Spark with DataBricks

https://github.com/luiscoco/Spark_with_DataBricks

https://github.com/rockthejvm/spark-essentials

1. Welcome

Apache Spark es un **framework de procesamiento de datos distribuidos** diseñado para ser rápido y de propósito general. Consta de **diferentes APIs y módulos** que permiten que sea utilizado por una gran variedad de profesionales en todas las etapas del ciclo de vida del dato.

1.1. DataBricks Community

https://community.cloud.databricks.com/

1.2. Scala Recap

SCAlable **LA**nguage: diseñado para escalar. Lenguaje híbrido entre programación orientada a objetos (OOP) y programación funcional (FP).

```
import scala.concurrent.{ExecutionContext, Future}
import scala.util.{Failure, Success}

object ScalaRecap extends App {
```

```scala
// values and variables
val aBoolean: Boolean = false

// expressions
val anIfExpression = if(2 > 3) "bigger" else "smaller"

// instructions vs expressions
val theUnit = println("Hello, Scala") // Unit = "no meaningful value" = void in other langua

// functions
def myFunction(x: Int): Int = 42

// OOP
class Animal
class Cat extends Animal
trait Carnivore {
  def eat(animal: Animal): Unit
}

class Crocodile extends Animal with Carnivore {
  override def eat(animal: Animal): Unit = println("Crunch!")
}

// singleton pattern
object MySingleton

// companions
object Carnivore

// generics
trait MyList[A]

// method notation
val x = 1 + 2
val y = 1.+(2)

// Functional Programming
val incrementer: Int => Int = x => x + 1
val incremented = incrementer(42)

// map, flatMap, filter
val processedList = List(1,2,3).map(incrementer)

// Pattern Matching
val unknown: Any = 45
val ordinal = unknown match {
  case 1 => "first"
  case 2 => "second"
  case _ => "unknown"
}

// try-catch
try {
```

```
  throw new NullPointerException
} catch {
  case _: NullPointerException => "some returned value"
  case _: Throwable => "something else"
}

// Future
import ExecutionContext.Implicits.global
val aFuture = Future {
  // some expensive computation, runs on another thread
  42
}

aFuture.onComplete {
  case Success(meaningOfLife) => println(s"I've found $meaningOfLife")
  case Failure(ex) => println(s"I have failed: $ex")
}

// Partial functions
val aPartialFunction: PartialFunction[Int, Int] = {
  case 1 => 43
  case 8 => 56
  case _ => 999
}

// Implicits

// auto-injection by the compiler
def methodWithImplicitArgument(implicit x: Int): Int = x + 43
implicit val implicitInt: Int = 67
val implicitCall = methodWithImplicitArgument

// implicit conversions - implicit defs
case class Person(name: String) {
  def greet: Unit = println(s"Hi, my name is $name")
}

implicit def fromStringToPerson(name: String): Person = Person(name)
"Bob".greet // fromStringToPerson("Bob").greet

// implicit conversion - implicit classes
implicit class Dog(name: String) {
  def bark: Unit = println("Bark!")
}
"Lassie".bark

/*
  - local scope
  - imported scope
  - companion objects of the types involved in the method call
 */
```

```
}
```

1.3. Spark First Principles

What Spark is?

Unified computing engine and libraries for ditributed data processing.

Unified computing engine

Spark supports a variety of data processing tasks: data loading, SQL queries, machine learning, streaming.

Unified: consistent, composable APIs in multiple languages, optimizations across different libraries.

Computing engine detached from data storage and I/O.

Libraries:

Standard: Spark SQL, MLib, Streaming, GraphX.

Hundreds of open-source third-party libraries.

Context of Big Data

Computing vs Data: data storage getting better and cheaper; gathering data keeps easier and cheaper and more important

Data needs to be distributed and processed in parallel

Standard single-CPU software cannot scale up

Motivation for Spark

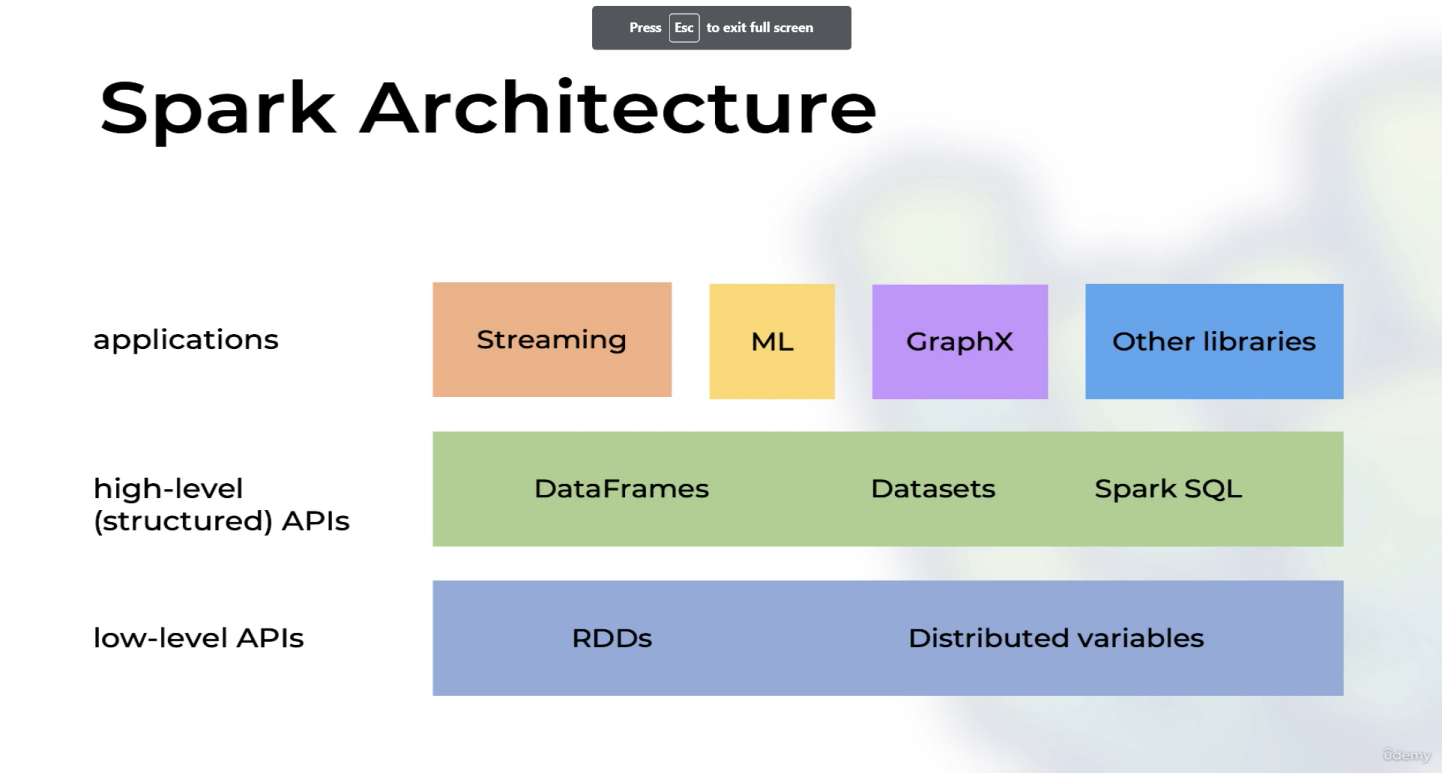
A 2009 UC Berkeley project by Matei Zaharia et al

MapReduce was the king of large distributed computation

Spark phase 1. A simple functional programming API. Optimize multi-step applications. In-memory computation and data sharing across nodes.

Spark phase 2. Interactive data science and ad-hoc computation. Spark shell and Spark SQL.

Spark phase 3. Same engine, new libraries. ML, Streaming, GraphX.



2. Spark Structured and API: DataFrames

2.1. DataFrames Basics

In Scala Spark, a DataFrame is a distributed collection of data organized into named columns.

It is similar to a table in a relational database or a data frame in R/Python.

Spark DataFrames provide a higher-level API compared to RDDs, making it easier to perform data manipulation tasks.

```
%scala
// Import SparkSession
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

// Create a SparkSession
val spark = SparkSession.builder
  .appName("DataFrame Basics")
  .getOrCreate()

// Create a simple DataFrame with some data
val data = Seq(("Alice", 25), ("Bob", 30), ("Charlie", 35))
val columns = Seq("Name", "Age")

import spark.implicits._
val df = data.toDF(columns: _*)
```

```
// Show the DataFrame
df.show()

// Select a specific column
val nameColumn = df("Name")

// Instead of nameColumn.show(), use df.select("Name").show()
df.select("Name").show()

// Filter the DataFrame based on a condition
val filteredDF = df.filter($"Age" > 30)
filteredDF.show()

// Perform some aggregation (e.g., calculate the average age)
val avgAge = df.agg(avg($"Age").as("AverageAge"))
avgAge.show()

// Join two DataFrames
val otherData = Seq(("Alice", "Engineer"), ("Bob", "Doctor"))
val otherColumns = Seq("Name", "Occupation")
val otherDF = otherData.toDF(otherColumns: _*)

val joinedDF = df.join(otherDF, "Name")
joinedDF.show()

// Stop the SparkSession
spark.stop()
```

```
+-------+---+
|   Name|Age|
+-------+---+
|  Alice| 25|
|    Bob| 30|
|Charlie| 35|
+-------+---+

+-------+
|   Name|
+-------+
|  Alice|
|    Bob|
|Charlie|
+-------+

+-------+---+
|   Name|Age|
+-------+---+
|Charlie| 35|
+-------+---+
```

Let me explain what's happening in the code:

a) **Creating SparkSession:** The entry point for Spark functionality.

b) **Creating a DataFrame:** Using a sequence of data and column names.

c) **Showing the DataFrame:** Displaying the content of the DataFrame.

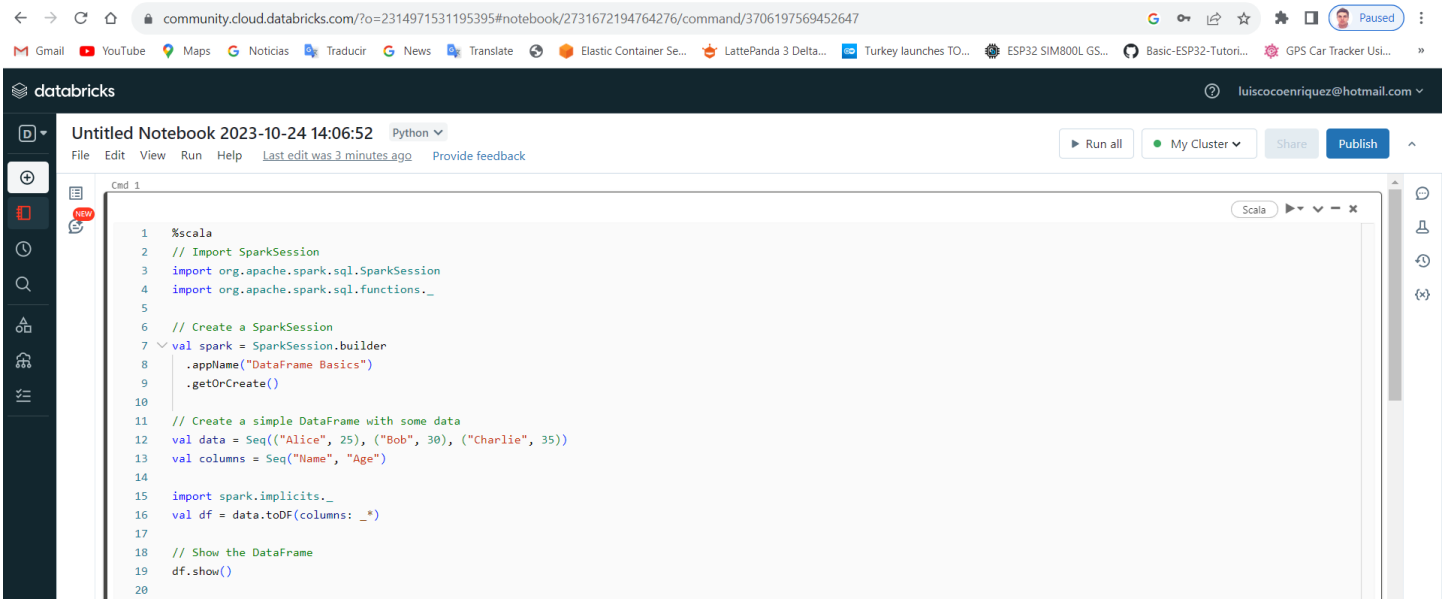
d) **Selecting a Column:** Accessing a specific column.

e) **Filtering Data:** Applying a filter condition.

f) **Aggregation:** Calculating the average age.

g) **Joining DataFrames:** Combining two DataFrames based on a common column.

h) **Stopping SparkSession:** Ending the SparkSession.

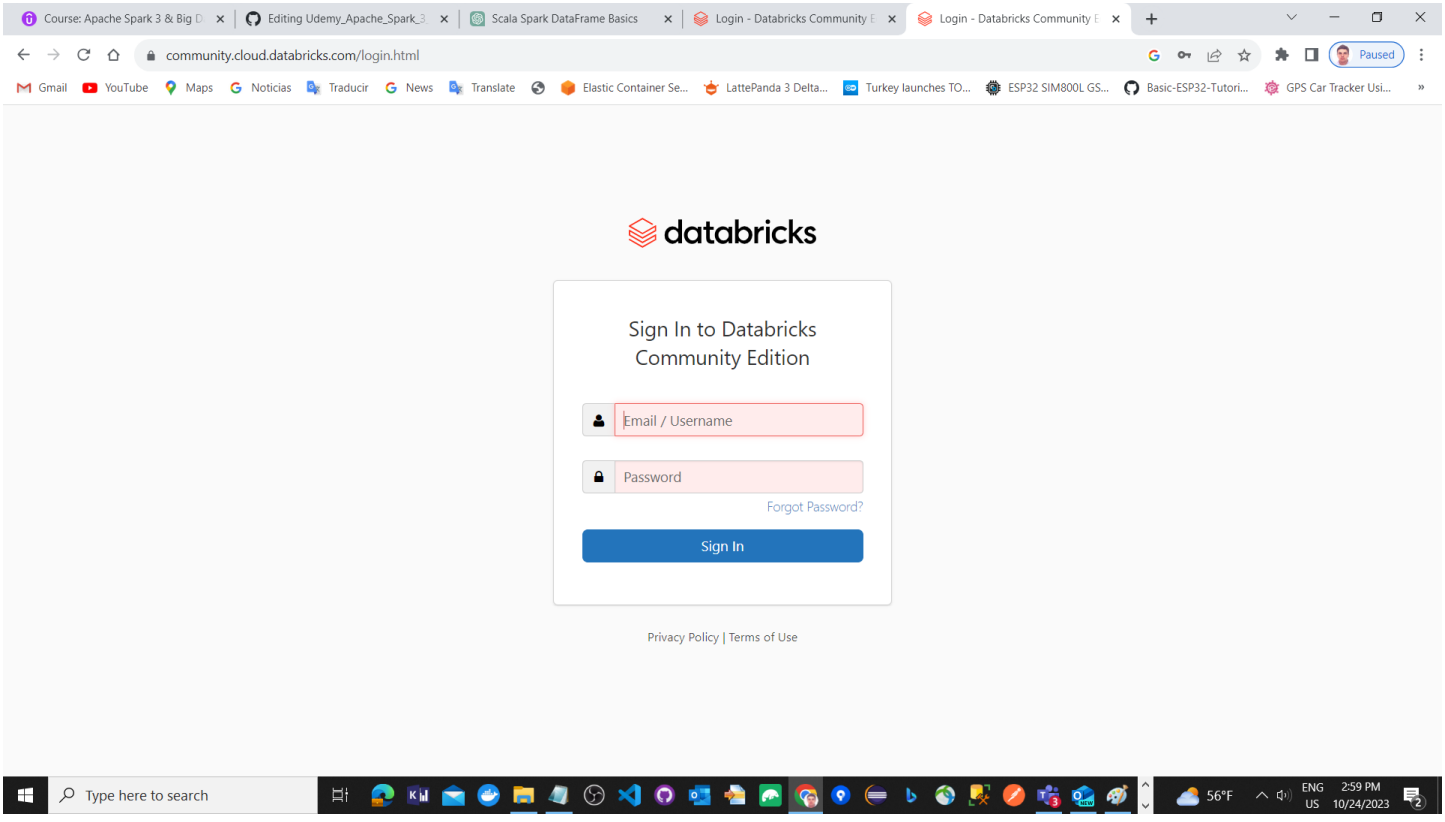


2.2. DataFrames Basics. Exercises

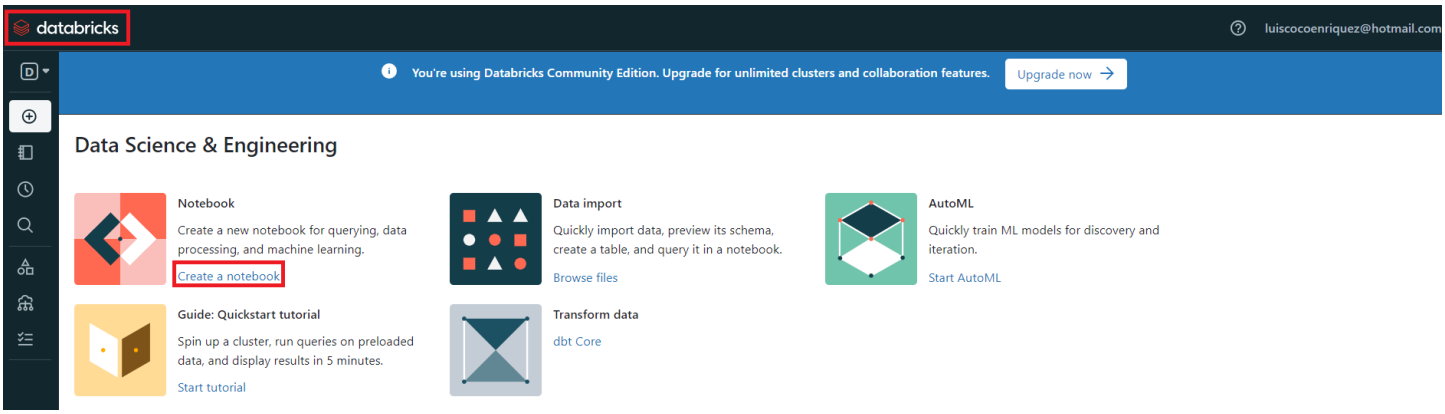
Here are a few more examples of common operations you might perform with DataFrames in Scala Spark:

Reading Data from a File with Scala Spark DataBricks:

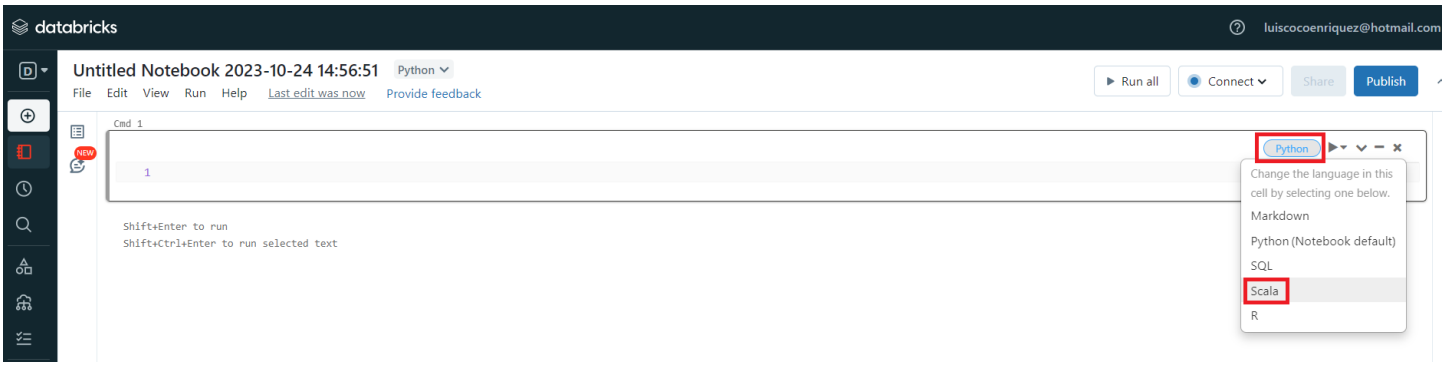
First we login in DataBricks Community edition

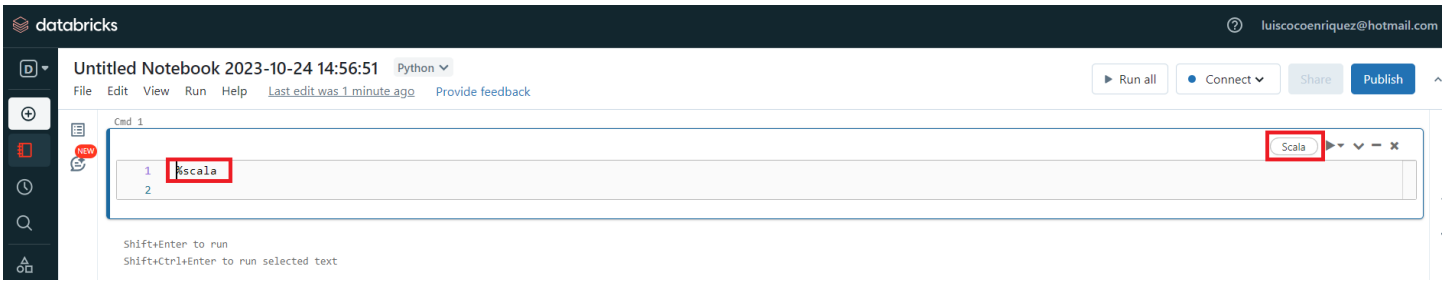


Second we create a new Notebook

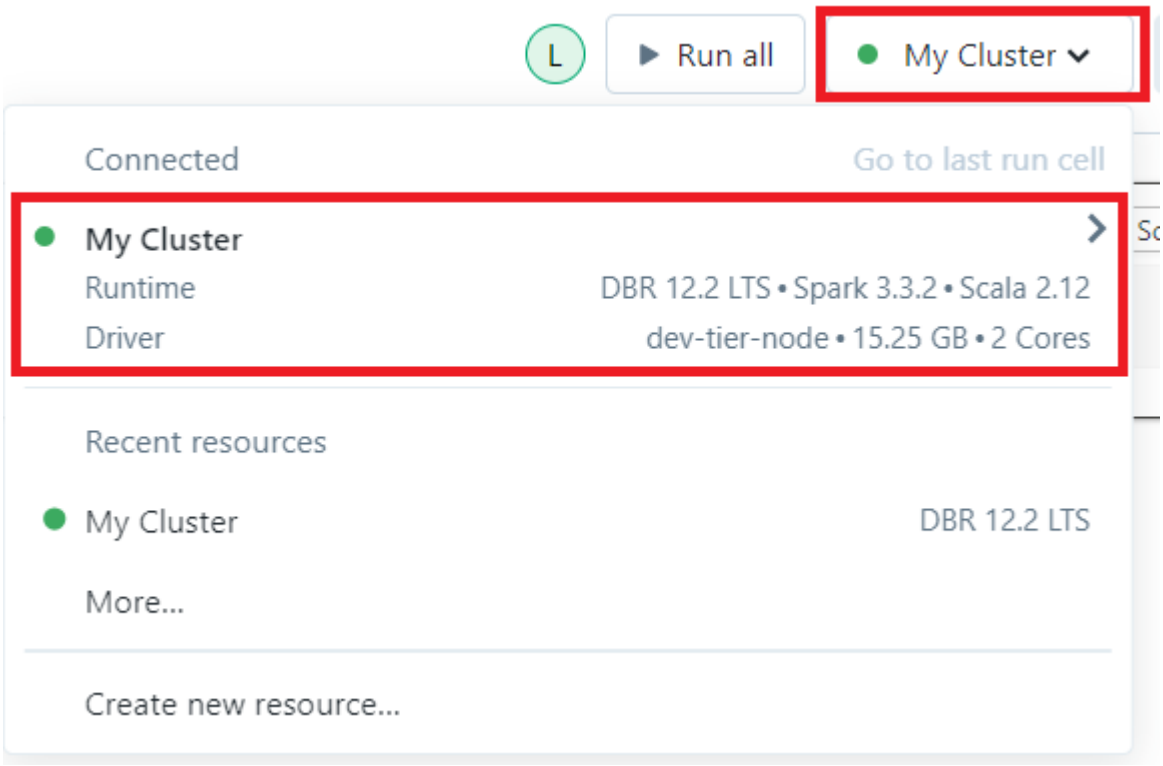


Third step we enter the scala code and we select in the





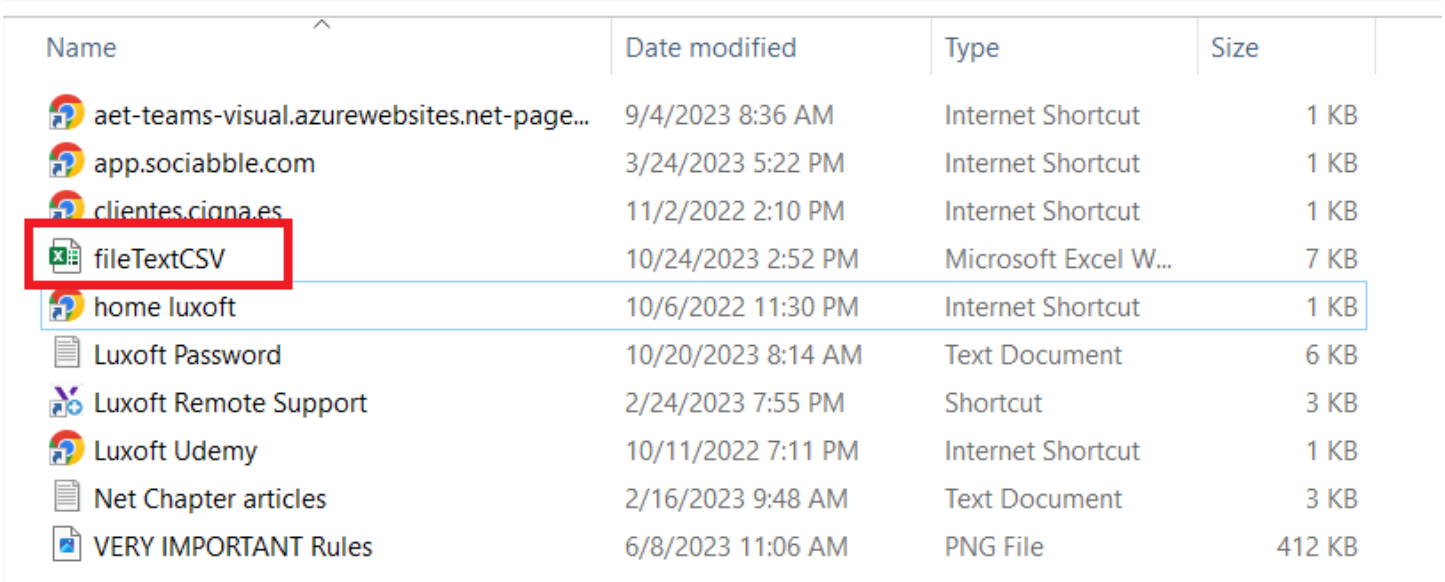
Four step we create a cluster and attach to it

