# Introducing Oracle Regular Expressions

*An Oracle White Paper*
*September 2003*

ORACLE

# Introducing Oracle Regular Expressions

## INTRODUCTION

Regular expressions provide a powerful means of identifying a pattern within a body of text. A pattern describes what the text to identify looks like and could be something quite simple, such as that which describes any three-letter word, to something quite complex, like that which describes an email address.

Oracle Regular Expressions provide a simple yet powerful mechanism for rapidly describing patterns and greatly simplifies the way in which you search, extract, format, and otherwise manipulate text in the database. The remainder of this document introduces Oracle regular Expressions and describes how to leverage their power using SQL and PL/SQL.

### History of Regular Expressions

If you are familiar with UNIX or Perl then you almost certainly have used regular expressions to search for or manipulate text data. Indeed, the indispensable searching tool 'grep' shares its middle two initials with the technology, standing for 'Global Regular Expression Print'. Adoption of regular expressions by the mainstream developer community was fueled mostly by web based technologies and the need to manipulate text data coming from and going to the browser. With the advent of CGI, web programmers were able to leverage regular expression capabilities from UNIX tools in order to perform important tasks that could not be done as easily before. While an impressive list of languages, such as Java, PHP, XML, and Python, have incorporated support for regular expressions, Perl stands out as the one language that helped their popularization beyond UNIX based platforms.

Pattern matching with regular expressions can be applied to all kinds of real word problems. They are used heavily in web applications to verify, parse, manipulate, and format data coming to and from the middle tier; a surprisingly large amount of middle tier processing is taken up with such string processing. They are used in bioinformatics to assist with identifying DNA and protein sequences. Linguists use regular expressions to aid research of natural languages. Server configuration is often done in terms of regular expressions such as in a mail server to help identify potential spammers, and are perhaps also used by the spammers themselves to effortlessly collect innocent victims email addresses from Internet based data stores. Many protocol and language standards accept regular expressions as filters

and validation constructs. In short, it is hard to imagine an application that could not benefit from the functionality that regular expressions offer.

Historically, regular expressions have been the domain of expert users, such as administrators or developers. Casual application users will most likely never come across them but that is not to say that they could not benefit from their power, it is mainly that the syntax is often not very intuitive to non-experts. This appears to be changing; everyone should take a second look to see if there is a task that could be simplified by using regular expressions. Indeed, many string searching and manipulation tasks cannot be performed without regular expressions. How would you attempt to search for all Internet addresses in a document? Most applications do not allow you to perform such pattern matching but once you become accustomed to their availability, you will surely wonder how you ever managed to live without them.

### Traditional Database Pattern Matching

A simple form of pattern matching has long been part of Oracle SQL through use of the LIKE condition that provides two variants of wildcarding with the zero or more of any character *percent* (%), and any character *underscore* (_). Such characters are termed metacharacters as they describe the pattern rather than partake in it. A simple example could be LIKE 'abc% 'that matches any row that beginning with 'abc'. In many cases, LIKE is limiting and other solutions have been sought as a means to perform more complex queries. Suppose you wanted to construct a query to search the company employee database for both 'John 'and 'Jon'. There is no way to specify a single pattern that represents these variants using LIKE as there is no means to express that a particular portion of text does not have to be present to prove a match, in this case the optional 'h'. Note that LIKE 'Jo%n 'is not restrictive enough for our task as it would also match 'Joan 'and 'Johan'.

To remedy this limitation, applications could perform pattern matching operations by constructing complex SQL or PL/SQL logic. This approach is both hard to write and difficult to maintain as even the simplest of logic could consume hours of development time. Often there is no way to represent a pattern in SQL alone, consider how difficult it would be to extract all email addresses from a body of text using SQL. The query would be almost impossible and would translate to many lines of PL/SQL source. It should be noted that although Oracle already provides a simple PL/SQL pattern matching package (OWA_PATTERN) it is far from being a full regular expression implementation and does not support several key required features such as alternation, the OR condition that allows matching either of two expressions. In this respect it is limiting in its ability to perform complex pattern matching operations.

It is still possible to obtain good regular expression support in the database by creating user-defined functions to publicly available regular expression libraries. While fine as a temporary solution, this approach has many limitations and issues. Callouts are difficult to develop, deploy, maintain, and in most cases, difficult for

the vendor to support. As they are not native to the database, callout solutions are rarely able to inherit database features such as locale support, performance enhancements, and most likely cannot handle larger datatypes. Perhaps the most limiting is that applications built on databases cannot rely on the callouts being available so it is difficult for developers to make use of them without requiring that they be deployed, often a risk not worth taking.

