I'm not a bot

User defined function

PySpark UDF (a.k.a User Defined Function) is the most useful feature of Spark SQL & DataFrame that is used to extend the PySpark build in capabilities. In this article, I will explain what is UDF? why do we need it and how to create and use it on DataFrame select(), withColumn() and SQL using PySpark (Spark with Python) examples. Note: UDF's are the most expensive operations hence use them only you have no choice and when essential. In the later section of the article, I will explain why using UDF's is an expensive operation in detail. Table of contents UDF's a.k.a User Defined Functions, If you are coming from SQL background, UDF's are nothing new to you as most of the traditional RDBMS databases support User Defined Functions, these functions need to register in the database library and use them on SQL as regular functions. PySpark UDF's are similar to UDF on traditional databases. In PySpark, you create a function in a Python syntax and wrap it with PySpark SQL udf() or register it as udf and use it on DataFrame and SQL respectively. UDF's are used to extend the functions of the framework and re-use these functions on multiple DataFrame's. For example, you wanted to convert every first letter of a word in a name string to a capital case; PySpark build-in features don't have this function hence you can create it a UDF and reuse this as needed on many Data Frames. UDF's are once created they can be re-used on several DataFrame's and SQL expressions. Before you create any UDF, do your research to check if the similar function you wanted is already available in Spark SQL Functions. PySpark SQL provides several predefined common functions and many more new functions are added with every release. hence, It is best to check before you reinventing the wheel. When you creating UDF's you need to design them very carefully otherwise you will come across optimization & performance issues. Before we jump in creating a UDF, first let's create a PySpark DataFrame. from pyspark.sql import SparkSession spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate() columns = ["Seqno","Name"] data = [("1", "john jones"), ("2", "tracey smith"), ("3", "amy sanders")] df = spark.createDataFrame(data=data,schema=columns) df.show(truncate=False) Yields below output. +-----+------------+ |Seqno|Names | +-----+------------+ |1 |john jones | |2 |tracey smith| |3 |amy sanders | +-----+------------+ The first step in creating a UDF is creating a Python function. Below snippet creates a function convertCase() which takes a string parameter and converts the first letter of every word to capital letter. UDF's take parameters of your choice and returns a value. def convertCase(str): resStr="" arr = str.split(" ") for x in arr: resStr= resStr + x[0:1].upper() + x[1:len(x)] + " " return resStr Note that there might be a better way to write this function. But for the sake of this article, I am not worried much about the performance and better ways. Now convert this function convertCase() to UDF by passing the function to PySpark SQL udf(), this function is available at org.apache.spark.sql.functions.udf package. Make sure you import this package before using it. PySpark SQL udf() function returns org.apache.spark.sql.expressions.UserDefinedFunction class object. from pyspark.sql.functions import col, udf from pyspark.sql.types import StringType # Converting function to UDF convertUDF = udf(lambda z: convertCase(z),StringType()) Note: The default type of the udf() is StringType hence, you can also write the above statement without return type. # Converting function to UDF # StringType() is by default hence not required convertUDF = udf(lambda z: convertCase(z)) Now you can use convertUDF() on a DataFrame column as a regular build-in function. df.select(col("Seqno"), \ convertUDF(col("Name")).alias("Name")) \ .show(truncate=False) This results below output. +-----+------------+ |Seqno|Name | +-----+------------+ |1 |John Jones | |2 |Tracey Smith | |3 |Amy Sanders | +-----+------------+ You could also use udf on DataFrame withColumn() function, to explain this I will create another upperCase() function which converts the input string to upper case. def upperCase(str): return str.upper() Let's convert upperCase() python function to UDF and then use it with DataFrame withColumn(). Below example converts the values of "Name" column to upper case and creates a new column "Curated Name" upperCaseUDF = udf(lambda z:upperCase(z),StringType()) df.withColumn("Cureated Name", upperCaseUDF(col("Name"))) \ .show(truncate=False) This yields below output. +-----+----------+-----------+ |Seqno|Name |Cureated Name| +-----+----------+-----------+ |1 |john jones |JOHN JONES | |2 |tracey smith|TRACEY SMITH | |3 |amy sanders |AMY SANDERS | +-----+----------+-----------+ In order to use convertCase() function on PySpark SQL, you need to register the function with PySpark by using spark.udf.register(). """ Using UDF on SQL """ spark.udf.register("convertUDF", convertCase,StringType()) df.createOrReplaceTempView("NAME_TABLE") spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE") \ .show(truncate=False) This yields the same output as 3.1 example. In the previous sections, you have learned creating a UDF is a 2 step process, first, you need to create a Python function, second convert function to UDF using SQL udf() function, however, you can avoid these two steps and create it with just a single step by using annotations. @udf(returnType=StringType()) def upperCase(str): return str.upper() df.withColumn("Cureated Name", upperCase(col("Name"))) \ .show(truncate=False) This results same output as section 3.2 One thing to aware is in PySpark/Spark does not guarantee the order of evaluation of subexpressions meaning expressions are not guarantee to evaluated left-to-right or in any other fixed order. PySpark reorders the execution for query optimization and planning hence, AND, OR, WHERE and HAVING expression will have side effects. So when you are designing and using UDF, you have to be very careful especially with null handling as these results runtime exceptions. """ No guarantee Name is not null will execute first If convertUDF(Name) like '%John%' execute first then you will get runtime error """ spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE " + \ "where Name is not null and convertUDF(Name) like '%John%'") \ .show(truncate=False) UDF's are error-pron whe non not designed carefully. for example, when you have a column that contains the value null on some records """ null check """ columns = ["Seqno","Name"] data = [("1", "john jones"), ("2", "tracey smith"), ("3", "amy sanders"), ('4',None)] df2 = spark.createDataFrame(data=data,schema=columns) df2.show(truncate=False) df2.createOrReplaceTempView("NAME_TABLE2") spark.sql("select convertUDF(Name) from NAME_TABLE2") \ .show(truncate=False) Note that from the above snippet, record with "Seqno 4" has value "None" for "name" column. Since we are not handling null with UDF function, using this on DataFrame returns below error. Note that in Python None is considered null. AttributeError: 'NoneType' object has no attribute 'split' at org.apache.spark.api.python.BasePythonRunner$ReaderIterator.handlePythonException(PythonRunner.scala:456) at org.apache.spark.sql.execution.python.PythonUDFRunner$$anon$1.read(PythonUDFRunner.scala:81) at org.apache.spark.sql.execution.python.PythonUDFRunner$$anon$1.read(PythonUDFRunner.scala:64) at org.apache.spark.api.python.BasePythonRunner$ReaderIterator.hasNext(PythonRunner.scala:410) at org.apache.spark.InterruptibleIterator.hasNext(InterruptibleIterator.scala:37) at scala.collection.Iterator$$anon$12.hasNext(Iterator.scala:440) Below points to remember Its always best practice to check for null inside a UDF function rather than checking for null outside. In any case, if you can't do a null check in UDF at lease use IF or CASE WHEN to check for null and call UDF conditionally. spark.udf.register("_nullsafeUDF", lambda str: convertCase(str) if not str is None else "" , StringType()) spark.sql("select _nullsafeUDF(Name) from NAME_TABLE2") \ .show(truncate=False) spark.sql("select Seqno, _nullsafeUDF(Name) as Name from NAME_TABLE2 " + \ " where Name is not null and _nullsafeUDF(Name) like '%John%'") \ .show(truncate=False) This executes successfully without errors as we are checking for null/none while registering UDF. UDFs are a black box to PySpark hence it can't apply optimization and you will lose all the optimization PySpark does on Dataframe/Dataset. When possible you should use Spark SQL built-in functions as these functions provide optimization. Consider creating UDF only when the existing built-in SQL function doesn't have it. Below is a complete UDF function example in Python import pyspark from pyspark.sql import SparkSession from pyspark.sql.functions import col, udf from pyspark.sql.types import StringType spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate() columns = ["Seqno","Name"] data = [("1", "john jones"), ("2", "tracey smith"), ("3", "amy sanders")] df = spark.createDataFrame(data=data,schema=columns) df.show(truncate=False) def convertCase(str): resStr="" arr = str.split(" ") for x in arr: resStr= resStr + x[0:1].upper() + x[1:len(x)] + " " return resStr """ Converting function to UDF """ convertUDF = udf(lambda z: convertCase(z)) df.select(col("Seqno"), \ convertUDF(col("Name")).alias("Name")) \ .show(truncate=False) def upperCase(str): return str.upper() upperCaseUDF = udf(lambda