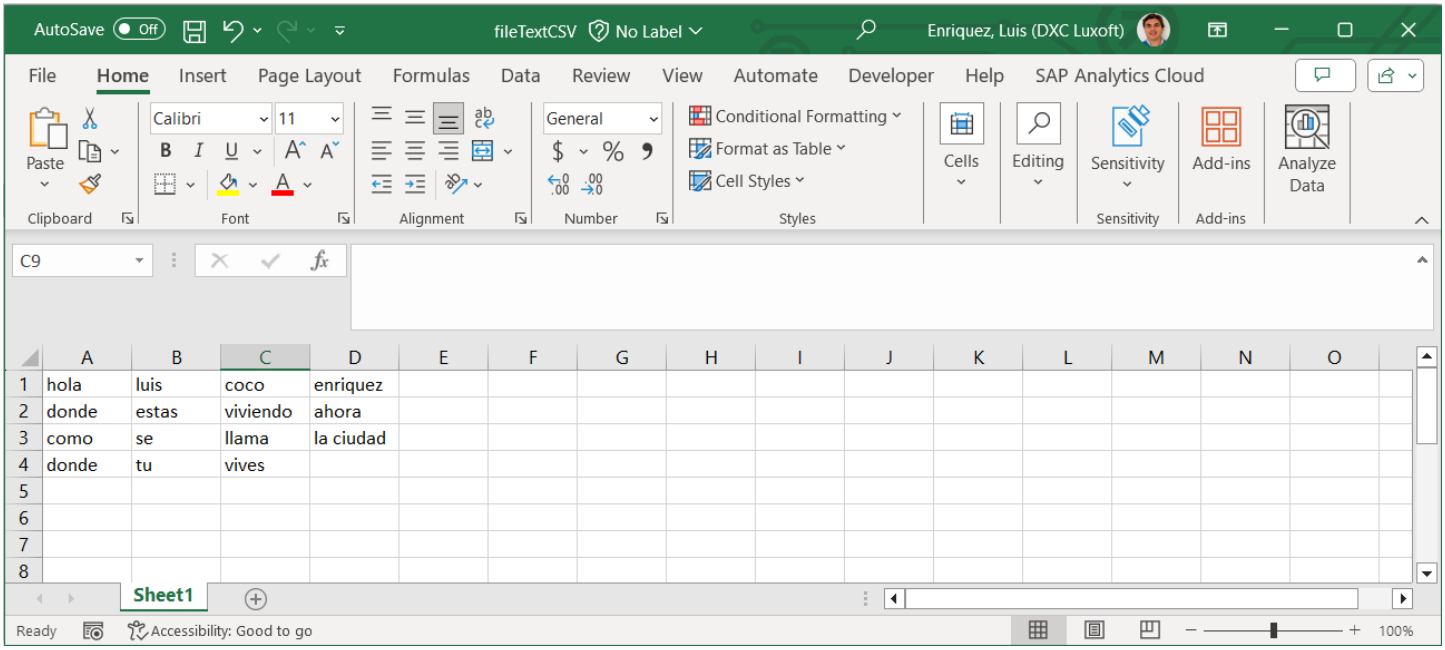


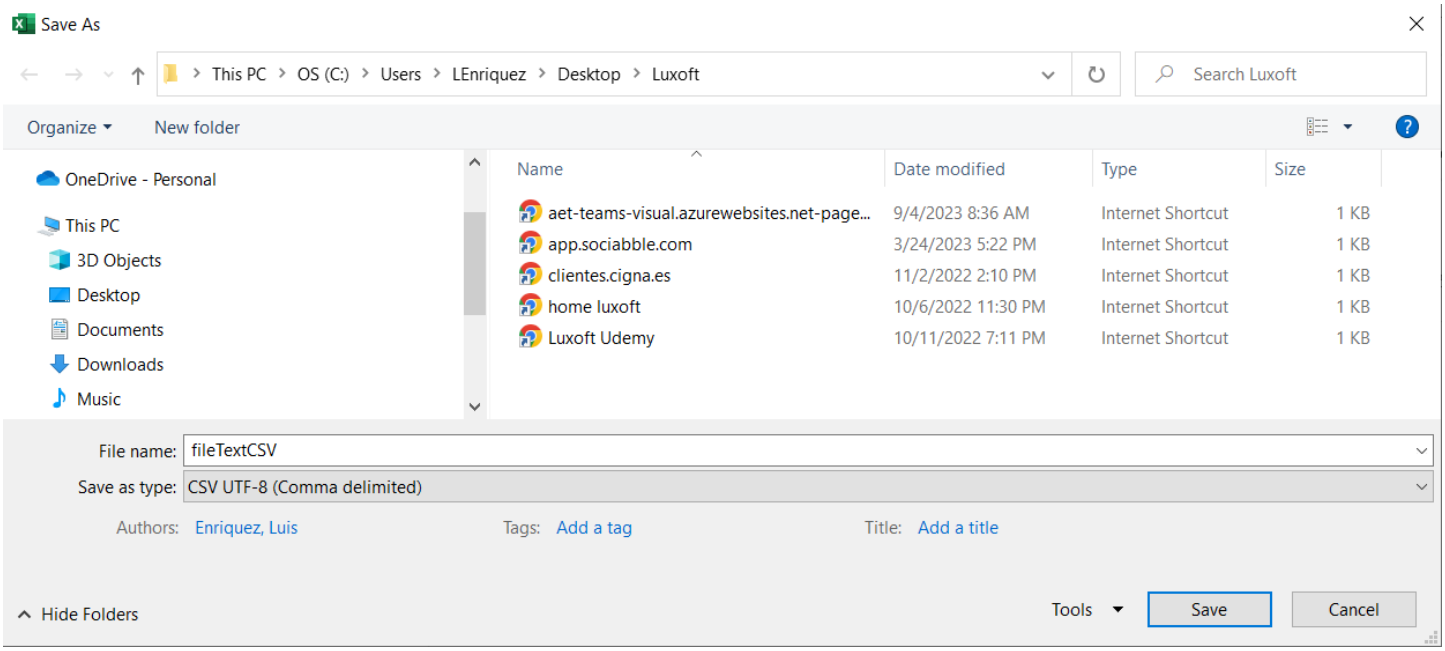
Fifth step we input the scala source code

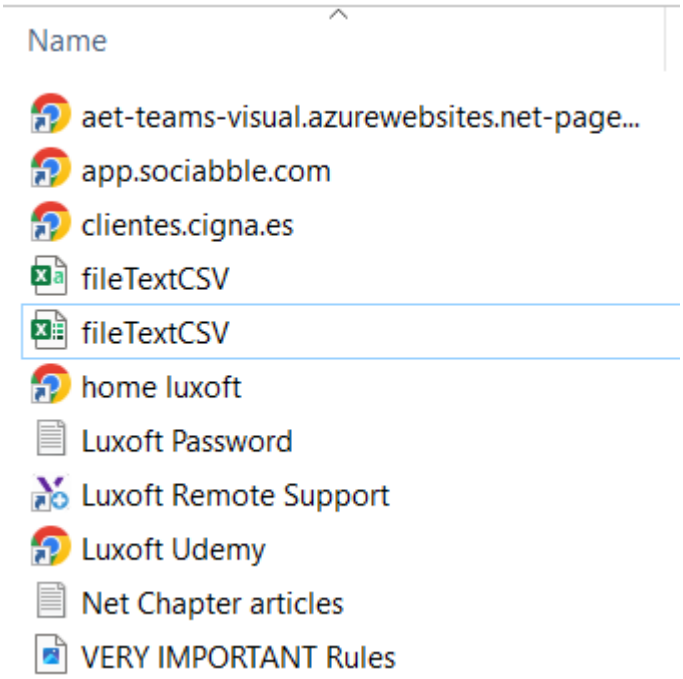
```
%scala
// Read a CSV file into a DataFrame
val csvPath = "/FileStore/tables/fileCSV.csv"  // Specify the correct DBFS path
val csvDF = spark.read.csv(csvPath)

// Show the content of the DataFrame
csvDF.show()
```

Sixth step, we create a CSV file in my local laptop

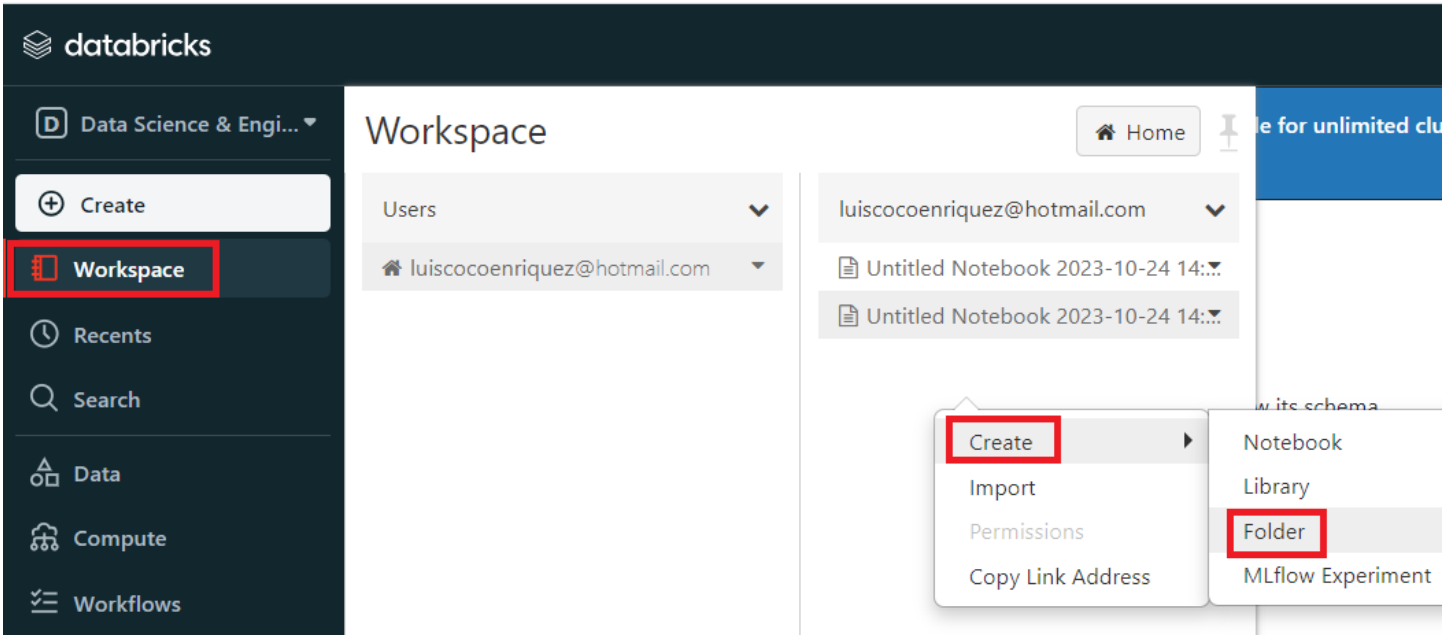



AutoSave Off
fileTextCSV No Label
Enriquez, Luis (DXC Luxoft)
File Home Insert Page Layout Formulas Data Review View Automate Developer Help SAP Analytics Cloud
Paste
Calibri 11
General
Conditional Formatting
Format as Table
Cell Styles
Cells
Editing
Sensitivity
Add-ins
Analyze Data
Clipboard Font Alignment Number Styles Sensitivity Add-ins
C9
A B C D E F G H I J K L M N O
1 hola luis coco enriquez
2 donde estas viviendo ahora
3 como se llama la ciudad
4 donde tu vives
5
6
7
8
Sheet1
Ready
Accessibility: Good to go
100%

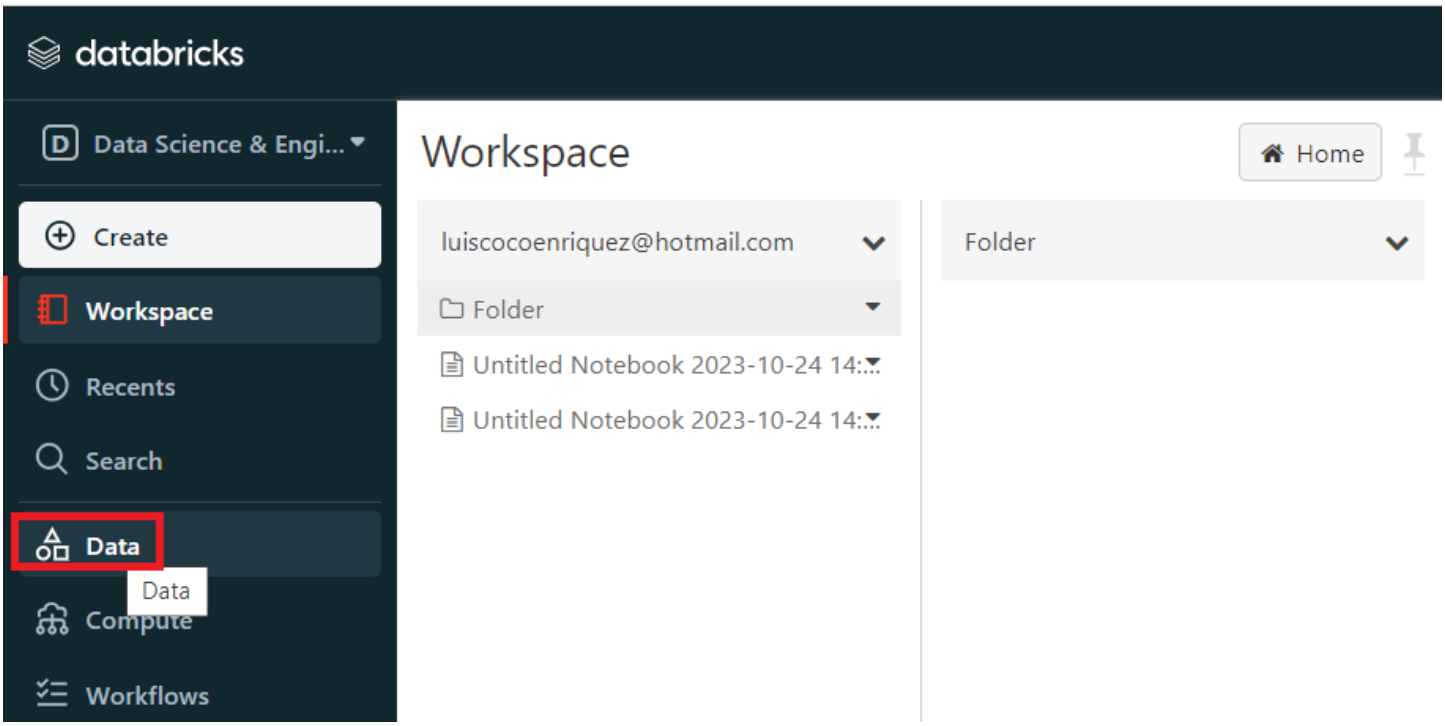

Save As
This PC > OS (C:) > Users > LEnriquez > Desktop > Luxoft
Search Luxoft
Organize
New folder
OneDrive - Personal
This PC
3D Objects
Desktop
Documents
Downloads
Music
Name
Date modified
Type
Size
aet-teams-visual.azurewebsites.net-page... 9/4/2023 8:36 AM Internet Shortcut 1 KB
app.sociabble.com 3/24/2023 5:22 PM Internet Shortcut 1 KB
clientes.cigna.es 11/2/2022 2:10 PM Internet Shortcut 1 KB
home luxoft 10/6/2022 11:30 PM Internet Shortcut 1 KB
Luxoft Udemy 10/11/2022 7:11 PM Internet Shortcut 1 KB
File name: fileTextCSV
Save as type: CSV UTF-8 (Comma delimited)
Authors: Enriquez, Luis
Tags: Add a tag
Title: Add a title
Hide Folders
Tools
Save
Cancel


Name
aet-teams-visual.azurewebsites.net-page...
app.sociabble.com
clientes.cigna.es
fileTextCSV
fileTextCSV
home luxoft
Luxoft Password
Luxoft Remote Support
Luxoft Udemy
Net Chapter articles
VERY IMPORTANT Rules

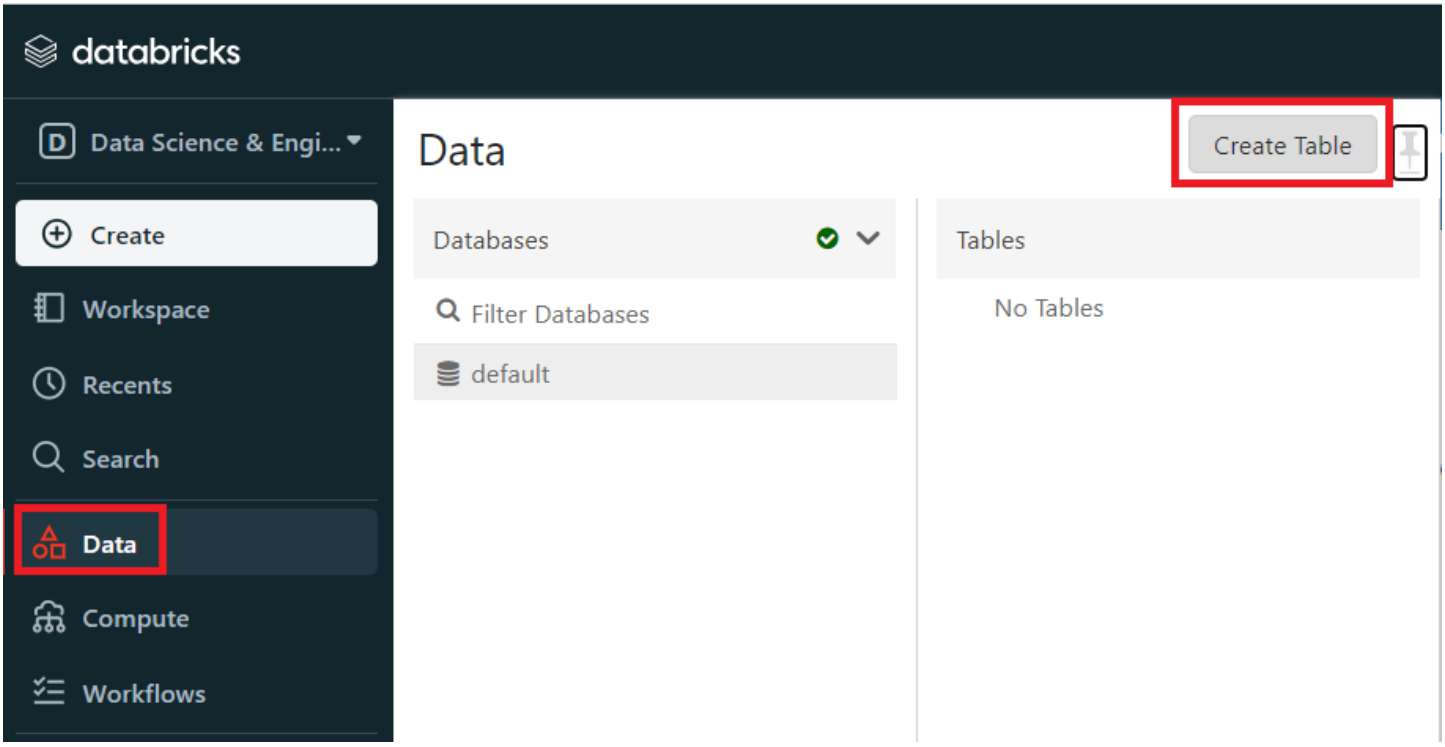
Seven step, click on workspace and create a new folder



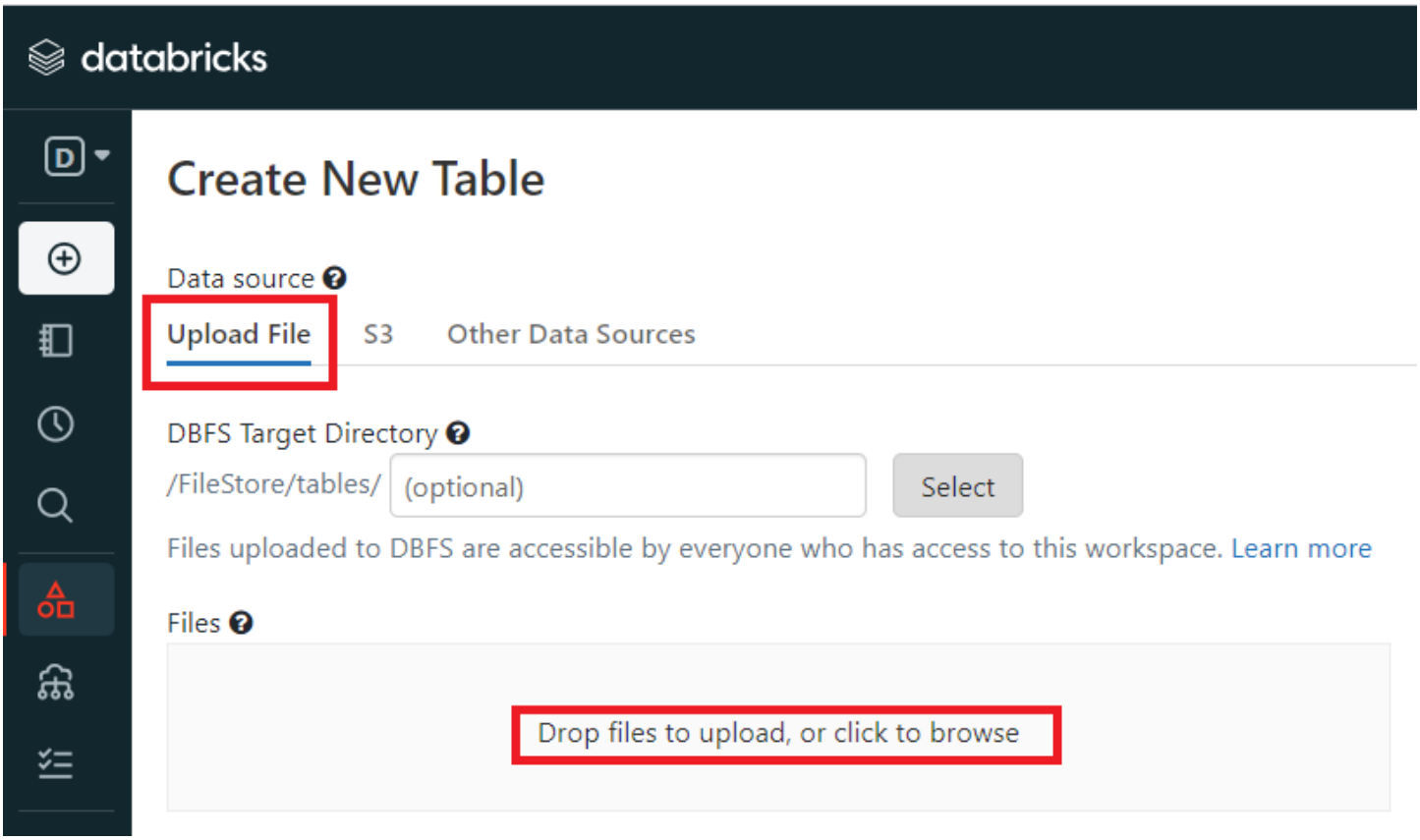
Eight step, click on the folder and then click on the Data button



