Paper DM-224

Your Database Can Do Complex String Manipulation Too!

Harry Droogendyk, Stratia Consulting Inc., Lynden, ON

ABSTRACT

Since databases often lacked the extensive string handling capabilities available in SAS®, SAS users were often forced to extract complex character data from the database into SAS for string manipulation. As database vendors make regular expression functionality more widely available for use in SQL, the need to move data into SAS for pattern matching, string replacement and character extraction is no longer (as) necessary.

This paper will cover enough regular expression patterns to make you dangerous, demonstrate the various REGEXP* SQL functions and provide practical applications for each.

INTRODUCTION

SAS has a wealth of character functions and call routines available in version 9.4 – 96 in all. Most databases have far fewer character functions *available in SQL*, e.g. Teradata v15 has 30 character functions, Oracle 12c has 40 character functions that can be used within an SQL query. SAS functions are powerful and flexible, especially when used in conjunction with looping and conditional features of the SAS language. As a consequence, SAS programmers often move data unnecessarily from the database into SAS simply to perform complex string manipulation. Or, it may be that *implicit* pass-through queries (those using the SAS/Access libname to read database tables) utilize SAS functions that the SAS Access engine cannot convert to database-specific syntax and pass to the database for processing. As a result, the data is brought into SAS and processed there.

While creative use of SQL string handling functions can help us process data where it makes sense, the family of regular expression functions now available in many databases affords better opportunities to process data within the database using SQL – without moving it out of the database! Though the majority of the examples in the paper were developed in Teradata v15, the syntax should be largely portable to the SQL dialects used in other databases.

Note that this is NOT a "regular expression paper" per se. While the examples offered do contain regular expression syntax and do work as advertised, the yawning gaps in my regex knowledge are vast and innumerable. My typical modus operandi is the timeless "poke and hope" methodology fueled by nuggets gleaned from sites that offer regular expression help (see links at the bottom of this paper). It's certainly possible that the regular expression syntax offered in this paper could have been written more concisely or efficiently by one more versed in the cryptic language of regex.

The intent of this paper is to illustrate a number of examples where SQL regular expression functions have allowed complex string manipulation operations to be performed in the database without pulling data into SAS. Since the focus is on SQL, any database procedural languages and the functions and capabilities available within those languages are not considered.

Most of the remainder of the paper will be a series of tables, syntax definitions and explanations and (hopefully) useful examples with a minimum of arid verbiage.

COMMONALITY BETWEEN SAS AND SQL FUNCTIONS

Many of the familiar SAS character functions *are* available in SQL. While some have different names than their SAS counterparts, it's certainly possible to do most *basic* string manipulation in SQL using these functions. And, SQL has additional functions or options that SAS does not have. Many string manipulation needs can be satisfied with built-in SQL functions without using more complex and less maintainable regular expression functions.

When possible keep it simple!

String Processing	SAS Function	Teradata Function
trim trailing spaces	TRIM(source)	TRIM(TRAILING FROM source)
trim leading & trailing spaces	STRIP(source)	TRIM(source)
character length	LENGTH(source)	LENGTH(source)
find character's 1st position in a string	INDEX (source, excerpt)	INDEX (source, excerpt)
find character's nth position in a string		INSTR(source, excerpt, start pos, occurrence)
extract portion of a string	SUBSTR(source, start, length)	SUBSTR(source, start, length)
extract portion of a string by delimiter	SCAN(source, occur, delimiter)	STRTOK(source, delimiter, occur)
split delimited values into rows		STRTOK_SPLIT_TO_TABLE(key, source, delimiter)
partial pattern matching	(NOT) LIKE 'abc%'	(NOT) LIKE 'abc%'
character replacement	TRANSLATE(source, replacestr, string)	oTRANSLATE(source, string, replacestr)
string replacement	TRANWRD(source,replacestr,string)	oREPLACE(source, string, replacestr)

Table 1. SAS / Teradata function equivalents

ENHANCED FUNCTIONALITY REQUIRED

There are times when the basic SQL string handling functions will not suffice. Some examples of these cases are:

- checking for numeric value in character string (Netezza has isNumeric available in SQL Extensions Toolkit, utterly inexplicably, most databases do not have this functionality, a null result from Teradata's TO_NUMBER function is reasonable proxy for the numeric check)
- alpha numeric pattern matching
- complex pattern matching, e.g. phone numbers with or without separators
- identifying the presence of a pattern anywhere in a string
- character replacement only where certain characters are (not) adjacent to the specified pattern
- extracting an unknown number of delimited values

It's in cases like these where the power and utility of regular expression functions allows the user to manipulate the data within the database and avoid unnecessarily data movement through the network to SAS.

REGULAR EXPRESSION BASICS

Despite the "poke and hope" disclaimer in the introduction, this paper must cover some regular expression basics to allow potential applications of the regular expression functionality to become more evident to the reader.

In the words of www.regexbuddy.com, a regular expression "is a special text string for describing a search pattern. You can think of regular expressions as wildcards on steroids". The special text string includes tokens to describe character types and patterns, anchors that specify string location, and qualifiers that specify in what quantity the character types and patterns occur. While there are several different regex dialects in use, the vast majority of the syntax is common between dialects. However, because the possibility of differences does exist, it's important to ascertain which dialect is used by the database to deliver successful results.