Some Oracle based applications use third party regular expression technology at the client level to provide pattern matching capabilities. Typically, data from the database is being extracted to a client, manipulated with regular expressions then passed on to another source, or even turned around and written right back to the database. Alternatively, data from a source external to the database is often manipulated with regular expressions before it is written in to the database, or possibly even filtered and rejected as input. It is evident that this approach is far from ideal as data is being pulled back and forth between database and client primarily in order to access regular expression functionality. In a similar manner to that dictated by the object-oriented paradigm, it is desirable that this logic be as close to the data as possible.

### Oracle Regular Expressions

The web paradigm has reached a reasonably stable state in terms of major software components. Most large-scale web applications consist of a database backend, a web server middle tier with access to a database, and the front end. While the middle tier technology employed tends to be quite dynamic in nature, ranging from simple CGI to large scale Java applications, the backend remains a solid component and in most cases is a database that understands SQL or a similar query language.

Oracle Regular Expressions build on the limitations of existing solutions by providing a sophisticated set of metacharacters for describing more complex patterns than previously possible, all native to the database. Leveraging their power, an application will not only run more efficiently but it can have an improved data flow and be more robust.

### Enhancing the Data Flow

While middle tier technologies have long had the ability to perform regular expression searching, support in the backend database is a valuable and often overlooked consideration. The introduction of Oracle Regular Expressions brings the database closer to the Internet by providing enhanced string manipulation features right within the database, providing the flexibility to perform regular expression based string manipulation at any tier.

Oracle Regular Expressions can be used in many data manipulation scenarios such as updating, selecting, and formatting for presentation. These scenarios are described in the following sections in terms of the data flow.

#### Database to Client

Regular expressions are especially helpful in a web application where data from the database needs to be filtered and formatted for presentation. As an example, consider an application that selects a phone number stored within a CHAR(10) column as a series of 10 digits in the format `XXXXXXXXXX`. The requirement is to format this column as `(XXX) XXX-XXXX` for presentation to the end user. Should this processing occur in the middle tier then other clients who have access to the same data will need to duplicate the formatting logic. A DBA querying the table through SQL*Plus, for example, will only be able to view the format as stored on disk without incorporating their own means of formatting. This forces all clients of the data to have special knowledge of how the telephone number is stored within the database; when clients have such an expectation it becomes difficult to change the backend format. As a means to resolve this it would be a trivial task to create a database view that uses the regular expression enabled replace function (REGEXP_REPLACE) to reformat the telephone number. All clients of the data will then simply need to query the view and will benefit from the centralized logic providing them only the pre-formatted version of the telephone number.

Regular expressions are often used in a client to filter and refine a result set to the desired rows. With the regular expression enabled LIKE condition (REGEXP_LIKE) this filtering can now easily occur within the database directly on the data minimizing network data flow and putting the job of getting the correct data right where it ought to be, in the hands of the database. Moving complex logic closer to the data in this manner creates a more robust environment as data not targeted for the end user does not have to leave the database and client processing is reduced.

#### Database Update

Oracle Database 10G allows you to perform updates in terms of regular expressions. Where traditionally it would have been easiest to select the query set, perform regular expression based updates within a client and write the results back to the database, it is now possible to perform all of this in a single update statement. The data never has to leave the database bringing benefits such as reduction in network traffic, tighter security, and improved performance.

#### Client to Database

Where regular expressions normally play a part in data flowing from client to database is in validating and formatting data for storage.

We may want to validate that a credit card number or an email address matches a certain pattern or we may want to reformat a user provided telephone number into a format that will make more sense when stored in the database. Much like the flow from database to client, while it is possible to perform such manipulation on the middle tier, only clients of the middle tier would have access to this logic.

When a regular expression describes the required format for data in a column, the regular expression itself is a property of the data and therefore should not be externalized in client logic. With column constraints defined in terms of regular expressions, we can bulletproof our database so that only data matching a certain pattern will be allowed into a table irrespective of the source that the data originates from, be it an external client or even an internal PL/SQL routine.

With the above scenarios in mind, it is evident that the combination of Oracle Regular Expressions and SQL is an extremely powerful one that can drastically change the way in which string manipulation and pattern matching is performed within an application.

## ORACLE REGULAR EXPRESSIONS KEY FEATURES

Most regular expression implementations are based to some extent on the documented behaviour of Basic and Extended Regular Expressions (BRE and ERE) as described in the POSIX[1] standard, often referred to as UNIX style regular expressions. The standard leaves plenty of room for extensions that most regular expression implementations readily take advantage of which means that in general, no two implementations are alike. It is because of this incompatibility issue that Oracle Regular Expressions are based on the POSIX ERE definition. This assures that any Oracle Regular Expression will have the same behaviour on similarly conformant implementations.

