CHAPTER 16 Regular Expressions and Hierarchical Queries

Oracle SQL By Example, Fourth Edition by Alice Rischert. Published by Prentice Hall. Copyright © 2008 by Pearson Education, Inc.

CHAPTER OBJECTIVES

In this chapter, you will learn about:

- Regular Expressions
- Hierarchical Queries

This lab will expand your knowledge of regular expressions so you can fully harness their potential. You will learn about the essential metacharacters and gain an understanding of how regular expressions are implemented within the context of the Oracle database.

Hierarchical queries are discussed in Lab 16.2. You will learn how to use the CONNECT BY clause and the PRIOR operator to graphically display a hierarchy and reveal the relationship of records within a table.

LAB 16.1 Regular Expressions

LAB OBJECTIVES

After this lab, you will be able to:

- Understand the Practical Applications for Regular Expressions
- Use Regular Expression Functionality within the Oracle Database

A regular expression is a notation for describing textual patterns. It consists of one or more literals and/or *metacharacters* that specify algorithms for performing complex text searches and modifications. A simple regular expression can consist only of character literals such as the regular expression ‘hat’. You read it as the letter *h* followed by the letters *a* and *t*. It will match character strings such as ‘hat’, ‘Manhattan’, and ‘chatter’. One of the metacharacters is the match-any character (.). For example, the regular expression ‘h.t’ matches strings such as ‘hot’, ‘hat’, or ‘shutter’.

Data validation, identification of duplicate word occurrences, detection of extraneous white spaces, and parsing of strings are just some of the uses of regular expressions. You can take advantage of a regular expression to determine valid formats for phone numbers, zip codes, Social Security numbers, IP addresses, file and path names, and so on. Furthermore, you can locate patterns such as HTML tags, e-mail addresses, numbers, dates, or anything else that fits a pattern within any textual data and replace them with another pattern.

Regular Expressions and the Oracle Database

You find regular expressions in many programming languages, most notably in Perl and the UNIX grep, egrep, awk, and sed utilities. In addition, the power and flexibility of regular expressions are available in the Oracle database, through the use of the Oracle SQL operator REGEXP_LIKE, and the REGEXP_INSTR, REGEXP_REPLACE, and REGEXP_SUBSTR functions. The operator and functions work much like the familiar LIKE operator and the INSTR, REPLACE, and SUBSTR functions. In any SQL statement where you can use the LIKE operator or these functions, you can take advantage of regular expressions.

From Chapter 12, “Create, Alter, and Drop Tables,” you may recall examples that demonstrate regular expressions in column check constraints to enforce data validation rules. You can also use regular expressions in a query to find a particular pattern, determine the starting position of a pattern, extract the substring of the pattern, or replace a pattern with another pattern. Oracle supports POSIX ERE (Portable Operating System Interface Extended Regular Expressions)–compliant regular expressions. Learning the syntax of the regular expression language is useful because you can extend this knowledge to many other software products and languages.

Before using the regular expression functionality, you need to understand the meaning of the metacharacters. After this brief introduction, you will see how to apply the new Oracle operator and functions within the Oracle database.

The Match-Any and Anchoring Metacharacters

The period (.) matches any character (except a newline character) in a regular expression (see Table 16.1). For example, the regular expression ‘x.z’ matches a string containing the letter *x*, followed by any other single character (except newline), and followed by the letter *z*. The strings ‘xpz’, ‘xxyzd’, and ‘xyz’ contain this pattern.

TABLE 16.1 Match-Any Character

METACHARACTER	DESCRIPTION
<	Matches any single character except a newline.

If you want to exactly match a three-character string in which the line begins with *x* and ends with *z*, you must anchor the regular expression to the start and end of the regular expression pattern. The caret (^) metacharacter indicates the start of a line, and the dollar symbol ($) designates the end of the line (see Table 16.2). Therefore, the regular expression '^x.z$' matches the strings 'xaz', 'xoz', and 'xyz'. To contrast this approach with the familiar pattern matching available with the LIKE operator, you can express a pattern such as 'x_z', where the underscore (_) is the one-character wildcard.

TABLE 16.2 Anchoring Metacharacters

METACHARACTER	DESCRIPTION
^	Anchors the expression to the start of the line.
$	Anchors the expression to the end of the line.

Exploring Quantifiers

Regular expressions allow you to specify occurrences of a character with a *quantifier*, also called a *repetition operator*. If you want a match that starts with the letter *x* and ends with the letter *z*, your regular expression looks like this: '^x.*z$'. The * metacharacter repeats the preceding match-any metacharacter (.) zero, one, or more times. The equivalent pattern with the LIKE operator is 'x%z', with the percent symbol (%) indicating zero, one, or multiple occurrences of any character. Valid matches for the pattern are 'xz', 'xyyyyz', 'xyz', and 'xkkkkkz'. As you review Table 16.3, notice that these repetition choices allow more options than the LIKE wildcard characters.

TABLE 16.3 Quantifier Operators

QUANTIFIER	DESCRIPTION	EXAMPLE
*	Matches 0 or more times.	'ca*t' matches 'ct', 'cat', 'caat', 'caaat', and so on.
?	Matches 0 or 1 time.	'ca?t' matches 'ct' or 'cat'.
+	Matches 1 or more times.	'ca+t' matches 'cat', 'caat', and so on. 'ct' is not a valid match.
{m}	Matches exactly m times.	'ca{3}t' matches 'caaat'.
{m,}	Matches m or more times.	'ca{3,}t' matches 'caaat', 'caaaat', and so on.
{m, n}	Matches at least m times but no more than n times	'ca{3,5}t' matches 'caaat', 'caaaat', and 'caaaaat'.

The POSIX Character Classes

The POSIX character classes (see Table 16.4) allow you to specify what type of character you are looking for. You must specify the class name in lowercase; otherwise, the POSIX character class is invalid.

TABLE 16.4 Predefined POSIX Character Classes

CHARACTER CLASS	DESCRIPTION
[:alpha:]	Alphabetic characters
[:lower:]	Lowercase alphabetic characters
[:upper:]	Uppercase alphabetic characters
[:digit:]	Numeric digits
[:alnum:]	Alphanumeric characters
[:space:]	Space characters (nonprinting), such as carriage return, newline, vertical tab, and form feed
[:punct:]	Punctuation characters
[:cntrl:]	Control characters (nonprinting)
[:print:]	Printable characters

For example, a regular expression such as '[[:digit:]]{5}' shows the POSIX character class digit delimited by colons and square brackets. The second set of brackets (as in [[:digit:]]) encloses a character class list and is required because POSIX character classes can only be used to construct a character list. The '{5}' is a quantifier; it specifies the exact five repetitions of the digit character class.

Character Lists

In addition to the predefined POSIX character classes, you can create your own character classes or lists. The square brackets ([ ]) symbol indicates a character list where any of the characters can be matched. With the hyphen (-), you define a range between the starting and ending points. For example, the regular expression [0-5] includes characters with digits from 0 through 5. Multiple ranges such as [a-zA-Z] include the upper- and lowercase characters. Ranges must be in order; a range such as [z-a] is not valid. If you want to include all characters a through z, you might want to consider using the [:lower:] POSIX character class because the POSIX standard supports multilingual environment.

When characters are not in a range, they can be placed in any order, such as [5738]; this expression would find any string that contains a 5, 7, 3, or 8.

A hyphen placed as the first character of a list (for example, [-abc]) indicates a literal hyphen and not a range. Within a character class, most metacharacters are treated as character literals rather than as special metacharacter operators. The special cases are the hyphen, caret, and backslash. You will see some examples shortly that illustrate their use and help you understand when a metacharacter should be the literal character instead.

Negation of Character Lists

Some metacharacters have different meanings, depending on their position within the regular expression. You already learned about the ^ start-of-line metacharacter. Its second meaning is the negation metacharacter; it negates a character list if it is the first character in the character list. For example, '[^58]' matches any character except 5 or 8. The strings '0', '395', 'abc', and '5890' are valid matches because all or some characters are "not 5 or 8"; however, '58', '85', and '585' are not matches. Table 16.5 provides an overview of the character list metacharacters.

TABLE 16.5 Character List Metacharacters

METACHARACTER	DESCRIPTION
[^]	Negated. If this metacharacter is the first character in the character list, it negates the list. Otherwise, it's the literal ^. (^can also mean the beginning of a line outside the bracketed expression, as described in the section,"The Match-Any and Anchoring Metacharacters.")
[char]	Matching character list. Indicates a character list. Most metacharacters inside a character list are interpreted as literals, with the exception of ^ and -.
-	Range.Represents characters in a range. To match the literal hyphen and not a range, it must be the first character inside the character list (for example, '[-a-z]').
[: :]	Character class.The predefined POSIX standard character classes include linguistic ranges for the current locale.
[.char.]	Collating sequence. Indicates a POSIX collation element, useful for foreign language support.

[=char=]	Character equivalence class. Lets you search for characters in the current locale that are equivalent. This can be useful for ignoring accents and case in foreign languages.

Perl Expressions

Oracle added Perl-influenced regular expressions, which abbreviate some of the POSIX character classes. For example, you can simplify '[[:digit:]]{5}' to '\d{5}'. The opposite of no digit in POSIX is '[^[:digit:]]', and the equivalent Perl extension is '\D'. Table 16.6 shows Oracle's Perl extensions.

TABLE 16.6 Perl Extensions

METACHARACTER	DESCRIPTION
\d	Numeric digits; same as POSIX [[:digit]].
\D	Not a digit; equivalent to POSIX [^[:digit]].
\w	A word, which includes alphanumeric values and the underscore (_). The POSIX equivalent is [[:alnum:]_].
\W	Not a word containing letters, numbers, or underscores but a special character, such as punctuation. The POSIX equivalent is [^[:alnum:]_].
\s	A white space. This is the same as POSIX [[:space:]].
\S	Not a space character; equivalent to [^[:space:]].
\A	Indicates that the following should only be considered at the beginning of a line. For example, the expression '\At' considers the first occurring letter t to be a match only if located at the beginning of a line.

\Z	This metacharacter consider the end of a string. For example, the string 'Cat Dog ' contains one space between the words 'Cat' and 'Dog' and another space after 'Dog'. The regular expression '\s\Z', successfully finds the white space at the end of the line.

The REGEXP_LIKE Operator

Now that you are familiar with the most important metacharacters, you can use the REGEXP_LIKE operator to see how the regular expression functionality is applied within the Oracle database. The following SQL query's WHERE clause shows the REGEXP_LIKE operator, which searches the ZIP column for a pattern that satisfies the regular expression [^[:digit:]]. It retrieves those rows in the ZIPCODE_EXAMPLE table for which the ZIP column values contain any character that is not a numeric digit.

```
SELECT zip
  FROM zipcode_example
 WHERE REGEXP_LIKE(zip, '[^[:digit:]]')
ZIP
-----
ab123
123xy
007ab
abcxy

4 rows selected.
```

This regular expression consists only of metacharacters —more specifically the POSIX character class digit, delimited by colons and square brackets. The second set of brackets (as in '[^[:digit:]]') encloses a character class list. As previously mentioned, this is required because you can use POSIX character classes only for constructing a character list.

The ZIPCODE_EXAMPLE table exists in your schema only if you downloaded and installed the additional script available from the companion Web site, located at www.oraclesqlbyexample.com.

Following is the syntax of the REGEXP_LIKE operator.

```
REGEXP_LIKE(source_string, pattern
  [, match_parameter])
```

The SOURCE_STRING supports character data types. The PATTERN parameter is another name for the regular expression. The MATCH_PARAMETER allows optional parameters, such as handling the newline character, retaining multiline formatting, and providing control over case-sensitivity. You will see some examples of this parameter later in the lab.