COMMON TOKENS

Token	Meaning
.	any character
\s	any whitespace (space, tab, line break, form feed)
\d, [:digit:]	any digit
[abc]	single character a, b or c
[^abc]	character other than a, b or c
[a-zA-Z]	characters in the range: a-z and A-Z
\w	any word (set of characters including [a-zA-Z0-9_]
(dog|cat)	“dog” or “cat”
(…)	capture everything enclosed in parenthesis
\	escape character used when searching for regex tokens, e.g. . ? + * $ ^

Table 2. Common regex tokens

COMMON ANCHORS

Anchor	Meaning
^	start of string
$	end of string
\b	word boundary (not supported by Oracle, consider using (\W|^|$) instead for)

Table 3. Common regex anchors

COMMON QUALIFIERS

Qualifier	Meaning
?	zero or one
*	zero or more
+	one or more
{3}	exactly 3 of whatever precedes, e.g. \d{3}
{3,6}	between 3-6 of whatever precedes, e.g. [a-z]{3,6}

Table 4. Common regex qualifiers

The following example illustrates at least one item from each of the three preceding regex categories. In this case, we're searching for a userid of a specific format found at the end of a line in a server log:

XY|AB[\d]{4}[a-z]+\.$

- XY|AB	literal XY or AB
- [\d]{4}	exactly four digits
- [a-z]+	one or more lower case alphabetic characters
- \.	literal period, escaped because period is also a token meaning any character
- $	at the end of the string

`Login at 2016-09-18 09:23:17 userid XY1234b.`	- match
`AB1234x. is the user’s ID`	- no match (userid value not found at end of line)

GENERAL NOTES:

1. ANSI SQL standards are a bit of a myth (there's no tooth fairy either, sorry). Each database vendor seems to tweak their implementation of each of these functions. ***Please consult your vendor's documentation*** for specific functionality and default function parameter information.
2. Different database vendors support different regex dialects. e.g. Oracle supports POSIX, Teradata typically follows the Perl variant.
3. When creating columns using REGEXP_SUBSTR and REGEXP_REPLACE, always CAST to the specific type and length required. Database defaults may be much different than desired, e.g. Teradata VARCHAR(8000).
4. Unless otherwise stated, the REGEXP function syntax and regular expression dialect have been tested on Teradata. YMMV.

REGULAR EXPRESSION FUNCTIONS

Function	Use
REGEXP_INSTR	find starting or ending position of a string pattern in the source
REGEXP_SIMILAR / LIKE	Boolean result - is string pattern in source?
REGEXP_SUBSTR	extract a portion of source that matches string pattern
REGEXP_REPLACE	replace a portion of source that matches string pattern
REGEXP_SPLIT_TO_TABLE	split delimited string into rows, delimiter defined by regex pattern

Table 5. REGEXP SQL functions

1. REGEXP_INSTR

REGEXP_INSTR is used to find the starting or ending position of the string satisfied by the search pattern. It extends the functionality of the "vanilla" INSTR function by allowing searches for a regex pattern rather than just basic text.

Parameter	Use / Values
source	column or literal to be searched
regex	regex pattern
start position	position in source to begin searching, relative to 1, default is 1
occurrence	occurrence of string matching regex pattern, default is 1
return	0=starting position of matched string, 1=position following end of matched string, default is 0
match	i=ignore case, c=case sensitive (additional, less commonly used values allowed as well)

Table 5. REGEXP_INSTR parameters

REGEXP_INSTR EXAMPLES

Typically the REGEXP_INSTR is used for the same purpose as the INSTR function – to locate the starting and/or ending positions to be used in a SUBSTR function. e.g.

```
SELECT SUBSTR ( 'string', REGEXP_INSTR('string', 'regex'), 5 ) as zip_code;
```

Example 1.1:

Find the position of the zip code in the following address label information:

```
   SELECT REGEXP_INSTR (
         'Joe Smith, 10045 Berry Lane, San Joseph, CA 91234',
         '[[:digit:]]{5}$')     -- 45
```

`[[:digit:]]`	digits, 0-9 characters
`{5}`	exactly five of the preceding characters, i.e. 5 digits
`$`	anchor at the end of the string, grab the last five digits

The regular expression is looking for five digits anchored to the end of the string. If the $ anchor were omitted, the street number would be returned instead of the zip code. Result of query is 45 - the starting position of zip code.

Example 1.2:

Find the position of the delivery date, the 2nd occurrence of eight digits preceded by a word boundary. The Part number beginning with DT would not be a match since those eight digits are not preceded by a word boundary.

```
SELECT REGEXP_INSTR (
    'Part DT12345678, ordered 20160928, delivered 20161001',
    '\b\d{8}',1,2,0)                                    -- 46
```

`\b`	word boundary (i.e. so part number with DT prefix isn't matched)
`\d{8}`	eight digits
`1,2,0`	1=starting position, 2=occurrence, 0=return starting position of match

The result is 46, the starting position of the delivery date. If delivery date (i.e. second set of eight digits preceded by a word boundary) was not found in the string, the returned result would have been 0.

Example 1.3:

Specifying a RETURN parameter of 1 will return the position *following* the matched string, *NOT the last position of the matched string*. Find the position following three non-alphabetic characters.

```
SELECT REGEXP_INSTR('123ABC','[^a-z]{3}',1,1,1,'i')     -- 4
```

`[^a-z]{3}`	exactly three non-alphabetic characters
`1,1,1`	1=starting position, 1=occurrence, 1=return position following end of match
`'i'`	case insensitive match, effectively regex pattern turns into [^a-zA-Z]

The result is 4, the position *following* the match.