z:upperCase(z),StringType()) df.withColumn("Cureated Name", upperCaseUDF(col("Name"))) \ .show(truncate=False) """ Using UDF on SQL """ spark.udf.register("convertUDF", convertCase,StringType()) df.createOrReplaceTempView("NAME_TABLE") spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE") \ .show(truncate=False) spark.sql("select Seqno, convertUDF(Name) as Name from NAME_TABLE " + \ "where Name is not null and convertUDF(Name) like '%John%'") \ .show(truncate=False) """ null check """ columns = ["Seqno","Name"] data = [("1", "john jones"), ("2", "tracey smith"), ("3", "amy sanders"), ('4',None)] df2 = spark.createDataFrame(data=data,schema=columns) df2.show(truncate=False) df2.createOrReplaceTempView("NAME_TABLE2") spark.udf.register("_nullsafeUDF", lambda str: convertCase(str) if not str is None else "" , StringType()) spark.sql("select _nullsafeUDF(Name) from NAME_TABLE2") \ .show(truncate=False) spark.sql("select Seqno, _nullsafeUDF(Name) as Name from NAME_TABLE2 " + \ " where Name is not null and _nullsafeUDF(Name) like '%John%'") \ .show(truncate=False) This example is also available at Spark GitHub project for reference. In this article, you have learned the following PySpark UDF is a User Defined Function that is used to create a reusable function in Spark. Once UDF created, that can be re-used on multiple DataFrames and SQL (after registering). The default type of the udf() is StringType. You need to handle nulls explicitly otherwise you will see side-effects. A user-defined function is a type of function in C language that is defined by the user himself to perform some specific task. It provides code reusability and modularity to our program. User-defined functions are different from built-in functions as their working is specified by the user and no header file is required for their usage. In this article, we will learn about user-defined function, function prototype, function definition, function call, and different ways in which we can pass parameters to a function. How to use User-Defined Functions in C? To use a user-defined function, we first have to understand the different parts of its syntax. The user-defined function in C can be divided into three parts: Function PrototypeFunction DefinitionFunction CallC Function Prototype A function prototype is also known as a function declaration which specifies the function's name, function parameters, and return type. The function prototype does not contain the body of the function. It is basically used to inform the compiler about the existence of the user-defined function which can be used in the later part of the program. Syntaxreturn_type function_name (type1 arg1, type2 arg2, ... typeN argN); We can also skip the name of the arguments in the function prototype. So, return_type function_name (type1 , type2 , ... typeN); C Function Definition Once the function has been called, the function definition contains the actual statements that will be executed. All the statements of the function definition are enclosed within { } braces. Syntaxreturn_type function_name (type1 arg1, type2 arg2 .... typeN argN) { // actual statements to be executed // return value if any } Note: If the function call is present after the function definition, we can skip the function prototype part and directly define the function. C Function Call In order to transfer control to a user-defined function, we need to call it. Functions are called using their names followed by round brackets. Their arguments are passed inside the brackets. Syntaxfunction_name(arg1, arg2, ... argN);Example of User-Defined Function The following C program illustrates how to use user-defined functions in our program. C // C Program to illustrate the use of user-defined function #include // Function prototype int sum(int, int); // Function definition int sum(int x, int y) { int sum; sum = x + y; return x + y; } // Driver code int main() { int x = 10, y = 11; // Function call int result = sum(x, y); printf("Sum of %d and %d = %d ", x, y, result); return 0; } OutputSum of 10 and 11 = 21 Components of Function Definition There are three components of the function definition: Function ParametersFunction BodyReturn Value1. Function Parameters Function parameters (also known as arguments) are the values that are passed to the called function by the caller. We can pass none or any number of function parameters to the function. We have to define the function name and its type in the function definition and we can only pass the same number and type of parameters in the function call. Example int foo (int a, int b); Here, a and b are function parameters. Note: C language provides a method using which we can pass variable number of arguments to the function. Such functions are called variadic function. 