The REGEXP_SUBSTR Function

The REGEXP_SUBSTR function returns the substring that matches the pattern, and the syntax is as follows.

```
REGEXP_SUBSTR(source_string,
pattern
 [, position [, occurrence
 [, match_parameter] [,subexpr]]])
```

The POSITION parameter indicates the starting position for the search, which defaults to 1, the beginning of the string. The default value for the OCCURRENCE parameter is 1, looking for the first occurrence of the pattern.

The following query uses the new REGEXP_SUBSTR function to find and return the five-digit zip code pattern within a string. The pattern requires five consecutive digits anchored to the end of the line, as indicated with the $ metacharacter; otherwise, you get the house number 12345 instead.

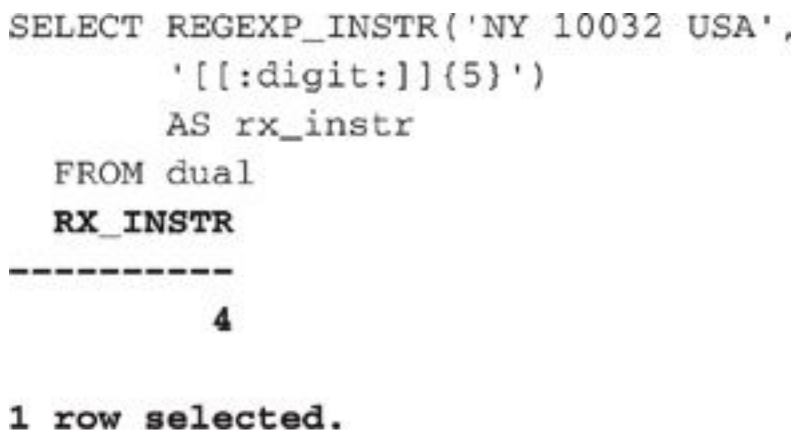
```
SELECT REGEXP_SUBSTR('Joe Smith, 12345 Berry Lane, Orta, CA 91234',
       '[[:digit:]]{5}$')
       AS substr
  FROM dual
SUBST
-----
91234

1 row selected.
```

The REGEXP_INSTR Function

The REGEXP_INSTR function works somewhat like the familiar INSTR function; however, it looks for a pattern rather than a specific string.

The next example uses the function to determine the starting position of the five-digit zip code pattern within a string.



```
SELECT REGEXP_INSTR('NY 10032 USA',
       '[[:digit:]]{5}')
       AS rx_instr
  FROM dual
  RX_INSTR
----------
         4

1 row selected.
```

You can indicate the starting position of the search and which occurrence of the pattern you want to find, both of which default to 1. The default value for the RETURN_OPTION parameter is 0, and it returns the starting position of the match. Alternatively, a RETURN_OPTION parameter value of 1 indicates the starting position of the next character following the match. The syntax of the REGEXP_INSTR function is as follows.

```
REGEXP_INSTR(source_string, pattern
 [, startposition [, occurrence [,
return_option
 [, match_parameter] [,subexpr]]]])
```

Subexpressions and Alternate Matches

A subexpression is a part of a regular expression and enclosed with a set of parentheses. A subexpression can be repeated a certain number of times. The regular expression ‘ba(na)*split’ allows 0 or more repetitions of the subexpression ‘na’, for matches such as ‘basplit’, ‘banasplit’, ‘bananasplit’, and ‘banananasplit’.

Parentheses are also used for alternation, with the vertical bar | symbol separating the alternates. The regular expression ‘t(a|e|i)n’ allows three possible choices between the letters *t* and *n*. Valid results include words such as ‘tan’, ‘ten’, ‘tin’, and ‘Pakistan’, but not ‘teen’, ‘mountain’, or ‘tune’. Alternatively, a character list such as ‘t[aei]n’ yields the identical result. Table 16.7 describes the use of these metacharacters.

TABLE 16.7 Alternate Matching and Grouping of Expressions

METACHARACTER	DESCRIPTION
|	Alternation. Separates alternates; usually used with the grouping operator ( ).
( )	Group. Group subexpressions into a unit for alternations, for quantifiers, or for backreferencing (see the section,“Backreferences,” later in this chapter).

The REGEXP_REPLACE Function

You learned about the REPLACE function in Chapter 4, “Character, Number, and Miscellaneous Functions.” It substitutes one string with another string. Assume that your data has extraneous spaces in the text, and you would like to replace them with a single space. If you use the REPLACE function, you have to list exactly how many spaces you want to replace. However, the number of extraneous spaces may not be the same everywhere in the text.

The following example has three spaces between 'Joe' and 'Smith'. The function's parameter specifies that two spaces are replaced with one space. In this case, the result leaves an extra space between 'Joe' and 'Smith'.

```
SELECT REPLACE('Joe   Smith','  ', ' ')
       AS replace
  FROM dual
REPLACE
----------
Joe  Smith

1 row selected.
```

The REGEXP_REPLACE function takes the substitution a step further: It replaces the matching pattern with a specified regular expression, allowing for complex search and replace operations. The following query replaces any two or more spaces, '{2,}', with a single space. The ( ) subexpression contains a single space, which can be repeated two or more times. As you see from the result, only one space exists between the 'Joe' and 'Smith'.

```
SELECT REGEXP_REPLACE('Joe   Smith',
       '( ){2,}', ' ')
       AS RX_REPLACE
  FROM dual
RX_REPLAC
---------
Joe Smith

1 row selected.
```

The syntax of the REGEXP_REPLACE function is as follows.

```
REGEXP_REPLACE(source_string,
pattern
 [, replace_string [, position
[,occurrence, [match_parameter]]]])
```

By default, the start position is 1, and the OCCURRENCE parameter defaults to 0, which indicates that all matches are replaced.

The REGEXP_COUNT Function

Oracle 11*g* introduced the REGEXP_COUNT function, which allows you to determine how many times a pattern occurs within a given string. In the following example, the REGEXP_COUNT function determines that the pattern ‘el’ appears four times in the sentence.

```
SELECT REGEXP_COUNT(
  'The shells she sells are surely seashells.',
  'el', 1, 'i') REGEXP_COUNT
  FROM dual
REGEXP_COUNT
---------------
4

1 rows selected
```

Exploring the Match Parameter Option

The REGEXP_LIKE operator and all regular expression functions contain an optional match parameter. It allows matching for case, ignoring newlines, and matching across multiple lines.

CASE-SENSITIVE MATCHES

The following example shows how to ignore the case. The ‘i’ value in the match parameter performs a case-insensitive search, and the ‘c’ parameter makes it case-sensitive (the default). The query searches all student rows where the first name matches the pattern ‘ta’, regardless of case. The result includes the name ‘Tamara’ in the result.

```
SELECT first_name
  FROM student
 WHERE REGEXP_LIKE(first_name, 'ta', 'i')
FIRST_NAME
-------------------------
Julita
Tamara
Benita
Rita
Sengita

5 rows selected.
```

MATCHING A PATTERN THAT CROSSES MULTIPLE LINES

Recall that the match-any character (.) matches all characters except the newline character. Sometimes you might need to search for a pattern that stretches across 704
multiple lines. The ‘n’ match parameter allows you to 705
include the newline character as part of the match-any character. The following SQL statement shows a three-line source string; the desired substring of the pattern is ‘cat.*dog’.

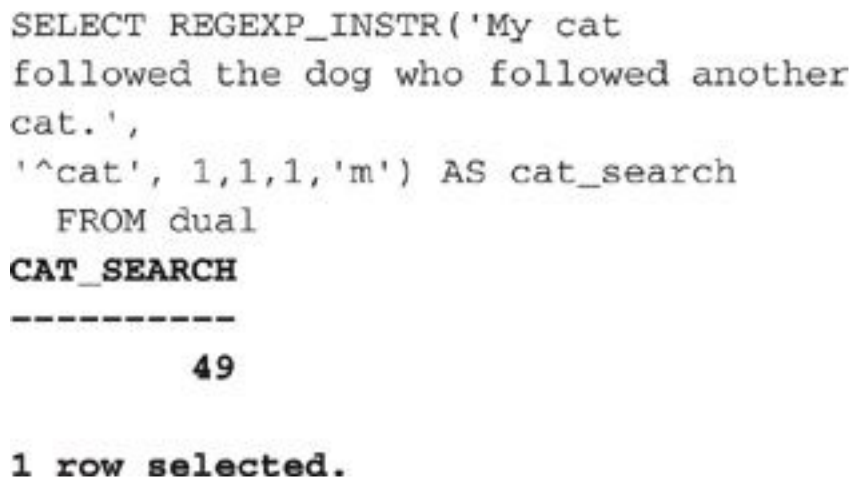
```
SELECT REGEXP_SUBSTR('My cat could have
followed the dog almost
immediately.', 'cat.*dog', 1, 1, 'n')
  FROM dual
REGEXP_SUBSTR('MYCAT
--------------------
cat could have
followed the dog

1 row selected.
```

The displayed output shows the substring that contains this pattern. The REGEXP_SUBSTR function lists the starting position of the search as 1, followed by the first occurrence, and then the ‘n’ match parameter option. If the ‘n’ option is omitted, the substring containing the pattern is not displayed.

TREATING A STRING AS A MULTILINE SOURCE

The multiline mode, ‘m’, effectively retains the source string as multiple logical lines and therefore allows the matching of the start- and end-of-line metacharacters. The next example shows a three-line string and determines the position of the pattern ‘^cat’. As indicated with the ^ metacharacter, the desired pattern is at the start of the line. The result shows that this pattern is found at position 49 of the string.



```
SELECT REGEXP_INSTR('My cat
followed the dog who followed another
cat.',
'^cat', 1,1,1,'m') AS cat_search
  FROM dual
CAT_SEARCH
----------
        49

1 row selected.
```

COMBINING MATCH PARAMETERS

You can combine match parameters. For example, ‘in’ makes the result case-insensitive and includes the newline character. However, an ‘ic’ parameter is contradictory and will default to case-sensitive matching. Table 16.8 contains an overview of the match parameter options.

TABLE 16.8 Match Parameter Choices

PARAMETER	DESCRIPTION
i	Match case-insensitive.
c	Match case-sensitive, the default.
n	A match for any character (.) in the pattern allows your search to include the newline character.
m	The source string is retained as multiple lines, and the anchoring metacharacters (^ and $) are respected as the start and end of each line.
x	Ignores all whitespace characters.

Backreferences

A useful feature of regular expression is the ability is to store subexpressions for later reuse; this is also referred to as *backreferencing*. This functionality allows sophisticated replace capabilities, such as swapping patterns in new positions or determining repeated word or letter occurrences. The matched part of the pattern is stored in a temporary buffer that is numbered from left to right and accessed with the '\digit' notation, whereby *digit* is a number between 1 through 9 and matches the digit-th subexpression, as indicated by a set of parentheses.

The following example shows the name ‘Ellen Hildi Smith’ transformed to ‘Smith, Ellen Hildi’.

```
SELECT REGEXP_REPLACE(
       'Ellen Hildi Smith',
       '(.*) (.*) (.*)', '\3, \1 \2')
  FROM dual
REGEXP_REPLACE('EL
------------------
Smith, Ellen Hildi

1 row selected.
```

The query lists three individual subexpressions enclosed by the parentheses. Each individual subexpression consists of a match-any metacharacter (.) followed by the * metacharacter, indicating that the match-any character must be matched 0 or more times. A space separates the subexpressions and must be matched as well. In this case, the parentheses effectively create subexpressions that capture the values and can be referenced with ‘\digit’. The first subexpression is assigned \1, the second \2, and so on. These stored values are then backreferenced in the function as ‘\3, \1 \2’, which effectively transform the string to the desired order and separates the third subexpression from the first with a comma.

