Transforming Apache Spark APIs

CSC 512 Course Project

Monica Metro
NC State University
mgmetro@ncsu.edu

Stanton Parham
NC State University
stparham@ncsu.edu

Abstract

The Apache Spark large-data processing tool contains three different APIs. A transformer to translate from RDD API to Dataset API and from RDD API to Dataframe API were developed and tested to facilitate the Spark Community's ability to use more than one API on a large dataset. Possible extensions such as adding support for Scala code to Spark SQL API were also analyzed.

1 Introduction

1.1 What is Spark

Apache Spark is a popular large-data processing tool. [3] Its high-level APIs and additional libraries make analytics easier. It supports various processing tools stemming from generic data analysis, machine learning, stream computing, and graph execution.

1.2 Spark APIs

Three specific Spark APIs useful for operation on large datasets are RDD API, Dataset API, and Dataframe API. [6] In order to offer high-level abstraction to make processing data easier, these APIs were developed with differences in performance and expressiveness. RDD API offers the most control over dataset processing and expressiveness, but typically produces the worst performance. Dataset API offers less control than RDD API but benefits from gains in performance and space efficiency. Dataframe API is similar to the Dataset API although it offers even less control in order to produce the best possible performance through its utilization of the Spark SQL API.

Since the APIs offer different benefits, data analysts may want to switch between them when processing data. Since their syntax and keywords sometimes differ, converting between them manually would be tedious. Instead, a transformer could be used to automatically convert one API to another.

1.3 Building Transformers

To better the Spark community by making it fast and easy to convert between the three different APIs, two transformers will be developed: one which converts from RDD API to Dataset API and another which converts RDD API to Dataframe API. To achieve this goal, the following steps are required:

1. Test cases of working RDD, Dataset, and Dataframe API programs with the same functionality/output need to be created to test the accuracy of the transformers (section 3)
2. A tokenizer to be used by both transformers (section 4)
3. Creating grammar and building the transformers that translate RDD API tokens into Dataset and Dataframe API code (sections 5 and 6)

2 Related Works

2.1 Transpilers

Transpilers, or source-to-source compilers, are a popular choice in the programming community for providing higher-level functionality to well-established programming languages that tend to introduce language features at a slow pace. One area where they have gained a lot of traction is in the world of JavaScript. There are several programming languages that have been created like CoffeeScript and TypeScript whose main purpose is not to be compiled down to machine code but instead to be transpiled to JavaScript. [4] [7]

The action of transpilers transforming from one language to another is similar to our transforming of the three Spark APIs. We will be able to draw on the experience of the community that has supported the massive rise of JavaScript transpilers for information regarding how to effectively compile source code to other source code.

2.2 Compiler Homework Assignments

The programming projects we have worked on as homework assignments in CSC 512 Compiler Construction are also a reliable source of information. The three assignments present several working base products that were used as an outline for the transformers. [9] [10] [8] Since the parser utilized in the assignments was top-down and LL(1), the created transformers discussed in sections 5 and 6 were as well.

3 Test Cases

A total of 18 Spark programs were created - 6 different test cases translated into each of the three Spark APIs such that they are equivalent in functionality. The output of all three written API programs for a single test case needs to be equal

PL'18, January 01–03, 2018, New York, NY, USA
2018.

in order use the test cases for analyzing transformer accuracy.

A Spark image containing Scala version 2.12.2 and Apache Spark version 2.2.0 was used to test the output of the programs in the Spark shell via Docker.[5]

The 6 initial RDD test cases were:

1. Word Counter - count the number of words separated by spaces in a file
2. Dataframe Example - an example of a complex if-else statement parameter in .map that works with Dataframe.
3. Sort Least to Greatest - sort numbers in a range by their sine values from least to greatest
4. Reduce By Key - reduces the input by a given key
5. Even Numbers - filters out the even numbers from a range
6. Three Chainable Functions - Functions used in the previous test cases where chained together (.range, .map, .filter, .sortBy, .reduce)