POSIX style regular expressions are currently not part of the SQL standard[2]. The standard does however introduce the SIMILAR predicate for pattern matching which could be considered a version of LIKE with regular expression support. SIMILAR maintains support for the LIKE behaviour of percent (%) and underscore (_) and adds some features that are found in POSIX but misses some key important features such as backreferences, interval quantifiers, and anchors. Furthermore, the standard has no means to locate, extract, or replace the matching pattern as SIMILAR only places focus on proving the existence of a pattern, in the same manner as LIKE.

### Interfaces

Oracle Regular Expressions are implemented by the following interfaces available in both SQL and PL/SQL:

| SQL Function | Description |
|---|---|
| `REGEXP_LIKE` | Determine whether pattern matches |
| `REGEXP_SUBSTR` | Determine what string matches the pattern |
| `REGEXP_INSTR` | Determine where the match occurred in the string |
| `REGEXP_REPLACE` | Search and replace a pattern |

**Table 1**

For detailed information on the valid arguments and syntax, refer to the sections on Conditions and Functions in the SQL Reference[3].

### Metacharacters

For a complete list of supported metacharacters please refer to the Appendix C of the SQL Reference[3]. They are listed here for reference and will not be described further.

| Syntax | Description | Classification |
|---|---|---|
| . | Match any character | Dot |
| a? | Match 'a' zero or one time | Quantifier |
| a* | Match 'a' zero or more times | Quantifier |
| a+ | Match 'a' one or more times | Quantifier |
| a\|b | Match either 'a' or 'b' | Alternation |
| a{m} | Match 'a' exactly m times | Quantifier |
| a{m,} | Match 'a' at least m times | Quantifier |
| a{m,n} | Match 'a' between m and n times | Quantifier |
| [abc] | Match either 'a' or 'b' or 'c' | Bracket Expression |
| (... ) | Group an expression | Subexpression |
| \n | Match nth subexpression | Backreference |
| [:cc:] | Match character class in bracket expression | Character Class |
| [.ce.] | Match collation element in bracket expression | Collation Element |
| [=ec=] | Match equivalence class in bracket expression | Equivalence Class |

**Table 2**

### Locale Support

Locale support refers to how the regular expression will behave under the properties of a given character set, language, territory, and sort order. As regular expressions are for matching text, and text is not limited to English, it stands to reason that they should be able to handle text in any language or character set and that the unique properties of that locale should be honoured. Most standards are rather vague when it comes to supporting different locales. POSIX, while stating that there are some considerations that should be taken to support locales, does not match up to the locale definition that Oracle provides.

Oracle Regular Expression pattern matching is sensitive to the underlying locale defined by the session environment. This affects all aspects of matching including whether it will be case or accent insensitive, whether a character is considered to fall within a range, what collation elements are considered valid, and so on. The engine is also strictly character based, as an example the dot (.) will match a single character in the current character set and never a single byte of the data.

It is important to realize that the results of a particular regular expression query could be different under a different locale.

Table 3 shows how certain features are affected by the underlying locale and what property of the locale affects it. The 'Character Set' dependency refers to the database character set or national character set while 'Sort Order' refers to the setting of NLS_SORT. If NLS_SORT is set to BINARY then the matching always follows the binary properties of the character set.

| Match Type | Dependency |
|---|---|
| Character based matching | Character Set |
| Character classes | Character Set |
| String comparisons | Sort Order |
| Range matching | Sort Order |
| Collation element | Sort Order |
| Equivalence class | Sort Order |

**Table 3**

#### Character sets

All Oracle character sets are fully supported by Oracle Regular Expressions. This includes all multibyte character sets, variants of Unicode such as UTF-8 (AL32UTF8) and UTF-16 (AL16UTF16), and all shift sensitive character sets. Metacharacters always operate on the data in terms of characters and never bytes. As the input character set is always that of the database character set (or national character set), there is no way to directly refer to arbitrary byte values or characters from other character sets.

#### Case and Accent Sensitivity

The regular expression functions are sensitive to NLS_SORT as described in Table 3. In matching string literals, if NLS_SORT consider two characters equal, then so does the regular expression function. This is a very useful feature that is not available in any other means of pattern matching within the Oracle Database.

As an example, consider the following statement:

```
ALTER SESSION SET NLS_SORT=GEN ERIC_BASELETTER;
 SELECT REGEXP_SUBSTR(‘Café’, ‘cafe’) FROM dual;
 -> Café
```

The query returned the entire string ‘Café’ as the NLS_SORT setting equates ‘C’ to ‘c’ that differ in case, and also ‘é’ to ‘e’ that differ in accents. Case sensitivity can always be overridden at a statement level using the case sensitivity match options.

