Paper SAS2249-2018

Shredding Your Data with the New DS2 RegEx Packages

Will Eason, SAS Institute Inc., Cary, NC

ABSTRACT

DS2's latest packages, PCRXFIND and PCRXREPLACE, wrap the functionality of previous PRX regular expression functions into sleek new packages. These just-in-time compiled regular expressions (or RegEx) can be used in multi-threaded environments to maximize throughput, while the object-oriented APIs simplify your connection to the powerful world of RegEx. Practical examples showcase using the new packages to execute RegEx on large data sets and include tips and techniques to get the most out of RegEx in distributed environments. We'll look at analyzing and filtering data sets using the new packages as well as using the fundamentals of text analytics to make processing large jobs faster. To top things off, we'll do the work with whichever smiling emoji best suits you, since the new packages are ready to handle your social media and international text processing needs.

INTRODUCTION

Regular expressions, or RegEx, are an essential tool for handling textual data. RegEx allow for users to create patterns representing specific strings of text allowing the user to quickly filter data or to derive useful information about text. RegEx play a critical role in recognizing pattern-identified entities within any text analytics pipeline. With the introduction of two new DS2 packages, PCRXFIND and PCRXREPLACE, RegEx are now accessible through a modern DS2 package interface.

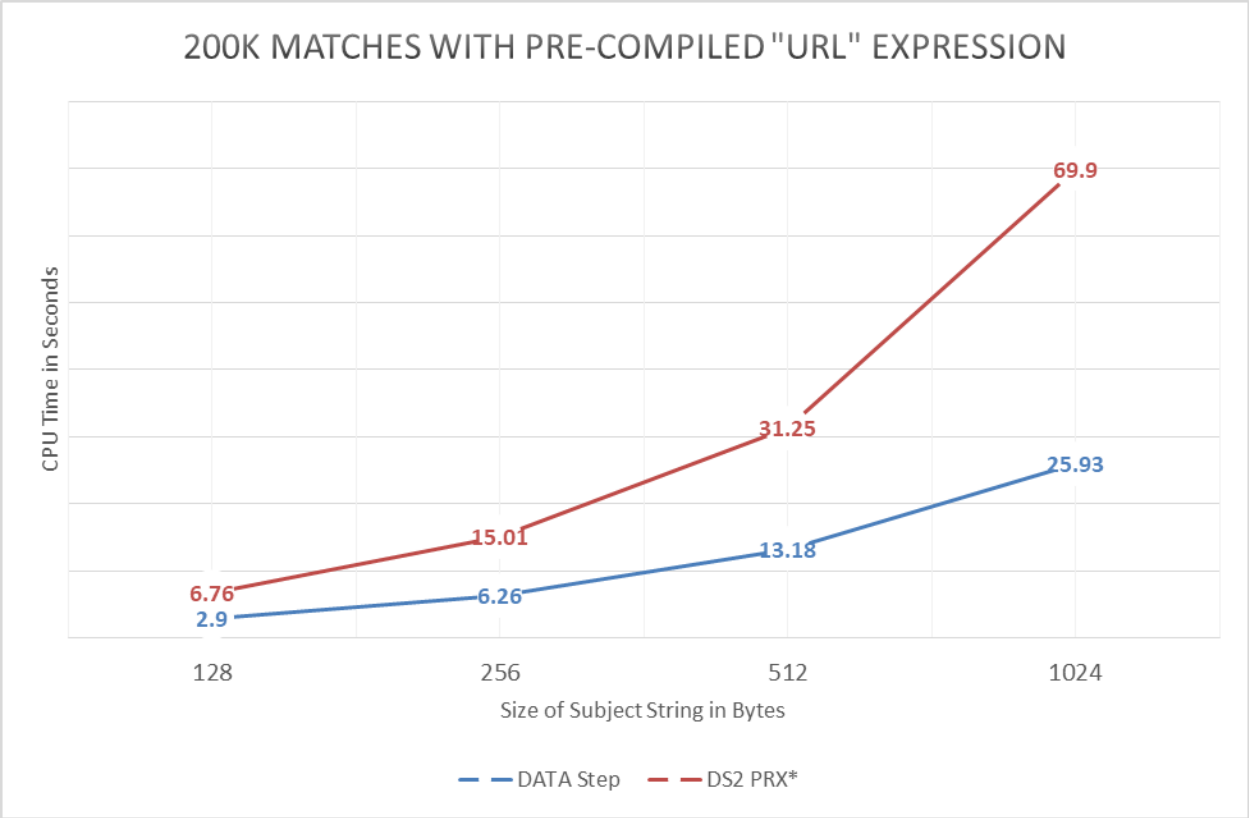
This paper first reviews the motivations for the new RegEx packages, then provides an overview of the new interface, and finally illustrates how to maximize the performance and utility of the new RegEx packages in your own DS2 programs.

