PySpark is an interface for Apache Spark in Python. It not only allows us to write Spark applications using Python APIs, but also provides the PySpark shell for interactively analyzing the data in a distributed environment.

*PySpark Basics*

Key Concepts – DataFrames, Pandas or Koalas, Visualizations

DataFrames are a columnar structure, a lot like a CSV file or an Excel Spreadsheet, and we can interact with these DataFrames using the regular Pandas API or this Koalas interface which has an interface that is able to talk to the Spark language and work with DataFrames at scale.

Data Lake is used for Object Storage. Databricks File System (DBFS Cluster) that is able to go through here and merge everything together, and then we get this big data operation. It could be an add function where we take an input, we do that calculation and apply it to all the rows on the DataFrame, and then do our maybe visualization.

*PySpark DataFrames*

#import pyspark class row from module sql

from pyspark.sql import \*

#create the departments

department1 = Row(id=’123456’, name=’Computer Science’)

#create the employees

Employee = Row(“firstname”, “lastname”, “email”, “salary”)

employee1 = Employee(‘michael’, ‘armbrust’, ‘no-reply@berkeley.edu’, 100000)

#create the DepartmentWithEmployees instances from Departments and Employees  
departmentWithEmployees1 = Row(department=department1, employees=[employee1, employee2])

*Join DataFrames*

unionDF = df1.union(df2)

Display(unionDF)

*UDF (User Defined Functions)*

If we wanted to build a user-defined function or a UDF, we would build a function, have an input, return something, but then register that function so that we can apply it to DataFrame. That’s really where the power comes in taking these functions and applying them at scale.

def squared(s):

return s \* s

spark.udf.register(“squareWithPython”, squared)

from pyspark.sql.functions import udf

from pyspark.sql.types import LongType

squared\_udf = udf(squared, LongType())

df=spark.table(“test”)

display(df.select(“id”, squared\_udf(“id”).alias(“id\_squared”)))

*PySpark Pandas API*

This feature is available on clusters that run Databricks Runtime 10.0 and above. For clusters that run Databricks Runtime 9.1 LTS and below use Koalas instead.

PySpark Pandas API is quite interesting because it solves a problem that Pandas has which is it doesn’t scale to big data. One of the limitations of the regular Pandas API is that it works on datasets that are more in the academic range. But when we have the real-world scenario, we need to have bigger data platforms to interact with.

Import pyspark.pandas as ps

*Pandas API Key Facts*

1. pandas-on-Spark DataFrame and pandas DataFrame are similar
2. Spark version is distributed (i.e., scales to data)

Import pyspark.pandas as ps

psdf = ps.range(10)

pdf = psdf.to\_pandas()

pdf.values

*Migration from Pandas to Pandas API on Spark*

import numpy as np

import pandas as pd

import pyspark.pandas as ps

#create a pandas series

pser = pd.Series([1, 3, 5, np.nan, 6, 8])

#create a pandas-on-spark series

psser = ps.Series([1, 3, 5, np.nan, 6, 8])

#create a pandas-on-spark series by passing a pandas series

psser = ps.Series(pser)

psser = ps.from\_pandas(pser)

#print pandas series

pser

#print pandas-on-spark series

psser

#sort index of pandas-on-spark series

psser.sort\_index()

#create a pandas dataframe

pdf = pd.DataFrame({‘A’: np.random.rand(5),

‘B’: np.random.rand(5)})

#create a pandas-on-spark dataframe

psdf = ps.DataFrame({‘A’: np.random.rand(5),

‘B’: np.random.rand(5)})

#create a pandas-on-spark dataframe by passing a pandas dataframe

psdf = ps.DataFrame(pdf)

psdf = ps.from\_pandas(pdf)

#print pandas dataframe

pdf

#print sorted pandas-on-spark dataframe

psdf.sort\_index(

#print top two values of pandas-on-spark

psdf.head(2)

#describe pandas-on-spark dataframe

psdf.describe()

#sort values of pandas-on-spark dataframe

psdf.sort\_values(by=’B’)

#transpose pandas-on-spark dataframe

psdf.transpose()

#get an option for pandas-on-spark

ps.get\_option(‘compute.max\_rows’)

#set an option and get an option for pandas-on-spark

ps.set\_option(‘compute.max\_rows’, 2000)

ps.get\_option(‘compute.max\_rows’)

#select columns of pandas-on-spark dataframe

psdf[‘A’]

psdf.A

psdf[[‘A’,’B’]]

#print values with loc in pandas-on-spark dataframe

psdf.loc[1:2]

#print values with iloc in pandas-on-spark dataframe

psdf.iloc[:3, 1:2]