#### Range

When a range is specified within a bracket expression, characters are considered to be in or out of the range depending on the current setting of NLS_SORT.

The default value for NLS_SORT derives itself from the setting of NLS_LANGUAGE, which is in turn normally derived from the client environment variable NLS_LANG. While the default language sort is normally useful for sorting purposes, it is not useful for comparisons such as those that take place when determining whether a certain character falls between the range end points. As an example, should NLS_LANGUAGE be set to FRENCH then the default sort order is FRENCH where ordering is as follows a < A < b < B. Should we specify a range [a-z] then following this ordering it can be noted that every alphabetic character other than ‘Z’ will fall into this range; this is most likely not the desired behaviour. Better sorts for ranges would be those that are case insensitive that treat case variants to be equal. This can be achieved by appending ‘_CI’ to the end of a sort name such as setting NLS_SORT to FRENCH_CI.

#### Character Class

Character classes, such as [:alpha:], are sensitive to the underlying character set. Each character within a character set has a certain number of properties, which can be extracted using regular expression character classes. For example, most of the Asian multibyte character sets have full width variants of numeric characters, both the half width and full with variants are considered to be digits and will match the [:digit:] character class. Also, rather than using [a-zA-Z] to define all alphabetical characters, you could use [:alpha:] and be portable across different languages and character sets as [:alpha:] consists of all alphabetical characters in the current character set.

As a usage example, consider a simple query to convert Java names enumerated in a table to a more readable format. This works by looking for any instance of a lower case letter immediately followed by an upper case letter, something that is rarely found in English (short of McDonald) but often used the Java namespace. It records these two letters in backreferences by using subexpressions, then replaces the first letter, followed by a space, followed by the second letter:

```
SELECT
 REGEXP_REPLACE(‘StringBuffer’,
                ‘([[:lower:]])([[:upper:]])’,
                ‘\1 \2’)
```

```
  FROM dual;
  -> String Buffer
```

For a complete list of character classes see Appendix C of the SQL Reference[3]

#### Collation Element

A collation element is any valid letter within a given alphabet. Any valid collation element is allowed, for example, ‘cs’ is considered a single letter in Hungarian but not in French:

```
 ALTER SESSION SET NLS_SORT=XHUNGARIAN;
  SELECT REGEXP_SUBSTR( ‘cs’, ‘[[.cs.]]) FROM dual;
  -> cs

  ALTER SESSION SET NLS_SORT=FRENCH;
  SELECT REGEXP_SUBSTR( ‘cs’, ‘[[.cs.]]) FROM dual;
  -> ORA-12731: invalid collation class
```

This is primarily useful in ranges where you only want to match certain letters in an alphabet where one of the endpoints is a multi-letter collation element. In traditional Spanish, for example, ‘ch’ is considered a single letter but if we simply specified

```
 ALTER SESSION SET NLS_SORT=XSPANISH;
  SELECT REGEXP_SUBSTR( ‘ch’, ‘[ch]’) FROM dual;
  -> c
```

This would match a single 'c' as an individual letter and never 'ch'. It is for this reason that POSIX introduced the collation element to the bracket expression.

```
 ALTER SESSION SET NLS_SORT=XSPANISH;
  SELECT REGEXP_SUBSTR( ‘ch’, ‘[[.ch.]]’) FROM dual;
  -> ch
```

For a list of valid collation elements see Table A-10 on Monolingual Linguistic Sorts in the Globalization Guide[4]. For an introduction to sorting and collation elements see the Oracle white paper on linguistic sorting[5].

#### Equivalence Class

The equivalence class allows searching for all characters that have a common base letter. Equivalence class is sensitive to NLS_SORT and works by converting both the source and destination characters to their base (lower case with accents removed) before performing a comparison. As the character is converted to lower case, it can also be used to isolate accent and case sensitivity to a single character.

```
 SELECT
  REGEXP_SUBSTR('Oracle', '[[=o=]]racle')
  FROM dual;
```

```
-> Oracle
```

## USING REGULAR EXPRESSIONS IN ORACLE

### Function Overview

This section introduces the functions that provide Oracle Regular Expressions support and shows some simple usage scenarios. All functions have similar signatures and support CHAR, VARCHAR2, CLOB, NCHAR, NVARCHAR, and NCLOB datatypes.

#### REGEXP_LIKE

The REGEXP_LIKE condition is a little different to the other REGEXP functions in that it only returns a Boolean indicating whether the pattern matched in the given string or not. No details on how or where the match occurred is provided. There is only one optional argument being the match option, position and occurrence are redundant.

