## ****CHAPTER 6**** ****Subqueries****

All the queries you’ve seen so far in this book have contained just one SELECT statement. In this chapter, you will learn the following:

Image How to place an inner SELECT statement within an outer SELECT, UPDATE, or DELETE statement. The inner SELECT is known as a subquery.

Image The features of the different types of subqueries.

Image How subqueries allow you to build up very complex statements from simple components.

### ****TYPES OF SUBQUERIES****

There are two basic types of subqueries:

Image **Single-row subqueries** return zero rows or one row to the outer SQL statement. There is a special case of a single-row subquery that contains exactly one column; this type of subquery is called a scalar subquery.

Image **Multiple-row subqueries** return one or more rows to the outer SQL statement.

In addition, there are three subtypes of subqueries that may return single or multiple rows:

Image **Multiple-column subqueries** return more than one column to the outer SQL statement.

Image **Correlated subqueries** reference one or more columns in the outer SQL statement. These are called "correlated" subqueries because they are related to the outer SQL statement through the same columns.

Image **Nested subqueries** are placed within another subquery. You can nest subqueries to a depth of 255.

You’ll learn about each of these types of subqueries and how to add them to SELECT, UPDATE, and DELETEstatements. Let’s plunge in and look at how to write single-row subqueries.

### ****WRITING SINGLE-ROW SUBQUERIES****

A single-row subquery is one that returns either zero rows or one row to the outer SQL statement. As you’ll see in this section, you may place a subquery in a WHERE clause, a HAVING clause, or a FROM clause of a SELECTstatement. You’ll also see some errors you might encounter when running subqueries.

#### ****Subqueries in a WHERE Clause****

You may place a subquery in the WHERE clause of another query. The following query contains a subquery placed in its WHERE clause; notice that the subquery is placed within parentheses (…):

**SELECT first\_name, last\_name**

**FROM customers**

**WHERE customer\_id =**

**(SELECT customer\_id**

**FROM customers**

**WHERE last\_name = 'Brown');**

FIRST\_NAME LAST\_NAME

---------- ----------

John Brown

This example retrieves the first\_name and last\_name of the row from the customers table whose last\_name is Brown. Let’s break this query down and analyze what’s going on. The subquery in the WHEREclause is

SELECT customer\_id

FROM customers

WHERE last\_name = 'Brown'

This subquery is executed first (and only once) and returns the customer\_id for the row whose last\_nameis Brown. The customer\_id for this row is 1, which is passed to the WHERE clause of the outer query. Therefore, the outer query returns the same result as the following query:

SELECT first\_name, last\_name

FROM customers

WHERE customer\_id = 1;

#### ****Using Other Single-Row Operators****

The subquery example shown at the start of the previous section used the equality operator (=) in the WHEREclause. You may also use other comparison operators, such as <>, <, >, <=, and >=, with a single-row subquery. The following example uses > in the outer query’s WHERE clause; the subquery uses the AVG() function to get the average price of the products, which is passed to the WHERE clause of the outer query. The entire query returns the product\_id, name, and price of products whose price is greater than that average price.

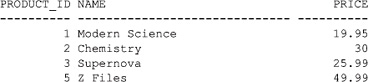
**SELECT product\_id, name, price**

**FROM products**

**WHERE price >**

**(SELECT AVG(price)**

**FROM products);**



Let’s break the example down to understand how it works. The following example shows the subquery run on its own:

**SELECT AVG(price)**

**FROM products;**

AVG(PRICE)

----------

19.7308333

Image

**NOTE**  
This subquery is an example of a scalar subquery, because it returns exactly one row containing one column. The value returned by a scalar subquery is treated as a single scalar value.

The value 19.7308333 returned by the subquery is used in the WHERE clause of the outer query, which is therefore equivalent to the following:

SELECT product\_id, name, price

FROM products

WHERE price > 19.7308333;

#### ****Subqueries in a HAVING Clause****

As you saw in [Chapter 4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#ch04), you use the HAVING clause to filter groups of rows. You may place a subquery in the HAVING clause of an outer query. Doing this allows you to filter groups of rows based on the result returned by your subquery.

The following example uses a subquery in the HAVING clause of the outer query. The example retrieves the product\_type\_id and the average price for products whose average price is less than the maximum of the average for the groups of the same product type:

**SELECT product\_type\_id, AVG(price)**

**FROM products**

**GROUP BY product\_type\_id**

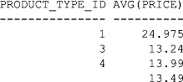
**HAVING AVG(price) <**

**(SELECT MAX(AVG(price))**

**FROM products**

**GROUP BY product\_type\_id)**

**ORDER BY product\_type\_id;**



Notice the subquery uses AVG() to first compute the average price for each product type. The result returned by AVG() is then passed to MAX(), which returns the maximum of the averages.

Let’s break the example down to understand how it works. The following example shows the output from the subquery when it is run on its own:

**SELECT MAX(AVG(price))**

**FROM products**

**GROUP BY product\_type\_id;**

MAX(AVG(PRICE))

---------------

26.22

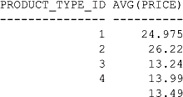
This value of 26.22 is used in the HAVING clause of the outer query to filter the group’s rows to those having an average price less than 26.22. The following query shows a version of the outer query that retrieves the product\_type\_id and average price of the products grouped by product\_type\_id:

**SELECT product\_type\_id, AVG(price)**

**FROM products**

**GROUP BY product\_type\_id**

**ORDER BY product\_type\_id;**



The groups with a product\_type\_id of 1, 3, 4, and null have an average price less than 26.22. As expected, these are the same groups returned by the query at the start of this section.

#### ****Subqueries in a FROM Clause (Inline Views)****

You may place a subquery in the FROM clause of an outer query. These types of subqueries are also known as inline views, because the subquery provides data in line with the FROM clause. The following simple example retrieves the products whose product\_id is less than 3:

**SELECT product\_id**

**FROM**

**(SELECT product\_id**

**FROM products**

**WHERE product\_id < 3);**

PRODUCT\_ID

----------

1

2

Notice that the subquery returns the rows from the products table whose product\_id is less than 3 to the outer query, which then retrieves and displays those product\_id values. As far as the FROM clause of the outer query is concerned, the output from the subquery is just another source of data.

The next example is more useful and retrieves the product\_id and price from the products table in the outer query, and the subquery retrieves the number of times a product has been purchased:

**SELECT prds.product\_id, price, purchases\_data.product\_count**

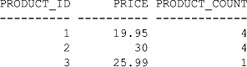
**FROM products prds,**

**(SELECT product\_id, COUNT(product\_id) product\_count**

**FROM purchases**

**GROUP BY product\_id) purchases\_data**

**WHERE prds.product\_id = purchases\_data.product\_id;**



Notice that the subquery retrieves the product\_id and COUNT(product\_id) from the purchases table and returns them to the outer query. As you can see, the output from subquery is just another source of data to the FROM clause of the outer query.

#### ****Errors You Might Encounter****

In this section, you’ll see some errors you might encounter. Specifically, you’ll see that a single-row subquery may return a maximum of one row and that a subquery may not contain an ORDER BY clause.

##### ****Single-Row Subqueries May Return a Maximum of One Row****

If your subquery returns more than one row, you’ll get the following error:

ORA-01427: single-row subquery returns more than one row.

For example, the subquery in the following statement attempts to pass multiple rows to the equality operator (=) in the outer query:

SQL> **SELECT product\_id, name**

**2 FROM products**

**3 WHERE product\_id =**

**4 (SELECT product\_id**

**5 FROM products**

**6 WHERE name LIKE '%e%');**

(SELECT product\_id

\*

ERROR at line 4:

ORA-01427: single-row subquery returns more than one row

There are nine rows in the products table whose names contain the letter e, and the subquery attempts to pass these rows to the equality operator in the outer query. Because the equality operator can handle only a single row, the query is invalid and an error is returned.

You’ll learn how to return multiple rows from a subquery later in the section "Writing Multiple-Row Subqueries."

##### ****Subqueries May Not Contain an ORDER BY Clause****

A subquery may not contain an ORDER BY clause. Instead, any ordering must be done in the outer query. For example, the following outer query has an ORDER BY clause at the end that sorts the product\_id values in descending order:

**SELECT product\_id, name, price**

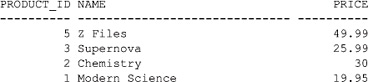
**FROM products**

**WHERE price >**

**(SELECT AVG(price)**

**FROM products)**

**ORDER BY product\_id DESC;**



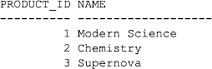
### ****WRITING MULTIPLE-ROW SUBQUERIES****

You use a multiple-row subquery to return one or more rows to an outer SQL statement. To handle a subquery that returns multiple rows, your outer query may use the IN, ANY, or ALL operator. As you saw in [Chapter 2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch02.html#ch02), you can use these operators to see if a column value is contained in a list of values; for example:

**SELECT product\_id, name**

**FROM products**

**WHERE product\_id IN (1, 2, 3);**



As you’ll see in this section, the list of values can come from a subquery.

Image

**NOTE**  
You can also use the EXISTS operator to check if a value is in a list returned by a correlated subquery. You’ll learn about this later, in the section "Writing Correlated Subqueries."

#### ****Using IN with a Multiple-Row Subquery****

As you saw in [Chapter 2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch02.html#ch02), you use IN to check if a value is in a specified list of values. The list of values may come from the results returned by a subquery. You can also use NOT IN to perform the logical opposite of IN: to check if a value is not in a specified list of values.

The following simple example uses IN to check if a product\_id is in the list of values returned by the subquery; the subquery returns the product\_id for products whose name contains the letter e:

**SELECT product\_id, name**

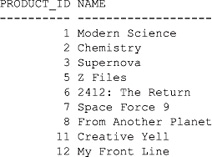
**FROM products**

**WHERE product\_id IN**

**(SELECT product\_id**

**FROM products**

**WHERE name LIKE '%e%');**



The next example uses NOT IN to get the products that are not in the purchases table:

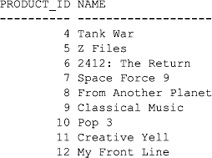
**SELECT product\_id, name**

**FROM products**

**WHERE product\_id NOT IN**

**(SELECT product\_id**

**FROM purchases);**



#### ****Using ANY with a Multiple-Row Subquery****

You use the ANY operator to compare a value with any value in a list. You must place an =, <>, <, >, <=, or >=operator before ANY in your query. The following example uses ANY to get the employees whose salary is less than any of the lowest salaries in the salary\_grades table:

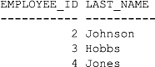
**SELECT employee\_id, last\_name**

**FROM employees**

**WHERE salary < ANY**

**(SELECT low\_salary**

**FROM salary\_grades);**



#### ****Using ALL with a Multiple-Row Subquery****

You use the ALL operator to compare a value with any value in a list. You must place an =, <>, <, >, <=, or >=operator before ALL in your query. The following example uses ALL to get the employees whose salary is greater than all of the highest salaries in the salary\_grades table:

**SELECT employee\_id, last\_name**

**FROM employees**

**WHERE salary > ALL**

**(SELECT high\_salary**

**FROM salary\_grades);**

no rows selected

As you can see, no employee has a salary greater than the highest salary.

### ****WRITING MULTIPLE-COLUMN SUBQUERIES****

The subqueries you’ve seen so far have returned rows containing one column. You’re not limited to one column: you can write subqueries that return multiple columns. The following example retrieves the products with the lowest price for each product type group:

**SELECT product\_id, product\_type\_id, name, price**

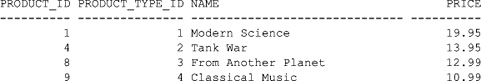
**FROM products**

**WHERE (product\_type\_id, price) IN**

**(SELECT product\_type\_id, MIN(price)**

**FROM products outer**

**GROUP BY product\_type\_id);**



Notice that the subquery returns the product\_type\_id and the minimum price for each group of products, and these are compared in the outer query’s WHERE clause with the product\_type\_id and price for each product.

### ****WRITING CORRELATED SUBQUERIES****

A correlated subquery references one or more columns in the outer SQL statement. These are called correlatedsubqueries, because they are related to the outer SQL statement through the same columns.

You typically use a correlated subquery when you need an answer to a question that depends on a value in each row contained in an outer query. For example, you might want to see whether there is a relationship between the data, but you don’t care how many rows are returned by the subquery, that is, you just want to check whether anyrows are returned, but you don’t care how many.

A correlated subquery is run once for each row in the outer query; this is different from a non-correlated subquery, which is run once prior to running the outer query. In addition, a correlated subquery can resolve null values. You’ll see examples in the following sections that illustrate these concepts.

#### ****A Correlated Subquery Example****

The following correlated subquery retrieves the products that have a price greater than the average for their product type:

**SELECT product\_id, product\_type\_id, name, price**

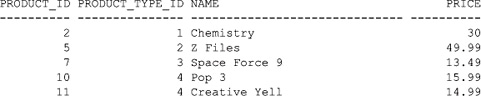
**FROM products outer**

**WHERE price >**

**(SELECT AVG(price)**

**FROM products inner**

**WHERE inner.product\_type\_id = outer.product\_type\_id);**



Notice that I’ve used the alias outer to label the outer query and the alias inner for the inner subquery. The reference to the product\_type\_id column in both the inner and outer parts is what makes the inner subquery correlated with the outer query. Also, the subquery returns a single row containing the average price for the product.

In a correlated subquery, each row in the outer query is passed one at a time to the subquery. The subquery reads each row in turn from the outer query and applies it to the subquery until all the rows from the outer query have been processed. The results from the entire query are then returned.

In the previous example, the outer query retrieves each row from the products table and passes them to the inner query. Each row is read by the inner query, which calculates the average price for each product where the product\_type\_id in the inner query is equal to the product\_type\_id in the outer query.

#### ****Using EXISTS and NOT EXISTS with a Correlated Subquery****

You use the EXISTS operator to check for the existence of rows returned by a subquery. Although you can use EXISTS with non-correlated subqueries, you’ll typically use it with correlated subqueries. The NOT EXISTSoperator does the logical opposite of EXISTS: it checks if rows do not exist in the results returned by a subquery.

##### ****Using EXISTS with a Correlated Subquery****

The following example uses EXISTS to retrieve employees who manage other employees; notice that I don’t care how many rows are returned by the subquery; I only care whether any rows are returned at all:

**SELECT employee\_id, last\_name**

**FROM employees outer**

**WHERE EXISTS**

**(SELECT employee\_id**

**FROM employees inner**

**WHERE inner.manager\_id = outer.employee\_id);**

Image

Because EXISTS just checks for the existence of rows returned by the subquery, a subquery doesn’t have to return a column—it can just return a literal value. This feature can improve the performance of your query. For example, the following query rewrites the previous example with the subquery returning the literal value 1:

**SELECT employee\_id, last\_name**

**FROM employees outer**

**WHERE EXISTS**

**(SELECT 1**

**FROM employees inner**

**WHERE inner.manager\_id = outer.employee\_id);**

Image

As long as the subquery returns one or more rows, EXISTS returns true; if the subquery returns no rows, EXISTS returns false. In the examples, I didn’t care how many rows are returned by the subquery: All I cared about was whether any rows (or no rows) are returned, so that EXISTS returns true (or false). Because the outer query requires at least one column, the literal value 1 is returned by the subquery in the previous example.

##### ****Using NOT EXISTS with a Correlated Subquery****

The following example uses NOT EXISTS to retrieve products that haven’t been purchased:

**SELECT product\_id, name**

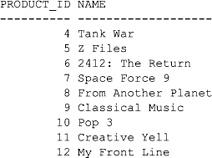
**FROM products outer**

**WHERE NOT EXISTS**

**(SELECT 1**

**FROM purchases inner**

**WHERE inner.product\_id = outer.product\_id);**



##### ****EXISTS and NOT EXISTS Versus IN and NOT IN****

Earlier in the section "Using IN with a Multiple-Row Subquery," you saw how the IN operator is used to check if a value is contained in a list. EXISTS is different from IN: EXISTS checks just for the existence of rows, whereas IN checks for actual values.

