Thanks for downloading this chapter from Murach’s Oracle SQL and PL/SQL for developers (2nd Edition). We hope it will show you how easy it is to learn from any Murach book, with its paired-pages presentation, its “how-to” headings, its practical coding examples, and its clear, concise style.

To view the full table of contents for this book, you can go to our web site. From there, you can read more about this book, you can find out about any additional downloads that are available, and you can review our other books on related topics.

Thanks for your interest in our books!
What developers have said about the previous edition

“I bought this book to take my SQL skills up a notch or more. This book really does that. It moves from the basics to advanced queries. The advanced queries are very well explained and you get useful skills and understand how to extend this to other queries.”

Posted at an online bookseller

“Inner and outer joins have never made as much sense as they did after completing Murach’s Oracle SQL. Additionally, working with user-defined PL/SQL functions and triggers was new ground for a self-taught coder like myself.”

Todd Jensen, GravitateOnline.com

“This book explained many of the ‘Oracle’ ways to do the stuff I already do with SQL Server. It is open daily when I work on Oracle code now.”

Posted at an online bookseller

“This book is worth its cost just for its detailed and complete description of the SQL SELECT statement. The book takes 4 chapters (129 pages) to cover all aspects of it. I have not found any other book, including Oracle’s own documentation, which more completely describes how to use the SELECT statement.”

Eric Mortensen, North East Ohio Oracle Users Group

“A lifesaver! Got stuck on one of my Oracle assignments and it took just one look in this book and I was all set and ready to take on other assignments. It is really a reliable reference book.”

Posted at an online bookseller

“A book that is great for Oracle novices, but also great for casual intermediate users like myself.”

David O’Meara, JavaRanch.com
The essential SQL skills

This section teaches you the essential SQL coding skills for working with the data in an Oracle database. The first four chapters show you how to retrieve data from a database using the SELECT statement. In chapter 3, you’ll learn how to code the basic clauses of the SELECT statement to retrieve data from a single table. In chapter 4, you’ll learn how to retrieve data from two or more tables. In chapter 5, you’ll learn how to summarize the data that you retrieve. And in chapter 6, you’ll learn how to code subqueries, which are SELECT statements that are coded within other statements.

Next, chapter 7 shows you how to use the INSERT, UPDATE, and DELETE statements to add new rows to a table, to modify rows in a table, and to remove rows from a table. Then, chapter 8 shows you how to work with the various types of data that Oracle supports and how to use some of the built-in functions that Oracle provides for working with these data types. When you complete the six chapters in this section, you’ll have all the skills you need to code SELECT, INSERT, UPDATE, and DELETE statements.
How to retrieve data from a single table

In this chapter, you’ll learn how to code SELECT statements that retrieve data from a single table. You should realize, though, that the skills covered here are the essential ones that apply to any SELECT statement, no matter how many tables it operates on and no matter how complex the retrieval. So you’ll want to have a good understanding of the material in this chapter before you go on to the chapters that follow.

An introduction to the SELECT statement .........................80
The basic syntax of the SELECT statement.................................80
SELECT statement examples ..............................................................82
How to code the SELECT clause ......................................................84
How to code column specifications .................................................84
How to name the columns in a result set ........................................86
How to code string expressions .......................................................88
How to code arithmetic expressions .................................................90
How to use scalar functions .............................................................92
How to use the Dual table ...............................................................94
How to use the DISTINCT keyword to eliminate duplicate rows .....96
How to use the ROWNUM pseudo column to limit the number of rows .................................................................98
How to code the WHERE clause ......................................................100
How to use the comparison operators .............................................100
How to use the AND, OR, and NOT logical operators.......................102
How to use the IN operator .............................................................104
How to use the BETWEEN operator ...............................................106
How to use the LIKE operator .......................................................108
How to use the IS NULL condition ................................................110
How to code the ORDER BY clause .................................................112
How to sort a result set by a column name .....................................112
How to sort a result set by an alias, an expression, or a column number .................................................................114
How to code the row limiting clause .............................................116
How to limit the number of rows ................................................116
How to return a range of rows ......................................................116
Perspective ..................................................................................118
An introduction to the SELECT statement

To help you learn to code the SELECT statement, this chapter starts by presenting its basic syntax. Next, it presents several examples that will give you an idea of what you can do with this statement. Then, the rest of this chapter will teach you the details of coding this statement.

The basic syntax of the SELECT statement

Figure 3-1 presents the basic syntax of the SELECT statement. The syntax summary at the top of this figure uses some conventions that are used throughout this book. First, capitalized words are called keywords, and you must spell them exactly as shown, though you can use whatever capitalization you prefer. For example, the following are equivalent: “SELECT”, “Select”, “select” and “sELeCt”. Second, you must provide replacements for words in lowercase. For example, you must enter a list of columns in place of select_list, and you must enter a table name in place of table_source.

Beyond that, you can choose between the items in a syntax summary that are separated by pipes (|) and enclosed in braces ({}), or brackets ([ ]). And you can omit items enclosed in brackets. If you have a choice between two or more optional items, the default item is underlined. And if an element can be coded multiple times in a statement, it’s followed by an ellipsis (...). You’ll see examples of pipes, braces, default values, and ellipses in syntax summaries later in this chapter. For now, compare the syntax in this figure with the coding examples in figure 3-2 to see how the two are related.

The syntax summary in this figure has been simplified so that you can focus on the four main clauses of the SELECT statement: the SELECT clause, the FROM clause, the WHERE clause, and the ORDER BY clause. Most of the SELECT statements you code will contain all four clauses. However, only the SELECT and FROM clauses are required.

The SELECT clause is always the first clause in a SELECT statement. It identifies the columns you want to include in the result set. These columns are retrieved from the base tables named in the FROM clause. Since this chapter focuses on retrieving data from a single table, the FROM clauses in all of the statements in this chapter name a single base table. In the next chapter, though, you’ll learn how to retrieve data from two or more tables. And as you progress through this book, you’ll learn how to select data from other sources such as views and expressions.

The WHERE and ORDER BY clauses are optional. The ORDER BY clause determines how the rows in the result set are to be sorted, and the WHERE clause determines which rows in the base table are to be included in the result set. The WHERE clause specifies a search condition that’s used to filter the rows in the base table. This search condition can consist of one or more Boolean expressions, or predicates. A Boolean expression is an expression that evaluates to True or False. When the search condition evaluates to True, the row is included in the result set.
The simplified syntax of the SELECT statement

\[
\text{SELECT select_list} \\
\text{FROM table_source} \\
[\text{WHERE search_condition}] \\
[\text{ORDER BY order_by_list}]
\]

The four clauses of the SELECT statement

<table>
<thead>
<tr>
<th>Clause</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>Describes the columns that will be included in the result set.</td>
</tr>
<tr>
<td>FROM</td>
<td>Names the table from which the query will retrieve the data.</td>
</tr>
<tr>
<td>WHERE</td>
<td>Specifies the conditions that must be met for a row to be included in the result set. This clause is optional.</td>
</tr>
<tr>
<td>ORDER BY</td>
<td>Specifies how the rows in the result set will be sorted. This clause is optional.</td>
</tr>
</tbody>
</table>

**Description**

- You use the basic SELECT statement shown above to retrieve the columns specified in the SELECT clause from the *base table* specified in the FROM clause and store them in a result set.
- The WHERE clause is used to *filter* the rows in the base table so that only those rows that match the search condition are included in the result set. If you omit the WHERE clause, all of the rows in the base table are included.
- The search condition of a WHERE clause consists of one or more *Boolean expressions*, or *predicates*, that result in a value of True, False, or Unknown. If the combination of all the expressions is True, the row being tested is included in the result set. Otherwise, it's not.
- If you include the ORDER BY clause, the rows in the result set are sorted in the specified sequence. Otherwise, the sequence of the rows is not guaranteed by Oracle.

**Note**

- The syntax shown above does not include all of the clauses of the SELECT statement. You'll learn about the other clauses later in this book.
In this book, we won’t use the terms “Boolean expression” or “predicate” because they don’t clearly describe the content of the WHERE clause. Instead, we’ll just use the term “search condition” to refer to an expression that evaluates to True or False.