#create a new column in pandas-on-spark dataframe

psser = ps.Series([100, 200, 300, 400, 500], index = [0, 1, 2, 3, 4])

from pyspark.pandas.config import set\_option, reset\_option

set\_option(“compute.ops\_on\_diff\_frames”, True)

psdf[‘C’] = psser

reset\_option(“compute.ops\_on\_diff\_frames”)

psdf

#apply python function with pandas-on-spark object

psdf.apply(np.cumsum, axis=1)

psdf.apply(lambda x: x \*\* 2)

#apply user-defined function on pandas-on-spark dataframe

def square(x) -> ps.Series[np.float64]:

return x \*\* 2

psdf.apply(square)

#working properly since size of data <= compute.shortcut\_limit (1000)

ps.DataFrame({‘A’: range(1000)}).apply(lambda col: col.max())

#not working properly since size of data > compute.shortcut\_limit (1000)

ps.DataFrame({‘A’: range(1001)}).apply(lambda col: col.max())

*Setup PySpark for Data Analysis*

Apache Spark native shell is in Scala. So, to instead leverage the Apache Spark capabilities in a language that may have more of the modules and utilities readily available for us to import, PySpark bridges that gap by making it available within Python. The “Py” in PySpark due to its interface feature for Apache Spark in Python.

To install PySpark in Jupyter Notebook,

Pip install pyspark

Post-installation, to check the version:

import pyspark

print(pyspark.\_\_version\_\_)

Just a sample code:

import findspark

findspark.init()

import pyspark

import random

sc = pyspark.SparkContext(appName="Pi")

num\_samples = 100000000

def inside(p):

x, y = random.random(), random.random()

return x\*x + y\*y < 1

count = sc.parallelize(range(0, num\_samples)).filter(inside).count()

pi = 4 \* count / num\_samples

print(pi)

sc.stop()

*Introduction to Spark and PySpark DataFrames*

*Objectives:*

1. Distributed solutions for Big Data
2. Lazy vs. eager evaluations
3. Considerations when choosing Spark and PySpark over Pandas

* Pandas is designed to run on single machine.
* Data analysis performance using Pandas is bound by machines memory.
* Data chunking is possible to do while analyzing the data using Pandas, but the overall performance will NOT be ideal while handing the errors.
* Upper limit for data size in Pandas is in Gigabytes (1? 5? 100?): As a thump rule, without chunking, the data size varies between 1 to 5 Gigabytes (109), and with chunking mostly, 100 Gigabytes.
* If the data size is in Terabytes (1012) or Petabytes (1015), then it is called Big Data, and while analyzing it, we need to consider some other solutions.

Two widely adopted solutions for analyzing Big Data are **Hadoop** and **Spark**. Both are using distributed computing. This means instead of using a single node or single thread, it uses multiple nodes. These nodes could be different computers, different virtual machines or different sub-processes, but regardless

they allow to scale horizontally, i.e., add more machines to increase performance not necessarily need to add much larger and more expensive machines.

Hadoop does this distributed computing by writing intermediate files during calculations. So, Hadoop doesn’t require a great amount of RAM.

Spark, which is a newer solution, and it is more performant, does work in memory but it’s using the memory of all these nodes. It has much larger memory to use than Pandas normally would.

* Spark at its core uses DataFrames, and these are distributed DataFrames, and they are written using the Java Virtual Machine.
* Spark is written in Scala.
* Spark offers a number of libraries, including the PySpark library, which allows to use Spark DataFrames and transformations in a Python context.
* Spark can process data from a wide range of data sources including the Hadoop HDFS (distributed data source), files on S3 and Streaming data.
* Spark uses lazy evaluation, whereas Pandas uses eager evaluations (calculates its result before the next operation is run; eager evaluation is great while working interactively, and it’s much more intuitive to debug). Lazy evaluation is taking advantage of PySpark. In lazy evaluation, the operations are stacked, and behind the scenes, an optimized transformation is calculated by Spark. This means the results are not calculated immediately after an individual operation but only after we hit an operation that requires that results calculation. This combined with its distributed nature are what made Spark so efficient for calculating Big Data.

What are some considerations when choosing Spark versus Pandas?

* If we work with a CSV file, which is smaller than Gigabytes (109), we don’t need Spark or Spark’s infrastructure. But if part of plan is to deal with Terabytes (1012) of data or if we find that Pandan’s implementation is not performant for the transformations, Spark is a great solution.
* Optimized integration plan strategy with lazy evaluation means that we have to also plan the way we write the transformations. If we write an operation that requires calculation, for example, an operation that does the equivalent of Pandas head (prints the first few rows of a DataFrame), that requires that any previous operations need to be run so that those results are calculated before displaying them. This means when we write PySpark code, we need to understand which operations are going to cause the result to be calculated and be strategic when we call those operations.
* PySpark is a layer on top of Spark, debugging can at times be challenging. Often when we look at the error in log, we see the Python exception then we see some code that relates to the translation layer, and then below that a Stack trace in the Scala layer. Hence, there is a lot to dig to figure out what’s wrong.