3.1 Results

Out of the 6 test cases, 5 where successfully implemented in RDD, Dataset, and Dataframe API such that the output of the program in the Spark shell was equivalent in functionally to the original RDD output. The reduce by key test case was not implemented in Dataframe because a reduce by key equivalent could not be found. Two extra Dataframe test cases were created in order to test the Dataframe transformer in section 6. The Spark programs that were used for testing RDD to Dataframe transformation called sc.range.map(UDF).collect() where the UDFs contained variant syntax and a call to .map was sometimes repeated. The three tests included:

1. Dataframe Example
2. Create Tuple
3. Complex Tuple

3.1.1 Different Return Types

Due to the differences in RDD, Dataset, and Dataframe API, the return type did not always match throughout all three APIs. An example of this is the word counter test case where the programs will count the number of words in a .txt file that are separated by a space. RDD will return an Int, Dataset will return Array[Int]=Array(3) after calling .collect(), and Dataframe will return a Long. Even though the return type is not the same, for our purposes, the functionality is equivalent.

3.1.2 Non-supported API

A few test cases will not work 100% with the created transformers due to small errors with the transforming rules. Instead of fixing the transforming rules to accommodate these programs, it was decided to keep the rules simple to enable for faster implementation. The word counter test case is an example of this. The transforming rules from RDD to Dataframe only include RDD API keywords sc, .map, .range, and .collect to make building the transformer easier. RDD and Dataset API .reduce is not supported by Dataframe, therefore this test case will not be transformed correctly by the Dataframe transformer. This test case, as well as the other test cases that are supported for Dataset but not Dataframe due to the Dataframe transformer grammar being more limited, were kept anyway in order to test the transformation rules from RDD to Dataset.

The reduce by key test case was the only test case where a Dataframe equivalent could not be found due to lack of reduceByKey/reduceByKeyAggregator support. The only test case created that could work with the Dataframe transforming rules was the "Dataframe example" test case. Because of this, two extra Spark programs mentioned above were created specifically to test the Dataframe transformer for accuracy against the transforming rules.

4 Tokenizer

The transformers need a tokenizer to break the RDD API syntax into parts, e.g. keywords, symbols, and other important language elements. Those tokens can then be transformed into Dataset API or Dataframe API code. Previous homework assignments, mentioned in section 2.2, have be used as an outline to complete the full transformers in C++ language, including the tokenizer. The used token grammar is defined in "token.txt".

5 RDD to Dataset Transformer

A transformer for RDD to Dataset Spark API was built based off the C++ homework assignments' parser mentioned in section 2.2. This base parser was both LL(1) and top-down, therefore care was taken to avoid left recursion and to incorporate as much left factoring as possible. Transforming rules were utilized to map specific RDD syntax to the corresponding Dataset syntax.

Additional grammar was created for parsing through Spark UDFs (user defined functions), instead of simply copy-and-pasting the UDFs over to stop the transforming of functions with syntax errors.

5.0.1 Results

The RDD to Dataset transformer was tested via the 6 test cases created in section 3. Out of the 6 test cases, all except the word counter test case (Figure 1) were successful while using the grammar outlined in the "dataset-grammar.txt" file. Success here is defined as parsing without errors and being transformed accurately from RDD to Dataset API. The correct Dataset output was determined in 3 as the same test cases were developed for RDD, Dataset, and Dataframe API.