**SELECT statement examples**

Figure 3-2 presents five SELECT statement examples. All of these statements retrieve data from the Invoices table.

The first statement in this figure retrieves all of the rows and columns from the Invoices table. This statement uses an asterisk (*) as a shorthand to indicate that all of the columns should be retrieved, and the WHERE clause is omitted so there are no conditions on the rows that are retrieved. You can see the results after this statement as they’re displayed by SQL Developer. Here, both horizontal and vertical scroll bars are displayed, indicating that the result set contains more rows and columns than can be displayed on the screen at one time.

Notice that this statement doesn’t include an ORDER BY clause. Without an ORDER BY clause, Oracle doesn’t guarantee the sequence in which the rows are presented. They might be in the sequence you expect, or they might not. As a result, if the sequence matters to you, you should include an ORDER BY clause.

The second statement retrieves selected columns from the Invoices table. As you can see, the columns to be retrieved are listed in the SELECT clause. Like the first statement, this statement doesn’t include a WHERE clause, so all the rows are retrieved. Then, the ORDER BY clause causes the rows to be sorted by the invoice_total column in ascending sequence. Later in this chapter, you’ll learn how to sort rows in descending sequence.

The third statement also lists the columns to be retrieved. In this case, though, the last column is calculated from two columns in the base table (credit_total and payment_total), and the resulting column is given the name total_credits. In addition, the WHERE clause specifies that only the invoice with an invoice_id of 17 should be retrieved.

The fourth SELECT statement includes a WHERE clause whose condition specifies a range of values. In this case, only invoices with invoice dates between May 1, 2014 and May 31, 2014 are retrieved. In addition, the rows in the result set are sorted by invoice date.

The last statement in this figure shows another variation of the WHERE clause. In this case, only those rows with an invoice_total greater than 50,000 are retrieved. Since none of the rows in the Invoices table satisfies this condition, the result set is empty.
A SELECT statement that retrieves all the data from the Invoices table

```
SELECT * 
FROM invoices 
```

(114 rows selected)

A SELECT statement that retrieves three columns from each row, sorted in ascending sequence by invoice_total

```
SELECT invoice_number, invoice_date, invoice_total 
FROM invoices 
ORDER BY invoice_total 
```

(114 rows selected)

A SELECT statement that retrieves two columns and a calculated value for a specific invoice

```
SELECT invoice_id, invoice_total, 
(credit_total + payment_total) AS total_credits 
FROM invoices 
WHERE invoice_id = 17 
```

A SELECT statement that retrieves all invoices between given dates

```
SELECT invoice_number, invoice_date, invoice_total 
FROM invoices 
WHERE invoice_date BETWEEN '01-MAY-2014' AND '31-MAY-2014' 
ORDER BY invoice_date 
```

(70 rows selected)

A SELECT statement that returns an empty result set

```
SELECT invoice_number, invoice_date, invoice_total 
FROM invoices 
WHERE invoice_total > 50000 
```

Figure 3-2  SELECT statement examples
How to code the SELECT clause

Figure 3-3 presents an expanded syntax for the SELECT clause. The keywords shown in the first line allow you to restrict the rows that are returned by a query. You’ll learn how to code them in a moment. But first, you’ll learn various techniques for identifying which columns are to be included in a result set.

How to code column specifications

Figure 3-3 summarizes the techniques you can use to code column specifications. You saw how to use some of these techniques in the previous figure. For example, you can code an asterisk in the SELECT clause to retrieve all of the columns in the base table, and you can code a list of column names separated by commas. Note that when you code an asterisk, the columns are returned in the order that they occur in the base table.

You can also code a column specification as an expression. For example, you can use an arithmetic expression to perform a calculation on two or more columns in the base table, and you can use a string expression to combine two or more string values. An expression can also include one or more functions. You’ll learn more about each of these techniques in the topics that follow.
**The expanded syntax of the SELECT clause**

```
SELECT [ALL|DISTINCT]
    column_specification [[AS] result_column]
    [, column_specification [[AS] result_column]] ...
```

**Five ways to code column specifications**

<table>
<thead>
<tr>
<th>Source</th>
<th>Option</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base table value</td>
<td>All columns</td>
<td>*</td>
</tr>
<tr>
<td></td>
<td>Column name</td>
<td>column_name</td>
</tr>
<tr>
<td>Calculated value</td>
<td>Result of a concatenation</td>
<td>String expression (see figure 3-5)</td>
</tr>
<tr>
<td></td>
<td>Result of a calculation</td>
<td>Arithmetic expression (see figure 3-6)</td>
</tr>
<tr>
<td></td>
<td>Result of a scalar function</td>
<td>Scalar function (see figure 3-7)</td>
</tr>
</tbody>
</table>

**Column specifications that use base table values**

The * is used to retrieve all columns

```
SELECT *
```

Column names are used to retrieve specific columns

```
SELECT vendor_name, vendor_city, vendor_state
```

**Column specifications that use calculated values**

An arithmetic expression is used to calculate balance_due

```
SELECT invoice_number, invoice_total - payment_total - credit_total AS balance_due
```

A string expression is used to derive full_name

```
SELECT first_name || ' ' || last_name AS full_name
```

**Description**

- Use SELECT * only when you need to retrieve all columns from a table. Otherwise, list the names of the columns you need.
- An expression is a combination of column names and operators that evaluate to a single value. In the SELECT clause, you can code arithmetic expressions, string expressions, and expressions that include one or more functions.
- After each column specification, you can code an AS clause to specify the name for the column in the result set. See figure 3-4 for details.

**Note**

- The other elements shown in the syntax summary above let you control the number of rows that are returned by a query. You can use the DISTINCT keyword to eliminate duplicate rows. See figure 3-9 for details.
How to name the columns in a result set

By default, a column in a result set is given the same name as the column in the base table. You can specify a different name, however, if you need to. You can also name a column that contains a calculated value. When you do that, the new column name is called a column alias. Figure 3-4 presents two techniques for creating column aliases.

The first technique is to code the column specification followed by the AS keyword and the column alias. This is the coding technique specified by the American National Standards Institute (ANSI, pronounced ‘ann-see’), and it’s illustrated by the first example in this figure.

The second technique is to code the column specification followed by a space and the column alias. This coding technique is illustrated by the second example. Whenever possible, though, you should use the first technique since the AS keyword makes it easier to identify the alias for the column, which makes your SQL statement easier to read and maintain.

When you code an alias, you must enclose the alias in double quotes if the alias contains a space or is a keyword that’s reserved by Oracle. In this figure, the first two examples specify an alias for the invoice_number column that uses two words with a space between them.

In addition, these two examples specify an alias for the invoice_date column that uses a keyword that’s reserved by Oracle: the DATE keyword. If you don’t enclose this keyword in double quotes, you will get an error when you attempt to execute either of these SQL statements. When you enter a statement into SQL Developer, it boldfaces keywords that are reserved by Oracle. This makes it easy to identify Oracle keywords when you’re writing SQL statements.

When you enclose an alias in double quotes, the result set uses the capitalization specified by the alias. Otherwise, the result set capitalizes all letters in the column name. In this figure, for instance, the first two columns in the first result set use the capitalization specified by the aliases. However, since no alias is specified for the third column, all letters in the name of this column are capitalized.