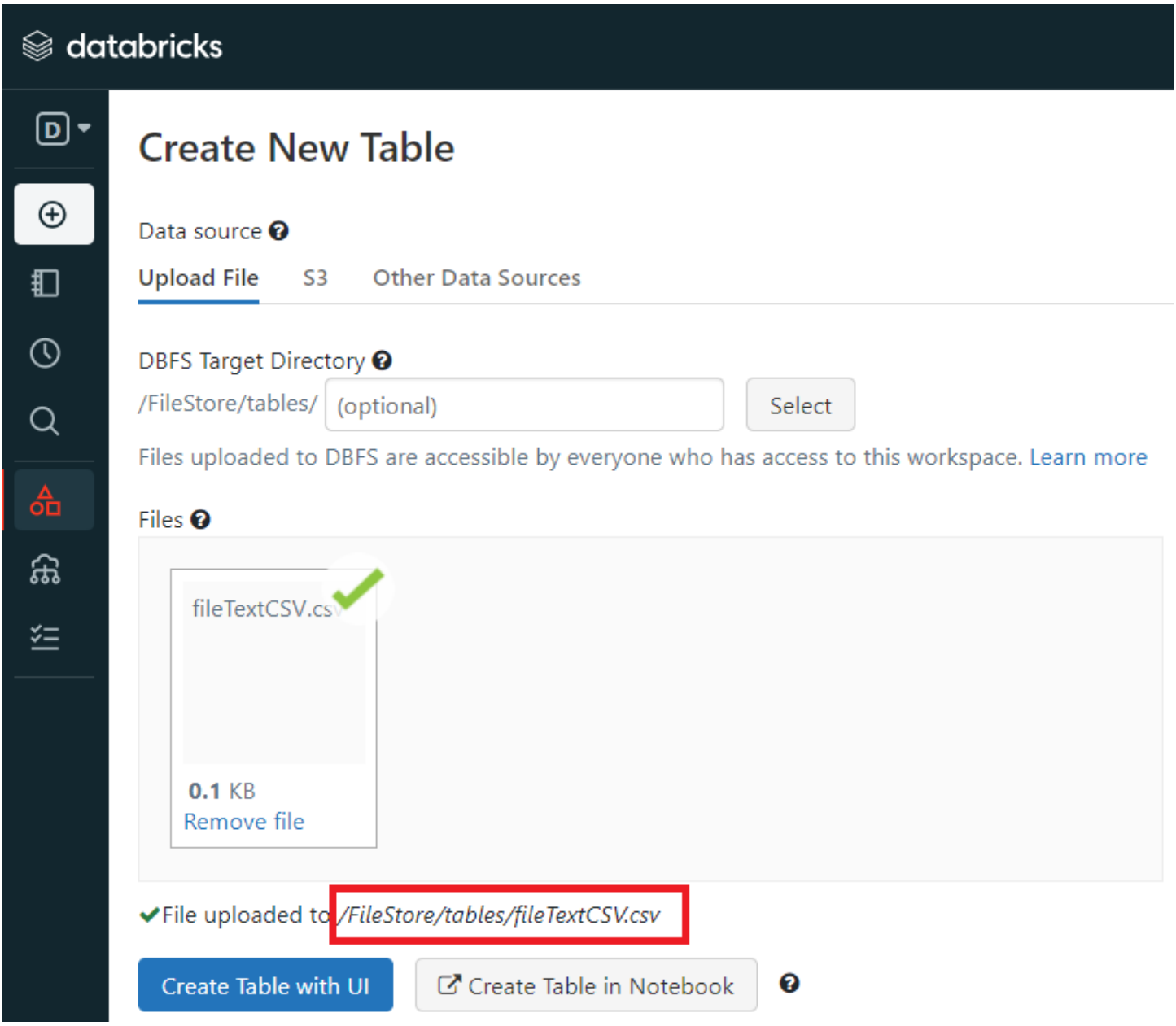
Nine step, press in the "Create Table" button



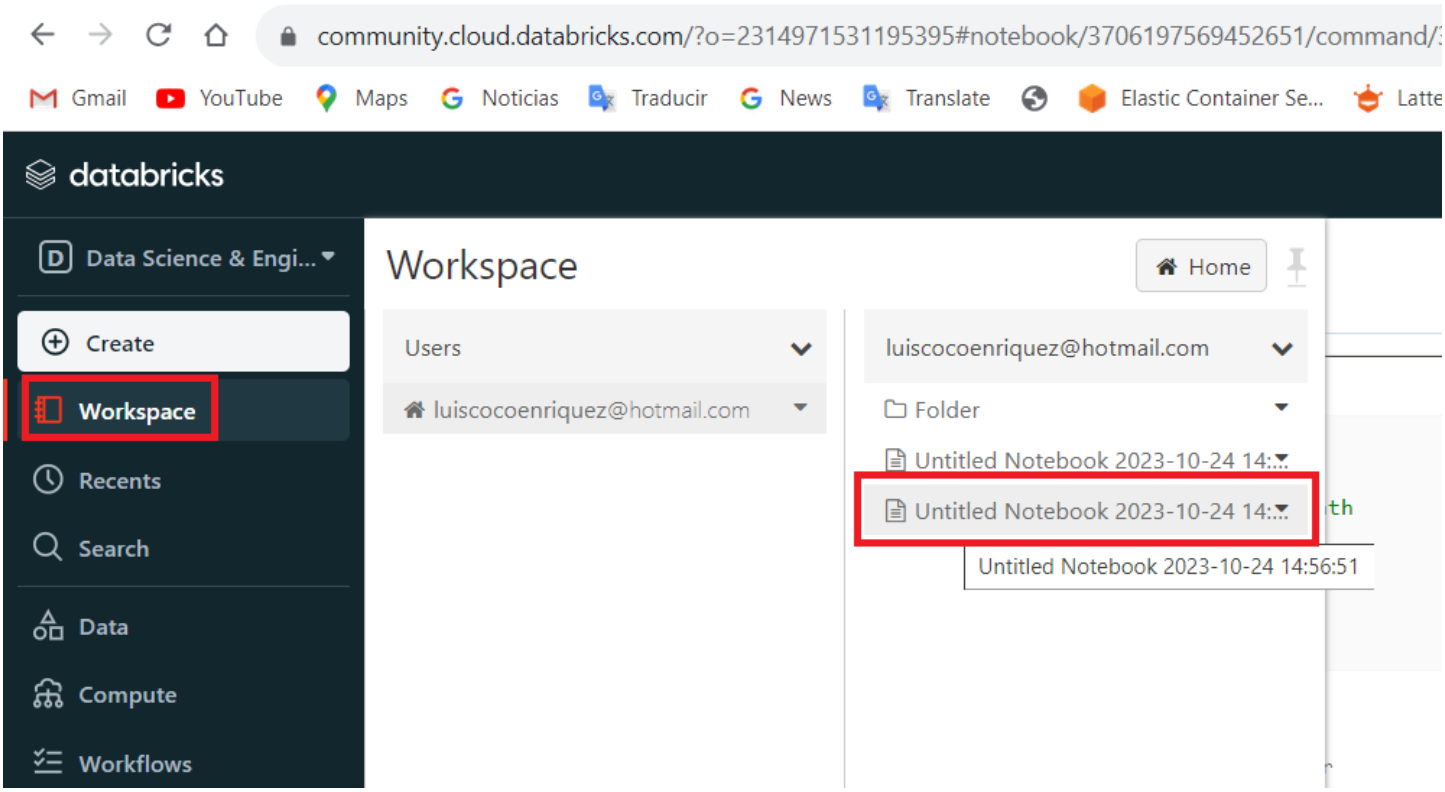
Ten step, we drag and drop the CSV file



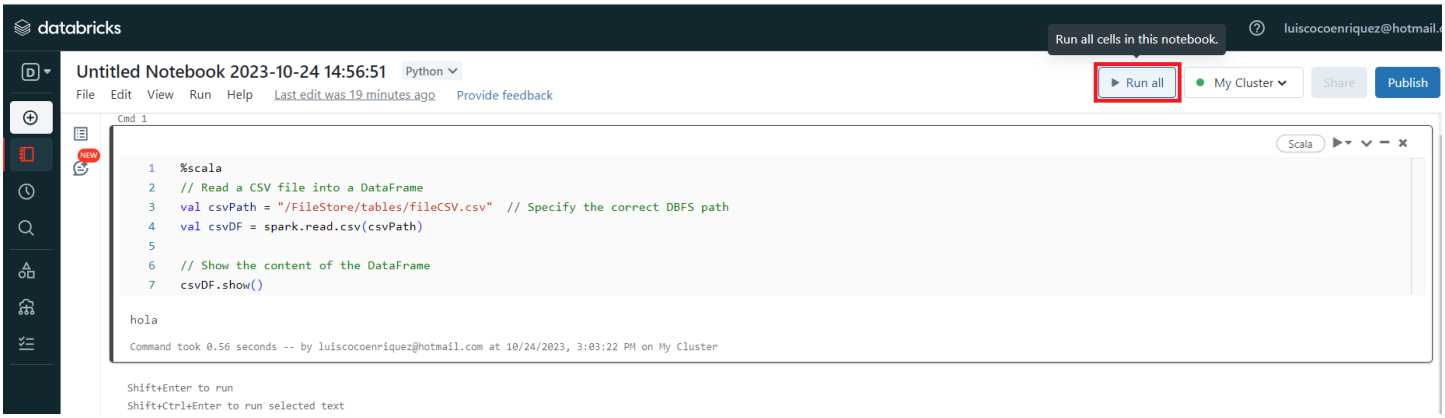
Eleven step, we copy the uploaded file path



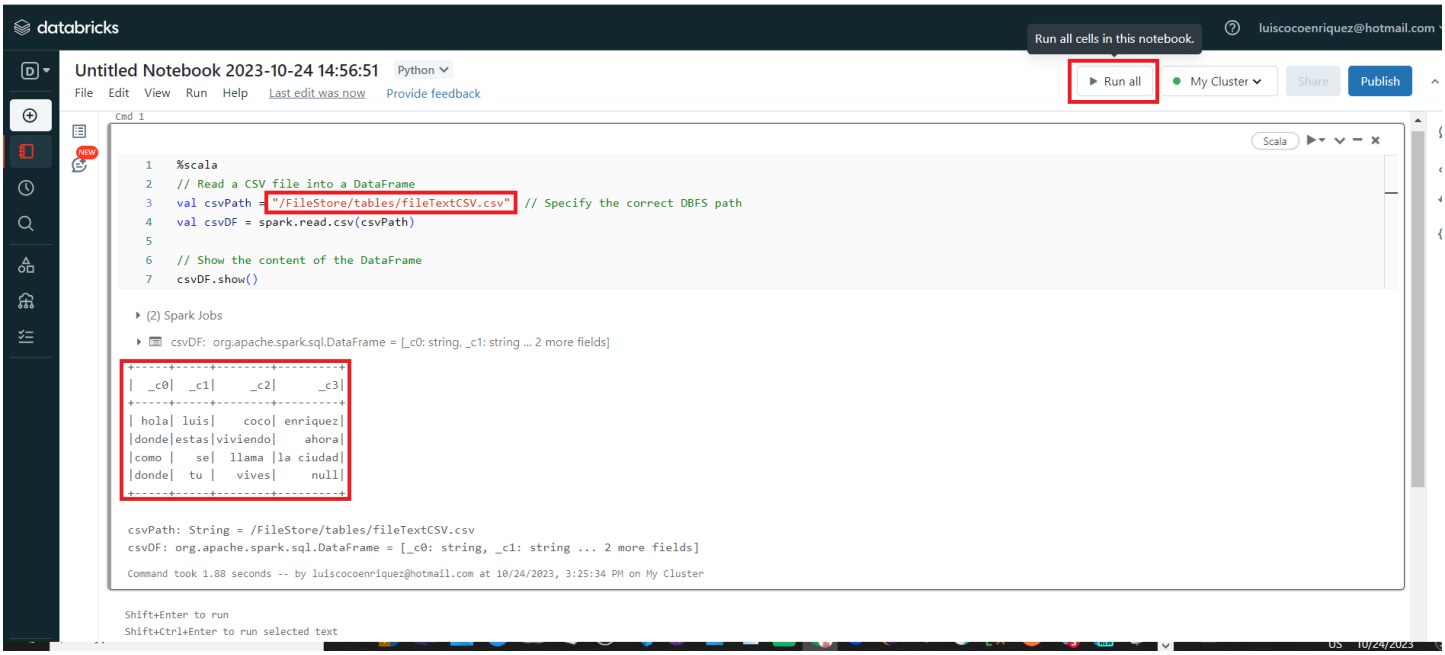
Then we click on Workspace and then we double click on the Notebook we would like to open



Finally we press on the Run button



Then we can see the output result



See other code samples:

```
// Read a CSV file into a DataFrame
val csvPath = "/path/to/your/file.csv"
val csvDF = spark.read.csv(csvPath)

// Read a Parquet file into a DataFrame
val parquetPath = "/path/to/your/file.parquet"
val parquetDF = spark.read.parquet(parquetPath)
```

Writing Data to a File:

```
// Write a DataFrame to a CSV file
val outputCsvPath = "/path/to/your/output.csv"
csvDF.write.csv(outputCsvPath)

// Write a DataFrame to a Parquet file
val outputParquetPath = "/path/to/your/output.parquet"
parquetDF.write.parquet(outputParquetPath)
```

Grouping and Aggregation:

```
// Group by a column and calculate the average age
val groupedDF = df.groupBy("Occupation").agg(avg("Age").as("AverageAge"))
groupedDF.show()
```

Adding a New Column:

```
// Add a new column based on a condition
val newDF = df.withColumn("Status", when($"Age" > 30, "Senior").otherwise("Junior"))
newDF.show()
```

Filtering with SQL Expression:

```
// Filter the DataFrame using SQL expression
val filteredSQLDF = df.filter("Age > 30")
filteredSQLDF.show()
```

Running SQL Queries:

```
Copy code
// Create a temporary SQL table
df.createOrReplaceTempView("people")

// Run a SQL query on the DataFrame
val sqlResult = spark.sql("SELECT * FROM people WHERE Age > 30")
sqlResult.show()
```

Handling Missing Values:

```
// Drop rows with any missing values
val dfWithoutNull = df.na.drop()

// Fill missing values with a specific value
val dfFilled = df.na.fill(0)
```

Exploding Arrays:

```
// Explode a column of arrays into separate rows
val arrayDF = Seq((1, Seq("apple", "orange")), (2, Seq("banana", "grape"))).toDF("id", "fruits
val explodedDF = arrayDF.select($"id", explode($"fruits").as("fruit"))
explodedDF.show()
```

2.3. How DataFrames Work

In Apache Spark, a DataFrame is a distributed collection of data organized into named columns.

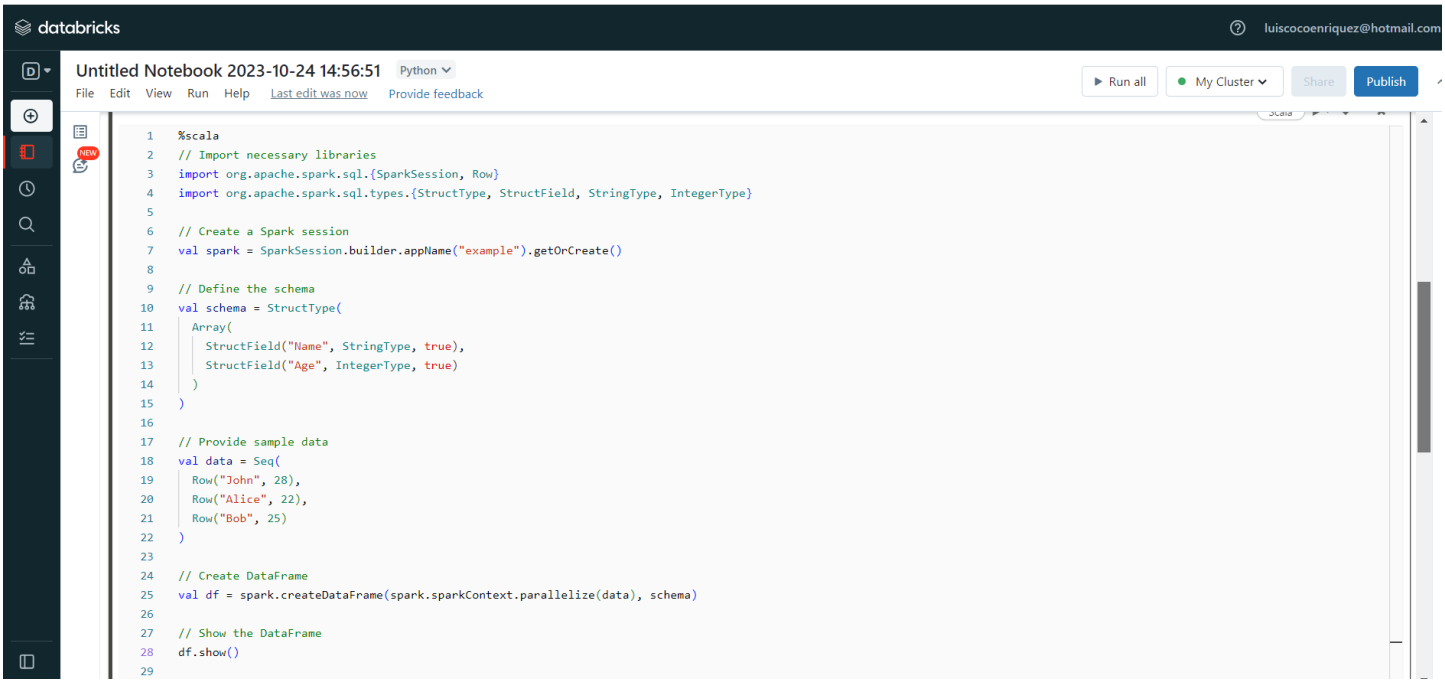
It is conceptually similar to a table in a relational database, or a data frame in R/Python, but with optimizations for distributed computing.

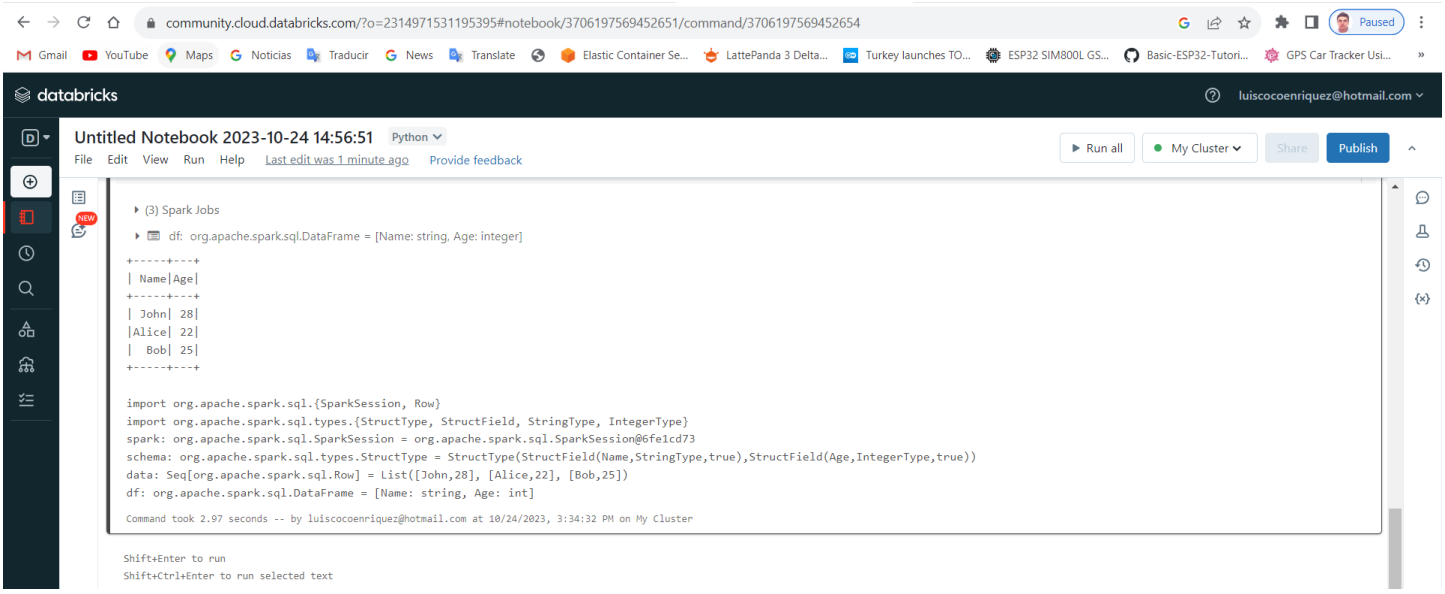
Here's a brief overview of how DataFrames work in Scala with Spark:

Creation:

You can create a DataFrame from various sources, such as Hive tables, external databases, existing RDDs (Resilient Distributed Datasets), or even from local collections.

```
// Import necessary libraries
import org.apache.spark.sql.{SparkSession, Row}
import org.apache.spark.sql.types.{StructType, StructField, StringType, IntegerType}

// Create a Spark session
val spark = SparkSession.builder.appName("example").getOrCreate()

// Define the schema
val schema = StructType(
  Array(
    StructField("Name", StringType, true),
    StructField("Age", IntegerType, true)
  )
)

// Provide sample data
val data = Seq(
  Row("John", 28),
  Row("Alice", 22),
  Row("Bob", 25)
)

// Create DataFrame
val df = spark.createDataFrame(spark.sparkContext.parallelize(data), schema)

// Show the DataFrame
df.show()
```





Transformation:

DataFrames support a wide range of operations, such as filtering, selecting, grouping, and aggregating data.

These operations are performed in a similar manner to SQL queries.

Here's an example that demonstrates filtering, selecting, grouping, and aggregating operations on the DataFrame created with the sample data:

```
// Import necessary libraries
import org.apache.spark.sql.functions._

// Filter the DataFrame to select people with age greater than 25
val filteredDF = df.filter("Age > 25")

// Select only the "Name" column
```

```
val selectedDF = df.select("Name")

// Group the DataFrame by age and count the occurrences
val groupedDF = df.groupBy("Age").count()

// Aggregate the DataFrame to find the average age
val avgAgeDF = df.agg(avg("Age"))

// Show the results
println("Filtered DataFrame:")
filteredDF.show()

println("Selected DataFrame:")
selectedDF.show()

println("Grouped DataFrame:")
groupedDF.show()

println("Average Age:")
avgAgeDF.show()
```

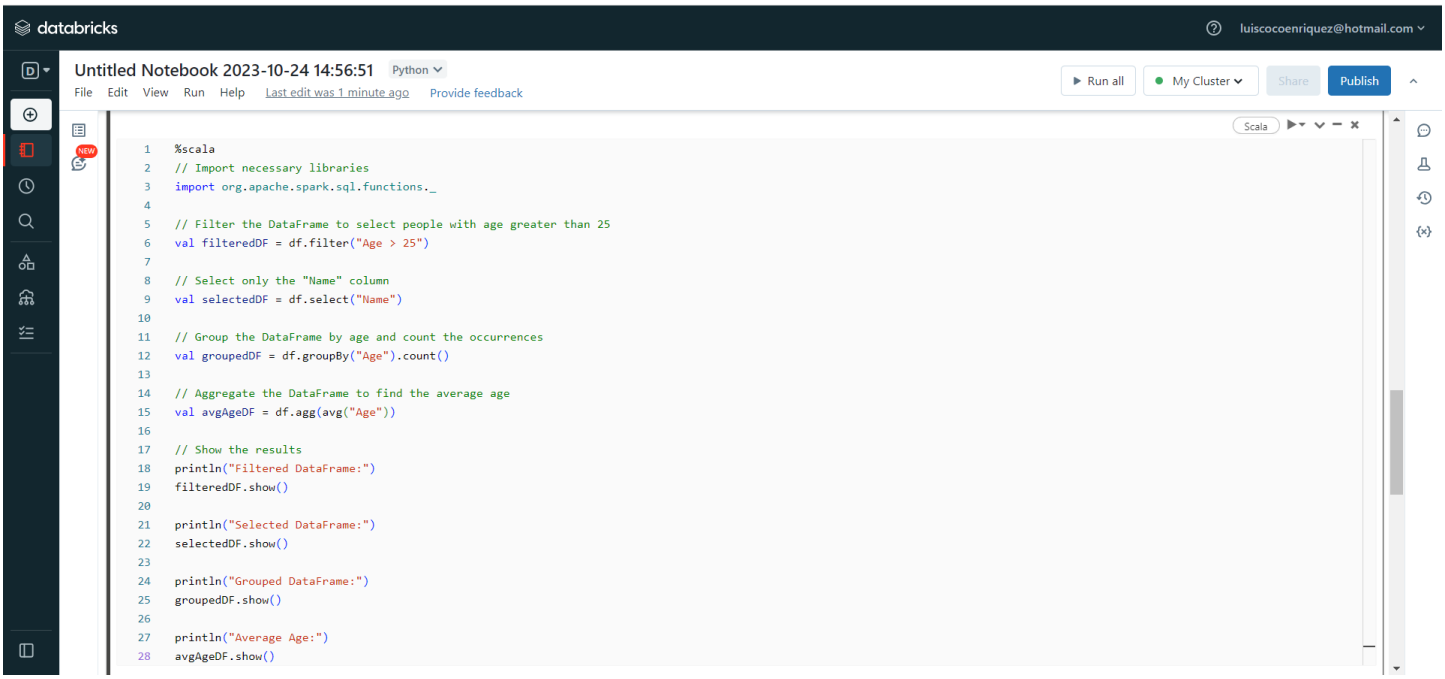
This code snippet demonstrates:

Filtering: Selecting individuals with an age greater than 25.

Selecting: Choosing only the "Name" column.

Grouping: Grouping the DataFrame by the "Age" column.

Aggregating: Calculating the average age of all individuals.



This is the output:

```
Filtered DataFrame:
+----+---+
|Name|Age|
+----+---+
|John| 28|
+----+---+

Selected DataFrame:
+-----+
| Name|
+-----+
| John|
|Alice|
|  Bob|
+-----+

Grouped DataFrame:
+---+-----+
|Age|count|
+---+-----+
| 28|    1|
| 22|    1|
| 25|    1|
+---+-----+

Average Age:
+--------+
|avg(Age)|
+--------+
|    25.0|
+--------+
```

Action:

Actions are operations that trigger the execution of the computation plan.