In summary, the characteristics of Spark:

* Build for Big Data transformations
* Optimized integration plan strategy
* Debugging challenges

All that being said, Spark and PySpark are great solutions for making transformations of Big Data.

*A Crash Course in PySpark*

*Course Structure*

* Data ingestion
* Data clean-up
* Answering some business questions by analyzing Big Data in PySpark, like
  + Between men and women, who get paid more on average
  + Split by job title, between men and women, who get paid more on average
  + Which city has the highest average salary? Does geography impact the average salary?

*Environment Configuration Code*

!apt-get install openjdk-8-jdk-headless -qq > /dev/null

!wget -q http://archive.apache.org/dist/spark/spark-3.1.1/spark-3.1.1-bin-hadoop3.2.tgz

!tar xf spark-3.1.1-bin-hadoop3.2.tgz

!pip install -q findspark

import os

os.environ["JAVA\_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"

os.environ["SPARK\_HOME"] = "/content/spark-3.1.1-bin-hadoop3.2"

import findspark

findspark.init()

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[\*]").getOrCreate()

spark

*A Crash Course in PySpark*

*Basic Syntax*

# pyspark initialization

import findspark

findspark.init()

#building sparksession

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[\*]").getOrCreate()

spark

#loading data

mydata = spark.read.format("csv").option("header","true").load("original.csv")

mydata.show()

#importing all pyspark functions

from pyspark.sql.functions import \*

#replacing NULL values with 'Unknown' for 'City' column and populating it to new column 'clean\_city'

mydata2 = mydata.withColumn("clean\_city", when(mydata.City.isNull(), 'Unknown').otherwise(mydata.City))

mydata2.show()

#filtering out NULL values from 'JobTitle' column

mydata2 = mydata2.filter(mydata2.JobTitle.isNotNull())

mydata2.show()

#removing $-sign from 'Salary' column and updating it to new column 'clean\_salary'

mydata2 = mydata2.withColumn("clean\_salary", mydata2.Salary.substr(2, 100).cast('float'))

mydata2.show()

#calculating salary mean to replace salary NULL values

mean = mydata2.groupby().avg("clean\_salary")

mean.show()

mean = mean.take(1)[0][0]

print(mean)

#replacing NULL values with salary mean and updating it to new column 'new\_salary'

from pyspark.sql.functions import lit

mydata2 = mydata2.withColumn("new\_salary", when(mydata2.clean\_salary.isNull(), lit(mean)).otherwise(mydata2.clean\_salary))

mydata2.show()

#selecting only 'Latitude' column

latitudes = mydata2.select("Latitude")

latitudes.show()

#filtering out NULL values from 'Latitude' column

latitudes = latitudes.filter(latitudes.Latitude.isNotNull())

latitudes.show()

#typecasting latitude values to float

latitudes = latitudes.withColumn('latitudes2', latitudes.Latitude.cast('float')).select('latitudes2')

latitudes.show()

#calculating median of latitudes

import numpy as np

median = np.median(latitudes.collect())

print(median)

#median imputation for NULL values of latitude

mydata2 = mydata2.withColumn('lat', when(mydata2.Latitude.isNull(), lit(median)).otherwise(mydata2.Latitude))

mydata2.show()

#Business Question 1: Between men and women, who get paid more on average?

import pyspark.sql.functions as sqlfunc

genders = mydata2.groupby('gender').agg(sqlfunc.avg('new\_salary').alias('AvgSalary'))

genders.show()

#Business Question 2 (Step 1): Split by job title, between men and women, who get paid more on average?

df = mydata2.withColumn('female\_salary', when(mydata2.gender == 'Female', mydata2.new\_salary).otherwise(lit(0)))

pdf = df.toPandas()

display(pdf)

#Business Question 2 (Step 2): Split by job title, between men and women, who get paid more on average?

df = df.withColumn('male\_salary', when(df.gender == 'Male', df.new\_salary).otherwise(lit(0)))

pdf = df.toPandas()

display(pdf)

#Business Question 2 (Step 3): Split by job title, between men and women, who get paid more on average?

df = df.groupBy('JobTitle').agg(sqlfunc.avg('female\_salary').alias('final\_female\_salary'), sqlfunc.avg('male\_salary').alias('final\_male\_salary'))

df.show()

#Business Question 2 (Step 4): Split by job title, between men and women, who get paid more on average?

df = df.withColumn('delta', df.final\_female\_salary - df.final\_male\_salary)

df.show()