Backreferences are also valuable for finding duplicate words, as in the following query, which looks for one or more alphanumeric characters followed by one or more spaces, followed by the same value found in the first subexpression. The result of the REGEXP_SUBSTR function shows the duplicated word *is*.

```
SELECT REGEXP_SUBSTR(
       'There is is a speed limit!',
       '([[:alnum:]]+)([[:space:]]+)\1') AS substr
  FROM dual
SUBST
-----
is is

1 row selected.
```

Word Boundaries

At times, you might want to match entire words, not just individual characters within a word or string. This is useful if you need to enclose certain words with HTML tags or simply replace whole words. For example, the regular expression ‘cat’ matches ‘cat’, ‘caterpillar’, or ‘location’. The following regular expression query replaces the word ‘cat’ in the string ‘The cat sat on the roof’ with the word ‘mouse’. If the input string is changed, to something like ‘location is everything’ or ‘caterpillar’, the replacement does not occur.

```
SELECT REGEXP_REPLACE('The cat sat on the roof',
       '(^|[^[:alpha:]])cat($|[^[:alpha:]])', ' mouse ')
  FROM dual
REGEXP_REPLACE('THECATSAT
------------------------
The mouse sat on the roof

1 row selected.
```

The pattern ‘cat’ starts with either a beginning-of-line character (^) or a non-alpha character ([^[:alpha:]]), and the two choices are separated by the alternation metacharacter (|). The non-alpha character is anything but a letter and includes punctuation, spaces, commas, and so on. Then the letters ‘cat’ can be followed by either the end–of-line character or another non-alpha character.

The Backslash Character

The backslash (\) has various meanings in a regular expression. You have already learned to apply it to backreference expressions; it can also be used as the escape character. For example, if you want to search for the * as a literal rather than use it as a metacharacter, you precede it with the backslash escape character. The expression then reads *, and the \ indicates that * is not the repetition operator but a literal *. Table 16.9

summarizes the escape and backreference metacharacters, and it provides some examples.

TABLE 16.9 The Escape and Backreference Metacharacters

METACHARACTER	DESCRIPTION	EXAMPLE
\char	The backslash indicates the escape character. The character following the escape character is matched as a literal rather than a metacharacter.	'abc*def' matches the string 'abc*def' because * is meant as the literal * rather than the repetition operator. When used within a character list, the literal \ does not need to be escaped. For example, the regular expression '[\abc]' matches the literal \. In this case, you do not need to escape the backslash.

\digit	The backslash with a digit between 1 and 9 matches the preceding digit-th checks for adjacent occurrences of parenthesized subexpression.	The regular expression '(abc)\1' the parenthesized subexpression 'abc'.

Many metacharacters do not need to be escaped when within a character list. For example, [.] indicates the literal period and does not requires a backslash escape character.

Applying Regular Expressions in Data Validation

Regular expressions are useful not only in queries but also for data validation. The following statement applies a column check constraint to the LAST_NAME column of the STUDENT table. This regular expression performs very basic validation.

```
ALTER TABLE student
  ADD CONSTRAINT stud_last_name_ck
CHECK
```

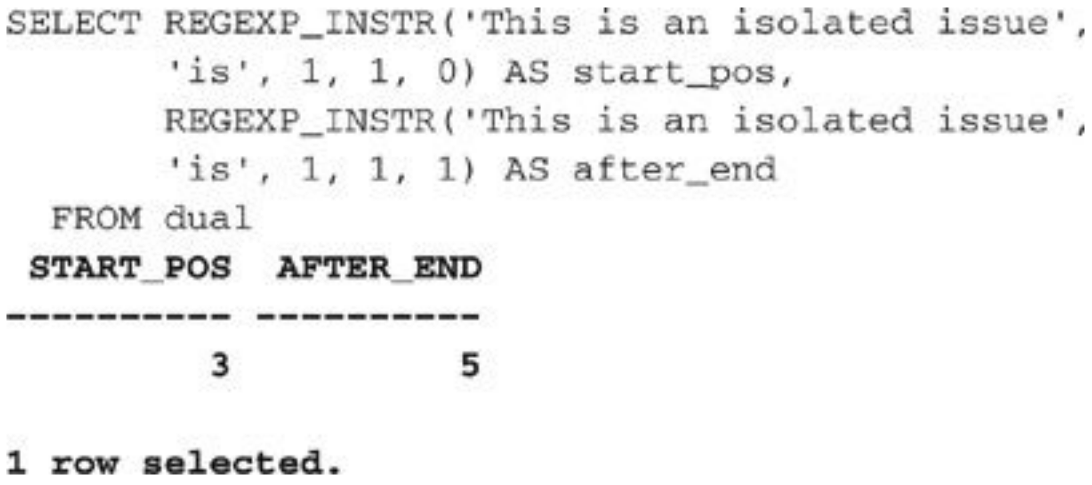
```
    (REGEXP_LIKE(last_name, '^[-
   [:alpha:] .,()'']*$'))
```

The only allowed characters are alphabetical characters (lower- or uppercase), spaces, hyphens, periods, commas, quotation marks, and parentheses. The brackets effectively create a character list encompassing these characters and the POSIX class [:alpha:]. The characters within the character list can appear in any order within the pattern. The hyphen (-) is the first character in the list and therefore indicates the literal hyphen. Names that pass the column validation include Miller-Johnson and Smith Woldo. There are two single quotation marks in the character list; they allow the single quotation mark character to appear. For example, a name such as O'Connor is a valid pattern. The * metacharacter follows the character list, thus allowing zero-to-many repetitions. The regular expression begins and ends with the anchoring characters ^ and $ to avoid any other characters before or after the pattern.

Understanding Matching Mechanics

When you are searching for patterns within text, you may come across instances in which the pattern can be found multiple times. The following example illustrates this scenario. The letters 'is' occur multiple times in the string ‘This is an isolated issue’. REGEXP_INSTR returns the first occurrence of the pattern in the first position of the string (the default), which displays the starting position of the pattern as 3 and the character following the end of the pattern as 5.

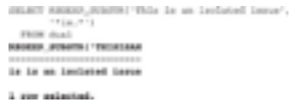
You can specify any subsequent occurrence with the appropriate occurrence parameter. If you need to find all occurrences, you might want to consider writing a small PL/SQL program to perform a loop to retrieve them.



```
SELECT REGEXP_INSTR('This is an isolated issue',
       'is', 1, 1, 0) AS start_pos,
       REGEXP_INSTR('This is an isolated issue',
       'is', 1, 1, 1) AS after_end
  FROM dual
 START_POS  AFTER_END
---------- ----------
         3          5

1 row selected.
```

The following example shows that when quantifiers are involved, Oracle’s regular expressions are *greedy*. This means the regular expression engine tries to find the longest possible match. This pattern begins with an optional space, followed by the letters ‘is’ and then optional characters.



```
SELECT REGEXP_SUBSTR('This is an isolated issue',
       ' ?is.*')
  FROM dual
REGEXP_SUBSTR('THISISAN
-----------------------
is is an isolated issue

1 row selected.
```

You can control the greediness of the expression with the ?’ metacharacter. In Table 16.3, you learned about the various quantifier operators. Adding the ?’ to the respective operator character makes a match non-greedy. The result now finds the shortest possible match.

```
SELECT REGEXP_SUBSTR('This is an isolated issue',
       '*is.*?')
  FROM dual
REGEXP_SUBSTR('THISISAN
-----------------------
is

1 row selected.
```

As you have noticed by now, the same metacharacters are used for different purposes. For example, the? metacharacter can make a search non-greedy, and it can also be used as a quantifier operator. The ^ character is the start-of-line indicator, and it is also used for negating a list. Carefully read the metacharacters within the context of the regular expression to ensure your correct understanding.

Comparing Regular Expressions to Existing Functionality

Regular expressions have several advantages over the familiar LIKE operator and INSTR, SUBSTR, and REPLACE functions. These traditional SQL functions have no facility for matching patterns. Only the LIKE operator performs matching of characters, through the use of the % and _ wildcards, but LIKE does not support repetitions of expressions, complex alternations, ranges of characters, character lists, POSIX character classes, and so on. Furthermore, the new regular expression functions allow detection of duplicate word occurrences and swapping of patterns. Table 16.10 contrasts and highlights the capabilities of regular expressions versus the traditional SQL operators and functions.

TABLE 16.10 Regular Expression Pattern Matching Versus Existing Functionality

REGEXP	LIKE AND SQL FUNCTIONS
Complex pattern matching with repetitions, character classes, negation, alternations, and so on.	Simple pattern matching for LIKE operator with % and _ indicating single or multiple characters, but does not support character classes, ranges, and repetitions. The INSTR, SUBSTR, and REPLACE functions do not have any pattern-matching capabilities.
Backreference capabilities allow sophisticated replace functionality.	Very basic replace functionality.
Not supported in Oracle versions prior to 10*g*.	All Oracle versions.
Choices of expression patterns are easily formulated with the alternation become very complex.	Alternations must be formulated with OR conditions, which can easily operator.
Sensitive to language, territory, sort order, and character set.	No support for various locales unless specifically coded within the criteria of the query.

Regular expressions are very powerful because they help solve complex problems. Some of the functionality in regular expressions is very difficult to duplicate using traditional SQL functions. When you learn the basic building blocks of this somewhat cryptic language, you will see that regular expressions become an indispensable part of your toolkit not only in the context of SQL but also with other programming languages. While trial and error are often necessary to get your individual pattern right, the elegance and power of the regular expressions are indisputable. 710

711

Regular Expression Resources

This lab illustrates a number of regular expression patterns; you may find yourself trying to come up with your own pattern to validate an e-mail address, a URL, or a credit card number. Writing a regular expression that covers all the different pattern possibilities is not a trivial task. The Web is a good starting point for finding commonly used patterns, and it helps to review similar patterns as a starting point for your individual validation requirements.

LAB 16.1 EXERCISES

a) Write a regular expression column constraint against the FIRST_NAME column of the STUDENT table to ensure that the first name starts with an uppercase character. The subsequent characters allowed are alphabetical letters, spaces, hyphens, quotation marks, and periods.

b) Describe the difference between the following two regular expressions and the corresponding result.

