CSC512 Project Final Report

Jingzhu He
Department of Computer Science
North Carolina State University
jhe16@ncsu.edu

Rithish Koneru
Department of Computer Science
North Carolina State University
rkoneru@ncsu.edu

1 Motivation

Spark is a big data processing framework, composing of various APIs. These APIs are designed for different data structure. For data with simple data structures, it is possible to use multiple APIs to implement. These APIs have functions working in a similar way. For example, `textFile` function in RDD API is used to build a dataset from existing text files. We find that `read.textFile` function in Dataset API has a similar usage.

Among the three APIs of Spark, RDD often has larger overhead, compared with Dataset and Dataframe APIs. Then if functions on top of RDD and the other two APIs can achieve the same result, it is better to use Dataset and Dataframe APIs. Then we want to develop a tool to scan the RDD code. If we find any piece of code in RDD can be replaced with functions on top of other APIs, we can replace this piece of code to lower the overhead. Morever, in Spark, there are various User Defined Functions (UDF), which are defined by user themselves. We find some of the UDF functions can be replaced with corresponding SQL sentences. Developers are more used to write their own Scala code without referring to the long list of the existing Spark SQL APIs. However, compared with UDF functions, the advantage is using SQL APIs can have less overhead. Then code transformation in Spark is quite useful in these two situations.

The Spark functions built on top of different APIs often have fixed formats. It inspires us to extract the token types and analyze its parse tree, in order to transform these functions across different APIs. For the simple UDF functions, the grammar of them is very easy to analyze. For example, the UDF functions can contain some mathematics operations or if-else statement. It provides the possibility to do the transformer from the UDF functions to SQL sentences.

There are some existing transformers between different languages. These transformers work at the source code level. For example, JSweet [4] is an Open Source Java to JavaScript transformer licensed under Apache Open Source License version 2. Some people are investigating different types of the Spark functions [3]. They talk about one type of function can have different formats in Spark. They list some simple examples. In addition, they talk a little about how to convert User Defined Function (UDF) into functions using standard Spark API. These exisint work inspire us to think about the possibility to develop transformer for Spark codes.

CSC512, Fall Semester, 2018
2018.

2 Objective

Spark supports three different APIs: RDD, DataFrame and Dataset [2]. Although these APIs are different in their runtime cost, most functions can be accomplished by all of the three APIs. Our goal is to translate the functions built on top of RDD API into those built on top of DataFrames and Datasets APIs. Moreover, we implement more UDF functions which output the same result with the Spark SQL sentences, if the input is the same. This is to inspire us to come up with a generalized method to replace the UDF functions with the SQL sentences when running the code. The advantage of the transformers is the much less overhead.

3 Challenges

To realize our goal, we need the solve the following challenges:

1. We are not familiar with Spark environment and Scala language. It is a little difficult for us to create the test cases of Scala functions on the three APIs and make them run smoothly. Moreover, we need to write UDF functions that has the same effect with the Spark SQL APIs, which is hard for a Spark beginner.
2. When transforming the UDF functions, we need to analyze the grammar and define the transformation rules by ourselves. There is no existing work or tools to analyze it. We need to know how the input program works. To develop a generalized rule, we need to be familiar with various variants of the input code and the code structure.
3. Replacing the tokens need to identify the start and end of the functions that are to be transformed. Developing the grammar and the parser from our scratch is also difficult for us.

4 Solutions

In this section, we introduce the detailed steps to realize the transformer for Spark. We introduce the Spark environment, how to transform RDD code to Dataset code, how to transform RDD code to Dataframe code and how to transform the UDF functions to SQL sentences.

Figure 1. Run Spark's shell on Docker container

4.1 Part 0

We choose to use well configured Docker container to set up Spark. We run the Docker container whose name is "spark". In this container with Spark, we can easily run `spark-shell` command to enter the Spark environment, as shown in Figure 1.

After building the Spark, we write six programs. Each program has three versions on top of RDD, Dataset and Dataframe APIs. Each program has same format of input. If input is same, the output remains same.

