operations like addition or subtraction on the column to change the data contained in it, much like inside .withColumn().

The difference between .select() and .withColumn() methods is that .select() returns only the columns you specify, while .withColumn() returns all the columns of the DataFrame in addition to the one you defined. It's often a good idea to drop columns you don't need at the beginning of an operation so that you're not dragging around extra data as you're wrangling. In this case, you would use .select() and not .withColumn().

Remember, a SparkSession called spark is already in your workspace, along with the Spark DataFrame flights.

# Select the first set of columns

selected1 = flights.select("tailnum", "origin", "dest")

# Select the second set of columns

temp = flights.select(flights.origin,flights.dest,flights.carrier)

# Define first filter

filterA = flights.origin == "SEA"

# Define second filter

filterB = flights.dest == "PDX"

# Filter the data, first by filterA then by filterB

selected2 = temp.filter(filterA).filter(filterB)

Similar to SQL, you can also use the .select() method to perform column-wise operations. When you're

selecting a column using the df.colName notation, you can perform any column operation and

the .select() method will return the transformed column. For example,

flights.select(flights.air\_time/60)

returns a column of flight durations in hours instead of minutes. You can also use the .alias() method to rename a column you're selecting. So if you wanted to .select() the column duration\_hrs (which isn't in your DataFrame) you could do

flights.select((flights.air\_time/60).alias("duration\_hrs"))

The equivalent Spark DataFrame method .selectExpr() takes SQL expressions as a string:

flights.selectEx

pr("air\_time/60 as duration\_hrs")

with the SQL as keyword being equivalent to the .alias() method. To select multiple columns, you can pass multiple strings.

Remember, a SparkSession called spark is already in your workspace, along with the Spark DataFrame flights.

Create a table of the average speed of each flight both ways.

* Calculate average speed by dividing the distance by the air\_time (converted to hours). Use the .alias() method name this column "avg\_speed". Save the output as the variable avg\_speed.
* Select the columns "origin", "dest", "tailnum", and avg\_speed (without quotes!). Save this as speed1.
* Create the same table using .selectExpr() and a string containing a SQL expression. Save this as speed2.
* # Define avg\_speed
* avg\_speed = (flights.distance/(flights.air\_time/60)).alias("avg\_speed")
* # Select the correct columns
* speed1 = flights.select('origin', 'dest', 'tailnum', avg\_speed)
* # Create the same table using a SQL expression
* speed2 = flights.selectExpr("origin", "dest", "tailnum", "distance/(air\_time/60) as avg\_speed")

**Aggregating**

All of the common aggregation methods, like .min(), .max(), and .count() are GroupedData methods.

These are created by calling the .groupBy() DataFrame method. You'll learn exactly what that means in a

few exercises. For now, all you have to do to use these functions is call that method on your DataFrame. For

example, to find the minimum value of a column, col, in a DataFrame, df, you could do

df.groupBy().min("col").show()

This creates a GroupedData object (so you can use the .min() method), then finds the minimum value

in col, and returns it as a DataFrame.

Now you're ready to do some aggregating of your own!

A SparkSession called spark is already in your workspace, along with the Spark DataFrame flights.

* Find the length of the shortest (in terms of distance) flight that left PDX by first .filter()ing and using the .min() method. Perform the filtering by referencing the column directly, not passing a SQL string.
* Find the length of the longest (in terms of time) flight that left SEA by filter()ing and using the .max() method. Perform the filtering by referencing the column directly, not passing a SQL string.

# Find the shortest flight from PDX in terms of distance

flights.filter(flights.origin == 'PDX').groupBy().min('distance').show()

# Find the longest flight from SEA in terms of air time

flights.filter(flights.origin=='SEA').groupBy().max('air\_time').show()

* Use the .avg() method to get the average air time of Delta Airlines flights (where the carrier column has the value "DL") that left SEA. The place of departure is stored in the
* column origin. show() the result.
* Use the .sum() method to get the total number of hours all planes in this dataset spent in the air by creating a column called duration\_hrs from the column air\_time. show() the result.
* # Average duration of Delta flights
* flights.filter(flights.carrier == "DL").filter(flights.origin == "SEA").groupBy().avg("air\_time").show()
* # Total hours in the air
* flights.withColumn("duration\_hrs", flights.air\_time/60).groupBy().sum("duration\_hrs").show()

**Grouping and Aggregating I**

Part of what makes aggregating so powerful is the addition of groups. PySpark has a whole class devoted to grouped data frames: pyspark.sql.GroupedData, which you saw in the last two exercises.