#Business Question 3: Which city has the highest average salary? Does geography impact the average salary?

cityavg = mydata2.groupBy('City').agg(sqlfunc.avg('new\_salary').alias('avgsalary'))

cityavg = cityavg.sort(col('avgsalary').desc())

cityavg.show()

*Join Functions*

# pyspark initialization

import findspark

findspark.init()

#building sparksession

import pyspark

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[\*]").getOrCreate()

#dataframe\_1

data\_1 = [["1", "sravan", "company 1"],

["2", "ojaswi", "company 1"],

["3", "rohith", "company 2"],

["4", "sridevi", "company 1"],

["5", "bobby", "company 1"]]

columns\_1 = ['ID', 'NAME', 'Company']

dataframe\_1 = spark.createDataFrame(data\_1, columns\_1)

dataframe\_1.show()

#dataframe\_2

data\_2 = [["1", "45000", "IT"],

["2", "145000", "Manager"],

["6", "45000", "HR"],

["5", "34000", "Sales"]]

columns\_2 = ['ID', 'salary', 'department']

dataframe\_2= spark.createDataFrame(data\_2, columns\_2)

dataframe\_2.show()

#inner join of two dataframes (only matching rows)

dataframe\_1.join(dataframe\_2, dataframe\_1.ID == dataframe\_2.ID, "inner").show()

#full outer join of two dataframes (both matching and non-matching rows)

dataframe\_1.join(dataframe\_2, dataframe\_1.ID == dataframe\_2.ID, "outer").show()

dataframe\_1.join(dataframe\_2, dataframe\_1.ID == dataframe\_2.ID, "full").show()

dataframe\_1.join(dataframe\_2, dataframe\_1.ID == dataframe\_2.ID, "fullouter").show()

#left join of two dataframes

# returning all rows from the first dataframe and only matched rows from the second dataframe with respect to the first dataframe

dataframe\_1.join(dataframe\_2, dataframe\_1.ID == dataframe\_2.ID, "left").show()

dataframe\_1.join(dataframe\_2, dataframe\_1.ID == dataframe\_2.ID, "leftouter").show()

#right join of two dataframes

# returning all rows from the second dataframe and only matched rows from the first dataframe with respect to the second dataframe

dataframe\_1.join(dataframe\_2, dataframe\_1.ID == dataframe\_2.ID, "right").show()

dataframe\_1.join(dataframe\_2, dataframe\_1.ID == dataframe\_2.ID, "rightouter").show()

#leftsemi join of two dataframes

# returns only those rows from first dataframe for which there are match in second dataframe

dataframe\_1.join(dataframe\_2, dataframe\_1.ID == dataframe\_2.ID, "leftsemi").show()

#leftanti join of two dataframes

# returns only those rows from first dataframe for which there are mismatch in second dataframe

dataframe\_1.join(dataframe\_2, dataframe\_1.ID == dataframe\_2.ID, "leftanti").show()

# cross join of two dataframes

# returns Cartesian product of two DataFrames

# combines every row from the first DataFrame with every row from the second DataFrame, resulting in a large, unfiltered result

dataframe\_1.crossJoin(dataframe\_2).show()

*Window Functions*

#pyspark initialization

import findspark

findspark.init()

#importing libraries

import pyspark

import pyspark.sql.functions as sqlfunc

from pyspark.sql.functions import \*

#building sparksession

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[\*]").getOrCreate()

#sample\_data

sampleData = (("Ram", 28, "Sales", 3000),

("Meena", 33, "Sales", 4600),

("Robin", 40, "Sales", 4100),

("Kunal", 25, "Finance", 3000),

("Ram", 28, "Sales", 3000),

("Srishti", 46, "Management", 3300),

("Jeny", 26, "Finance", 3900),

("Hitesh", 30, "Marketing", 3000),

("Kailash", 29, "Marketing", 2000),

("Sharad", 39, "Sales", 4100)

)

columns = ["Employee\_Name", "Age", "Department", "Salary"]

df = spark.createDataFrame(data=sampleData, schema=columns)

df.printSchema()

df.show()

# defining window partition

from pyspark.sql.window import Window

windowPartition = Window.partitionBy("Department").orderBy("Age")

#analytical window function

# cume\_dist() window function is used to get the cumulative distribution within a window partition

from pyspark.sql.functions import cume\_dist

df.withColumn("cume\_dist", cume\_dist().over(windowPartition)).show()

#analytical window function

# lag() function: access previous rows’ data as per the defined offset value in the function

from pyspark.sql.functions import lag

df.withColumn('lag', lag('Salary', 1).over(windowPartition)).show()

#analytical window function

# lead() function: access next rows data as per the defined offset value in the function

from pyspark.sql.functions import lead

df.withColumn("lead", lead("Salary", 1).over(windowPartition)).show()

