Regular Expressions: Say What?

Support for Regular Expressions were added to the Oracle database to expand on the single and multi character wildcard searches to allow for more complicated search patterns. This greatly enhances the searching capabilities, however writing regular expressions can prove quite tricky, sometimes it requires a different way of thinking.

Do you really need to learn about Regular Expressions? And why should you learn about Regular Expressions *now*?

Just to give you a reason: Data redaction and the MATCH_RECOGNIZE clause also use Regular Expressions as part of their syntax. If you want to use these very powerful functions of the Oracle 12c database, you will have to learn Regular Expressions. Neither function will be covered in this article.

Regular Expressions Functions

When the exact spelling isn't known, the Oracle database allowed for searching using a wildcard notation. Traditionally this functionality was limited to zero or multiple character wildcards. Using an underscore (_) a single character could be used as a wildcard. With the percentage sign (%) zero or more characters could be used as the unknown characters.

Starting with Oracle database 10g regular expressions are supported, these functions can be identified by the prefix REGEXP.

The following functions are available:

REGEXP_LIKE

REGEXP_SUBSTR

REGEXP_INSTR

REGEXP_REPLACE

REGEXP_COUNT

The last function, REGEXP_COUNT, was introduced with Oracle database 11g.

All of these regular expression functions can be used in SQL as well as PL/SQL. The names of these functions express what these functions do. They are comparable to the non-regular-expression counterpart. The non-regular-expression REPLACE has a regular expression counterpart REGEXP_REPLACE and so on.

Depending on which regular expression function is used, the number of arguments will vary, in general the first argument is the source string and the second argument is the regular expression pattern.

With regular expression there are a number of meta characters to describe the pattern that you are looking for. A full list including a short description of these metacharacters are listed in the table below.

^	Matches the beginning of a string. If used with a match_parameter of 'm', it matches the start of a line anywhere within expression. The meaning inside a bracket expression is different: see [^]
$	Matches the end of a string. If used with a match_parameter of 'm', it matches the end of a line anywhere within expression.
*	Matches zero or more occurrences.
+	Matches one or more occurrences.
?	Matches zero or one occurrence.
.	Matches any character except NULL
|	Used like an "XOR" to specify more than one alternative.
[]	Used to specify a matching list where you are trying to match any one of the characters in the list.
[^]	Used to specify a non matching list where you are trying to match any character except for the ones in the list.
()	Used to group expressions as a subexpression
{m}	Matches m times.
{m,}	Matches at least m times.
{m, n}	Matches at least m times, but no more than n times.
\n	n is a number between 1 and 9. Matches the nth subexpression found in a bracket expression.

[..]	Matches one collation element that can be more than one character.
[::]	Matches character classes.
[==]	Matches equivalence classes.

Looking for telephone numbers in a piece of text is easy for a human to do, even when the telephone number has different formats. In order to find the telephone numbers using SQL is a lot harder. This is where Regular Expression excel.

The pattern that we could write to extract the telephone number might look like the following expression:

```
...-...-....
```

Each period represent a character, you can tell that by looking at the metadata characters from the table above.

With an expression like this, you might get false positives, pieces of the text which you're not looking for but got returned anyway.

A better expression would be the following:

```
[0-9]{3}-[0-9]{3}-[0-9]{4}
```

Now the expression only matches three numbers followed by a hyphen, followed by three number, followed by a hyphen and lastly four numbers.

But what if there is a different character used as separator between the numbers? Say for instance a space is used? The pattern will evolve to :

```
[0-9]{3}(-| )[0-9]{3}(-| )[0-9]{4}
```

Or maybe there are multiple characters used as separator between the numbers, like a period (which has special meaning, so it must be escaped)

```
[0-9]{3}(-| |\.)[0-9]{3}(-| |\.)[0-9]{4}
```

Conclusion

As you can see from the examples above, with not much explanation, Regular Expressions may become hard to interpret fast. The best advice I can give you: keep the table of metadata characters handy; read the expression out loud with the meaning in normal words like "exactly three characters in the range of zero through nine followed by either a hyphen, space or period"

and so on. Use Google,.. if you need to create a Regular Expression for a certain format, you are not alone. There is probably someone out there who already solved your problem, you just have to find it.

Alex Nuijten is an independent consultant (allAPEX), specializing in Oracle database development with PL/SQL and Oracle Application Express (APEX) and member of the Smart4APEX Guild.

Besides his consultancy work, he conducts training classes, mainly in APEX, SQL and PL/SQL. Alex has been a speaker at numerous international conferences, and received several Best Speaker awards.

He wrote many articles in Oracle related magazines, and at regular intervals he writes about Oracle Application Express and Oracle database development on his blog "Notes on Oracle" (nuijten.blogspot.com). Alex is co-author of the following books "Oracle APEX Best Practices" (published by Packt Publishers) and "Real World SQL and PL/SQL" (published by Oracle Press).

Because of his contributions to the Oracle community, Alex was awarded the Oracle ACE Director membership in August 2010.

You can contact Alex via email: alex@allapex.nl