4.2 Part 1

The transformer from RDD code to Dataset code will be developed using the steps below:

1. Tokenize the given RDD program.
2. Scan the program and populate tokens.
3. Parse the program to decide which production to use.
4. Main program will be written which parses the program.

For Part 1, we have 9 transformation rules. The idea is to replace the specific identifier in the transformation rules with the corresponding one. For example, we have one RDD function and we want to convert it to Dataset function. If an identifier "sc" appears in the RDD function, we replace it with "spark" for Dataset API. We are working on it now.

We tokenize the problem into five token types, i.e., numbers, identifiers, strings, chars and symbols. It is easy to implement the tranformation rules for "sc", "range", "textFile", "map", "filter" and "collect". The difficult thing is for the tranformation rules of "reduce", "reduceByKey" and "sortBy". They contain the UDF functions. We keep the UDF functions in the transformated code. It is easy to identify the start of the three functions. We only need to identify the function names. A little difficult thing is how to identify the end of the UDF functions and add the ")". We find that at the end of the function, there is a ")". However, "()" can also exists in the UDF functions. Normally, after calling one API, it calls another API using ".". So our idea is to identify the consecutive ").". If we identify it, we replace the extension with the corresponding ones in the transformation rules. There is another possibility that the API does not call any other APIs after it. IN this case, if we identify there is no more token in the input code, we replace the extension with the corresponding ones in the transformation rules.

In this project, we only have five token types, i.e., numbers, identifiers, strings, chars and symbols. However, in the input program, there may exists some spaces or line feeds. To handle this situation, we scan the input program twice. During the first scanning, we remove all the spaces and line feeds. We only keep the five token types. During the second scanning, we do the corresponding transformation.

4.3 Part 2

To transform RDD code to Dataframe code, we need to know that Dataframe code provides some interface to use SQL sentences. If the RDD code contains UDF functions, we can identify it and replace it with SQL sentence. In our implementation, the grammar of UDF function is quite simple. It only contains some assignments, statements and if-else statements.

The grammar to analyze the UDF functions is given in the tutorial. Firstly, we need to left factor and eliminate left recursion. Specifically, we need to convert `<MapOps>`,`<TupleExpr>`, `<AssignExprs>`, ,`<SimpleExpr>` and `<PureExpr>`. They all contain left recursion. Once this is done, it was found that all the rules except <SimpleExpr> and <AssignExprsPrime> are in LL(1) i.e. intersection of FIRST+ sets is not empty. So, to tackle this problem while parsing the input program, we made use of backtracking whenever we encounter either of these non-terminals.

It is easy to implement the transformation rules, other than the transformation from UDF to SQL sentence. We only need to replace the tokens. For example, if we find there is a keyword and the token value is "sc", we replace it with "spark".

For the translation of UDF functions, we have defined our own rules as follows. Since we have already analyzed the grammar, we can easily identify the start of the UDF functions. The UDF function is composed of a set of `AssignExpr` followed by a set of `PureExpr`. The `AssignExpr` is used to define new variables. The format of `AssignExpr` is `val identifier = <PureExpr>`. The `identifier` is the new variable introduced in the UDF functions. We can use a Hashmap to store the new variables of the `AssignExpr` and the corresponding definitions, i.e., the `<PureExpr>` in the `AssignExpr`. In the transformed code, the `AssignExpr` should not be included. However, we should replace the variables defined in the `AssignExpr` with the corresponding definitions in `<PureExpr>`. For the `<PureExpr>`, each `<PureExpr>` represents one element of the new tuple which is the output of the `map` function. For example, in `map(i=>val j=i%3;(i, if(j==0)i*10 else i*2))`, it represents there are two elements mapped by an original element. The first is the same with the original element. The second contains a condition statement to determine the value. The `AssignExpr`, i.e., `val j=i%3`, introduces a new variable `j`. The definition

is i%3. We should replace j with i%3 in the output code. For each <PureExpr>, we need to count the number of occurrences in the UDF function. We add as_count at the end of the sentence. count represents the number of occurrences in the UDF function. In the UDF function, it has the format <identifier> => <Expression>. When we identify <identifier>, we replace it with "_1". For example, for input code of map(i=>val j=i%3;(i, if(j==0)i*10 else i*2)), the output code contains two elements selectExpr("_1 as _1", "if(_1%3==0,_1*10,_1*2) as _2").