When you code a column that contains a calculated value, it’s a good practice to specify an alias for the calculated column. If you don’t, Oracle will assign the entire calculation as the name, which can be unwieldy, as shown in the third example. As a result, you usually assign a name to any column that’s calculated from other columns in the base table.
Two SELECT statements that name the columns in the result set

A SELECT statement that uses the AS keyword

```sql
-- DATE is a reserved keyword.
-- As a result, it must be enclosed in quotations.
SELECT invoice_number AS "Invoice Number", invoice_date AS "Date",
       invoice_total AS total
FROM invoices
```

A SELECT statement that omits the AS keyword

```sql
SELECT invoice_number "Invoice Number", invoice_date "Date",
       invoice_total total
FROM invoices
```

The result set for both SELECT statements

<table>
<thead>
<tr>
<th>Invoice Number</th>
<th>Date</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Q558872</td>
<td>25-FEB-14</td>
<td>116.54</td>
</tr>
<tr>
<td>2 Q549443</td>
<td>14-MAR-14</td>
<td>1083.58</td>
</tr>
<tr>
<td>3 P-0608</td>
<td>11-APR-14</td>
<td>20551.18</td>
</tr>
<tr>
<td>4 P-0259</td>
<td>16-APR-14</td>
<td>26851.4</td>
</tr>
<tr>
<td>5 MAB01489</td>
<td>16-APR-14</td>
<td>936.93</td>
</tr>
</tbody>
</table>

A SELECT statement that doesn’t provide a name for a calculated column

```sql
SELECT invoice_number, invoice_date, invoice_total,
       invoice_total - payment_total - credit_total
FROM invoices
```

<table>
<thead>
<tr>
<th>INVOICE_NUMBER</th>
<th>INVOICE_DATE</th>
<th>INVOICE_TOTAL</th>
<th>INVOICE_TOTAL-PAYMENT_TOTAL-CREDIT_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Q558872</td>
<td>25-FEB-14</td>
<td>116.54</td>
<td>0</td>
</tr>
<tr>
<td>2 Q549443</td>
<td>14-MAR-14</td>
<td>1083.58</td>
<td>0</td>
</tr>
<tr>
<td>3 P-0608</td>
<td>11-APR-14</td>
<td>20551.18</td>
<td>19551.18</td>
</tr>
<tr>
<td>4 P-0259</td>
<td>16-APR-14</td>
<td>26851.4</td>
<td>0</td>
</tr>
<tr>
<td>5 MAB01489</td>
<td>16-APR-14</td>
<td>936.93</td>
<td>0</td>
</tr>
</tbody>
</table>

Description

- By default, a column in the result set is given the same name as the column in the base table. If that’s not what you want, you can specify a column alias for the column.
- One way to name a column is to use the AS keyword as shown in the first example above. Although the AS keyword is optional, it enhances readability.
- If an alias includes spaces or an Oracle reserved keyword, you must enclose it in double quotes.
- When you enclose an alias in quotes, the result set uses the capitalization specified by the alias. Otherwise, the result set capitalizes all letters in the column name.
How to code string expressions

A *string expression* consists of a combination of one or more character
columns and *literal values*. To combine, or *concatenate*, the columns and values,
you use the *concatenation operator* (||). This is illustrated by the examples in
figure 3-5.

The first example shows how to concatenate the vendor_city and
vendor_state columns in the Vendors table. Notice that because no alias is
assigned to this column, Oracle assigns a name, which is the entire expression.
Also notice that the data in the vendor_state column appears immediately after
the data in the vendor_city column in the results. That’s because of the way
vendor_city is defined in the database. Because it’s defined as a variable-length
column (the VARCHAR2 data type), only the actual data in the column is
included in the result. In contrast, if the column had been defined with a fixed
length, any spaces after the name would have been included in the result. You’ll
learn about data types and how they affect the data in your result set in chapter 8.

The second example shows how to format a string expression by adding
spaces and punctuation. Here, the vendor_city column is concatenated with a
*string literal*, or *string constant*, that contains a comma and a space. Then, the
vendor_state column is concatenated with that result, followed by a string literal
that contains a single space and the vendor_zip_code column.

Occasionally, you may need to include a single quotation mark or an
apostrophe within a literal string. If you simply type a single quote, however, the
system will misinterpret it as the end of the literal string. As a result, you must
code two single quotation marks in a row. This is illustrated by the third example
in this figure.
How to concatenate string data

```sql
SELECT vendor_city, vendor_state, vendor_city || vendor_state
FROM vendors
```

![Vendor City and State Concatenation](image1)

How to format string data using literal values

```sql
SELECT vendor_name, 
    vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
```

![Vendor Address Formatting](image2)

How to include apostrophes in literal values

```sql
SELECT vendor_name || '''s address: ', 
    vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code
FROM vendors
```

![Vendor Address with Apostrophes](image3)

**Description**

- A string expression can consist of one or more character columns, one or more literal values, or a combination of character columns and literal values.
- The columns specified in a string expression must contain string data (that means they’re defined with the CHAR or VARCHAR2 data type).
- The literal values in a string expression also contain string data, so they can be called string literals or string constants. To create a literal value, enclose one or more characters within single quotation marks (').
- You can use the concatenation operator (||) to combine columns and literals in a string expression.
- You can include a single quote within a literal value by coding two single quotation marks (').

---

Figure 3-5   How to code string expressions
How to code arithmetic expressions

Figure 3-6 shows how to code arithmetic expressions. To start, it summarizes the four arithmetic operators you can use in this type of expression. Then, it presents two examples that show how to use these operators.

The SELECT statement in the first example includes an arithmetic expression that calculates the balance due for an invoice. This expression subtracts the payment_total and credit_total columns from the invoice_total column. The resulting column is given the name balance_due.

When Oracle evaluates an arithmetic expression, it performs the operations from left to right based on the order of precedence. This order says that multiplication and division are done first, followed by addition and subtraction. If that’s not what you want, you can use parentheses to specify how you want an expression evaluated. Then, the expressions in the innermost sets of parentheses are evaluated first, followed by the expressions in outer sets of parentheses. Within each set of parentheses, the expression is evaluated from left to right in the order of precedence.

To illustrate how parentheses and the order of precedence affect the evaluation of an expression, consider the second example in this figure. Here, the expressions in the second and third columns both use the same operators. However, when Oracle evaluates the expression in the second column, it performs the multiplication operation before the addition operation because multiplication comes before addition in the order of precedence. In contrast, when Oracle evaluates the expression in the third column, it performs the addition operation first because it’s enclosed in parentheses. As you can see in the result set, these two expressions result in different values.

Unlike some other databases, Oracle doesn’t provide a modulo operator that can be used to return the remainder of a division operation. Instead, you must use the MOD function as described in the next figure.
The arithmetic operators in order of precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
</tbody>
</table>

A SELECT statement that calculates the balance due

```sql
SELECT invoice_total, payment_total, credit_total,
       invoice_total - payment_total - credit_total AS balance_due
FROM invoices
```

<table>
<thead>
<tr>
<th>INVOICE_TOTAL</th>
<th>PAYMENT_TOTAL</th>
<th>CREDIT_TOTAL</th>
<th>BALANCE_DUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>116.54</td>
<td>116.54</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1083.58</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20551.18</td>
<td>0</td>
<td>1200</td>
<td>19351.18</td>
</tr>
<tr>
<td>26881.4</td>
<td>26881.4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>936.93</td>
<td>936.93</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

A SELECT statement that uses parentheses to control the sequence of operations

```sql
SELECT invoice_id,
       invoice_id + 7 * 3 AS order_of_precedence,
       (invoice_id + 7) * 3 AS add_first
FROM invoices
ORDER BY invoice_id
```

<table>
<thead>
<tr>
<th>INVOICE_ID</th>
<th>ORDER_OF_PRECEDENCE</th>
<th>ADD_FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
<td>27</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>26</td>
<td>36</td>
</tr>
</tbody>
</table>

Description

- Unless parentheses are used, the operations in an expression take place from left to right in the order of precedence. For arithmetic expressions, multiplication and division are done first, followed by addition and subtraction.
- Whenever necessary, you can use parentheses to clarify or override the sequence of operations. Then, the operations in the innermost sets of parentheses are done first, followed by the operations in the next sets, and so on.
How to use scalar functions

Figure 3-7 introduces you to scalar functions, which operate on a single value and return a single value. These functions work differently than the aggregate functions described in chapter 5 that are used to summarize data. For now, don’t worry about the details of how these functions work, because you’ll learn more about them in chapter 8. Instead, just focus on how they’re used in column specifications.

To code a function, you begin by entering its name followed by a set of parentheses. If the function requires one or more parameters, you enter them within the parentheses and separate them with commas. When you enter a parameter, you need to be sure it has the correct data type.

The first example in this figure shows how to use the SUBSTR function to extract the first character of the vendor_contact_first_name and vendor_contact_last_name columns. The first parameter of this function specifies the column name; the second parameter specifies the starting position; and the third parameter specifies the number of characters to return. The results of the two functions are then concatenated to form initials, as shown in the result set for this statement.