MOTIVATION

Several motivating factors played a role in both the design of a new interface to call regular expressions as well as the engine that would back the new expressions. Ultimately, a modernized package interface was crafted that, as of SAS® 9.4M5 and SAS Viya® 3.3, is backed by Perl Compatible Regular Expressions. The interface provides users a more accessible way to access RegEx within their DS2 code which in turn runs on an engine that supports the same Perl-based regular expressions that DATA step and DS2 did previously.

PERFORMANCE GAP

DATA step has been a performance benchmark for DS2 since the inception of DS2 in SAS. While DS2 and DATA step offer different capabilities and different focuses, it was only a matter of time before the performance of the regular expression implementation in DS2 and DATA step was compared. A basic test suite reveals that DS2 prior to the new RegEx packages, was lacking in performance. The performance of a simple URL search case is documented in Graph 1 below. The test involved running the URL regex against the same dataset with PRXMATCH in DATA step and PRXMATCH in DS2. The regular expression was compiled once in both cases, prior to any matching.



Graph 1. A Comparison of Performance between Equivalent PRX *Functions in DATA Step and DS2

Note that as the size of the subject string increases beyond 64 bytes, the DS2 implementation sees an incredible increase in execution time as illustrated by Graph 1. The regular expression used to generate the data was a simple approach to finding URLs as seen below:

```
expr = '/(\w+:\/\/)?(\w+)\.(com|org|net|gov|io)(\/\w+)*(\w+\.\w+)?/';
```

While the times heavily favor DATA step as string size increases, they are only partially indicative of the performance gap. The features offered must be considered, and DATA step offers no UTF-8 support. Using a classic RegEx character class shorthand, such as "`\w+`", shows DATA step failing to recognize characters that fall outside Latin1 as anything other than system characters. The discrepancy in speed as well as the modernized features of the DS2 engine were considered as the new RegEx implementation was created in the form of PCRXFIND and PCRXREPLACE.

MODERNIZING INTERFACE

The interface of the old PRX* functions, such as PRXPARSE and PRXMATCH, is strictly function-based and does not capitalize on the new object-oriented packages that DS2 offers. Functions such as PRXPARSE return an identifier or handle as a double that users then have to pass between functions. Leveraging DS2 packages would allow users to declare their desired expressions upfront. Users can create more accessible code by calling a method of a package named such that the intended result of the expression is clear. The following example illustrates the new clarity:

```
proc ds2;
  <...>
  dcl package pcrxFind phoneNumberFinder( '/\(([0-9]{3})\) \d{3}-\d{4}/' );
  <...>
```

In the example of both, a call to `phoneNumberFinder.match( … )` clearly searches for a phone number.

In addition to making the intent of a regular expression call clearer, placing the expressions with the context of a package allows for further streamlining of some basic RegEx function calls, such as getting the text from the previous match. Finally, packages promote the early declaration of the desired regular expressions so that each package represents a single RegEx compilation that can occur in the INIT method of a DS2 program. This keeps costly RegEx compiles out of the RUN method.