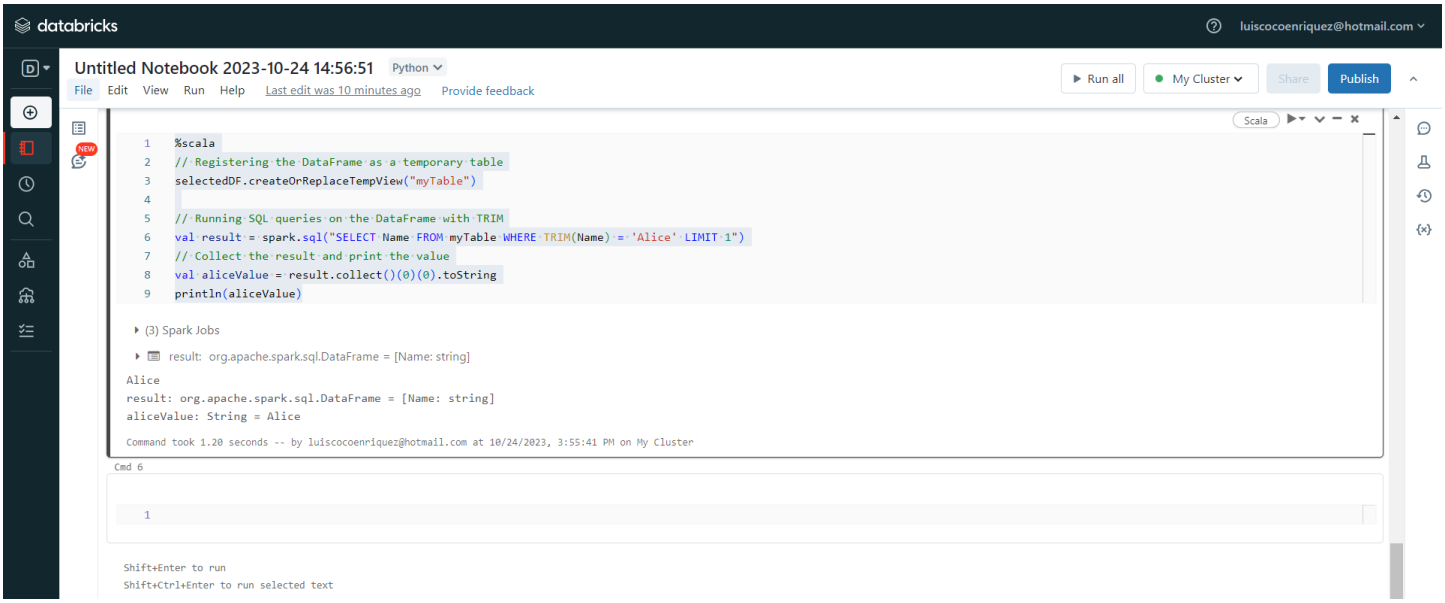
Examples include show() to display the data, count() to count the number of rows, and collect() to retrieve all data to the driver program.

```
// Display the first 10 rows
selectedDF.show(10)

// Count the number of rows
val rowCount = selectedDF.count()
```

Integration with Spark SQL:

DataFrames seamlessly integrate with Spark SQL, allowing you to run SQL queries on your DataFrames.

```
// Registering the DataFrame as a temporary table
selectedDF.createOrReplaceTempView("myTable")

// Running SQL queries on the DataFrame with TRIM
val result = spark.sql("SELECT Name FROM myTable WHERE TRIM(Name) = 'Alice' LIMIT 1")
// Collect the result and print the value
val aliceValue = result.collect()(0)(0).toString
println(aliceValue)
```



Optimizations:

Spark optimizes the execution plan of DataFrames using a query optimizer called Catalyst.

This helps in improving performance by optimizing the physical execution plan of the operations.

Here, spark is an instance of SparkSession, which is the entry point to programming Spark with the DataFrame and SQL API.

It provides a way to configure Spark application settings and access Spark functionality.

Remember that Spark is designed for distributed computing, and DataFrames are processed in parallel across a cluster of machines.

Spark provides various optimizations to improve the performance of DataFrame operations. Let's explore a few optimization techniques:

Predicate Pushdown:

Spark's Catalyst optimizer can push down filters closer to the data source.

This means that if you filter a DataFrame using a condition, Spark may optimize the physical execution plan by applying the filter as close to the data source as possible, reducing the amount of data that needs to be processed.

```
val result = spark.sql("SELECT * FROM myTable WHERE age > 21")
```

In this example, if the "age" column is part of the data source, Spark may push down the filter directly to the data source.

Projection Pushdown:

Similar to predicate pushdown, Spark can optimize queries by pushing down projections.

If your query only needs a subset of columns, Spark may optimize the execution plan by selecting only those columns at an earlier stage, reducing data movement.

```
val result = spark.sql("SELECT name, age FROM myTable")
```

In this case, if the data source supports projection pushdown, Spark may optimize the query by selecting only the "name" and "age" columns at the source.

Broadcast Joins:

Spark can optimize join operations by broadcasting smaller DataFrames to all nodes in the cluster, avoiding shuffling of large amounts of data.

```
val result = df1.join(broadcast(df2), "id")
```

The broadcast function hints Spark to use a broadcast join for the smaller DataFrame (df2 in this case).

Caching:

You can explicitly cache DataFrames or tables in memory to avoid recomputing them. This can be useful when you have iterative algorithms or when a DataFrame is reused multiple times.

```
myTable.cache()
```

This caches the DataFrame myTable in memory, and subsequent operations on it may benefit from the cached data.

Partitioning:

Ensuring that your DataFrames are properly partitioned can significantly improve performance, especially when performing operations that involve shuffling.

```
val partitionedDF = myTable.repartition(col("someColumn"))
```

This repartitions the DataFrame based on the values in "someColumn," which can optimize certain operations.

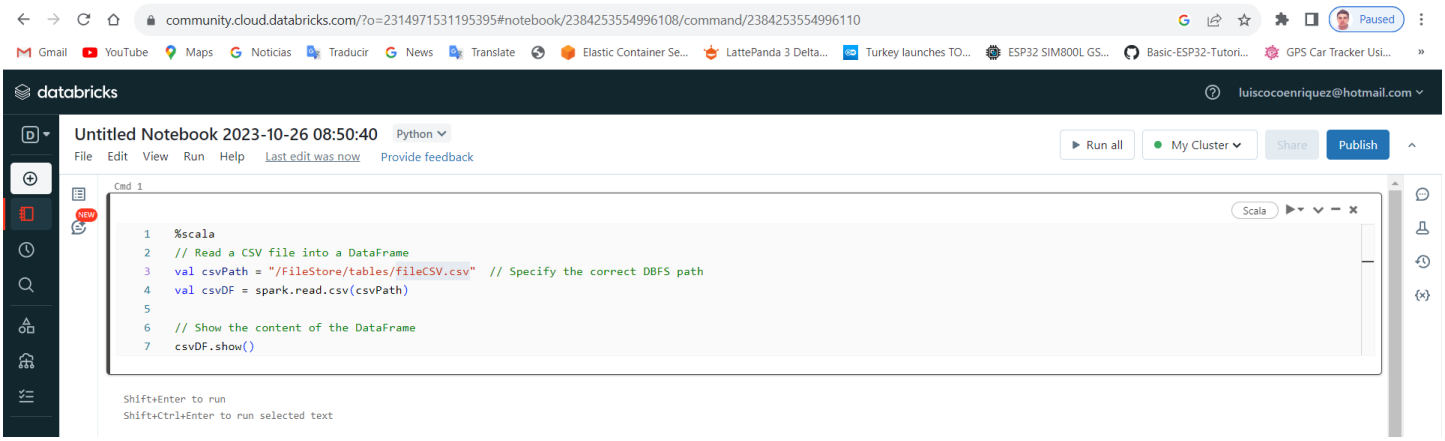
2.4. Data Sources

Sure, when it comes to working with Scala and Spark in Databricks, you might want to leverage various data sources.

CSV:

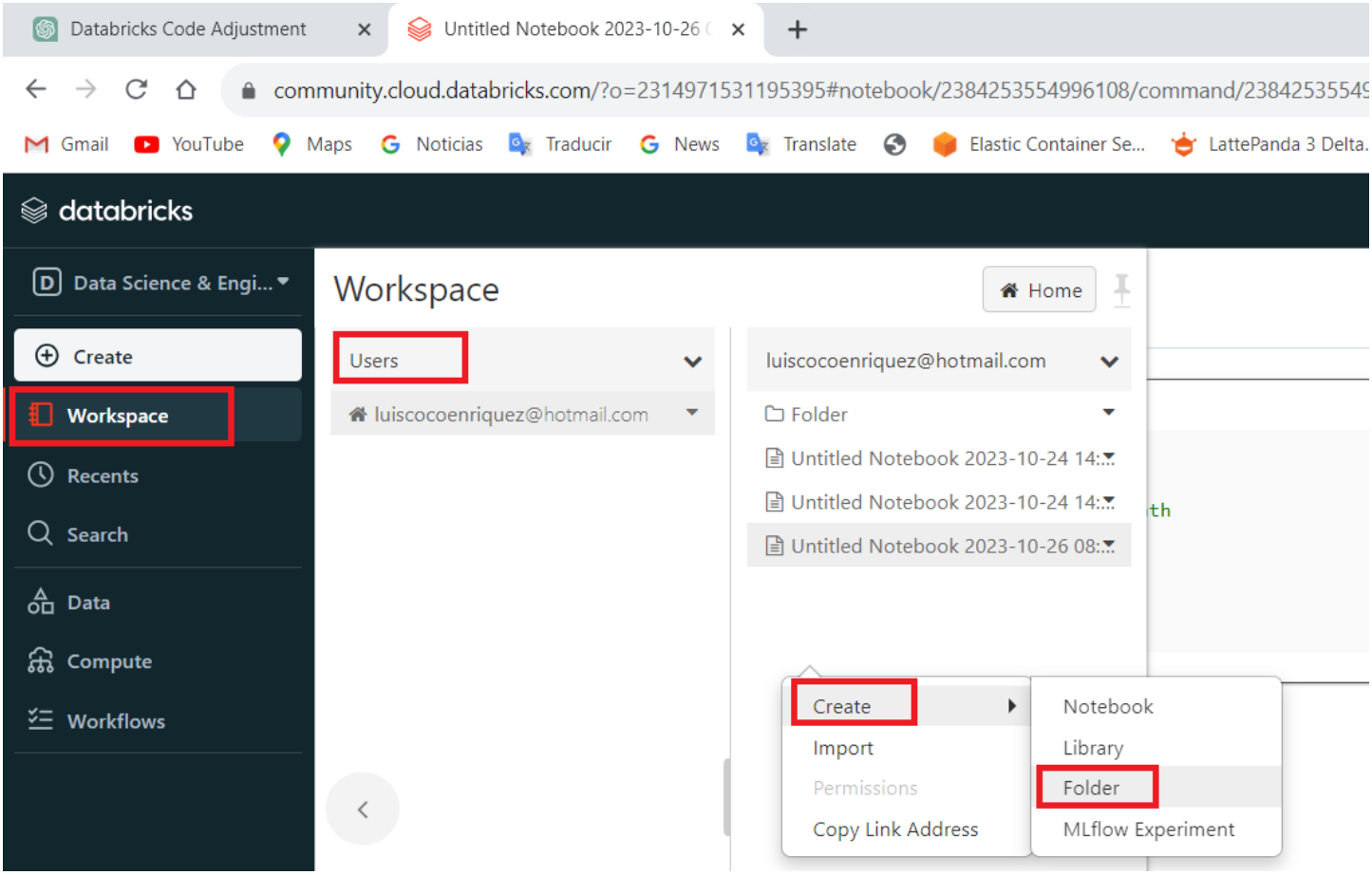
```
%scala
// Read a CSV file into a DataFrame
val csvPath = "/FileStore/tables/fileCSV.csv"  // Specify the correct DBFS path
val csvDF = spark.read.csv(csvPath)

// Show the content of the DataFrame
csvDF.show()
```

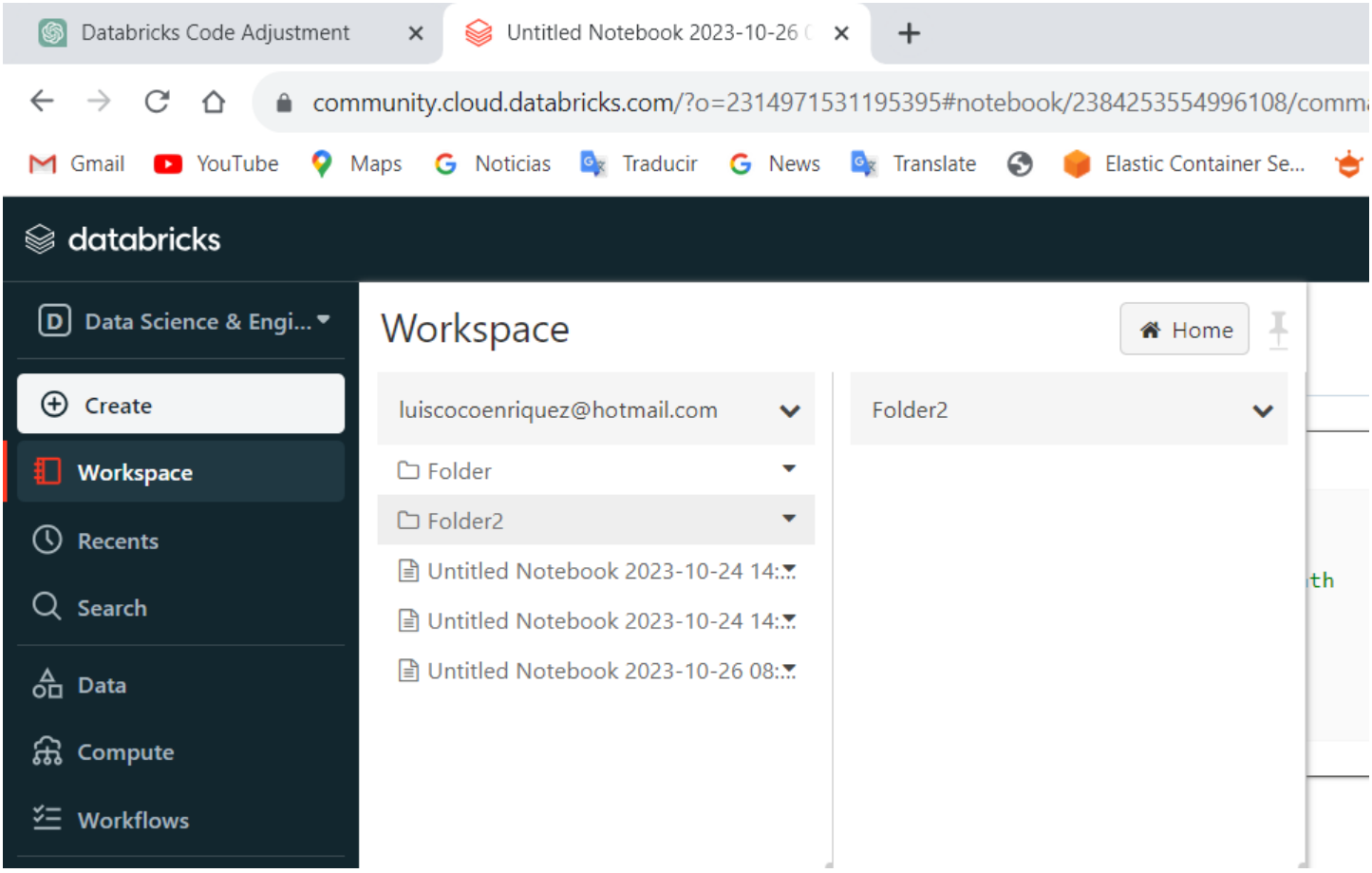
We first enter the code in a Notebook(in DataBricks)



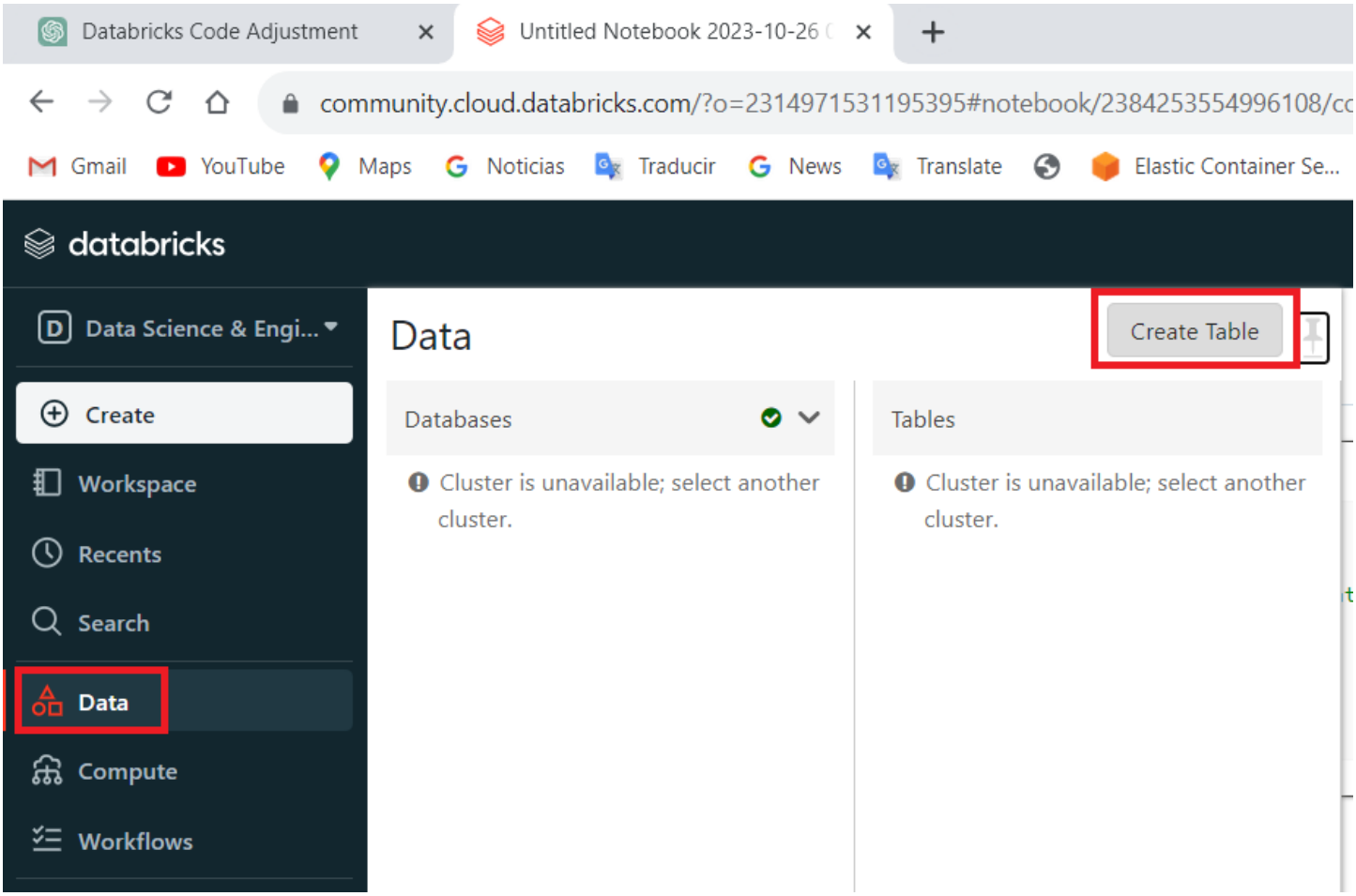
Then we upload the CSV file to DataBricks.

We select the menu option "Workspace" then we right click to create a new Folder. We set the folder name.

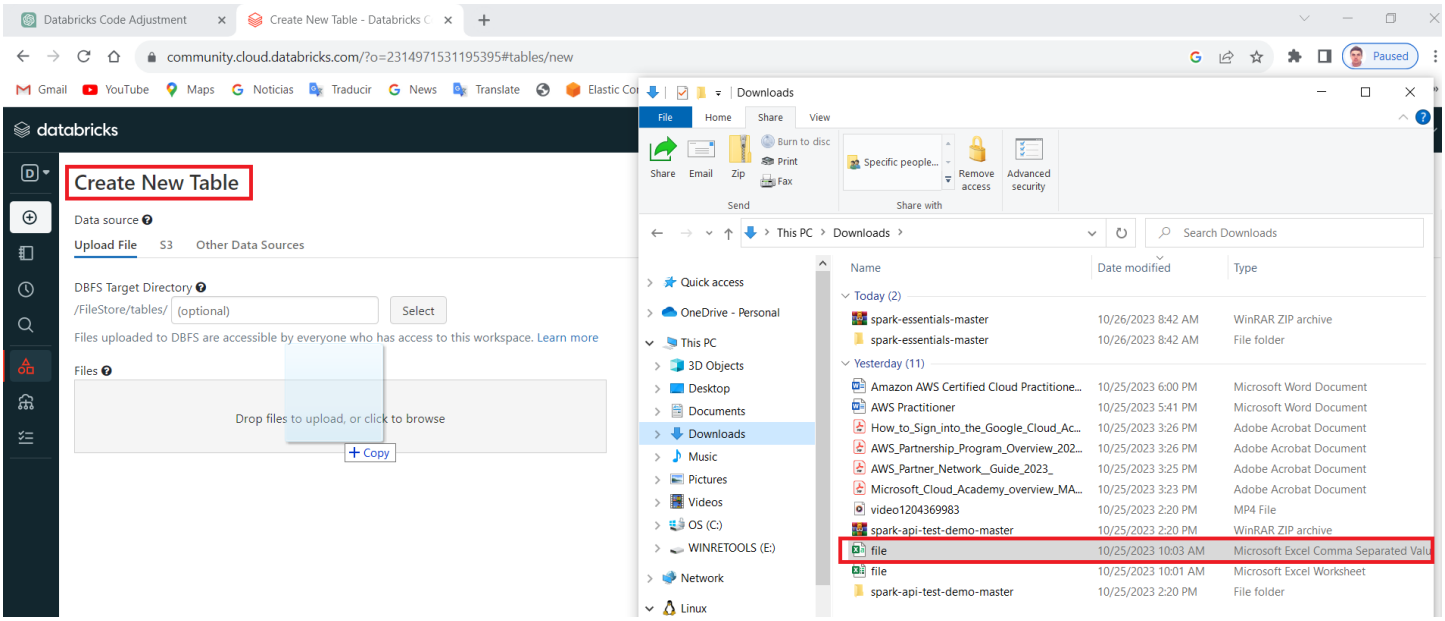


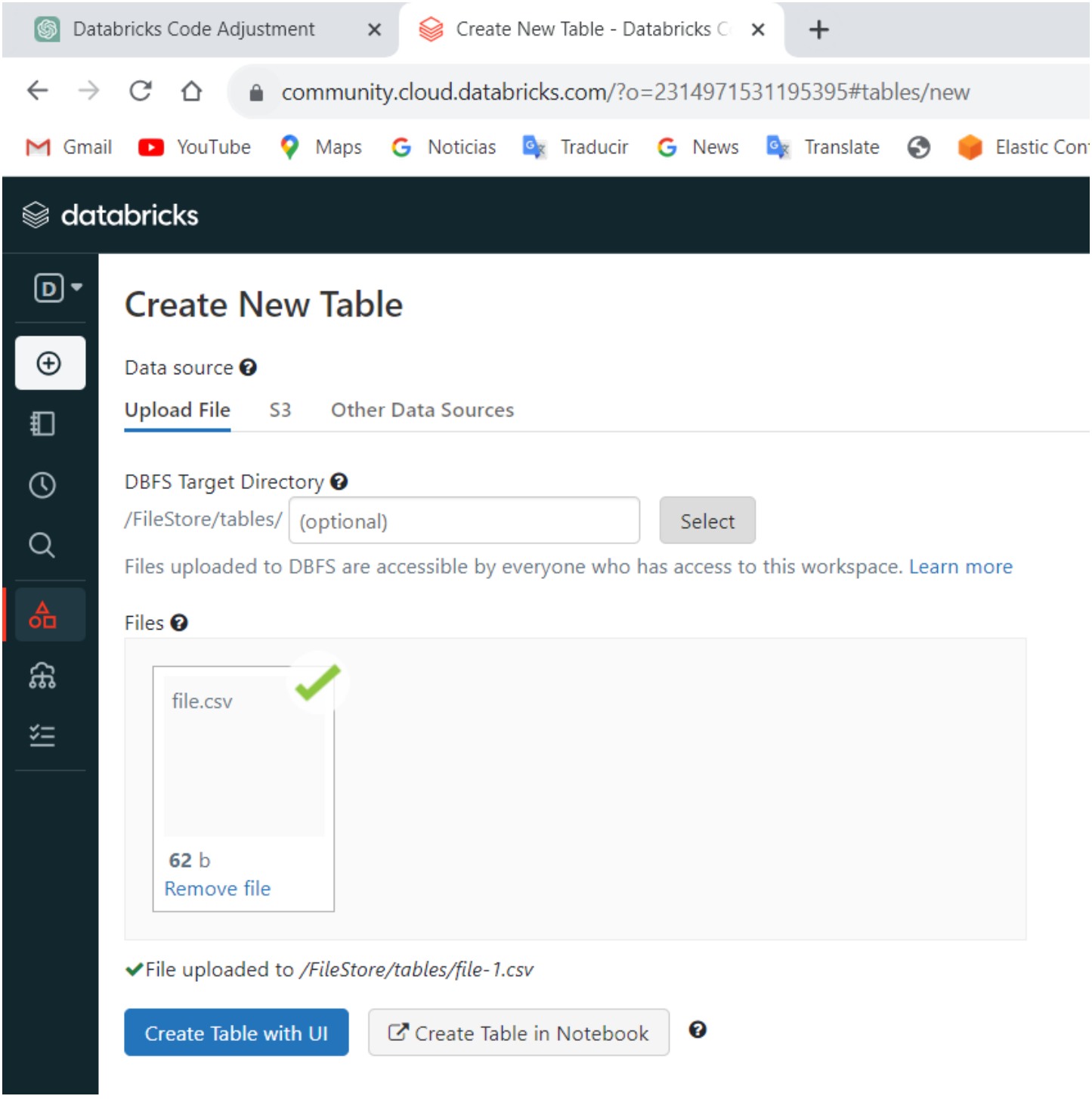


After creating the Folder we select the folder and we go to the menu option "Data" and we press the "Create Table" button

