RDD to Dataframe/Dataset APIs

A study in compiling an accessible API into less expressive versions that run much more effectively

Luke Roosje
Student
Computer Science
North Carolina State University
Raleigh, North Carolina
leroosje@ncsu.edu

Qiaodan Wang
Student
Computer Science
North Carolina State University
Raleigh, North Carolina
qwang32@ncsu.edu

1 Introduction

Compiling is well known as a method of changing high level languages into machine code, and every programmer has had experience using one. However, a compiler's usefulness isn't restricted to that usual case. A more general definition is that a compiler converts some program into another; whether it be within the language, to machine or assembly, or even to a language of equal or higher complexity. With this, the applications of compilers become much more broad; not only are they a black box that allow computers to understand our higher-level code, but a valuable tool in reducing overhead, increasing speed, and converting programs into a more appropriate language. The latter application is what we are focusing on in our work, but we are focusing our compiler towards an even smaller scope; converting statements in a language into another API within that same language.

2 Background and Motivation

Our work pertains exclusively to Spark, a popular open source language that is used for data processing. This language has many APIs within it that all follow the same grammar, but use vastly different syntax and objects. The three that we focus on are the RDD, Dataset, and Dataframe APIs. Each is distinct, and has strengths and weaknesses.

RDD is the easiest to use of the three APIs. It functions by partitioning a full dataset and fulfilling queries with these partitions. It is also the most expressive of the three APIs, able to complete the most functions. The other two APIs are entirely subsumed by it. However, they are faster.

Dataframe is the second most expressive API. It works on named columns, similar to a relational database and uses commands in SQL. This is at the core of section 9, and will be explained in far more detail in that section.

Dataset is the least expressive of the three, and is subsumed by Dataframe. It uses row objects, similar to Dataframe, but they are named with JVM or Row Objects. This abstraction allows more efficient computations.

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

Our motivation in this project is to combine advantages of these three APIs, expressiveness from RDD API and performance from Dataset API and Dataframe API. To achieve tha

3 Problem

Spark offers three primary APIs: RDD, Dataframe, and Dataset. Each offers further structure than the previous to the data being processed by them. RDD uses partitions of the full dataset, Dataframe uses named columns (similar to a relational database) and Dataset uses JVM or Row objects, where rows are defined on data types where possible. Because of this increased complexity, efficiency also increases dramatically, but the expressiveness and accessibility decrease. However, while the expressiveness of these APIs are as follows: $e(Dataset) \subset e(Dataframe) \subset e(RDD)$, these subsets are not substantially smaller. Thus, most people using RDD are missing the benefits of Dataframe or Dataset.

Because of each API's caveats, each is better at something than the others; as well as worse at other things. Learning different APIs can be unfriendly, and use valuable time, and there is no guarantee that users are actually generating efficient code to represent what they know well in others. This leads to queries and operations not being as efficient as they could be, in both memory and time. In high volumes, these inefficiencies could bog down a system, and use valuable resources unnecessarily.

4 Context of the Field

The problems that govern our work are entirely within the field of compiler construction. In fact, coverting code in one API to another is within the subfield of source-to-source compilers, which are compilers that take a program in a certain language as input, and produce an equivalent program in any other given language. Unlike traditional compilers, this does not have any restrictions of moving languages down in level and abstraction; a source-to-source compiler can translate Assembly to a functionally equivilent program in java, if so desired.

One of the most important parts of source-to-source compilers is how effectively it is able to process programs. This can be achieved by following standard compiler construction techniques. This means that there will be at least a front end, composed of a scanner and parser, some IRs within or between layers, and eventually the regeneration of code that is desired. We know this is possible, because many mature source-to-source compilers already exist, such as CoffeeScript, Coccinelle, and DMS.

5 Solution

In order to give developers access to advantages of APIs they are unfamiliar with, without requiring lost time and no guarantee of efficacy, we propose that a compiler be developed that accurately transforms a given API of Spark into another.

This tool will be a static program that is fed an input file of the prescribed API that outputs a functionally equivalent program in the desired API. It will not run or compile its output into machine-readable languages, as we leave that to the standard Spark Compiler. Thus, the user will still require some knowledge of how to use the generic Spark Compiler to run different APIs , but it will be a significantly reduced overhead compared to having to develop in them.