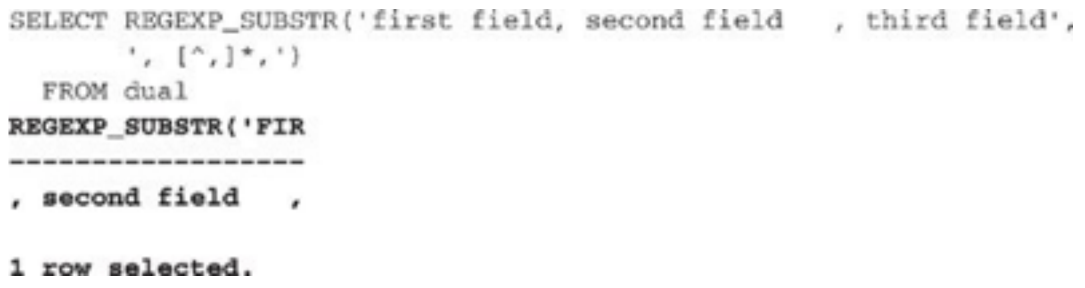
```
SELECT zip,
       REGEXP_INSTR(zip, '[[:digit:]]{5}') exp1,
       REGEXP_INSTR(zip, '[[:digit:]{5}]') exp2
  FROM zipcode_example
ZIP         EXP1       EXP2
----- ---------- ----------
ab123          0          3
007ab          0          1
123xy          0          1
abcxy          0          0
10025          1          1

5 rows selected.
```

c) The following SQL statement creates a table called DOC_LOCATION and adds a regular expression column check constraint to the FILE_NAME column. List examples of different file names that will pass the validation.

```
CREATE TABLE doc_location
 (doc_id NUMBER,
  file_name VARCHAR2(200)
CONSTRAINT doc_loc_file_name_ck
  CHECK (REGEXP_LIKE(file_name,
   '^([a-zA-Z]:|[\])[\]([^\]+
[\])*[^?*;”<>|\/]+\.[a-zA-Z]
{1,3}$')))
```

d) Explain the regular expression metacharacters used in the following SQL statement and their effect on the resulting output.



```
SELECT REGEXP_SUBSTR('first field, second field   , third field',
       ', [^,]*,')
  FROM dual
REGEXP_SUBSTR('FIR
------------------
, second field   ,

1 row selected.
```

e) Explain what the following statement accomplishes.

```
CREATE TABLE zipcode_regexp_test
  (zip VARCHAR2(5) CONSTRAINT
zipcode_example_ck
   CHECK(REGEXP_LIKE(zip,
    '[[:digit:]]{5}(-[[:digit:]]
{4})?$')))
```

f) Describe the result of the following query.

```
SELECT REGEXP_INSTR('Hello',
'x?'),
  REGEXP_INSTR('Hello', 'xy?')
FROM dual
```

g) Describe the individual components of the following regular expression check constraint.

```
ALTER TABLE instructor
ADD CONSTRAINT inst_phone_ck
CHECK
  (REGEXP_LIKE(phone,
'^(\([[:digit:]]{3}\)|[[:digit:]]
{3})[- ]?[[:digit:]]{3}[- ]?
[[:digit:]]{4}$'))
```

LAB 16.1 EXERCISE ANSWERS

a) Write a regular expression column constraint against the FIRST_NAME column of the STUDENT table to ensure that the first name starts with an uppercase character. The subsequent characters allowed are alphabetical letters, spaces, hyphens, quotation marks, and periods.

ANSWER: The individual components of the regular expression are listed in Table 16.11.

```
ALTER TABLE student
  ADD CONSTRAINT
stud_first_name_ck CHECK
  (REGEXP_LIKE(first_name,
'^[[:upper:]]{1}[-[:alpha:] .'']*
$'))
```

TABLE 16.11 First Name Regular Expression Example

METACHARACTER	DESCRIPTION
^	Start-of-line metacharacter; anchors the pattern to the beginning of the line and therefore does not permit leading characters before the pattern.
[	Start of class list.
[:upper:]	Uppercase alphabetic POSIX character class.
]	End of class list.
{1}	Exactly one repetition of the uppercase alphabetical character class list.
[	Start of another character list.
-	A hyphen; this does not indicate range because it is at the beginning of the character list.
[:alpha:]	POSIX alphabetical character class.
	Blank space.
.	The period, not the match-any character.

''	Two individual quotes, indicating a single quote.
]	End of the second character list.
*	Zero to many repetitions of the character list.
$	The end-of-line metacharacter; anchors the pattern to the end of the line and therefore does not permit any other characters following the pattern.

This solution looks for allowable characters. You can approach regular expression validation by defining which characters to exclude, with the ^negation character at the beginning of the character list, or by using the NOT REGEXP_LIKE operator. As always, there are a number of ways to solve the problem, depending on the individual requirements and circumstances. Careful testing for various ranges of values and scenarios ensures that the regular expression satisfies the desired validation rules.

b) Describe the difference between the following two regular expressions and the corresponding result.

```
SELECT zip,
       REGEXP_INSTR(zip, '[[:digit:]]{5}') exp1,
       REGEXP_INSTR(zip, '[[:digit:]{5}]') exp2
  FROM zipcode_example
ZIP         EXP1       EXP2
----- ---------- ----------
ab123          0          3
007ab          0          1
123xy          0          1
abcxy          0          0
10025          1          1

5 rows selected.
```

ANSWER: The difference between the two regular expressions is the location of the repetition operator. The first regular expression requires exactly five occurrences of the POSIX digit class. The result shows the starting position of those rows that match the pattern. A row such as 'ab123' does not have five consecutive numbers; therefore, REGEXP_INSTR returns 0.

In the second regular expression, the position of the repetition operator '{5}' was purposely misplaced within the character list, and as a result, the regular expressions requires the occurrence of either a digit or the opening and closing braces ({ }). Therefore, the zip value 'ab123' fulfils this requirement at starting position 3.

c) The following SQL statement creates a table called DOC_LOCATION and adds a regular expression column check constraint to the FILE_NAME column. List examples of different file names that will pass the validation.

```
CREATE TABLE doc_location
 (doc_id NUMBER,
  file_name VARCHAR2(200)
CONSTRAINT doc_loc_file_name_ck
  CHECK (REGEXP_LIKE(file_name,
  '^([a-zA-Z]:|[\])[\]([^\]+
[\])*[^?*;”<>|\/]+\.[a-zA-Z]
{1,3}$')))
```

ANSWER: Valid file names include c:\filename.txt, c:\mydir\filename.d, c:\myfile\mydir\filename.sql, and \\myserver\mydir\filename.doc.

The regular expression checks for these valid Windows file name and directory conventions. The pattern begins with either a drive letter followed by a colon and a backslash or with a double backslash, which indicates the server name. This is possibly followed by subdirectory names. The subsequent file name ends with a one-, two-, or three-letter extension. Table 16.12 shows the individual components of the regular expression broken down by drive/machine name, directory, file name, and extension.

TABLE 16.12 File Name Validation

METACHARACTER(S)	DESCRIPTION
^	No leading characters are permitted prior to the start of the pattern.
([a-zA-Z]:|[\])	This subexpression allows either a drive letter followed by a colon or a machine name, as indicated with the backslash. (Machine names start with two backslashes, as you'll see later.) The choices are separated by the | alternation operator. The backslash character is enclosed within the character list because it is not meant as the escape character or as a backreference. Valid patterns can start with c: or \.
[\]	The backslash is again a literal backslash; a valid start of the pattern may now look like c:\ or \\.

([^\]+[\])*	The next subexpression allows 0, 1, or multiple repetitions. This subexpression builds the machine name and/or the directory name(s). It starts with one or many characters, as indicated by the + quantities, but the first character cannot be a backslash character. It is ended by a backslash. Effectively, a valid pattern so far can read as c:\, c:\mydir\, c:\mydir\mydir2\, \\myserver\, or \\myserver\mydir\.
[^?*;"<>|\/]+	This part of the regular expression validates the file name. A file name can consist of one or more characters, hence the +, but may not contain any of the characters listed in the character lists, as indicated with the ^ negation character. The start of a valid pattern may be c:\filename, c:\mydir\filename, c:\mydir\mydir2\filename, \\myserver\filename, or \\myserver\mydir\filename.

\.[a-zA-Z]{1,3}	The file name is followed by a period. Here the period must be escaped; otherwise, it indicates the match-any character. The period is followed by an alphabetical one-, two-, or three-letter extension.
$	The end-of-line metacharacter ends the regular expression and ensures that no other characters are permitted.

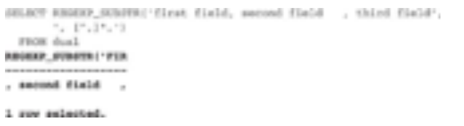
Compared to the other regular expressions you have seen, this regular expression is fairly long. If you try out the statement, be sure the regular expression fits on one line. Otherwise, the end-of-line character is part of the regular expression, and you will not get the desired result.

The regular expression chosen in this case permits valid Windows file names but does not include all allowable variations.

714

715

d) Explain the regular expression metacharacters used in the following SQL statement and their effect on the resulting output.



```
SELECT REGEXP_SUBSTR('first field, second field   , third field',
       ', [^,]*,')
  FROM dual
REGEXP_SUBSTR('FIR
------------------
, second field   ,

1 row selected.
```

ANSWER: The REGEXP_SUBSTR function extracts part of a string that matches the pattern ', [^,]*,’. The function looks for a comma followed by a space, then zero or more characters that are not commas, and then another comma.

As you see from this example, you can use regular expressions to extract values from a comma-separated string. The occurrence parameter of the function lets you pick the appropriate values. The pattern must be modified if you look for the first or last value in the string.

e) Explain what the following statement accomplishes.

```
CREATE TABLE zipcode_regexp_test
 (zip VARCHAR2(5) CONSTRAINT
zipcode_example_ck
  CHECK(REGEXP_LIKE(zip,
  '[[:digit:]]{5}(-[[:digit:]]
{4})?$')))
```

ANSWER: The statement creates a table named ZIPCODE_REGEXP_TEST with a column called ZIP. The constraint checks whether the entered value in the ZIP column is either in a 5-digit zip code or the 5-digit + 4 zip code format.

In this example, the parenthesized subexpression (-[[:digit:]]{4}) is repeated zero or one times, as indicated by the ? repetition operator. The various components of this regular expression example are explained in Table 16.13.

TABLE 16.13 Explanation of 5-digit + 4 Zip Code Expression

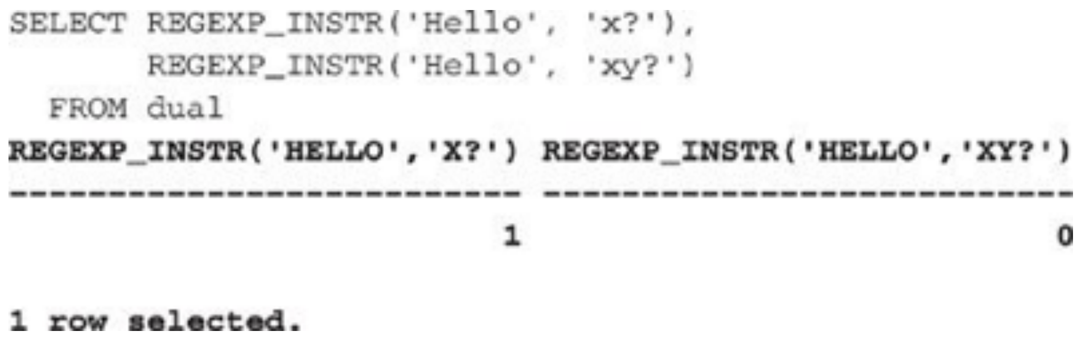
METACHARACTER(S)	DESCRIPTION
^	Start-of-line anchoring metacharacter.
[	Start of character list.
[:digit:]	POSIX numeric digit class.
]	End of character list.
{5}	Repeat exactly five occurrences of the character list.
(	Start of subexpression.
-	A literal hyphen, because it is not a range metacharacter inside a character list.
[	Start of character list.
[:digit:]	POSIX [:digit:] class.
]	End of character list.
{4}	Repeat exactly four occurrences of the character list.
)	Closing parenthesis, to end the subexpression.

?	The ? quantifier matches the grouped subexpression 0 or 1 time thus making the 4-digit code optional.
$	Anchoring metacharacter, to indicate the end of the line.

f) Describe the result of the following query.

```
SELECT REGEXP_INSTR('Hello',
'x?'),
  REGEXP_INSTR('Hello', 'xy?')
FROM dual
```

Answer: The REGEXP_INSTR function returns a starting position of 1, even though the pattern 'x?' cannot be found anywhere in the string 'Hello'. The second function call looks for the pattern 'xy?', and the function returns a 0 because the required letter *x* followed by an optional letter *y* doesn't exist in the source string.



```
SELECT REGEXP_INSTR('Hello', 'x?'),
       REGEXP_INSTR('Hello', 'xy?')
  FROM dual
REGEXP_INSTR('HELLO','X?') REGEXP_INSTR('HELLO','XY?')
-------------------------- ---------------------------
                         1                           0

1 row selected.
```

The first function returns the result 1, which indicates that the pattern exists at the first position of the source string. However, 'x?’ is optional, so it can match an empty string, and it does so in the example.