#sample\_data

sampleData = ((101, "Ram", "Biology", 80),

(103, "Meena", "Social Science", 78),

(104, "Robin", "Sanskrit", 58),

(102, "Kunal", "Physics", 89),

(101, "Ram", "Biology", 80),

(106, "Hitesh", "Maths", 88),

(108, "Jeny", "Physics", 75),

(107, "Hitesh", "Maths", 88),

(109, "Kailash", "Maths", 90),

(105, "Sharad", "Social Science", 84)

)

columns = ["Roll\_No", "Student\_Name", "Subject", "Marks"]

df2 = spark.createDataFrame(data=sampleData, schema=columns)

df2.printSchema()

df2.show()

# defining window partition

from pyspark.sql.window import Window

windowPartition1 = Window.partitionBy("Subject").orderBy("Marks")

#ranking window function

# row\_number(): used to give a sequential number to each row present in the table

from pyspark.sql.functions import row\_number

df2.withColumn("row\_number", row\_number().over(windowPartition1)).show()

#ranking window function

# rank(): used to give ranks to rows specified in the window partition

# leaves gaps in rank if there are ties

from pyspark.sql.functions import rank

df2.withColumn("rank", rank().over(windowPartition1)).show()

#ranking window function

#percent\_rank(): provides rank to rows in a percentile format

from pyspark.sql.functions import percent\_rank

df2.withColumn("percent\_rank", percent\_rank().over(windowPartition1)).show()

#ranking window function

# dense\_rank(): similar to rank() function, there is only one difference - dense\_rank() function doesn't leave gaps in rank when there are ties

from pyspark.sql.functions import dense\_rank

df2.withColumn("dense\_rank", dense\_rank().over(windowPartition1)).show()

#sample\_data

sampleData = (("Ram", "Sales", 3000),

("Meena", "Sales", 4600),

("Robin", "Sales", 4100),

("Kunal", "Finance", 3000),

("Ram", "Sales", 3000),

("Srishti", "Management", 3300),

("Jeny", "Finance", 3900),

("Hitesh", "Marketing", 3000),

("Kailash", "Marketing", 2000),

("Sharad", "Sales", 4100)

)

columns = ["Employee\_Name", "Department", "Salary"]

df3 = spark.createDataFrame(data=sampleData, schema=columns)

df3.printSchema()

df3.show()

# defining window partition

from pyspark.sql.window import Window

windowPartitionAgg = Window.partitionBy("Department")

#aggregate window functions - avg, sum, min, max

from pyspark.sql.functions import col,avg,sum,min,max

df3.withColumn("Avg", avg(col("Salary")).over(windowPartitionAgg))\

.withColumn("Sum", sum(col("Salary")).over(windowPartitionAgg))\

.withColumn("Min", min(col("Salary")).over(windowPartitionAgg))\

.withColumn("Max", max(col("Salary")).over(windowPartitionAgg)).show()

*Join Strategies*

#pyspark initialization

import findspark

findspark.init()

#building spark session

import pyspark

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[\*]").getOrCreate()

spark

#Points to Note

# Spark chooses a join strategy based on the size of the data.

# To avoid costly shuffle and sort operations, it favors hash-based join strategies, especially when data can be broadcasted.

# Spark supports both Equi Join (using “=”) and Non-Equi Join (using “<,” “>”, “≥,” “≤”)

# clusters: connecting multiple computers (desktops/laptops) through network. 20 core per machine, and 100GB RAM in each. Hence, for 10 clusters, 200 core and 1000 GB (1 TB) RAM.

# Clusters are working in master-slave architecture; slaves are the worker nodes.

# Master node consists of YARN (Yet Another Resource Negotiator), and resource manager, whereas, worker nodes consist Node Manager

# Spark Architecture: <https://youtu.be/xJVonk4yxJY?si=jiXGgEfHymD_sy0Y>

#dataframe\_1

df\_clients = [

(0, "client1"),

(1, "client2"),

(2, "client3")]

columns = ("client\_id", "name")

df\_clients = spark.createDataFrame(df\_clients, columns)

df\_clients.show()

#dataframe\_2

df\_orders = [

(0, "order1", 100),

(1, "order2", 200),

(2, "order3", 150)]

columns = ("client\_id", "order\_id", "order\_amount")

df\_orders = spark.createDataFrame(df\_orders, columns)

df\_orders.show()

#Broadcast Hash Join

# It’s ideal when one DataFrame is small enough to fit in the memory of each executor.

# Spark broadcasts the smaller DataFrame to all workers.

# This minimizes data shuffling and accelerates the join operation, as the join occurs within the same node, resulting in a decrease in network overhead.