We need to pay attention to the <identifier>.<identifier>. Actually, it is a special data format in Spark. For example, in map(r=>r._1+r._2), it represents we return the sum of the first and second element of the give data tuple. Because all the <identifier>.<identifier> refers to this kind of data format in the simple grammar of UDF function, we only need to replace <identifier1>.<identifier2> with <identifier2>. In the selectExpr function of Dataframe API, _1 and _2 represent the first and second element, and so on.

4.4 Part 3

In this part, we implement the Scala code to achieve the same results with the SQL APIs. We look through the webpage [1] to check the APIs' functionalities. When implementing the Scala's APIs, we need to check the input and output to make sure they have the same data type. For simplicity, some of the input type are restricted to one type. For example, hex function can take both the integer and string as input, and output the hexadecimal string. When implementation the Scala code, we can utilize some integreted packages. We can also use existing Java APIs to implement required functions.

5 Lessons and experiences

We gain a lot of experience playing with Spark in this project. We also have some useful findings through this project. When we do the transformation from RDD code to Dataset code, we only replace the tokens to realize it. We utilize some trick in the structure of the nice RDD APIs that to be transformed. However, not all the RDD APIs can be replaced with Dataset APIs by only replacing tokens. Moreover, the transformed output Dataset code contains UDF functions that is predefined by ourselves, e.g., reduceAggregator and reduceByKeyAggregator. If we want to design a generalized tool to transform RDD code to Dataset code, only replacing tokens are not enough. If we continue to expand the method we use, an important issue is that it is hard to complete the Dataset's UDF functions' creation for the equivalent RDD APIs. There is a lot of work to check each RDD API that whether there is equivalent Dataset API and write the UDF functions if not. A more flexible solution is to analyze the grammar, as we have done in Part2.

When we do the transformation from RDD code to Dataframe code, we analyze the grammar of the UDF functions and convert it to SQL sentences. It is a useful try in the transformer's development. We need to analyze the grammar and design the parser. We define the rules applicable in the UDF functions that only contain assignment, statement and if-else statements. Although it is simple, it shows the possibility to transform UDF functions. We learn to implement the grammar and the parser from our scratch. Moreover, we generate some simple rules to transform the UDF functions. In this experience, we learn how the UDF functions work and learn the functionality of RDD and Dataframe APIs.

Finally, we implement various Scala codes which are equivalent to the existing Spark SQL APIs. We learn how to compose UDF functions. Narmally, the UDF functions contain simple expressions. Compared with the UDF functions in Part 2, the difference is that we have more structures, e.g., loop and various form of input arguments. Moreover, some Scala code can use packages to implement the functions. Therefore, it is hard to design a generalized transformer to replace the Scala code with the corresponding SQL sentences.

6 Results

6.1 Part 0

In this part, we implement six programs which are built on top of RDD, DataSet and DataFrame APIs. They have exactly the same outputs when inputs are same. The six programs which are as follows.

count: This function is used for us to count how many rows a text file has. For RDD API, the function is sc.textFile ("filename").count(). filename represents the absolute of a file's directory. For Dataset API, the function is also spark.read.textFile("filename").count(). For Dataframe API, the function is still spark.read.text("filename") .count(). These three functions all read text file into a variable and return a long integer represents number of rows.

collect: This function is used to combine all the text of a file into an array. Each element in the array represents a row of the text file. For RDD API, the function is sc.textFile("filename").collect(). For Dataset API, the function is spark.read.textFile("filename") .collect(). For Daraframe API, the function is spark.read .text("filename.txt").collect().map(_.getString(0)).

first: This function is used to return the first element from the data. For RDD API, the function is sc.textFile("filename") .first(). For Dataset API, the function is spark.read .textFile("filename").first(). For Dataframe API, the function is spark.read.text("filename").first() .getString(0).