BACKWARD COMPATIBILITY

A final motivating factor driving the API design process and the engine selection was backward compatibility. The PCRX package methods are backed by a "Perl-compatible" regular expression engine that matches the functionality of the previous PRX functions. In addition, the interface was designed in a way to make transitioning to the new packages straightforward.

INTRODUCING PCRXFIND AND PCRXREPLACE

PCRXFIND and PCRXREPLACE are Perl-compatible regular expression packages that bundle the functionality to find and analyze a match and its corresponding group information (PCRXFIND) as well as offer substitution capabilities (PCRXREPLACE). The interface leverages DS2 packages allowing for modern, readable code.

INTERFACE

PCRXFIND

METHODS	ARGUMENTS	RETURN	DESCRIPTION
NEW PCRXFIND	[CHARACTER expression]	VOID	Instantiates a new instance of the PCRXFIND package
PARSE	CHARACTER expression	INTEGER rc	Parses the given expression. Returns a return code indicating whether the parse was successful.
MATCH	CHARACTER subject, [INTEGER offset]	BIGINT offset	Performs a match using the previously compiled expression. The match metadata, such as group indexes and the length, is then saved within the PCRXFIND instance for subsequent calls to getGroup* and getMatch* methods.
GETGROUP	INTEGER groupID, IN_OUT CHARACTER group	INTEGER rc	Retrieves the text corresponding to the specified capture groupID and places the text into the in_out string provided. Based on the previous match.
GETGROUPSTART	INTEGER groupID	BIGINT offset	Returns the starting index of the specified capture groupID based on the previous match.
GETGROUPEND	INTEGER groupID	BIGINT offset	Returns the ending index of the specified capture groupID based on the previous match.
GETGROUPLENGTH	INTEGER groupID	BIGINT length	Returns the length of the specified capture groupID based on the previous match.

METHODS	ARGUMENTS	RETURN	DESCRIPTION
GETMATCH	IN_OUT CHARACTER match	INTEGER rc	Retrieves the text of the entire previous match and places the text into the provided IN_OUT string.
GETMATCHSTART	NONE	BIGINT offset	Returns the starting index of the previous match.
GETMATCHEND	NONE	BIGINT end	Returns the ending index of the previous match.
GETMATCHLENGTH	NONE	BIGINT length	Returns the length of the previous match.

Table 1. List of Available PCRXFIND Methods and Operators

PCRXREPLACE

METHODS	ARGUMENTS	RETURN	DESCRIPTION
NEW PCRXREPLACE	[CHARACTER replacement expression]	VOID	Instantiates a new instance of the PCRXFIND package. Optionally parse the expression here.
PARSE	CHARACTER expression	INTEGER rc	Parses the given expression. Returns a code indicating whether the parse was successful.
APPLY	CHARACTER subject, [INTEGER numtimes]	BIGINT offset	Applies the previously parsed substitution expression to the given subject string. An optional argument allows the user to specify a certain number of times the change will be applied.

Table 2. List of Available PCRXREPLACE Methods and Operators

The PCRX* packages are designed to have a familiar but more robust interface than the old PRX functions. Some notations from the PRX* functions have been kept to ease the transition to the new packages, while others have been streamlined to make regular expressions more accessible inside DS2 programs.

Perhaps the most noticeable interface feature shared between the two packages and the old PRX* functions is the string argument within the PARSE method.

```
proc ds2;
<..>
dcl package pcrxFind pcrx_taco( '/taco/i' );
dcl double prx_taco;
<..>
pcrx_taco.parse( '/ram/i' );
prx_taco = PRXPARSE( '/taco/i' );
<..>
```

Note that the new PCRX* packages maintain the use of slashes as delimiters within expressions. Users can enter their regular expression between two forward slashes, such as in '/expression/i', and can then add any necessary modifying flags trailing the second slash.