broadcast\_df = df\_clients.hint('BROADCAST').join(df\_orders, on = "client\_id", how = "inner")

# broadcast\_df.explain(mode="formatted")

broadcast\_df.show()

#Shuffle Hash Join

# It’s suitable when neither of the joined tables can fit in memory.

# It involves a shuffle phase, where data is redistributed across partitions based on the join key.

# Be careful when using this strategy, as it may incur higher network and disk I/O costs, which largely decreases the performance due to the full shuffle.

shuffle\_hash\_df = df\_clients.hint("SHUFFLE\_HASH").join(df\_orders, on = "client\_id", how = "inner")

shuffle\_hash\_df.show()

#Sort Merge Join

# It’s appropriate when both tables are large and cannot fit in the memory.

# It Involves sorting both tables based on the join key and then merging them.

# It provides good performance for certain types of queries but requires sorting, which can be computationally expensive.

# Shuffle: The data from both tables is partitioned based on the join key. This partitioning ensures that records with the same join key are directed to the same partition.

# Sort: Within each partition, the data is then sorted based on the join key.

# Merge: The sorted data is subsequently merged across partitions to execute the join operation.

sort\_merge\_join\_df = df\_clients.hint("MERGE").join(df\_orders, on = "client\_id", how = "inner")

sort\_merge\_join\_df.show()

#Cartesian Product Join

# It involves joining every row from the first table with every row from the second table, making it highly resource-intensive.

# This strategy should be avoided for large datasets due to a significant increase in the number of records.

cartesian\_product\_join\_df = df\_clients.hint("SHUFFLE\_REPLICATE\_NL").join(df\_orders, on = "client\_id", how = "inner")

cartesian\_product\_join\_df.show()

#Broadcast Nested Loop Join

# It’s useful when joining a large table with a small table that doesn’t fit in memory but has a filter condition.

# The smaller table is broadcasted, and a nested loop is used to join matching records.

spark.conf.set("spark.sql.crossJoin.enabled", True)

broadcast\_nested\_loop\_join\_df = df\_clients.hint("BROADCAST").join(df\_orders)

broadcast\_nested\_loop\_join\_df.show()

*Date Functions*

#pyspark initialization

import findspark

findspark.init()

#building spark session

import pyspark

import pyspark.sql.functions as sqlfunc

from pyspark.sql.functions import \*

from pyspark.sql import SparkSession

spark = SparkSession.builder.master('local[\*]').getOrCreate()

spark

#DataFrame creation

data = [

(0, '2025-02-11'),

(1, '2025-02-10'),

(2, '2025-02-09'),

(3, '2025-02-08'),

(4, '2025-02-07')]

columns = ["id", "date\_yyyy\_MM\_dd\_string"]

df = spark.createDataFrame(data, columns)

df.show()

#date functions

df\_1 = df.withColumn('date\_yyyy\_MM\_dd\_date', to\_date(df.date\_yyyy\_MM\_dd\_string, 'yyyy-MM-dd'))

df\_2 = df\_1.withColumn('date\_MM\_dd\_yyyy\_string', date\_format(df\_1.date\_yyyy\_MM\_dd\_date, 'MM-dd-yyyy'))

df\_3 = df\_2.withColumn('current\_date\_yyyy\_MM\_dd', current\_date())

df\_4 = df\_3.withColumn('current\_date\_add\_10', date\_add(df\_3.current\_date\_yyyy\_MM\_dd, 10))

df\_5 = df\_4.withColumn('current\_date\_sub\_10', date\_sub(df\_3.current\_date\_yyyy\_MM\_dd, 10))

df\_6 = df\_5.withColumn('date\_difference', datediff(df\_5.current\_date\_add\_10, df\_5.current\_date\_sub\_10))

df\_6.show()

df\_6.printSchema()

*Solved Questions*

#pyspark initialization

import findspark

findspark.init()

#building spark session

import pyspark

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[\*]").getOrCreate()

#reading CSV

df = spark.read.format("csv").option("header", "true").load("test\_data.csv")

df.show()

#groupby 'CGI\_Location' to get concatenated list of employees

import pyspark.sql.functions as sqlfunc

employee\_location = df.groupBy('CGI\_Location').agg(sqlfunc.concat\_ws(', ', sqlfunc.collect\_set(df.Employee\_Name)).alias('Employees'))

employee\_location.show()

#sorting 'CGI\_Location'

from pyspark.sql.functions import \*

employee\_location = employee\_location.sort(col('CGI\_Location').asc())

employee\_location.show()

#DataFrame creation to demonstate 'getItem' method

data = [

("Amit", [1, 2, 3]),

("Prashant", [4, 5, 6])]

columns = ["name", "numbers"]

df = spark.createDataFrame(data, columns)

df.show()