union and count: This function is used to union two data and count the elements from the combined data. For RDD API, the function is sc.textFile("filename1").union(sc.textFile("filename2")).count(). For Dataset API,

```
root@b44d4391a50a:/app# spark-shell
Spark context Web UI available at http://172.17.0.2:4040
Spark context available as 'sc' (master = local[*], app id = local-1544209589005).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_131)
Type in expressions to have them evaluated.
Type :help for more information.

scala> sc.textFile("a.txt").collect()
res0: Array[String] = Array(aaa, bbb, ccc)

scala> spark.read.textFile("a.txt").collect()
res1: Array[String] = Array(aaa, bbb, ccc)

scala> spark.read.text("a.txt").collect().map(_.getString(0))
res2: Array[String] = Array(aaa, bbb, ccc)

scala>
```

Figure 2. Three implementations of `collect` function on top of RDD, Dataset and Dataframe APIs.

the function is `spark.read.textFile("filename1").union(spark.read.textFile("filename2")).count()`. For Dataframe API, the function is `spark.read.text("filename1").union(spark.read.text("filename2")).count()`.

take: This function is used to print out the designated number of elements from the data. For RDD API, the function is `sc.textFile("filename").take(n)`. `n` represents the number of elements that need to be returned. For Dataset API, the function is `spark.read.textFile("filename").take(n)`. For Dataframe API, the function is `spark.read.text("filename").take(n).map(_.getString(0))`.

isEmpty: This function is used for check whether the input data is empty or not. For RDD API, the function is `sc.textFile("filename").isEmpty()`. For Dataset API, the function is `spark.read.textFile("filename").take(1).isEmpty`. For Dataframe API, the function is also `spark.read.text("filename").take(1).isEmpty`.

As an example, we show how the `collect` function works in Figure 2. We enter the Spark shell and input the three implementations on top of RDD, Dataset and Dataframe APIs. This function is to take all the string into a string array. We read the same text file and the output results are the same. We also pay attention to the return value type of each function and keep them same.

6.2 Part 1

We implement Part 1 on top of Java language. The program structure is based on Homework 1. We use three test cases to test it. The test cases are as follows.

Test 1: The input RDD program is `sc.range(1,10).map(i=>(i%5, 2)).reduceByKey((a:Int, b:Int) => a+b).collect()`. The output code is `spark.range(1,10).map(i=>(i%5, 2)).groupByKey(_._1).agg(reduceByKeyAggregator((a:Int, b:Int) => a+b)).collect()`.

Test 2: The input RDD program is `sc.range(1,10).sortBy((a:Long)=>(%2)).collect()`. This input code is to sort out the integer from 1 to 9 according to whether they are even and odd. Even numbers are listed before odd numbers. The output code is `spark.range(1,10).map(row =>(((a:Long)=>(a%2))(row), row)).orderBy("_1").map(_._2).collect()`

Test 3: The input RDD program is `sc.range(1,10).map(i=>(4, i+2)) .reduceByKey((a:Long, b:Long) => a-b).collect()`. In this code, the `reduceByKey` requires two long integer inputs and output just one. The output code is `spark.range(1,10).map(i=>(4,i+2)).groupByKey(_._1).agg(reduceByKeyAggregator((a:Long,b:Long)=>a-b)).collect()`

As an example, we show how Test 1 works in Figure 3. We can see both the input and output codes achieve the same results in the Spark shell.

6.3 Part 2

In this part, we use three test cases to display our transformation results. As we can see, the UDF function's grammar is restricted to a simple form, only containing if-else statement and simple expressions. To test the correctness of our program, we use multiple `AssignExpr` and `PureExpr` using different operations to achieve the goal. Our test cases and the corresponding transformated code are as follows.