Image

**TIP**  
EXISTS typically offers better performance than IN with subqueries. Therefore, you should useEXISTS rather than IN wherever possible.

You should be careful when writing queries that use NOT EXISTS or NOT IN. When a list of values contains a null value, NOT EXISTS returns true, but NOT IN returns false. Consider the following example that uses NOT EXISTS and retrieves the product types that don’t have any products of that type in the products table:

**SELECT product\_type\_id, name**

**FROM product\_types outer**

**WHERE NOT EXISTS**

**(SELECT 1**

**FROM products inner**

**WHERE inner.product\_type\_id = outer.product\_type\_id);**

Image

Notice one row is returned by this example. The next example rewrites the previous query to use NOT IN; notice that no rows are returned:

**SELECT product\_type\_id, name**

**FROM product\_types**

**WHERE product\_type\_id NOT IN**

**(SELECT product\_type\_id**

**FROM products);**

no rows selected

No rows are returned because the subquery returns a list of product\_id values, one of which is null (the product\_type\_id for product #12 is null). Because of this, NOT IN in the outer query returns false, and therefore no rows are returned. You can get around this by using the NVL() function to convert nulls to a value. In the following example, NVL() is used to convert null product\_type\_id values to 0:

**SELECT product\_type\_id, name**

**FROM product\_types**

**WHERE product\_type\_id NOT IN**

**(SELECT NVL(product\_type\_id, 0)**

**FROM products);**

Image

This time the row appears.

These examples illustrate another difference between correlated and non-correlated subqueries: a correlated query can resolve null values.

### ****WRITING NESTED SUBQUERIES****

You can nest subqueries inside other subqueries to a depth of 255. You should use this technique sparingly—you may find your query performs better using table joins. The following example contains a nested subquery; notice that it is contained within a subquery, which is itself contained in an outer query:

**SELECT product\_type\_id, AVG(price)**

**FROM products**

**GROUP BY product\_type\_id**

**HAVING AVG(price) <**

**(SELECT MAX(AVG(price))**

**FROM products**

**WHERE product\_type\_id IN**

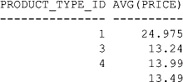
**(SELECT product\_id**

**FROM purchases**

**WHERE quantity > 1)**

**GROUP BY product\_type\_id)**

**ORDER BY product\_type\_id;**



As you can see, this example is quite complex and contains three queries: a nested subquery, a subquery, and the outer query. These query parts are run in that order. Let’s break the example down into the three parts and examine the results returned. The nested subquery is

SELECT product\_id

FROM purchases

WHERE quantity > 1

This subquery returns the product\_id for the products that have been purchased more than once. The rows returned by this subquery are

PRODUCT\_ID

----------

2

1

The subquery that receives this output is

SELECT MAX(AVG(price))

FROM products

WHERE product\_type\_id IN

(… output from the nested subquery …)

GROUP BY product\_type\_id

This subquery returns the maximum average price for the products returned by the nested subquery. The row returned is

MAX(AVG(PRICE))

---------------

26.22

This row is returned to the following outer query:

SELECT product\_type\_id, AVG(price)

FROM products

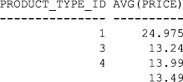
GROUP BY product\_type\_id

HAVING AVG(price) <

(… output from the subquery …)

ORDER BY product\_type\_id;

This query returns the product\_type\_id and average price of products that are less than average returned by the subquery. The rows returned are



These are the rows returned by the complete query shown at the start of this section.

### ****WRITING UPDATE AND DELETE STATEMENTS CONTAINING SUBQUERIES****

So far, you’ve only seen subqueries contained in a SELECT statement. As you’ll see in this section, you can also put subqueries inside UPDATE and DELETE statements.

#### ****Writing an UPDATE Statement Containing a Subquery****

In an UPDATE statement, you can set a column to the result returned by a single-row subquery. For example, the following UPDATE statement sets employee #4’s salary to the average of the high salary grades returned by a subquery:

**UPDATE employees**

**SET salary =**

**(SELECT AVG(high\_salary)**

**FROM salary\_grades)**

**WHERE employee\_id = 4;**

1 row updated.

Doing this increases employee #4’s salary from $500,000 to $625,000 (this is the average of the high salaries from the salary\_grades table).

Image

**NOTE**  
If you execute the UPDATE statement, remember to execute a ROLLBACK to undo the change. That way, your results will match those shown later in this book.

#### ****Writing a DELETE Statement Containing a Subquery****

You can use the rows returned by a subquery in the WHERE clause of a DELETE statement. For example, the following DELETE statement removes the employee whose salary is greater than the average of the high salary grades returned by a subquery:

**DELETE FROM employees**

**WHERE salary >**

**(SELECT AVG(high\_salary)**

**FROM salary\_grades);**

1 row deleted.

This DELETE statement removes employee #1.

Image

**NOTE**  
If you execute the DELETE statement, remember to execute a ROLLBACK to undo the removal of the row.

### ****SUMMARY****

In this chapter, you learned the following:

Image A subquery is a query placed within a SELECT, UPDATE, or DELETE statement.

Image Single-row subqueries return zero or one row.

Image Multiple-row subqueries return one or more rows.

Image Multiple-column subqueries return more than one column.

Image Correlated subqueries reference one or more columns in the outer SQL statement.

Image Nested subqueries are subqueries placed within another subquery.

**CHAPTER 7  
Advanced Queries**

In this chapter, you will see how to

Image Use the set operators, which allow you to combine rows returned by two or more queries.

Image Use the TRANSLATE() function to translate characters in one string to characters in another string.

Image Use the DECODE() function to search for a certain value in a set of values.

Image Use the CASE expression to perform if-then-else logic in SQL.

Image Perform queries on hierarchical data.

Image Use the ROLLUP and CUBE clauses to get subtotals and totals for groups of rows.

Image Take advantage of the analytic functions, which perform complex calculations, such as finding the top-selling product type for each month, the top salespersons, and so on.

Image Perform inter-row calculations with the MODEL clause.

Image Use the new Oracle Database 11*g* PIVOT and UNPIVOT clauses, which are useful for seeing overall trends in large amounts of data.

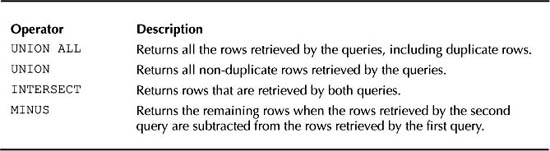
Let’s plunge in and examine the set operators.

**USING THE SET OPERATORS**

The set operators allow you to combine rows returned by two or more queries. [Table 7-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch07.html#table_7-1) shows the four set operators.

You must keep in mind the following restriction when using a set operator: *The number of columns and the column types returned by the queries must match, although the column names may be different*.

You’ll learn how to use each of the set operators shown in [Table 7-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch07.html#table_7-1) shortly, but first let’s look at the example tables used in this section.



**TABLE 7-1** *Set Operators*

**The Example Tables**

The products and more\_products tables are created by the store\_schema.sql script using the following statements:

CREATE TABLE products (

product\_id INTEGER

CONSTRAINT products\_pk PRIMARY KEY,

product\_type\_id INTEGER

CONSTRAINT products\_fk\_product\_types

REFERENCES product\_types(product\_type\_id),

name VARCHAR2(30) NOT NULL,

description VARCHAR2(50),

price NUMBER(5, 2)

);

CREATE TABLE more\_products (

prd\_id INTEGER

CONSTRAINT more\_products\_pk PRIMARY KEY,

prd\_type\_id INTEGER

CONSTRAINT more\_products\_fk\_product\_types

REFERENCES product\_types(product\_type\_id),

name VARCHAR2(30) NOT NULL,

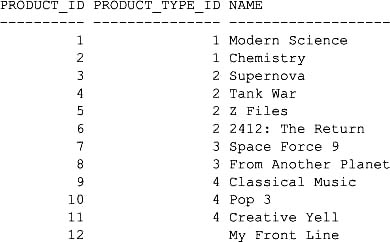
available CHAR(1)

);

The following query retrieves the product\_id, product\_type\_id, and name columns from the products table:

**SELECT product\_id, product\_type\_id, name**

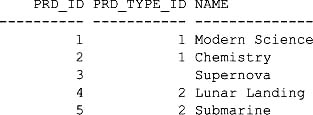
**FROM products;**



The next query retrieves the prd\_id, prd\_type\_id, and name columns from the more\_products table:

**SELECT prd\_id, prd\_type\_id, name**

**FROM more\_products;**



**Using the UNION ALL Operator**

The UNION ALL operator returns all the rows retrieved by the queries, including duplicate rows. The following query uses UNION ALL; notice that all the rows from products and more\_products are retrieved, including duplicates:

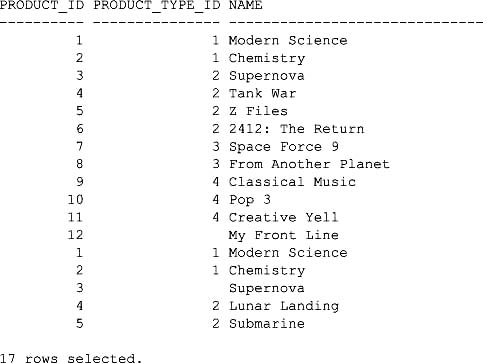
**SELECT product\_id, product\_type\_id, name**

**FROM products**

**UNION ALL**

**SELECT prd\_id, prd\_type\_id, name**

**FROM more\_products;**



You can sort the rows using the ORDER BY clause followed by the position of the column. The following example uses ORDER BY 1 to sort the rows by the first column retrieved by the two queries (product\_idand prd\_id):

**SELECT product\_id, product\_type\_id, name**

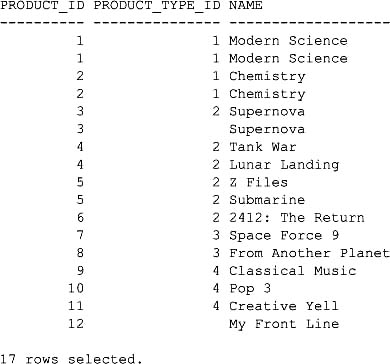
**FROM products**

**UNION ALL**

**SELECT prd\_id, prd\_type\_id, name**

**FROM more\_products**

**ORDER BY 1;**



**Using the UNION Operator**

The UNION operator returns only the non-duplicate rows retrieved by the queries. The following example uses UNION; notice the duplicate "Modern Science" and "Chemistry" rows are not retrieved, and so only 15 rows are returned:

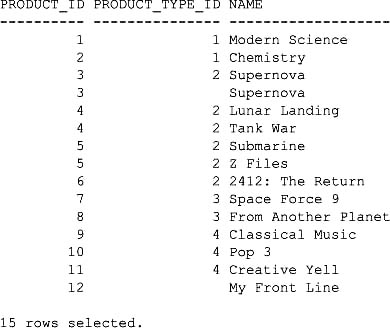
**SELECT product\_id, product\_type\_id, name**

**FROM products**

**UNION**

**SELECT prd\_id, prd\_type\_id, name**

**FROM more\_products;**



**Using the INTERSECT Operator**

The INTERSECT operator returns only rows that are retrieved by both queries. The following example uses INTERSECT; notice that the "Modern Science" and "Chemistry" rows are returned:

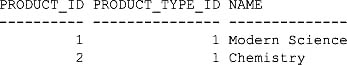
**SELECT product\_id, product\_type\_id, name**

**FROM products**

**INTERSECT**

**SELECT prd\_id, prd\_type\_id, name**

**FROM more\_products;**



**Using the MINUS Operator**

The MINUS operator returns the remaining rows when the rows retrieved by the second query are subtracted from the rows retrieved by the first query. The following example uses MINUS; notice that the rows from more\_products are subtracted from products and the remaining rows are returned:

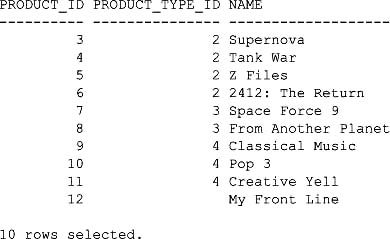
**SELECT product\_id, product\_type\_id, name**

**FROM products**

**MINUS**

**SELECT prd\_id, prd\_type\_id, name**

**FROM more\_products;**



**Combining Set Operators**

You can combine more than two queries with multiple set operators, with the returned results from one operator feeding into the next operator. By default, set operators are evaluated from top to bottom, but you should indicate the order using parentheses in case Oracle Corporation changes this default behavior in future software releases.

In the examples in this section, I’ll use the following product\_changes table (created by the store\_schema.sql script):

CREATE TABLE product\_changes (

product\_id INTEGER

CONSTRAINT prod\_changes\_pk PRIMARY KEY,

product\_type\_id INTEGER

CONSTRAINT prod\_changes\_fk\_product\_types

REFERENCES product\_types(product\_type\_id),

name VARCHAR2(30) NOT NULL,

description VARCHAR2(50),

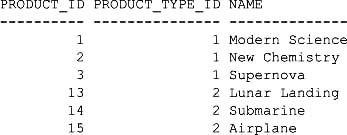
price NUMBER(5, 2)

);

The following query returns the product\_id, product\_type\_id, and name columns from the product\_changes table:

**SELECT product\_id, product\_type\_id, name**

**FROM product\_changes;**



The next query does the following:

Image Uses the UNION operator to combine the results from the products and more\_products tables. (The UNION operator returns only the non-duplicate rows retrieved by the queries.)

Image Uses the INTERSECT operator to combine the results from the previous UNION operator with the results from the product\_changes table. (The INTERSECT operator only returns rows that are retrieved by both queries.)

Image Uses parentheses to indicate the order of evaluation, which is: (1) the UNION between the products and more\_products tables; (2) the INTERSECT.

**(SELECT product\_id, product\_type\_id, name**

**FROM products**

**UNION**

**SELECT prd\_id, prd\_type\_id, name**

**FROM more\_products)**

**INTERSECT**

**SELECT product\_id, product\_type\_id, name**

**FROM product\_changes;**

Image

The following query has the parentheses set so that the INTERSECT is performed first; notice the different results returned by the query compared with the previous example:

**SELECT product\_id, product\_type\_id, name**

**FROM products**

**UNION**

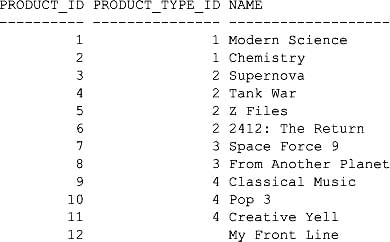
**(SELECT prd\_id, prd\_type\_id, name**

**FROM more\_products**

**INTERSECT**

**SELECT product\_id, product\_type\_id, name**

**FROM product\_changes);**



This concludes the discussion of the set operators.

**USING THE TRANSLATE() FUNCTION**

TRANSLATE (*x*, *from\_string*, *to\_string*) converts the occurrences of characters in *from\_string*found in *x* to corresponding characters in *to\_string*. The easiest way to understand how TRANSLATE()works is to see some examples.