Consider a usage example where you were given the task to write an expression that could search for rows containing common inflections of the verb 'fly'. The following regular expression would do the job nicely matching fly, flying, flew, flown, and flies.

```
SELECT c1 FROM t1 WHERE
  REGEXP_LIKE(c1, 'fl(y(ing)?|(ew)|(own)|(ies))');
```

#### REGEXP_SUBSTR

This function returns the actual data that matches the specified pattern. In cases where it is not obvious how the pattern matched, REGEXP_INSTR can be called to locate the exact character offsets. Using the scenario above:

```
SELECT  REGEXP_SUBSTR(
  'the bird flew over the river',
  'fl(y(ing)?|(ew)|(own)|(ies))')
FROM dual;
-> flew
```

#### REGEXP_INSTR

The REGEXP_INSTR function performs a regular expression match and returns the character position of either the beginning or end of the match. Unlike INSTR, REGEXP_INSTR is not able to work from the end of a string. When analyzing the return value it often helps to view the match position as being the position right before the character count returned. Special care must be taken when using the output of this function as input to other SQL functions as they might not interpret the values in the same manner. Some common scenarios regarding the return values are described below.

#### *Empty String*

Any regular expression can match the empty string in a particular location. This concept does not exist in other pattern matching functions so it is important to know how to interpret the return results of REGEXP_INSTR when an empty string match occurs. Consider the following expression:

```
SELECT REGEXP_INSTR('abc', 'd?') FROM dual;
  -> 1
```

This function call returns 1 indicating that the match commenced at the first character. We know that really 'd?' matches the empty string at the beginning of 'abc' so if we apply the rule that the match commences at the position just before the character indicated by the return value then this will become a little clearer. Issuing the same query indicating that the return value be the end of the match returns 1, the match also completes on the first character.

```
SELECT REGEXP_INSTR('abc', 'd?', 1, 1, 1)
  FROM dual;
  -> 1
```

Now we know that the match starts just before character 1 and ends just before character 1 meaning that this expression matched the empty string just before the 'a'.

#### *Last Character*

The position of the end of a match consumes the last character in that match. As the actual position of the match is just before the return character, REGEXP_INSTR can return a character count that does not necessarily exist within the string. As an example, note the return value of the following query:

```
SELECT REGEXP_INSTR('abc', 'abc', 1, 1, 1)
  FROM dual;
  -> 4
```

Although there are only 3 characters in 'abc' the return value was 4. The match should be interpreted as ending before the 4th character or right after the 3rd character.

### REGEXP_REPLACE

The power of Oracle Regular Expressions really becomes evident when coupled with the ability to replace the pattern matched. This function works by looking for an occurrence of a regular expression and replacing it with the contents of a supplied text literal. The replacement text literal can also contain backreferences to subexpressions included in the match giving extremely granular control over your search and replace operations.

#### *Simple HTML Filter*

With REGEXP_REPLACE it is simple to filter out certain parts of data. This example shows how a very simple HTML filter could be written.

```
SELECT REGEXP_REPLACE (c1, '<[^>]+>')
  FROM t1;
```

#### *Using Backreferences*

References to matched subexpressions are valid within the replacement string and are identified as \n where n refers to the nth subexpression. In order to specify the backslash (\) to be part of the replace expression, it must be escaped with the backslash as in '\\'.

### Arguments

Oracle Regular Expression functions take a number of arguments that affect the way in which matching will occur. All functions take the mandatory source string and regular expression and the optional match parameters but beyond these there are several other important arguments. All arguments have defaults but in SQL there is no mechanism to allow you to explicitly indicate that you want the default behaviour without leaving the arguments out of the function call. This means that if you want to specify the right most argument but require the default behaviour of those proceeding, you are required to explicitly define the default values.

Both the source string and the regular expression itself are SQL character expressions and can just as easily come from a column than a text literal. The regular expression itself must be valid otherwise the function will return a compilation error.

#### Offset

All REGEXP SQL functions provide an offset argument to indicate the character position to commence the matching. It should be noted that the caret (^) anchor is always tied to the beginning of the string rather than the offset so a pattern that is anchored with an offset greater than 1, will never match:

```
SELECT REGEXP_SUBSTR('bbb', '^b', 2) FROM dual;
  -> NULL
```

One exception to this rule is when the multiline option is set, as the anchor could match an embedded newline character within the string.

#### Occurrence

All functions also take an occurrence that specifies you require the $n^{th}$ matching expression in REGEXP_SUBSTR and REGEXP_INSTR, the default for which is 1. For REGEXP_REPLACE the occurrence defaults to 0 meaning replace all matching expressions, otherwise it will only replace the expression specified in the

occurrence parameter. An occurrence for which there exists no match will return NULL.

The REGEXP_INSTR function takes an optional position parameter to indicate whether the beginning (0) or end (1) of the match is the required return value. The default is the beginning of the match.