The second example shows how to use the TO_CHAR function. This function converts a column with a DATE or NUMBER data type to a character string. A common use for it is in concatenation operations, where all the data being concatenated must be string data. This function has two parameters. The first parameter, which specifies the column name, is required. The second parameter, which specifies a format mask for the column, is optional. In this example, a format mask of ‘MM/DD/YYYY’ is used to convert the payment_date column from a DATE type to a CHAR type. This format mask specifies that the date should be displayed with a two-digit month, followed by a forward slash, followed by a two-digit day, followed by another forward slash, followed by a four-digit year.

In the second example, the payment_date column for Invoice # P-0608 is NULL. Note that this causes the TO_CHAR function to return an empty string for the payment date.

The third example uses the SYSDATE function to return the current date. Since this function doesn’t accept any parameters, you don’t need to code any parentheses after the name of the function. In fact, if you do code parentheses after the name of the function, you will get an error when you execute the statement. In this example, the second column uses the SYSDATE function to return the current date, and the third column uses the SYSDATE function to calculate the number of days between the two dates. Here, the third column also uses the ROUND function to round the decimal value that’s returned by the calculation to an integer.

The fourth example shows how to use the MOD function to return the remainder of a division of two integers. Here, the second column contains an expression that returns the remainder of the division operation when the invoice-id column is divided by 10. In the result set, you can see the results for IDs 9 through 11 (the remainders are 9, 0, and 1).
A SELECT statement that uses the SUBSTR function

```
SELECT vendor_contact_first_name, vendor_contact_last_name,
       SUBSTR(vendor_contact_first_name, 1, 1) ||
       SUBSTR(vendor_contact_last_name, 1, 1) AS initials
FROM vendors
```

<table>
<thead>
<tr>
<th>VENDOR_CONTACT_FIRST_NAME</th>
<th>VENDOR_CONTACT_LAST_NAME</th>
<th>INITIALS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cecar</td>
<td>Arodondo</td>
<td>CA</td>
</tr>
<tr>
<td>Rachael</td>
<td>Danielson</td>
<td>RD</td>
</tr>
<tr>
<td>2gv</td>
<td>Alondra</td>
<td>2A</td>
</tr>
<tr>
<td>Selina</td>
<td>Edgardo</td>
<td>SE</td>
</tr>
<tr>
<td>Daniel</td>
<td>Bradlee</td>
<td>DS</td>
</tr>
</tbody>
</table>

A SELECT statement that uses the TO_CHAR function

```
SELECT 'Invoice: # ' || invoice_number || ', dated ' ||
       TO_CHAR(payment_date, 'MM/DD/YYYY') ||
       ' for $' || TO_CHAR(payment_total) AS "Invoice Text"
FROM invoices
```

<table>
<thead>
<tr>
<th>InvoiceText</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Invoice: # 0568872, dated 04/11/2014 for 0116.54</td>
</tr>
<tr>
<td>2 Invoice: # 0545443, dated 05/14/2014 for 01081.58</td>
</tr>
<tr>
<td>3 Invoice: # P0123, dated for 0</td>
</tr>
<tr>
<td>4 Invoice: # P0258, dated 05/12/2014 for 02681.4</td>
</tr>
<tr>
<td>5 Invoice: # NAB01489, dated 05/13/2014 for 4936.98</td>
</tr>
</tbody>
</table>

A SELECT statement that uses the SYSDATE and ROUND functions

```
SELECT invoice_date,
       SYSDATE AS today,
       ROUND(SYSDATE - invoice_date) AS invoice_age_in_days
FROM invoices
```

<table>
<thead>
<tr>
<th>INVOICE_DATE</th>
<th>TODAY</th>
<th>INVOICE_AGE_IN_DAYS</th>
</tr>
</thead>
<tbody>
<tr>
<td>18-JUL-14</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>20-JUN-14</td>
<td></td>
<td>29</td>
</tr>
<tr>
<td>17-JUN-14</td>
<td></td>
<td>35</td>
</tr>
</tbody>
</table>

A SELECT statement that uses the MOD function

```
SELECT invoice_id,
       MOD(invoice_id, 10) AS Remainder
FROM invoices
ORDER BY invoice_id
```

<table>
<thead>
<tr>
<th>INVOICE_ID</th>
<th>REMAINDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
</tbody>
</table>

**Description**
- A SQL statement can include a *function*. A function performs an operation and returns a value.
- For more information on using functions, see chapter 8.
How to use the Dual table

The Dual table is automatically available to all users. This table is useful for testing expressions that use literal values, arithmetic calculations, and functions as shown in figure 3-8. In particular, the Dual table is often used in the documentation that shows how Oracle’s built-in scalar functions work.

In the example in this figure, the second column in the result set shows the value of the calculation 10 minus 7, and the third column shows the date that’s returned by the SYSDATE function. This shows that you can perform test calculations in more than one column of the Dual table.
A SELECT statement that uses the Dual table

```
SELECT 'test'  AS test_string,
      10-7    AS test_calculation,
      SYSDATE AS test_date
FROM Dual
```

Description

- The Dual table is automatically created and made available to users.
- The Dual table is useful for testing expressions that use literal values, arithmetic, operators, and functions.
How to use the DISTINCT keyword to eliminate duplicate rows

By default, all of the rows in the base table that satisfy the search condition in the WHERE clause are included in the result set. In some cases, though, that means that the result set will contain duplicate rows, or rows whose column values are identical. If that’s not what you want, you can include the DISTINCT keyword in the SELECT clause to eliminate the duplicate rows.

Figure 3-9 illustrates how this works. Here, both SELECT statements retrieve the vendor_city and vendor_state columns from the Vendors table. The first statement, however, doesn’t include the DISTINCT keyword. Because of that, the same city and state can appear in the result set multiple times. In the results shown in this figure, for example, you can see that “Anaheim CA” occurs twice. In contrast, the second statement includes the DISTINCT keyword, so each city/state combination is included only once.
A SELECT statement that returns all rows

```
SELECT vendor_city, vendor_state
FROM vendors
ORDER BY vendor_city
```

(122 rows selected)

A SELECT statement that eliminates duplicate rows

```
SELECT DISTINCT vendor_city, vendor_state
FROM vendors
ORDER BY vendor_city
```

(53 rows selected)

Description

- The DISTINCT keyword prevents duplicate (identical) rows from being included in the result set.
- The ALL keyword causes all rows matching the search condition to be included in the result set, regardless of whether rows are duplicated. Since this is the default, it’s a common practice to omit the ALL keyword.
- To use the DISTINCT or ALL keyword, code it immediately after the SELECT keyword.
How to use the ROWNUM pseudo column to limit the number of rows

In addition to eliminating duplicate rows, you can limit the number of rows that are retrieved by a SELECT statement. To do that, you can use the ROWNUM pseudo column as shown in figure 3-10. A *pseudo column* works similarly to a column in a table. However, you can only use a pseudo column to select data. In other words, you can’t insert, update, or delete the values stored in a pseudo column.

If you want to learn more about how pseudo columns work, you can search the Oracle Database SQL Reference manual for *pseudocolumn*. Note that the Oracle documentation doesn’t include a space between the words pseudo and column.

The first example shows how to limit the number of rows in the result set to the first five rows. Here, the ROWNUM pseudo column is used in the WHERE clause to return the first five rows in the Invoices table.

The second example shows how to add an ORDER BY clause to sort the first five rows of the table so the largest invoice total is displayed first. Since the sort operation is applied after the first five rows are retrieved, this doesn’t retrieve the five largest invoice totals in the Invoices table. Instead, it returns the first five rows of the table and then sorts them.

If you want to return the five largest invoice totals for the entire Invoices table, you need to sort the result set before you use the ROWNUM pseudo column to limit the number of rows included in the result set. To do that, you can use a *subquery* as shown in the third example. In the FROM clause, this example supplies a SELECT statement that sorts the Invoices table instead of supplying the name of the Invoices table. As a result, the table is sorted before the WHERE clause is applied.