The following example uses TRANSLATE() to shift each character in the string SECRET MESSAGE: MEET ME IN THE PARK by four places to the right: A becomes E, B becomes F, and so on:

**SELECT TRANSLATE(’SECRET MESSAGE: MEET ME IN THE PARK',**

**'ABCDEFGHIJKLMNOPQRSTUVWXYZ',**

**'EFGHIJKLMNOPQRSTUVWXYZABCD')**

**FROM dual;**

Image

The next example takes the output of the previous example and shifts the characters four places to the left: E becomes A, F becomes B, and so on:

**SELECT TRANSLATE('WIGVIX QIWWEKI: QIIX QI MR XLI TEVO',**

**'EFGHIJKLMNOPQRSTUVWXYZABCD',**

**'ABCDEFGHIJKLMNOPQRSTUVWXYZ')**

**FROM dual;**

Image

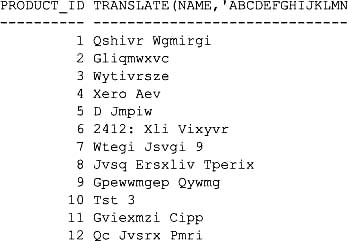
You can of course pass column values to TRANSLATE(). The following example passes the name column from the products table to TRANSLATE(), which shifts the letters in the product name four places to the right:

**SELECT product\_id, TRANSLATE(name**

**'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'**

**'EFGHIJKLMNOPQRSTUVWXYZABCDefghijklmnopqrstuvwxyzabcd')**

**FROM products;**



You can also use TRANSLATE() to convert numbers. The following example takes the number 12345 and converts 5 to 6, 4 to 7, 3 to 8, 2 to 9, and 1 to 0:

**SELECT TRANSLATE(12345,**

**54321,**

**67890)**

**FROM dual;**

TRANS

-----

09876

**USING THE DECODE() FUNCTION**

DECODE(*value*, *search\_value*, *result*, *default\_value*) compares *value* with *search\_value*. If the values are equal, DECODE() returns *result*; otherwise, *default\_value* is returned. DECODE() allows you to perform if-then-else logic in SQL without having to use PL/SQL. Each of the parameters to DECODE() can be a column, a literal value, a function, or a subquery.

Image

**NOTE**  
DECODE() *is an old Oracle proprietary function, and therefore you should use* CASE *expressions instead if you are using Oracle Database 9* i *and above (you will learn about* CASE *in the next section). The* DECODE() *function is mentioned here because you may encounter it when using older Oracle databases*.

The following example illustrates the use of DECODE() with literal values; DECODE() returns 2 (1 is compared with 1, and because they are equal 2 is returned):

**SELECT DECODE(1, 1, 2, 3)**

**FROM dual;**

DECODE(1,1,2,3)

---------------

2

The next example uses DECODE() to compare 1 to 2, and because they are not equal 3 is returned:

**SELECT DECODE(1, 2, 1, 3)**

**FROM dual;**

DECODE(1,2,1,3)

---------------

3

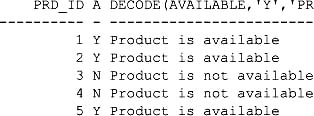
The next example compares the available column in the more\_products table; if available equals Y, the string 'Product is available' is returned; otherwise, 'Product is not available' is returned:

**SELECT prd\_id, available,**

**DECODE(available, 'Y', 'Product is available',**

**'Product is not available')**

**FROM more\_products;**



You can pass multiple search and result parameters to DECODE(), as shown in the following example, which returns the product\_type\_id column as the name of the product type:

**SELECT product\_id, product\_type\_id,**

**DECODE(product\_type\_id,**

**1, 'Book',**

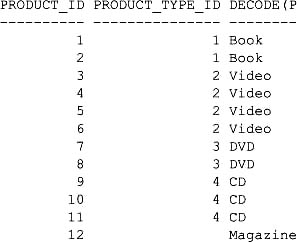
**2, 'Video',**

**3, ’DVD',**

**4, 'CD',**

**'Magazine')**

**FROM products;**



Notice that

Image If product\_type\_id is 1, Book is returned.

Image If product\_type\_id is 2, Video is returned.

Image If product\_type\_id is 3, DVD is returned.

Image If product\_type\_id is 4, CD is returned.

Image If product\_type\_id is any other value, Magazine is returned.

**USING THE CASE EXPRESSION**

The CASE expression performs if-then-else logic in SQL and is supported in Oracle Database 9*i* and above. The CASE expression works in a similar manner to DECODE(), but you should use CASE because it is ANSI-compliant and forms part of the SQL/92 standard. In addition, the CASE expression is easier to read.

There are two types of CASE expressions:

Image Simple case expressions, which use expressions to determine the returned value

Image Searched case expressions, which use conditions to determine the returned value

You’ll learn about both of these types of CASE expressions next.

**USING SIMPLE CASE EXPRESSIONS**

Simple CASE expressions use embedded expressions to determine the value to return. Simple CASE expressions have the following syntax:

CASE *search\_expression*

WHEN *expression1* THEN *result1*

WHEN *expression2* THEN *result2*

…

WHEN *expressionN* THEN *resultN*

ELSE *default\_result*

END

where

Image *search\_expression* is the expression to be evaluated.

Image *expression1, expression2,…, expressionN* are the expressions to be evaluated against *search\_expression*.

Image *result1, result2,…, resultN* are the returned results (one for each possible expression). If *expression1* evaluates to *search\_expression*, *result1* is returned, and similarly for the other expressions.

Image *default\_result* is returned when no matching expression is found.

The following example shows a simple CASE expression that returns the product types as names:

**SELECT product\_id, product\_type\_id,**

**CASE product\_type\_id**

**WHEN 1 THEN 'Book'**

**WHEN 2 THEN 'Video'**

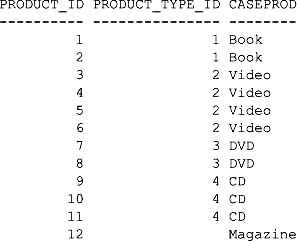
**WHEN 3 THEN ’DVD'**

**WHEN 4 THEN 'CD'**

**ELSE 'Magazine'**

**END**

**FROM products;**



**Using Searched CASE Expressions**

Searched CASE expressions use conditions to determine the returned value. Searched CASE expressions have the following syntax:

CASE

WHEN *condition1* THEN *result1*

WHEN *condition2* THEN *result2*

…

WHEN *conditionN* THEN *resultN*

ELSE *default\_result*

END

where

Image *condition1, condition2,…, conditionN* are the expressions to be evaluated.

Image *result1, result2,…, resultN* are the returned results (one for each possible condition). If *condition1* is true, *result1* is returned, and similarly for the other expressions.

Image *default\_result* is returned when there is no condition that returns true.

The following example illustrates the use of a searched CASE expression:

**SELECT product\_id, product\_type\_id**

**CASE**

**WHEN product\_type\_id = 1 THEN 'Book'**

**WHEN product\_type\_id = 2 THEN 'Video'**

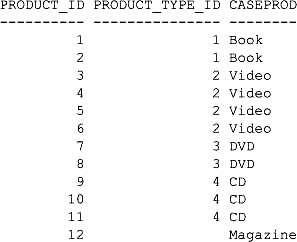
**WHEN product\_type\_id = 3 THEN ’DVD'**

**WHEN product\_type\_id = 4 THEN 'CD'**

**ELSE 'Magazine'**

**END**

**FROM products;**



You can use operators in a searched CASE expression, as shown in the following example:

**SELECT product\_id, price,**

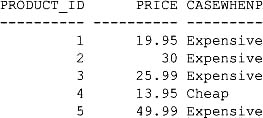
**CASE**

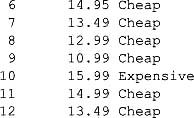
**WHEN price > 15 THEN 'Expensive'**

**ELSE 'Cheap'**

**END**

**FROM products;**





You will see more advanced examples of CASE expressions later in this chapter and in [Chapter 16](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch16.html#ch16).

**HIERARCHICAL QUERIES**

You’ll quite often encounter data that is organized in a hierarchical manner. Examples include the people who work in a company, a family tree, and the parts that make up an engine. In this section, you’ll see queries that access a hierarchy of employees who work for our imaginary store.

**The Example Data**

You’ll see the use of a table named more\_employees, which is created by the store\_schema.sql script as follows:

CREATE TABLE more\_employees (

employee\_id INTEGER

CONSTRAINT more\_employees\_pk PRIMARY KEY,

manager\_id INTEGER

CONSTRAINT more\_empl\_fk\_fk\_more\_empl

REFERENCES more\_employees(employee\_id),

first\_name VARCHAR2(10) NOT NULL,

last\_name VARCHAR2(10) NOT NULL,

title VARCHAR2(20),

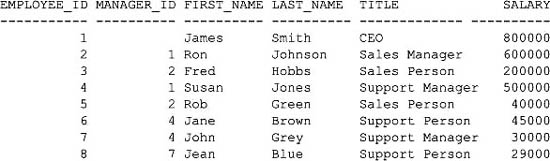
salary NUMBER(6, 0)

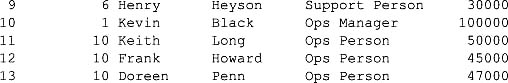
);

The manager\_id column is a self-reference back to the employee\_id column of the more\_employeestable; manager\_id indicates the manager of an employee (if any). The following query returns the rows from more\_employees:

**SELECT \***

**FROM more\_employees;**





As you can see, it’s difficult to pick out the employee relationships from this data. [Figure 7-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch07.html#fig_7-1) shows the relationships in a graphical form.

As you can see from [Figure 7-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch07.html#fig_7-1), the elements—or *nodes*—form a tree. Trees of nodes have the following technical terms associated with them:

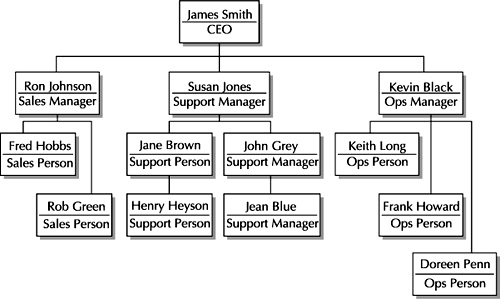
Image **Root node** The root is the node at the top of the tree. In the example shown in [Figure 7-1](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch07.html#fig_7-1), the root node is James Smith, the CEO.

Image **Parent node** A parent is a node that has one or more nodes beneath it. For example, James Smith is the parent to the following nodes: Ron Johnson, Susan Jones, and Kevin Black.

Image **Child node** A child is a node that has one parent node above it. For example, Ron Johnson’s parent is James Smith.

Image **Leaf node** A leaf is a node that has no children. For example, Fred Hobbs and Rob Green are leaf nodes.

You use the CONNECT BY and START WITH clauses of a SELECT statement to perform hierarchical queries, as described next.



**FIGURE 7-1** *Employee relationships*

**Using the CONNECT BY and START WITH Clauses**

The syntax for the CONNECT BY and START WITH clauses of a SELECT statement is

SELECT [LEVEL], *column*, *expression*,…

FROM *table*

[WHERE *where\_clause*]

[[START WITH *start\_condition*] [CONNECT BY PRIOR *prior\_condition*]];

where

Image LEVEL is a pseudo column that tells you how far into a tree you are. LEVEL returns 1 for a root node, 2 for a child of the root, and so on.

Image *start\_condition* specifies where to start the hierarchical query. You must specify a START WITH clause when writing a hierarchical query. An example *start\_condition* is employee\_id = 1, which specifies the query starts from employee #1.

Image *prior\_condition* specifies the relationship between the parent and child rows. You must specify a CONNECT BY PRIOR clause when writing a hierarchical query. An example *prior\_condition* is employee\_id = manager\_id, which specifies the relationship is between the parent employee\_id and the child manager\_id—that is, the child’s manager\_id points to the parent’s employee\_id.

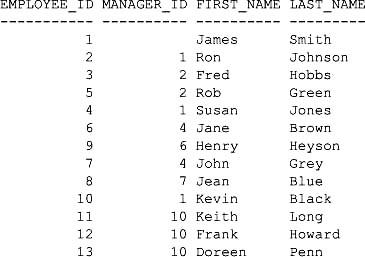
The following query illustrates the use of the START WITH and CONNECT BY PRIOR clauses; notice that the first row contains the details of James Smith (employee #1), the second row contains the details of Ron Johnson, whose manager\_id is 1, and so on:

**SELECT employee\_id, manager\_id, first\_name, last\_name**

**FROM more\_employees**

**START WITH employee\_id = 1**

**CONNECT BY PRIOR employee\_id = manager\_id;**



**Using the LEVEL Pseudo Column**

The next query illustrates the use of the LEVEL pseudo column to display the level in the tree:

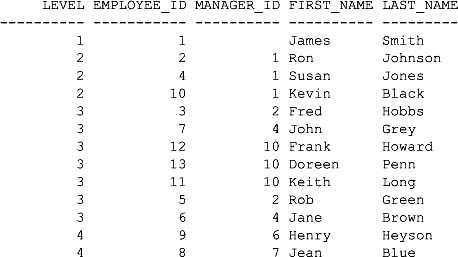
**SELECT LEVEL, employee\_id, manager\_id, first\_name, last\_name**

**FROM more\_employees**

**START WITH employee\_id = 1**

**CONNECT BY PRIOR employee\_id = manager\_id**

**ORDER BY LEVEL;**



The next query uses the COUNT() function and LEVEL to get the number of levels in the tree:

**SELECT COUNT(DISTINCT LEVEL)**

**FROM more\_employees**

**START WITH employee\_id = 1**

**CONNECT BY PRIOR employee\_id = manager\_id;**

COUNT(DISTINCTLEVEL)

--------------------

4

**Formatting the Results from a Hierarchical Query**

You can format the results from a hierarchical query using LEVEL and the LPAD() function, which left-pads values with characters. The following query uses LPAD(' ', 2 \* LEVEL -1) to left-pad a total of 2 \* LEVEL - 1 spaces; the result indents an employee’s name with spaces based on their LEVEL (that is, LEVEL 1 isn’t padded, LEVEL 2 is padded by two spaces, LEVEL 3 by four spaces, and so on):

**SET PAGESIZE 999**

**COLUMN employee FORMAT A25**

**SELECT LEVEL,**

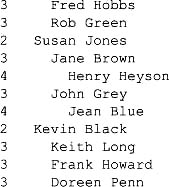
**LPAD(' ', 2 \* LEVEL - 1) || first\_name || ' ' || last\_name AS employees**

**FROM more\_employees**

**START WITH employee\_id = 1**

**CONNECT BY PRIOR employee\_id = manager\_id;**

Image



The employee relationships are easy to pick out from these results.

**Starting at a Node Other than the Root**

You don’t have to start at the root node when traversing a tree: you can start at any node using the START WITHclause. The following query starts with Susan Jones; notice that LEVEL returns 1 for Susan Jones, 2 for Jane Brown, and so on:

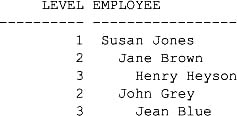
**SELECT LEVEL,**

**LPAD(' ', 2 \* LEVEL - 1) || first\_name || ' ' || last\_name AS employee**

**FROM more\_employees**

**START WITH last\_name = 'Jones'**

**CONNECT BY PRIOR employee\_id = manager\_id;**



If the store had more than one employee with the same name, you could simply use the employee\_id in the query’s START WITH clause. For example, the following query uses Susan Jones' employee\_id of 4:

**SELECT LEVEL,**

**LPAD(' ', 2 \* LEVEL - 1) || first\_name || ' ' || last\_name AS employee**

**FROM more\_employees**

**START WITH employee\_id = 4**

**CONNECT BY PRIOR employee\_id = manager\_id;**

This query returns the same rows as the previous one.

**Using a Subquery in a START WITH Clause**

You can use a subquery in a START WITH clause. For example, the following query uses a subquery to select the employee\_id whose name is Kevin Black; this employee\_id is passed to the START WITH clause:

**SELECT LEVEL,**

**LPAD(' ', 2 \* LEVEL - 1) || first\_name || ' ' || last\_name AS employee**

**FROM more\_employees**

**START WITH employee\_id = (SELECT employee\_id**

**FROM more\_employees**

**WHERE first\_name = 'Kevin'**

**AND last\_name = 'Black'**

**)**

**CONNECT BY PRIOR employee\_id = manager\_id;**



**Traversing Upward Through the Tree**

You don’t have to traverse a tree downward from parents to children: you can start at a child and traverse upward. You do this by switching child and parent columns in the CONNECT BY PRIOR clause. For example, CONNECT BY PRIOR manager\_id = employee\_id connects the child’s manager\_id to the parent’s employee\_id.