2. REGEXP_SIMILAR

REGEXP_SIMILAR returns a Boolean result (1=true, 0=false) to indicate if data matching the regex pattern is found in the source string. Since **Teradata requires that the *entire* string match the search pattern**, the regular expression pattern specified must account for other characters within the source string as well.

Parameter	Use / Values
source	column or literal to be searched
regex	regex pattern
match	i=ignore case, c=case sensitive (additional, less commonly used values allowed as well)

Table 6. REGEXP_SIMILAR parameters

REGEXP_SIMILAR EXAMPLES

The REGEXP_SIMILAR function is often used in CASE statements or WHERE clauses.

```
SELECT * FROM schema.table
 where REGEXP_SIMILAR(char_column, 'regex', 'i' ) = 1
```

Example 2.1:

Select only rows where the text field contains programmers' userids, i.e. userids starting with "pg", ending in five digits. The examples below provide some insight into the vagaries of the function.

```
SELECT REGEXP_SIMILAR ('Userid is pg26581', '\bpg\d{5}\b')     ;     -- 0
```

`\b`	word boundary at beginning of userid pattern
`pg`	literal "pg"
`\d{5}`	five digits
`\b`	word boundary at end of userid pattern

Since REGEXP_SIMILAR matches the *entire* string, the presence of "Userid is" results in a nomatch.

```
SELECT REGEXP_SIMILAR ('Userid is apg26581a','.*\bpg\d{5}\b.*')  ;     -- 0
```

`.*`	zero or more of any character
`\b`	word boundary at beginning of string
pg	literal "pg"
`\d{5}`	five digits
`\b`	word boundary at end of string
`.*`	zero or more of any character

The .* tokens have allowed for characters before and/or after the userid value, but the \b token demands that a word boundary precede and follow the userid value. The presence of "a" before and after "pg26581" results in a nomatch.

```
SELECT REGEXP_SIMILAR ('Userid is pg26581', '.*\bpg\d{5}\b.*')  ;     -- 1
SELECT REGEXP_SIMILAR ('pg26581 - userid',  '.*\bpg\d{5}\b.*')  ;     -- 1
```

`.*`	zero or more of any character
`\b`	word boundary at beginning of string
pg	literal "pg"
`\d{5}`	five digits
`\b`	word boundary at end of string
`.*`	zero or more of any character

The valid userid value is surrounded by word boundaries – match.

```
SELECT REGEXP_SIMILAR ('Userid is apg26581a','.*pg\d{5}.*')       ;     -- 1
```

`.*`	zero or more of any character
pg	literal "pg"
`\d{5}`	five digits
`.*`	zero or more of any character

The last example illustrates what's really a false match. Since the word boundary requirements have been omitted from the regular expression, the embedded userid value is deemed a match when it probably should not be.

Example 2.2:

The following example is for an **Oracle** database. If data matching the regex pattern is found within the source string, the result is true – regardless what precedes or follows the matched string, i.e. no need to include the .* required by Teradata. Since Oracle does not support \b for a word boundary, an alterative pattern was specified.

```
SELECT REGEXP_LIKE ('Userid is pg26581', ' (\W|^|$)pg\d{5} (\W|^|$)');    -- 1
```

`(\W|^|$)`	beginning of string, end of string, or not a word character (not alphanumeric)
`pg`	literal “pg”
`\d{5}`	five digits
`(\W|^|$)`	not a word character (not alphanumeric)

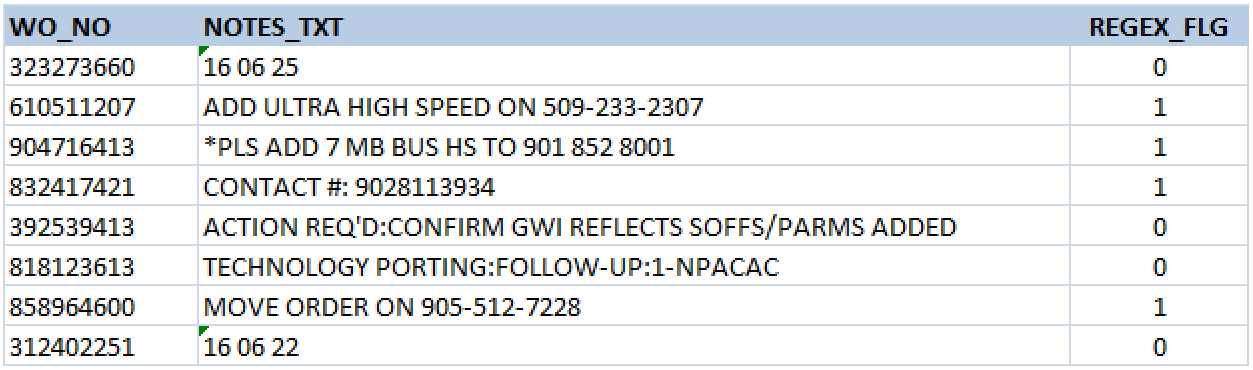
Example 2.3:

Select only rows where the NOTE_TXT field contains a phone number (Teradata).

```
SELECT wo_no, notes_txt, REGEXP_SIMILAR ( notes_txt,
                         '.*\d{3}[\s.-]?\d{3}[\s.-]?\d{4}.*') AS regex_flg
```

`.*`	zero or more of any character
`\d{3}`	exactly three digits
`[\s.-]?`	zero or one whitespace or period or dash characters
`\s`	whitespace
`.-`	period or dash
`?`	zero or one occurrences of the characters defined by the pattern between [ ]
`\d{3}`	exactly three digits
`[\s.-]?`	zero or one whitespace or period or dash characters
`\d{4}`	exactly four digits
`.*`	zero or more of any character

As the REGEX_FLG value in the output below indicates, phone numbers with various separators and those without, matched the regular expression pattern.