For more information about working with subqueries, see chapter 6. In addition, if the ROWNUM pseudo column isn’t adequate for your needs, you might want to use the ROW_NUMBER function described in chapter 8.
A SELECT statement that uses the ROWNUM pseudo column to limit the number of rows in the result set

```
SELECT vendor_id, invoice_total
FROM invoices
WHERE ROWNUM <= 5
```

![Table 1](image1)

A SELECT statement that sorts the result set after the WHERE clause

```
SELECT vendor_id, invoice_total
FROM invoices
WHERE ROWNUM <= 5
ORDER BY invoice_total DESC
```

![Table 2](image2)

A SELECT statement that sorts the result set before the WHERE clause

```
SELECT vendor_id, invoice_total
FROM (SELECT * FROM invoices
     ORDER BY invoice_total DESC)
WHERE ROWNUM <= 5
```

![Table 3](image3)

Description

- You can use the ROWNUM pseudo column to limit the number of rows included in the result set.
- If you want to sort the result set before you use the ROWNUM pseudo column to limit the number of rows included in the result set, you can use a subquery as shown in the third example. For more information about working with subqueries, see chapter 6.
How to code the WHERE clause

The WHERE clause in a SELECT statement filters the rows in the base table so that only the rows you need are retrieved. In the topics that follow, you’ll learn a variety of ways to code this clause.

How to use the comparison operators

Figure 3-11 shows you how to use the comparison operators in the search condition of a WHERE clause. As you can see in the syntax summary at the top of this figure, you use a comparison operator to compare two expressions. If the result of the comparison is True, the row being tested is included in the query results.

The examples in this figure show how to use some of the comparison operators. The first WHERE clause, for example, uses the equal operator (=) to retrieve only those rows whose vendor_state column have a value of IA. Since the state code is a string literal, it must be enclosed in single quotes.

In contrast, a numeric literal like the one in the second WHERE clause isn’t enclosed in quotes. This clause uses the greater than (>) operator to retrieve only those rows that have a balance due greater than zero.

The third WHERE clause illustrates another way to retrieve all the invoices with a balance due. Like the second clause, it uses the greater than operator. Instead of comparing the balance due to a value of zero, however, it compares the invoice total to the total of the payments and credits that have been applied to the invoice.

The fourth WHERE clause illustrates how you can use comparison operators other than the equal operator with string data. In this example, the less than operator (<) is used to compare the value of the vendor_name column to a literal string that has the letter M in the first position. That will cause the query to return all vendors with names that begin with the letters A through L.

You can also use the comparison operators with date literals, as illustrated by the fifth and sixth WHERE clauses. The fifth clause will retrieve rows with invoice dates on or before May 31, 2014, and the sixth clause will retrieve rows with invoice dates on or after May 1, 2014. Like string literals, date literals must be enclosed in single quotes. In addition, you can use different formats to specify dates, as shown by the two date literals shown in this figure. You’ll learn more about the acceptable date formats and date comparisons in chapter 8.

The last WHERE clause shows how you can test for a not equal condition. To do that, you code a less than sign followed by a greater than sign. In this case, only rows with a credit total that’s not equal to zero will be retrieved.

Whenever possible, you should compare expressions that have similar data types. If you attempt to compare expressions that have different data types, Oracle may implicitly convert the data types for you. Although implicit conversions are often acceptable, they will occasionally yield unexpected results. In chapter 8, you’ll learn how to explicitly convert data types so your comparisons will always yield the results that you want.
The syntax of the WHERE clause with comparison operators

\[
\text{WHERE expression_1 operator expression_2}
\]

The comparison operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal</td>
</tr>
</tbody>
</table>

Examples of WHERE clauses that retrieve...

Vendors located in Iowa

WHERE vendor_state = 'IA'

Invoices with a balance due (two variations)

WHERE invoice_total – payment_total – credit_total > 0
WHERE invoice_total > payment_total + credit_total

Vendors with names from A to L

WHERE vendor_name < 'M'

Invoices on or before a specified date

WHERE invoice_date <= '31-MAY-14'

Invoices on or after a specified date

WHERE invoice_date >= '01-MAY-14'

Invoices with credits that don’t equal zero

WHERE credit_total <> 0

Description

- You can use a comparison operator to compare any two expressions that result in like data types. Although unlike data types may be converted to data types that can be compared, the comparison may produce unexpected results.
- If the result of a comparison results in a True value, the row being tested is included in the result set. If it’s False or Unknown, the row isn’t included.
- To use a string literal or a date literal in a comparison, enclose it in quotes. To use a numeric literal, enter the number without quotes.
- Character comparisons are case-sensitive. ‘CA’ and ‘Ca’, for example, are not equivalent.
How to use the AND, OR, and NOT logical operators

Figure 3-12 shows how to use logical operators in a WHERE clause. You can use the AND and OR operators to combine two or more search conditions into a compound condition. And you can use the NOT operator to negate a search condition. The examples in this figure illustrate how these operators work.

The first two examples illustrate the difference between the AND and OR operators. When you use the AND operator, both conditions must be true. So in the first example, only those vendors located in Springfield, New Jersey, are retrieved from the Vendors table. When you use the OR operator, though, only one of the conditions must be true. So in the second example, all vendors located in New Jersey and all the vendors located in Pittsburgh are retrieved.

The third example shows a compound condition that uses two NOT operators. As you can see, this expression is difficult to understand. Because of that, you should avoid using this operator. The fourth example in this figure, for instance, shows how the search condition in the third example can be rephrased to eliminate the NOT operator. As a result, the condition in the fourth example is much easier to understand.

The last two examples in this figure show how the order of precedence for the logical operators and the use of parentheses affect the result of a search condition. By default, the NOT operator is evaluated first, followed by AND and then OR. However, you can use parentheses to override the order of precedence or to clarify a logical expression, just as you can with arithmetic expressions. In the next to last example, for instance, no parentheses are used, so the two conditions connected by the AND operator are evaluated first. In the last example, though, parentheses are used so the two conditions connected by the OR operator are evaluated first.
The syntax of the WHERE clause with logical operators

```
WHERE [NOT] search_condition_1 {AND|OR} [NOT] search_condition_2 ...
```

Examples of queries using logical operators

A search condition that uses the AND operator
```
WHERE vendor_state = 'NJ' AND vendor_city = 'Springfield'
```

A search condition that uses the OR operator
```
WHERE vendor_state = 'NJ' OR vendor_city = 'Pittsburgh'
```

A search condition that uses the NOT operator
```
WHERE NOT (invoice_total >= 5000 OR NOT invoice_date <= '01-JUL-2014')
```

The same condition rephrased to eliminate the NOT operator
```
WHERE invoice_total < 5000 AND invoice_date <= '01-JUL-2014'
```

A compound condition without parentheses
```
SELECT invoice_number, invoice_date, invoice_total
FROM invoices
WHERE invoice_date > '01-MAY-2014' OR invoice_total > 500
AND invoice_total – payment_total – credit_total > 0
ORDER BY invoice_number
```

(91 rows selected)

The same compound condition with parentheses
```
WHERE (invoice_date > '01-MAY-2014' OR invoice_total > 500)
AND invoice_total – payment_total – credit_total > 0
ORDER BY invoice_number
```

(39 rows selected)

Description

- You can use the AND and OR logical operators to create compound conditions that consist of two or more conditions. You use the AND operator to specify that the search must satisfy both of the conditions, and you use the OR operator to specify that the search must satisfy at least one of the conditions.
- You can use the NOT operator to negate a condition, but that can make the search condition difficult to understand. If it does, you should rephrase the condition to eliminate NOT.
- When Oracle evaluates a compound condition, it evaluates the operators in this sequence: (1) NOT, (2) AND, and (3) OR. You can use parentheses to override this order of precedence or to clarify the sequence in which the operations will be evaluated.
How to use the IN operator

Figure 3-13 shows how to code a WHERE clause that uses the IN operator. When you use this operator, the value of the test expression is compared with the list of expressions in the IN phrase. If the test expression is equal to one of the expressions in the list, the row is included in the query results. This is illustrated by the first example in this figure, which will return all rows whose terms_id column is equal to 1, 3, or 4.

You can also use the NOT operator with the IN phrase to test for a value that’s not in a list of expressions. This is illustrated by the second example in this figure. In this case, only those vendors who are not in California, Nevada, or Oregon are retrieved.