The following query starts with Jean Blue and traverses upward all the way to James Smith; notice that LEVELreturns 1 for Jean Blue, 2 for John Grey, and so on:

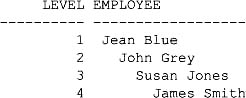
**SELECT LEVEL,**

**LPAD(' ', 2 \* LEVEL - 1) || first\_name || ' ' || last\_name AS employee**

**FROM more\_employees**

**START WITH last\_name = 'Blue'**

**CONNECT BY PRIOR manager\_id = employee\_id;**



**Eliminating Nodes and Branches from a Hierarchical Query**

You can eliminate a particular node from a query tree using a WHERE clause. The following query eliminates Ron Johnson from the results using WHERE last\_name != 'Johnson':

**SELECT LEVEL,**

**LPAD(' ', 2 \* LEVEL - 1) || first\_name || ' ' || last\_name AS employee**

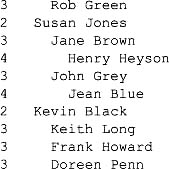
**FROM more\_employees**

**WHERE last\_name != 'Johnson'**

**START WITH employee\_id = 1**

**CONNECT BY PRIOR employee\_id = manager\_id;**

Image



You’ll notice that although Ron Johnson is eliminated from the results, his employees Fred Hobbs and Rob Green are still included. To eliminate an entire branch of nodes from the results of a query, you add an AND clause to your CONNECT BY PRIOR clause. For example, the following query uses AND last\_name != 'Johnson' to eliminate Ron Johnson and all his employees from the results:

**SELECT LEVEL,**

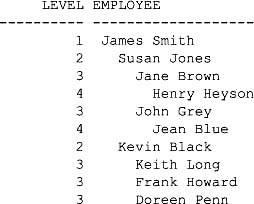
**LPAD(' ', 2 \* LEVEL - 1) || first\_name || ' ' || last\_name AS employee**

**FROM more\_employees**

**START WITH employee\_id = 1**

**CONNECT BY PRIOR employee\_id = manager\_id**

**AND last\_name != 'Johnson';**



**Including Other Conditions in a Hierarchical Query**

You can include other conditions in a hierarchical query using a WHERE clause. The following example uses a WHERE clause to show only employees whose salaries are less than or equal to $50,000:

**SELECT LEVEL,**

**LPAD(' ', 2 \* LEVEL - 1) || first\_name || ' ' || last\_name AS employee,**

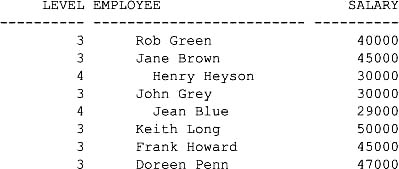
**salary**

**FROM more\_employees**

**WHERE salary <= 50000**

**START WITH employee\_id = 1**

**CONNECT BY PRIOR employee\_id = manager\_id;**



This concludes the discussion of hierarchical queries. In the next section, you’ll learn about advanced group clauses.

**USING THE EXTENDED GROUP BY CLAUSES**

In this section, you’ll learn about

Image ROLLUP, which extends the GROUP BY clause to return a row containing a subtotal for each group of rows, plus a row containing a grand total for all the groups.

Image CUBE, which extends the GROUP BY clause to return rows containing a subtotal for all combinations of columns, plus a row containing the grand total.

First, let’s look at the example tables used in this section.

**The Example Tables**

You’ll see the use of the following tables that refine the representation of employees in our imaginary store:

Image divisions, which stores the divisions within the company

Image jobs, which stores the jobs within the company

Image employees2, which stores the employees

These tables are created by the store\_schema.sql script. The divisions table is created using the following statement:

CREATE TABLE divisions (

division\_id CHAR(3)

CONSTRAINT divisions\_pk PRIMARY KEY,

name VARCHAR2(15) NOT NULL

);

The following query retrieves the rows from the divisions table:

**SELECT \***

**FROM divisions;**

DIV NAME

--- ----------

SAL Sales

OPE Operations

SUP Support

BUS Business

The jobs table is created using the following statement:

CREATE TABLE jobs (

job\_id CHAR(3)

CONSTRAINT jobs\_pk PRIMARY KEY,

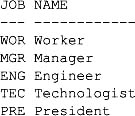
name VARCHAR2(20) NOT NULL

);

The next query retrieves the rows from the jobs table:

**SELECT \***

**FROM jobs;**



The employees2 table is created using the following statement:

CREATE TABLE employees2 (

employee\_id INTEGER

CONSTRAINT employees2\_pk PRIMARY KEY,

division\_id CHAR(3)

CONSTRAINT employees2\_fk\_divisions

REFERENCES divisions(division\_id),

job\_id CHAR(3) REFERENCES jobs(job\_id),

first\_name VARCHAR2(10) NOT NULL,

last\_name VARCHAR2(10) NOT NULL,

salary NUMBER(6, 0)

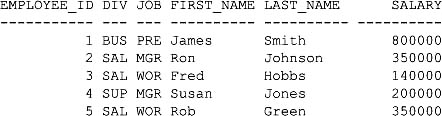
);

The following query retrieves the first five rows from the employees2 table:

**SELECT \***

**FROM employees2**

**WHERE ROWNUM <= 5;**



**USING THE ROLLUP CLAUSE**

The ROLLUP clause extends GROUP BY to return a row containing a subtotal for each group of rows, plus a row containing a total for all the groups.

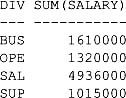
As you saw in [Chapter 4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#ch04), you use GROUP BY to group rows into blocks with a common column value. For example, the following query uses GROUP BY to group the rows from the employees2 table by department\_id and uses SUM() to get the sum of the salaries for each division\_id:

**SELECT division\_id, SUM(salary)**

**FROM employees2**

**GROUP BY division\_id**

**ORDER BY division\_id;**



**Passing a Single Column to ROLLUP**

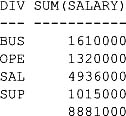
The following query rewrites the previous example to use ROLLUP; notice the additional row at the end, which contains the total salaries for all the groups:

**SELECT division\_id, SUM(salary)**

**FROM employees2**

**GROUP BY ROLLUP(division\_id)**

**ORDER BY division\_id;**



Image

**NOTE**  
*If you need the rows in a specific order, you should use an* ORDER BY *clause. You need to do this just in case Oracle Corporation decides to change the default order of rows returned by* ROLLUP.

**Passing Multiple Columns to ROLLUP**

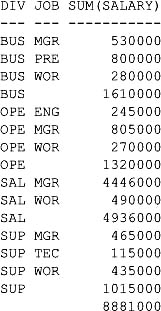
You can pass multiple columns to ROLLUP, which then groups the rows into blocks with the same column values. The following example passes the division\_id and job\_id columns of the employees2 table to ROLLUP, which groups the rows by those columns; in the output, notice that the salaries are summed by division\_id and job\_id, and that ROLLUP returns a row with the sum of the salaries in each division\_id, plus a row at the end with the salary grand total:

**SELECT division\_id, job\_id, SUM(salary)**

**FROM employees2**

**GROUP BY ROLLUP(division\_id, job\_id)**

**ORDER BY division\_id, job\_id;**



**Changing the Position of Columns Passed to ROLLUP**

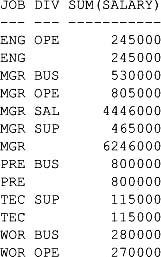
The next example switches division\_id and job\_id; this causes ROLLUP to calculate the sum of the salaries for each job\_id:

**SELECT job\_id, division\_id, SUM(salary)**

**FROM employees2**

**GROUP BY ROLLUP(job\_id, division\_id)**

**ORDER BY job\_id, division\_id;**



Image

**Using Other Aggregate Functions with ROLLUP**

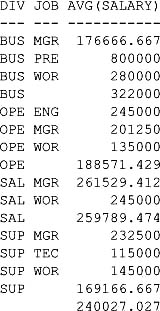
You can use any of the aggregate functions with ROLLUP (for a list of the main aggregate functions, see [Table 4-8](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#table_4-8) in [Chapter 4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#ch04)). The following example uses AVG() to calculate the average salaries:

**SELECT division\_id, job\_id, AVG(salary)**

**FROM employees2**

**GROUP BY ROLLUP(division\_id, job\_id)**

**ORDER BY division\_id, job\_id;**



**Using the CUBE Clause**

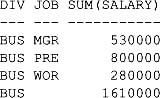
The CUBE clause extends GROUP BY to return rows containing a subtotal for all combinations of columns, plus a row containing the grand total. The following example passes division\_id and job\_id to CUBE, which groups the rows by those columns:

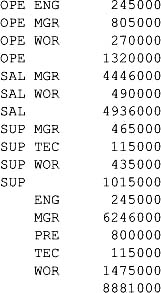
**SELECT division\_id, job\_id, SUM(salary)**

**FROM employees2**

**GROUP BY CUBE(division\_id, job\_id)**

**ORDER BY division\_id, job\_id;**





Notice that the salaries are summed by division\_id and job\_id. CUBE returns a row with the sum of the salaries for each division\_id, along with the sum of all salaries for each job\_id near the end. At the very end is a row with the grand total of the salaries.

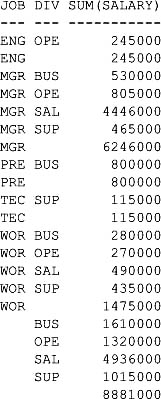
The next example switches division\_id and job\_id:

**SELECT job\_id, division\_id, SUM(salary)**

**FROM employees2**

**GROUP BY CUBE(job\_id, division\_id)**

**ORDER BY job\_id, division\_id;**



**Using the GROUPING() Function**

The GROUPING() function accepts a column and returns 0 or 1. GROUPING() returns 1 when the column value is null and returns 0 when the column value is non-null. GROUPING() is used only in queries that use ROLLUP or CUBE. GROUPING() is useful when you want to display a value when a null would otherwise be returned.

**Using GROUPING() with a Single Column in a ROLLUP**

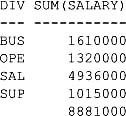
As you saw earlier in the section "Passing a Single Column to ROLLUP," the last row in the example’s result set contained a total of the salaries:

**SELECT division\_id, SUM(salary)**

**FROM employees2**

**GROUP BY ROLLUP(division\_id)**

**ORDER BY division\_id;**



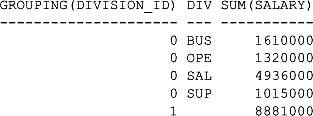
The division\_id column for the last row is null. You can use the GROUPING() function to determine whether this column is null, as shown in the following query; notice GROUPING() returns 0 for the rows that have non-null division\_id values and returns 1 for the last row that has a null division\_id:

**SELECT GROUPING(division\_id), division\_id, SUM(salary)**

**FROM employees2**

**GROUP BY ROLLUP(division\_id)**

**ORDER BY division\_id;**



**Using CASE to Convert the Returned Value from GROUPING()**

You can use the CASE expression to convert the 1 in the previous example to a meaningful value. The following example uses CASE to convert 1 to the string 'All divisions':

**SELECT**

**CASE GROUPING(division\_id)**

**WHEN 1 THEN 'All divisions'**

**ELSE division\_id**

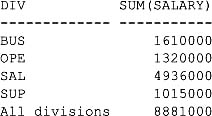
**END AS div,**

**SUM(salary)**

**FROM employees2**

**GROUP BY ROLLUP(division\_id)**

**ORDER BY division\_id;**



**Using CASE and GROUPING() to Convert Multiple Column Values**

The next example extends the idea of replacing null values to a ROLLUP containing multiple columns (division\_id and job\_id); notice that null division\_id values are replaced with the string 'All divisions' and that null job\_id values are replaced with 'All jobs':

**SELECT**

**CASE GROUPING(division\_id)**

**WHEN 1 THEN 'All divisions'**

**ELSE division\_id**

**END AS div,**

**CASE GROUPING(job\_id)**

**WHEN 1 THEN 'All jobs'**

**ELSE job\_id**

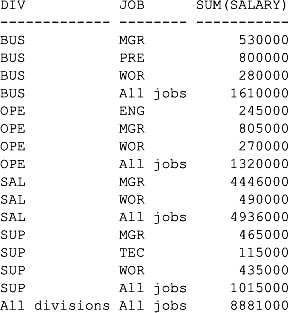
**END AS job,**

**SUM(salary)**

**FROM employees2**

**GROUP BY ROLLUP(division\_id, job\_id)**

**ORDER BY division\_id, job\_id;**



**Using GROUPING() with CUBE**

You can use the GROUPING() function with CUBE, as in this example:

**SELECT**

**CASE GROUPING(division\_id)**

**WHEN 1 THEN 'All divisions'**

**ELSE division\_id**

**END AS div,**

**CASE GROUPING(job\_id)**

**WHEN 1 THEN 'All jobs'**

**ELSE job\_id**

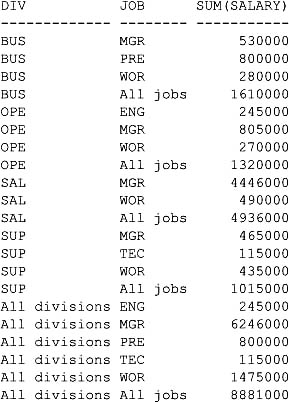
**END AS job,**

**SUM(salary)**

**FROM employees2**

**GROUP BY CUBE(division\_id, job\_id)**

**ORDER BY division\_id, job\_id;**



**Using the GROUPING SETS Clause**

You use the GROUPING SETS clause to get just the subtotal rows. The following example uses GROUPING SETS to get the subtotals for salaries by division\_id and job\_id:

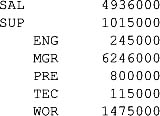
**SELECT division\_id, job\_id, SUM(salary)**

**FROM employees2**

**GROUP BY GROUPING SETS(division\_id, job\_id)**

**ORDER BY division\_id, job\_id;**

Image



Notice that only subtotals for the division\_id and job\_id columns are returned; the total for all salaries is not returned. You’ll see how to get the total as well as the subtotals using the GROUPING\_ID() function in the next section.

Image

**TIP**  
*The* GROUPING SETS *clause typically offers better performance than* CUBE. *Therefore, you should use* GROUPING SETS *rather than* CUBE *wherever possible*.

**Using the GROUPING\_ID() Function**

You can use the GROUPING\_ID() function to filter rows using a HAVING clause to exclude rows that don’t contain a subtotal or total. The GROUPING\_ID() function accepts one or more columns and returns the decimal equivalent of the GROUPING bit vector. The GROUPING bit vector is computed by combining the results of a call to the GROUPING() function for each column in order.

**Computing the GROUPING Bit Vector**

Earlier in the section "Using the GROUPING() Function," you saw that GROUPING() returns 1 when the column value is null and returns 0 when the column value is non-null; for example:

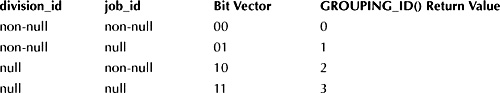
Image If both division\_id and job\_id are non-null, GROUPING() returns 0 for both columns. The result for division\_id is combined with the result for job\_id, giving a bit vector of 00, whose decimal equivalent is 0. GROUPING\_ID() therefore returns 0 when division\_id and job\_id are non-null.

Image If division\_id is non-null (the GROUPING bit is 0), but job\_id is null (the GROUPING bit is 1), the resulting bit vector is 01 and GROUPING\_ID() returns 1.

Image If division\_id is null (the GROUPING bit is 1), but job\_id is non-null (the GROUPING bit is 0), the resulting bit vector is 10 and GROUPING\_ID() returns 2.

Image If both division\_id and job\_id are null (both GROUPING bits are 0), the bit vector is 11 and GROUPING\_ID() returns 3.

The following table summarizes these results.