WO_NO	NOTES_TXT	REGEX_FLG
323273660	16 06 25	0
610511207	ADD ULTRA HIGH SPEED ON 509-233-2307	1
904716413	*PLS ADD 7 MB BUS HS TO 901 852 8001	1
832417421	CONTACT #: 9028113934	1
392539413	ACTION REQ'D:CONFIRM GWI REFLECTS SOFFS/PARMS ADDED	0
818123613	TECHNOLOGY PORTING:FOLLOW-UP:1-NPACAC	0
858964600	MOVE ORDER ON 905-512-7228	1
312402251	16 06 22	0

Table 7. REGEXP_SIMILAR phone number example

3. REGEXP_SUBSTR

REGEXP_SUBSTR is similar to REGEXP_INSTR, but instead of returning the position of the matched string, it returns the matched string itself.

Parameter	Use / Values
source	column or literal to be searched
regex	regex pattern
start position	position in source to begin searching, relative to 1
occurrence	occurrence of string matching regex pattern
match	i=ignore case, c=case sensitive (additional, less commonly used values allowed as well)

Table 8. REGEXP_SUBSTR parameters

REGEXP_SUBSTR EXAMPLES

Rather than simply testing the presence of data matching a regex pattern, or identifying the starting position of matching data, REGEXP_SUBSTR can be used to extract the matched data from the text.

```
SELECT REGEXP_SUBSTR('string', 'regex') AS text;
```

Example 3.1:

Building on the phone number example in the REGEXP_SIMILAR section, REGEXP_SUBSTR can be used to extract phone numbers from the text. Note the difference between the regular expression pattern used in the REGEXP_SIMILAR function and that used for REGEXP_SUBSTR when executing in Teradata.

```
SELECT wo_no, notes_txt
 , REGEXP_SUBSTR ( notes_txt, '\d{3}[\s.-]?\d{3}[\s.-]?\d{4}',1,1) AS phone1
 , REGEXP_SUBSTR ( notes_txt, '\d{3}[\s.-]?\d{3}[\s.-]?\d{4}',1,2) AS phone2
  FROM schema.table
 WHERE REGEXP_SIMILAR ( notes_txt, '.*\d{3}[\s.-]?\d{3}[\s.-]?\d{4}.*')
```

Where the Teradata REGEXP_SIMILAR requires the .* tokens to account for text before or after the phone number, including those tokens in the REGEXP_SUBSTR function would result in the entire value of NOTES_TXT being returned and not just the desired phone number.

o `\d{3}`	exactly three digits
o `[\s.-]?`	zero or one whitespace or period or dash characters
`\s`	whitespace
`.-`	period or dash
`?`	zero or one occurrences of the characters defined by the pattern between [ ]
o `\d{3}`	exactly three digits
o `[\s.-]?`	zero or one whitespace or period or dash characters
o `\d{4}`	exactly four digits

Additional parameters in the REGEXP_SUBSTR function are:

- `1,1` search starting in position 1, grab 1st occurrence
- `1,2` search starting in position 1, grab 2nd occurrence

Example 3.2:

Regular expressions also allow us to identify data matching a pattern that is before, after or in proximity to another pattern or value. In the example below, we want to identify a "host number" if it is available. If the data field, ATTR_KEY, contains the literal value "Host" followed by a period, the digits immediately following "Host." are the host number. Because ATTR_KEY can contain any number of period-delimited components, we cannot simply hard-code the REGEXP_SUBSTR occurrence option and extract the 4th period-delimited item and assume it will be the host number.

```
attr_key = 'Device.Hosts.Host.10.X_OUI_History.Layer1Interface'

SELECT REGEXP_SUBSTR (attr_key, '(?<=Host\.)[0-9]+') as host_no     -- 10
```

o `(?<=Host\.)`	positive lookbehind – is "Host." in the string ?
o `[0-9]+`	one or more digits

http://www.regular-expressions.info/lookaround.html states:

"Lookahead and lookbehind, collectively called "lookaround", are zero-length assertions just like the start and end of line, and start and end of word anchors explained earlier in this tutorial. The difference is that lookaround actually matches characters, *but then gives up the match,* returning only the result: match or no match. That is why they are called "assertions". They do not consume characters in the string, but only assert whether a match is possible or not". (*emphasis* mine)

While the many intricacies of "lookaround"s is beyond the scope of this paper, a short explanation is required. ☺ In processing this mess, the regular expression first matches one or more digits [0-9]+, finding "10" in position 19 of the string. But, before "10" is accepted, it first verifies that the text in the lookbehind regex can *also* be matched, i.e. does the string which precedes the digits match the lookbehind regex. In this case "Host." does precede the 10 and the REGEXP_SUBSTR returns 10.