Passing the RETURN_OPTION parameter value 0 (beginning of the string) and 1 (the character after the end of the string) returns again the same result, indicating that it matches an empty string.

```
SELECT REGEXP_INSTR('Hello', 'x?',1,1,0)
         AS start_pos,
         REGEXP_INSTR('Hello', 'x?',1,1,1)
         AS after_end
   FROM dual
 START_POS  AFTER_END
---------- ----------
         1          1

1 row selected.
```

The next query matches an empty string, much like the previous 'x?' pattern, because the set of parentheses around the enclosed 'xy' makes both letters optional.

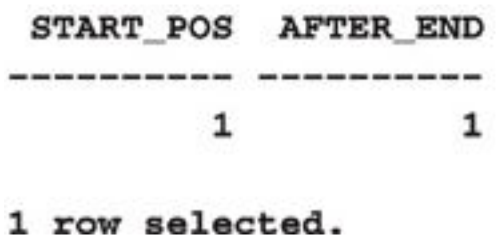
```
SELECT REGEXP_INSTR('Hello',
'(xy)?', 1, 1, 0)
```

```
  AS start_pos,
  REGEXP_INSTR('Hello', '(xy)?',
1, 1, 1)
```

716

717

```
AS after_end
FROM dual
```



```
 START_POS  AFTER_END
---------- ----------
         1          1

1 row selected.
```

g) Describe the individual components of the following regular expression check constraint. ALTER TABLE instructor

```
  ADD CONSTRAINT inst_phone_ck
CHECK
   (REGEXP_LIKE(phone,
'^(\([[:digit:]]{3}\)|[[:digit:]]
{3})[- ]?[[:digit:]]{3}[- ]?
[[:digit:]]{4}$'))
```

ANSWER: The check constraint validates the entries in the PHONE column of the INSTRUCTOR table. The phone number must follow a specific pattern, such as a ###-#######, ### #######, (###) ###-####, and so on. The individual components are listed in Table 16.14.

TABLE 16.14 A Simple Phone Number Regular Expression Example

METACHARACTER(S)	DESCRIPTION
^	Start-of-line metacharacter, which doesn't permit leading characters before the regular expression.
(	Start of subexpression, which allows two choices for area code validation: Either a three-digit number enclosed by parentheses or a three-digit number without parentheses.
\(	The backslash escape character, which indicates that the following open parenthesis character represents a literal rather than a metacharacter.
[[:digit:]]{3}	An expression indicating that the following numbers of the area code can be any three digits.

\)	Escape character, which indicates that the following character is not a metacharacter; in this case it is the closing parentheses for the area code.
|	The alternation metacharacter, which specifies the end of the first choice, which is the area code enclosed by parentheses, and the start of the next area code choice, which does not require that the area code be enclosed by parentheses.
[[:digit:]]{3}	Three required digits.
)	End of the subexpression alternation.
[-]?	An optional character consisting either of a hyphen or a space following the area code.
[[:digit:]]{3}	The first three digits of the phone number following the area code.
[-]?	The first three digits of the phone number are separated by either an optional hyphen or space.

[[:digit:]]{4}	The last four digits of the phone number.
$	End-of-line metacharacter, which ends the regular expression and ensures that no other characters are permitted.

You can expand this phone number example to include different formatting separators, such as optional extension formats or even the entry of letters, such as 800-DON-OTCAL. Perhaps you want to validate that the first digit of an area code or a phone number may not begin with a 0 or 1. As you can see, validating a seemingly simple phone number can involve complex alternations, logic, and an arsenal of metacharacters. If you want to include foreign phone numbers in your validation, then the regular expression becomes even more involved.

You might consider simplifying the validation altogether by allowing only 10 numbers in the phone column. The appropriate display of the data to the desired format can then be accomplished through a view or a SELECT statement.

Another way to approach phone number validation is to separate the entire phone number into different components, such as the area code, the exchange, and the remaining four digits.

Lab 16.1 Quiz

In order to test your progress, you should be able to answer the following questions.

1) The following query is valid.

```
SELECT REGEXP_LIKE('10025',
'[[:digit:]]')
 FROM dual
```

_______a) True

______b) False

2) Choose all the valid values for the regular expressions 'hat{4,1}’.

______a) hat

______b) haaat

______c) hatttt

______d) hathathat

______e) It is an invalid regular expression.

3) Based on the following query, which value will be shown in the resulting output?

```
SELECT REGEXP_REPLACE('ABC10025',
'[^[:digit:]]{1,5}$', '@')
FROM dual
```

_____a) @@@@@

_____b) ABC10025

_____c) @@@10025

_____d) ABC@@@@@

_____e) @10025

_____f) It is an invalid regular expression.

4) The following two regular expressions are equivalent.

```
([[:space:]]|[[:punct:]])+
[[:space:][:punct:]]+
```

_____a) True

_____b) False

5) The following query is invalid.

```
SELECT REGEXP_SUBSTR('ABC10025',
'[[:ALPHA:]]')
```

```
FROM dual
```

_____a) True

_____b) False

6) Based on the following regular expression, the value 'CD' will be returned.

```
SELECT REGEXP_SUBSTR('abCDefgH',
'[[:upper:]]+')
 FROM dual
```

_____a) True

_____b) False

7) The following two regular expressions are equivalent.

```
^[[:digit:]]{5}$
^[0-9]{5}$
```

_____a) True

_____b) False

ANSWERS APPEAR IN APPENDIX A.

LAB 16.2 Hierarchical Queries

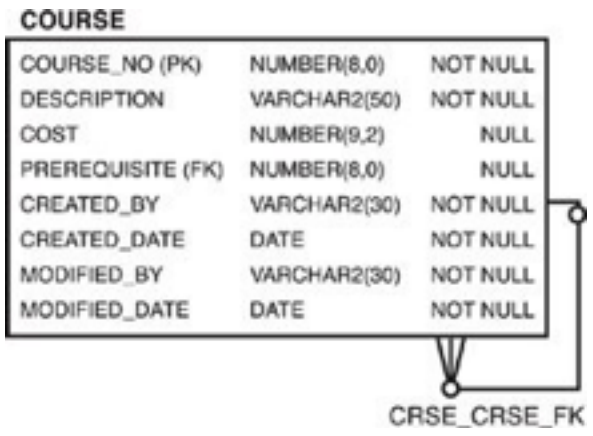
LAB OBJECTIVES

After this lab, you will be able to:

- ▶ Restrict the Result Set in Hierarchical Queries
- ▶ Move Up and Down the Hierarchy Tree

A *recursive relationship*, also called a *self-referencing relationship*, exists on the COURSE table in the STUDENT schema (see Figure 16.1). This recursive relationship is between the columns COURSE_NO and PREREQUISITE. It is just like any other parent–child table relationship, except the relationship is with itself.

FIGURE 16.1 The self-referencing relationship of the COURSE table



COURSE

COURSE_NO (PK)	NUMBER(8,0)	NOT NULL
DESCRIPTION	VARCHAR2(50)	NOT NULL
COST	NUMBER(9,2)	NULL
PREREQUISITE (FK)	NUMBER(8,0)	NULL
CREATED_BY	VARCHAR2(30)	NOT NULL
CREATED_DATE	DATE	NOT NULL
MODIFIED_BY	VARCHAR2(30)	NOT NULL
MODIFIED_DATE	DATE	NOT NULL

The PREREQUISITE column is a foreign key that references its own table's primary key. Only valid course numbers can be entered as prerequisites. Any attempt to insert or update the PREREQUISITE column to a value for which no COURSE_NO exists is rejected. A course can have zero or one prerequisite. For a course to be considered a prerequisite, it must appear at least once in the PREREQUISITE column.

This relationship between the parent and child can be depicted in a query result as a hierarchy or tree, using Oracle's CONNECT BY clause and the PRIOR operator. The following result visually displays the relationship of the courses that have the course number 310, Operating Systems, as their prerequisite.

```
310 Operating Systems
  130 Intro to Unix
   132 Basics of Unix Admin
    134 Advanced Unix Admin
     135 Unix Tips and Techniques
  330 Network Administration
145 Internet Protocols
```

Reading from the outside in, the student first needs to take Operating Systems and then decide on either Intro to Unix or Internet Protocols. If the student completes the Intro to Unix course, he or she may choose between the Basics of Unix Admin class and the Network Administration class. If the student completes the Basics of Unix Admin, he or she may enroll in Advanced Unix Admin. After completion of this course, the student may enroll in Unix Tips and Techniques.

You can also travel the hierarchy in the reverse direction. If a student wants to take course number 134, Advanced Unix Administration, you can determine the required prerequisite courses until you reach the first course required.

In the business world, you may often encounter hierarchical relationships, such as the relationship between a manager and employees. Every employee may have at most one manager (parent), and to be a manager (parent), one must manage one or multiple employees (children). The root of the tree is the company's president; the president does not have a parent and, therefore, shows a NULL value in the parent column.

The CONNECT BY Clause and the PRIOR Operator

To accomplish the hierarchical display, you need to construct a query with the CONNECT BY clause and the

PRIOR operator. You identify the relationship between the parent and the child by placing the PRIOR operator before the parent column. To find the children of a parent, Oracle evaluates the expression qualified by the PRIOR operator for the parent row. Rows for which the condition is true are the children of the parent. With the following CONNECT BY clause, you can see the order of courses and the sequence in which they need to be taken.

```
CONNECT BY PRIOR course_no =
prerequisite
```

The COURSE_NO column is the parent, and the PREREQUISITE column is the child. The PRIOR operator is placed in front of the parent column COURSE_NO. Depending on which column you prefix with the PRIOR operator, you can change the direction of the hierarchy.

The CONNECT BY condition can contain additional conditions to filter the rows and eliminate branches from the hierarchy tree.

The START WITH Clause

The START WITH clause determines the root rows of the hierarchy. The records for which the START WITH clause is true are first selected. All children are retrieved from these records going forward. Without this clause, Oracle uses all rows in the table as root rows.

The following query selects the parent course number 310, its child rows, and, for each child, its respective descendents. The LPAD function, together with the LEVEL pseudocolumn, accomplishes the indentation.

```
SELECT LPAD(' ', 3*(LEVEL-1)) ||course_no
       || '  ' ||description
  FROM course
 START WITH course_no = 310
CONNECT BY PRIOR course_no = prerequisite
LPAD('',3*(LEVEL-1))||COURSE_NO||''||DESCRIPTION
------------------------------------------------
310  Operating Systems
   130  Intro to Unix
      132  Basics of Unix Admin
         134  Advanced Unix Admin
            135  Unix Tips and Techniques
      330  Network Administration
   145  Internet Protocols

7 rows selected.
```

Following is the syntax of the CONNECT BY clause.

```
[START WITH condition]
CONNECT BY [NOCYCLE] condition
```

The optional NOCYCLE parameter allows the query to continue even if a loop exists in the hierarchy. You will see some examples of loops later, in the exercises for this lab.

Understanding LEVEL and LPAD

The pseudocolumn LEVEL returns the number 1 for the root of the hierarchy, 2 for the child, 3 for the grandchild, and so on. The LPAD function allows you to visualize the hierarchy by indenting it with spaces. The length of the padded characters is calculated with the LEVEL function.

In Chapter 10, “Complex Joins,” you learned about self-joins. You might wonder how they compare to hierarchical queries. The fundamental difference is that the CONNECT BY clause allows you to visually display the hierarchy and the relationships to other rows.