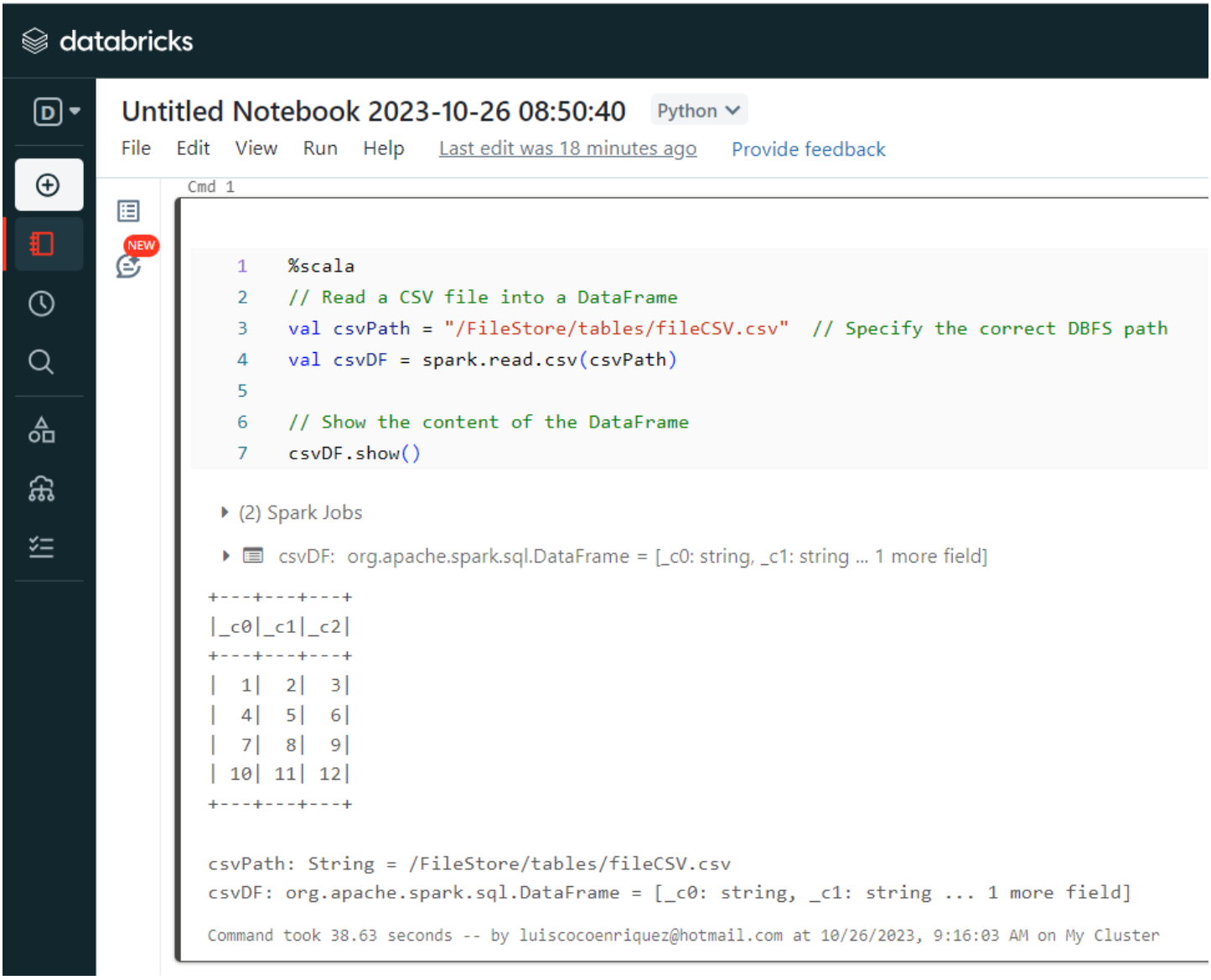


We drag and drop the CSV file to be uploaded to DataBricks



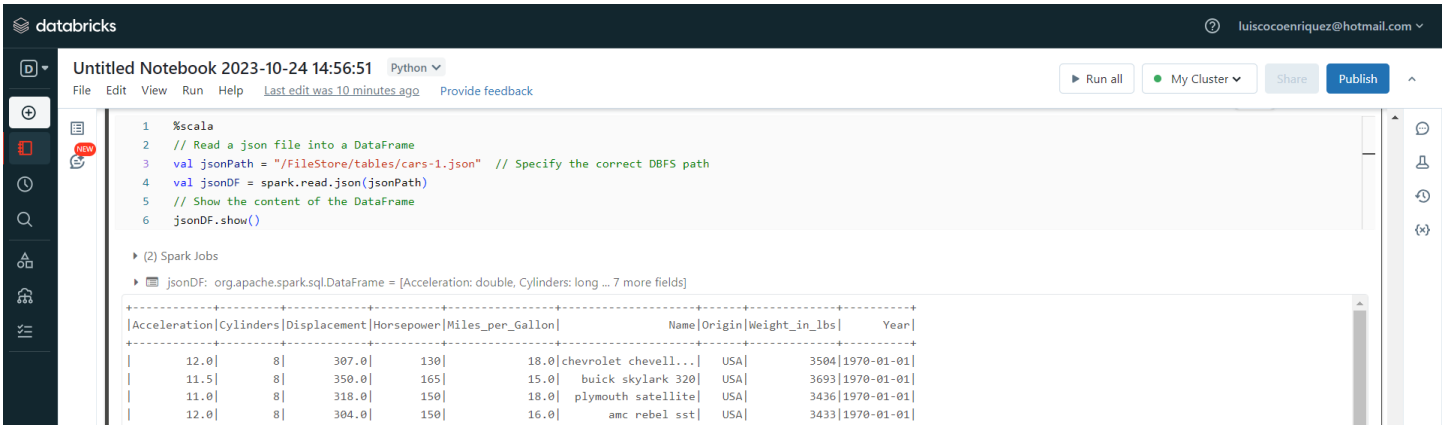


Finally, we execute the scala code in the DataBricks Notebook



JSON:

```
%scala
// Read a json file into a DataFrame
val jsonPath = "/FileStore/tables/cars-1.json"  // Specify the correct DBFS path
val jsonDF = spark.read.json(jsonPath)
// Show the content of the DataFrame
jsonDF.show()
```



Parquet:

```
val df = spark.read.parquet("/path/to/parquet/file")
```

Avro:

```
val df = spark.read.format("avro").load("/path/to/avro/file")
```

Delta Lake:

```
Copy code
val df = spark.read.format("delta").load("/path/to/delta/lake")
```

ORC:

```
val df = spark.read.format("orc").load("/path/to/orc/file")
```

JDBC:

```
val jdbcUrl = "jdbc:mysql://hostname:port/dbname"
val df = spark.read.format("jdbc").option("url", jdbcUrl).option("dbtable", "tablename").load(
```

Cassandra:

```
val df = spark.read.format("org.apache.spark.sql.cassandra").option("keyspace", "keyspace").op
```

Remember to replace the file paths and connection details with your specific information.

These examples assume that you have a SparkSession named spark already created.

2.5. Data Sources. Exercises

```
import org.apache.spark.sql.{SaveMode, SparkSession}
import org.apache.spark.sql.types._

object DataSources extends App {
```

```
  val spark = SparkSession.builder()
    .appName("Data Sources and Formats")
    .getOrCreate()

  // Existing code for schema definitions and reading cars data

  val carsDF = spark.read
    .format("json")
    .schema(carsSchema)
    .option("mode", "failFast")
    .option("path", "dbfs:/FileStore/tables/cars.json") // Use dbfs:/FileStore/tables/ for Dat
    .load()

  // Writing DFs
  carsDF.write
    .format("json")
    .mode(SaveMode.Overwrite)
    .save("dbfs:/FileStore/tables/cars_dupe.json")

  // Existing code for reading/writing JSON, CSV, Parquet, Text files, and reading from a remo

  /**
    * Exercise: read the movies DF, then write it as
    * - tab-separated values file
    * - snappy Parquet
    * - table "public.movies" in the Postgres DB
    */

  val moviesDF = spark.read.json("dbfs:/FileStore/tables/movies.json") // Update path for Data

  // TSV
  moviesDF.write
    .format("csv")
    .option("header", "true")
    .option("sep", "\t")
    .save("dbfs:/FileStore/tables/movies.csv")

  // Parquet
  moviesDF.write.save("dbfs:/FileStore/tables/movies.parquet")

  // Save to DF
  moviesDF.write
    .format("jdbc")
    .option("driver", driver)
    .option("url", url)
    .option("user", user)
    .option("password", password)
    .option("dbtable", "public.movies")
    .save()
}
```

2.6. DataFrames Columns and Expressions

In Apache Spark with Scala, DataFrames are a fundamental abstraction representing a distributed collection of data organized into named columns.

Spark DataFrames can be created from various data sources, and once created, you can perform operations on them using transformations and actions.

Creating a DataFrame:

```
// Import necessary libraries
import org.apache.spark.sql.{SparkSession, DataFrame}

// Create a Spark session
val spark = SparkSession.builder.appName("example").getOrCreate()

// Sample data
val data = Seq(("Alice", 25), ("Bob", 30), ("Charlie", 22))

// Define the schema for the DataFrame
val schema = List("Name", "Age")

// Create a DataFrame
val df = spark.createDataFrame(data).toDF(schema: _*)
```

Showing the DataFrame:

```
df.show()
```

Selecting Columns:

```
// Selecting a single column
val nameColumn = df("Name")

// Selecting multiple columns
val selectedColumns = df.select("Name", "Age")
```

Adding a New Column:

```
Copy code
// Adding a new column
val updatedDF = df.withColumn("AgeAfter5Years", df("Age") + 5)
```

Filtering Rows:

```
// Filtering based on a condition
val filteredDF = df.filter(df("Age") > 25)
```

Grouping and Aggregating:

```
// Grouping by a column and calculating the average
val groupedDF = df.groupBy("Name").agg(avg("Age"))
```

Joining DataFrames:

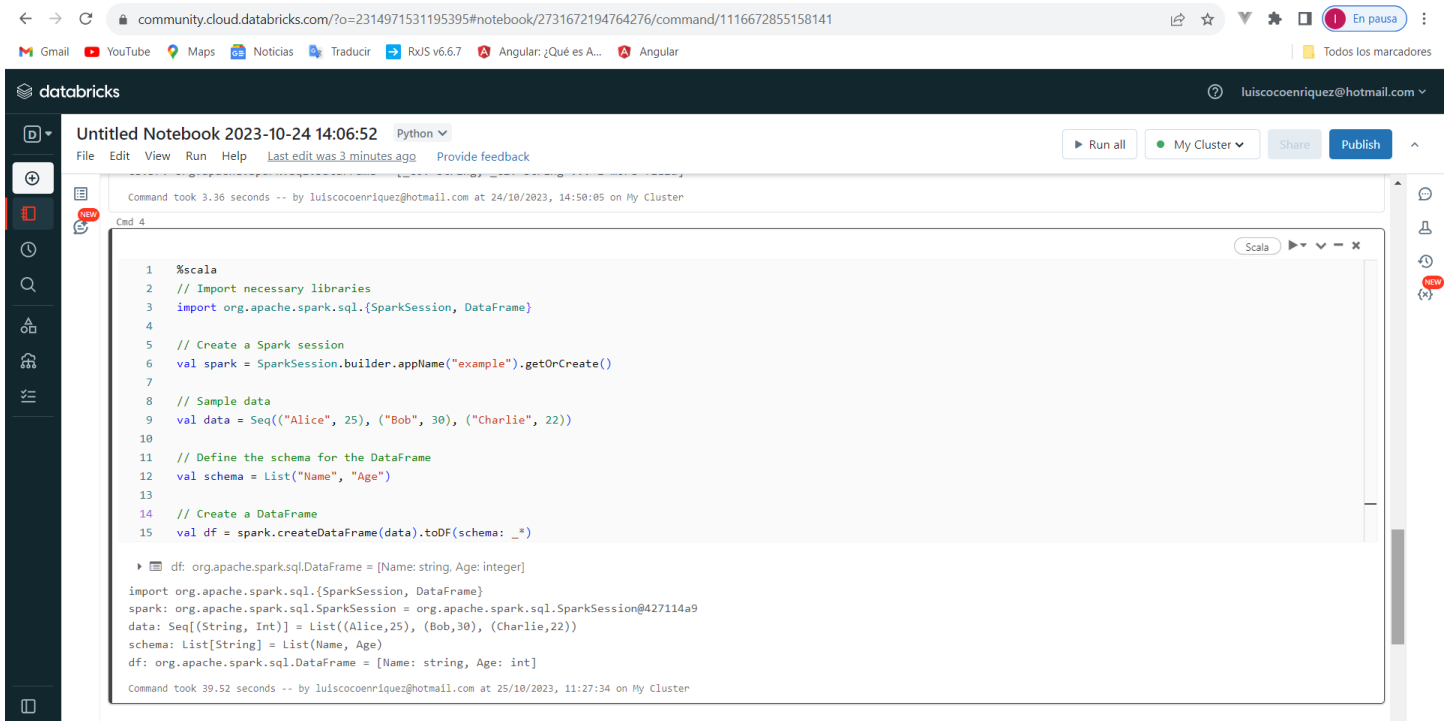
```
// Create another DataFrame
val otherData = Seq(("Alice", "Engineer"), ("Bob", "Doctor"))
val otherDF = spark.createDataFrame(otherData).toDF("Name", "Occupation")

// Joining DataFrames
val joinedDF = df.join(otherDF, "Name")
```

These are just some basic operations.

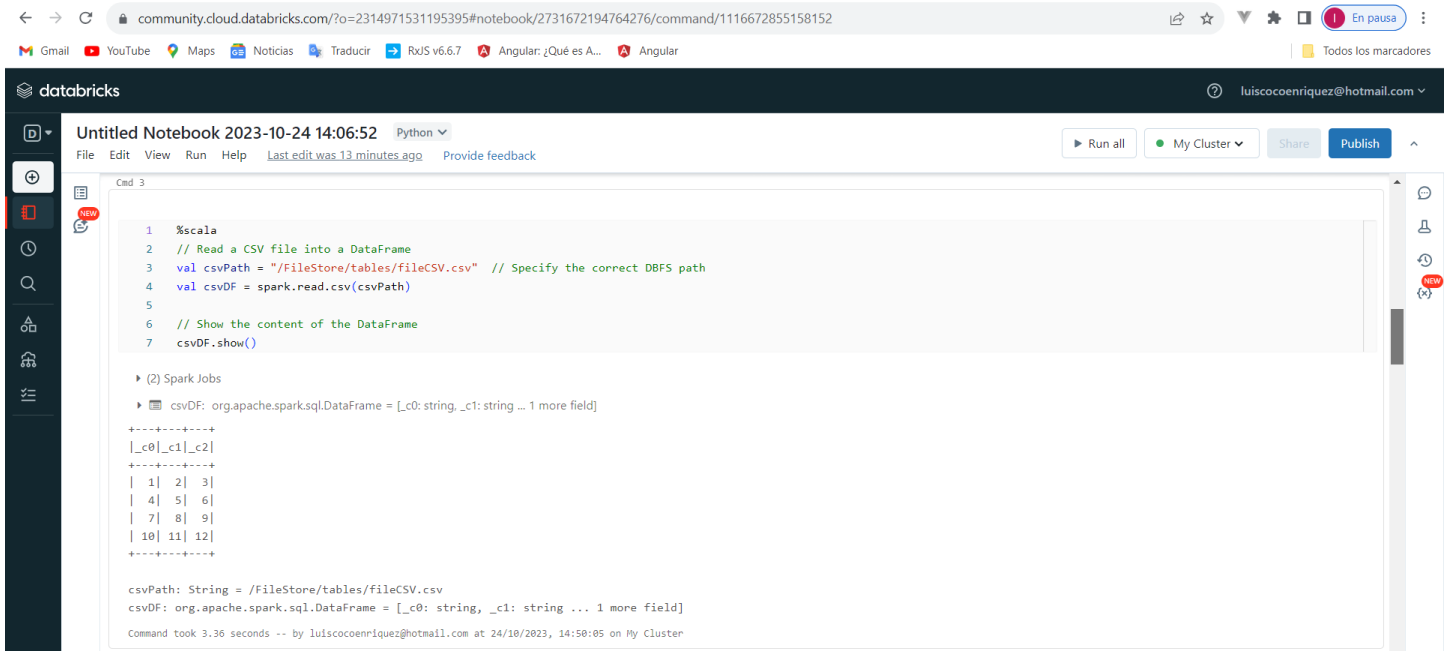
Spark supports a rich set of transformations and actions that you can perform on DataFrames.

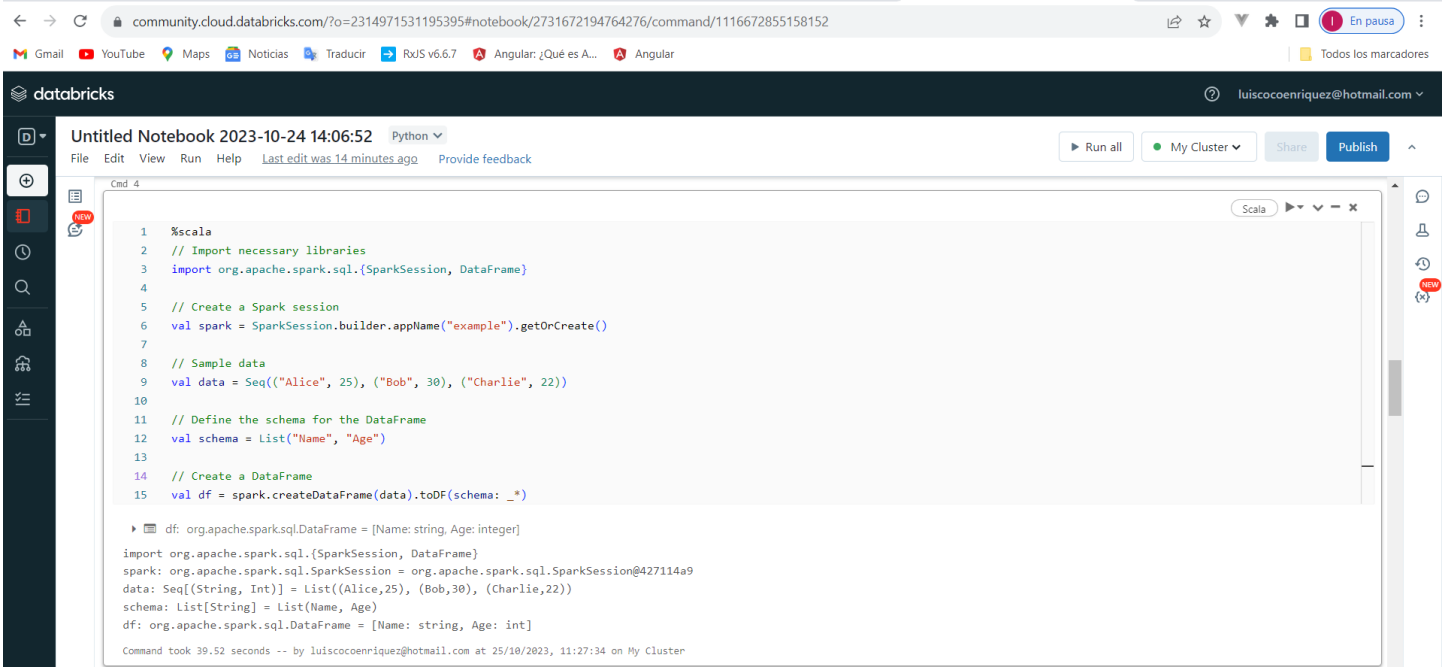
DataFrames provide a high-level API that abstracts away many of the complexities of distributed processing.

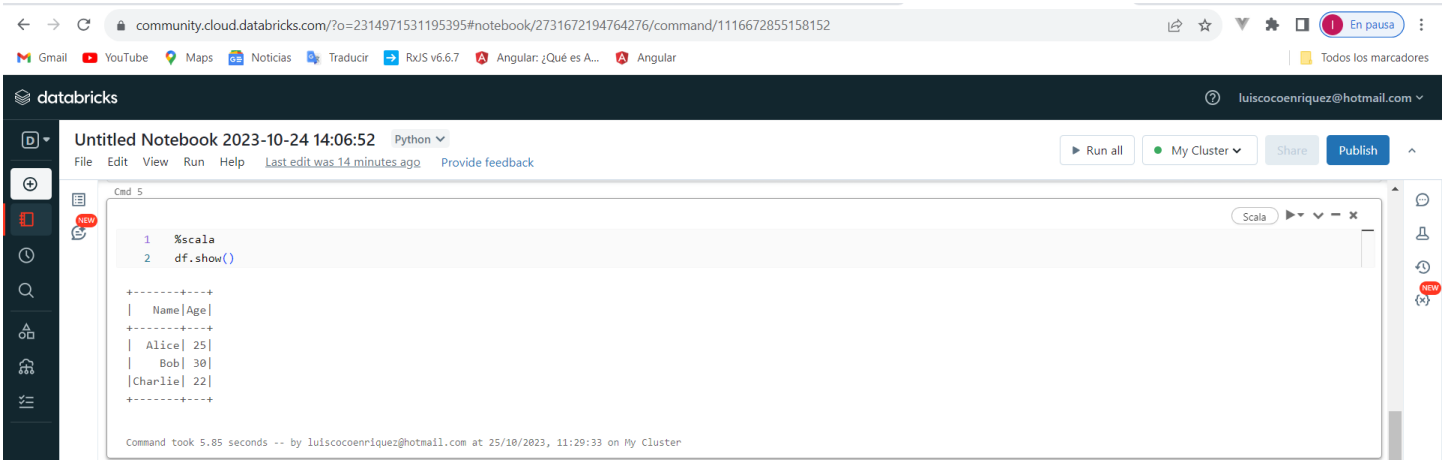
2.7. DataFrames Columns and Expressions. Exercises

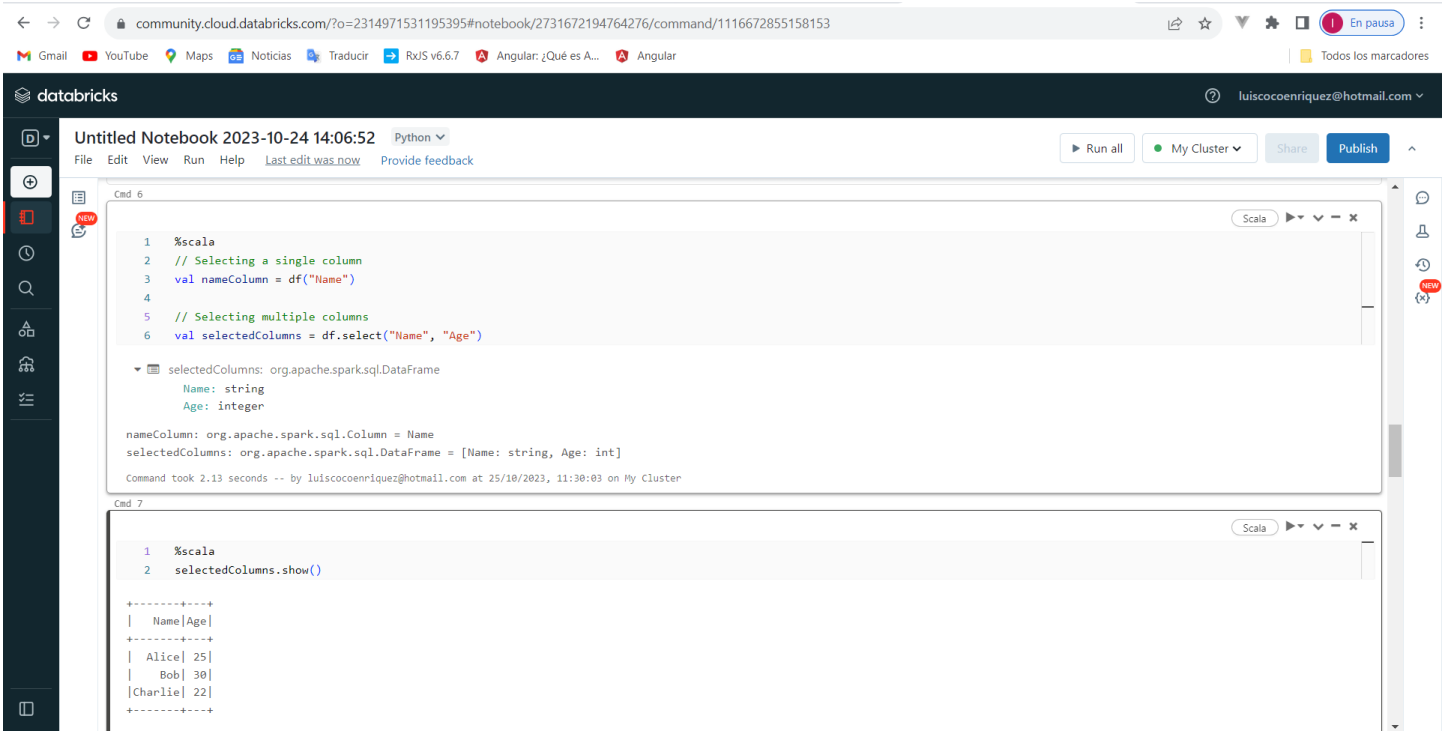

community.cloud.databricks.com/?o=2314971531195395#notebook/2731672194764276/command/1116672855158141
databricks
luiscocoenriquez@hotmail.com
Untitled Notebook 2023-10-24 14:06:52
Python
File Edit View Run Help Last edit was 3 minutes ago Provide feedback
Run all
My Cluster
Share
Publish
Command took 3.36 seconds -- by luiscocoenriquez@hotmail.com at 24/10/2023, 14:50:05 on My Cluster
Cmd 4
Scala
%scala
// Import necessary libraries
import org.apache.spark.sql.{SparkSession, DataFrame}
// Create a Spark session
val spark = SparkSession.builder.appName("example").getOrCreate()
// Sample data
val data = Seq(("Alice", 25), ("Bob", 30), ("Charlie", 22))
// Define the schema for the DataFrame
val schema = List("Name", "Age")
// Create a DataFrame
val df = spark.createDataFrame(data).toDF(schema: _*)
df: org.apache.spark.sql.DataFrame = [Name: string, Age: integer]
import org.apache.spark.sql.{SparkSession, DataFrame}
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@427114a9
data: Seq[(String, Int)] = List((Alice,25), (Bob,30), (Charlie,22))
schema: List[String] = List(Name, Age)
df: org.apache.spark.sql.DataFrame = [Name: string, Age: int]
Command took 39.52 seconds -- by luiscocoenriquez@hotmail.com at 25/10/2023, 11:27:34 on My Cluster