Example 3.3:

From the same data used in the second example, extract the last period-delimited value. Since the number of period-delimited values varies among ATTR_KEY values we cannot simply extract the 6th occurrence using the REGEXP_SUBSTR option.

```
attr_key = 'Device.Hosts.Host.10.X_OUI_History.Layer1Interface'

SELECT REGEXP_SUBSTR(attr_key, '[^.]*$') as attr_key_last   -- Layer1Interface
```

- `[^.]` match any character other than a period
- `*` zero or more times
- `$` assert position at the end of the string

The regex is starting at the end of the string, and gobbling zero or more characters until it hits a period. Note that the period does not have to be escaped when it's within [ ].

4. REGEXP_REPLACE

REGEXP_REPLACE is similar to REGEXP_SUBSTR but instead of extracting data matching the regular expression, the function replaces the matched text with other values (or nulls).

Parameter	Use / Values
source	column or literal to be searched
regex	regex pattern
replacement	replacement string *or pattern*
start position	position in source to begin searching, relative to 1
occurrence	occurrence of string matching regex pattern, **0=all**
match	i=ignore case, c=case sensitive (additional, less commonly used values allowed as well)

Table 9. REGEXP_REPLACE parameters

REGEXP_REPLACE EXAMPLES

REGEXP_REPLACE is used to replace the contents of strings that match a regular expression pattern. The replace string can also include additional regular expression patterns.

```
SELECT REGEXP_REPLACE('string', 'regex', 'replace', 1, 0, 'i') AS text;
```

Example 4.1:

Remove leading zeroes from a network identifier. Values that do not begin with 0 should not be altered. Results after each query.

```
SELECT REGEXP_REPLACE ('PON-01','^0+',NULL)      -- PON-01
SELECT REGEXP_REPLACE ('01A','^0+',NULL)         -- 1A
SELECT REGEXP_REPLACE ('00','^0+',NULL)          -- zero length string
```

- `^` assert position at the start of the string
- `0+` one or more zeroes
- `NULL` replacement value, i.e. remove zero, replace with nothing

Example 4.2:

Since different characters can be used as separators between phone number components, typically when storing phone numbers all separators are removed. Rather than executing multiple oTranslate function calls, one REGEXP_REPLACE can be used to accomplish the same result. The result of each query below is 5196472472.

```
SELECT REGEXP_REPLACE('519-647-2472','[^0-9]','',1,0)    AS PHONE_NO;
SELECT REGEXP_REPLACE('519.647.2472','[^0-9]','',1,0)    AS PHONE_NO;
SELECT REGEXP_REPLACE('519 647 2472','[^0-9]','',1,0)    AS PHONE_NO;
SELECT REGEXP_REPLACE('(519) 647-2472','[^0-9]','',1,0) AS PHONE_NO;
```

- `[^0-9]` match any non-digit
- `''` replace with that which rocks dream of - nothing
- `1` start at position 1
- `0` replace all occurrences

Example 4.3:

Sometimes we only want to replace a string if it is preceded or followed by something specific while other occurrences of the string are to be left unchanged. The nasty call log data example below contains Province / City / NPA-NXX groups separated by commas (red font below is one such group, blue another). Unfortunately, when multiple NPA-NXX values are found for a single Province / City, those multiple NPA-NXX values are also comma delimited. Since two different delimited groups of values use the same delimiter, one of the delimiters must be changed to allow the data to be parsed accurately.

The data as found in the log:

`New Brunswick/KEDGWICK/506283, 506284, New Brunswick/BLACKVILLE/506586, 506843,`

The data as required for accurate parsing:

`New Brunswick/KEDGWICK/506283;506284, New Brunswick/BLACKVILLE/506586;506843,`

The group delimiting commas have to stay, while those delimiting NPA-NXX values must be changed to semi-colons. Note: there may be other ways to solve this problem using other regex patterns / instructions (perhaps “lookaround”?) – those solutions will be left to the reader’s imaginative experimentation. ☺

```
with test_data as (
    SELECT 'New Brunswick/KEDGWICK/506283, 506284, New
Brunswick/BLACKVILLE/506586, 506843,' as log_value
    )
SELECT REGEXP_REPLACE(log_value,'\,\s+(\d)',';\1',1,0)      as fixed_value
   from test_data;
```

- `log_value` source column name
- `\,\s+(\d)`
  - `\,` match comma
  - `\s+` one or more white space characters
  - `(\d)` match a digit, capturing group
- `;\1` replacement string, replace with ; and contents of capturing group
  - `;` literal semi-colon
  - `\1` back reference to the digit matched by the first capturing group (\d)
- `1` starting position in the string
- `0` replace all occurrences

Capturing groups were necessary to retain the digit portion of the matched string since the digit is required in the final result. See the Recommended Reading section for links related to capturing groups and back references.