The word counter test case was successful after making a quick change to <field> to allow for Spark functions such as

Figure 1. Word Counter Test Case (RDD version)

```
sc.textFile("words.txt")
    .map(line => line.split(" ").size)
    .reduce( (a:Int, b:Int) => a+b )
```

Figure 2. <field> function in Dataset Grammar

Current Grammar:

```
<field> --> . <id>

          | EPSILON
```

Proposed Grammar:

```
<field> --> . <id> <field>

          | . <id> ( "string" ) <field>

          | EPSILON
```

.split and .size. A while loop was created to finish copy-and-pasting the UDF until a new Spark keyword function (.map, .sort, .reduce, etc.) was found upon finding keyword "split" or "size. To do this, <mapUDF> was set to append the UDF to the file even upon errors being found and <chainordone>, which looks for periods before Spark keyword functions, was built to call the next function even if a period was not found. This is because the while loop parsing the .map UDF didn't stop until a new Spark keyword function was found. Since the parser is LL(1) that does not allow backtracking, by the time the keyword was or was not found (in this case .split or .size was found instead) the period was passed and could not be backtracked to.

These breaks in the <mapUDF> and <chainordone> grammar may be avoidable by allowing the current <mapUDF> grammar to contain functions like .size and .split(). This would involve making <field> repeatable by having it call itself after completion, as well as creating another possibility for <field> to allow for .split("string"), such as <field> –> . <id> ("string") <field>. The current and proposed grammar are displayed in Figure 2. In addition, <field> would then have to left factored. These fixes were not completed due to time constraints.

5.0.2 Conclusions

The RDD to Dataset transformer is functional for test cases that are similar to the ones built in section 3. RDD files that contain more than keyword functions sc, textFile, map, sortBy, reduce, reduceByKey, filter, and collect will not be compatible with this transformer. Since UDFs were parsed via grammar and not just copy-and-pasted, functions that use UDFs unlike the UDFs in the test cases created in section 3 may also not be compatible with this transformer. This includes, map UDFs that contain Spark functions similar to .size and .split("string"), as only .size and .split("string") were implemented.

6 RDD to Dataframe Transformer

A transformer for RDD to Dataframe Spark API was built based off the C++ homework assignments' parser mentioned in section 2.2. This base parser was both LL(1) and top-down, therefore care was taken to avoid left recursion and to incorporate as much left factoring as possible. Transforming rules were utilized to map specific RDD syntax to the corresponding Dataframe syntax. Additional grammar was created to replace Spark UDFs (user defined functions) with Spark SQL expressions.

6.1 UDF to SQL translation

The conversion from RDD to Dataframe API was handled without using an AST tree. Although an AST tree may have been ideal, we found that it was an unnecessary construction for the transformations needed. The solution was largely similar to the approach taken for section 5 but with the key difference being how the conversion of a user defined function to SQL expressions was handled. There were a few key aspects of the conversion process that we took note of in order to craft our approach.

First, the number of SQL expressions generated from a UDF is equal to the number of elements returned at the end of the UDF. If a single expression was returned, then that would result in a single SQL expression. If a tuple of 3 elements was returned, then that would result in 3 SQL expressions. To account for this we simply generated a SQL expression for each element we saw that was returned.

Second, all declared variables in the UDF, if referencing previous variables, ultimately depend on the single parameter given to the UDF. Since the UDF param must be translated into "_1" in the SQL expressions, we realized that if we simply substituted "_1" in for the UDF param as we ran through the declared variables, then each future reference to a particular variable would be correct if we simply replaced it with itâĂŹs literal value (which will have been transformed to use "_1" instead of the UDF param at that point). With this in mind, we simply kept an array of all of the variables (name, value pairs) as we ran through the variable declarations and substituted in the order they were declared. These fully translated variables were then ready to be easily substituted into the return expressions.

Figure 3. Dataframe UDFs sorted by complexity

```
Create Tuple
.map(i=>(i%11, 1))

Complex Tuple
.map(i=>{(if(i%2==0)i else 0, i)})

Dataframe Example
.map( i => {val j=i%3; (i, if (j==0) i*10 else i*2) })
.map(r => r._1+r._2)
```

6.1.1 Results

All 3 Dataframe transformer approved test cases were successful while using the grammar outlined in the "dataframe-grammar.txt" file. Success here is defined as parsing without errors and being transformed accurately from RDD to Dataset API. The correct Dataframe output was determined in 3.