Hierarchy Path

You can show the path of a value from the root to the last node of the branch for any of the rows by using the SYS_CONNECT_BY_PATH function. The following query example displays the hierarchy path, and the course numbers are separated by forward slashes.

```
SELECT LPAD(' ', 1*(LEVEL-1))
       ||SYS_CONNECT_BY_PATH(course_no, '/') AS "Path" ,
       description
  FROM course
 START WITH course_no = 310
CONNECT BY PRIOR course_no = prerequisite
Path                         DESCRIPTION
------------------------- -------------------------
/310                         Operating Systems
 /310/130                    Intro to Unix
  /310/130/132               Basics of Unix Admin
   /310/130/132/134          Advanced Unix Admin
    /310/130/132/134/135     Unix Tips and Techniques
  /310/130/330               Network Administration
 /310/145                    Internet Protocols

7 rows selected.
```

The SYS_CONNECT_BY_PATH function, which is valid only for a hierarchical query, has the following syntax.

```
SYS_CONNECT_BY_PATH (column, char)
```

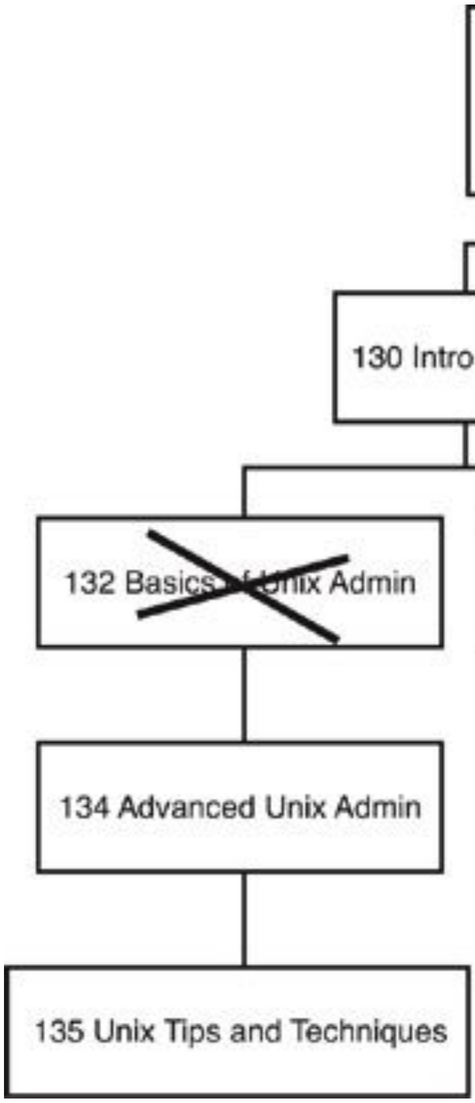
Pruning the Hierarchy Tree

A hierarchy can be described as a tree; if you want to remove specific rows from the result, you can use either the WHERE clause to eliminate individual rows or the CONNECT BY clause to eliminate branches.

Figure 16.2 graphically depicts the effect of the WHERE clause on the rest of the hierarchy. The WHERE clause effectively eliminates individual rows from the hierarchy. 724

725

FIGURE 16.2 Using the WHERE clause to eliminate rows



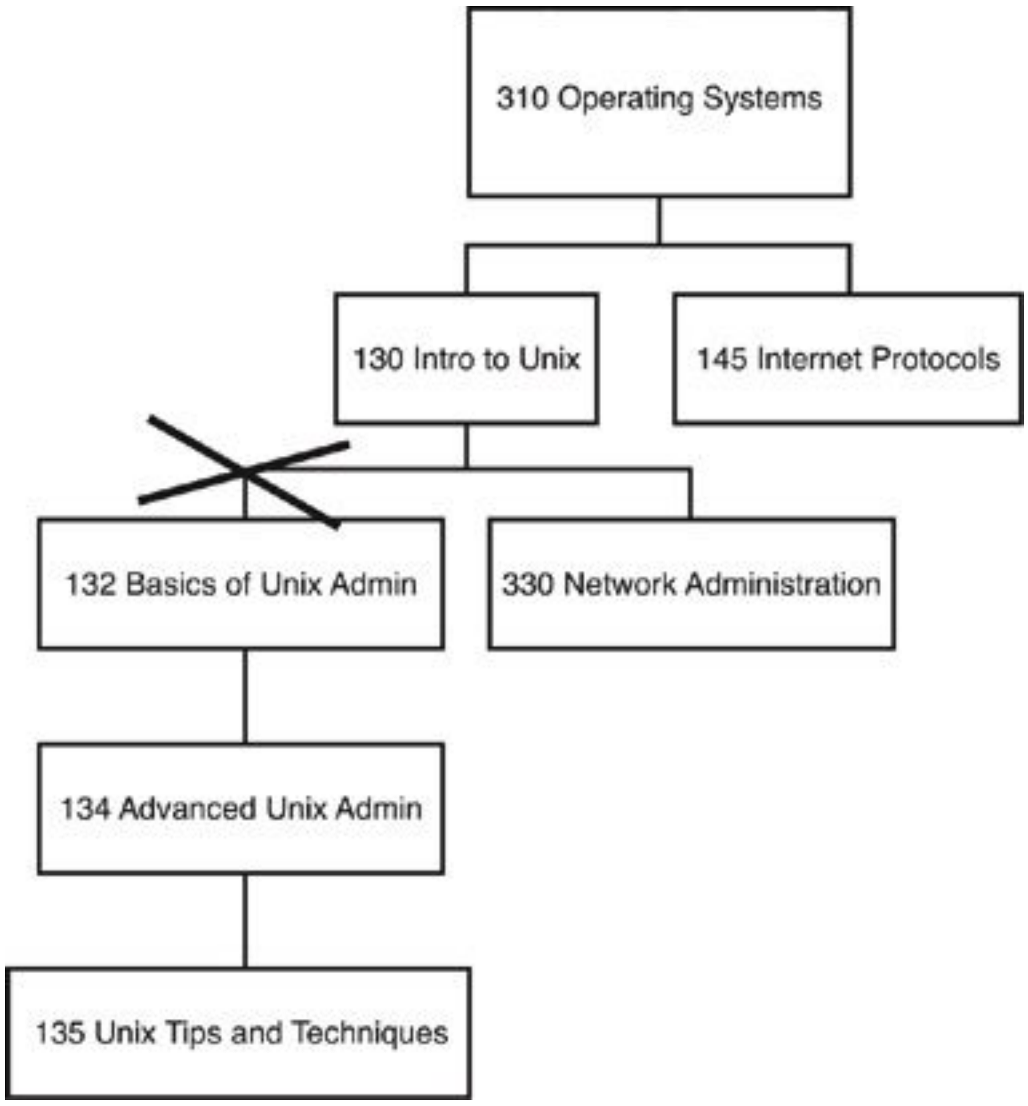
Only the rows that satisfy the condition of the WHERE clause are included in the result. The following SQL statement shows the WHERE clause that eliminates the specific row. Notice that the child rows of the eliminated course are listed.

```
SELECT LPAD(' ', 3*(LEVEL-1)) ||course_no
       || '  ' ||description AS hierarchy
  FROM course
 WHERE course_no <> 132
 START WITH course_no = 310
CONNECT BY PRIOR course_no = prerequisite
HIERARCHY
-------------------------------------------
310  Operating Systems
   130  Intro to Unix
         134  Advanced Unix Admin
            135  Unix Tips and Techniques
      330  Network Administration
   145  Internet Protocols

6 rows selected.
```

Figure 16.3 displays the scenario when the condition is moved to the CONNECT BY clause, causing the removal of a branch of the tree.

FIGURE 16.3 Using the CONNECT BY clause to eliminate an entire branch



The condition is part of the CONNECT BY clause, and when you examine the result, you find that COURSE_NO 132 and its respective descendants have been eliminated.

```
SELECT LPAD(' ', 3*(LEVEL-1)) ||course_no
       || '  ' ||description AS hierarchy
  FROM course
 START WITH course_no = 310
CONNECT BY PRIOR course_no = prerequisite
   AND course_no <> 132
HIERARCHY
-----------------------------------
310  Operating Systems
   130  Intro to Unix
      330  Network Administration
   145  Internet Protocols

4 rows selected.
```

Accessing Root Row Data with the CONNECT_BY_ROOT Operator

The CONNECT_BY_ROOT operator returns column data from the root row. The following SQL statement displays the course number of the root row in the column labeled ROOT. Note that the CONNECT_BY_ROOT operator is invalid in the START WITH and the CONNECT BY clauses.

```
SELECT description, course_no,
       CONNECT_BY_ROOT course_no AS root,
       LPAD(' ', 1*(LEVEL-1))
       ||SYS_CONNECT_BY_PATH(course_no, '/') AS "Path"
  FROM course
 START WITH course_no IN (310, 130)
CONNECT BY PRIOR course_no = prerequisite
DESCRIPTION                COURSE_NO ROOT Path
-------------------------- ---------- ---- -------------------------
Intro to Unix                     130  130 /130
Basics of Unix Admin              132  130  /130/132
Advanced Unix Admin               134  130   /130/132/134
Unix Tips and Techniques          135  130    /130/132/134/135
Network Administration            330  130  /130/330
Operating Systems                 310  310 /310
Intro to Unix                     130  310  /310/130
Basics of Unix Admin              132  310   /310/130/132
Advanced Unix Admin               134  310    /310/130/132/134
Unix Tips and Techniques          135  310     /310/130/132/134/135
Network Administration            330  310   /310/130/330
Internet Protocols                145  310  /310/145

12 rows selected.
```

The CONNECT_BY_ISLEAF Pseudocolumn

The CONNECT_BY_ISLEAF is a pseudocolumn that displays the value 1 if the row is the last child, also referred to as the leaf, of the hierarchy tree, as defined with the CONNECT BY clause. The output of the following query displays for course numbers 135, 330, and 145 the value 1 in the LEAF column; the others show zero as they are either root or branch nodes.

```
SELECT course_no, LPAD(' ', 1*(LEVEL-1))
       ||SYS_CONNECT_BY_PATH(course_no, '/') AS "Path",
       LEVEL, CONNECT_BY_ISLEAF AS leaf
  FROM course
 START WITH course_no = 310
CONNECT BY PRIOR course_no = prerequisite
 COURSE_NO Path                          LEVEL       LEAF
---------- ------------------------- ---------- ----------
       310 /310                               1          0
       130  /310/130                          2          0
       132   /310/130/132                     3          0
       134    /310/130/132/134                4          0
       135     /310/130/132/134/135           5          1
       330   /310/130/330                     3          1
       145  /310/145                          2          1

7 rows selected.
```

Joining Tables

Prior to Oracle9*i*, joins in hierarchical queries were not allowed. To achieve somewhat similar results, you had to write inline views or use custom-written PL/SQL functions to display any columns from related tables. The effect of a join in a hierarchical query is shown in the following example. The query joins the COURSE and SECTION tables and includes the SECTION_ID column in the result.

The join uses the common COURSE_NO column. The root rows are chosen via the START WITH clause. Here only those root rows with a COURSE_NO of 310 are selected as the root rows on which the hierarchy will be based. From the root row, the children, grandchildren, and any further descendants are determined.