2. Function Body The function body is the set of statements that are enclosed within { } braces. They are the statements that are executed when the function is called. Example int foo (int a, int b) { int sum = a + b; return sum; } Here, the statements between { and } is function body. 3. Return Value The return value is the value returned by the function to its caller. A function can only return a single value and it is optional. If no value is to be returned, the return type is defined as void. The return keyword is used to return the value from a function. Syntax return (expression); Example int foo (int a, int b) { return a + b; } Note: We can use pointers or structures to return multiple values from a function in C. Passing Parameters to User-Defined Functions We can pass parameters to a function in C using two methods: Call by ValueCall by Reference1. Call by value In call by value, a copy of the value is passed to the function and changes that are made to the function are not reflected back to the values. Actual and formal arguments are created in different memory locations. Example C // C program to show use of // call by value #include void swap(int a, int b) { int temp = a; a = b; b = temp; } // Driver code int main() { int x = 10, y = 20; printf("Values of x and y before swap are: %d, %d", x, y); swap(x, y); printf("Values of x and y after swap are: %d, %d", x, y); return 0; } OutputValues of x and y before swap are: 10, 20 Values of x and y after swap are: 10, 20 Note: Values aren't changed in the call by value since they aren't passed by reference. 2. Call by Reference In a call by Reference, the address of the argument is passed to the function, and changes that are made to the function are reflected back to the values. We use the pointers of the required type to receive the address in the function. Example C // C program to implement // Call by Reference #include void swap(int* a, int* b) { int temp = *a; *a = *b; *b = temp; } // Driver code int main() { int x = 10, y = 20; printf("Values of x and y before swap are: %d, %d", x, y); swap(&x, &y); printf("Values of x and y after swap are: %d, %d", x, y); return 0; } OutputValues of x and y before swap are: 20, 10 For more details, refer to this article - Difference between Call by Value and Call by Reference Advantages of User-Defined Functions The advantages of using functions in the program are as follows: One can avoid duplication of code in the programs by using functions. Code can be written more quickly and be more readable as a result.Code can be divided and conquered using functions. This process is known as Divide and Conquer. It is difficult to write large amounts of code within the main function, as well as testing and debugging. Our one task can be divided into several smaller sub-tasks by using functions, thus reducing the overall complexity.For example, when using pow, sqrt, etc. in C without knowing how it is implemented, one can hide implementation details with functions.With little to no modifications, functions developed in one program can be used in another, reducing the development time. A user-defined function is a type of function in C language that is defined by the user himself to perform some specific task. It provides code reusability and modularity to our program. User-defined functions are different from built-in functions as their working is specified by the user and no header file is required for their usage. In this article, we will learn about user-defined function, function prototype, function definition, function call, and different ways in which we can pass parameters to a function. How to use User-Defined Functions in C? To use a user-defined function, we first have to understand the different parts of its syntax. The user-defined function in C can be divided into three parts: Function PrototypeFunction DefinitionFunction CallC Function Prototype A function prototype is also known as a function declaration which specifies the function's name, function parameters, and return type. The function prototype does not contain the body of the function. It is basically used to inform the compiler about the existence of the user-defined function which can be used in the later part of the program. Syntaxreturn_type function_name (type1 arg1, type2 arg2, ... typeN argN); We can also skip the name of the arguments in the function prototype. So, return_type function_name (type1 , type2 , ... typeN); C Function Definition Once the function has been called, the function definition contains the actual statements that will be executed. All the statements of the function definition are enclosed within { } braces. Syntaxreturn_type function_name (type1 arg1, type2 arg2 .... typeN argN) { // actual