**An Example Query That Illustrates the Use of GROUPING\_ID()**

The following example passes division\_id and job\_id to GROUPING\_ID(); notice that the output from the GROUPING\_ID() function agrees with the expected returned values documented in the previous section:

**SELECT**

**division\_id, job\_id,**

**GROUPING(division\_id) AS DIV\_GRP,**

**GROUPING(job\_id) AS JOB\_GRP,**

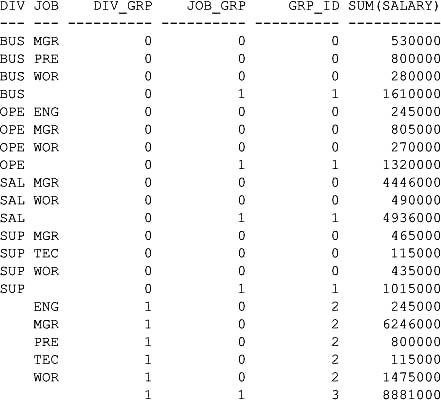
**GROUPING\_ID(division\_id, job\_id) AS grp\_id,**

**SUM(salary)**

**FROM employees2**

**GROUP BY CUBE(division\_id, job\_id)**

**ORDER BY division\_id, job\_id;**



**A Useful Application of GROUPING\_ID()**

One useful application of GROUPING\_ID() is to filter rows using a HAVING clause. The HAVING clause can exclude rows that don’t contain a subtotal or total by simply checking if GROUPING\_ID() returns a value greater than 0. For example:

**SELECT**

**division\_id, job\_id,**

**GROUPING\_ID(division\_id, job\_id) AS grp\_id,**

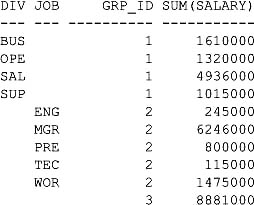
**SUM(salary)**

**FROM employees2**

**GROUP BY CUBE(division\_id, job\_id)**

**HAVING GROUPING\_ID(division\_id, job\_id) > 0**

**ORDER BY division\_id, job\_id;**



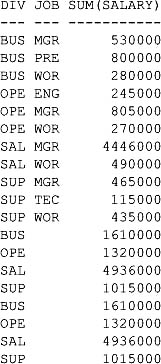
**Using a Column Multiple Times in a GROUP BY Clause**

You can use a column many times in a GROUP BY clause. Doing this allows you to reorganize your data or report on different groupings of data. For example, the following query contains a GROUP BY clause that uses division\_id twice, once to group by division\_id and again in a ROLLUP:

**SELECT division\_id, job\_id, SUM(salary)**

**FROM employees2**

**GROUP BY division\_id, ROLLUP(division\_id, job\_id);**



Notice, however, that the last four rows are duplicates of the previous four rows. You can eliminate these duplicates using the GROUP\_ID() function, which you’ll learn about next.

**Using the GROUP\_ID() Function**

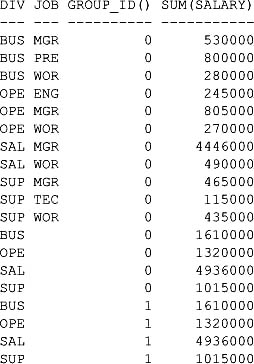
You can use the GROUP\_ID() function to remove duplicate rows returned by a GROUP BY clause. GROUP\_ID() doesn’t accept any parameters. If *n* duplicates exist for a particular grouping, GROUP\_ID returns numbers in the range 0 to *n* − 1.

The following example rewrites the query shown in the previous section to include the output from GROUP\_ID(); notice that GROUP\_ID() returns 0 for all rows except the last four, which are duplicates of the previous four rows, and that GROUP\_ID() returns 1:

**SELECT division\_id, job\_id, GROUP\_ID(), SUM(salary)**

**FROM employees2**

**GROUP BY division\_id, ROLLUP(division\_id, job\_id);**



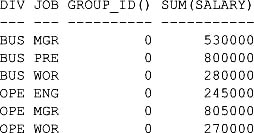
You can eliminate duplicate rows using a HAVING clause that allows only rows whose GROUP\_ID() is 0; for example:

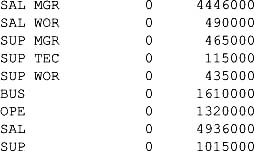
**SELECT division\_id, job\_id, GROUP\_ID(), SUM(salary)**

**FROM employees2**

**GROUP BY division\_id, ROLLUP(division\_id, job\_id)**

**HAVING GROUP\_ID() = 0;**





This concludes the discussion of the extended GROUP BY clauses.

**USING THE ANALYTIC FUNCTIONS**

The database has many built-in analytic functions that enable you to perform complex calculations, such as finding the top-selling product type for each month, the top salespersons, and so on. The analytic functions are organized into the following categories:

Image **Ranking functions** enable you to calculate ranks, percentiles, and *n*-tiles (tertiles, quartiles, and so on).

Image **Inverse percentile functions** enable you to calculate the value that corresponds to a percentile.

Image **Window functions** enable you to calculate cumulative and moving aggregates.

Image **Reporting functions** enable you to calculate things like market share.

Image **Lag and lead functions** enable you to get a value in a row where that row is a certain number of rows away from the current row.

Image **First and last functions** enable you to get the first and last values in an ordered group.

Image **Linear regression functions** enable you to fit an ordinary-least-squares regression line to a set of number pairs.

Image **Hypothetical rank and distribution functions** enable you to calculate the rank and percentile that a new row would have if you inserted it into a table.

You’ll learn about these functions shortly, but first let’s examine the example table used next.

**The Example Table**

You’ll see the use of the all\_sales table in the following sections. The all\_sales table stores the sum of all the sales by dollar amount for a particular year, month, product type, and employee. The all\_sales table is created by the store\_schema.sql script as follows:

CREATE TABLE all\_sales (

year INTEGER NOT NULL,

month INTEGER NOT NULL,

prd\_type\_id INTEGER

CONSTRAINT all\_sales\_fk\_product\_types

REFERENCES product\_types(product\_type\_id),

emp\_id INTEGER

CONSTRAINT all\_sales\_fk\_employees2

REFERENCES employees2(employee\_id),

amount NUMBER(8, 2),

CONSTRAINT all\_sales\_pk PRIMARY KEY (

year, month, prd\_type\_id, emp\_id

)

);

As you can see, the all\_sales table contains five columns, which are as follows:

Image **YEAR** stores the year the sales took place.

Image **MONTH** stores the month the sales took place (1 to 12).

Image **PRD\_TYPE\_ID** stores the product\_type\_id of the product.

Image **EMP\_ID** stores the employee\_id of the employee who handled the sales.

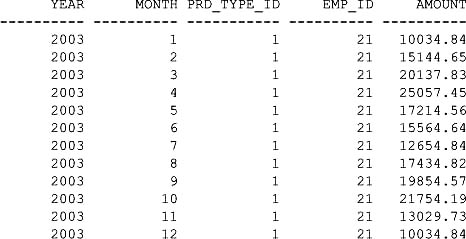
Image **AMOUNT** stores the total dollar amount of the sales.

The following query retrieves the first 12 rows from the all\_sales table:

**SELECT \***

**FROM all\_sales**

**WHERE ROWNUM <= 12;**



Image

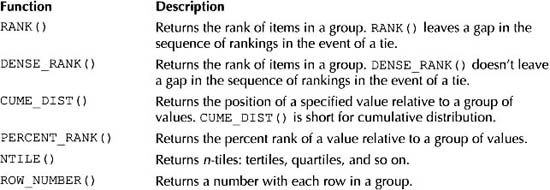
**NOTE**  
*The* all\_sales *table actually contains a lot more rows than this, but for space considerations I’ve omitted listing them all here*.

Let’s examine the ranking functions next.

**Using the Ranking Functions**

You use the ranking functions to calculate ranks, percentiles, and *n*-tiles. The ranking functions are shown in [Table 7-2](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch07.html#table_7-2).

Let’s examine the RANK() and DENSE\_RANK() functions first.



**TABLE 7-2** *The Ranking Functions*

**Using the RANK() and DENSE\_RANK() Functions**

You use RANK() and DENSE\_RANK() to rank items in a group. The difference between these two functions is in the way they handle items that tie: RANK() leaves a gap in the sequence when there is a tie, but DENSE\_RANK() leaves no gaps. For example, if you were ranking sales by product type and two product types tie for first place, RANK() would put the two product types in first place, but the next product type would be in third place. DENSE\_RANK() would also put the two product types in first place, but the next product type would be in second place.

The following query illustrates the use of RANK() and DENSE\_RANK() to get the ranking of sales by product type for the year 2003; notice the use of the keyword OVER in the syntax when calling the RANK() and DENSE\_RANK() functions:

**SELECT**

**prd\_type\_id, SUM(amount),**

**RANK() OVER (ORDER BY SUM(amount) DESC) AS rank,**

**DENSE\_RANK() OVER (ORDER BY SUM(amount) DESC) AS dense\_rank**

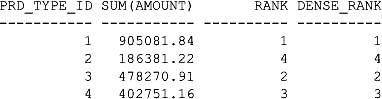
**FROM all\_sales**

**WHERE year = 2003**

**AND amount IS NOT NULL**

**GROUP BY prd\_type\_id**

**ORDER BY prd\_type\_id;**



Notice that sales for product type #1 are ranked first, sales for product type #2 are ranked fourth, and so on. Because there are no ties, RANK() and DENSE\_RANK() return the same ranks.

The all\_sales table actually contains nulls in the AMOUNT column for all rows whose PRD\_TYPE\_IDcolumn is 5; the previous query omits these rows because of the inclusion of the line "AND amount IS NOT NULL" in the WHERE clause. The next example includes these rows by leaving out the AND line from the WHEREclause:

**SELECT**

**prd\_type\_id, SUM(amount),**

**RANK() OVER (ORDER BY SUM(amount) DESC) AS rank,**

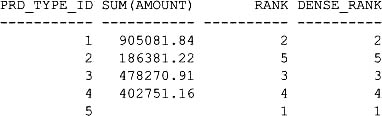
**DENSE\_RANK() OVER (ORDER BY SUM(amount) DESC) AS dense\_rank**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY prd\_type\_id**

**ORDER BY prd\_type\_id;**



Notice that the last row contains null for the sum of the AMOUNT column and that RANK() and DENSE\_RANK() return 1 for this row. This is because by default RANK() and DENSE\_RANK() assign the highest rank of 1 to null values in descending rankings (that is, DESC is used in the OVER clause) and the lowest rank in ascending rankings (that is, ASC is used in the OVER clause).

**Controlling Ranking of Null Values Using the NULLS FIRST and NULLS LAST Clauses** When using an analytic function, you can explicitly control whether nulls are the highest or lowest in a group using NULLS FIRST or NULLS LAST. The following example uses NULLS LAST to specify that nulls are the lowest:

**SELECT**

**prd\_type\_id, SUM(amount),**

**RANK() OVER (ORDER BY SUM(amount) DESC NULLS LAST) AS rank,**

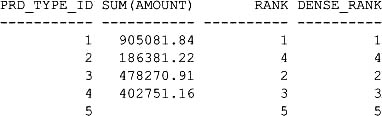
**DENSE\_RANK() OVER (ORDER BY SUM(amount) DESC NULLS LAST) AS dense\_rank**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY prd\_type\_id**

**ORDER BY prd\_type\_id;**



**Using the PARTITION BY Clause with Analytic Functions** You use the PARTITION BY clause with the analytic functions when you need to divide the groups into subgroups. For example, if you need to subdivide the sales amount by month, you can use PARTITION BY month, as shown in the following query:

**SELECT**

**prd\_type\_id, month, SUM(amount),**

**RANK() OVER (PARTITION BY month ORDER BY SUM(amount) DESC) AS rank**

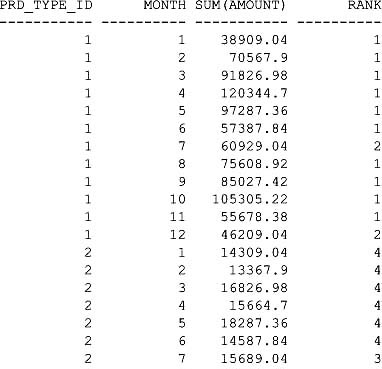
**FROM all\_sales**

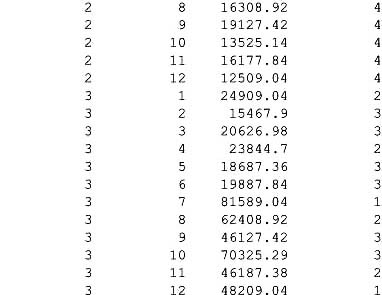
**WHERE year = 2003**

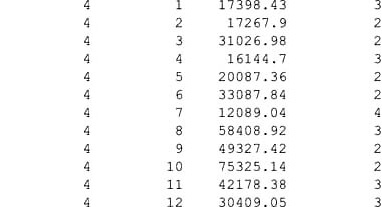
**AND amount IS NOT NULL**

**GROUP BY prd\_type\_id, month**

**ORDER BY prd\_type\_id, month;**







**Using ROLLUP, CUBE, and GROUPING SETS Operators with Analytic Functions** You can use the ROLLUP, CUBE, and GROUPING SETS operators with the analytic functions. The following query uses ROLLUP and RANK() to get the sales rankings by product type ID:

**SELECT**

**prd\_type\_id, SUM(amount),**

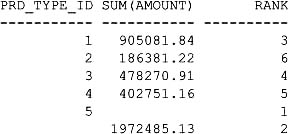
**RANK() OVER (ORDER BY SUM(amount) DESC) AS rank**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY ROLLUP(prd\_type\_id)**

**ORDER BY prd\_type\_id;**



The next query uses CUBE and RANK() to get all rankings of sales by product type ID and employee ID:

**SELECT**

**prd\_type\_id, emp\_id, SUM(amount),**

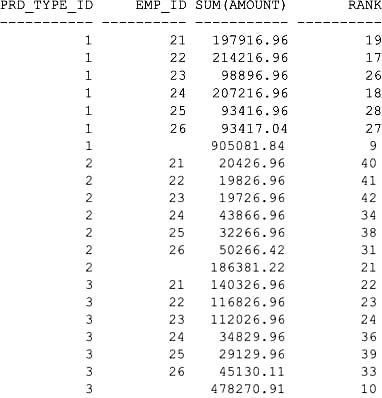
**RANK() OVER (ORDER BY SUM(amount) DESC) AS rank**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY CUBE(prd\_type\_id, emp\_id)**

**ORDER BY prd\_type\_id, emp\_id;**



The next query uses GROUPING SETS and RANK() to get just the sales amount subtotal rankings:

**SELECT**

**prd\_type\_id, emp\_id, SUM(amount),**

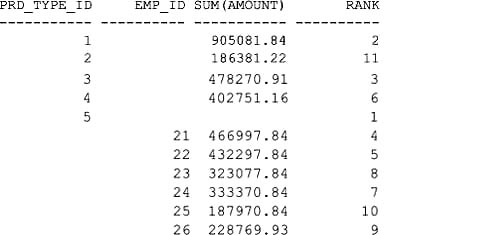
**RANK() OVER (ORDER BY SUM(amount) DESC) AS rank**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY GROUPING SETS(prd\_type\_id, emp\_id)**

**ORDER BY prd\_type\_id, emp\_id;**



**Using the CUME\_DIST() and PERCENT\_RANK() Functions**

You use CUME\_DIST() to calculate the position of a specified value relative to a group of values; CUME\_DIST() is short for cumulative distribution. You use PERCENT\_RANK() to calculate the percent rank of a value relative to a group of values.