You've learned how to create a grouped DataFrame by calling the .groupBy() method on a DataFrame with

no arguments.

Now you'll see that when you pass the name of one or more columns in your DataFrame to

the .groupBy() method, the aggregation methods behave like when you use a GROUP BY statement in a SQL query!

Remember, a SparkSession called spark is already in your workspace, along with the Spark DataFrame flights.

* Create a DataFrame called by\_plane that is grouped by the column tailnum.
* Use the .count() method with no arguments to count the number of flights each plane made.
* Create a DataFrame called by\_origin that is grouped by the column origin.
* Find the .avg() of the air\_time column to find average duration of flights from PDX and SEA.

# Group by tailnum

by\_plane = flights.groupBy("tailnum")

# Number of flights each plane made

by\_plane.count().show()

# Group by origin

by\_origin = flights.groupBy("origin")

# Average duration of flights from PDX and SEA

by\_origin.avg('air\_time').show()

**Grouping and Aggregating II**

In addition to the GroupedData methods you've already seen, there is also the .agg() method. This method

lets you pass an aggregate column expression that uses any of the aggregate functions from the pyspark.sql.functions submodule.

This submodule contains many useful functions for computing things like standard deviations. All the aggregation functions in this submodule take the name of a column in a GroupedData table.

Remember, a SparkSession called spark is already in your workspace, along with the Spark DataFrame flights. The grouped DataFrames you created in the last exercise are also in your workspace.

* Import the submodule pyspark.sql.functions as F.
* Create a GroupedData table called by\_month\_dest that's grouped by both the month and dest columns. Refer to the two columns by passing both strings as separate arguments.
* Use the .avg() method on the by\_month\_dest DataFrame to get the average dep\_delay in each month for each destination.
* Find the standard deviation of dep\_delay by using the .agg() method with the function F.stddev().

# Import pyspark.sql.functions as F

import pyspark.sql.functions as F

# Group by month and dest

by\_month\_dest = flights.groupBy('month','dest')

# Average departure delay by month and destination

by\_month\_dest.avg('dep\_delay').show()

# Standard deviation of departure delay

by\_month\_dest.agg(F.stddev('dep\_delay')).show()

**Joining II**

In PySpark, joins are performed using the DataFrame method .join(). This method takes three arguments. The first is the second DataFrame that you want to join with the first one. The second argument, on, is the name of the key column(s) as a string. The names of the key column(s) must be the same in each table. The

third argument, how, specifies the kind of join to perform. In this course we'll always use the value how="leftouter".

The flights dataset and a new dataset called airports are already in your workspace.

* Examine the airports DataFrame by calling .show(). Note which key column will let you join airports to the flights table.
* Rename the faa column in airports to dest by re-assigning the result of airports.withColumnRenamed("faa", "dest") to airports.
* Join the flights with the airports DataFrame on the dest column by calling the .join() method on flights. Save the result as flights\_with\_airports.
  + The first argument should be the other DataFrame, airports.
  + The argument on should be the key column.
  + The argument how should be "leftouter".
* Call .show() on flights\_with\_airports to examine the data again. Note the new information that has been added.

# Examine the data

print(airports.show())

# Rename the faa column

airports = airports.withColumnRenamed('faa','dest')

# Join the DataFrames

flights\_with\_airports = flights.join(airports,on='dest',how='leftouter')

# Examine the new DataFrame

print(flights\_with\_airports.show())

Good work! Before you get started modeling, it's important to know that Spark only handles numeric data. That means all of the columns in your DataFrame must be either integers or decimals (called 'doubles' in Spark).

When we imported our data, we let Spark guess what kind of information each column held. Unfortunately, Spark doesn't always guess right and you can see that some of the columns in our DataFrame are strings containing numbers as opposed to actual numeric values.

To remedy this, you can use the .cast() method in combination with the .withColumn() method. It's important to note that .cast() works on columns, while .withColumn() works on DataFrames.

The only argument you need to pass to .cast() is the kind of value you want to create, in string form. For example, to create integers, you'll pass the argument "integer" and for decimal numbers you'll use "double".