6 Plan

As a proof of concept, we have developed two transformers that take a Spark program RDD API as input. The first outputs the same program in Dataset, and the second outputs Dataframe. Furthermore, due to scope, the transformers are only functional on a somewhat limited set of commands. These tools can be made to be much more robust with more time.

Additionally, we have written various Spark SQL commands in Scala, showing how our proposed solution could be extended further. These examples show further problems that we may encounter as well, none of which are insurmountable. Eventually, the transformations we conducted by hand can be automated.

7 Familiarization with Spark

In order to generate a tool dealing exclusively with Spark, we first had to familiarize ourselves with it. As with other languages, the grammar and syntax of this language was foreign, and it took significant time and effort to learn how to write commands in a given API. The challenge of learning Dataset and Dataframe following RDD highlighted to us why this tool is necessary at all; our research revealed the strengths of each, but accessing them was challenging.

8 Compiling RDD to Dataset API

Compiling RDD into Dataset API is the sole purpose of the first tool we created. The tool begins by using a front end to partition the input program into tokens based off a prescribed grammar, and the back end interprets this IR and outputs code appropriately.

8.1 Mechanics of Tokenization

Firstly, a token is an artifact of a program that has a specific meaning, known to the programmer and the compiler. Everything within a given program is a token, and partitioning a program into its respective tokens allows for efficient processing by latter stages of the compiler. While there are many ways to break down a program into tokens, we used the following context-free grammar, provided by Dr. Xipeng Shen.

<letter> –> a | b | ... | y | z | A | B | ... | Z | _
<digit> –> 0 | 1 | ... | 9
<number> –> <digit>+
<identifier> –> <letter> (<letter> | <digit>)*
<string> –> any string between (and including) the closest pair of double quotation marks.
<char> –> a character between (and including) a pair of single quotation marks.
<symbol> –> any non-space character that is not a part of other tokens

Something worth noting is that each production rule's name is unique and meaningful; the importance of this will be discussed later.

Implementation Implementing this tokenization was the first major problem we encountered; it was something we were unfamiliar with, and required serious time and effort in order to make it work properly. In the end, we were able to successfully adapt code from (and used with permission of) Dr. Xipeng Shen.

Tokenization works through a Scanner, a traditional tool in compilers. This scanner parses the input file, one token at a time, outputting subsequent Pairs, an object composed of a Token and its corresponding Name. The name is whichever derivation the Token is accepted by; for instance, a token with the value 'a' is stored as a Pair of 'a',char.

This greatly increases how robust the code is, and allows for individual behavior and rules for each. This is critical, because each may have a vastly different function within the language itself. For instance, an identifier called value1 may need to have its actual value and type worked on, but a string "value1" could be used in entirely different contexts.

The grammar above actually determines how the program is partitioned. The input is read one character at a time, and the character is appended to the current artifact being checked. Once it successfully matches one of the derivations, the scanner continues appending characters until it fails to pattern match. Once this occurs, the artifact minus the latest

character is returned as the value of the token, and the most recent match is returned as the name.

Our scanner returns these pairs one at a time, when called. The tokens are processed immediately; we do not wait for further tokens to be found to process them, because it is unnecessary within the scope of our project.

8.2 Transformation Rules

This processing is the stage where output code is written, and the subset of commands that we are transforming are changed. The commands that need to be transformed are as follows:

RDD API	Dataset API
sc	spark
range	range
textFile	read.textFile
map	map
filter	filter
reduce(<func>)	select(reduceAggregator (<func>)).collect()
reduceByKey(<func>)	groupByKey(_._1).agg(reduceByKeyAggregator(<func>))
sortBy(<func>)	map(row=>((<func>)(row), row)) .orderBy("_1").map(_._2)
collect	collect

These nine commands can be broken into three subsets that each have similar implementation. Notice that four of them have the same representation in both RDD and Dataset; these are treated the same as any other artifact in the input program and written directly.

The second set is those that require a "simple" transformation. The two commands, sc and textFile, can be easily transformed within the token they are found in, because the original commands are composed of only one. To conduct this transformation, we simply check a token to see if it matches, and assign its value the corresponding Dataset API command.