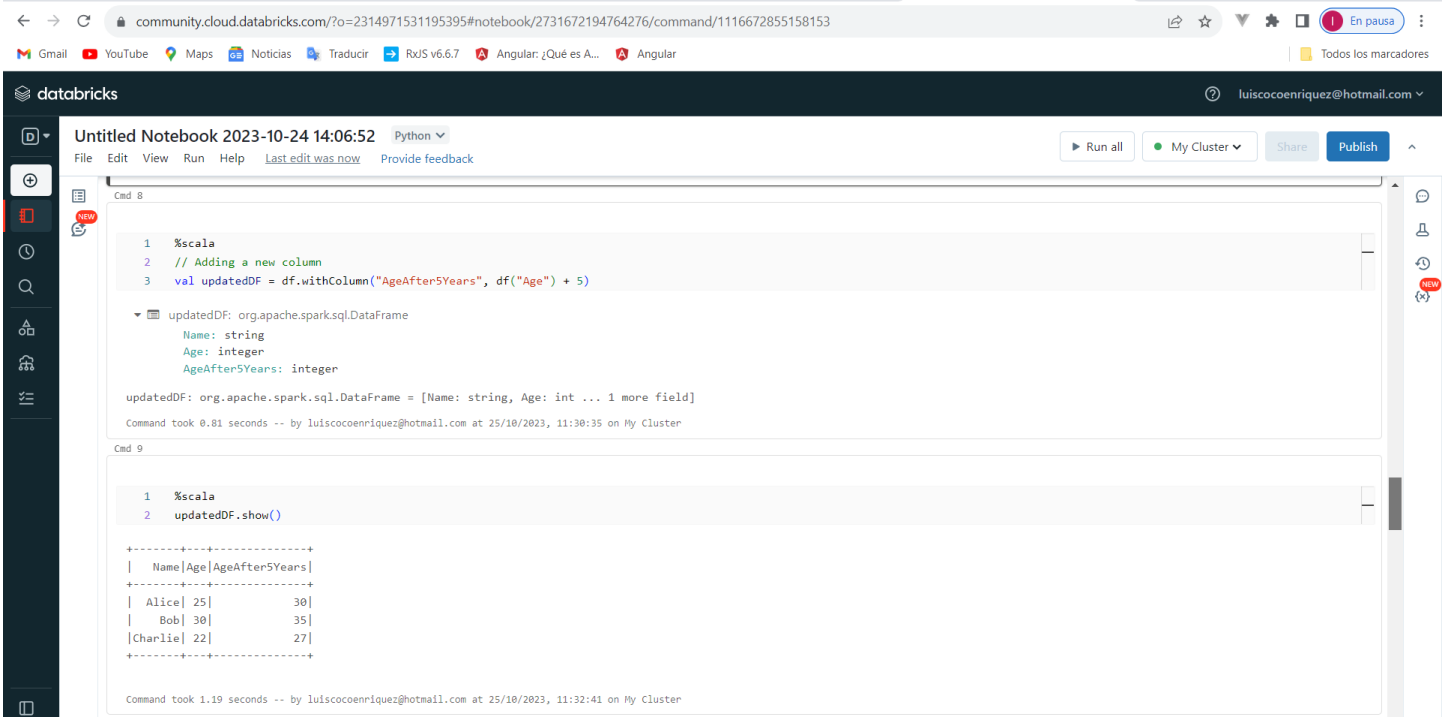

community.cloud.databricks.com/?o=2314971531195395#notebook/2731672194764276/command/1116672855158152
databricks
luiscocoenriquez@hotmail.com
Untitled Notebook 2023-10-24 14:06:52
Python
File Edit View Run Help Last edit was 13 minutes ago Provide feedback
Run all
My Cluster
Share
Publish
Cmd 2
%scala
// Read a CSV file into a DataFrame
val csvPath = "/FileStore/tables/fileCSV.csv" // Specify the correct DBFS path
val csvDF = spark.read.csv(csvPath)
(1) Spark Jobs
csvDF: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string ... 1 more field]
csvPath: String = /FileStore/tables/fileCSV.csv
csvDF: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string ... 1 more field]
Command took 2.98 seconds -- by luiscocoenriquez@hotmail.com at 24/10/2023, 14:49:48 on My Cluster

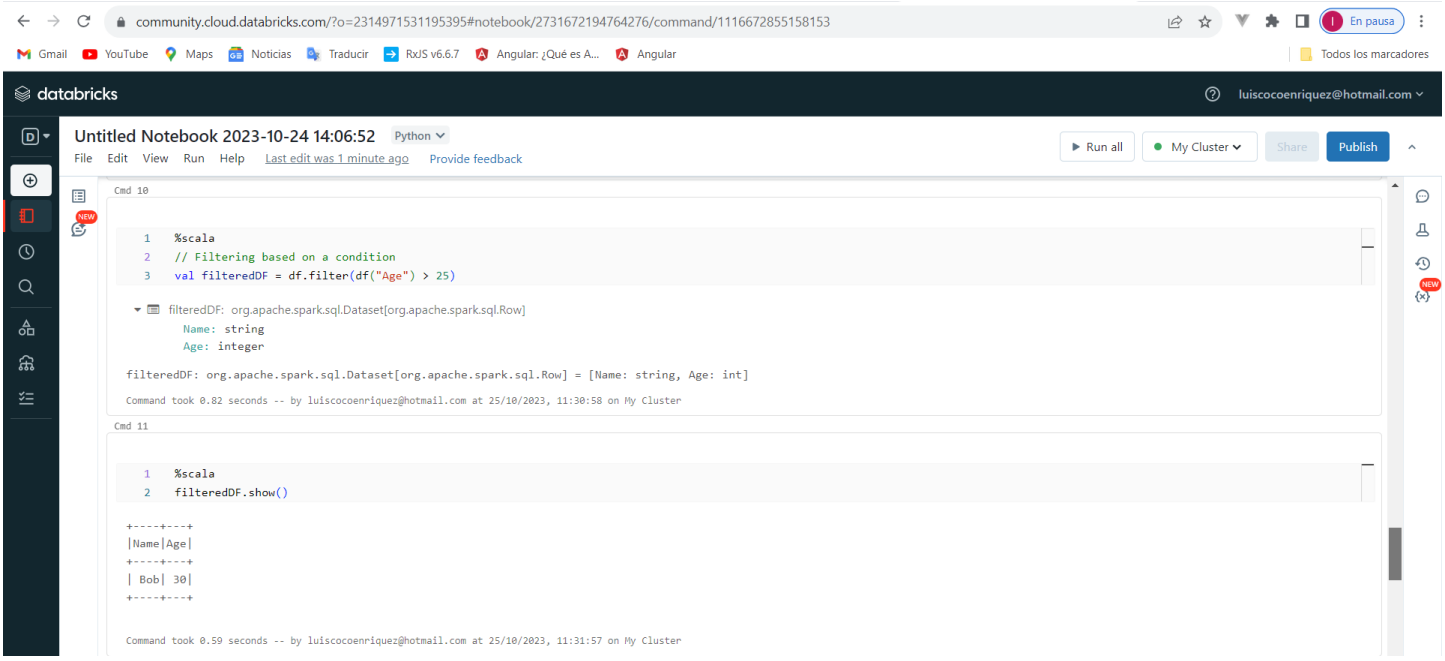

community.cloud.databricks.com/?o=2314971531195395#notebook/2731672194764276/command/1116672855158152
databricks
luiscocoenriquez@hotmail.com
Untitled Notebook 2023-10-24 14:06:52
Python
File Edit View Run Help Last edit was 13 minutes ago Provide feedback
Run all
My Cluster
Share
Publish
Cmd 3
%scala
// Read a CSV file into a DataFrame
val csvPath = "/FileStore/tables/fileCSV.csv" // Specify the correct DBFS path
val csvDF = spark.read.csv(csvPath)
// Show the content of the DataFrame
csvDF.show()
(2) Spark Jobs
csvDF: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string ... 1 more field]
+---+---+---+
|_c0|_c1|_c2|
+---+---+---+
1	2	3
4	5	6
7	8	9
10	11	12
+---+---+---+
csvPath: String = /FileStore/tables/fileCSV.csv
csvDF: org.apache.spark.sql.DataFrame = [_c0: string, _c1: string ... 1 more field]
Command took 3.36 seconds -- by luiscocoenriquez@hotmail.com at 24/10/2023, 14:50:05 on My Cluster

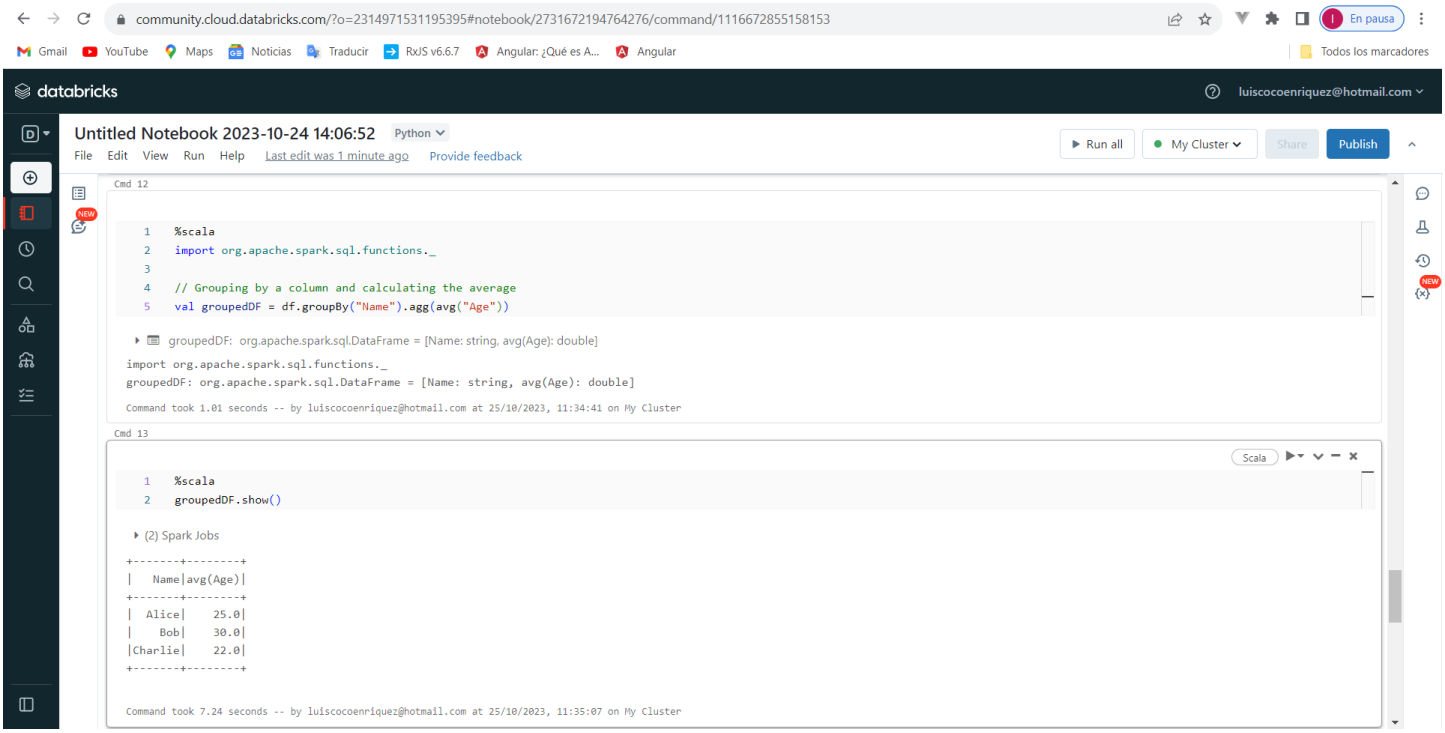

community.cloud.databricks.com/?o=2314971531195395#notebook/2731672194764276/command/1116672855158152
databricks
luiscocoenriquez@hotmail.com
Untitled Notebook 2023-10-24 14:06:52
Python
File Edit View Run Help Last edit was 14 minutes ago Provide feedback
Run all
My Cluster
Share
Publish
Cmd 4
Scala
%scala
// Import necessary libraries
import org.apache.spark.sql.{SparkSession, DataFrame}
// Create a Spark session
val spark = SparkSession.builder.appName("example").getOrCreate()
// Sample data
val data = Seq(("Alice", 25), ("Bob", 30), ("Charlie", 22))
// Define the schema for the DataFrame
val schema = List("Name", "Age")
// Create a DataFrame
val df = spark.createDataFrame(data).toDF(schema: _*)
df: org.apache.spark.sql.DataFrame = [Name: string, Age: integer]
import org.apache.spark.sql.{SparkSession, DataFrame}
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@427114a9
data: Seq[(String, Int)] = List((Alice,25), (Bob,30), (Charlie,22))
schema: List[String] = List(Name, Age)
df: org.apache.spark.sql.DataFrame = [Name: string, Age: int]
Command took 39.52 seconds -- by luiscocoenriquez@hotmail.com at 25/10/2023, 11:27:34 on My Cluster

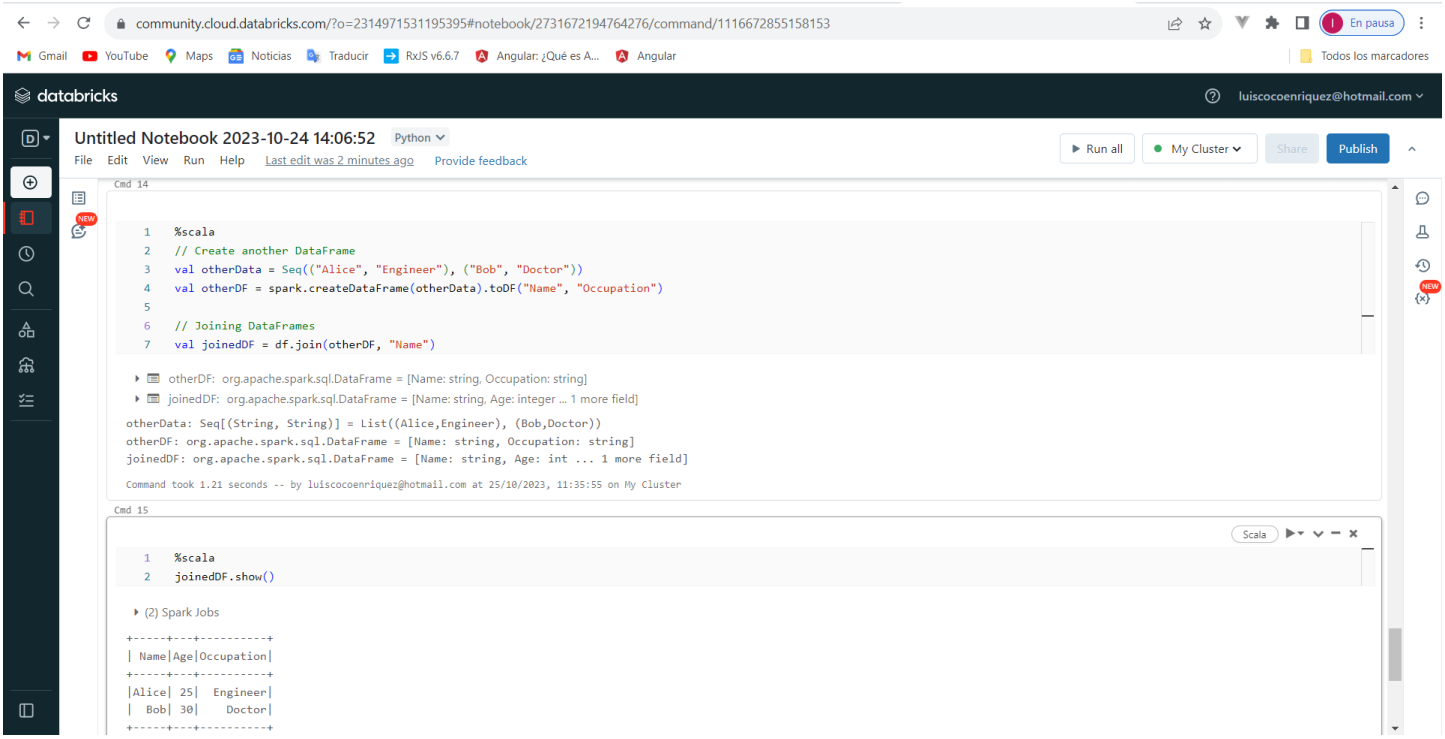

community.cloud.databricks.com/?o=2314971531195395#notebook/2731672194764276/command/1116672855158152
databricks
luiscocoenriquez@hotmail.com
Untitled Notebook 2023-10-24 14:06:52
Python
File Edit View Run Help Last edit was 14 minutes ago Provide feedback
Run all
My Cluster
Share
Publish
Cmd 5
Scala
%scala
df.show()
+-------+---+
| Name|Age|
+-------+---+
| Alice| 25|
| Bob| 30|
|Charlie| 22|
+-------+---+
Command took 5.85 seconds -- by luiscocoenriquez@hotmail.com at 25/10/2023, 11:29:33 on My Cluster


community.cloud.databricks.com/?o=2314971531195395#notebook/2731672194764276/command/1116672855158153
databricks
luiscocoenriquez@hotmail.com
Untitled Notebook 2023-10-24 14:06:52
Python
File Edit View Run Help Last edit was now Provide feedback
Run all
My Cluster
Share
Publish
Cmd 6
Scala
%scala
// Selecting a single column
val nameColumn = df("Name")
// Selecting multiple columns
val selectedColumns = df.select("Name", "Age")
selectedColumns: org.apache.spark.sql.DataFrame
Name: string
Age: integer
nameColumn: org.apache.spark.sql.Column = Name
selectedColumns: org.apache.spark.sql.DataFrame = [Name: string, Age: int]
Command took 2.13 seconds -- by luiscocoenriquez@hotmail.com at 25/10/2023, 11:30:03 on My Cluster
Cmd 7
Scala
%scala
selectedColumns.show()
+-------+---+
| Name|Age|
+-------+---+
| Alice| 25|
| Bob| 30|
|Charlie| 22|
+-------+---+


community.cloud.databricks.com/?o=2314971531195395#notebook/2731672194764276/command/1116672855158153
databricks
luiscocoenriquez@hotmail.com
Untitled Notebook 2023-10-24 14:06:52
Python
File Edit View Run Help Last edit was now Provide feedback
Run all
My Cluster
Share
Publish
Cmd 8
%scala
// Adding a new column
val updatedDF = df.withColumn("AgeAfter5Years", df("Age") + 5)
updatedDF: org.apache.spark.sql.DataFrame
Name: string
Age: integer
AgeAfter5Years: integer
updatedDF: org.apache.spark.sql.DataFrame = [Name: string, Age: int ... 1 more field]
Command took 0.81 seconds -- by luiscocoenriquez@hotmail.com at 25/10/2023, 11:30:35 on My Cluster
Cmd 9
%scala
updatedDF.show()
+-------+---+--------------+
| Name|Age|AgeAfter5Years|
+-------+---+--------------+
| Alice| 25| 30|
| Bob| 30| 35|
|Charlie| 22| 27|
+-------+---+--------------+
Command took 1.19 seconds -- by luiscocoenriquez@hotmail.com at 25/10/2023, 11:32:41 on My Cluster


community.cloud.databricks.com/?o=2314971531195395#notebook/2731672194764276/command/1116672855158153
databricks
luiscocoenriquez@hotmail.com
Untitled Notebook 2023-10-24 14:06:52
Python
File Edit View Run Help Last edit was 1 minute ago Provide feedback
Run all
My Cluster
Share
Publish
Cmd 10
%scala
// Filtering based on a condition
val filteredDF = df.filter(df("Age") > 25)
filteredDF: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]
Name: string
Age: integer
filteredDF: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [Name: string, Age: int]
Command took 0.82 seconds -- by luiscocoenriquez@hotmail.com at 25/10/2023, 11:30:58 on My Cluster
Cmd 11
%scala
filteredDF.show()
+----+---+
|Name|Age|
+----+---+
| Bob| 30|
+----+---+
Command took 0.59 seconds -- by luiscocoenriquez@hotmail.com at 25/10/2023, 11:31:57 on My Cluster





2.8. DataFrame Aggregations

I'd be happy to help you with DataFrame aggregations in Scala using Apache Spark in Databricks!

DataFrame aggregations involve grouping data based on one or more columns and then performing some aggregate functions on the grouped data.

Let's say you have a DataFrame called df with columns name, age, and salary.

```
// Import necessary Spark libraries
import org.apache.spark.sql.functions._
```

```
import org.apache.spark.sql.expressions.Window

// Assuming you already have a DataFrame called df
// If not, you can read data into a DataFrame using spark.read

// Adding a 'Salary' column with sample values
val dfWithSalary = df.withColumn("Salary", lit(50000)) // You can replace 50000 with your desi

// Display the DataFrame with the added 'Salary' column
dfWithSalary.show()

// 1. Group by 'Name' and calculate the average salary
val avgSalaryDF = dfWithSalary.groupBy("Name").agg(avg("Salary").as("average_salary"))

// Display the result
avgSalaryDF.show()

// 2. Group by 'Age' and get the maximum salary
val maxSalaryDF = dfWithSalary.groupBy("Age").agg(max("Salary").as("max_salary"))

// Display the result
maxSalaryDF.show()

// 3. Use Window functions to calculate the rank based on salary
val windowSpec = Window.orderBy(desc("Salary"))
val rankDF = dfWithSalary.withColumn("rank", rank().over(windowSpec))

// Display the result
rankDF.show()

// 4. Group by 'Name' and get the count of records for each name
val countDF = dfWithSalary.groupBy("Name").agg(count("*").as("record_count"))

// Display the result
countDF.show()
```

In this example:

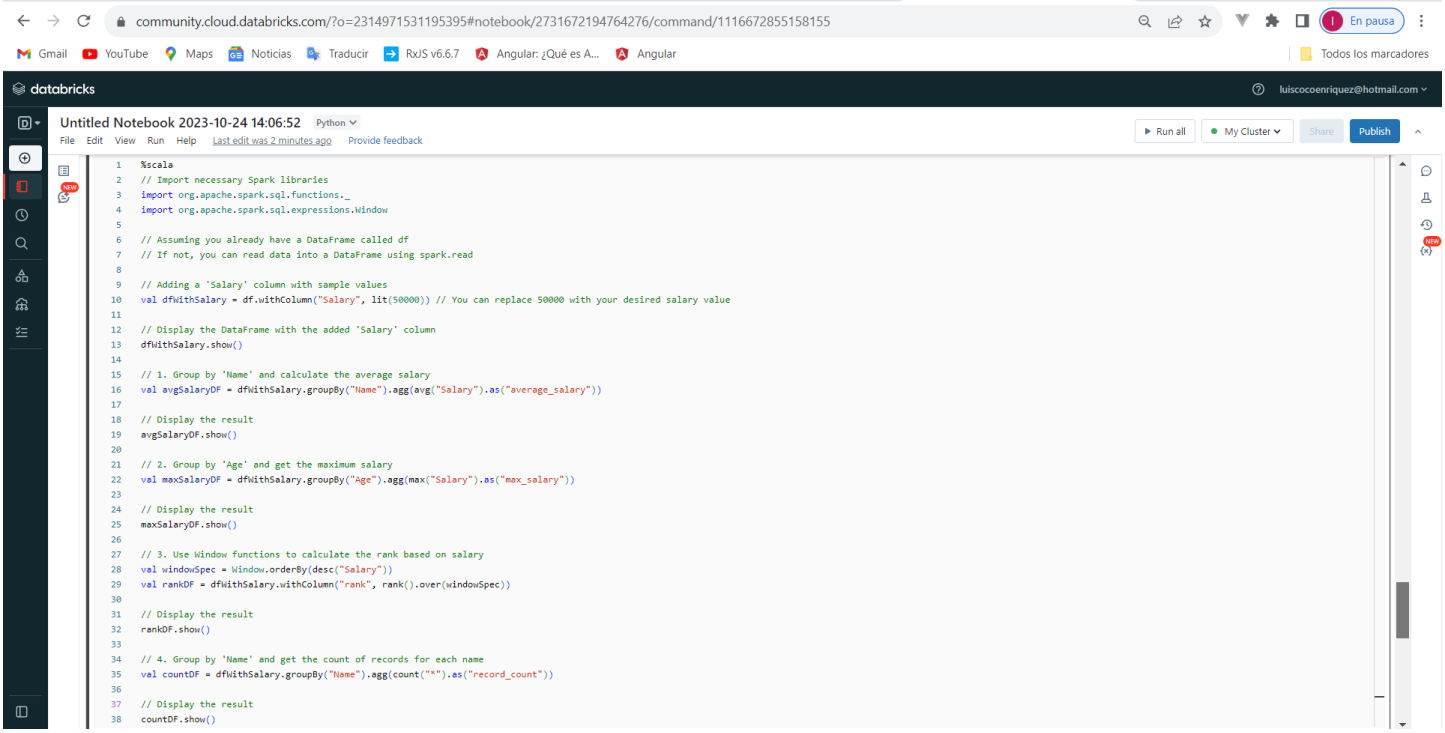
groupBy is used to specify the columns for grouping.

agg is used to perform aggregate functions like avg, max, count, etc.

Window functions (like rank in this case) are used for operations that involve sorting and ranking within partitions.

Remember to adjust column names and aggregation functions based on your actual DataFrame structure and requirements.