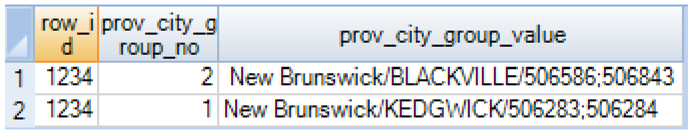
Now that the delimiters between the numeric values has been replaced with semi-colons, the big Province / City / NPA-NXX values chunks can be extracted using a comma delimiter, perhaps by a query involving

REGEXP_SUBSTR as seen below. Since the number of Province / City / NPA-NXX groups is unknown, SYS_CALENDAR.CALENDAR is being used as an "iterating index" in REGEXP_SUBSTR to extract all occurrences of the groups in the source string.

```
select    row_id
     ,    day_of_calendar          as prov_city_group_no
     ,    regexp_substr(fixed_value,'[^,]+',1, day_of_calendar)
                                   as prov_city_group_value
  from      fixed_separators  a,
           ( select day_of_calendar
              from sys_calendar.calendar
             where day_of_calendar between 1 and
                  ( select max(length(fixed_value) -
                                length(oreplace(fixed_value,',', NULL)))
                         from fix_separators )
           )  b     -- sub-query to avoid large product join with CALENDAR

 where       b.day_of_calendar between 1 and (length(a.fixed_value) -
                  length(oreplace(a.fixed_value, ',', NULL)))
```

Results:



	row_id	prov_city_group_no	prov_city_group_value
1	1234	2	New Brunswick/BLACKVILLE/506586;506843
2	1234	1	New Brunswick/KEDGWICK/506283;506284

5. REGEXP_SPLIT_TO_TABLE

REGEXP_SPLIT_TO_TABLE differs from its cousin STRTOK_SPLIT_TO_TABLE in that a regular expression is used to define the delimiting character(s) rather than STRTOK_SPLIT_TO_TABLE's single value. The *SPLIT_TO_TABLE functions allow the user to transpose multiple, delimited values from one column into a single value in multiple rows.

	Primary Key	Value
Original	1	123;456;789
After SPLIT_TO_TABLE	1	123
	1	456
	1	789

The *SPLIT_TO_TABLE functions are unlike any other string handling functions in the SQL world. Unfortunately their functionality is restricted by their somewhat crippled implementation - IMHO. ☺ The output of these functions is limited to a key value, a count and the extracted value. The key value must then be used in subsequent queries to join back to the original table if additional columns from the original table are required in the final result set. See the appendix for a complex example that required the creation of an intermediate key using the HASH_MD5 function.

In particular, the Teradata *SPLIT_TO_TABLE functions are quite finicky and strangely uncooperative at times (may not apply to other database vendors implementations or more skillful Teradata users), e.g.

- the source "table" could not be a temporary table structure created with a WITH clause in *some* conditions
- only INTEGER and VARCHAR data types could be used as output keys, despite documentation indicating NUMERIC could be used
- parsed output could only be delivered in UNICODE character set, not LATIN, thus doubling size requirements
- output key had to be the same character set as the parsed output

One hopes database vendors, particularly Teradata, will continue to improve this function to enhance its usefulness.

```
select     *
  from   table (
    REGEXP_SPLIT_TO_TABLE(source_table.outkey
                        ,  source_table.string_to_be_parsed
                        , 'regex', 'i')
         RETURNS  ( outkey             varchar(32)  character set unicode
                 ,  parsed_cnt         integer
                 ,  parsed_value       varchar(8000)  character set unicode
                   )
               ) AS split
```

Parameter	Use / Values
source_table.outkey	source table column to be used as output data's key column
source_table.string_to_be_parsed	source table column be parsed
regex	regular expression that defines delimiting criteria
match	i=ignore case, c=case sensitive (additional, less commonly used values allowed as well)
outkey	definition of column containing outkey – Teradata required UNICODE since outkey character set must match parsed_value character set
parsed_cnt	definition of column storing parsed value count, 1,2,3 …
parsed_value	definition of column storing parse value – Teradata required UNICODE (which requires 2x the length)

Table 10. REGEXP_SPLIT_TO_TABLE parameters

REGEXP_SPLIT_TO_TABLE EXAMPLE

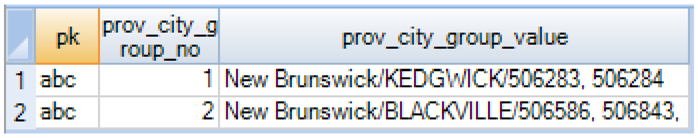
Example 5.1:

Example 4.3 in the REGEXP_REPLACE section illustrated a method to replace only *certain* delimiting commas with semi-colons to allow parsing of in the source string via the iterative REGEXP_SUBSTR query. REGEXP_SPLIT_TO_TABLE will allow us to skip the intermediate semi-colon delimiter change and use a different regular expression to identify the big Province / City / NPA-NXX chunks and parse them into rows all in one step.

```
with test_data as (
    SELECT 'abc' as pk, 1 as hee, 'def' as haw,
               'New Brunswick/KEDGWICK/506283, 506284, New
Brunswick/BLACKVILLE/506586, 506843,' as log_value
    )
select *
  from  table (
    REGEXP_SPLIT_TO_TABLE(test_data.pk
                        ,  test_data.log_value
                        , '\,\s+(?=[^0-9])', 'i')
     RETURNS ( pk                     varchar(6)   character set unicode
            ,  prov_city_group_no     integer
            ,  prov_city_group_value varchar(128) character set unicode
             )
               ) AS sg
```

o `test_data.pk`	source column to be used as output key
o `test_data.log_value`	source column to be parsed
o `\,\s+(?=[^0-9])`	
`\,`	match comma
`\s+`	one or more white space characters
`(?=[^0-9])`	positive lookahead – does non-digit character follow comma & whitespace ?
o `i`	case insensitive – doesn't apply here, but function requires the parm
o `pk`	output key column definition
o `prov_city_group_no`	parsed value count column definition
o `prov_city_group_value`	parsed value column definition