Test 1: The input RDD code is `sc.range(10,100).map(i=>val j=i%3; val m=j*2; (m, if(j==0)i*5 else i, m+1)).map(r=>r._1*r._2+r._3).collect()`. In this code, it introduce two new variables, i.e., `j` and `m`. For the first `map` function, it returns a tuple with first elements, which is separated by commas. In the second `map` function, it returns the sum of the three elements. The transformed code is `spark.range(10,100).selectExpr("id as _1").selectExpr("_1%3*2 as _1", "if(_1%3==0, _1*5, _1) as _2", "_1%3*2+1 as _3").selectExpr("_1*_2 +_3 as _1").collect()`.

Test 2: The input RDD code is `sc.range(10,100).map(i=>val j=i%3; val m=3+j; (m+2, if(j==0) m else i, i, if(m!=0) m else m*2)).map(r=>r._1*r._2-r._3+r._4).collect()`. In this code, it introduce two new variables, i.e., `j` and `m`. For the first `map` function, it returns a tuple of four elements. The second and the fourth elements contain if-else statement. For the second `map`, it returns one element after mathematics operations of the four elements. The output is `spark.range(10,100).selectExpr("id as _1").selectExpr("3+_1%3+2 as _1" ,"if(_1%3==0,3+_1%3,_1) as _2","_1 as _3","if(3+_1%3!=0,3+_1%3,3+_1%3*2) as _4").selectExpr("_1*_2-_3+_4 as _1").collect()`. It matches our rules.

Test 3: The input RDD code is `sc.range(10,20) .map(i=>val j=i%3; val m=j*2; val k=m-2-j; (m+2, if(j>0) m else i, m, if(k+2>0) k else i)).map(r=>r._1-r._2+r._3*r._4) .collect()`. For the first `map` function, it returns a tuple of four elements. In the second `map` function, it does simple mathematics operations. The output is `spark.range(10,20).selectExpr("id as _1").selectExpr("_1%3*2+2 as _1" ,"if(_1%3>0,_1%3*2,_1) as _2","_1%3*2 as _3","if(_1%3*2-2+_1%3+2>0,_1%3*2-2+_1%3,_1) as`

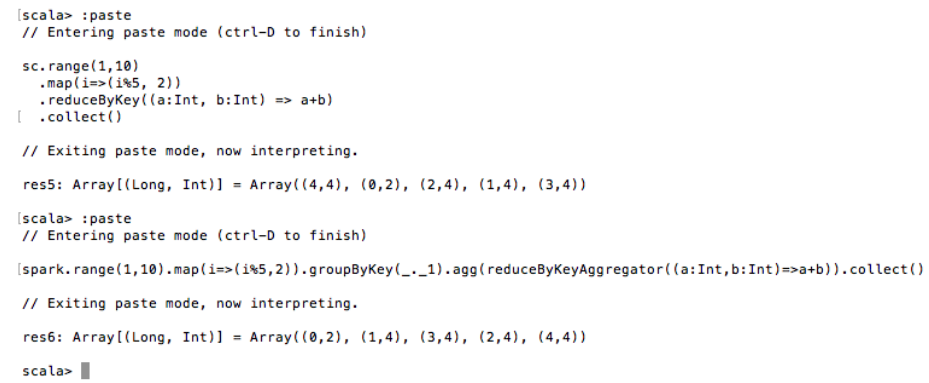

Figure 3. Result of Test 1 in Part 1.

`_4").selectExpr("_1-_2+_3*_4 as _1") .collect()`. It matches our rules.

As an example, we show how the Test 1 works in Figure **??**. We enter the Spark shell and paste the input code and the transformed code. We can see the results of RDD functions and Dataframe functions are the same. OUr test code contains a lot of operations. It proves that our transfromer can handle complicated mathematics operation and lots of statements.