You see a large number of rows because some courses have multiple sections. For example, COURSE_NO 132, Basics of Unix Admin, has two sections: SECTION_ID 139 and 138. For each section, the hierarchy is listed with the respective child sections. The individual child sections then show their child rows and so on.

```
SELECT LPAD(' ', 3*(LEVEL-1)) || c.course_no||' '||
       description AS hierarchy, s.section_id
  FROM course c, section s
 WHERE c.course_no = s.course_no
 START WITH c.course_no = 310
CONNECT BY PRIOR c.course_no = prerequisite
HIERARCHY                                SECTION_ID
---------------------------------------- ----------
310 Operating Systems                           103
   130 Intro to Unix                            107
      330 Network Administration                104
      132 Basics of Unix Admin                  139
         134 Advanced Unix Admin                110
            135 Unix Tips and Techniques        112
            135 Unix Tips and Techniques        115
            135 Unix Tips and Techniques        114
            135 Unix Tips and Techniques        113
         134 Advanced Unix Admin                111
            135 Unix Tips and Techniques        112
            135 Unix Tips and Techniques        115
            135 Unix Tips and Techniques        114
            135 Unix Tips and Techniques        113
         134 Advanced Unix Admin                140
            135 Unix Tips and Techniques        112
            135 Unix Tips and Techniques        115
            135 Unix Tips and Techniques        114
            135 Unix Tips and Techniques        113
      132 Basics of Unix Admin                  138
         134 Advanced Unix Admin                110
            135 Unix Tips and Techniques        112
            135 Unix Tips and Techniques        115
...
139 rows selected.
```

Sorting

The following query lists all the courses that require COURSE_NO 20 as a prerequisite. Examine the order of the rows with COURSE_NO 100, 140, 142, 147, and 204 in the following result. These five rows share the same hierarchy level (and the same parent PREREQUISITE value). The order in a hierarchy level is rather arbitrary.

```
SELECT LEVEL, LPAD(' ', 2*(LEVEL-1)) || c.course_no
       AS course_no,
       description, prerequisite AS pre
  FROM course c
 START WITH c.course_no = 20
CONNECT BY PRIOR c.course_no = prerequisite
 LEVEL COURSE_NO            DESCRIPTION                    PRE
------ -------------------- ------------------------------ -----
     1 20                   Intro to Information Systems
     2   100                Hands-On Windows                  20
     2   140                Systems Analysis                  20
     3     25               Intro to Programming             140
     4       240            Intro to the BASIC Language       25
     4       420            Database System Principles        25
     5         144          Database Design                  420
     2   142                Project Management                20
     2   147                GUI Design Lab                    20
     2   204                Intro to SQL                      20
     3     80               Programming Techniques           204
     4       120            Intro to Java Programming         80
...
     5         210          Oracle Tools                     220

20 rows selected.
```

If you want to order the result by the DESCRIPTION column, in alphabetical order, without destroying the hierarchical default order of the CONNECT BY clause, you use the ORDER SIBLINGS BY clause. It preserves the hierarchy and orders the siblings as specified in the ORDER BY clause. 727

730

```
SELECT LEVEL, LPAD(' ', 2*(LEVEL-1)) || c.course_no
       AS course_no,
       description, prerequisite AS pre
  FROM course c
 START WITH c.course_no = 20
CONNECT BY PRIOR c.course_no = prerequisite
 ORDER SIBLINGS BY description
LEVEL COURSE_NO          DESCRIPTION                    PRE
----- ------------------ ------------------------------ -----
    1 20                 Intro to Information Systems
    2   147              GUI Design Lab                    20
    2   100              Hands-On Windows                  20
    2   204              Intro to SQL                      20
    3     80             Programming Techniques           204
    4       120          Intro to Java Programming         80
    5         122        Intermediate Java Programming    120
    6           124      Advanced Java Programming        122
    6           125      Java Developer I                 122
...

20 rows selected.
```

Any other ORDER BY clause has the effect of the DESCRIPTION column now taking precedence over the default ordering. For example, the result of ordering by the DESCRIPTION column without the SIBLINGS keyword results in the following, where the hierarchy order is no longer intact.

```
SELECT LEVEL, LPAD(' ', 2*(LEVEL-1)) || c.course_no
       AS course_no,
       description, prerequisite AS pre
  FROM course c
 START WITH c.course_no = 20
CONNECT BY PRIOR c.course_no = prerequisite
 ORDER BY description
 LEVEL COURSE_NO            DESCRIPTION                    PRE
------ -------------------- ------------------------------ -----
     6           124        Advanced Java Programming       122
     8               450    DB Programming with Java        350
     5         144          Database Design                 420
...
     2   142                Project Management               20
     2   140                Systems Analysis                 20

20 rows selected.
```

Ordering by the LEVEL pseudocolumn results in all the parents being grouped together and then all children, all the grandchildren, and so on.

```
SELECT LEVEL, LPAD(' ', 2*(LEVEL-1)) || c.course_no
       AS course_no,
       description, prerequisite AS pre
  FROM course c
 START WITH c.course_no = 20
CONNECT BY PRIOR c.course_no = prerequisite
 ORDER BY LEVEL
 LEVEL COURSE_NO            DESCRIPTION                    PRE
------ -------------------- ------------------------------ -----
     1 20                   Intro to Information Systems
     2   100                Hands-On Windows                 20
     2   204                Intro to SQL                     20
     2   147                GUI Design Lab                   20
...
     8               430    Java Developer III              350

20 rows selected.
```

LAB 16.2 EXERCISES

a) Show the course number and course description of courses that have course number 310 as a prerequisite. Make these records the root of your hierarchical query. Display all the courses that can be taken after these root courses have been completed as child records. Include the LEVEL pseudocolumn as an additional column.

b) Execute the following query. What do you observe about the result?

```
SELECT LEVEL, LPAD(' ',
6*(LEVEL-1)) ||course_no
   || ' ' ||description hier
  FROM course
 START WITH course_no = 310
 CONNECT BY PRIOR course_no =
prerequisite
AND LEVEL <= 3
```

c) What does the following START WITH clause accomplish?

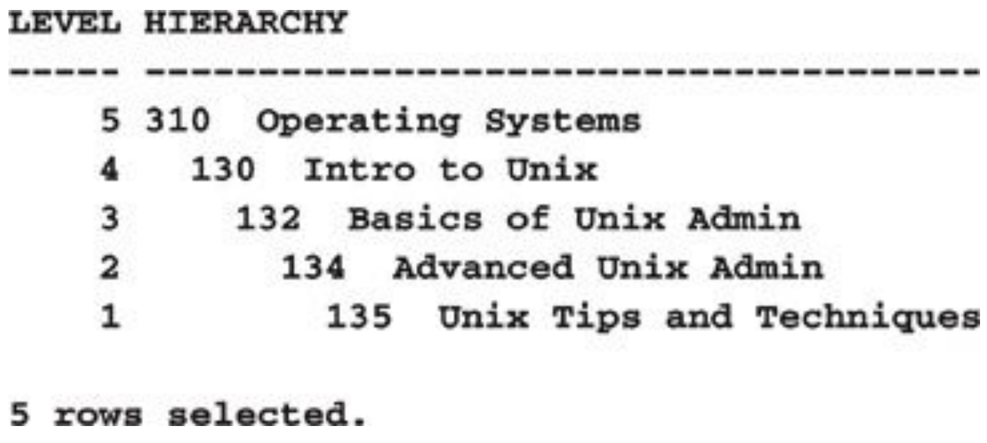
```
SELECT LEVEL, LPAD(' ',
3*(LEVEL-1)) ||course_no
  || ' ' ||description hier
```

```
 FROM course
 START WITH prerequisite IS NULL
 CONNECT BY PRIOR course_no =
prerequisite
```

d) Execute the following query, placing the PRIOR operator on the PREREQUISITE column. How does the result compare to the results of the previously issued queries?

```
SELECT LEVEL, LPAD(' ',
6*(LEVEL-1)) ||course_no
  || ' ' ||description hierarchy
 FROM course
START WITH course_no = 132
 CONNECT BY course_no = PRIOR
prerequisite
```

e) Write the SQL statement to display the following result.



```
LEVEL HIERARCHY
----- -----------------------------------
    5 310   Operating Systems
    4   130   Intro to Unix
    3     132   Basics of Unix Admin
    2       134   Advanced Unix Admin
    1         135   Unix Tips and Techniques

5 rows selected.
```

f) Insert the following record into the COURSE table and execute the query. What error message do you get, and why? Roll back the INSERT statement after you issue the SELECT statement.

```
INSERT INTO course
 (course_no, description,
prerequisite,
 created_by, created_date,
modified_by, modified_date)
VALUES
 (1000, 'Test', 1000,
 'TEST', SYSDATE, 'TEST',
SYSDATE)

SELECT course_no, prerequisite
 FROM course
 START WITH course_no = 1000
CONNECT BY PRIOR course_no =
prerequisite
 ROLLBACK
```

LAB 16.2 EXERCISE ANSWERS

a) Show the course number and course description of courses that have course number 310 as a prerequisite. Make these records the root of your hierarchical query. Display all the courses that can be taken after these root courses have been completed as child records. Include the LEVEL pseudocolumn as an additional column.

ANSWER: The START WITH clause starts the hierarchy with the prerequisite course number 310. The PRIOR operator identifies COURSE_NO as the parent record for which all the children are retrieved.

```
SELECT LEVEL, LPAD(' ', 6*(LEVEL-1)) ||course_no
       || '  ' ||description hierarchy
  FROM course
 START WITH prerequisite = 310
CONNECT BY PRIOR course_no = prerequisite
    LEVEL HIERARCHY
--------- ---------------------------------------------
        1 130   Intro to Unix
        2       132   Basics of Unix Admin
        3             134   Advanced Unix Admin
        4                   135   Unix Tips and Techniques
        2       330   Network Administration
        1 145   Internet Protocols

6 rows selected.
```

The START WITH condition returns two records, one for the Intro to Unix class and the second for Internet Protocols. These are the root records from the hierarchy.

```
START WITH prerequisite = 310
```

The PRIOR operator in the CONNECT BY clause identifies COURSE_NO as the parent. Child records are those records with the same course number in the PREREQUISITE column. The following two CONNECT BY clauses are equivalent.

```
CONNECT BY PRIOR course_no =
prerequisite
```

```
CONNECT BY prerequisite = PRIOR
course_no
```

If you use the PRIOR operator on the PREREQUISITE column, you reverse the hierarchy and travel in the opposite direction. You will see examples of this shortly.

Finally, you need to add the LEVEL function as a single column to display the hierarchy level of each record. If you also want to show the hierarchy visually with indentations, use the combination of LEVEL and LPAD (which has the following syntax).

```
LPAD(char1, n [, char2])
```

The LPAD function uses the first argument as a literal. If char2 is not specified, by default it is filled from the left with blanks up to the length shown as parameter n. The following SELECT clause indents each level with six additional spaces. (You can choose any number of spaces you like.)

```
SELECT LEVEL, LPAD(' ',
6*(LEVEL-1)) ||course_no
|| ' ' ||description hierarchy
```

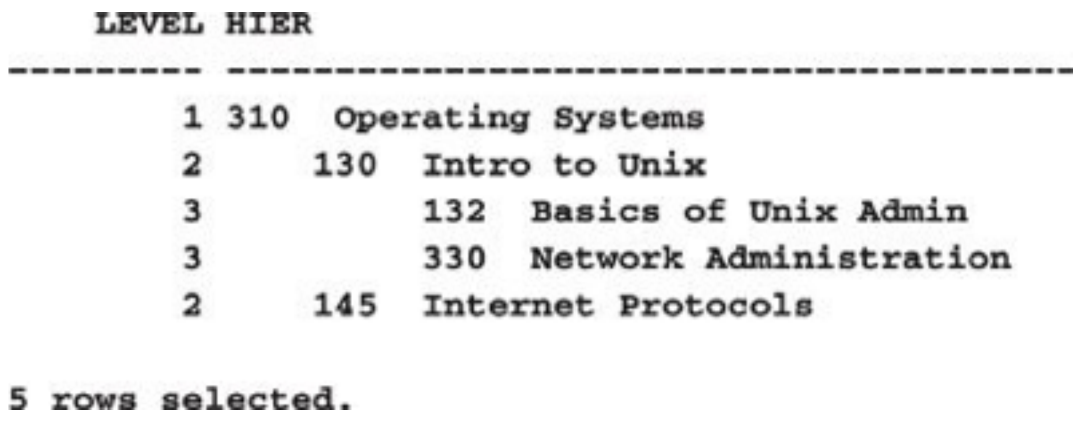
The length for the first level is 0 (Level 1 − 1 = 0); therefore, this level is not indented. The second level is indented by 12 spaces (6 * (2 − 1) = 6), the next by 12 (6 * (3 − 1) = 12), and so on. The resulting padded spaces are then concatenated with the course number and course description.

b) Execute the following query. What do you observe about the result?

```
SELECT LEVEL, LPAD(' ',
6*(LEVEL-1)) ||course_no
  || ' ' ||description hier
  FROM course
START WITH course_no = 310
CONNECT BY PRIOR course_no =
prerequisite
```

```
AND LEVEL <= 3
```

ANSWER: The LEVEL pseudocolumn restricts the rows in the CONNECT BY clause to show only the first three levels of the hierarchy.



```
    LEVEL HIER
--------- ----------------------------------------
        1 310   Operating Systems
        2     130   Intro to Unix
        3         132   Basics of Unix Admin
        3         330   Network Administration
        2     145   Internet Protocols

5 rows selected.
```

In the previous exercise, you learned that the WHERE clause eliminates the particular row but not its children. You restrict child rows with conditions in the CONNECT BY clause. Here the PRIOR operator applies to the parent row, and the other side of the equation applies to the child record. A qualifying child needs to have the correct parent, and it must have a LEVEL number of 3 or less.

c) What does the following START WITH clause accomplish?