Results:



	pk	prov_city_group_no	prov_city_group_value
1	abc	1	New Brunswick/KEDGWICK/506283, 506284
2	abc	2	New Brunswick/BLACKVILLE/506586, 506843,

Teradata's REGEXP_SPLIT_TO_TABLE function requires that the parsed value column be UNICODE and that the output key character set be the same.. However, Teradata's STRTOK_SPLIT_TO_TABLE *does allow* the output column key to be LATIN even when the parsed value is UNICODE. In the words of Ralph Waldo Emerson, "A foolish consistency is the hobgoblin of little minds, adored by little statesmen and philosopher and divines". ☺

See the appendix for a comprehensive example that employs both STRTOK_SPLIT_TO_TABLE and REGEXP_SPLIT_TO_TABLE.

CONCLUSION

It's often easiest and most comfortable to fall back into the familiar SAS world that we know so well. Unfortunately, doing so often forces us to move *far* too much data from the database down to the SAS server for processing. Often, the results must then be pushed back to the database for downstream use. Since we want to be good stewards of the resources at our disposal, it's often necessary to push the envelope, move outside our comfort zone and stretch the cranial cavity. SQL functionality has continued to mature, new features and capabilities are added in each release – **take advantage of the right tool for the right job in the right place to do things efficiently**.

You may have heard that regular expressions are referred to as "write once, read never" code. That may be partially true, but to ensure your code is as maintainable as possible, **please do document well**. As seen in the paper's examples, it's helpful to document each token, anchor and qualifier within the regex to provide the most granular information possible.

REFERENCES

http://docs.openlinksw.com/virtuoso/fn_regexp_instr.html
http://www.regular-expressions.info/lookaround.html
http://www.regular-expressions.info/brackets.html
http://www.regular-expressions.info/backref.html
http://www.regular-expressions.info/replacebackref.html

ACKNOWLEDGMENTS

I owe a debt of gratitude to a number of folks who've helped me with regular expressions and the REGEXP* SQL functions. Unfortunately most of them I only know through their avatar or screen name on various forums and they will remain oblivious to how much they've helped me and so many others. Two people must be specifically acknowledged since they have directly contributed towards this paper's content:

1. Dieter Noeth, the guru of all things Teradata, who answered some questions around the prickly bits of REGEXP_SPLIT_TO_TABLE. If Dieter doesn't know it, you don't need to. ☺

2. Alice Lieu who patiently proofread the paper and offered very helpful content suggestions. Any errors that remain are mine.

RECOMMENDED READING

- http://www.regular-expressions.info/
- https://www.regexbuddy.com
- https://regex101.com/ → **GREAT** tool to develop and test regex