statements to be executed // return value if any } Note: If the function call is present after the function definition, we can skip the function prototype part and directly define the function. C Function Call In order to transfer control to a user-defined function, we need to call it. Functions are called using their names followed by round brackets. Their arguments are passed inside the brackets. Syntaxfunction_name(arg1, arg2, ... argN);Example of User-Defined Function The following C program illustrates how to use user-defined functions in our program. C // C Program to illustrate the use of user-defined function #include // Function prototype int sum(int, int); // Function definition int sum(int x, int y) { int sum; sum = x + y; return x + y; } // Driver code int main() { int x = 10, y = 11; // Function call int result = sum(x, y); printf("Sum of %d and %d = %d ", x, y, result); return 0; } OutputSum of 10 and 11 = 21 Components of Function Definition There are three components of the function definition: Function ParametersFunction BodyReturn Value1. Function Parameters Function parameters (also known as arguments) are the values that are passed to the called function by the caller. We can pass none or any number of function parameters to the function. We have to define the function name and its type in the function definition and we can only pass the same number and type of parameters in the function call. Example int foo (int a, int b); Here, a and b are function parameters. Note: C language provides a method using which we can pass variable number of arguments to the function. Such functions are called variadic function. 2. Function Body The function body is the set of statements that are enclosed within { } braces. They are the statements that are executed when the function is called. Example int foo (int a, int b) { int sum = a + b; return sum; } Here, the statements between { and } is function body. 3. Return Value The return value is the value returned by the function to its caller. A function can only return a single value and it is optional. If no value is to be returned, the return type is defined as void. The return keyword is used to return the value from a function. Syntax return (expression); Example int foo (int a, int b) { return a + b; } Note: We can use pointers or structures to return multiple values from a function in C. Passing Parameters to User-Defined Functions We can pass parameters to a function in C using two methods: Call by ValueCall by Reference1. Call by value In call by value, a copy of the value is passed to the function and changes that are made to the function are not reflected back to the values. Actual and formal arguments are created in different memory locations. Example C // C program to show use of // call by value #include void swap(int a, int b) { int temp = a; a = b; b = temp; } // Driver code int main() { int x = 10, y = 20; printf("Values of x and y before swap are: %d, %d", x, y); swap(x, y); printf("Values of x and y after swap are: %d, %d", x, y); return 0; } OutputValues of x and y before swap are: 10, 20 Values of x and y after swap are: 10, 20 Note: Values aren't changed in the call by value since they aren't passed by reference. 2. Call by Reference In a call by Reference, the address of the argument is passed to the function, and changes that are made to the function are reflected back to the values. We use the pointers of the required type to receive the address in the function. Example C // C program to implement // Call by Reference #include void swap(int* a, int* b) { int temp = *a; *a = *b; *b = temp; } // Driver code int main() { int x = 10, y = 20; printf("Values of x and y before swap are: %d, %d", x, y); swap(&x, &y); printf("Values of x and y after swap are: %d, %d", x, y); return 0; } OutputValues of x and y before swap are: 20, 10 For more details, refer to this article - Difference between Call by Value and Call by Reference Advantages of User-Defined Functions The advantages of using functions in the program are as follows: One can avoid duplication of code in the programs by using functions. Code can be written more quickly and be more readable as a result.Code can be divided and conquered using functions. This process is known as Divide and Conquer. It is difficult to write large amounts of code within the main function, as well as testing and debugging. Our one task can be divided into several smaller sub-tasks by using functions, thus reducing the overall complexity.For example, when using pow, sqrt, etc. in C without knowing how it is implemented, one can hide implementation details with functions.With little to no modifications, functions developed in one program can be used in another, reducing the development time.