The remaining three commands are much more complex; <func> denotes a function within these commands that must be passed over to the Dataset API's command perfectly. This function may also be composed of many tokens, meaning we must transform over many tokens. This seems to contradict the processing model described above! However, it still works within our system. Upon finding a token that matches one of the commands, that token is assigned a new value, the substring of the Dataset command before <func> occurs. After, our scanner continues processing tokens normally, until reaching a closing parenthesis. Once that is reached, it appends the remainder of the sub-string. Once that is complete, the new code is written to the output file.

8.3 Testing

In order to verify that the transformer was working properly, we generated test cases, covering each transformation rule. They are as follows:

```
sc.textFile("test.txt").count(),
sc.textFile("test.txt").map(line => line.toInt).sum(),
sc.textFile("test.txt").sortBy(line => line.toInt,true,1).first(),
sc.textFile("test.txt").map(line=>line.toInt).max(),
sc.textFile("test.txt").map(line => line.toInt).filter(num =>
num > 5).foreach(println),
sc.range(1,101).filter(num=>num%2==0).sum(),
reduce(paiubdfpasdifasdf),
reduceByKey(ouasbdfibasdpofasd)
```

This is the output:

```
spark.read.textFile("test.txt").count(),
spark.read.textFile("test.txt").map(line => line.toInt).sum(),
spark.read.textFile("test.txt").map
(row=>((line => line.toInt,true,1)(row), row)).
orderBy(_1).map(_._2).first(),
spark.read.textFile("test.txt").map(line=>line.toInt).max(),
spark.read.textFile("test.txt").map(line => line.toInt).filter(num
=> num > 5).foreach(println),
spark.range(1,101).filter(num=>num%2==0).sum(),
select(reduceAggregator(paiubdfpasdifasdf)).collect(),
groupByKey(_._1).agg(
reduceByKeyAggregator(ouasbdfibasdpofasd))
```

Each of these test cases were successfully transformed according to the rules described above. The trickiest part is balancing the parenthesis, and tracking where tokens end and start. However, with enough time and adjustments, we managed to make it correctly.

9 Compiling RDD to Dataframe API

Compiling RDD API into Dataframe API is the goal of our second transformation tool. The definition of token is identical to the same set we used when compiling RDD to Dataset API. However the transformation rules are different, and most importantly we need to take user defined function into consideration.

User defined functions are functions defined by users, they are usually used as parameters in intermediate RDD API calls such as ***map***. Because user defined functions could be complex and there is no way to predict and enumerate all the possibilities, we need to parse the input and generate an abstract syntax tree and convert the tree to relevant Dataframe API using the transformation rules provided. There are multiple powerful tools and frameworks to parse the input language and generate the abstract syntax tree with given grammar. Because we are both comfortable programming with Java,

finally We chose ***ANTLR*** as the transformation tool to generate scanner and parser for our goal. With the help of ***ANTLR***, we can :

(1) Scan the program written in RDD API and tokenize the program on basis of token definitions given in the project instruction.

(2) Parse the input tokens and build an AST with the tokens.

(3) Validate the AST nodes and modify them with corresponding grammar of Dataframe API.

(4) Traverse the transformed AST to produce target source code.

9.1 Introduction of ANTLR

ANTLR(ANother Tool for Language Recognition) is an excellent tool to generate parsers and lexers, taking a grammar as input. Parsers can generate abstract syntax trees automatically with the given grammar, and ANTLR also provides tree walkers to walk through the generated abstract syntax tree and listeners to trigger any functions you want during the tree traversal process. The input lexer and parser of our transformation tool is as follows.

9.2 ANTLR Lexer

Lexer is a tokenization tool taking a character stream as input and outputs tokens matched by lexer rules. The lexer grammar of this project are as follows.

```
SC : 'sc';
VAL : 'val';
IF : 'if';
ELSE : 'else';
ASSIGN : '=';
DOT : '.';
LB : '{';
RB : '}';
LP : '(';
RP : ')';
COMMA : ',';
SEMICOLON : ';';
ARROW : '=>';
OP : '+' | '-' | '*' | '%';
COMP : '==' | '<' | '>' | '!=' | '>=' | '<=';
DIGIT : ('0'..'9');
LETTER : ('a'..'z') | ('A'..'Z') | '_';
ID : LETTER (LETTER | DIGIT)*;
NUMBER : DIGIT+;
```