You can put this call to .cast() inside a call to .withColumn() to overwrite the already existing column, just like you did in the previous chapter!

Spark requires numeric data for modeling. So far this hasn't been an issue; even boolean columns can easily be converted to integers without any trouble. But you'll also be using the airline and the plane's destination as features in your model. These are coded as strings and there isn't any obvious way to convert them to a numeric data type.

Fortunately, PySpark has functions for handling this built into the pyspark.ml.features submodule. You can create what are called 'one-hot vectors' to represent the carrier and the destination of each flight. A one-hot vector is a way of representing a categorical feature where every observation has a vector in which all elements are zero except for at most one element, which has a value of one (1).

Each element in the vector corresponds to a level of the feature, so it's possible to tell what the right level is by seeing which element of the vector is equal to one (1).

The first step to encoding your categorical feature is to create a StringIndexer. Members of this class are Estimators that take a DataFrame with a column of strings and map each unique string to a number. Then, the Estimator returns a Transformer that takes a DataFrame, attaches the mapping to it as metadata, and returns a new DataFrame with a numeric column corresponding to the string column.

The second step is to encode this numeric column as a one-hot vector using a OneHotEncoder. This works exactly the same way as the StringIndexer by creating an Estimator and then a Transformer. The end result is a column that encodes your categorical feature as a vector that's suitable for machine learning routines!

So we need to create a StringIndexer and a OneHotEncoder, and the Pipeline will take care of the rest.

Carrier column

# Create a StringIndexer

carr\_indexer = StringIndexer(inputCol="carrier",outputCol="carrier\_index")

# Create a OneHotEncoder

carr\_encoder = OneHotEncoder(inputCol="carrier\_index",outputCol="carrier\_fact")

Destination Column

# Create a StringIndexer

dest\_indexer = StringIndexer(inputCol="dest",outputCol="dest\_index")

# Create a OneHotEncoder

dest\_encoder = OneHotEncoder(inputCol="dest\_index",outputCol="dest\_fact")

**Assemble a vector**

The last step in the Pipeline is to combine all of the columns containing our features into a single column. This has to be done before modeling can take place because every Spark modeling routine expects the data to be in this form. You can do this by storing each of the values from a column as an entry in a vector. Then, from the model's point of view, every observation is a vector that contains all of the information about it and a label that tells the modeler what value that observation corresponds to.

Because of this, the pyspark.ml.feature submodule contains a class called VectorAssembler. This Transformer takes all of the columns you specify and combines them into a new vector column.

# Make a VectorAssembler

vec\_assembler = VectorAssembler(inputCols=["month", "air\_time", "carrier\_fact", "dest\_fact", "plane\_age"], outputCol="features")

**Create the pipeline**

You're finally ready to create a Pipeline!

Pipeline is a class in the pyspark.ml module that combines all the Estimators and Transformers that you've already created. This lets you reuse the same modeling process over and over again by wrapping it up in one simple object. Neat, right?

* Import Pipeline from pyspark.ml.
* Call the Pipeline() constructor with the keyword argument stages to create a Pipeline called flights\_pipe.
* stages should be a list holding all the stages you want your data to go through in the pipeline. Here this is just: [dest\_indexer, dest\_encoder, carr\_indexer, carr\_encoder, vec\_assembler]
* # Import Pipeline
* from pyspark.ml import Pipeline
* # Make the pipeline
* flights\_pipe = Pipeline(stages=[dest\_indexer, dest\_encoder, carr\_indexer, carr\_encoder, vec\_assembler])
* **Test vs Train**
* After you've cleaned your data and gotten it ready for modeling, one of the most important steps is to split the data into a *test set* and a *train set*. After that, don't touch your test data until you think you have a good model! As you're building models and forming hypotheses, you can test them on your training data to get an idea of their performance.
* Once you've got your favorite model, you can see how well it predicts the new data in your test set. This never-before-seen data will give you a much more realistic idea of your model's performance in the real world when you're trying to predict or classify new data.
* In Spark it's important to make sure you split the data **after** all the transformations. This is because operations like StringIndexer don't always produce the same index even when given the same list of strings.
* Why is it important to use a test set in model evaluation?

A)By evaluating your model with a test set you can get a good idea of performance on new data

**Transform the data**

Hooray, now you're finally ready to pass your data through the Pipeline you created!