Occurrence is particularly useful when working with delimited text, such as newline, comma, or colon delimited text, as you can extract any arbitrary line or field. Consider this example that extracts the 5th field being the full name from a typical entry in the colon delimited UNIX password file:

```
SELECT REGEXP_SUBSTR(
   'joe:x:123:123:Joe Bloggs:/home/joe :/bin/sh',
   '[^:]+', 1, 5)
 FROM dual;
 -> Joe Bloggs
```

#### Multiline

The multiline match option changes the behaviour of the anchors so that they match at the start and end of each line in the string rather than the start and end of the entire string. This setting is useful when you have newline delimited text data stored in the database and you want to perform a search on a line basis.

As an example, consider a database where email is archived in its raw format within a CLOB row, one mail per row. The task at hand is to extract the Subject header from each row. We might be tempted to start out by writing the expression as `'^Subject: .*$'`. This expression would work wonders with grep as the regular expression is applied to each line in the specified file rather than the entire document. In Oracle, the expression is applied to the entire contents of the given string. In this case the anchors will match only at the first and last characters within the whole email text and the expression would fail to match so in order for this to work properly we need to enable multiline mode by specifying 'm' in the match options parameter. Our new expression would look like this:

```
SELECT
 REGEXP_SUBSTR(mail, '^Subject: .*$', 1, 1, 'm')
 FROM mail_table;
```

#### Newline

The default behaviour of the dot (.) is to match any character except the newline character, as defined by the underlying operating system. Should a column contain newline characters, the result of the expression '.*' will be to match only up to, and not including, the first newline character. The match parameter 'n' indicates that the dot (.) should match everything, including the newline character.

A simple example that demonstrates this behaviour is shown below. This assumes a UNIX platform where a newline is indicated by a single Line Feed character

denoted by decimal 10. We introduce the newline character into the sample data using the SQL CHR function and concatenation operator.

```
SELECT
 REGEXP_SUBSTR('abc' || CHR(10) || 'def' , '.*')
 FROM dual;
 -> abc
```

The above query does not consume the new line character but if we specify the newline match parameter, the query will match the entire string.

```
SELECT
 REGEXP_SUBSTR('abc' || CHR(10) || 'def' ,
                  '.*', 1, 1, 'n')
 FROM dual;
 -> abc
    def
```

By the same token, if you are searching a large document for a series of terms and anticipate using the dot in such a manner, then you will need to enable this option.

## Using with DDL

#### Constraints

You can use Oracle Regular Expressions to filter data that is allowed to enter a table by using constraints. The following example shows how a column could be configured to allow only alphabetical characters within a VARCHAR2 column. This will disallow all punctuation, digits, spacing elements, and so on, from entering the table.

```
CREATE TABLE t1 (
   c1 VARCHAR2(20), CHECK
     (REGEXP_LIKE(c1, '^[[:alpha:]]+$')));

 INSERT INTO t1 VALUES ('newuser');
 -> 1 row created.

 INSERT INTO t1 VALUES ('newuser1');
 -> ORA-02290: check constraint violated

 INSERT INTO t1 VALUES ('new-user');
 -> ORA-02290: check constraint  violated
```

As the description of allowable data is tied in with the column definition, this acts as a shield to all incoming data so it is no longer required to filter such data on the client before inserting into the database.

#### Indexes

It is normal to improve performance when accessing a table by creating an index on frequently accessed and easily indexable columns. Oracle Regular Expressions are not easily able to make use of these indexes as they rely on knowing how the data within the column begins and do not lend well to functions that seek for columns with an abstract pattern. It is possible, however, to make use of functional indexes for cases where the same expression is expected to be issued on a column within a query. Functional indexes are created based on the results

```
CREATE INDEX t1_ind ON t1 (REGEXP_SUBSTR(c1, 'a'));

  SELECT c1
  FROM t1
  WHERE REGEXP_SUBSTR(c1, 'a') = 'a';
```

#### Views

Views are a great mechanism for query subsetting or formatting data before it is presented to the end user. In this example we show how combining views with regular expressions makes it easy to reformat data. Suppose there was a requirement to mangle an email address, perhaps in order to avoid automatic detection but remain readable. One way we could do this would be to insert a space between each character. With REGEXP_REPLACE and backreferences, we are able to replace every letter by itself followed by a space:

```
CREATE VIEW v1 AS
   SELECT empno,
   REGEXP_REPLACE(email, '(.)', '\1 ') email
   FROM emp;

  SELECT email FROM v1 WHERE empno = 7369;
  ->  j d o e @ s o m e w h e r e . c o m
```

### Using with PL/SQL

The Oracle Regular Expression functions do not have to be part of a SQL statement and are fully supported as a PL/SQL built-in function.