Since the grammar for the RDD to Dataframe transformer was limited, only 3 test cases were used that progress in complexity. The UDFs from these test cases are compared in Figure 3. The three test cases used included:

1. Create Tuple
2. Complex Tuple
3. Dataframe Example

Success here is defined as parsing without errors and being transformed accurately from RDD to Dataframe API. The correct Dataframe output was determined in 3 as the same test cases were developed for RDD, Dataset, and Dataframe API. The create tuple and the complex tuple test cases were specifically created due to Dataframe transforming rules only being compatible with sc, range, map, and collect Spark function calls. Only 1 of the initial 6 test cases contained only these function calls (Dataframe Example).

6.1.2 Conclusions

The RDD to Dataframe transformer is not very useful because of only accepting a small number of Spark functions. A typical Dataframe program an analyst would actually used to test a large dataset would not be this simple. However, adding additional support for more Spark functions would simply require the same steps completed for map functions calls. First, change the function call to selectExpr or another appropriate Dataframe function, then parse the UDF.

7 Extensions

7.1 Adding Support for Scala Code

Spark provides high-level APIs in Scala, Java, and Python. [3] Currently, the transformers only support simple program conversions between the APIs based on Scala. It would be beneficial to add support for transforming code from these high-level APIs to existing Spark functions to promote transition from these APIs to Spark. Code support would also promote optimization as user-defined functions may not be as efficient as Spark's implementations. The possibility of adding support for high-level APIs was analyzed with Scala since the transformers discussed in section ?? were built upon the Scala-based Spark APIs.

7.1.1 Scala APIs

Ten existing Spark functions were chosen to implement in Scala code. Brief summaries of their functionality are below:

1. concat - returns the concatenation of given strings
2. concat_ws - returns the concatenation of given strings by a given separator
3. day - returns the day of month of the current timestamp
4. last_day - returns the last day of the month which the current timestamp belongs to in form "YYYY-MM-DD"
5. lpad - returns a given string left-padded with a given pad (a string) to a given length
6. ltrim - removes leading space characters from a given string
7. repeat - returns a given string repeated the given number of times
8. reverse - returns the reversed given string
9. rpad - returns a given string left-padded with a given pad (a string) to a given length or return a shortened string if the given length < given string length
10. rtrim - removes trailing space characters from a given string

[2]

7.1.2 Results

The same Docker Spark shell that was used to test the test cases from section 3 that contains Scala version 2.12.2 and Apache Spark version 2.2.0 was used to test both the Spark functions and the Scala code. [5] The same set of at least three different test cases were completed for both the Scala functions and Spark functions to test for accuracy of the Scala implementation. For all ten functions, all the test case results match the Spark function return type and result value.

7.1.3 Function Variants

Scala, like any other high-level programming language, contains enough elements and data structures to allow for an infinite amount of code possibilities. One function with even a very specific functionality can often be programmed in numerous ways. Since pre-determining all the variants a function could possess character by character for parsing is impossible, a Scala to Spark function transformer would have to find the output of the function, find the input of the function, and compare the two to see if the functionality matches that of an existing Spark function. There are several

Figure 4. reverse Code Block Variant 1

```
def rpad(str: String, len: Int, pad: String) = {

  var ret = ""
  var i = str.size
  var diff = 0

  if( len <= str.size ) {
    ret = str.substring(0, len)
  }
  else { ...
  }
  ret
}
```

Figure 5. reverse Code Block Variant 2

```
def rpad(str: String, len: Int, pad: String) = {

  var ret = ""
  var i = str.size
  var diff = 0

  if( len <= str.size ) {
    str.substring(0, len)
  }
  else { ...
    ret
  }
}
```

problems with comparing input and output in Scala that are discussed in this section.