* Create the DataFrame piped\_data by calling the Pipeline methods .fit() and .transform() in a chain. Both of these methods take model\_data as their only argument.

# Fit and transform the data

piped\_data = flights\_pipe.fit(model\_data).transform(model\_data)

**Split the data**

Now that you've done all your manipulations, the last step before modeling is to split the data!

Use the DataFrame method .randomSplit() to split piped\_data into two pieces, training with 60% of the data, and test with 40% of the data by passing the list [.6, .4] to the .randomSplit() method.

# Split the data into training and test sets

training, test = piped\_data.randomSplit([.6, .4])

Getting versions

# Print the version of SparkContext

print("The version of Spark Context in the PySpark shell is", sc.version)

# Print the Python version of SparkContext

print("The Python version of Spark Context in the PySpark shell is", sc.pythonVer)

# Print the master of SparkContext

print("The master of Spark Context in the PySpark shell is", sc.master)

# Create a Python list of numbers from 1 to 100

numb = range(1, 101)

# Load the list into PySpark

spark\_data = sc.parallelize(numb)

# Load a local file into PySpark shell

lines = sc.textFile(file\_path)

# Print my\_list in the console

print("Input list is", my\_list)

# Square all numbers in my\_list

squared\_list\_lambda = list(map(lambda x: x\*\*2, my\_list))

# Print the result of the map function

print("The squared numbers are", squared\_list\_lambda)

# Print my\_list2 in the console

print("Input list is:", my\_list2)

# Filter numbers divisible by 10

filtered\_list = list(filter(lambda x: (x%10 == 0), my\_list2))

# Print the numbers divisible by 10

print("Numbers divisible by 10 are:", filtered\_list)

A Partition is a logical division of large distributed data set.

numRDD = sc.parallelize(range(10),minPartitions=6)

fileRDD = sc.textFile(‘Readme.md’,minPartitions=6)

To see number of partitions use getNumPartitions() method.

# Create an RDD from a list of words

RDD = sc.parallelize(["Spark", "is", "a", "framework", "for", "Big Data processing"])

# Print out the type of the created object

print("The type of RDD is", type(RDD))

# Print the file\_path

print("The file\_path is", file\_path)

# Create a fileRDD from file\_path

fileRDD = sc.textFile(file\_path)

# Check the type of fileRDD

print("The file type of fileRDD is", type(fileRDD))

# Check the number of partitions in fileRDD

print("Number of partitions in fileRDD is", fileRDD.getNumPartitions())

# Create a fileRDD\_part from file\_path with 5 partitions

fileRDD\_part = sc.textFile(file\_path, minPartitions = 5)

# Check the number of partitions in fileRDD\_part

print("Number of partitions in fileRDD\_part is", fileRDD\_part.getNumPartitions())

RDD’s in Spark support two types of operations, 1.Transformations 2.Actions

**1.Transformations** create new RDD’s. Transformations follow lazy evaluations. Lazy evaluation means that Spark does not evaluate each transformation as they arrive, but instead queues them together and evaluate all at once, as an Action is called.

Basic RDD Transformations:- map(),filter(),flatMap(),union()

**map():-** Map function takes a function and a list as a arguments and applies that function for every element in the given list and gives the o/p for the whole list.

RDD = sc.parallelise([1,2,3,4,5,6])

RDD\_map = RDD.map(lambda x: x\*\*2)

**filter():-** Filter function takes function and a list as arguments and applies that function for every element in the list and returns the o/p only when the element satisfies the condition(True elements) and append to the list.

RDD = sc.parallise([1,2,3,4,5,6])

RDD\_filter = RDD.filter(lambda x: x>2)

**flatMap():** FlatMap Transformation takes each element in the list and return one or more values for that element according to the condition applied to that element.