If you look at the syntax of the IN phrase shown at the top of this figure, you’ll see that you can code a subquery in place of a list of expressions. Subqueries are a powerful tool that you’ll learn about in chapter 6. For now, though, you should know that a subquery is simply a SELECT statement within another statement. In the third example in this figure, for instance, a subquery is used to return a list of vendor_id values for vendors who have invoices dated May 1, 2014. Then, the WHERE clause retrieves a vendor row only if the vendor is in that list. Note that for this to work, the subquery must return a single column, in this case, vendor_id.
The syntax of the WHERE clause with the IN operator

```
WHERE test_expression [NOT] IN ({subquery|expression_1 [, expression_2]...})
```

Examples of the IN operator

**The IN operator with a list of numeric literals**

```
WHERE terms_id IN (1, 3, 4)
```

**The IN operator preceded by NOT**

```
WHERE vendor_state NOT IN ('CA', 'NV', 'OR')
```

**The IN operator with a subquery**

```
WHERE vendor_id IN
  (SELECT vendor_id
    FROM invoices
    WHERE invoice_date = '01-MAY-2014')
```

Description

- You can use the IN operator to test whether an expression is equal to a value in a list of expressions. Each of the expressions in the list must evaluate to the same type of data as the test expression.
- The list of expressions can be coded in any order without affecting the order of the rows in the result set.
- You can use the NOT operator to test for an expression that’s not in the list of expressions.
- You can also compare the test expression to the items in a list returned by a subquery as illustrated by the third example above. You’ll learn more about coding subqueries in chapter 6.
How to use the BETWEEN operator

Figure 3-14 shows how to use the BETWEEN operator in a WHERE clause. When you use this operator, the value of a test expression is compared to the range of values specified in the BETWEEN phrase. If the value falls within this range, the row is included in the query results.

The first example in this figure shows a simple WHERE clause that uses the BETWEEN operator. It retrieves invoices with invoice dates between May 1, 2014 and May 31, 2014. Note that the range is inclusive, so invoices with invoice dates of May 1 and May 31 are included in the results.

The second example shows how to use the NOT operator to select rows that are not within a given range. In this case, vendors with zip codes that aren’t between 93600 and 93799 are included in the results.

The third example shows how you can use a calculated value in the test expression. Here, the payment_total and credit_total columns are subtracted from the invoice_total column to give the balance due. Then, this value is compared to the range specified in the BETWEEN phrase.

The last example shows how you can use calculated values in the BETWEEN phrase. Here, the first value selects the function SYSDATE (which represents the current date), and the second value is SYSDATE plus 30 days. So the query results will include all those invoices that are due between the current date and 30 days from the current date.

However, please note the warning about date comparisons in this figure. In particular, an invoice-date of May 31, 2014 at 2:00 PM isn’t less than or equal to “31-May-2008”, and it isn’t between “01-May-2014” and “31-May-2014”. To learn more about date comparisons, please read chapter 8.
The syntax of the WHERE clause with the BETWEEN operator

```
WHERE test_expression [NOT] BETWEEN begin_expression AND end_expression
```

Examples of the BETWEEN operator

**The BETWEEN operator with literal values**

```
WHERE invoice_date BETWEEN '01-MAY-2014' AND '31-MAY-2014'
```

**The BETWEEN operator preceded by NOT**

```
WHERE vendor_zip_code NOT BETWEEN 93600 AND 93799
```

**The BETWEEN operator with a test expression coded as a calculated value**

```
WHERE invoice_total – payment_total – credit_total BETWEEN 200 AND 500
```

**The BETWEEN operator with the upper and lower limits coded as calculated values**

```
WHERE invoice_due_date BETWEEN SYSDATE AND (SYSDATE + 30)
```

**Description**

- You can use the BETWEEN operator to test whether an expression falls within a range of values. The lower limit must be coded as the first expression, and the upper limit must be coded as the second expression. Otherwise, the result set will be empty.
- The two expressions used in the BETWEEN operator for the range of values are inclusive. That is, the result set will include values that are equal to the upper or lower limit.
- You can use the NOT operator to test for an expression that’s not within the given range.

**Warning about date comparisons**

- All columns that have the DATE data type include both a date and time, and so does the value returned by the SYSDATE function. But when you code a date literal like ‘31-May-2014’, the time defaults to 00:00:00 on a 24-hour clock, or 12:00 AM (midnight). As a result, a date comparison may not yield the results you expect. For instance, May 31, 2014 at 2:00 PM isn’t between ‘01-May-2014’ and ‘31-May-2014’.
- To learn more about date comparisons, please see chapter 8.
How to use the LIKE operator

One final operator you can use in a search condition is the LIKE operator, shown in figure 3-15. You use this operator along with the wildcards shown at the top of this figure to specify the string pattern, or mask, that you want to match. The examples in this figure show how this works.

In the first example, the LIKE phrase specifies that all vendors in cities that start with the letters SAN should be included in the query results. Here, the percent sign (%) indicates that any characters can follow these three letters. So San Diego and Santa Ana are both included in the results.

The second example selects all vendors whose vendor name starts with the letters COMPU, followed by any one character, the letters ER, and any characters after that. Two vendor names that match that pattern are Compuserve and Computerworld.

The LIKE operator provides a powerful technique for finding information in a database that can’t be found using any other technique.
The syntax of the WHERE clause with the LIKE operator

```
WHERE match_expression [NOT] LIKE pattern
```

**Wildcard symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Matches any string of zero or more characters.</td>
</tr>
<tr>
<td>_</td>
<td>Matches any single character.</td>
</tr>
</tbody>
</table>

**WHERE clauses that use the LIKE operator**

<table>
<thead>
<tr>
<th>Example</th>
<th>Results that match the mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHERE vendor_city LIKE 'SAN%'</td>
<td>“San Diego” and “Santa Ana”</td>
</tr>
<tr>
<td>WHERE vendor_name LIKE 'COMPU_ER%'</td>
<td>“Compuserve” and “Computerworld”</td>
</tr>
</tbody>
</table>

**Description**

- You use the LIKE operator to retrieve rows that match a string pattern, called a mask. Within the mask, you can use special characters, called wildcard characters, that determine which values in the column satisfy the condition.
- You can use the NOT operator before the LIKE operator. Then, only those rows with values that don’t match the string pattern will be included in the result set.
How to use the IS NULL condition

In chapter 1, you learned that a column can contain a null value. A null isn’t the same as zero, a blank string that contains one or more spaces (’ ‘), or an empty string that doesn’t contain any spaces (’’). Instead, a null value indicates that the data is not applicable, not available, or unknown. When you allow null values in one or more columns, you need to know how to test for them in search conditions. To do that, you can use the IS NULL condition, as shown in figure 3-16.

This figure uses a table named null_sample to illustrate how to search for null values. This table contains two columns. The first column, invoice_id, is an identification column. The second column, invoice_total, contains the total for the invoice, which can be a null value. As you can see in the first example, the invoice with an invoice_id of 3 contains a null value.

The second example in this figure shows what happens when you retrieve all the invoices with invoice totals equal to zero. Notice that the row with a null invoice total isn’t included in the result set. Likewise, it isn’t included in the result set that contains all the invoices with invoices totals that aren’t equal to zero, as illustrated by the third example. Instead, you have to use the IS NULL condition to retrieve rows with null values, as shown in the fourth example.

You can also use the NOT operator with the IS NULL condition as illustrated in the last example in this figure. When you use this operator, all of the rows that don’t contain null values are included in the query results.
The syntax of the WHERE clause with the IS null condition

\[
\text{WHERE expression IS [NOT] NULL}
\]

The contents of the Null_Sample table

```
SELECT * 
FROM null_sample
```

<table>
<thead>
<tr>
<th>INVOICE_ID</th>
<th>INVOICE_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>125</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>(null)</td>
</tr>
<tr>
<td>4</td>
<td>2199.99</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

A SELECT statement that retrieves rows with zero values

```
SELECT * 
FROM null_sample
WHERE invoice_total = 0
```

<table>
<thead>
<tr>
<th>INVOICE_ID</th>
<th>INVOICE_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

A SELECT statement that retrieves rows with non-zero values

```
SELECT * 
FROM null_sample
WHERE invoice_total <> 0
```

<table>
<thead>
<tr>
<th>INVOICE_ID</th>
<th>INVOICE_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>125</td>
</tr>
<tr>
<td>2</td>
<td>2199.99</td>
</tr>
</tbody>
</table>

A SELECT statement that retrieves rows with null values

```
SELECT * 
FROM null_sample
WHERE invoice_total IS NULL
```

<table>
<thead>
<tr>
<th>INVOICE_ID</th>
<th>INVOICE_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>(null)</td>
</tr>
</tbody>
</table>

A SELECT statement that retrieves rows without null values

```
SELECT * 
FROM null_sample
WHERE invoice_total IS NOT NULL
```

<table>
<thead>
<tr>
<th>INVOICE_ID</th>
<th>INVOICE_TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>125</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2199.99</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3-16 How to use the IS NULL condition
How to code the ORDER BY clause

The ORDER BY clause specifies the sort order for the rows in a result set. In most cases, you can use column names from the base table to specify the sort order as you saw in some of the examples earlier in this chapter. However, you can also use other techniques to sort the rows in a result set, as described in the topics that follow.