#importing functions

import pyspark.sql.functions as F

from pyspark.sql.functions import \*

#demonstrating 'getItem'

df = df.withColumn("first\_number", col("numbers").getItem(0))

df = df.withColumn("second\_number", col("numbers").getItem(1))

df = df.withColumn("third\_number", col("numbers").getItem(2))

df.show()

#DataFrame creation to demonstrate 'collect\_list' and 'collect\_set'

customized\_data = [

("cust\_1", "acc\_1"),

("cust\_2", "acc\_2"),

("cust\_3", "acc\_3"),

("cust\_4", "acc\_4"),

("cust\_1", "acc\_1"),

("cust\_2", "acc\_2"),

("cust\_3", "acc\_3"),

("cust\_4", "acc\_4"),

("cust\_1", "acc\_1"),

("cust\_2", "acc\_2")]

columns = ["customer\_id", "account\_type"]

df = spark.createDataFrame(customized\_data, columns)

df.show()

#demonstrating 'collect\_list'

df\_grouped\_collect\_list = df.groupby("customer\_id").agg(sqlfunc.concat\_ws(", ", collect\_list(df.account\_type)).alias("concat\_account\_type"))

df\_grouped\_collect\_list.show()

#demonstrating 'collect\_set'

df\_grouped\_collect\_set = df.groupby("customer\_id").agg(sqlfunc.concat\_ws(", ", collect\_set(df.account\_type)).alias("concat\_account\_type"))

df\_grouped\_collect\_set.show()

*Core Concepts*

#initializing pyspark

import findspark

findspark.init()

#building sparksession

import pyspark

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[\*]").getOrCreate()

spark

#reading CSV with syntax\_1

df = spark.read.format("csv").option("header", "true").load("original.csv")

df.show()

#reading CSV with syntax\_2

df1 = spark.read.csv("original.csv", header=True)

df1.show()

#printing datatypes of a DataFrame

df1.dtypes

#customized schema

from pyspark.sql.types import \*

myschema = StructType([

StructField("id", IntegerType()),

StructField("first\_name", StringType()),

StructField("last\_name", StringType()),

StructField("gender", StringType()),

StructField("City", StringType()),

StructField("JobTitle", StringType()),

StructField("Salary", StringType()),

StructField("Latitude", StringType()),

StructField("Longitude", FloatType())

])

df2 = spark.read.csv("original.csv", header=True, schema=myschema)

df2.show()

#printing datatypes of a DataFrame with customized schema

df2.dtypes

#printing top 10 rows

df2.head(10)

#printing headers with schema

df2.first

#printing basic statistical parameters of columns

df2.describe().show()

#printing columns

df2.columns

#printing no. of rows

df2.count()

#printing no. of distinct rows

df2.distinct().count()

#dropping 'NA' values

df2\_dropped = df2.na.drop()

df2\_dropped.show(5)

#extracting data for which 'JobTitle' is NotNull

df2\_null\_jobs = df2.filter(df2.JobTitle.isNotNull())

df2\_null\_jobs.show(5)

#replacing NULL values with 'Unknown' keyword for 'City' column

from pyspark.sql.functions import \*

df2\_handled = df2.withColumn("clean\_city", when(df2.City.isNull(), "Unknown").otherwise(df2.City))

df2\_handled.show(5)

#dropping duplicates

df2\_no\_duplicates = df2.dropDuplicates()

df2\_no\_duplicates.show(5)

#selecting specific columns

df2\_select = df2.select("first\_name", "last\_name")

df2\_select.show(5)

#renaming column

df2\_renamed = df2.withColumnRenamed("first\_name", "fn")

df2\_renamed.show(5)

#filtering specific row w.r.t 'last\_name'

df2\_filter = df2.filter(df2.last\_name == "Von Welden")

df2\_filter.show()

#filtering rows w.r.t the presence of a phrase in 'first\_name' column

# df2\_filter = df2.filter(df2.first\_name.like("%era")) #ending with 'era'

# df2\_filter = df2.filter(df2.first\_name.like("Kim%")) #starting with 'Kim'

df2\_filter = df2.filter(df2.first\_name.like("%era%")) #contains 'era'

df2\_filter.show()

#filtering rows for which 'first\_name' column ends with 'din'

df2\_filter = df2.filter(df2.first\_name.endswith("din"))

df2\_filter.show(5)

#filtering rows for which 'first\_name' column starts with 'Alv'

df2\_filter = df2.filter(df2.first\_name.startswith("Alv"))

df2\_filter.show(5)

#filtering rows for which 'id' column contains values in-between 100 and 110

df2\_filter = df2.filter(df2.id.between(100, 110))

df2\_filter.show()