2.8. DataFrame Aggregations. Exercises




```
(8) Spark Jobs
dfWithSalary:org.apache.spark.sql.DataFrame = [Name: string, Age: integer ... 1 more field]
avgSalaryDF:org.apache.spark.sql.DataFrame = [Name: string, average_salary: double]
maxSalaryDF:org.apache.spark.sql.DataFrame = [Age: integer, max_salary: integer]
rankDF:org.apache.spark.sql.DataFrame = [Name: string, Age: integer ... 2 more fields]
countDF:org.apache.spark.sql.DataFrame = [Name: string, record_count: long]
+-------+---+------+
|   Name|Age|Salary|
+-------+---+------+
|  Alice| 25| 50000|
|    Bob| 30| 50000|
|Charlie| 22| 50000|
+-------+---+------+

+-------+--------------+
|   Name|average_salary|
+-------+--------------+
|  Alice|       50000.0|
|    Bob|       50000.0|
|Charlie|       50000.0|
+-------+--------------+

+---+----------+
|Age|max_salary|
+---+----------+
| 25|     50000|
| 30|     50000|
| 22|     50000|
+---+----------+

+-------+---+------+----+
|   Name|Age|Salary|rank|
```

```
+-------+---+------+----+
|  Alice| 25| 50000|   1|
|    Bob| 30| 50000|   1|
|Charlie| 22| 50000|   1|
+-------+---+------+----+

+-------+------------+
|   Name|record_count|
+-------+------------+
|  Alice|           1|
|    Bob|           1|
|Charlie|           1|
+-------+------------+

import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.Window
dfWithSalary: org.apache.spark.sql.DataFrame = [Name: string, Age: int ... 1 more field]
avgSalaryDF: org.apache.spark.sql.DataFrame = [Name: string, average_salary: double]
maxSalaryDF: org.apache.spark.sql.DataFrame = [Age: int, max_salary: int]
windowSpec: org.apache.spark.sql.expressions.WindowSpec = org.apache.spark.sql.expressions.Win
rankDF: org.apache.spark.sql.DataFrame = [Name: string, Age: int ... 2 more fields]
countDF: org.apache.spark.sql.DataFrame = [Name: string, record_count: bigint]
```

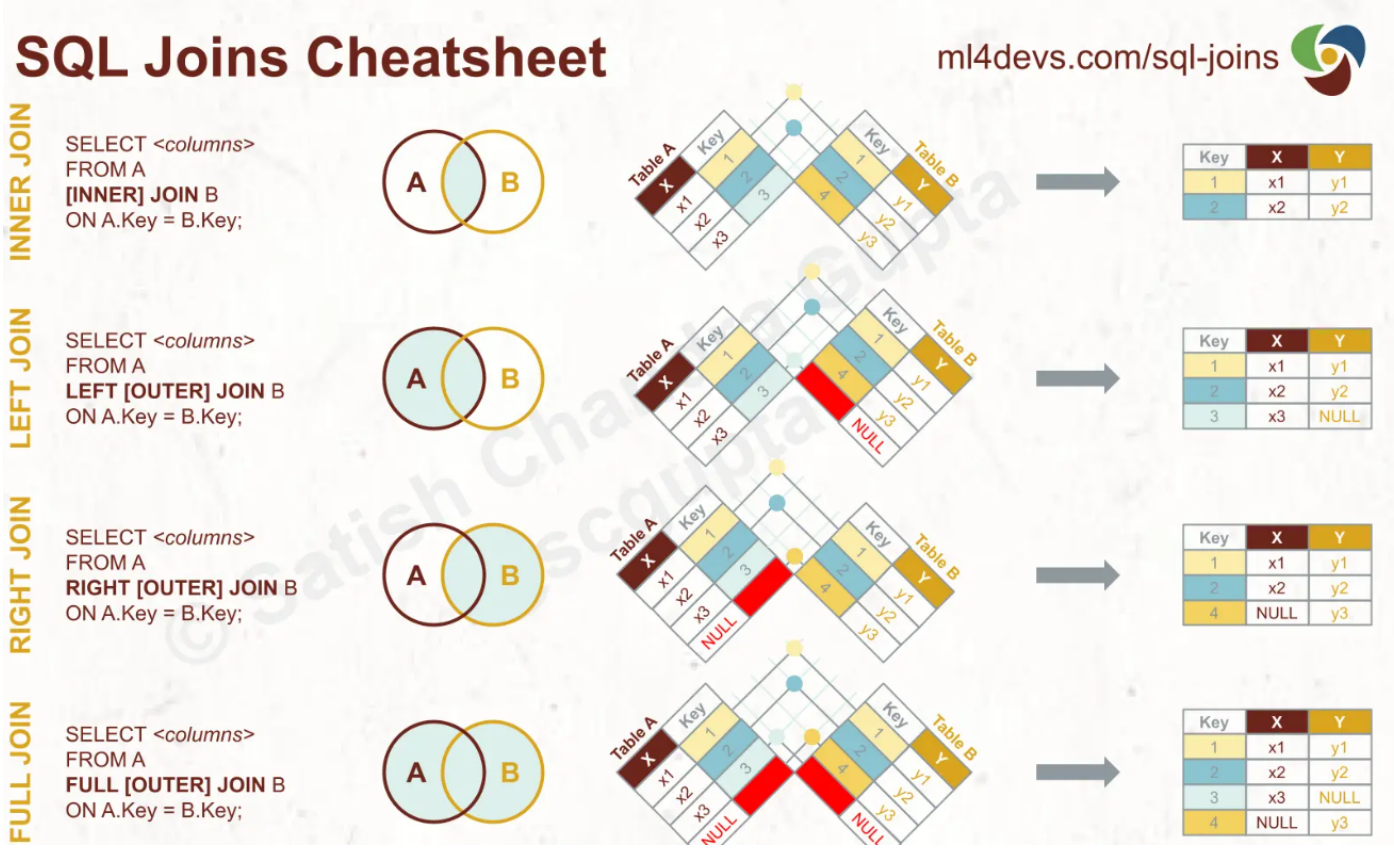
Ýou can also assign different salaries values to each row:

```
// Adding a 'Salary' column with different values based on age
val dfWithDifferentSalaries = df.withColumn(
  "Salary",
  when(col("Age") < 25, lit(45000))
    .when(col("Age") >= 25 && col("Age") < 30, lit(55000))
    .otherwise(lit(60000)) // Default salary for other cases
)
```

2.9. DataFrame Joins

In Scala Spark with DataBricks, DataFrame joins are a common operation when you want to combine two DataFrames based on a common column.

There are several types of joins available, such as inner join, outer join, left join, and right join.



Let's assume you have two DataFrames, df1 and df2, and you want to join them based on a common column, say "commonColumn".

Inner Join:

An inner join returns only the rows where there is a match in both DataFrames.

```
val resultDF = df1.join(df2, "commonColumn")
```

Left Join:

A left join returns all the rows from the left DataFrame (df1) and the matched rows from the right DataFrame (df2).

If there is no match, it fills with null values.

```
val resultDF = df1.join(df2, Seq("commonColumn"), "left")
```

Right Join:

A right join returns all the rows from the right DataFrame (df2) and the matched rows from the left DataFrame (df1).

If there is no match, it fills with null values.

```
val resultDF = df1.join(df2, Seq("commonColumn"), "right")
```

Outer Join (Full Outer Join):

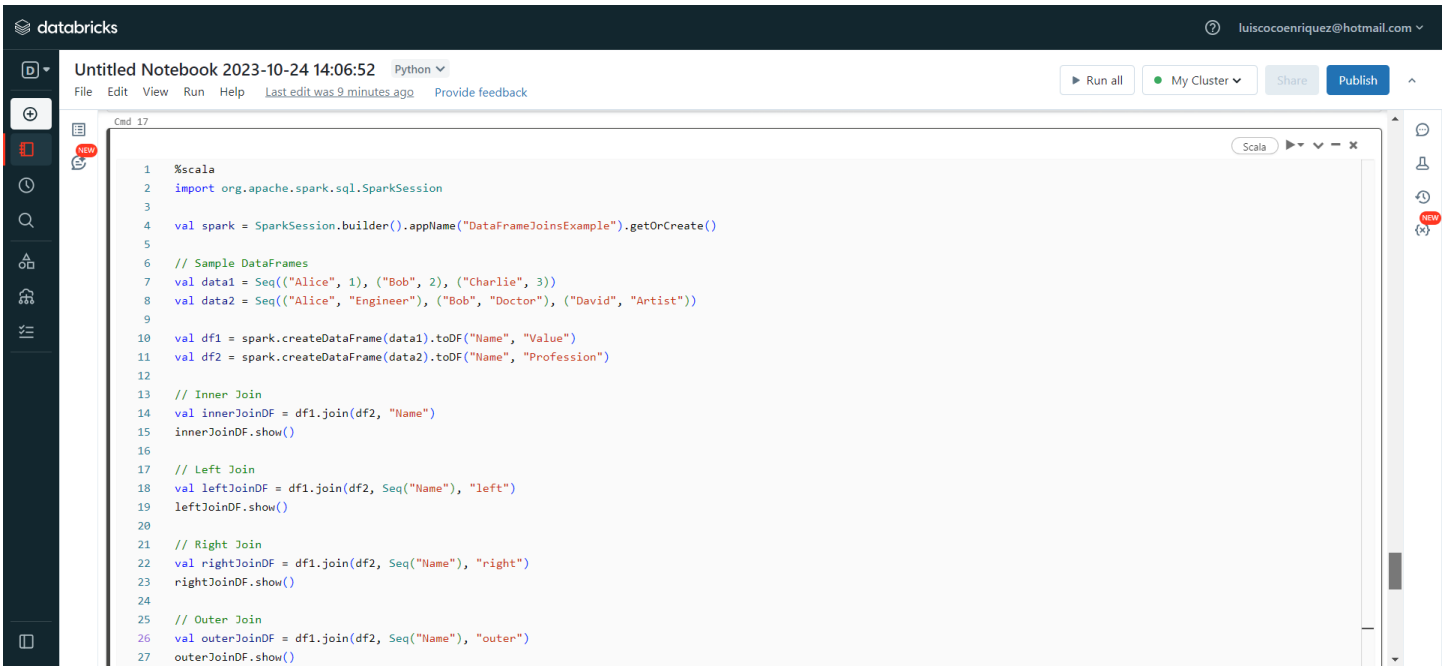
An outer join returns all the rows when there is a match in either the left or right DataFrame. If there is no match, it fills with null values.

```
val resultDF = df1.join(df2, Seq("commonColumn"), "outer")
```

Here's a simple example with some random data:

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder().appName("DataFrameJoinsExample").getOrCreate()

// Sample DataFrames
val data1 = Seq(("Alice", 1), ("Bob", 2), ("Charlie", 3))
val data2 = Seq(("Alice", "Engineer"), ("Bob", "Doctor"), ("David", "Artist"))

val df1 = spark.createDataFrame(data1).toDF("Name", "Value")
val df2 = spark.createDataFrame(data2).toDF("Name", "Profession")

// Inner Join
val innerJoinDF = df1.join(df2, "Name")
innerJoinDF.show()

// Left Join
val leftJoinDF = df1.join(df2, Seq("Name"), "left")
leftJoinDF.show()

// Right Join
val rightJoinDF = df1.join(df2, Seq("Name"), "right")
rightJoinDF.show()

// Outer Join
val outerJoinDF = df1.join(df2, Seq("Name"), "outer")
outerJoinDF.show()
```

2.10. DataFrame Joins. Exercises




```
(9) Spark Jobs
df1:org.apache.spark.sql.DataFrame = [Name: string, Value: integer]
df2:org.apache.spark.sql.DataFrame = [Name: string, Profession: string]
innerJoinDF:org.apache.spark.sql.DataFrame = [Name: string, Value: integer ... 1 more field]
leftJoinDF:org.apache.spark.sql.DataFrame = [Name: string, Value: integer ... 1 more field]
rightJoinDF:org.apache.spark.sql.DataFrame = [Name: string, Value: integer ... 1 more field]
outerJoinDF:org.apache.spark.sql.DataFrame = [Name: string, Value: integer ... 1 more field]
+-----+-----+----------+
| Name|Value|Profession|
+-----+-----+----------+
|Alice|    1|  Engineer|
|  Bob|    2|    Doctor|
+-----+-----+----------+

+-------+-----+----------+
|   Name|Value|Profession|
+-------+-----+----------+
|  Alice|    1|  Engineer|
|    Bob|    2|    Doctor|
|Charlie|    3|      null|
+-------+-----+----------+

+-----+-----+----------+
| Name|Value|Profession|
+-----+-----+----------+
|Alice|    1|  Engineer|
|  Bob|    2|    Doctor|
|David| null|    Artist|
+-----+-----+----------+

+-------+-----+----------+
|   Name|Value|Profession|
+-------+-----+----------+
|  Alice|    1|  Engineer|
```

```
|    Bob|    2|    Doctor|
|Charlie|    3|      null|
|  David| null|    Artist|
+-------+-----+----------+

import org.apache.spark.sql.SparkSession
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@427114a9
data1: Seq[(String, Int)] = List((Alice,1), (Bob,2), (Charlie,3))
data2: Seq[(String, String)] = List((Alice,Engineer), (Bob,Doctor), (David,Artist))
df1: org.apache.spark.sql.DataFrame = [Name: string, Value: int]
df2: org.apache.spark.sql.DataFrame = [Name: string, Profession: string]
innerJoinDF: org.apache.spark.sql.DataFrame = [Name: string, Value: int ... 1 more field]
leftJoinDF: org.apache.spark.sql.DataFrame = [Name: string, Value: int ... 1 more field]
rightJoinDF: org.apache.spark.sql.DataFrame = [Name: string, Value: int ... 1 more field]
outerJoinDF: org.apache.spark.sql.DataFrame = [Name: string, Value: int ... 1 more field]
```

3. Spark Types and Datasets

3.1. Working with Common Spark Data Types

Let's dive into the common Spark Data Types in Scala using DataFrames in Apache Spark with DataBricks.

Spark DataFrames are built on top of the Spark SQL engine, and they provide a distributed collection of data organized into named columns.

Here are some common Spark Data Types and examples using Scala with DataFrames:

1. StringType:

Represents strings of characters. In Scala, it's represented as StringType.

```
import org.apache.spark.sql.types._

val schema = StructType(Seq(StructField("name", StringType, true)))
```

2. IntegerType:

Represents 32-bit signed integers. In Scala, it's represented as IntegerType.

```
import org.apache.spark.sql.types._

val schema = StructType(Seq(StructField("age", IntegerType, true)))
```

3. LongType:

Represents 64-bit signed integers. In Scala, it's represented as LongType.

```
import org.apache.spark.sql.types._

val schema = StructType(Seq(StructField("salary", LongType, true)))
```

4. DoubleType:

Represents 64-bit double-precision floating-point numbers. In Scala, it's represented as DoubleType.

```
import org.apache.spark.sql.types._

val schema = StructType(Seq(StructField("rating", DoubleType, true)))
```

5. BooleanType:

Represents boolean values true or false. In Scala, it's represented as BooleanType.

```
import org.apache.spark.sql.types._

val schema = StructType(Seq(StructField("isStudent", BooleanType, true)))
```

6. ArrayType:

Represents arrays of elements. In Scala, it's represented as ArrayType.

```
import org.apache.spark.sql.types._

val schema = StructType(Seq(StructField("grades", ArrayType(IntegerType), true)))
```

7. MapType:

Represents key-value pairs. In Scala, it's represented as MapType.

```
import org.apache.spark.sql.types._

val schema = StructType(Seq(StructField("gradesMap", MapType(StringType, IntegerType), true)))
```

8. StructType:

Represents a structure or a record. In Scala, it's represented as StructType.

```
import org.apache.spark.sql.types._

val schema = StructType(Seq(
  StructField("name", StringType, true),
  StructField("age", IntegerType, true)
))
```

9. TimestampType:

Represents a timestamp with a time zone. In Scala, it's represented as TimestampType.

```
import org.apache.spark.sql.types._

val schema = StructType(Seq(StructField("eventTime", TimestampType, true)))
```

These examples demonstrate the basic usage of common Spark Data Types in Scala using DataFrames.

You can use these types to define the schema of your DataFrames when working with Spark.

Data Types example

Let's include examples of how to create DataFrames based on the specified schema and populate them with data.

```
import org.apache.spark.sql.{SparkSession, Row}
import org.apache.spark.sql.types._

// Create a Spark session
val spark = SparkSession.builder.appName("SparkExample").getOrCreate()

// Define the schema
val schema = StructType(Seq(
  StructField("name", StringType, true),
  StructField("age", IntegerType, true),
  StructField("salary", LongType, true),
  StructField("rating", DoubleType, true),
  StructField("isStudent", BooleanType, true),
  StructField("grades", ArrayType(IntegerType), true),
  StructField("gradesMap", MapType(StringType, IntegerType), true),
  StructField("eventTime", TimestampType, true)
))

// Create a DataFrame with the specified schema
val data = Seq(
```

```
    Row("John", 25, 50000L, 4.5, true, Seq(90, 85, 92), Map("Math" -> 90, "English" -> 85), java
    Row("Alice", 22, 60000L, 4.8, false, Seq(95, 88, 94), Map("Math" -> 95, "English" -> 88), ja
)

// Create the DataFrame
val df = spark.createDataFrame(spark.sparkContext.parallelize(data), schema)

// Show the DataFrame
df.show()
```

In this example, we first define the schema using StructType and then create a DataFrame (df) using the createDataFrame method.

The Row class is used to represent each row of data, and we create an RDD (Resilient Distributed Dataset) from a sequence of rows.

The show() method is then used to display the contents of the DataFrame.

```
(3) Spark Jobs
df:org.apache.spark.sql.DataFrame = [name: string, age: integer ... 6 more fields]
+-----+---+------+------+---------+------------+--------------------+-------------------+
| name|age|salary|rating|isStudent|      grades|           gradesMap|          eventTime|
+-----+---+------+------+---------+------------+--------------------+-------------------+
| John| 25| 50000|   4.5|     true|[90, 85, 92]|{Math -> 90, Engl...|2023-10-25 12:30:00|
|Alice| 22| 60000|   4.8|    false|[95, 88, 94]|{Math -> 95, Engl...|2023-10-25 13:15:00|
+-----+---+------+------+---------+------------+--------------------+-------------------+

import org.apache.spark.sql.{SparkSession, Row}
import org.apache.spark.sql.types._
spark: org.apache.spark.sql.SparkSession = org.apache.spark.sql.SparkSession@427114a9
schema: org.apache.spark.sql.types.StructType = StructType(StructField(name,StringType,true),S
data: Seq[org.apache.spark.sql.Row] = List([John,25,50000,4.5,true,List(90, 85, 92),Map(Math -
df: org.apache.spark.sql.DataFrame = [name: string, age: int ... 6 more fields]
```

3.2. Working with Complex Spark Data Types

Spark provides support for complex data types, allowing you to work with nested and structured data in a distributed computing environment.

The most common complex data types in Spark are: StructType, StructField, ArrayType, MapType, and nested StructTypes.

Let's start with **StructType** and **StructField**.

These are used to define a structure for your data, similar to a table schema. Here's an example:

```
import org.apache.spark.sql.types._

// Define a schema with two fields: name and age
val schema = StructType(Seq(
  StructField("name", StringType, true),
  StructField("age", IntegerType, true)
))

// Create a DataFrame with the defined schema
val data = Seq(("John", 25), ("Jane", 30), ("Doe", null))
val df = spark.createDataFrame(data).toDF("name", "age")

// Apply the schema to the DataFrame
val dfWithSchema = spark.createDataFrame(df.rdd, schema)

dfWithSchema.show()
```

Next, let's look at **ArrayType**. This is used for representing arrays or lists in your data:

```
// Define a schema with an array of integers
val arraySchema = StructType(Seq(
  StructField("numbers", ArrayType(IntegerType, true), true)
))

// Create a DataFrame with the defined schema
val arrayData = Seq((Seq(1, 2, 3)), (Seq(4, 5)), (null))
val arrayDF = spark.createDataFrame(arrayData).toDF("numbers")

// Apply the schema to the DataFrame
val arrayDFWithSchema = spark.createDataFrame(arrayDF.rdd, arraySchema)

arrayDFWithSchema.show()
```

Now, let's explore **MapType**, which is used to represent key-value pairs:

```
// Define a schema with a map of string keys and integer values
val mapSchema = StructType(Seq(
  StructField("info", MapType(StringType, IntegerType, true), true)
))

// Create a DataFrame with the defined schema
val mapData = Seq((Map("score" -> 90, "rank" -> 1)), (Map("score" -> 85)), (null))
val mapDF = spark.createDataFrame(mapData).toDF("info")

// Apply the schema to the DataFrame
val mapDFWithSchema = spark.createDataFrame(mapDF.rdd, mapSchema)

mapDFWithSchema.show()
```

These examples showcase the use of complex data types in Spark DataFrames using Scala.

They provide a way to represent and work with structured and nested data efficiently in a distributed computing environment.

More advanced examples using Spark's complex data types with Scala in a Databricks environment

Nested Structures with **StructType**:

```
import org.apache.spark.sql.types._

// Define a schema with nested structures
val nestedSchema = StructType(Seq(
  StructField("name", StringType, true),
  StructField("age", IntegerType, true),
  StructField("address", StructType(Seq(
    StructField("city", StringType, true),
    StructField("state", StringType, true)
  )), true)
))

// Create a DataFrame with the defined schema
val nestedData = Seq(("John", 25, ("San Francisco", "CA")), ("Jane", 30, ("New York", "NY")),
val nestedDF = spark.createDataFrame(nestedData).toDF("name", "age", "address")

// Apply the schema to the DataFrame
val nestedDFWithSchema = spark.createDataFrame(nestedDF.rdd, nestedSchema)

nestedDFWithSchema.show()
```

Arrays of StructType:

```
Copy code
// Define a schema with an array of structures
val arrayOfStructSchema = StructType(Seq(
  StructField("person", StringType, true),
  StructField("contacts", ArrayType(StructType(Seq(
    StructField("type", StringType, true),
    StructField("number", StringType, true)
  )), true), true)
))

// Create a DataFrame with the defined schema
val arrayOfStructData = Seq(("John", Seq(("email", "john@example.com"), ("phone", "123-456-789
                            ("Jane", Seq(("email", "jane@example.com"))))
val arrayOfStructDF = spark.createDataFrame(arrayOfStructData).toDF("person", "contacts")
```

```
// Apply the schema to the DataFrame
val arrayOfStructDFWithSchema = spark.createDataFrame(arrayOfStructDF.rdd, arrayOfStructSchema

arrayOfStructDFWithSchema.show()
```

MapType with Nested Structures:

```
// Define a schema with a map of string keys and nested structures as values
val mapOfStructSchema = StructType(Seq(
  StructField("attributes", MapType(StringType, StructType(Seq(
    StructField("value", StringType, true),
    StructField("unit", StringType, true)
  )), true), true)
))

// Create a DataFrame with the defined schema
val mapOfStructData = Seq((Map("height" -> ("5.8", "feet"), "weight" -> ("150", "lbs"))),
                          (Map("height" -> ("5.5", "feet"))))
val mapOfStructDF = spark.createDataFrame(mapOfStructData).toDF("attributes")

// Apply the schema to the DataFrame
val mapOfStructDFWithSchema = spark.createDataFrame(mapOfStructDF.rdd, mapOfStructSchema)

mapOfStructDFWithSchema.show()
```

These examples demonstrate more advanced use cases of complex data types in Spark DataFrames.

They include nested structures, arrays of structures, and MapType with nested structures as values, providing flexibility to handle diverse data scenarios in distributed computing environments.

3.3. Managing Nulls in Data

Dealing with null values is an essential part of data processing, and Scala Spark in Databricks provides several ways to handle them. Here are some common techniques:

Dropping Null Values:

drop method is used to eliminate rows with null values.

```
val dfWithoutNulls = originalDF.na.drop()
```

This removes any row containing at least one null value.

Filling Null Values:

fill method can be used to replace null values with specific values.

```
val dfFilled = originalDF.na.fill("default_value")
```

Replace nulls with a default value.

Imputing Null Values:

Imputation involves replacing null values with some calculated values, often the mean or median of the column.

```
val meanValue = originalDF.select(avg("column_name")).first()(0).asInstanceOf[Double]
val dfImputed = originalDF.na.fill(meanValue, Seq("column_name"))
```

Replace nulls in a specific column with the mean value.

Handling Nulls in Conditions:

You can use isNull or isNotNull functions for filtering based on null values.

```
val dfNotNull = originalDF.filter(col("column_name").isNotNull)
```

This keeps only the rows where a specific column is not null.

Coalesce:

coalesce can be used to select the first non-null value from a set of columns.

```
Copy code
val dfCoalesced = originalDF.withColumn("new_column", coalesce(col("column1"), col("column2"))
```

Create a new column with the first non-null value from two existing columns.

Remember, the choice of method depends on the specific requirements of your data analysis.

You might need to use a combination of these techniques based on the nature of your data.

3.3. More advanced topics related to managing nulls in Scala Spark on Databricks

User-Defined Functions (UDFs):

Sometimes, you may need a more complex logic to fill or transform null values.

In such cases, you can define your own functions using udf:

```
import org.apache.spark.sql.functions.udf

val customFill = udf((value: String) => if (value == null) "custom_value" else value)

val dfCustomFilled = originalDF.withColumn("new_column", customFill(col("column_name")))
```

This allows you to apply custom logic when filling or transforming nulls.

Handling Nulls in Window Functions:

When working with window functions, null values can impact the results. You can use the ignoreNulls option to handle them:

```
import org.apache.spark.sql.expressions.Window

val windowSpec = Window.partitionBy("partition_column").orderBy("order_column")

val dfWithRank = originalDF.withColumn("rank", rank().over(windowSpec).ignoreNulls())
```

This example uses the rank window function, and ignoreNulls helps in handling nulls gracefully.

Handling Nulls in Machine Learning Pipelines:

Dealing with nulls in feature columns is crucial for machine learning.

Spark ML provides a Imputer transformer to fill null values in feature columns:

```
import org.apache.spark.ml.feature.Imputer

val imputer = new Imputer()
  .setInputCols(Array("feature_column1", "feature_column2"))
  .setOutputCols(Array("imputed_feature_column1", "imputed_feature_column2"))
  .setStrategy("mean")

val model = imputer.fit(originalDF)
val dfImputedFeatures = model.transform(originalDF)
```

This is particularly useful when preparing data for machine learning models.

Handling Nested Data Structures:

If your DataFrame contains nested structures like arrays or maps, handling nulls can be more intricate.

Spark provides functions like explode, inline, and getItem for working with such structures.

```
val dfExploded = originalDF.select("column_name", explode(col("nested_array")).as("exploded_co
```

This example explodes an array column into separate rows.

```
%scala
import org.apache.spark.sql.{SparkSession, DataFrame}
import org.apache.spark.sql.functions._

// Create a Spark session
val spark = SparkSession.builder
  .appName("Explode Example")
  .master("local[2]")  // Use local mode for simplicity
  .getOrCreate()

// Create a sample DataFrame
val data = Seq(
  ("A", Seq(1, 2, 3)),
  ("B", Seq(4, 5)),
  ("C", Seq(6))
)

// Define the schema
val schema = List("column_name", "nested_array")

// Create the originalDF
val originalDF: DataFrame = spark.createDataFrame(data).toDF(schema: _*)

// Show the original DataFrame
originalDF.show()

// Apply explode to create dfExploded
val dfExploded: DataFrame = originalDF.select(col("column_name"), explode(col("nested_array")))

// Show the exploded DataFrame
dfExploded.show()
```

This is the output in DataBricks

```
+-----------+------------+
|column_name|nested_array|
+-----------+------------+
|          A|   [1, 2, 3]|
|          B|      [4, 5]|
|          C|         [6]|
+-----------+------------+

+-----------+---------------+
```

```
|column_name|exploded_column|
+-----------+---------------+
|          A|              1|
|          A|              2|
|          A|              3|
|          B|              4|
|          B|              5|
|          C|              6|
+-----------+---------------+
```

3.4. Type-Safe Data Processing: Datasets

In Apache Spark, a Dataset is a distributed collection of data that provides a higher-level API than RDDs (Resilient Distributed Datasets) and allows for strong typing.

Datasets are available in Scala and Java and offer the benefits of both DataFrames and RDDs.

Here's a brief overview of Datasets in Scala Spark with Databricks:

Creation of Dataset:

You can create a Dataset from a DataFrame or by loading data from external sources like a Parquet file, JSON file, etc.

