Contents

[Introduction 1](#_Toc203423275)

[Why do we need a Spark UDF ? 1](#_Toc203423276)

[Types of UDFs 1](#_Toc203423277)

[Performance Association with UDFs 1](#_Toc203423278)

[Dataset Description 2](#_Toc203423279)

[Upload The Input Dataset on HDFS 2](#_Toc203423280)

[Loading The File Into Spark Dataframe 3](#_Toc203423281)

[Weather Dataframe Exploration 3](#_Toc203423282)

[Check Data Is Loaded 3](#_Toc203423283)

[Creating and Using User-Defined Functions (UDFs) 3](#_Toc203423284)

[Step 1: Import the necessary libraries 4](#_Toc203423285)

[Step 2: Define the UDF logic 4](#_Toc203423286)

[Step 3: Convert the UDF to a SparkSQL UDF 4](#_Toc203423287)

[Step 4: Apply the UDF to our data 5](#_Toc203423288)

[Registering the UDF 5](#_Toc203423289)

[Dropping the UDF 6](#_Toc203423290)

[Summary 6](#_Toc203423291)

## Introduction

One of the most important things in Spark is that you can define your own functions called User defined Functions (UDFs). Which allows us to write our own custom transformations functions that can be used to process data in Spark DataFrames. UDFs can be written in Scala, Java, Python or R.

## Why do we need a Spark UDF ?

UDF’s are used to extend the functions of the framework and re-use this function on several DataFrame. For example if you wanted to convert the every first letter of a word in a sentence to capital case, Spark build-in features doesn’t have this function hence you can create it as UDF and reuse this as needed on many DataFrames. UDF’s are once created they can be re-use on several DataFrame’s and SQL expressions.

## Types of UDFs

Basically, there are three types of UDFs available in Spark:

* **Scalar UDFs:**  
  UDFs that take one or more input columns and return a single output column are known as Scala UDFs.
* **Aggregate UDFs:**  
  UDFs take one or more input columns and return a single output value after aggregating the input data.
* **Vector UDFs:**  
  A Vector in Spark can be defined as a dense or sparse vector of doubles that is used to represent a feature vector in machine learning applications Vector UDFs allow us to define custom functions that operate on Vector columns in Spark DataFrames.

In this tutorial we will cover the Scalar UDFs only.

### Performance Association with UDFs

Using UDFs can be an expensive process, as they may require data serialization and deserialization. Therefore, it’s important to use UDFs judiciously and only when built-in functions cannot meet your requirements.

Let’s explore these implications in detail:

1. **Serialization and Deserialization Overhead:**  
   When using UDFs, the data needs to be serialized and deserialized between the JVM and the user-defined function this leads to significant performance overhead.
2. **Garbage Collection:**  
   While using UDFs they can create temporary objects that can accumulate in the JVM heap, leading to garbage collection overhead.
3. **Resource Utilization:**  
   UDFs can easily chew up loads of your resources, such as CPU and memory. So it’s necessary you carefully tune the Spark configuration parameters, such as spark.driver.memory, spark.executor.memory, spark.executor.cores, and spark.executor.instances.
4. **Data Skew:**  
   UDFs can cause data skew, where some partitions have significantly more data than others. This can result in performance degradation and resource contention.

## Dataset Description

In this tutorial, we will be using the weather observations dataset (2021-1k.csv). The file has 8 columns but we are interested in the first 4 columns described as following:

| **Field** | **Description** |
| --- | --- |
| **ID** | 11 character station identification code. |
| **YEAR/MONTH/DAY** | 8 character date in YYYYMMDD format (e.g. 19860529 = May 29, 1986). |
| **ELEMENT** | 4 character indicator of element type. |
| **DATA VALUE** | 5 character data value for ELEMENT. |
| **M-FLAG** | 1 character Measurement Flag. |
| **Q-FLAG** | 1 character Quality Flag. |
| **S-FLAG** | 1 character Source Flag. |
| **OBS-TIME** | 4-character time of observation in hour-minute format (i.e. 0700 =7:00 am). |

The input data set contains data for one month or more daily weather data.

To execute the Spark statements of this tutorial we will be using a Zeppelin note with the **spark**interpreter (**%spark**).

%spark

sc.version

## Upload The Input Dataset on HDFS

To load the the CSV file into a Spark DataFrame, let’s upload it on HDFS. We start by creating a directory on HDFS and then put the file into this directory.

%sh

# upload data to hdfs

hdfs dfs -mkdir -p /tutorials/spark/udfs

hdfs dfs -put /home/training/Data/2021-1k.csv /tutorials/spark/udfs/

### Loading The File Into Spark Dataframe

The easiest way to load the csv file into a Spark dataframe is to use the **read.format** function (Spark SQL API). The input file has no  header. We will instruct the function to infer the schema as we didn’t provide it explicitly. The input data is coma (**,**) separated and we provide the columns names explicitly.

%spark

// load the file using the SparkSQL API

​val weather = spark.read

.format("csv")

.option("header","false")

.option("inferSchema","true")

.load("/tutorials/spark/udfs/2021-1k.csv")

.toDF("id", "date", "element", "celsius", "f1", "f2", "f3", "f4")

.cache

### Weather Dataframe Exploration

Let’s explore the newly loaded dataframe. First we will print its schema. Then we will perform a row count on the dataframe*.*

%spark

// Print the dataframe schema

weather.printSchema

### Check Data Is Loaded

Use show to verify that the data have been loaded properly. Include LIMIT to retrieve only the first 10 rows.