In PCRXFIND, the package expects the expression string to start and end with a slash followed by any matching flags. For PCRXREPLACE, the expression is composed of three parts: the RegEx itself, the substitution text, and the flags, all separated by forward slashes. An example is "/replacethis/withthis/g". Finally, the PCRX* PARSE methods each take flags trailing the final forward slash such that a "find"

expression will have two forward slashes and a “replace” expression will have three slashes. To create an expression searching for a forward slash, the slash must be escaped by a backslash such as in ‘/\//’, which searches for a single slash. Note that the previous string does not contain the letter V, but a sequence of slashes: forward slash, backward slash, forward slash, forward slash. Other special characters used for the definition of regular expressions can be similarly searched for by escaping those characters (such as parentheses, +, *, and so on) with a backslash.

Using packages allows DS2 to track the status of expressions within each instance of a regular expression package. Consider several RegEx packages defined to search for a variety of animals.

```
proc ds2;
<...>
dcl package pcrxFind ram( '/ram/i' );
dcl package pcrxFind wolf( '/wolf/i' );
dcl package pcrxFind wildcat( '/wildcat/i' );
<...>
ram.match( mySubject );
wolf.match( mySubject );
wildcat.match( mySubject );
<...>
```

Each of these expressions–being defined in their own package–is able to track their match metadata independently. Calling GETMATCHSTART on ram, wolf, and wildcat will reveal the index where each of the above matches began.

NOTES ON PCRXREPLACE

PCRXREPLACE is very similar to PCRXFIND but has some key differences. First, the “expression” argument differs from that of PCRXFIND since it includes not just an expression and flags, but a substitution component as well. For example:

```
proc ds2;
<..>
myPCRXFindInstance.parse( '/find this/' );
myPCRXReplaceInstance.parse( '/change this/to this/g' );
<..>
```

The syntax of PCRXREPLACE mimics that of the function PRXCHANGE. The text “change this” represents the expression to search for, which will then be replaced with “to this”. The flag “g” is exclusive to PCRXREPLACE and applies the expression globally, similar to passing -1 as the number of times to PRXCHANGE. PCRXREPLACE also supports substitution text with capture groups:

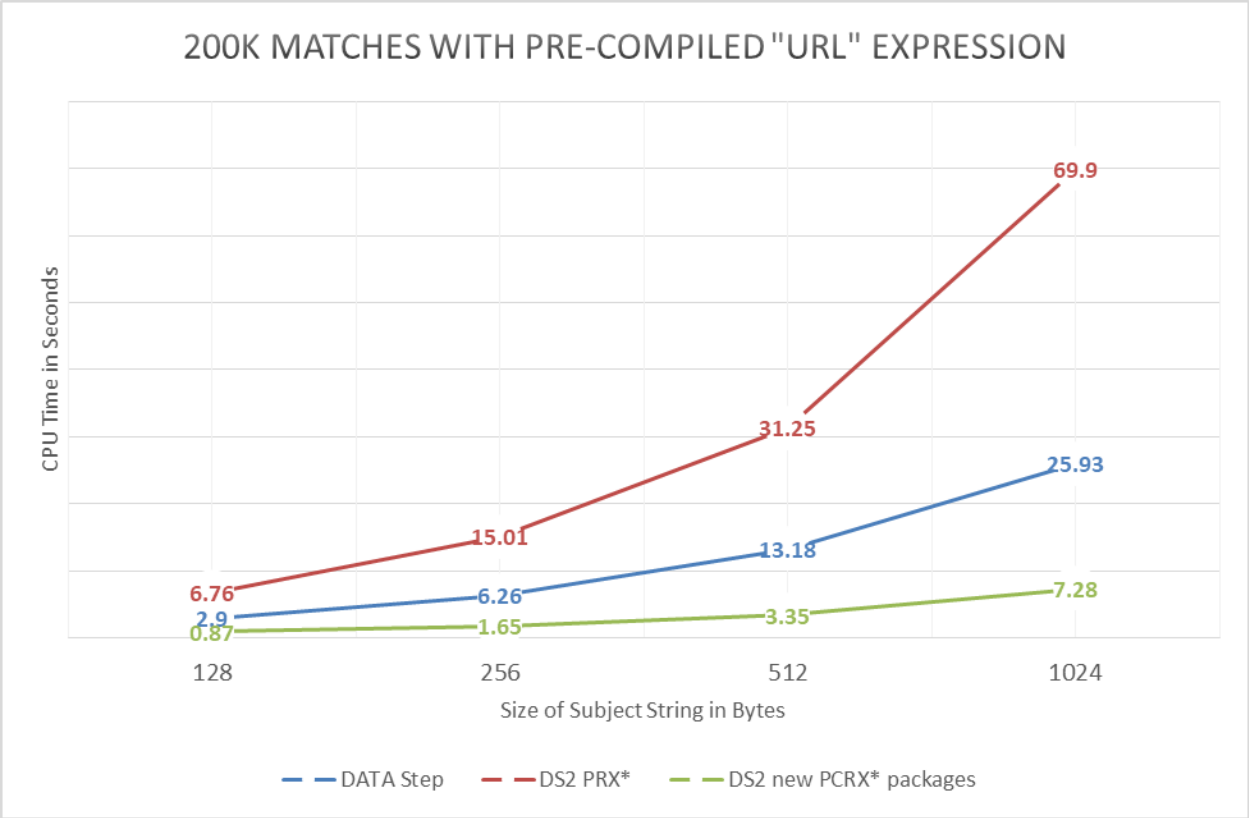
```
proc ds2;
<..>
myPCRXReplaceInstance.parse( '/(\w+) (\w+)/$2:$1/g' );
myPCRXReplaceInstance.apply( textToSwapOrder );
<..>
```

The example above parses an expression that uses dollar notation to reference the text of a capture group within the substitution text. $1 will represent the first (\w+) match while $2 represents the second. The two capture groups will be swapped and a colon will be placed between them.

Note: If you would like to substitute a dollar sign in ($), use $$ within the substitution expression.

UPGRADED PERFORMANCE NUMBERS

As seen in Graph 1, the previous performance of DS2’s regular expression functions fell behind DATA step significantly, especially as the size of the subject text increased. The results of the new packages speak for themselves, and can be seen in Graph 2 below.



Graph 2. Performance Comparison with New PCRX* Packages

EXAMPLES AND PERFORMANCE OPTIMIZATIONS

FINDING ALL MATCHES

Perhaps one of the most basic regular expression examples would be finding every single instance of a match in a body of text. This can be accomplished by combining a DO loop and an instance of the PCRXFIND package. Here is an example:

```
proc ds2;
data _null_;
method init();
   dcl package pcrxfind sentenceEnd( '/(\w+) (\w+) (\w+)\./x' );
   dcl varchar(1024) exampleText;
   dcl int offset; /* To keep track of where in the example text we are */
   offset = 1;
   exampleText = 'This is example text showcasing the pcrx packages.
                  The sentenceEnd regex finds the three words
                  before a period. One final sentence for demonstration.';

   do while ( (sentenceEnd.match( exampleText, offset )) > 0 );
      offset = sentenceEnd.getMatchEnd();
      put offset=;
   end;
end;
enddata;
<..>
offset=51 match=the pcrx packages.
offset=146 match=before a period.
```

```
offset=184 match=sentence for demonstration.
```

The example above finds every-non overlapping instance of the example expression. Modifying the next search index from starting at the match end to the character after the previous match start allows for finding matches that might be overlapping. The following code change demonstrates setting the offset one character after the match began:

```
offset = sentenceEnd.getMatchEnd();
/* becomes ... */
offset = sentenceEnd.getMatchStart() + 1;
```