As an example, to create a function that performs several regular expression operations on the provided data, the code would look something like the following:

```
    src := REGEXP_REPLACE (src,  '<regexp_1>');
    src := REGEXP_REPLACE (src,  '<regexp_2>');
    src := REGEXP_REPLACE (src,  '<regexp_3>');
```

PL/SQL can also be used to enhance regular expression functionality. The following shows some PL/SQL that could be used to create a function that returns the $n^{th}$ subexpression:

```
CREATE FUNCTION regexp_subx (
```

```
  input VARCHAR2,
  regx  VARCHAR2,
  subx  NUMBER) RETURN VARCHAR2
IS
  ret VARCHAR2(4000);
BEGIN
  ret := REGEXP_SUBSTR (input, regx);
  ret := REGEXP_REPLACE (ret, regx,  '\' || subx);

  RETURN (ret);
END regexp_subx;
/
```

### Performance Considerations

Due to the inherent complexity of the compile and match logic, regular expression functions can perform slower than their non-regular expression counter parts. When an expression is provided, internally a compiler begins to process the string to try to convert it to an internal format, this process also ensures that the regular expression is well formed and contains no errors. This cost alone can be expensive especially when compared to parsing the arguments of LIKE for which it is impossible to construct a badly formed expression. It is also possible that the regular expression functions run faster as the compiled regular expression is highly optimized at run time. This is especially true when a complex regular expression would have to be written in a number of different SQL conditions rather than a single regular expression based function. Bear in mind that the regular expression functions are not able to make use of normal indexes.

The compilation is a little different for REGEXP_LIKE as it is optimized to work in a manner where it is not required that Oracle proves where a match occurs, we only need to prove whether the pattern occurs in the string or not. For this reason, REGEXP_LIKE can be considerably faster than other regular expression functions and may be used as a preprocessing phase to the other, more expensive, regular expression functions.

For complex expressions that would require several LIKE statements or possibly PL/SQL logic, Oracle Regular Expressions can be considerably faster at performing the same logic within a single function but it is difficult to quantify how much faster as it depends not only on the regular expression, but also on the data being matched against. Sometimes the combination of regular expression and a particular segment of text can provide many thousands of alternatives that Oracle has to process in order to prove that a match exists, or in the worst case, that it does not exist as every possible alternative must be exhausted.

The trick to writing fast performing regular expressions is to aim at avoiding costly backtracking that most often comes with liberal use of the infinite quantifiers (+ and *). This statement is especially valid when dealing with large volumes of data,

the matching engine can consume large amounts of CPU cycles and memory particularly when the pattern does not exist as every possibility has to be exhausted.

For a full discussion on how to write fast performing regular expressions, refer to Mastering Regular Expressions[6].

## ADVANCED TOPICS

This section introduces some advanced topics that are aimed at the DBA or application developer using Oracle Regular Expressions. Some features that are particular to this implementation follow a quick overview of the architecture.

### Architecture

Oracle Regular Expressions are based on the POSIX ERE standard[1] and employs and architecture known as a Traditional Nondeterministic Finite Automata (NFA).

There is one requirement of POSIX ERE that many popular implementations, for very good reasons, choose not to conform to and that is the requirement to match the longest possible pattern. If we were to describe a pattern that tried to match "a" or "ab" as "(a|ab)" then, according to POSIX, "ab" should always succeed over "a" as it is the longer of the two. The reason most implementations do not conform to this is that it is very expensive in terms of CPU cycles to compute the longest possible match and still conform to other requirements of the POSIX standard. This approach is more expensive because even if the matching engine finds a match, it still has to cycle through all alternatives originating from the same character position to determine if any of them result in a longer match. It should be noted that greediness and longest match are two quite different things as greediness simply refers to the fact that the quantifiers, such as * and +, will attempt to consume as much text as possible for the match.

When a pattern is provided to a regular expression function Oracle attempts to compile the expression into an optimized internal form. This is very much like Java code being compiled to byte codes interpreted on a virtual machine. Oracle Regular Expressions have their own compiler and virtual machine. Once compilation has succeeded, the result is then applied to each row in the query so that recompilation does not have to occur for each row.

### Looping behaviour

Because it is possible to match an empty string with Oracle Regular Expressions, functions that take occurrence as a match option need to handle cases where consecutive empty strings could match so as to avoid needless looping. The way Oracle resolves this is by nudging forward a character in cases where a match could be caught in a loop.