#filtering rows for which 'first\_name' column contains 'Dorisa'

df2\_filter = df2.filter(df2.first\_name.isin("Dorisa"))

df2\_filter.show()

#extracting a substring from 'first\_name' column

df2\_substr = df2.select(df2.first\_name, df2.first\_name.substr(1, 5).alias("name"))

df2\_substr.show()

#filtering with OR condition

# df2\_filter = df2.filter(df2.first\_name.isin("Alene", "Aime"))

# df2\_filter = df2.filter(df2.City.like("%Pedra"))

df2\_filter = df2.filter(df2.first\_name.isin("Alene", "Aime") | (df2.City.like("%la Baja")))

df2\_filter.show()

#filtering with 'and' condition

df2\_filter = df2.filter((df2.id >= 10) & (df2.id <= 20))

df2\_filter.show()

#registering a temporary table

df2.registerTempTable("Original")

#SQL-query\_1

query\_1 = spark.sql(

'select \*\

from original')

query\_1.show(5)

#SQL-query\_2

query\_2 = spark.sql(

'select concat(first\_name, " ", last\_name) as full\_name\

from original\

where gender = "Male"')

query\_2.show()

#removing $-sign from 'Salary' column and creating a new column

df2\_clean\_salary = df2.withColumn("clean\_salary", df2.Salary.substr(2, 100).cast('float'))

df2\_clean\_salary.show(5)

#creating a new column for 'monthly\_salary'

df2\_monthly\_salary = df2\_clean\_salary.withColumn('monthly\_salary', df2\_clean\_salary.clean\_salary/12)

df2\_monthly\_salary.show()

#creating a new column 'are\_they\_female?'

df2\_are\_female = df2.withColumn("are\_they\_female?", when(df2.gender == "Female", "Yes").otherwise("No"))

df2\_are\_female.show(10)

#creating a new column 'are\_they\_male?'

df2\_are\_male = df2.withColumn("are\_they\_male?", when(df2.gender == "Male", "Yes").otherwise("No"))

df2\_are\_male.show(10)

#creating a new column 'clean\_salary' by removing $-sign from 'Salary' column

df2 = df2.withColumn("clean\_salary", df2.Salary.substr(2, 100).cast("float"))

df2.show(10)

#applying 'groupby' condition for gender-wise total\_salary

import pyspark.sql.functions as sqlfunc

df2\_groupby = df2.groupby("gender").agg(sqlfunc.sum(df2.clean\_salary).alias("total\_salary"))

df2\_groupby.show()

#applying 'groupby' condition for gender-wise total\_salary, average\_salary, min\_salary and max\_salary

df2\_groupby\_additional = df2.groupby("gender").agg(sqlfunc.sum(df2.clean\_salary).alias("total\_salary"),

sqlfunc.avg(df2.clean\_salary).alias("avg\_salary"),

sqlfunc.min(df2.clean\_salary).alias("min\_salary"),

sqlfunc.max(df2.clean\_salary).alias("max\_salary"))

df2\_groupby\_additional.show()

#applying 'groupby' condition for gender-wise and city-wise total\_salary, average\_salary, min\_salary and max\_salary

df2\_groupby\_additional\_1 = df2.groupby("gender", "City").agg(sqlfunc.sum(df2.clean\_salary).alias("total\_salary"),

sqlfunc.avg(df2.clean\_salary).alias("avg\_salary"),

sqlfunc.min(df2.clean\_salary).alias("min\_salary"),

sqlfunc.max(df2.clean\_salary).alias("max\_salary"))

df2\_groupby\_additional\_1.show()

*Challenges*

#initializing pyspark

import findspark

findspark.init()

#building sparksession

import pyspark

import pyspark.sql.functions as sqlfunc

from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local[\*]").getOrCreate()

spark

#reading CSV

df = spark.read.csv("challenge.csv", header = True)

df.show(5)

#creating a new column 'is\_country\_mexico?'

from pyspark.sql.functions import \*

df\_1 = df.withColumn("is\_country\_mexico?", when(df.Country == "Mexico", "Yes").otherwise("No"))

df\_1.show(50)

#applying 'groupby' condition on 'is\_country\_mexico' to get the total\_Bytes\_used

df\_2 = df\_1.groupby("is\_country\_mexico?").agg(sqlfunc.sum(df\_1.Bytes\_used).alias("total\_used"))

df\_2.show()

#applying 'groupby' condition on 'Country' to get the distinct\_count\_of\_ip\_addresses in descending order

df\_3 = df\_1.groupby('Country').agg(sqlfunc.countDistinct(df\_1.ip\_address).alias('distinct\_number\_of\_ip\_address'))

df\_3.sort(col("distinct\_number\_of\_ip\_address").desc()).show()