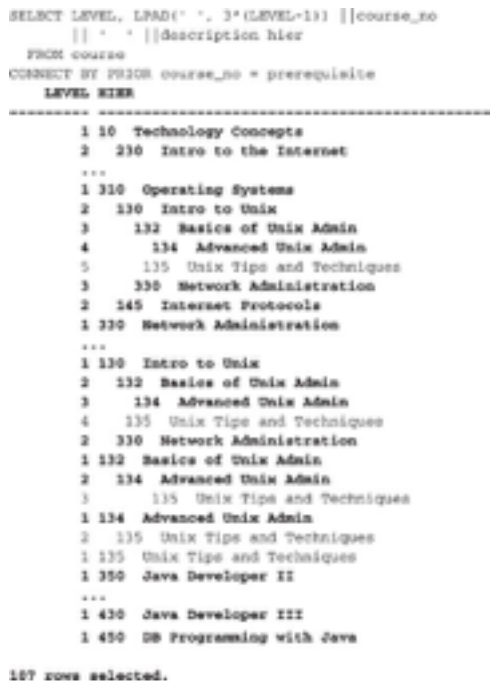
```
SELECT LEVEL, LPAD(' ',
3*(LEVEL-1)) ||course_no
```

```
   || ' ' ||description hier
FROM course
 START WITH prerequisite IS NULL
CONNECT BY PRIOR course_no =
prerequisite
```

ANSWER: This query's START WITH clause identifies all the root rows of the COURSE table. Those are the courses that have no prerequisites.

Although START WITH is optional with hierarchical queries, you typically identify the root rows of the hierarchy. That's the starting point for all rows.

The following statement displays the result of a query that doesn't have a START WITH clause.



```
SELECT LEVEL, LPAD(' ', 3*(LEVEL-1)) ||course_no
       || ' ' ||description hier
  FROM course
CONNECT BY PRIOR course_no = prerequisite
    LEVEL HIER
--------- ------------------------------------------
        1 10  Technology Concepts
        2    230  Intro to the Internet
        ...
        1 310  Operating Systems
        2    130  Intro to Unix
        3       132  Basics of Unix Admin
        4          134  Advanced Unix Admin
        5        135  Unix Tips and Techniques
        3       330  Network Administration
        2    145  Internet Protocols
        1 330  Network Administration
        ...
        1 130  Intro to Unix
        2    132  Basics of Unix Admin
        3       134  Advanced Unix Admin
        4     135  Unix Tips and Techniques
        2    330  Network Administration
        1 132  Basics of Unix Admin
        2    134  Advanced Unix Admin
        3          135  Unix Tips and Techniques
        1 134  Advanced Unix Admin
        2    135  Unix Tips and Techniques
        1 135  Unix Tips and Techniques
        1 350  Java Developer II
        ...
        1 430  Java Developer III
        1 450  DB Programming with Java

107 rows selected.
```

Although such a query is not very useful, it helps to understand why the records appear multiple times. When the START WITH clause is not specified, every record in the table is considered the root of the hierarchy. Therefore, for every record in the table, the hierarchy is displayed, and the courses are repeated multiple times.

For example, the course number 135, Unix Tips and Techniques, is returned five times. From the root 310, Operating Systems, it is five levels deep in the hierarchy. It is repeated for the root course number 130, Intro to Unix, and then for 132, Basics of Unix Admin, and then for 134, Advanced Unix Admin, and finally for itself.

d) Execute the following query, placing the PRIOR operator on the PREREQUISITE column. How does the result compare to the results of the previously issued queries?

```
SELECT LEVEL, LPAD(' ',
6*(LEVEL-1)) ||course_no
 || ' ' ||description hierarchy
 FROM course
START WITH course_no = 132
```

```
 CONNECT BY course_no = PRIOR
prerequisite
```

ANSWER: The PREREQUISITE column becomes the parent, and the COURSE_NO column becomes the child. This effectively reverses the direction of the hierarchy compared to the previously issued queries.

The result of the query shows all the prerequisites a student needs to take before enrolling in course number 132, Basics of Unix Administration.

```
    LEVEL HIERARCHY
--------- -----------------------------------
        1 132  Basics of Unix Admin
        2     130  Intro to Unix
        3         310  Operating Systems

3 rows selected.
```

The student needs to take course number 310, Operating Systems, and then course number 130, Intro to Unix, before taking course number 132, Basics of Unix Admin.

e) Write the SQL statement to display the following result.

```
LEVEL HIERARCHY
----- -------------------------------------
    5 310  Operating Systems
    4   130  Intro to Unix
    3    132  Basics of Unix Admin
    2     134  Advanced Unix Admin
    1      135  Unix Tips and Techniques

5 rows selected.
```

ANSWER: The rows show you the prerequisite courses for course number 135 as a root. The ORDER BY clause orders the result by the hierarchy level.

```
SELECT LEVEL, LPAD(' ', 2*(5-
LEVEL)) ||course_no
  || ' ' ||description hierarchy
FROM course
 START WITH course_no = 135
CONNECT BY course_no = PRIOR
prerequisite
 ORDER BY LEVEL DESC
```

Because the result shows the prerequisites, the PRIOR operator needs to be applied on the PREREQUISITE column. PREREQUISITE becomes the parent column.

```
CONNECT BY course_no = PRIOR
prerequisite
```

The ORDER BY clause orders the records by the hierarchy level, in descending order. The indentation with the LPAD function is different from previous examples. You now subtract the number 5 from each level and multiply the result by 2, resulting in the largest indentation for the root.

f) Insert the following record into the COURSE table and execute the query. What error message do you get, and why? Roll back the INSERT statement after you issue the SELECT statement.

```
INSERT INTO course
 (course_no, description,
prerequisite,
 created_by, created_date,
modified_by, modified_date)
VALUES
 (1000, 'Test', 1000,
'TEST', SYSDATE, 'TEST', SYSDATE)
```

```
SELECT course_no, prerequisite
 FROM course
START WITH course_no = 1000
```

```
 CONNECT BY PRIOR course_no =
prerequisite
ROLLBACK
```

ANSWER: The INSERT statement causes the course number 1000 to be its own parent and child. This results in a loop in the hierarchy and is reported by the hierarchical query.

```
SELECT course_no, prerequisite
  FROM course
 START WITH course_no = 1000
CONNECT BY PRIOR course_no = prerequisite

ERROR:
ORA-01436: CONNECT BY loop in user data


no rows selected
```

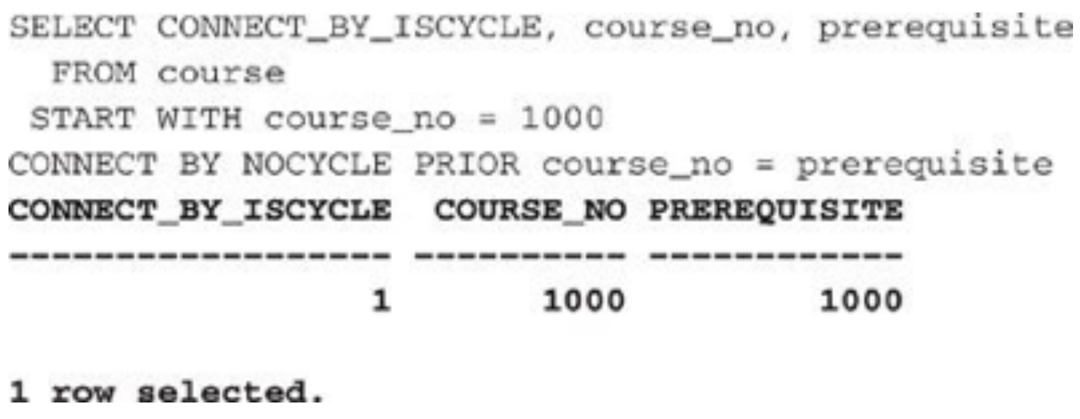
This is quite an obvious loop; because it is in the same record, the row is both the parent and the child. However, you can run the query without any error if you use the NOCYCLE parameter following CONNECT BY.

```
SELECT course_no, prerequisite
  FROM course
 START WITH course_no = 1000
CONNECT BY NOCYCLE PRIOR course_no = prerequisite
 COURSE_NO PREREQUISITE
---------- ------------
      1000         1000

1 row selected.
```

The pseudocolumn CONNECT_BY_ISCYCLE enables you to detect the offending row by displaying the value 1; otherwise, it shows 0. The pseudocolumn works only when the NOCYCLE parameter of the CONNECT BY clause is specified. 736

737



```
SELECT CONNECT_BY_ISCYCLE, course_no, prerequisite
  FROM course
 START WITH course_no = 1000
CONNECT BY NOCYCLE PRIOR course_no = prerequisite
CONNECT_BY_ISCYCLE  COURSE_NO PREREQUISITE
------------------ ---------- ------------
                 1       1000         1000

1 row selected.
```

Loops can be buried deep within the hierarchy and can be difficult to find when many rows are involved, unless you can use the CONNECT_BY_ISCYCLE pseudocolumn.

737

738

Lab 16.2 Quiz

In order to test your progress, you should be able to answer the following questions.

1) The ORDER BY clause does not order the columns within a hierarchy, but it does order the columns in

the order stated in the ORDER BY clause, unless the SIBLINGS keyword is used.

_____a) True

_____b) False

2) Which column is the parent in the following SQL statement?

```
CONNECT BY PRIOR emp = manager
```

_____a) The EMP column

_____b) The MANAGER column

_____c) None of the above

3) You can use joins in hierarchical queries.

_____a) True

_____b) False

4) The pseudocolumn CONNECT_BY_ISLEAF displays 0 if the row is the last branch of the hierarchy tree.

_____a) True

_____b) False

ANSWERS APPEAR IN APPENDIX A.

WORKSHOP

The projects in this section are meant to prompt you to utilize all the skills you have acquired throughout this chapter. The answers to these projects can be found at the companion Web site to this book, located at www.oraclesqlbyexample.com.

1) Name other hierarchical relationships you are familiar with.

2) Change the prerequisite of course number 310, Operating Systems, a root row in the hierarchy, from a null value to 145, Internet Protocols. Write the query to detect the loop in the hierarchy, using the CONNECT_BY_ISCYCLE pseudocolumn.

3) Why doesn't the following query return any rows?

```
SELECT *
  FROM instructor
WHERE REGEXP_LIKE(instructor_id,
'[:digit:]')
```

```
no rows selected
```

4) Add a Social Security number column to the STUDENT table or create a separate table with this column. Write a column check constraint that verifies that the Social Security number is entered in the correct ###-##-#### format.