In this particular case, running the same example with the modified offset calculation yields many more matches as seen below:

```
offset=34 match=the pcrx packages.
offset=35 match=he pcrx packages.
offset=36 match=e pcrx packages.
offset=131 match=before a period.
offset=132 match=efore a period.
offset=133 match=fore a period.
offset=134 match=ore a period.
offset=135 match=re a period.
offset=136 match=e a period.
... and so on.
```

ERROR HANDLING AND RELATED NOTES

The PCRX packages attempt to report errors in a clear manner to the user to make using and debugging with them easier. In the following example, an instance of PCRXFIND is declared with an invalid expression:

```
proc ds2;
data _null_;
method init();
   dcl package pcrxfind myfinder ();
   dcl int rc;
   rc = myfinder.parse( '/test' );
   put rc=;
end;
enddata;
run; quit;
...
rc=
ERROR: Error reported by DS2 package pcrxfind:
ERROR: Illegal expression: invalid number of forward slashes: /test
```

In this case, the error is expressed in two ways: 1) an explicit error message and 2) the return code returned by the call to parse is null. The user can check the return code to see if it's null to avoid crashes later. Calling the match or apply functions on the appropriate PCRX package without a successful prior parse will result in an error.

The errors that the PCRX functions express are generally at the standard error level and are written to the log with the exception of preventable errors. Calling MATCH or APPLY without a previously successful parse will result in a fatal error. These fatal errors can be avoided by checking the return codes of prior operations. Similar to MATCH and APPLY, which require a successful parse prior to their use,, methods dependent on the metadata of a match such as GETGROUP or GETMATCHSTART will result in an error if no previous match has been made.

Note: The only unpreventable fatal error occurs when the constructor of a PCRX package has an expression that is invalid. This results in a fatal error since there is no mechanism for users to handle a failed package constructor.

MAXIMIZING PERFORMANCE

The engine used by PCRX is Perl-compatible and uses a backtracking engine that allows for negative look behinds. This means that standard regular expression performance optimizations will generally apply.

At the time of this paper, there are two natively supported encodings for regular expressions: UTF-8 and Latin1. These encodings were chosen due to their wide international reach. In order to take advantage of Latin1 performance, the expression passed into the PCRX package with the PARSE method must be encoded in Latin1 and the data passed to the package must be encoded in Latin1. Within a UTF-8 session, a Latin1 expression can be created by storing the expression in a string with the appropriate character set as seen below:

```
dcl varchar(512) character set latin1 myExpression;
myExpression = '/(\w+) (\w+) (\w+)\./';
myPCRXFindPackageInst.parse( myExpression );
```

Due to the single-byte nature of Latin1, its performance will be substantial higher than UTF-8 encoded text. However, both Latin1 and UTF-8 encodings show significant improvement when compared against the previous DS2 implementation and DATA step as seen in Graph 2.

CONCLUSION

The new RegEx packages, PCRXFIND, and PCRXREPLACE, offer backward compatible functionality with a versatile and modernized interface backed by a high-performance engine. The new packages make regular expressions more accessible in DS2 and show significant improvements of performance relative to the old PRX functions. The new packages offer an interface that reflects an effort focused on making an intuitive and familiar interface for all of your RegEx needs.

RESOURCES

- SAS Institute Inc. 2017. *SAS 9.4 DS2 Language Reference*. Cary, NC: SAS Institute Inc. Available http://support.sas.com/documentation/onlinedoc/viya/index.html.

- "PCRE - Perl Compatible Regular Expressions." Available http://pcre.org/.

ACKNOWLEDGMENTS

The author would like to thank Robert Ray for his guidance during the implementation of the new PCRX packages. In addition, the author is thankful to Michael Bowlus, Bryan Ewbank, Al Kulik, Kat Schikore, and the BIT team at SAS for their feedback during development.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Will Eason
100 SAS Campus Drive
Cary, NC 27513
SAS Institute, Inc.
Will.Eason@sas.com
http://www.sas.com