How to sort a result set by a column name

Figure 3-17 presents the expanded syntax of the ORDER BY clause. As you can see, you can sort by one or more expressions in either ascending or descending sequence. This is illustrated by the three examples in this figure.

The first two examples show how to sort the rows in a result set by a single column. In the first example, the rows in the vendors table are sorted in ascending sequence by the vendor_name column. Since ascending is the default sequence, the ASC keyword is omitted. In the second example, the rows are sorted by the vendor_name column in descending sequence.

To sort by more than one column, you simply list the names in the ORDER BY clause separated by commas as shown in the third example. Here, the rows in the Vendors table are first sorted by the vendor_state column in ascending sequence. Then, within each state, the rows are sorted by the vendor_city column in ascending sequence. Finally, within each city, the rows are sorted by the vendor_name column in ascending sequence. This can be referred to as a nested sort because one sort is nested within another.

Although all of the columns in this example are sorted in ascending sequence, you should know that doesn’t have to be the case. For example, we could have sorted by the vendor_name column in descending sequence like this:

ORDER BY vendor_state, vendor_city, vendor_name DESC

Note that the DESC keyword in this example applies only to the vendor_name column. The vendor_state and vendor_city columns are still sorted in ascending sequence.

If you study the first example in this figure, you can see that capital letters come before lowercase letters in an ascending sort. As a result, “ASC Signs” comes before “Abbey Office Furnishings” in the result set. For some business applications, this is acceptable. But if it isn’t, you can use the LOWER function to convert the column to lowercase letters in the ORDER BY clause like this:

ORDER BY LOWER(vendor_name)

Then, the rows will be sorted in the correct alphabetical sequence. In chapter 8, you can learn more about this function.
The expanded syntax of the ORDER BY clause

```
ORDER BY expression [ASC|DESC] [, expression [ASC|DESC]] ...
```

An ORDER BY clause that sorts by one column in ascending sequence

```
SELECT vendor_name,
       vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
ORDER BY vendor_name
```

An ORDER BY clause that sorts by one column in descending sequence

```
SELECT vendor_name,
       vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
ORDER BY vendor_name DESC
```

An ORDER BY clause that sorts by three columns

```
SELECT vendor_name,
       vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
ORDER BY vendor_state, vendor_city, vendor_name
```

Description

- The ORDER BY clause specifies how you want the rows in the result set sorted. You can sort by one or more columns, and you can sort each column in either ascending (ASC) or descending (DESC) sequence. ASC is the default.
- By default, in an ascending sort, special characters appear first in the sort sequence, followed by numbers, then by capital letters, then by lowercase letters, and then by null values. In a descending sort, this sequence is reversed.
- With one exception, you can sort by any column in the base table, regardless of whether it’s included in the SELECT clause. The exception is if the query includes the DISTINCT keyword. Then, you can only sort by columns included in the SELECT clause.

Figure 3-17 How to sort a result set by a column name
How to sort a result set by an alias, an expression, or a column number

Figure 3-18 presents three more techniques that you can use to specify sort columns. First, you can use a column alias that’s defined in the SELECT clause. The first SELECT statement in this figure, for example, sorts by a column named address, which is an alias for the concatenation of the vendor_city, vendor_state, and vendor_zip_code columns. Within the address column, the result set is sorted by the vendor_name column.

You can also use an arithmetic or string expression in the ORDER BY clause, as illustrated by the second example in this figure. Here, the expression consists of the vendor_contact_last_name column concatenated with the vendor_contact_first_name column. Here, neither of these columns is included in the SELECT clause.

The last example in this figure shows how you can use column numbers to specify a sort order. To use this technique, you code the number that corresponds to the column of the result set, where 1 is the first column, 2 is the second column, and so on. In this example, the ORDER BY clause sorts the result set by the second column, which contains the concatenated address, then by the first column, which contains the vendor name. The result set returned by this statement is the same as the result set returned by the first statement.

Notice, however, that the statement that uses column numbers is more difficult to read because you have to look at the SELECT clause to see what columns the numbers refer to. In addition, if you add or remove columns from the SELECT clause, you may also have to change the ORDER BY clause to reflect the new column positions. As a result, we don’t recommend this coding technique.
An ORDER BY clause that uses an alias

```
SELECT vendor_name,
    vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
ORDER BY address, vendor_name
```

An ORDER BY clause that uses an expression

```
SELECT vendor_name,
    vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
ORDER BY vendor_contact_last_name || vendor_contact_first_name
```

An ORDER BY clause that uses column positions

```
SELECT vendor_name,
    vendor_city || ', ' || vendor_state || ' ' || vendor_zip_code AS address
FROM vendors
ORDER BY 2, 1
```

Description

- The ORDER BY clause can include a column alias that’s specified in the SELECT clause.
- The ORDER BY clause can include any valid expression. The expression can refer to any column in the base table, even if it isn’t included in the result set.
- The ORDER BY clause can use numbers to specify the columns to use for sorting. In that case, 1 represents the first column in the result set, 2 represents the second column, and so on.
Section 2  The essential SQL skills

How to code the row limiting clause

With Oracle 12c and later, you can use the OFFSET and FETCH clauses to limit the number of rows that are returned in the result set and to set the starting point for the rows that are returned. For most queries, you want to retrieve the entire result set. As a result, you typically don’t need this clause. However, there may be times when you want to use this clause to retrieve a subset of a larger result set as shown in figure 3-19.

How to limit the number of rows

In the simplest form of the row limiting clause, you omit the optional OFFSET clause and code the FETCH clause. Then, the number of rows in the result set is, at most, the number you specify. However, if the result set is smaller than the number you specify, the FETCH clause has no effect.

The SELECT statement in the first example includes the FETCH FIRST 5 ROWS ONLY clause. As a result, the entire result set is five rows. Without the FETCH clause, this statement would return 114 rows. Because the result set is sorted by invoice total in descending order, this result set represents the five largest invoices.

If necessary, you can also use the PERCENT keyword to limit the number of rows by percentage. In the first example, for instance, you could code FETCH FIRST 30 PERCENT ROWS ONLY to retrieve the first 30% of rows. Because the result set is sorted by invoice total in descending sequence, this result set would represent the top 30% of largest invoices.

Similarly, you can use the WITH TIES keywords to include rows where the sort value is the same as the sort value for the last row in the result set. In the first example, for instance, the result set is sorted by the invoice_total column. As a result, you could code FETCH NEXT 5 ROWS ONLY WITH TIES to add any other rows that have an invoice total of 20551.18 to the result set. This would cause the result set to return more than 5 rows.

How to return a range of rows

If you code the optional OFFSET clause, it represents an offset, or starting point for the result set. This offset starts from a value of 0, which refers to the first row in the result set. In the second example, then, the offset is 2 so the result set starts with the third invoice. Then, the FETCH clause limits the result set to 3 rows.

The third example has an offset of 100, so the result set starts with row 101. Then, the FETCH clause limits the result set to 1000 rows. Since the table contains only 114 rows, though, the result set contains just the last 14 rows in the table.