RDD=sc.parallise([‘Hello World’, ’Hello How are you’])

RDD\_flatmap = RDD.flatMap(lambda x: x.split(‘ ’))

o/p:- [‘Hello’,’World’,’Hello’,’How’,’are’,’you’]

**union():-** Union Transformation returns the union of one RDD with another RDD.

inputRDD = sc.textFile(‘logs.txt’)

errorRDD = inputRDD.filter(lambda x: ‘error’ in x.split())

warningsRDD = inputRDD.filter(lambda x: ‘warnings’ in x.split())

combinedRDD = errorRDD.union(warningsRDD)

**2.Actions** perform computations on RDD’s, (Operations returns a value after doing computations on RDD).

RDD Basic Actions:- collect(),take(N),first(),count()

collect():- collect returns all the elements of the dataset as an array.**EX:** RDD\_map.collect():- [1,4,9,6]

take(N): take returns only the first N elements of the dataset as an array. **EX:** RDD\_map.take(2):- [1,4]

first():- first Action returns only first element from the dataset **EX:** RDD\_map.first():- [1]

count():- count action returns total number of elements or rows in RDD. **EX:** RDD\_map.count():- 4

# Create map() transformation to cube numbers

cubedRDD = numbRDD.map(lambda x: x\*\*3)

# Collect the results

numbers\_all = cubedRDD.collect()

# Print the numbers from numbers\_all

for numb in numbers\_all:

    print(numb)

# Filter the fileRDD to select lines with Spark keyword

fileRDD\_filter = fileRDD.filter(lambda x: 'Spark' in x)

# How many lines are there in fileRDD?

print("The total number of lines with the keyword Spark is", fileRDD\_filter.count())

# Print the first four lines of fileRDD

for line in fileRDD\_filter.take(4):

  print(line)

Pair RDD’s Actions:-

Real life data sets are key/value pairs, Each row is a key and maps to one or more values. Pair RDD is a special data structure to work with this kind of data.

**Pair RDD**: key is the identifier and value is data.

**Creating Pair RDD:- Two ways to create pair RDD**

1. **From a list of key-value tuple**
2. **From a regular RDD**

My\_tuple = [(‘Sam’,23),(‘Mary’,34),(‘Peter’,25)]

pairRDD\_tuple = sc.parallelise(my\_tuple)

my\_list= [‘Sam 23’,’Mary 34’,’Peter 25’]

regular\_RDD = mylist.parallelise(my\_list’)

pair\_RDD = regular\_RDD.map(lambda x: (x.split(“ “)[0]),x.split(“ ”)[1])

All regular transformations works on RDD, but functions should bbe function with key value pair RDD’s. Some of paired RDD transformations are…

**reduceByKey(func):-** Combine values with same keys using a function. It runs parallel operations for each key in the dataset.

regularRDD = sc.parallelise([(“Messi”,23),(“Ronaldo”,34),(“Neymar”,22),(“Messi”,24)])

pairRDD\_reducebykey = regularRDD.reduceByKey(lambda x,y: x+y)

pairRDD\_reducebykey.collect()

o/p:- [(“Messi”,47),(“Ronaldo”,34),(“Neymar”,22)]

# Create PairRDD Rdd with key value pairs

Rdd = sc.parallelize([(1,2),(3,4),(3,6),(4,5)])

# Apply reduceByKey() operation on Rdd

Rdd\_Reduced = Rdd.reduceByKey(lambda x, y: x+y)

# Iterate over the result and print the output

for num in Rdd\_Reduced.collect():

  print("Key {} has {} Counts".format(num[0], num[1]))

**groupByKey():-** group values with same key..

airports = [(‘US’,”JFK”),(“UK”,”LHR”),(“FR”,”CDG”),(“US”,”SFO”)]

regularRDD = sc.parallelise(airports)

pairRDD\_grp = regularRDD.groupByKey().collect()

for cnt,air in pairRDD\_grp:

print(cont, list(air))

**sortByKey():-** Return an RDD with sorted by the key.

pairRDD\_reducebykey\_rev.sortByKey(ascending=False).collect()

**join():-** join two RDD’s based on their key.

RDD1.join(RDD2).collect()

# Create PairRDD Rdd with key value pairs

Rdd = sc.parallelize([(1,2),(3,4),(3,6),(4,5)])

# Apply reduceByKey() operation on Rdd

Rdd\_Reduced = Rdd.reduceByKey(lambda x, y: x+y)

# Sort the reduced RDD with the key by descending order

Rdd\_Reduced\_Sort = Rdd\_Reduced.sortByKey(ascending=False)

# Iterate over the result and retrieve all the elements of the RDD

for num in Rdd\_Reduced\_Sort.collect():

  print("Key {} has {} Counts".format(num[0], num[1]))