Loops Loops are an important element of logic flow within any programming language. Transforming from Scala to a Spark function would require being able to determine the output of a loop regardless of whether the logic used while loops or for loops. Code blocks 4 and 5 are two different implementations of Spark's reverse function in Scala. One via a while loop; the other a for loop. In this case, the final output of the function is being held in the "ret" variable. For functions like reverse where the final output is all the function requires, e.g. return the given string reversed, a parser could search for these final outputs and could compare the output to the function's input to determine which (if any) Spark function it matches. It would be unnecessary to parse the function character by character to see what was happening in the code line by line making programming elements like loop usage, variable names, etc. that differ from programmer to programmer a non-issue. Instead, one could simply throw the function inputs with expected outputs to test against. Unfortunately, finding the final output of a function is Scala is not as easy as other languages as discussed next.

Figure 6. rpad Code Block Variant 1

```
def rpad(str: String, len: Int, pad: String) = {

  var ret = ""
  var i = str.size
  var diff = 0

  if( len <= str.size ) {
    ret = str.substring(0, len)
  }
  else { ...
  }
  ret
}
```

Figure 7. rpad Code Block Variant 2

```
def rpad(str: String, len: Int, pad: String) = {

  var ret = ""
  var i = str.size
  var diff = 0

  if( len <= str.size ) {
    str.substring(0, len)
  }
  else { ...
    ret
  }
}
```

Function Return Values The Scala Glossary defines the return value as the "result of the function". [1] Results are yielded from expressions. Using this way of thinking, instead of using return statements, Scala "returns" the last expression that was executed and actual return statements are looked down upon by the community. Because of this, parsing output in Scala is more difficult than your typical language as the output of the function cannot (usually) be determined by searching for the "return" keyword. For example, code blocks 6 and 7 are two different implementations that can be used for part of the login within the Spark rpad function written in Scala.

If the given integer length of the wanted string is less than the actual string's size, the string will be shortened to the integer length and returned. Shortening a string can happen is numerous ways. The code blocks demonstrate two different ways: one where the output is saved to a variable and "returned" later when the "ret" variable is called and one where the output is not saved to a variable at all. How would a parser know that either of these expressions were the final output of a Scala function without the "return" keyword typically used? Especially, since this if statement is followed by an else statement where the bulk of the rpad code is executed. To transform the Scala implementation of rpad to Spark's rpad function, both possible return values would

have to be determined by the parser in order to determine that the code met the rpad functionality requirements of either returning a string with padding on the right side or a shortened string.

7.1.4 Conclusions

Since pre-defining all the possible ways a programmer could structure their code, Scala code to Spark API transformation would require the transformer defining function input and all possible function output. These output would then be compared to the input to see if it matches any of the Spark function definitions. For example, Spark function concat() adds any number of given strings together. The transformer would define two strings, e.g. "foo" and "bar", as the input. The transformer would then find the output "foobar" and determine that the Scala code meets the requirements for concat(). In order for this to be accomplished, the transformer would have to be able to run a Spark shell to test the Scala input, or every Spark function would have to be defined such that the Scala input could be tested in the definitions. The Scala output could then be compared to the Spark output.

7.2 RDD to Dataset API

The transformers could use extensions to add more functionality. Adding support for Scala code to Spark functions transformation as discussed above in section 7.1 is one example. More support could also be added for RDD API to Dataset API transformation.

7.2.1 Spark Function Support

The transformer for RDD to Dataset Spark API programs only works with a few select Spark functions (sc, range, textFile, filter, sortBy, reduce, reduceByKey, collect). Of course, Spark has more functions than just these. To make the transformer more usable, more functions would need to be supported.

7.2.2 Spark UDF Support

Spark UDF (user defined functions) used as a parameter while calling Spark functions was parsed via a defined grammar. Because of this, there might be a UDF syntax that is not compatible with the transformer. The word counter test case in Figure 1 is working in the transformer, but grammar rules had to be broken to allow for its UDF syntax. However, fixing the grammar for the word counter test case (Figure 2) as well as adding support for additional UDF syntax should hopefully only require small tweaks in the UDF grammar.