%spark

// Show first 10 rows

weather.show(10)

Count the number of rows in the dataframe.

%spark

// Show the row count

weather.count

## Creating and Using User-Defined Functions (UDFs)

Spark SQL provides several predefined common functions and many more new functions are added with every release. hence, It is best to check before you jump into coding your function if  the similar function you wanted is already available in Spark SQL Functions.

When you creating UDF’s you need to design them very carefully otherwise you will come across performance issues.

### Step 1: Import the necessary libraries

The **udf**function is provided by the org.apache.spark.sql.functions package, and we will use it to create our UDF.

We will start by importing the necessary libraries for our code to work:

%spark

// Import libraries

import spark.implicits.\_

import org.apache.spark.sql.functions.{udf}

### Step 2: Define the UDF logic

Next, we will define our UDF logic.

We want to create a UDF that convert an integer representing [Celsius](https://en.wikipedia.org/wiki/Fahrenheit) degrees to its corresponding value in [Fahrenheit](https://en.wikipedia.org/wiki/Fahrenheit) degrees. We will define a function that takes an Integer as an Input and returns a Double that is the Fahrenheit value. (<https://en.wikipedia.org/wiki/Fahrenheit>).

[A table with numbers and symbols

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/12/celsius-fahrenheit-formula.jpg)

%spark

// Define the UDF

def toFahrenheit (i: Int): Double = { ((i/100d) \* 9d/5d + 32d )} // Convert Clesius to Fahrenheit

In Scala we don’t use return keyword (it is optional). We need to make sure that input to this function should not be a null value.

The small **‘d’** letter indicates that these values should be represented as Double and not as Integer.

### Step 3: Convert the UDF to a SparkSQL UDF

Now that we have defined our UDF logic, we can convert it to a SparkSQL UDF using the **udf**function.

The **udf**function takes a function as an argument and returns a UDF that can be applied to a DataFrame. In this case, we pass our toFahrenheit function as an argument, and the udf function returns a UDF that we can use to convert celsius values to fahrenheit values in our DataFrame.

%spark

// Convert the UDF to SparkSQL UDF

val toFahrenheitUDF = udf(toFahrenheit \_)

### Step 4: Apply the UDF to our data

Now that we have defined our UDF, we can apply it to our data. We will add a new column to the weather DataFrame, and apply our UDF to convert each Celsius value.

The temperature values are in hundredths of a degree Celsius, but are expressed as whole integers. We will divide by 100.0 to get whole degrees Celsius and we will round the calculated / converted values to only two decimals.

%spark

// Calculate the celcius value : divide by 100.0 to get whole degrees Celsius

// Call the UDF toFahrenheitUDF and add a new column 'fahrenheit'

// Round the values to two decimals only

z.show(

weather

.withColumn("celsius", $"celsius" / 100)

.withColumn("celsius", round(col("celsius"), 2))

.withColumn("fahrenheit", toFahrenheitUDF($"celsius"))

.withColumn("fahrenheit", round(col("fahrenheit"), 2))

.select("id", "date", "element", "celsius", "fahrenheit")

//.show

)

[A screenshot of a calendar

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/12/celsius-fahrenheit-converted-columns.jpg)

As expected, each value in the “celsius” column has been converted to Fahrenheit and added to a new column “fahrenheit”.

## Registering the UDF

If we want to reuse our UDF in other parts of our Spark application, we need to register it with Spark using the spark.udf.register method. Once the UDF registered, Spark will serialize the function on the driver and transfer it over the network to all executor processes.

The first argument to spark.udf.register is the name of the UDF, and the second argument is the UDF itself.

%spark

// Register the UDF with Spark

spark.udf.register("toFahrenheitUDF", toFahrenheitUDF)

Now, we can use the registered UDF in other parts of our Spark application by referencing it by its name.  Then we create a temporary view of the DataFrame using createOrReplaceTempView, which allows us to query the DataFrame using Spark SQL.

We then write a Spark SQL query that uses our UDF to convert the ‘celsius’ column to the corresponding fahrenheit value. We use the AS keyword to rename the new column to “fahrenheit” and round the result to only two decimals. Finally, we show the resulting DataFrame using the show method.

%spark

// We can use the registered UDF

weather.createOrReplaceTempView("weather")

spark.sql("SELECT celsius, round(toFahrenheitUDF(celsius), 2) as fahrenheit FROM weather")

.show

[A screenshot of a paper

AI-generated content may be incorrect.](http://localhost/wp-content/uploads/2023/12/celsius-fahrenheit-registered-converted-columns.jpg)

## Dropping the UDF

Regular memory maintenance is key to running your Spark applications smoothly. One essential good practice is dropping unused Spark SQL views/tables and UDFs.

You drop a user-defined function (UDF) using the DROP Function command and drop a View/Table using the Drop command.

%spark

// Cleanup

spark.sql("drop temporary function toFahrenheitUDF")

weather.drop("weather")

Stop Spark context to free resources.

%spark

sc.stop

## Summary

In summary, creating a UDF in Spark is straightforward in Scala. We first define the UDF logic, convert it to a Spark UDF using the **udf** function, and apply it to our data using the withColumn method. We can also register the UDF with Spark and use it in Spark SQL queries.

Open Zeppelin Note

[Using User-Defined Functions In Spark](http://localhost:19995/#/notebook/2JG2JSGCK)