The following query illustrates the use of CUME\_DIST() and PERCENT\_RANK() to get the cumulative distribution and percent rank of sales:

**SELECT**

**prd\_type\_id, SUM(amount),**

**CUME\_DIST() OVER (ORDER BY SUM(amount) DESC) AS cume\_dist,**

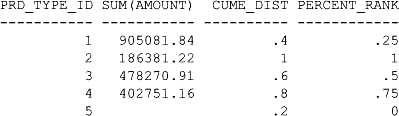
**PERCENT\_RANK() OVER (ORDER BY SUM(amount) DESC) AS percent\_rank**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY prd\_type\_id**

**ORDER BY prd\_type\_id;**



**Using the NTILE() Function**

You use NTILE ( *buckets* ) to calculate *n*-tiles (tertiles, quartiles, and so on); *buckets* specifies the number of "buckets" into which groups of rows are placed. For example, NTILE(2) specifies two buckets and therefore divides the rows into two groups of rows; NTILE(4) divides the groups into four buckets and therefore divides the rows into four groups.

The following query illustrates the use of NTILE(); notice that 4 is passed to NTILE() to split the groups of rows into four buckets:

**SELECT**

**prd\_type\_id, SUM(amount),**

**NTILE(4) OVER (ORDER BY SUM(amount) DESC) AS ntile**

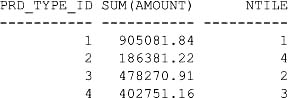
**FROM all\_sales**

**WHERE year = 2003**

**AND amount IS NOT NULL**

**GROUP BY prd\_type\_id**

**ORDER BY prd\_type\_id;**



**Using the ROW\_NUMBER() Function**

You use ROW\_NUMBER() to return a number with each row in a group, starting at 1. The following query illustrates the use of ROW\_NUMBER():

**SELECT**

**prd\_type\_id, SUM(amount),**

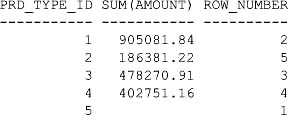
**ROW\_NUMBER() OVER (ORDER BY SUM(amount) DESC) AS row\_number**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY prd\_type\_id**

**ORDER BY prd\_type\_id;**



This concludes the discussion of ranking functions.

**Using the Inverse Percentile Functions**

In the section "Using the CUME\_DIST() and PERCENT\_RANK() Functions," you saw that CUME\_DIST() is used to calculate the position of a specified value relative to a group of values. You also saw that PERCENT\_RANK() is used to calculate the percent rank of a value relative to a group of values.

In this section, you’ll see how to use the inverse percentile functions to get the value that corresponds to a percentile. There are two inverse percentile functions: PERCENTILE\_DISC (*x*) and PERCENTILE\_CONT (*x*). They operate in a manner the reverse of CUME\_DIST() and PERCENT\_RANK(). PERCENTILE\_DISC(*x*) examines the cumulative distribution values in each group until it finds one that is greater than or equal to *x*. PERCENTILE\_CONT(*x*) examines the percent rank values in each group until it finds one that is greater than or equal to *x*.

The following query illustrates the use of PERCENTILE\_CONT() and PERCENTILE\_DISC() to get the sum of the amount whose percentile is greater than or equal to 0.6:

**SELECT**

**PERCENTILE\_CONT(0.6) WITHIN GROUP (ORDER BY SUM(amount) DESC)**

**AS percentile\_cont,**

**PERCENTILE\_DISC(0.6) WITHIN GROUP (ORDER BY SUM(amount) DESC)**

**AS percentile\_disc**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY prd\_type\_id;**

Image

If you compare the sum of the amounts shown in these results with those shown in the earlier section "Using the CUME\_DIST() and PERCENT\_RANK() Functions," you’ll see that the sums correspond to those whose cumulative distribution and percent rank are 0.6 and 0.75, respectively.

**Using the Window Functions**

You use the window functions to calculate things like cumulative sums and moving averages within a specified range of rows, a range of values, or an interval of time. As you know, a query returns a set of rows known as the result set. The term "window" is used to describe a subset of rows within the result set. The subset of rows "seen" through the window is then processed by the window functions, which return a value. You can define the start and end of the window.

You can use a window with the following functions: SUM(), AVG(), MAX(), MIN(), COUNT(), VARIANCE(), and STDDEV(); you saw these functions in [Chapter 4](https://www.safaribooksonline.com/library/view/oracle-database-11g/9780071498500/ch04.html#ch04). You can also use a window with FIRST\_VALUE() and LAST\_VALUE(), which return the first and last values in a window. (You’ll learn more about the FIRST\_VALUE() and LAST\_VALUE() functions later in the section "Getting the First and Last Rows Using FIRST\_VALUE() and LAST\_VALUE().")

In the next section, you’ll see how to perform a cumulative sum, a moving average, and a centered average.

**Performing a Cumulative Sum**

The following query performs a cumulative sum to compute the cumulative sales amount for 2003, starting with January and ending in December; notice that each monthly sales amount is added to the cumulative amount that grows after each month:

**SELECT**

**month, SUM(amount) AS month\_amount,**

**SUM(SUM(amount)) OVER**

**(ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)**

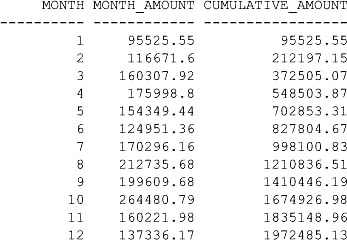
**AS cumulative\_amount**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY month**

**ORDER BY month;**



This query uses the following expression to compute the cumulative aggregate:

SUM(SUM(amount)) OVER

(ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)

AS cumulative\_amount

Let’s break down this expression:

Image SUM(amount) computes the sum of an amount. The outer SUM() computes the cumulative amount.

Image ORDER BY month orders the rows read by the query by month.

Image ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW defines the start and end of the window. The start is set to UNBOUNDED PRECEDING, which means the start of the window is fixed at the first row in the result set returned by the query. The end of the window is set to CURRENT ROW; CURRENT ROWrepresents the current row in the result set being processed, and the end of the window slides down one row after the outer SUM() function computes and returns the current cumulative amount.

The entire query computes and returns the cumulative total of the sales amounts, starting at month 1, and then adding the sales amount for month 2, then month 3, and so on, up to and including month 12. The start of the window is fixed at month 1, but the bottom of the window moves down one row in the result set after each month’s sales amounts are added to the cumulative total. This continues until the last row in the result set is processed by the window and the SUM() functions.

Don’t confuse the end of the window with the end of the result set. In the previous example, the end of the window slides down one row in the result set as each row is processed (i.e., the sum of the sales amount for that month is added to the cumulative total). In the example, the end of the window starts at the first row, the sum sales amount for that month is added to the cumulative total, and then the end of the window moves down one row to the second row. At this point, the window sees two rows. The sum of the sales amount for that month is added to the cumulative total, and the end of the window moves down one row to the third row. At this point, the window sees three rows. This continues until the twelfth row is processed. At this point, the window sees twelve rows.

The following query uses a cumulative sum to compute the cumulative sales amount, starting with June of 2003 (month 6) and ending in December of 2003 (month 12):

**SELECT**

**month, SUM(amount) AS month\_amount,**

**SUM(SUM(amount)) OVER**

**(ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS**

**cumulative\_amount**

**FROM all\_sales**

**WHERE year = 2003**

**AND month BETWEEN 6 AND 12**

**GROUP BY month**

**ORDER BY month;**

MONTH MONTH\_AMOUNT CUMULATIVE\_AMOUNT

---------- ------------ -----------------

6 124951.36 124951.36

7 170296.16 295247.52

8 212735.68 507983.2

9 199609.68 707592.88

10 264480.79 972073.67

11 160221.98 1132295.65

12 137336.17 1269631.82

**Performing a Moving Average**

The following query computes the moving average of the sales amount between the current month and the previous three months:

**SELECT**

**month, SUM(amount) AS month\_amount,**

**AVG(SUM(amount)) OVER**

**(ORDER BY month ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)**

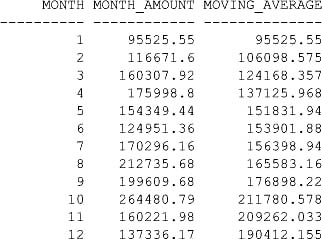
**AS moving\_average**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY month**

**ORDER BY month;**



Notice that the query uses the following expression to compute the moving average:

AVG(SUM(amount)) OVER

(ORDER BY month ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)

AS moving\_average

Let’s break down this expression:

Image SUM(amount) computes the sum of an amount. The outer AVG() computes the average.

Image ORDER BY month orders the rows read by the query by month.

Image ROWS BETWEEN 3 PRECEDING AND CURRENT ROW defines the start of the window as including the three rows preceding the current row; the end of the window is the current row being processed.

So, the entire expression computes the moving average of the sales amount between the current month and the previous three months. Because for the first two months less than the full three months of data are available, the moving average is based on only the months available.

Both the start and the end of the window begin at row #1 read by the query. The end of the window moves down after each row is processed. The start of the window moves down only after row #4 has been processed, and subsequently moves down one row after each row is processed. This continues until the last row in the result set is read.

**Performing a Centered Average**

The following query computes the moving average of the sales amount centered between the previous and next month from the current month:

**SELECT**

**month, SUM(amount) AS month\_amount,**

**AVG(SUM(amount)) OVER**

**(ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)**

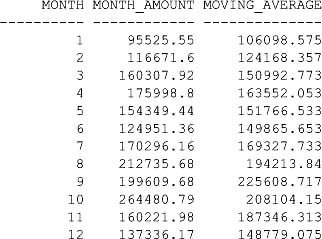
**AS moving\_average**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY month**

**ORDER BY month;**



Notice that the query uses the following expression to compute the moving average:

AVG(SUM(amount)) OVER

(ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)

AS moving\_average

Let’s break down this expression:

Image SUM(amount) computes the sum of an amount. The outer AVG() computes the average.

Image ORDER BY month orders the rows read by the query by month.

Image ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING defines the start of the window as including the row preceding the current row being processed. The end of the window is the row following the current row.

So, the entire expression computes the moving average of the sales amount between the current month and the previous month. Because for the first and last month less than the full three months of data are available, the moving average is based on only the months available.

The start of the window begins at row #1 read by the query. The end of the window begins at row #2 and moves down after each row is processed. The start of the window moves down only once row #2 has been processed. Processing continues until the last row read by the query is processed.

**Getting the First and Last Rows Using FIRST\_VALUE() and LAST\_VALUE()**

You use the FIRST\_VALUE() and LAST\_VALUE() functions to get the first and last rows in a window. The following query uses FIRST\_VALUE() and LAST\_VALUE() to get the previous and next month’s sales amount:

**SELECT**

**month, SUM(amount) AS month\_amount,**

**FIRST\_VALUE(SUM(amount)) OVER**

**(ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)**

**AS previous\_month\_amount,**

**LAST\_VALUE(SUM(amount)) OVER**

**(ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)**

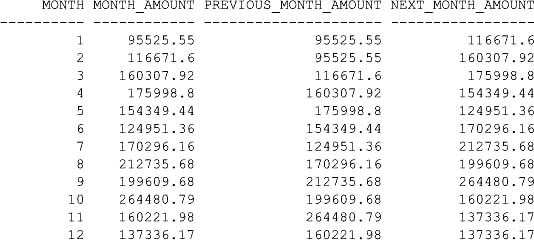
**AS next\_month\_amount**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY month**

**ORDER BY month;**



The next query divides the current month’s sales amount by the previous month’s sales amount (labeled as curr\_div\_prev) and also divides the current month’s sales amount by the next month’s sales amount (labeled as curr\_div\_next):

**SELECT**

**month, SUM(amount) AS month\_amount,**

**SUM(amount)/FIRST\_VALUE(SUM(amount)) OVER**

**(ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)**

**AS curr\_div\_prev,**

**SUM(amount)/LAST\_VALUE(SUM(amount)) OVER**

**(ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)**

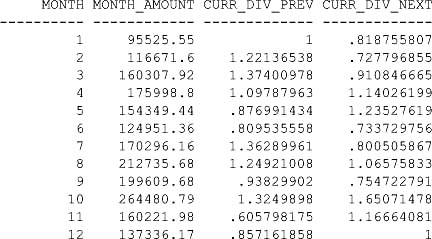
**AS curr\_div\_next**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY month**

**ORDER BY month;**



This concludes the discussion of window functions.

**Using the Reporting Functions**

You use the reporting functions to perform calculations across groups and partitions within groups.

You can perform reporting with the following functions: SUM(), AVG(), MAX(), MIN(), COUNT(), VARIANCE(), and STDDEV(). You can also use the RATIO\_TO\_REPORT() function to compute the ratio of a value to the sum of a set of values.

In this section, you’ll see how to perform a report on a sum and use the RATIO\_TO\_REPORT() function.

**Reporting on a Sum**

For the first three months of 2003, the following query reports

Image The total sum of all sales for all three months (labeled as total\_month\_amount).

Image The total sum of all sales for all product types (labeled as total\_product\_type\_amount).

**SELECT**

**month, prd\_type\_id,**

**SUM(SUM(amount)) OVER (PARTITION BY month)**

**AS total\_month\_amount,**

**SUM(SUM(amount)) OVER (PARTITION BY prd\_type\_id)**

**AS total\_product\_type\_amount**

**FROM all\_sales**

**WHERE year = 2003**

**AND month <= 3**

**GROUP BY month, prd\_type\_id**

**ORDER BY month, prd\_type\_id;**

MONTH PRD\_TYPE\_ID TOTAL\_MONTH\_AMOUNT TOTAL\_PRODUCT\_TYPE\_AMOUNT

---------- ----------- ------------------ -------------------------

1 1 95525.55 201303.92

1 2 95525.55 44503.92

1 3 95525.55 61003.92

1 4 95525.55 65693.31

1 5 95525.55

2 1 116671.6 201303.92

2 2 116671.6 44503.92

2 3 116671.6 61003.92

2 4 116671.6 65693.31

2 5 116671.6

3 1 160307.92 201303.92

3 2 160307.92 44503.92

3 3 160307.92 61003.92

3 4 160307.92 65693.31

3 5 160307.92

Notice that the query uses the following expression to report the total sum of all sales for all months (labeled as total\_month\_amount):

SUM(SUM(amount)) OVER (PARTITION BY month)

AS total\_month\_amount

Let’s break down this expression:

Image SUM(amount) computes the sum of an amount. The outer SUM() computes the total sum.

Image OVER (PARTITION BY month) causes the outer SUM() to compute the sum for each month.

The previous query also uses the following expression to report the total sum of all sales for all product types (labeled as total\_product\_type\_amount):

SUM(SUM(amount)) OVER (PARTITION BY prd\_type\_id)

AS total\_product\_type\_amount

Let’s break down this expression:

Image SUM(amount) computes the sum of an amount. The outer SUM() computes the total sum.

Image OVER (PARTITION BY prd\_type\_id) causes the outer SUM() to compute the sum for each product type.

**Using the RATIO\_TO\_REPORT() Function**

You use the RATIO\_TO\_REPORT() function to compute the ratio of a value to the sum of a set of values.

For the first three months of 2003, the following query reports

Image The sum of the sales amount by product type for each month (labeled as prd\_type\_amount).

Image The ratio of the product type’s sales amount to the entire month’s sales (labeled as prd\_type\_ratio), which is computed using RATIO\_TO\_REPORT().

**SELECT**

**month, prd\_type\_id,**

**SUM(amount) AS prd\_type\_amount,**

**RATIO\_TO\_REPORT(SUM(amount)) OVER (PARTITION BY month) AS prd\_type\_ratio**

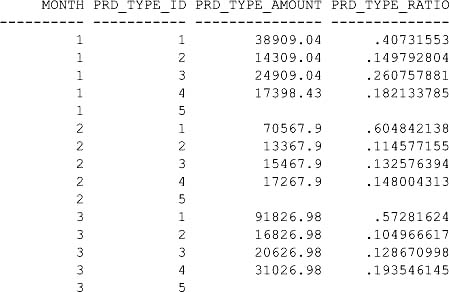
**FROM all\_sales,**

**WHERE year = 2003,**

**AND month <= 3,**

**GROUP BY month, prd\_type\_id,**

**ORDER BY month, prd\_type\_id;**



Notice that the query uses the following expression to compute the ratio (labeled as prd\_type\_ratio):

RATIO\_TO\_REPORT(SUM(amount)) OVER (PARTITION BY month) AS prd\_type\_ratio

Let’s break down this expression:

Image SUM(amount) computes the sum of the sales amount.