If you want to retrieve all rows from the offset to the end of the result set, you can omit the FETCH clause. In the third example, for instance, you could code OFFSET 100 ROWS to retrieve the same result set.
Chapter 3  
How to retrieve data from a single table

The syntax of the row limiting clause

\[ \text{OFFSET offset \{ ROW | ROWS \}} \]
\[ \text{FETCH \{ FIRST | NEXT \}\{ \{ rowcount \{ percent PERCENT \}}\}} \]
\{ ROW | ROWS \}\{ ONLY | WITH TIES \}\]

A FETCH clause that retrieves the first five rows

```sql
SELECT vendor_id, invoice_total
FROM invoices
ORDER BY invoice_total DESC
FETCH FIRST 5 ROWS ONLY
```

An OFFSET clause that starts with the third row and fetches three rows

```sql
SELECT invoice_id, vendor_id, invoice_total
FROM invoices
ORDER BY invoice_id
OFFSET 2 ROWS FETCH NEXT 3 ROWS ONLY
```

An OFFSET clause that starts with the 101st row

```sql
SELECT invoice_id, vendor_id, invoice_total
FROM invoices
ORDER BY invoice_id
OFFSET 100 ROWS FETCH NEXT 1000 ROWS ONLY
```

(14 rows)

Description

- Typically, you use an ORDER BY clause to sort the result set before you apply the row limiting clause.
- You can use the FETCH clause to limit the number of rows returned by the SELECT statement.
- You can use the OFFSET clause to specify the first row to return, where the first row is 0, the second row is 1, and so on.
- You can use the PERCENT keyword to fetch the specified percentage of rows.
- You can use the WITH TIES keyword to include additional rows if the sort value is the same as the sort value for the last row in the result set.
Perspective

The goal of this chapter has been to teach you the basic skills for coding SELECT statements. You’ll use these skills in almost every SELECT statement you code. As you’ll see in the chapters that follow, however, there’s a lot more to coding SELECT statements than what’s presented here. In the next three chapters, then, you’ll learn additional skills for coding SELECT statements.

Terms

<table>
<thead>
<tr>
<th>base table</th>
<th>order of precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyword</td>
<td>function</td>
</tr>
<tr>
<td>filter</td>
<td>parameter</td>
</tr>
<tr>
<td>Boolean expression</td>
<td>date literal</td>
</tr>
<tr>
<td>predicate</td>
<td>comparison operator</td>
</tr>
<tr>
<td>expression</td>
<td>logical operator</td>
</tr>
<tr>
<td>column alias</td>
<td>compound condition</td>
</tr>
<tr>
<td>string expression</td>
<td>pseudo column</td>
</tr>
<tr>
<td>concatenate</td>
<td>subquery</td>
</tr>
<tr>
<td>concatenation operator</td>
<td>string pattern</td>
</tr>
<tr>
<td>literal value</td>
<td>mask</td>
</tr>
<tr>
<td>string literal</td>
<td>wildcard</td>
</tr>
<tr>
<td>string constant</td>
<td>null value</td>
</tr>
<tr>
<td>arithmetic expression</td>
<td>nested sort</td>
</tr>
<tr>
<td>arithmetic operator</td>
<td></td>
</tr>
</tbody>
</table>

Exercises

Run some of the examples in this chapter

In these exercises, you’ll use Oracle SQL Developer to run some of the scripts for the examples in this chapter. This assumes that you already know how to use SQL Developer, as described in chapter 2.

1. Start Oracle SQL Developer.
2. Open the script for fig3-02a that you should find in this directory:
   c:\murach\oracle_sql\scripts\ch03.
   Then, press the F9 key or click on the Execute Statement button to run the script. This shows you the data that’s in the Invoices table that you’ll be working with in this chapter.
3. Open and run the script for fig3-02b.
4. Open and run the scripts for any of the other examples in this chapter that you’re interested in reviewing.
Enter and run your own SELECT statements

In these exercises, you’ll enter and run your own SELECT statements. To do that, you can open the script for an example that is similar to the statement you need to write, copy the statement into a new Worksheet window, and then modify the statement. That can save you both time and syntax errors.

5. Write a SELECT statement that returns three columns from the Vendors table: vendor_name, vendor_contact_last_name, and vendor_contact_first_name.

Then, run this statement.

Next, add code to this statement so it sorts the result set by last name and then first name. Then, run this statement again. This is a good way to build and test a statement, one clause at a time.

6. Write a SELECT statement that returns one column from the Vendors table named full_name. Create this column from the vendor_contact_first_name and vendor_contact_last_name columns, and format it like this: last name, comma, space, first name (for example, “Doe, John”). Next, sort the result set by last name and then first name. Then, filter the result set for contacts whose last name begins with the letter A, B, C, or E.

7. Write a SELECT statement that returns four columns from the Invoices table named Due Date, Invoice Total, 10%, and Plus 10%. These columns should contain this data:

<table>
<thead>
<tr>
<th>Due Date</th>
<th>The invoice_due_date column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invoice Total</td>
<td>The invoice_total column</td>
</tr>
<tr>
<td>10%</td>
<td>10% of the value of invoice_total</td>
</tr>
<tr>
<td>Plus 10%</td>
<td>The value of invoice_total plus 10%</td>
</tr>
</tbody>
</table>

(For example, if invoice_total is 100, 10% is 10, and Plus 10% is 110.) Next, filter the result set so it returns only those rows with an invoice total that’s greater than or equal to 500 and less than or equal to 1000. Then, sort the result set in descending sequence by invoice_due_date.

8. Write and run a SELECT statement that returns four columns from the Invoices table named Number, Total, Credits, and Balance Due. These columns should include this data:

<table>
<thead>
<tr>
<th>Number</th>
<th>The invoice_number column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>The invoice_total column</td>
</tr>
<tr>
<td>Credits</td>
<td>Sum of the payment_total and credit_total columns</td>
</tr>
<tr>
<td>Balance Due</td>
<td>Invoice_total minus the sum of payment_total and credit_total</td>
</tr>
</tbody>
</table>
Next, filter for invoices with a balance due that’s greater than or equal to $500. Then, sort the result set by balance due in descending sequence. Last, use the ROWNUM pseudo column so the result set contains only the rows with the 10 largest balance dues.

**Work with nulls and use the Dual table**

9. Write a SELECT statement that returns the balance due and the payment date from the Invoices table, but only when the payment_date column contains a null value. Then, modify the WHERE clause so it returns any invalid rows (rows in which the balance due is zero and the payment date is null).

10. Use the Dual table to create a row with these columns:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starting Principal</td>
<td>Starting principle which should be equal to $51,000</td>
</tr>
<tr>
<td>New Principal</td>
<td>Starting principal plus a 10% increase</td>
</tr>
<tr>
<td>Interest</td>
<td>6.5% of the new principal</td>
</tr>
<tr>
<td>Principal + Interest</td>
<td>The new principal plus the interest (add the expression you used for the new principal calculation to the expression you used for the interest calculation)</td>
</tr>
</tbody>
</table>

Now, add a column named “System Date” that uses the TO_CHAR function to show the results of the SYSDATE function when it’s displayed with this format:

`'dd-mon-yyyy hh24:mi:ss'`

This format will display the day, month, year, hours, minutes, and seconds of the system date, and this will show that the system date also includes a time. (You should be able to figure out how to use the TO_CHAR and SYSDATE functions by studying figure 3-7.)
How to build your Oracle SQL skills

The easiest way is to let Murach’s Oracle SQL and PL/SQL for Developers (2nd Edition) be your guide! So if you’ve enjoyed this chapter, I hope you’ll get your own copy of the book today. You can use it to:

- Teach yourself how to code SQL statements to retrieve and maintain the data in an Oracle database
- Design and create your own Oracle databases using SQL DDL statements
- Use Oracle’s procedural language, PL/SQL, to take advantage of powerful features like stored procedures, functions, and triggers
- Pick up new skills whenever you want to or need to by focusing on material that’s new to you
- Look up coding details or refresh your memory on forgotten details when you’re in the middle of developing a database application
- Loan to your colleagues who are always asking you questions about Oracle SQL coding

To get your copy, you can order online at www.murach.com or call us at 1-800-221-5528 (toll-free in the U.S. and Canada). And remember, when you order directly from us, this book comes with my personal guarantee:

100% Guarantee

You must be satisfied. Each book you buy directly from us must outperform any competing book or course you’ve ever tried, or send it back within 60 days for a full refund…no questions asked.

Thanks for your interest in Murach books!