```
// Creating a Dataset from a DataFrame
val datasetFromDataFrame = dataFrame.as[YourCaseClass]

// Loading data directly into a Dataset
val dataset = spark.read.json("path/to/json/file").as[YourCaseClass]
```

Typed API:

One of the main advantages of Datasets is the ability to use a typed API. This means that you can work with strongly-typed Scala objects instead of relying on untyped Row objects.

```
// Define a case class to represent your data structure
case class YourCaseClass(name: String, age: Int)

// Use the Dataset with the typed API
val result = dataset.filter(_.age > 21).groupBy("name").agg(avg("age"))
```

Performance:

Datasets provide better performance optimizations compared to RDDs or DataFrames due to the use of Spark's Tungsten execution engine.

The strong typing also allows for better compile-time type checking.

Compatibility with DataFrame API:

Datasets are interoperable with the DataFrame API, so you can seamlessly switch between the two.

You can convert a Dataset to a DataFrame and vice versa.

```
val df = dataset.toDF()
```

Encoders:

Datasets use encoders to convert between JVM objects and Spark SQL's internal binary format.

Spark provides encoders for most common types, and you can also create custom encoders for your specific types.

```
// Implicit encoder for case class
import org.apache.spark.sql.Encoders
implicit val yourEncoder = Encoders.product[YourCaseClass]
```

In Databricks, which is a cloud-based platform for big data analytics built on top of Apache Spark, you can work with Datasets in a collaborative environment.

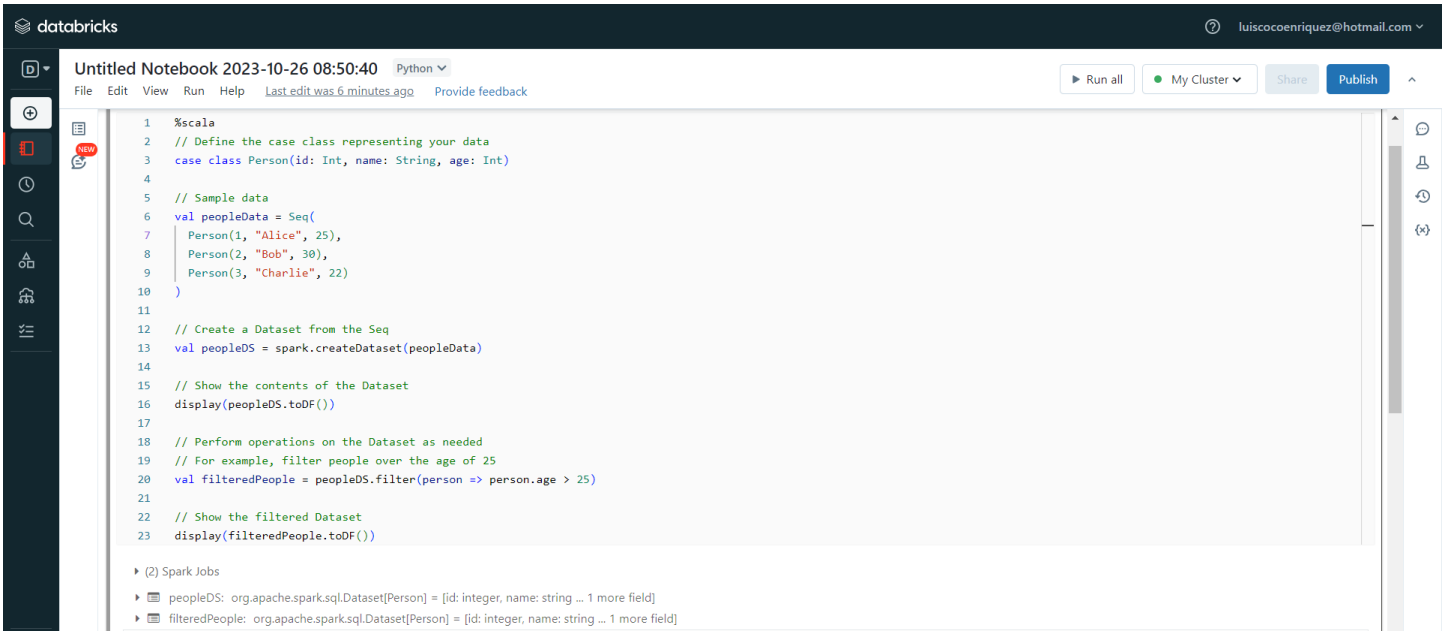
You can create notebooks, develop and test code, and visualize data using Databricks' features.

The integration is seamless, and you can take advantage of Databricks clusters for distributed computing.

3.5. Datasets Exercise

```
// Define the case class representing your data
case class Person(id: Int, name: String, age: Int)

// Sample data
val peopleData = Seq(
  Person(1, "Alice", 25),
  Person(2, "Bob", 30),
  Person(3, "Charlie", 22)
)

// Create a Dataset from the Seq
val peopleDS = spark.createDataset(peopleData)

// Show the contents of the Dataset
display(peopleDS.toDF())
```

```
// Perform operations on the Dataset as needed
// For example, filter people over the age of 25
val filteredPeople = peopleDS.filter(person => person.age > 25)

// Show the filtered Dataset
display(filteredPeople.toDF())
```



More DataSet operations samples

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.Dataset

// Define the case class
case class Person(name: String, age: Int, city: String)
case class Address(city: String, state: String)
```

```
// Sample data
val data = Seq(
  Person("John", 25, "New York"),
  Person("Alice", 30, "San Francisco"),
  Person("Bob", 28, "New York")
)

val addressData = Seq(
  Address("New York", "NY"),
  Address("San Francisco", "CA")
)

// Create Datasets
val personDataset: Dataset[Person] = data.toDS()
val addressDataset: Dataset[Address] = addressData.toDS()

// Display initial data
println("Initial Data:")
personDataset.show()

// Filtering
val filteredData = personDataset.filter(person => person.age > 25)
println("Filtered Data:")
filteredData.show()

// Mapping/Transforming
val transformedData = personDataset.map(person => Person(person.name.toUpperCase, person.age,
println("Transformed Data:")
transformedData.show()

// Grouping and Aggregation
val result = personDataset.groupBy("city").agg(avg("age"), max("age"))
println("Aggregated Result:")
result.show()

// Joining
val joinedData = personDataset.join(addressDataset, "city")
println("Joined Data:")
joinedData.show()

// Sorting
val sortedData = personDataset.sort(asc("age"))
println("Sorted Data:")
sortedData.show()

// Selecting Columns
val selectedData = personDataset.select("name", "age")
println("Selected Columns:")
selectedData.show()

// Distinct Values
val distinctData = personDataset.distinct()
println("Distinct Data:")
```

```
distinctData.show()

// Caching
personDataset.cache()
println("Data Cached")

// Counting
val count = personDataset.count()
println(s"Count: $count")

// Display final cached data
println("Final Cached Data:")
personDataset.show()

// Stop the Spark Session
spark.stop()
```

This is the output for the above code:

```
Initial Data:
+-----+---+-------------+
| name|age|         city|
+-----+---+-------------+
| John| 25|     New York|
|Alice| 30|San Francisco|
|  Bob| 28|     New York|
+-----+---+-------------+

Filtered Data:
+-----+---+-------------+
| name|age|         city|
+-----+---+-------------+
|Alice| 30|San Francisco|
|  Bob| 28|     New York|
+-----+---+-------------+

Transformed Data:
+-----+---+-------------+
| name|age|         city|
+-----+---+-------------+
| JOHN| 25|     New York|
|ALICE| 30|San Francisco|
|  BOB| 28|     New York|
+-----+---+-------------+

Aggregated Result:
+-------------+--------+--------+
|         city|avg(age)|max(age)|
+-------------+--------+--------+
|     New York|    26.5|      28|
|San Francisco|    30.0|      30|
```

```
+-------------+--------+--------+

Joined Data:
+-------------+-----+---+-----+
|         city| name|age|state|
+-------------+-----+---+-----+
|     New York| John| 25|   NY|
|San Francisco|Alice| 30|   CA|
|     New York|  Bob| 28|   NY|
+-------------+-----+---+-----+

Sorted Data:
+-----+---+-------------+
| name|age|         city|
+-----+---+-------------+
| John| 25|     New York|
|  Bob| 28|     New York|
|Alice| 30|San Francisco|
+-----+---+-------------+

Selected Columns:
+-----+---+
| name|age|
+-----+---+
| John| 25|
|Alice| 30|
|  Bob| 28|
+-----+---+

Distinct Data:
+-----+---+-------------+
| name|age|         city|
+-----+---+-------------+
| John| 25|     New York|
|Alice| 30|San Francisco|
|  Bob| 28|     New York|
+-----+---+-------------+

Data Cached
Count: 3
Final Cached Data:
+-----+---+-------------+
| name|age|         city|
+-----+---+-------------+
| John| 25|     New York|
|Alice| 30|San Francisco|
|  Bob| 28|     New York|
+-----+---+-------------+
```

3.6. More advanced topics about DataSets

Let's dive into some more advanced operations with Datasets in Scala Spark using Databricks:

Window Functions:

Window functions are powerful for performing operations over a specified range of rows related to the current row.

```
import org.apache.spark.sql.expressions.Window

val windowSpec = Window.partitionBy("city").orderBy("age")

val rankColumn = rank().over(windowSpec)
val rankedData = personDataset.withColumn("rank", rankColumn)

println("Ranked Data:")
rankedData.show()
```

UDFs (User-Defined Functions):

You can use UDFs to apply custom functions to your data.

```
import org.apache.spark.sql.functions.udf

val ageSquareUDF = udf((age: Int) => age * age)
val squaredAgeData = personDataset.withColumn("squaredAge", ageSquareUDF($"age"))

println("Squared Age Data:")
squaredAgeData.show()
```

Handling Null Values:

Dealing with null values is a common task. You can use na to handle nulls.

```
val dataWithNulls = personDataset.na.fill("Unknown", Seq("city"))

println("Data with Nulls Handled:")
dataWithNulls.show()
```

Pivot and Unpivot:

Pivot and unpivot operations are useful for transforming data.

```
val pivotedData = personDataset.groupBy("city").pivot("name").agg(avg("age"))
```

```
println("Pivoted Data:")
pivotedData.show()
```

Dynamic Partition Pruning:

Dynamic Partition Pruning helps to optimize queries by skipping unnecessary partitions.

```
spark.conf.set("spark.sql.optimizer.dynamicPartitionPruning.enabled", "true")

val prunedData = personDataset.filter($"city" === "New York" || $"city" === "San Francisco")

println("Pruned Data:")
prunedData.show()
```

Bucketing:

Bucketing is a technique to organize data into buckets based on hash functions.

```
val bucketedData = personDataset.write.bucketBy(3, "city").saveAsTable("bucketed_table")

println("Bucketed Data:")
bucketedData.show()
```

Handling JSON Data:

You can work with JSON data easily using Spark.

```
val jsonData = """
   {"name": "Eve", "age": 22, "city": "Los Angeles"}
   {"name": "Charlie", "age": 35, "city": "Seattle"}
"""

val jsonDataset = spark.read.json(Seq(jsonData).toDS())

println("JSON Data:")
jsonDataset.show()
```

These examples cover a range of advanced operations with Datasets in Scala Spark.

Each of these operations addresses different aspects of data manipulation, transformation, and optimization in Spark.

The above DataSets advance topics samples are included in this code:

```
import org.apache.spark.sql.functions._
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.Dataset

// Define the case class
case class Person(name: String, age: Int, city: String)

// Sample data
val data = Seq(
  Person("John", 25, "New York"),
  Person("Alice", 30, "San Francisco"),
  Person("Bob", 28, "New York"),
  Person("Eve", 22, "Los Angeles"),
  Person("Charlie", 35, "Seattle")
)

// Create Dataset
val personDataset: Dataset[Person] = data.toDS()

// Display initial data
println("Initial Data:")
personDataset.show()

// 1. Window Functions
val windowSpec = Window.partitionBy("city").orderBy("age")
val rankColumn = rank().over(windowSpec)
val rankedData = personDataset.withColumn("rank", rankColumn)
println("Ranked Data:")
rankedData.show()

// 2. UDFs (User-Defined Functions)
import org.apache.spark.sql.functions.udf
val ageSquareUDF = udf((age: Int) => age * age)
val squaredAgeData = personDataset.withColumn("squaredAge", ageSquareUDF($"age"))
println("Squared Age Data:")
squaredAgeData.show()

// 3. Handling Null Values
val dataWithNulls = personDataset.na.fill("Unknown", Seq("city"))
println("Data with Nulls Handled:")
dataWithNulls.show()

// 4. Pivot and Unpivot
val pivotedData = personDataset.groupBy("city").pivot("name").agg(avg("age"))
println("Pivoted Data:")
pivotedData.show()

// 5. Dynamic Partition Pruning
spark.conf.set("spark.sql.optimizer.dynamicPartitionPruning.enabled", "true")
val prunedData = personDataset.filter($"city" === "New York" || $"city" === "San Francisco")
println("Pruned Data:")
prunedData.show()
```

```
// 6. Bucketing
personDataset.write.bucketBy(3, "city").saveAsTable("bucketed_table")
println("Bucketing Done")

// 7. Handling JSON Data
val jsonData = """
   {"name": "Eve", "age": 22, "city": "Los Angeles"}
   {"name": "Charlie", "age": 35, "city": "Seattle"}
"""
val jsonDataset = spark.read.json(Seq(jsonData).toDS())
println("JSON Data:")
jsonDataset.show()

// Stop the Spark Session
spark.stop()
```

3.7. Differences between DataFrame and DataSet

In summary, both DataFrames and Datasets in Spark provide high-level, distributed data manipulation APIs.

DataFrames offer a more **SQL-like**, schema-aware interface, while **Datasets** provide a **type-safe, object-oriented programming interface** that allows you to work with custom classes.

The choice between them depends on your specific use case and the level of type safety you require.

DataFrame:

A DataFrame in Spark is an immutable distributed collection of data organized into named columns.

It represents a table of data with rows and columns, much like a traditional relational database table.

It provides a programming interface for data manipulation using a language-integrated API.

You can think of a DataFrame as an abstraction built on top of RDD (Resilient Distributed Dataset), but with more structured and optimized operations.

It allows you to perform various operations like filtering, aggregation, and joins on your data.

Dataset:

A Dataset is a distributed collection of data that provides the benefits of strong typing, expressive transformations, and functional programming.

It is an extension of the DataFrame API and is available in Spark 1.6 and later versions.

While DataFrames are limited to the types that can be represented in Spark SQL, Datasets allow you to work with custom classes and provide a type-safe, object-oriented programming interface.

Datasets can be thought of as a type-safe version of DataFrames, with the advantages of both static typing and the flexibility of DataFrames.

Datasets can be used with both Java and Scala, but the type safety is particularly beneficial in Scala.

4. Spark SQL

4.1. Spark as a "Database" with Spark SQL Shell

4.2. Spark SQL

Spark SQL is a Spark module for structured data processing that provides a programming interface for data manipulation using SQL queries.

Databricks is a cloud-based platform built on top of Apache Spark, which simplifies big data analytics.

1. Create a DataFrame:

You can create a DataFrame from a data source like a CSV file or a Parquet file.

```
// Read CSV into a DataFrame
val df = spark.read
  .option("header", true)
  .option("inferSchema", true)
  .csv("/path/to/data.csv")
```

2. Create a temporary table:

Register the DataFrame as a temporary table.

```
// Register the DataFrame as a temporary table
df.createOrReplaceTempView("people")
```

3. Run SQL queries:

Execute SQL queries on the temporary table.

```
// Run a SQL query
val result = spark.sql("SELECT * FROM people")
```

4. Perform operations with DataFrame API and SQL:

Combine SQL queries with DataFrame operations.

```
// Use SQL to filter data
val filteredData = spark.sql("SELECT * FROM people WHERE age > 25")

// Use DataFrame API to perform additional transformations
val finalResult = filteredData.groupBy("name").agg(avg("age"))
```

5. Save the result:

Save the processed data, for example, as a Parquet file.

```
// Save the result as a Parquet file
finalResult.write.parquet("/path/to/output")
```

4.3. Spark SQL More Advanced Samples

1. Window Functions:

Window functions allow you to perform calculations across a specified range of rows related to the current row.

```
import org.apache.spark.sql.expressions.Window
import org.apache.spark.sql.functions._

// Define a window specification
val windowSpec = Window.partitionBy("department").orderBy("salary")

// Calculate the rank within each department based on salary
val rankColumn = rank().over(windowSpec)

// Apply the window function to the DataFrame
val rankedDF = df.withColumn("rank", rankColumn)
```

2. User-Defined Functions (UDFs):

You can define your own functions and use them in Spark SQL.

```
// Define a UDF
val squared: Double => Double = (x: Double) => x * x
val squaredUDF = udf(squared)
```

```
// Apply the UDF in a SQL query
val result = spark.sql("SELECT name, age, squaredUDF(salary) as squaredSalary FROM people")
```

3. Working with Nested Data:

If your data has nested structures, you can use Spark SQL to query and manipulate them.

```
// Assume the DataFrame has a column named "address" which is a struct
// Extract values from the nested struct
val result = spark.sql("SELECT name, age, address.city FROM people")
```

4. Temporal Data and Date Functions:

Spark SQL provides functions for working with temporal data.

```
// Calculate age based on birthdate
val result = spark.sql("SELECT name, birthdate, DATEDIFF(current_date(), birthdate) as age FRO
```

5. Subqueries:

You can use subqueries for more complex analyses.

```
val subquery = spark.sql("SELECT department, AVG(salary) as avgSalary FROM people GROUP BY dep
val result = spark.sql("SELECT name, department, salary, subquery.avgSalary FROM people JOIN s
```

5. Low-Level Spark.

5.1. RDDs

In Apache Spark, **Resilient Distributed Datasets (RDDs)** are the fundamental data structure.

RDDs are immutable, distributed collections of objects that can be processed in parallel.

Here's a brief explanation with some Scala Spark code samples, assuming you're using DataBricks:

Creating RDDs:

You can create RDDs in various ways, such as by parallelizing an existing collection or by reading data from an external source.

```
// Parallelizing a collection to create an RDD
val data = Array(1, 2, 3, 4, 5)
val rdd = sc.parallelize(data)
// Printing the result
rdd.collect().foreach(println)

// Reading data from a file to create an RDD
val textRDD = sc.textFile("dbfs:/path/to/textfile.txt")
```

Transformations:

RDDs support two types of operations: transformations and actions.

Transformations create a new RDD from an existing one.

```
%scala
// Parallelizing a collection to create an RDD
val data = Array(1, 2, 3, 4, 5)
val rdd = sc.parallelize(data)
val transformedRDD = rdd.map(x => x * 2)

// Printing the result
transformedRDD.collect().foreach(println)
```

2 4 6 8 10

Actions:

Actions return a value to the driver program or write data to an external storage system.

```
// Reduce action: Sum all elements of the RDD
val sum = rdd.reduce((x, y) => x + y)
println(s"Sum: $sum")

// Collect action: Retrieve all elements of the RDD to the driver program
val collectedData = rdd.collect()
```

Sum: 15 sum: Int = 15

Caching:

You can persist an RDD in memory for faster reuse.

```
rdd.persist()
```

Example RDDs with DataBricks:

Here's a sample CSV file content that you can use for your code:

csv``` Name,Age,Location John,25,New York Alice,30,San Francisco Bob,28,Los Angeles Eva,35,Chicago

Now let's see the sample:

```scala
// Assuming you have a DataBricks cluster and a SparkContext (sc) is available

// Read data from a CSV file into an RDD
val csvRDD = sc.textFile("dbfs:/path/to/data.csv")

// Perform some transformations
val processedRDD = csvRDD
  .filter(line => line.contains("specificPattern"))
  .map(line => line.split(","))
  .flatMap(array => array)

// Persist the processed RDD in memory
processedRDD.persist()

// Perform an action
val count = processedRDD.count()
println(s"Count: $count")
```

Assuming the CSV file content is as provided earlier, the code is designed to count the occurrences of the "specificPattern" within the CSV file.

However, in the given code, there is no actual filtering based on a "specificPattern," so the count would represent the total number of elements in the processedRDD.

Given the sample CSV content:

```
Copy code
Name,Age,Location
John,25,New York
Alice,30,San Francisco
Bob,28,Los Angeles
Eva,35,Chicago
```

The count would be the total number of elements after the flatMap operation.

In this case, each word or element in the CSV file would be considered, and the count would be the total number of words.

Assuming no additional occurrences of "specificPattern," the output would be:

```
Count: 15
```

This is because there are 15 elements (words) in the processedRDD after the flatMap operation.

Remember, RDDs are the low-level abstraction in Spark.

In practice, DataFrames and Datasets are often preferred for structured data processing due to their higher-level abstractions and optimizations.

5.2. RDDs, Part 2 + Exercises

Here are a few more Scala Spark RDD code snippets that you can use in Databricks:

Reading and Processing Text File:

```
// Read data from a text file into an RDD
val textRDD = sc.textFile("dbfs:/path/to/textfile.txt")

// Perform transformations
val wordCountRDD = textRDD
  .flatMap(line => line.split("\\s+"))
  .map(word => (word, 1))
  .reduceByKey(_ + _)

// Display the word count
wordCountRDD.collect().foreach(println)
```

Filtering and Transformation:

```
// Read data from a CSV file into an RDD
val csvRDD = sc.textFile("dbfs:/path/to/data.csv")

// Filter and transform data
val filteredRDD = csvRDD
  .filter(line => line.contains("filterCondition"))
  .map(line => line.split(","))
  .map(array => (array(0), array(1).toInt)) // Assuming the first and second columns are strin

// Display the filtered data
filteredRDD.collect().foreach(println)
```

Joining Two RDDs:

```
// Read data from two text files into RDDs
val rdd1 = sc.textFile("dbfs:/path/to/data1.txt").map(line => (line.split(",")(0), line.split(
val rdd2 = sc.textFile("dbfs:/path/to/data2.txt").map(line => (line.split(",")(0), line.split(

// Perform inner join
val joinedRDD = rdd1.join(rdd2)

// Display the joined data
joinedRDD.collect().foreach(println)
```

Custom Transformation:

```
// Read data from a text file into an RDD
val inputRDD = sc.textFile("dbfs:/path/to/input.txt")

// Define a custom transformation function
def customTransform(line: String): String = {
  // Your custom logic here
  // This example converts the line to uppercase
  line.toUpperCase()
}

// Apply the custom transformation
val transformedRDD = inputRDD.map(customTransform)

// Display the transformed data
transformedRDD.collect().foreach(println)
```

5.2. More advanced sample for RDDs in Scala Spark with DataBricks

Here are some more advanced Scala Spark RDD code snippets that involve more complex operations:

Pair RDD Operations:

```
// Read data from a text file into an RDD
val textRDD = sc.textFile("dbfs:/path/to/textfile.txt")

// Create a pair RDD of words with their counts
val wordCountRDD = textRDD
  .flatMap(line => line.split("\\s+"))
  .map(word => (word, 1))
  .reduceByKey(_ + _)

// Find the word with the highest count
```

```
val maxWord = wordCountRDD.max()(Ordering.by(_._2))

// Display the word with the highest count
println(s"Word with the highest count: ${maxWord._1}, Count: ${maxWord._2}")
```

IMPORTANT NOTE:

This code is written in Scala and uses Apache Spark's Resilient Distributed Datasets (RDD) to perform word counting on a collection of text data.

Let's break it down step by step:

textRDD: This presumably represents an RDD (Resilient Distributed Dataset) containing lines of text.

flatMap(line => line.split("\s+")): The flatMap operation is used to split each line of text into individual words. It takes each line, applies the split("\s+") operation, which splits the line into words using whitespace as a delimiter, and then flattens the resulting sequences of words into a single sequence.

map(word => (word, 1)): The map operation transforms each word into a key-value pair (word, 1). Here, word is the word itself, and 1 is an initial count assigned to each word.

reduceByKey(_ + _): This operation is used to aggregate the counts for each unique word. It groups the key-value pairs by the key (word) and then applies the provided function (_ + _), which adds up the counts for each word.

So, in summary, the code processes a collection of text lines, splits them into words, assigns an initial count of 1 to each word, and then counts the occurrences of each unique word using the reduceByKey operation.

The result is a new RDD, wordCountRDD, where each word is paired with its count in the original text data.

Broadcast Variables:

```
// Define a broadcast variable
val broadcastVar = sc.broadcast(Array(1, 2, 3))

// Use the broadcast variable in a transformation
val rdd = sc.parallelize(Array(4, 5, 6))
val resultRDD = rdd.map(x => x + broadcastVar.value(0))

// Display the result
resultRDD.collect().foreach(println)
```

Accumulators:

```
// Define an accumulator variable
val accumulator = sc.accumulator(0)

// Read data from a text file into an RDD and perform a transformation
val textRDD = sc.textFile("dbfs:/path/to/textfile.txt")
textRDD.foreach(line => accumulator += line.split("\\s+").length)

// Display the total word count using the accumulator
println(s"Total Word Count: ${accumulator.value}")
```

Caching and Persistence:

```
Copy code
// Read data from a text file into an RDD
val textRDD = sc.textFile("dbfs:/path/to/textfile.txt")

// Perform a series of transformations
val transformedRDD = textRDD
  .flatMap(line => line.split("\\s+"))
  .map(word => (word, 1))
  .reduceByKey(_ + _)

// Persist the transformed RDD in memory
transformedRDD.persist(StorageLevel.MEMORY_ONLY)

// Perform an action
val count = transformedRDD.count()
println(s"Word Count: $count")
```

These examples showcase more advanced concepts such as pair RDD operations, broadcast variables, accumulators, and caching.