9.3 ANTLR Parser

Parser takes the tokens generated from lexer as input, and generate the abstract syntax tree with given parsing grammar. The parsing grammar of this project is as follows.

```
start :
SC DOT 'range' LP NUMBER COMMA NUMBER RP
mapops DOT 'collect' LP RP;

mapops :
mapops DOT 'map' LP udf RP
|;

udf :
ID ARROW expression;

expression :
simpleexpression
| LB complexexpression RB;

simpleexpression :
pureexpression
| LP tupleexpression RP;

tupleexpression
: pureexpression COMMA pureexpression
| tupleexpression COMMA pureexpression;

complexexpression :
simpleexpression
| assignmentexpressions
–SEMICOLON simpleexpression;

assignmentexpressions :
assignmentexpression
|assignmentexpressions
–SEMICOLON assignmentexpression;

assignmentexpression :
VAL ID ASSIGN pureexpression;

pureexpression : ID
| ID DOT ID
| NUMBER
| LP pureexpression RP
| pureexpression OP pureexpression
| IF LP comparisonexpression RP pureexpression
–ELSE pureexpression;

comparisonexpression :
pureexpression COMP pureexpression
;
```

9.4 Translation Rules

After generating the abstract syntax tree, the next step is to convert the AST to Dataframe API, the translation rules of this project are as follows.

RDD API	DataFrame API
sc	spark
range(m,n)	range(m,n).selectExpr("id as _1")
map(***UDF***)	selectExpr(***SQL***)
collect	collect

The most challenging part of translation is that we need to create rules for translating from UDF to SQL, in order to achieve that we converted all variables in UDF to columns in and stored the mapping information in a *HashMap*, then did the translation recursively.

9.5 Test Cases

Three sample test cases are as follows:

```
Sample Input 1(RDD):
sc.range(10,100)
  .map(i=>{val j=i%3;(i, if(j==0)i*10 else i*2)})
  .map(r=>r._1+r._2)
  .collect()
Sample Output 1(Dataframe):
spark.range(10, 100).selectExpr("id as _1")
     .selectExpr("_1 as _1",
     "if(_1%3 == 0,_1 * 10,_1 * 2) as _2")
     .selectExpr("_1 + _2 as _1")
     .collect()

Sample Input 2(RDD):
sc.range(10,100)
  .map(i=>(i,i+i,i*10))
  .map(r=>r._1+r._2+r._3)
  .collect()
Sample Output 2(Dataframe):
spark.range(10, 100).selectExpr("id as _1")
     .selectExpr("_1 as _1","_1 + _1 as _2",
     "_1 * 10 as _3")
     .selectExpr("_1 + _2 + _3 as _1")
     .collect()

Sample Input 3(RDD):
sc.range(10,100)
  .map(i=>i+3)
  .collect()
Sample Output 3(Dataframe):
spark.range(10, 100).selectExpr("id as _1")
     .selectExpr("_1 + 3 as _1")
     .collect()
```

10 Creating Scala code for Spark SQL APIs

As part of extending our work past transforming simple commands within three specific APIs, we have also developed alternatives to some Spark SQL APIs. These specific and useful commands are a sort of black box, and could be slowing down the system in unknown ways. Further, translating them to a different language allows us to describe more limitations to our work, which can be overcome with future work.

We have chosen ten different Spark SQL APIs. They are relatively simple, but require interesting implementations. Many are similar, and some even use the functionality of others. They will be described below.

day(<string>) and lastday(<string>)

These two functions take an input string of the format "YEAR-MO-DY". It is a very rigid string structure, and as such, it is abusable. day(<string>) is the simpler of the two functions above; it simply returns the day of the date passed to it. We achieved the same functionality by returning the sub-string of the final two characters, which represent the day. lastday(<string>) is trickier. It outputs a string in the same format as above, but with the last day of the month it is passed. For example, giving it "1996-02-03" returns "1996-02-28". By checking the month, we can output the sub-string containing the year and month of the input with the last day of the appropriate month appended.

concat(<string[]>) and repeat(<string>, <int>)

These two functions are interesting; repeat uses the same functionality as concat to carry out its intended purpose. Concat is a very straightforward function; it concatenates a set of strings together. Repeat, however, outputs a string that is the input string repeated n times. Thus, it can easily be carried out by concatenating the input string to a string that is initialized as empty n times.

concat_ws(<string>,<string[]>)