This is best shown with the REGEXP_REPLACE function in the following example:

```
SELECT REGEXP_REPLACE('abc', 'd?', 'X') FROM dual;
```

```
-> XaXbXcX
```

The expression attempts to replace all occurrences of 'd' with 'X', if 'd' is not found then it replaces the empty string with 'X' anyway. The result may seem surprising as the expression seems to consume the 'a', 'b', and 'c' in its result. What is actually happening is that Oracle realizes it will be caught in a loop if it attempts to replace each occurrence of this expression so it jumps forward a single character before attempting to match the expression a second time thus resulting in the output above. This behaviour can also be found in REGEXP_SUBSTR and REGEXP_INSTR when occurrence is used.

### Backreferences

Oracle extends POSIX ERE by adding support for backreferences within a regular expression. This is considered a valid extension of POSIX ERE as the behaviour of a backslash followed by something other than a metacharacter, is undefined.

In order to specify a back reference, a group expression must precede in the regular expression. As an example, '\1(abc)' will cause error ORA-12727 as the grouping appears after the back reference. The back reference may appear within the grouping it refers to as in the expression '(a\1?)+'. Note that in this case the occurrence of the back reference must be optional otherwise the expression will never match as the back reference is only ever set when the matching engine passes through the right parentheses. A back reference that is not set is quite different to a back reference that is set and contains an empty string. A back reference that is not set will never match. A back reference which is set to the empty string that is applied to one of the infinite quantifiers (+, or *) will not cause a loop, the engine will perform one check for the empty string and pass control over to the next component in the expression.

### SQL Text Literals

Strings, known as character or text literals in SQL, have particular requirements as documented in the SQL Reference[3]. The following is not specific to regular expressions and applies to SQL text literals in general, but is particularly pertinent to Oracle Regular Expressions as they are represented within text literals.

An escape character must precede a single quotation mark within the literal:

```
SELECT REGEXP_SUBSTR('that''s all', 'hat''s')
  FROM dual;
  -> hat's
```

The alternative quoting mechanism can also be used to avoid the need to escape the quotation mark but you must be careful to choose a quote delimiter that you do not anticipate will be present within the source or regular expression with a trailing quotation mark. In this example, the quote delimiter is the single quote so we must be sure that two consecutive single quotes never appear in the regular expression.

```
SELECT REGEXP_SUBSTR(q''that's all'', q''hat's'')
  FROM dual;
  -> hat's
```

Note that this escaping mechanism is quite different to other environments where regular expressions can be found, as backslash (\) is most often required to escape itself.

### SQL*Plus

When issuing SQL from within SQL*Plus several considerations need to be made. An ampersand (&) will be treated as a define variable so should it appear in the pattern as a text literal, you will be prompted to enter it's value. This behaviour can be removed by setting the SQL*Plus variable DEFINE to be OFF.

If the SQL*Plus variable ESCAPE is set to ON then any instance of the escape variable will be stripped. This is unfortunate for regular expressions as the default escape character is backslash (\) which is a common metacharacter. To be safe it is best to ensure that ESCAPE is set to OFF before issuing a regular expression query.

### NULL and the empty string

Oracle treats NULL and the empty string ('') as the same thing. This is quite different to the normal treatment of the empty string in other regular expression implementations where it happens to match everything. With Oracle Regular Expressions, in order to be conformant with other functions defined in SQL, the explicitly empty string will cause the function to return NULL and will match nothing. Note that the empty string itself will still match in the expected manner in expressions such as '()', '(a|), and 'a?'.

## CONCLUSION

The Oracle Database 10G provides the most comprehensive functionality for developing versatile, scalable, and high performance database applications for Grid computing. String searching, manipulation, validation, and formatting are at the heart of all applications that deal with text data; regular expressions are considered the most sophisticated means of performing such operations. The introduction of native regular expression support to SQL and PL/SQL in the Oracle Database revolutionizes the ability to search for and manipulate text within the database by providing expressive power in queries, data definitions and string manipulations.

---

[1] IEEE Std 1003.1, Open Group Technical Standard
[2] ISO/IEC 9075-2:1999 SQL
[3] Oracle Database SQL Reference 10G
[4] Oracle Database Globalization Guide 10G
[5] Sorting Your Linguistic Data inside the Oracle9i Database, An Oracle Technical White Paper, September 2002

[6] Mastering Regular Expressions, Second Edition, Jeffrey, E. F. Friedl, ISBN 0596002890

**Introducing Oracle Regular Expressions**
**September 2003**
**Author: Peter Linsley**
**Contributing Authors: Barry Trute**

**Oracle Corporation**
**World Headquarters**
**500 Oracle Parkway**
**Redwood Shores, CA 94065**
**U.S.A.**

**Worldwide Inquiries:**
**Phone: +1.650.506.7000**
**Fax: +1.650.506.7200**
**www.oracle.com**