Image OVER (PARTITION BY month) causes the outer SUM() to compute the sum of the sales amount for each month.

Image The ratio is computed by dividing the sum of the sales amount for each product type by the sum of the entire month’s sales amount.

This concludes the discussion of reporting functions.

**Using the LAG() and LEAD() Functions**

You use the LAG() and LEAD() functions to get a value in a row where that row is a certain number of rows away from the current row. The following query uses LAG() and LEAD() to get the previous and next month’s sales amount:

**SELECT**

**month, SUM(amount) AS month\_amount,**

**LAG(SUM(amount), 1) OVER (ORDER BY month) AS previous\_month\_amount,**

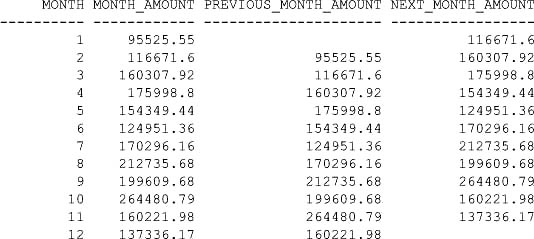
**LEAD(SUM(amount), 1) OVER (ORDER BY month) AS next\_month\_amount**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY month**

**ORDER BY month;**



Notice that the query uses the following expressions to get the previous and next month’s sales:

LAG(SUM(amount), 1) OVER (ORDER BY month) AS previous\_month\_amount,

LEAD(SUM(amount), 1) OVER (ORDER BY month) AS next\_month\_amount

LAG(SUM(amount), 1) gets the previous row’s sum of the amount. LEAD(SUM(amount), 1) gets the next row’s sum of the amount.

**Using the FIRST and LAST Functions**

You use the FIRST and LAST functions to get the first and last values in an ordered group. You can use FIRSTand LAST with the following functions: MIN(), MAX(), COUNT(), SUM(), AVG(), STDDEV(), and VARIANCE().

The following query uses FIRST and LAST to get the months in 2003 that had the highest and lowest sales:

**SELECT**

**MIN(month) KEEP (DENSE\_RANK FIRST ORDER BY SUM(amount))**

**AS highest\_sales\_month,**

**MIN(month) KEEP (DENSE\_RANK LAST ORDER BY SUM(amount))**

**AS lowest\_sales\_month**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY month**

**ORDER BY month;**

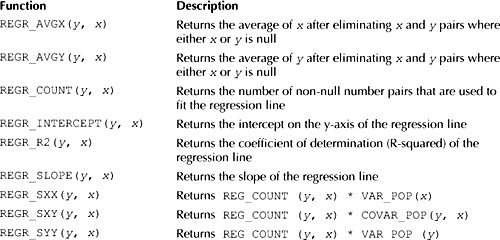
HIGHEST\_SALES\_MONTH LOWEST\_SALES\_MONTH

------------------- ------------------

1 10

**Using the Linear Regression Functions**

You use the linear regression functions to fit an ordinary-least-squares regression line to a set of number pairs. You can use the linear regression functions as aggregate, windowing, or reporting functions. The following table shows the linear regression functions. In the function syntax, *y* is interpreted by the functions as a variable that depends on *x*.



The following query shows the use of the linear regression functions:

**SELECT**

**prd\_type\_id,**

**REGR\_AVGX(amount, month) AS avgx,**

**REGR\_AVGY(amount, month) AS avgy,**

**REGR\_COUNT(amount, month) AS count,**

**REGR\_INTERCEPT(amount, month) AS inter,**

**REGR\_R2(amount, month) AS r2,**

**REGR\_SLOPE(amount, month) AS slope,**

**REGR\_SXX(amount, month) AS sxx,**

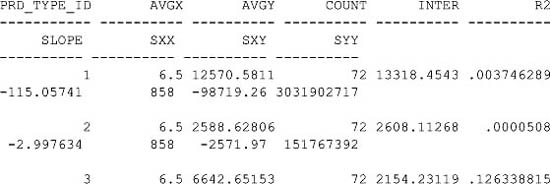
**REGR\_SXY(amount, month) AS sxy,**

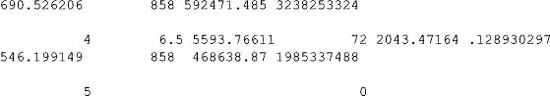
**REGR\_SYY(amount, month) AS syy**

**FROM all\_sales**

**WHERE year = 2003**

**GROUP BY prd\_type\_id;**





**Using the Hypothetical Rank and Distribution Functions**

You use the hypothetical rank and distribution functions to calculate the rank and percentile that a new row would have if you inserted it into a table. You can perform hypothetical calculations with the following functions: RANK(), DENSE\_RANK(), PERCENT\_RANK(), and CUME\_DIST().

An example of a hypothetical function will be given after the following query, which uses RANK() and PERCENT\_RANK() to get the rank and percent rank of sales by product type for 2003:

**SELECT**

**prd\_type\_id, SUM(amount),**

**RANK() OVER (ORDER BY SUM(amount) DESC) AS rank,**

**PERCENT\_RANK() OVER (ORDER BY SUM(amount) DESC) AS percent\_rank**

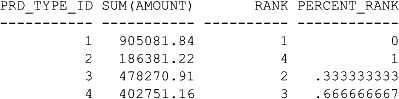
**FROM all\_sales**

**WHERE year = 2003**

**AND amount IS NOT NULL**

**GROUP BY prd\_type\_id**

**ORDER BY prd\_type\_id;**



The next query shows the hypothetical rank and percent rank of a sales amount of $500,000:

**SELECT**

**RANK(500000) WITHIN GROUP (ORDER BY SUM(amount) DESC)**

**AS rank,**

**PERCENT\_RANK(500000) WITHIN GROUP (ORDER BY SUM(amount) DESC)**

**AS percent\_rank,**

**FROM all\_sales,**

**WHERE year = 2003,**

**AND amount IS NOT NULL,**

**GROUP BY prd\_type\_id,**

**ORDER BY prd\_type\_id;**

RANK PERCENT\_RANK

---------- ------------

2 .25

As you can see, the hypothetical rank and percent rank of a sales amount of $500,000 are 2 and .25.

This concludes the discussion of hypothetical functions.

**USING THE MODEL CLAUSE**

The MODEL clause was introduced with Oracle Database 10*g* and enables you to perform inter-row calculations. The MODEL clause allows you to access a column in a row like a cell in an array. This gives you the ability to perform calculations in a similar manner to spreadsheet calculations. For example, the all\_sales table contains sales information for the months in 2003. You can use the MODEL clause to calculate sales in future months based on sales in 2003.

**An Example of the MODEL Clause**

The easiest way to learn how to use the MODEL clause is to see an example. The following query retrieves the sales amount for each month in 2003 made by employee #21 for product types #1 and #2 and computes the predicted sales for January, February, and March of 2004 based on sales in 2003:

**SELECT prd\_type\_id, year, month, sales\_amount**

**FROM all\_sales**

**WHERE prd\_type\_id BETWEEN 1 AND 2**

**AND emp\_id = 21**

**MODEL**

**PARTITION BY (prd\_type\_id)**

**DIMENSION BY (month, year)**

**MEASURES (amount sales\_amount) (**

**sales\_amount[1, 2004] = sales\_amount[1, 2003],**

**sales\_amount[2, 2004] = sales\_amount[2, 2003] + sales\_amount[3, 2003],**

**sales\_amount[3, 2004] = ROUND(sales\_amount[3, 2003] \* 1.25, 2)**

**)**

**ORDER BY prd\_type\_id, year, month;**

Let’s break down this query:

Image PARTITION BY (prd\_type\_id) specifies that the results are partitioned by prd\_type\_id.

Image DIMENSION BY (month, year) specifies that the dimensions of the array are month and year. This means that a cell in the array is accessed by specifying a month and year.

Image MEASURES (amount sales\_amount) specifies that each cell in the array contains an amount and that the array name is sales\_amount. To access the cell in the sales\_amount array for January 2003, you use sales\_amount[1, 2003], which returns the sales amount for that month and year.

Image After MEASURES come three lines that compute the future sales for January, February, and March of 2004:

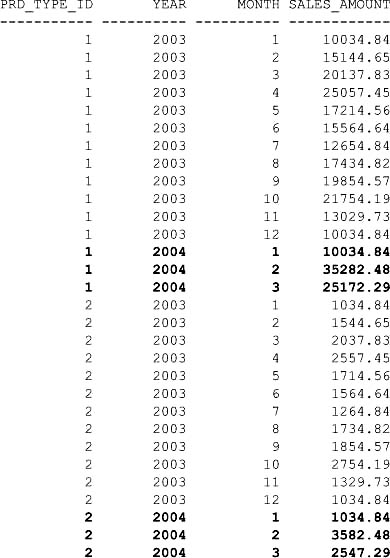
Image sales\_amount[1, 2004] = sales\_amount[1, 2003] sets the sales amount for January 2004 to the amount for January 2003.

Image sales\_amount[2, 2004] = sales\_amount[2, 2003] + sales\_amount[3, 2003] sets the sales amount for February 2004 to the amount for February 2003 plus March 2003.

Image sales\_amount[3, 2004] = ROUND(sales\_amount[3, 2003] \* 1.25, 2) sets the sales amount for March 2004 to the rounded value of the sales amount for March 2003 multiplied by 1.25.

Image ORDER BY prd\_type\_id, year, month simply orders the results returned by the entire query.

The output from the query is shown in the following listing; notice that the results contain the sales amounts for all months in 2003 for product types #1 and #2, plus the predicted sales amounts for the first three months in 2004 (which I’ve made bold to make them stand out):



**Using Positional and Symbolic Notation to Access Cells**

In the previous example, you saw how to access a cell in an array using the following notation: sales\_amount[1, 2004], where 1 is the month and 2004 is the year. This is referred to as positional notation because the meaning of the dimensions is determined by their position: the first position contains the month and the second position contains the year.

You can also use symbolic notation to explicitly indicate the meaning of the dimensions, as in, for example, sales\_amount[month=1, year=2004]. The following query rewrites the previous query to use symbolic notation:

**SELECT prd\_type\_id, year, month, sales\_amount**

**FROM all\_sales**

**WHERE prd\_type\_id BETWEEN 1 AND 2**

**AND emp\_id = 21**

**MODEL**

**PARTITION BY (prd\_type\_id),**

**DIMENSION BY (month, year),**

**MEASURES (amount sales\_amount) (,**

**sales\_amount[month=1, year=2004] = sales\_amount[month=1, year=2003],**

**sales\_amount[month=2, year=2004] =,**

**sales\_amount[month=2, year=2003] + sales\_amount[month=3, year=2003],**

**sales\_amount[month=3, year=2004] =,**

**ROUND(sales\_amount[month=3, year=2003] \* 1.25, 2)**

**)**

**ORDER BY prd\_type\_id, year, month;**

When using positional or symbolic notation, it is important to be aware of the different way they handle null values in the dimensions. For example, sales\_amount[null, 2003] returns the amount whose month is null and year is 2003, but sales\_amount[month=null, year=2004] won’t access a valid cell because null=null always returns false.

**Accessing a Range of Cells Using BETWEEN and AND**

You can access a range of cells using the BETWEEN and AND keywords. For example, the following expression sets the sales amount for January 2004 to the rounded average of the sales between January and March of 2003:

sales\_amount[1, 2004] =

ROUND(AVG(sales\_amount)[month BETWEEN 1 AND 3, 2003], 2)

The following query shows the use of this expression:

**SELECT prd\_type\_id, year, month, sales\_amount**

**FROM all\_sales**

**WHERE prd\_type\_id BETWEEN 1 AND 2**

**AND emp\_id = 21**

**MODEL**

**PARTITION BY (prd\_type\_id)**

**DIMENSION BY (month, year)**

**MEASURES (amount sales\_amount) (**

**sales\_amount[1, 2004] =**

**ROUND(AVG(sales\_amount)[month BETWEEN 1 AND 3, 2003], 2)**

**)**

**ORDER BY prd\_type\_id, year, month;**

**Accessing All Cells Using ANY and IS ANY**

You can access all cells in an array using the ANY and IS ANY predicates. You use ANY with positional notation and IS ANY with symbolic notation. For example, the following expression sets the sales amount for January 2004 to the rounded sum of the sales for all months and years:

sales\_amount[1, 2004] =

ROUND(SUM(sales\_amount)[ANY, year IS ANY], 2)

The following query shows the use of this expression:

**SELECT prd\_type\_id, year, month, sales\_amount**

**FROM all\_sales**

**WHERE prd\_type\_id BETWEEN 1 AND 2**

**AND emp\_id = 21**

**MODEL**

**PARTITION BY (prd\_type\_id)**

**DIMENSION BY (month, year)**

**MEASURES (amount sales\_amount) (**

**sales\_amount[1, 2004] =**

**ROUND(SUM(sales\_amount)[ANY, year IS ANY], 2)**

**)**

**ORDER BY prd\_type\_id, year, month;**

**Getting the Current Value of a Dimension Using CURRENTV()**

You can get the current value of a dimension using the CURRENTV() function. For example, the following expression sets the sales amount for the first month of 2004 to 1.25 times the sales of the same month in 2003; notice the use of CURRENTV() to get the current month, which is 1:

sales\_amount[1, 2004] =

ROUND(sales\_amount[CURRENTV(), 2003] \* 1.25, 2)

The following query shows the use of this expression:

**SELECT prd\_type\_id, year, month, sales\_amount**

**FROM all\_sales**

**WHERE prd\_type\_id BETWEEN 1 AND 2**

**AND emp\_id = 21**

**MODEL**

**PARTITION BY (prd\_type\_id)**

**DIMENSION BY (month, year)**

**MEASURES (amount sales\_amount) (**

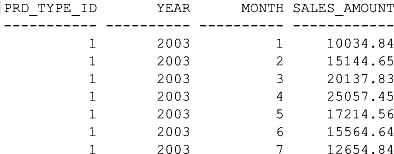
**sales\_amount[1, 2004] =**

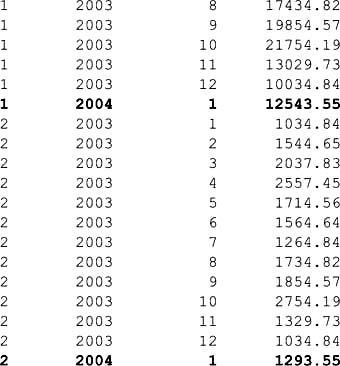
**ROUND(sales\_amount[CURRENTV(), 2003] \* 1.25, 2)**

**)**

**ORDER BY prd\_type\_id, year, month;**

The output from this query is as follows (I’ve highlighted the values for 2004 in bold):





**Accessing Cells Using a FOR Loop**

You can access cells using a FOR loop. For example, the following expression sets the sales amount for the first three months of 2004 to 1.25 times the sales of the same months in 2003; notice the use of the FOR loop and the INCREMENT keyword that specifies the amount to increment month by during each iteration of the loop:

sales\_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =

ROUND(sales\_amount[CURRENTV(), 2003] \* 1.25, 2)

The following query shows the use of this expression:

**SELECT prd\_type\_id, year, month, sales\_amount**

**FROM all\_sales**

**WHERE prd\_type\_id BETWEEN 1 AND 2**

**AND emp\_id = 21**

**MODEL**

**PARTITION BY (prd\_type\_id)**

**DIMENSION BY (month, year)**

**MEASURES (amount sales\_amount) (**

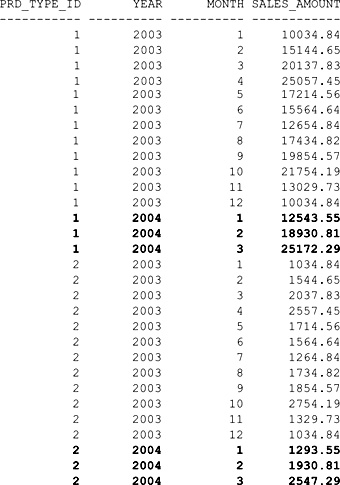
**sales\_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =**

**ROUND(sales\_amount[CURRENTV(), 2003] \* 1.25, 2)**

**)**

**ORDER BY prd\_type\_id, year, month;**

The output from this query is as follows (I’ve highlighted the values for 2004 in bold):



**Handling Null and Missing Values**

In this section, you’ll learn how to handle null and missing values using the MODEL clause.