6.4 Part 3

In this part, we choose 10 commonly used SQL APIs to implement the equivalent Scala function. These functions are referred to the official document of Spark SQL APIs [1]. We will describe them as follows:

concat: This function is used to concatenate all the input strings into one long string. Since the number of input string is not fixed, we use `args` to represent all the input strings. We use a loop to read each string in `args` and concatenate them together.

concat_ws: This function is used for concatenate all the input strings, separated by the first string. Our implementation is similar to the `concat` function. The difference is that we read the string from the second one. We we read another string, we add the first string to separate them.

hex: This function is used to convert the input to hexadecimal representations. Since in the tutorial the input is restricted to string, we do not need to consider about integers. When designing the Scala code, we adopt the common method to convert the string to hexadecimal string. We transfer the input string to byte arrays and convert each byte to the corresponding hexadecimal character.

repeat: This function is used to repeat the input string for several times. There are two input arguments. The first is the string and the second is the number of the repeating times. It is easy to iterate the adding string operation for the required number of repeating times.

rpad: This function is used to padding an input string to the given length. It has three input arguments, an input string, the given length of the output length and the padding string. It has two condition, if the input string's length is longer than the given length, we cut the input string to the given length using `substring` function. If the input string's length is longer than the given length, we add the left of repeated padding string to the right.

lpad: This function is similar to the `rpad` function. The difference is that the padding string is added to the left of the input string.

translate: This function is used to replace the particular character with other ones in one string. It has three input strings. The first is the input string. The second is the string that contains the characters that need to be replaced. Correspondingly, the third is the string that contains the replaced characters. For the input string, we look up the characters in the second string and replace it with the character in the third string.

reverse: This function is used to revert the given string and return it. We read the string from the last character and add it to the last of a new string.

day: The function is used to output the day of month of the given date. The input date has a string format of "YYYY-MM-DD". When we read this string. we only need to split the string by "-" and get the "DD". We convert it to integer and return it.

dayofyear: This function is used to output the day of the year for the given date. When implementing this function, since Java has the API to do it, we import the Java package and return the day as an integer.

As an example, we show the result for `concat` function in Figure 5. Firstly we enter the paste mode in the Spark shell. Then we paste the Scala implementation into it. After we exit the paste mode, Spark begin interpreting the code. We test the Scala code with three test cases. After that, we use the `concat`, the Spark SQL API to run the test cases again. The results are the same. In each function's implementation,

```
scala> :paste
// Entering paste mode (ctrl-D to finish)

sc.range(10,100)
  .map(i=>{val j=i%3; val m=j*2; (m, if(j==0)i*5 else i, m+1)})
  .map(r=>r._1*r._2+r._3)
  .collect()

// Exiting paste mode, now interpreting.

res14: Array[Long] = Array(23, 49, 1, 29, 61, 1, 35, 73, 1, 41, 85, 1, 47, 97, 1, 53, 109, 1, 59, 121, 1, 65, 133, 1, 71, 145, 1, 77, 157, 1, 83, 169, 1, 89, 181, 1, 95, 193, 1, 101, 205, 1, 107, 217, 1,
113, 229, 1, 119, 241, 1, 125, 253, 1, 131, 265, 1, 137, 277, 1, 143, 289, 1, 149, 301, 1, 155, 313, 1, 161, 325, 1, 167, 337, 1, 173, 349, 1, 179, 361, 1, 185, 373, 1, 191, 385, 1, 197, 397, 1)

scala> :paste
// Entering paste mode (ctrl-D to finish)

spark.range(10,100).selectExpr("id as _1").selectExpr("_1%3*2 as _1" ,"if(_1%3==0,_1*5,_1) as _2","_1%3*2+1 as _3")
.selectExpr("_1*_2+_3 as _1")
.collect()

// Exiting paste mode, now interpreting.

res15: Array[org.apache.spark.sql.Row] = Array([23], [49], [1], [29], [61], [1], [35], [73], [1], [41], [85], [1], [47], [97], [1], [53], [109], [1], [59], [121], [1], [65], [133], [1], [71], [145], [1],
[77], [157], [1], [83], [169], [1], [89], [181], [1], [95], [193], [1], [101], [205], [1], [107], [217], [1], [113], [229], [1], [119], [241], [1], [125], [253], [1], [131], [265], [1], [137], [277], [1],
 [143], [289], [1], [149], [301], [1], [155], [313], [1], [161], [325], [1], [167], [337], [1], [173], [349], [1], [179], [361], [1], [185], [373], [1], [191], [385], [1], [197], [397], [1])

scala>
```