7.2.3 Type Errors

A third important extension to the RDD to Dataset transformer is changing the transformation rules to allow for type changes. For example, in the word counter test case 1, the parameter (a:Int, b:Int) was specifically chosen because Dataset API will throw a type error without specifying

Figure 8. Dataset Type Errors with sc.range

Original RDD:

```
sc.range(0, 10)
        .reduce( (a:Long, b:Long) => a+b)
```

Original Dataset after transformation via rules:

```
spark.range(0, 10)
        .select(reduceAggregator( (a:Long, b:Long) => a+b))
        .collect()
```

Corrected Dataset:

```
spark.range(0, 10).as[Long]
        .select(reduceAggregator((a:Long , b:Long) => a+b))
        .collect()
```

Int. This same parameter can be written as (a,b => a+b) in RDD API without problems. Currently, our transformer does not add :Int or :Long to any parameters to fix this type error. Instead, the transformer expects the UDF to work in both RDD and Dataset without change. Another example of this is displayed in Figure 8. Here, sc.range is called instead of sc.textFile. The range function must be casted as a long to work in Dataset. Since the RDD to Dataset transformation rules translate sc.range() to spark.range(), this example would not be transformed correctly by the transformer such that an error would be reached in the spark-shell. The grammar would have to be expanded to allow for type changes when transforming from RDD to Dataset.

7.3 RDD to Dataframe API

The RDD to Dataframe transforming rules only allowed for calls to RDD functions sc, range, map, and collect. Additional grammar to allow for more function calls would be beneficial. The grammar for Dataframe also only allows for specific UDF syntax. Adding support for additional UDF syntax would make the transformer more useful.

8 Conclusions

To better the Spark community by making it fast and easy to convert between the three different APIs, two transformers were developed: one which converts from RDD API to Dataset API and another which converts RDD API to Dataframe API. This was accomplished by creating test cases to test the transformers for accuracy, defining a tokenizer

to be used by both transformers, and by building the transformers off pre-defined grammar that translates RDD API tokens into Dataset and Dataframe API code.

Due to the large differences in RDD, Dataset, and Dataframe syntax and functionality, creating test cases that worked for all 3 API was challenging. There are still some syntax problems within the supported Spark function list. Two extra test cases were created for testing the Dataframe transformer since the utilized transforming rules only support RDD calls to sc, range, map, and collect.

The Dataset transformer was largely successful, however additional grammar changes would make it more complete as well as more useful. The biggest challenges would entail adding more compatible Spark functions in addition to sc, range, textFile, map, sortBy, reduce, reduceByKey, filter, and collect, as well as adding support for other Spark functions such as size and split utilized within UDFs.

The Dataframe transformer was completed after the Dataset transformer and therefore makes use of the same grammar functions for sc, map, and collect except for how the map UDF is parsed. Although the Dataframe transformer was success full, it is short on functionality due to the limited number of supported RDD function calls that can be translated to Dataframe via the transforming rules.

Both of the transformers will need to be extended further to be valuable to the Spark community.

References

[1] [n. d.]. Scala Glossary. https://docs.scala-lang.org/glossary/index.html

[2] [n. d.]. Spark Functions. https://spark.apache.org/docs/2.3.0/api/sql/index.html

[3] [n. d.]. Spark Overview. http://spark.apache.org/docs/latest/

[4] 2018. Introduction
. https://coffeescript.org/

[5] Prashanth Babu. 2016. p7hb/docker-spark. https://hub.docker.com/r/p7hb/docker-spark/

[6] Jules Damji. 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

[7] Microsoft. 2018. TypeScript. https://www.typescriptlang.org/

[8] Xipeng Shen. 2018. HW4-Part2: Code Generator.

[9] Shen Xipeng. 2018. HW1: Scanner.

[10] Shen Xipeng. 2018. HW3: Parser Programming
.