**Using IS PRESENT**

IS PRESENT returns true if the row specified by the cell reference existed prior to the execution of the MODELclause. For example:

sales\_amount[CURRENTV(), 2003] IS PRESENT

will return true if sales\_amount[CURRENTV(), 2003] exists.

The following expression sets the sales amount for the first three months of 2004 to 1.25 multiplied by the sales of the same months in 2003:

sales\_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =

CASE WHEN sales\_amount[CURRENTV(), 2003] IS PRESENT THEN

ROUND(sales\_amount[CURRENTV(), 2003] \* 1.25, 2)

ELSE

0

END

The following query shows the use of this expression:

**SELECT prd\_type\_id, year, month, sales\_amount**

**FROM all\_sales**

**WHERE prd\_type\_id BETWEEN 1 AND 2**

**AND emp\_id = 21**

**MODEL**

**PARTITION BY (prd\_type\_id)**

**DIMENSION BY (month, year)**

**MEASURES (amount sales\_amount) (**

**sales\_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =**

**CASE WHEN sales\_amount[CURRENTV(), 2003] IS PRESENT THEN**

**ROUND(sales\_amount[CURRENTV(), 2003] \* 1.25, 2)**

**ELSE**

**0**

**END**

**)**

**ORDER BY prd\_type\_id, year, month;**

The output of this query is the same as the example in the previous section.

**Using PRESENTV()**

PRESENTV(*cell*, *expr1*, *expr2*) returns the expression *expr1* if the row specified by the *cell* reference existed prior to the execution of the MODEL clause. If the row doesn’t exist, the expression *expr2* is returned. For example:

PRESENTV(sales\_amount[CURRENTV(), 2003],

ROUND(sales\_amount[CURRENTV(), 2003] \* 1.25, 2), 0)

will return the rounded sales amount if sales\_amount[CURRENTV(), 2003] exists; otherwise 0 will be returned.

The following query shows the use of this expression:

**SELECT prd\_type\_id, year, month, sales\_amount**

**FROM all\_sales**

**WHERE prd\_type\_id BETWEEN 1 AND 2**

**AND emp\_id = 21**

**MODEL**

**PARTITION BY (prd\_type\_id)**

**DIMENSION BY (month, year)**

**MEASURES (amount sales\_amount) (**

**sales\_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =**

**PRESENTV(sales\_amount[CURRENTV(), 2003],**

**ROUND(sales\_amount[CURRENTV(), 2003] \* 1.25, 2), 0)**

**)**

**ORDER BY prd\_type\_id, year, month;**

**Using PRESENTNNV()**

PRESENTNNV(*cell, expr1, expr2*) returns the expression *expr1* if the row specified by the *cell*reference existed prior to the execution of the MODEL clause and the cell value is not null. If the row doesn’t exist or the cell value is null, the expression *expr2* is returned. For example,

PRESENTNNV(sales\_amount[CURRENTV(), 2003],

ROUND(sales\_amount[CURRENTV(), 2003] \* 1.25, 2), 0)

will return the rounded sales amount if sales\_amount[CURRENTV(), 2003] exists and is not null; otherwise 0 will be returned.

**Using IGNORE NAV and KEEP NAV**

IGNORE NAV returns

Image 0 for null or missing numeric values.

Image An empty string for null or missing string values.

Image 01-JAN-2000 for null or missing date values.

Image Null for all other database types.

KEEP NAV returns null for null or missing numeric values. Be aware that KEEP NAV is the default.

The following query shows the use of IGNORE NAV:

**SELECT prd\_type\_id, year, month, sales\_amount**

**FROM all\_sales**

**WHERE prd\_type\_id BETWEEN 1 AND 2**

**AND emp\_id = 21**

**MODEL IGNORE NAV**

**PARTITION BY (prd\_type\_id)**

**DIMENSION BY (month, year)**

**MEASURES (amount sales\_amount) (**

**sales\_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =**

**ROUND(sales\_amount[CURRENTV(), 2003] \* 1.25, 2)**

**)**

**ORDER BY prd\_type\_id, year, month;**

**Updating Existing Cells**

By default, if the cell referenced on the left side of an expression exists, then it is updated. If the cell doesn’t exist, then a new row in the array is created. You can change this default behavior using RULES UPDATE, which specifies that if the cell doesn’t exist, a new row will not be created.

The following query shows the use of RULES UPDATE:

**SELECT prd\_type\_id, year, month, sales\_amount**

**FROM all\_sales**

**WHERE prd\_type\_id BETWEEN 1 AND 2**

**AND emp\_id = 21**

**MODEL**

**PARTITION BY (prd\_type\_id)**

**DIMENSION BY (month, year)**

**MEASURES (amount sales\_amount)**

**RULES UPDATE (**

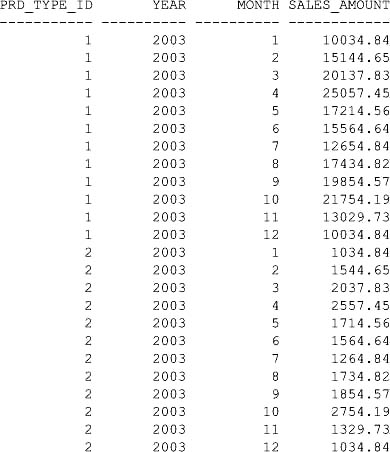
**sales\_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004] =**

**ROUND(sales\_amount[CURRENTV(), 2003] \* 1.25, 2)**

**)**

**ORDER BY prd\_type\_id, year, month;**

Because cells for 2004 don’t exist and RULES UPDATE is used, no new rows are created in the array for 2004; therefore, the query doesn’t return rows for 2004. The following listing shows the output for the query—notice there are no rows for 2004:



**USING THE PIVOT AND UNPIVOT CLAUSES**

The PIVOT clause is new for Oracle Database 11*g* and enables you to rotate rows into columns in the output from a query, and, at the same time, to run an aggregation function on the data. Oracle Database 11*g* also has an UNPIVOT clause that rotates columns into rows in the output from a query.

PIVOT and UNPIVOT are useful to see overall trends in large amounts of data, such as trends in sales over a period of time. You’ll see queries that show the use of PIVOT and UNPIVOT in the following sections.

**A Simple Example of the PIVOT Clause**

The easiest way to learn how to use the PIVOT clause is to see an example. The following query shows the total sales amount of product types #1, #2, and #3 for the first four months in 2003; notice that the cells in the query’s output show the sum of the sales amounts for each product type in each month:

**SELECT \***

**FROM (**

**SELECT month, prd\_type\_id, amount**

**FROM all\_sales**

**WHERE year = 2003**

**AND prd\_type\_id IN (1, 2, 3)**

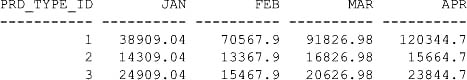
**)**

**PIVOT (**

**SUM(amount) FOR month IN (1 AS JAN, 2 AS FEB, 3 AS MAR, 4 AS APR)**

**)**

**ORDER BY prd\_type\_id;**



Starting with the first line of output, you can see there was

Image $38,909.04 of product type #1 sold in January.

Image $70,567.90 of product type #1 sold in February.

Image …and so on for the rest of the first line.

The second line of output shows there was

Image $14,309.04 of product type #2 sold in January.

Image $13,367.90 of product type #2 sold in February.

Image …and so on for the rest of the output.

Image

**NOTE**  
PIVOT *is a powerful tool that allows you to see trends in sales of types of products over a period of months. Based on such trends, a real store could use the information to alter their sales tactics and formulate new marketing campaigns*.

The previous SELECT statement has the following structure:

SELECT \*

FROM (

*inner\_query*

)

PIVOT (

*aggregate\_function* FOR *pivot\_column* IN (*list\_of\_values*)

)

ORDER BY…;

Let’s break down the previous example into the structural elements:

Image There is an inner and outer query. The inner query gets the month, product type, and amount from the all\_sales table and passes the results to the outer query.

Image SUM(amount) FOR month IN (1 AS JAN, 2 AS FEB, 3 AS MAR, 4 AS APR) is the line in the PIVOT clause.

Image The SUM() function adds up the sales amounts for the product types in the first four months (the months are listed in the IN part). Instead of returning the months as 1, 2, 3, and 4 in the output, the AS part renames the numbers to JAN, FEB, MAR, and APR to make the months more readable in the output.

Image The month column from the all\_sales table is used as the pivot column. This means that the months appear as columns in the output. In effect, the rows are rotated—or *pivoted*—to view the months as columns.

Image At the very end of the example, the ORDER BY prd\_type\_id line simply orders the results by the product type.

**Pivoting on Multiple Columns**

You can pivot on multiple columns by placing those columns in the FOR part of the PIVOT. The following example pivots on both the month and prd\_type\_id columns, which are referenced in the FOR part; notice that the list of values in the IN part of the PIVOT contains a value for the month and prd\_type\_id columns:

**SELECT \***

**FROM (**

**SELECT month, prd\_type\_id, amount**

**FROM all\_sales**

**WHERE year = 2003**

**AND prd\_type\_id IN (1, 2, 3)**

**)**

**PIVOT (**

**SUM(amount) FOR (month, prd\_type\_id) IN (**

**(1, 2) AS JAN\_PRDTYPE2,**

**(2, 3) AS FEB\_PRDTYPE3,**

**(3, 1) AS MAR\_PRDTYPE1,**

**(4, 2) AS APR\_PRDTYPE2**

**)**

**);**

JAN\_PRDTYPE2 FEB\_PRDTYPE3 MAR\_PRDTYPE1 APR\_PRDTYPE2

------------ ------------ ------------ ------------

14309.04 15467.9 91826.98 15664.7

The cells in the output show the sum of the sales amounts for each product type in the specified month (the product type and month to query are placed in the list of values in the IN part). As you can see from the query output, there were the following sales amounts:

Image $14,309.04 of product type #2 in January

Image $15,467.90 of product type #3 in February

Image $91,826.98 of product type #1 in March

Image $15,664.70 of product type #2 in April

You can put any values in the IN part to get the values of interest to you. In the following example, the values of the product types are shuffled in the IN part to get the sales for those product types in the specified months:

**SELECT \***

**FROM (**

**SELECT month, prd\_type\_id, amount**

**FROM all\_sales**

**WHERE year = 2003**

**AND prd\_type\_id IN (1, 2, 3)**

**)**

**PIVOT (**

**SUM(amount) FOR (month, prd\_type\_id) IN (**

**(1, 1) AS JAN\_PRDTYPE1,**

**(2, 2) AS FEB\_PRDTYPE2,**

**(3, 3) AS MAR\_PRDTYPE3,**

**(4, 1) AS APR\_PRDTYPE1**

**)**

**);**

JAN\_PRDTYPE1 FEB\_PRDTYPE2 MAR\_PRDTYPE3 APR\_PRDTYPE1

------------ ------------ ------------ ------------

38909.04 13367.9 20626.98 120344.7

As you can see from this output, there were the following sales amounts:

Image $38,909.04 of product type #1 in January

Image $13,367.90 of product type #2 in February

Image $20,626.98 of product type #3 in March

Image $120,344.70 of product type #1 in April

**Using Multiple Aggregate Functions in a Pivot**

You can use multiple aggregate functions in a pivot. For example, the following query uses SUM() to get the total sales for the product types in January and February and AVG() to get the averages of the sales:

**SELECT \***

**FROM (**

**SELECT month, prd\_type\_id, amount**

**FROM all\_sales**

**WHERE year = 2003**

**AND prd\_type\_id IN (1, 2, 3)**

**)**

**PIVOT (**

**SUM(amount) AS sum\_amount,**

**AVG(amount) AS avg\_amount**

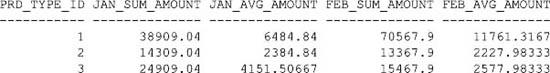
**FOR (month) IN (**

**1 AS JAN, 2 AS FEB**

**)**

**)**

**ORDER BY prd\_type\_id;**



As you can see, the first line of output shows for product type #1:

Image A total of $38,909.04 and an average of $6,484.84 sold in January

Image A total of $70,567.90 and an average of $11,761.32 sold in February

The second line of output shows for product type #2:

Image A total of $14,309.04 and an average of $2,384.84 sold in January

Image A total of $13,367.90 and an average of $2,227.98 sold in February

…and so on for the rest of the output.

**Using the UNPIVOT Clause**

The UNPIVOT clause rotates columns into rows. The examples in this section use the following table named pivot\_sales\_data (created by the store\_schema.sql script); pivot\_sales\_data is populated by a query that returns a pivoted version of the sales data:

CREATE TABLE pivot\_sales\_data AS

SELECT \*

FROM (

SELECT month, prd\_type\_id, amount

FROM all\_sales

WHERE year = 2003

AND prd\_type\_id IN (1, 2, 3)

)

PIVOT (

SUM(amount) FOR month IN (1 AS JAN, 2 AS FEB, 3 AS MAR, 4 AS APR)

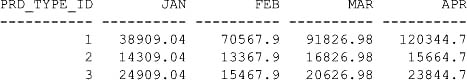
)

ORDER BY prd\_type\_id;

The following query returns the contents of the pivot\_sales\_data table:

**SELECT \***

**FROM pivot\_sales\_data;**



The next query uses UNPIVOT to get the sales data in an unpivoted form:

**SELECT \***

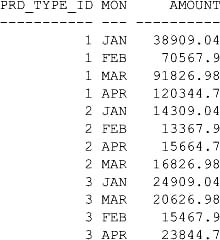
**FROM pivot\_sales\_data**

**UNPIVOT (**

**amount FOR month IN (JAN, FEB, MAR, APR)**

**)**

**ORDER BY prd\_type\_id;**



Notice that the query rotates the pivoted data. For example, the monthly sales totals that appear in the horizontal rows of pivot\_sales\_data are shown in the vertical AMOUNT column.

Image

**TIP**  
*Consider using* UNPIVOT *when you have a query that returns rows with many columns and you want to view those columns as rows*.

**SUMMARY**

In this chapter, you learned the following:

Image The set operators (UNION ALL, UNION, INTERSECT, and MINUS) allow you to combine rows returned by two or more queries.

Image TRANSLATE (*x, from\_string, to\_string*) translates characters in one string to characters in another string.

Image DECODE (*value, search\_value, result,default\_value*) compares *value* with *search\_value*. If the values are equal, DECODE() returns *search\_value*; otherwise *default\_value* is returned. DECODE() allows you to perform if-then-else logic in SQL.

Image CASE is similar to DECODE(). You should use CASE because it is ANSI-compliant.

Image Queries may be run against data that is organized into a hierarchy.

Image ROLLUP extends the GROUP BY clause to return a row containing a subtotal for each group of rows, plus a row containing a grand total for all the groups.

Image CUBE extends the GROUP BY clause to return rows containing a subtotal for all combinations of columns, plus a row containing the grand total.

Image The database has many built-in analytic functions that enable you to perform complex calculations, such as finding the top-selling product type for each month, the top salespersons, and so on.

Image The MODEL clause performs inter-row calculations and allows you to treat table data as an array. This gives you the ability to perform calculations in a similar manner to spreadsheet calculations.

Image The Oracle Database 11*g* PIVOT and UNPIVOT clauses are useful for seeing overall trends in large amounts of data.