Figure 4. Result of the Test 1 in Part 2.

```
root@b44d4391a50a:/app/concat# bash test.sh
Spark context Web UI available at http://172.17.0.2:4040
Spark context available as 'sc' (master = local[*], app id = local-1544136022059).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_131)
Type in expressions to have them evaluated.
Type :help for more information.

scala> :paste
// Entering paste mode (ctrl-D to finish)

def concat(args: String*): String = {
  var st: String = ""
  for( arg <- args ){
    st = st + arg
  }
  return st
}


// Exiting paste mode, now interpreting.

concat: (args: String*)String

scala>

scala> concat(" foo", "bar")
res0: String = " foobar"

scala> concat("tri", " foo", "bar ", "12#3")
res1: String = tri foobar 12#3

scala> concat("foo", " ", "bar")
res2: String = foo bar

scala>

scala> spark.sql("select concat(' foo', 'bar')").as[String].first
res3: String = " foobar"

scala> spark.sql("select concat('tri', ' foo', 'bar ', '12#3')").as[String].first
res4: String = tri foobar 12#3

scala> spark.sql("select concat('foo', ' ', 'bar')").as[String].first
res5: String = foo bar

scala> :quit
root@b44d4391a50a:/app/concat#
```

Figure 5. Result of the Scala implementation for `concat` function

we pay attention to the return type of the corresponding SQL API and keep them same.

7 Remaining issues and possible solutions

A major concern of the future work is that whether it is possible to implement a transformer to replace the Scala code with the corresponding SQL sentences. This is also what we want to study in Part 3. In my opinion, it is very hard to design a generalized tool for all the SQL APIs. However, it is possible to design a simpler transformer for some of the SQL APIs.

Designing the transformer for the Spark SQL APIs is case by case. For example, it is possible to deisgn the transformer for `concat` function. In `concat` function, it returns the concatenating string of the given string arrays. If we want to implement this code by Scala, we need to have a loop to go through the input string and use the "+" operation to append to the final string. In our transformer, if we can identify there is a string array input and we find a loop over the string array, only containing the appending operation, then we are sure the piece of Scala codes implement the `concat` function. For some Spark APIs, it is very hard to implement a transformer. For example, the `dayofyear` function returns the number of the day of the year. When implementing the Scala code, we use the Java package to achieve it. If we want to design a generalized tool, we need to identify all the possible packages in Java and Scala. Another challenge is that if we do not implement it using package, then the implementation is complicated to be analyzed. In the `dayofyear` function, we need to have a string to represent how many days a month have. The idea is to add all the days in the previous months and day of this month. Therefore, the add operation is quite different if the dates are in different month. Besides, since the day of month varies in February, it adds the difficulty to implement it.

Spark SQL API can have various forms of inputs, which make the transformer design more difficult. For example, in the `hex` function, it can not only take the string as input, but also take the integer in decimal and binary. The implementation of each type of input is different. We need to check each possibility to design the transformer. It is hard to achieve it.

In conclusion, designing a generalized transformer from UDF functions to Spark SQL sentences is difficult to achieve. Each Spark SQL API has its unique problem to solve. However, we can design the transformer for those SQL APIs that have very straight-forward implementation of Scala codes.

8 How to run the code

Our submission contains four folder, i.e., part0, part1, part2 and part3. Each part is an individual program. Please check the README.txt in each folder to see how to run it.

References

[1] 2012. Official Document of Spark SQL APIs. https://spark.apache.org/docs/2.3.0/api/sql/index.html.

[2] 2016. A Tale of Three Apache Spark APIs: RDDs, DataFrames, and Datasets. https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html.

[3] 2018. The different type of Spark functions (custom transformations, column functions, UDFs). https://medium.com/@mrpowers/the-different-type-of-spark-functions-custom-transformations-column-functions-udfs-bf556c9d0ce7.

[4] Renaud Pawlak. 2015. JSweet: insights on motivations and design.