CONTACT INFORMATION

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

Name:	Harry Droogendyk
Company:	Stratia Consulting Inc.
Web:	www.stratia.ca

APPENDIX

The code below is a real-life example showing some of the REGEXP* functions used in the wild. The input data from &_intTable._raw is similar to the data found in the LOG_VALUE examples in the main body of the paper. Each temp table appears on a separate page to allow entire sections to be seen together. Teradata requires WITH temp tables to be defined in reverse order. Soooo, start on the 2nd last page, scroll up to here, then back to the last page….

```
proc sql noprint;
   connect to teradata ( user=me pass="my pass" server=myServer mode=teradata connection=global);

   /*  I couldn't get the REGEXP_SPLIT_TO_TABLE thing to work with a WITH temp table, creating volatile

       ROW_ID is DECIMAL, not acceptable to REGEXP_SPLIT_TO_TABLE, convert to CHAR
   */

   execute (
      create multiset volatile table post_trunc as (
         select    transaction_id
               ,  cast(cast(row_id as format 'z(16)' ) as char(16))                     as row_id
               ,  operation_id
               ,  audit_dtm
               ,  auditobject_3
               ,  auditobject
               ,  auditfunction
               ,  auditfield
               ,  case when operation_id = 'Add' then new_value else old_value end      as audit_value
               ,  old_value
               ,  new_value
               ,  userid
               ,  load_dtm
           from &_intTable._raw
          where auditfield  = 'Geographical Tree'
            and audit_dtm   > '2015-03-25 00:00:00'
            )   with data primary index (row_id) on commit preserve rows
         ) by teradata;
```

```
execute (
   create multiset volatile table &_intTable._1 as (

      /*  Create something reasonable out of the weirdness in the geographical entries in Audit Log
            - yes, intermediate WITH tables must be defined in reverse order, yeah Teradata
      */

      /* 3. split out NPANXX values and bring in province / city extracted in previous step.  The MD5 hash_key
            created in the previous step is our unique joining key.

            strtok_split_to_table(
                  hash_key            key value to be applied to each new row created, used to tie back to previous temp table
                  npanxx_group_value  string to be split
                  ','                 delimiter
                           )

               returns   (
                  hash_key            key value to be applied to each new row created, used to tie back to previous temp table
                                      which contains row_id which allows us to join to the original data.
                  npanxx_no           extracted piece count
                  npanxx_value        extracted piece, must be UNICODE apparently, UNICODE requires 2x the length of LATIN,
                                      hence 12 rather than the 6 length of the NPANXX value
                     )

            strtok_split_to_table is finicky.  The    npanxx_group_value must be UNICODE ( requires double the normal length ),
            but the hash_key can be LATIN, unlike what's required for REGEXP_SPLIT_TO_TABLE - go figure.  Joining to previous temp
            table to grab ROW_ID, Province and City name values.
      */

      with npanxx_split as (
         select sstt.*, pcg.row_id, pcg.province_nm, pcg.city_nm
           from table (
               STRTOK_SPLIT_TO_TABLE(prov_city_groups.hash_key, prov_city_groups.npanxx_group_value, ',')
                      RETURNS (   hash_key        varchar(32) CHARACTER SET LATIN
                              ,   npanxx_no       integer
                              ,   npanxx_value    varchar(12) CHARACTER SET UNICODE )
                        ) AS      sstt,
               prov_city_groups   pcg
          where sstt.hash_key   = pcg.hash_key
            )
```

```
/*  2. prov_city_group_values are made up of 3 components:
          a. province name
          b. city name
          c. NPA/NXX values separated by commas

          [^/}+
             ^  - start of string
             /  - delimiter
             +  - be greedy, go to end of string

          We must create a unique key of the row_id || province || city to allow us to join
          the npanxx_split back to prov_city_groups.  The REGEXP_SPLIT_TO_TABLE and STRTOK_
          SPLIT_TO_TABLE functions only allow the "outkey" to be carried into the result set
   */

   , prov_city_groups as (
      select    row_id
             ,  prov_city_group_no

             ,  cast(trim(regexp_substr(prov_city_group_value,'[^/]+',1,1))     as varchar(32)) as province_nm
             ,  cast(trim(regexp_substr(prov_city_group_value,'[^/]+',1,2))     as varchar(64)) as city_nm
             ,  regexp_substr(prov_city_group_value,'[^/]+',1,3)                as npanxx_group_value

             ,  sysudtlib.hash_md5(coalesce(row_id, '0') || coalesce(province_nm, '0') || coalesce(city_nm, '0')   )         as hash_key

        from split_groups
         )
```

```
/*  1. Split out the province/city/npanxx groups

         Source:   one row   New Brunswick/KEDGWICK/506283, 506284, New Brunswick/BLACKVILLE/506586, 506843,
                             --------------------------------------------------------------------------------
                             12345678901234567890123456789012345678901234567890123456789012345678901234567890
                                     1         2         3         4         5         6         7         8

         Result:   two rows  New Brunswick/KEDGWICK/506283, 506284,
                             New Brunswick/BLACKVILLE/506586, 506843,

         regexp_split_to_table(
            row_id                 key value to be applied to each new row created, used to tie back to full audit data record
            audit_value            string to be split

            '\,\s(?=[^0-9])'       match comma followed by whitespace that is NOT followed by a digit
                  \,               escape literal comma
                  \s+              white space, 1 or more
                  (?=[^0-9])       ?= positive lookahead, assert that the regex [^0-9] can be matched, in sample
                                   above, only matched by ", " at position 38-39 because position 40 is not a digit

            'i'                    case insensitive, meaningless because we're not matching alpha characters, but function requires parm
                     )

         returns   (
            row_id                 key value to be applied to each new row created, used to tie back to full audit data record
            prov_city_group_no     extracted piece count
            prov_city_group_value  extracted piece, must be UNICODE apparently, UNICODE requires 2x the length of LATIN,
                                   hence 8,000 rather than the 4,000 length of the audit_value field
               )

         regexp_split_to_table is finicky.  The outkey ( row_id ) must be character, hence the conversion in post_trunc.  The
         npanxx_group_value must be UNICODE ( requires double the normal length ).
   */

   , split_groups as (
      select    *
        from  table (
          REGEXP_SPLIT_TO_TABLE(post_trunc.row_id, post_trunc.audit_value, '\,\s+(?=[^0-9])', 'i')
                RETURNS (   row_id                 varchar(32)   CHARACTER SET UNICODE
                        ,   prov_city_group_no     integer
                        ,   prov_city_group_value  varchar(8000) CHARACTER SET UNICODE )
                     ) AS sg
         )
        /* 4. Bring it all together, oh my nerves... this is soooooo exciting.  :-)

           ----------------------------------------------------------------------------------------------
           Some transactions have duplicate npanxx values, eg. transaction_id = 235221.  Dedup by program,
           transaction_id, audit_dtm, operation_id
           ----------------------------------------------------------------------------------------------

           Need the trim() because some values have leading spaces.
        */

        select    a.transaction_id
              ,   cast(a.row_id as decimal(18))   as row_id
              ,   a.operation_id
              ,   a.audit_dtm
              ,   a.auditobject_3
              ,   a.auditfield                    as geo_target_qualifier_cd
              ,   ns.province_nm
              ,   ns.city_nm
              ,   trim(ns.npanxx_value)           as npanxx_from
              ,   trim(ns.npanxx_value)           as npanxx_to
              ,   a.userid
              ,   a.load_dtm

          from    post_trunc      a,
                  npanxx_split    ns

          where   a.row_id        = ns.row_id
            and ns.npanxx_value   is not null

        /*  Need to dedup extract values, there are dups in the list of NPA-NXX values  */

        qualify row_number() over ( partition by auditobject_3, transaction_id, audit_dtm, operation_id, npanxx_from
                                    order by load_dtm desc
                          ) = 1

        )   with data primary index (auditobject_3) on commit preserve rows
     ) by teradata ;

quit;
```