This function has two input parameters, the first parameter is a separator and the second is an array of string, this function concatenates the strings with the separator. The ***mkString*** method provided by Scala, similar to ***String.join()*** method in java, makes this function easy to implement.

reverse(<string>)

This function is a case unlike the four above it; it outputs the input string, but reversed. It achieves this by iterating backwards through the input string, appending individual

characters to a new input string.

lpad(<string> str,<integer> len,<string> pad) and rpad(<string> str,<integer> len,<string> pad)

These two functions take two strings and an integer as input, and left-pad or right-pad the **str** with **pad**, finally return the padded string shortened to **len** characters. In our implementation we first check if the length of **str** is smaller than **len**, we just return the truncated substring in this case, otherwise we do the left or right concatenation with **pad** until the length of **str** is **len**.

ltrim(<string> str) and rtrim(<string> str)

These two functions take a string as input and return it after removing all the leading or tailing whitespace. To implement this function we used a pointer scanning through the string to find the first non-whitespace character, and return the substring before or after this pointer.

An interesting problem that arose when creating these scala implementations is how specific each one's implementation is. While some are similar or totally subsumed by others, they are still different enough to warrant individual attention. A transformer for these functions would need to be very detailed and case oriented; not far from what we implemented in the our own transformers. However, the complexity of these functions separates them from our implemented versions. In interest of good programming practices and efficacy, an object oriented transformer would be ideal. Things like day and lastday could easily have the same parent class, an interface for functions that require that date input, and each be children with their own unique implementations. Similarly, While concat and repeat have different arguments, the similarity of their implementation could lead to some interesting interactions in the compiler, where parent classes could reference others. This could lead to horizontal interaction, rather than exclusively vertical. Lastly, a large problem that we glanced over is input filtering. Functions to do with date will only work as intended with the intended input; a logical consequence of our implementation of day will have ANY input that isn't proper give that input with 30 concatenated to the end of it. If implemented directly into an actual compiler, this could lead to very unintended consequences and errors.

11 Limitations

Our work is composed of two proof of concepts and a purely speculative exercise. It is not robust, nor is it practical in its current state. However, this does not indicate problems with the concept addressed in the introduction, but rather imposes a set of limitations that can be resolved with more time and attention.

The most prominent limitation is how rigid and niche our transformers are, due to the small scope of functions being transformed. While our transformers effectively convert a small set of functions, it would be near useless in practice. The actual amount of functions that can be used in RDD dwarfs our set, meaning that the vast majority would remained unchanged after execution.

Secondly, our transformers have been built to transform correct sample inputs into correct outputs. As such, it does not filter input, or handle various types of errors. Our context-free grammars do successfully catch syntax errors, but that is also limited. An important step in pushing this work further is to completely filter input and cover all possible valid inputs,

Another problem is that our output is very rigid. There may exist a case in which require something different than what we create, or could have a far more efficient implementation.

Finally, the example scala programs are virtually useless in their current state. We have indeed made them correctly, and made sure they function equivalently, but there is very little advantage to using them in the way they are within the context of our problem. A transformer must be created to implement what we suggested, and it must handle far more cases than the ten we have worked with. Even creating this transformer has its own problems, as detailed in section 9.

12 Conclusions

In this project we developed two Spark API converters, taking programs written in RDD API and converting them to Dataset API and Dataframe API with the same functionality. By using our converters, people can write code in RDD API, as it's the most expressive one among three APIs, and take advantage of performance of Dataset API or Dataframe API.

Although our converters are fully tested, there are limitations that they are subject to certain transformation rules. For example, our RDD to Dataframe converter can only handle input with **sc**, **range**, **map** and **collect**, and the user defined functions are restricted to only identifier and expressions.

Further, our implemented scala versions of Spark SQL functions need to have a compiler actually generated for them; this is a challenging problem, as we have not generated a proof-of-concept for such a machine. However, we are confident it exists within the realm of possibility.

More generally, each point addressed in Limitations can be considered an open problem; they all have significant drawbacks, which can be solved with more time and effort.

During this semester-long project we learned how to write code using these three APIs, and moreover, we got a deeper understanding of compiler techniques and gained

new knowledge on how to engineer a compiler by developing these two converters. Further, the conceptual exercise of creating equivalent functions in a different language made us aware of some of the larger problems compilers face; and how solutions can be made to overcome them.