CHAPTER 7

Advanced Queries
In this chapter, you will see how to

- Use the set operators, which allow you to combine rows returned by two or more queries.
- Use the `TRANSLATE()` function to translate characters in one string to characters in another string.
- Use the `DECODE()` function to search for a certain value in a set of values.
- Use the `CASE` expression to perform if-then-else logic in SQL.
- Perform queries on hierarchical data.
- Use the `ROLLUP` and `CUBE` clauses to get subtotals and totals for groups of rows.
- 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.
- Perform inter-row calculations with the `MODEL` clause.
- Use the new Oracle Database 11g 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 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 shortly, but first let's look at the example tables used in this section.

#### TABLE 7-1  Set Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNION ALL</td>
<td>Returns all the rows retrieved by the queries, including duplicate rows.</td>
</tr>
<tr>
<td>UNION</td>
<td>Returns all non-duplicate rows retrieved by the queries.</td>
</tr>
<tr>
<td>INTERSECT</td>
<td>Returns rows that are retrieved by both queries.</td>
</tr>
<tr>
<td>MINUS</td>
<td>Returns the remaining rows when the rows retrieved by the second query are subtracted from the rows retrieved by the first query.</td>
</tr>
</tbody>
</table>
Chapter 7: Advanced Queries

The Example Tables

The products and more_products tables are created by the store_schema.sql script using the following statements:

```sql
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:

```sql
SELECT product_id, product_type_id, name
FROM products;
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRODUCT_TYPE_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Modern Science</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Chemistry</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Supernova</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Tank War</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Z Files</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2412: The Return</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>Space Force 9</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>From Another Planet</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>Classical Music</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>Pop 3</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>Creative Yell</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>My Front Line</td>
</tr>
</tbody>
</table>

The next query retrieves the prd_id, prd_type_id, and name columns from the more_products table:

```sql
SELECT prd_id, prd_type_id, name
FROM more_products;
```
Oracle Database 11g SQL

<table>
<thead>
<tr>
<th>PRD_ID</th>
<th>PRD_TYPE_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Modern Science</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Chemistry</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>Supernova</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Lunar Landing</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Submarine</td>
</tr>
</tbody>
</table>

### 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:

```sql
SELECT product_id, product_type_id, name
FROM products
UNION ALL
SELECT prd_id, prd_type_id, name
FROM more_products;
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRODUCT_TYPE_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Modern Science</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Chemistry</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Supernova</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Tank War</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Z Files</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2412: The Return</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>Space Force 9</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>From Another Planet</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>Classical Music</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>Pop 3</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>Creative Yell</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>My Front Line</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Modern Science</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Chemistry</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Supernova</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Lunar Landing</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Submarine</td>
</tr>
</tbody>
</table>

17 rows selected.

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_id` and `prd_id`):
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;
```

```
PRODUCT_ID PRODUCT_TYPE_ID NAME
---------- --------------- -------------------
1 1 Modern Science
1 1 Modern Science
2 1 Chemistry
2 1 Chemistry
3 2 Supernova
3 Supernova
4 2 Tank War
4 2 Lunar Landing
5 2 Z Files
5 2 Submarine
6 2 2412: The Return
7 3 Space Force 9
8 3 From Another Planet
9 4 Classical Music
10 4 Pop 3
11 4 Creative Yell
12 My Front Line
```

15 rows selected.
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:

```sql
SELECT product_id, product_type_id, name
FROM products
INTERSECT
SELECT prd_id, prd_type_id, name
FROM more_products;
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRODUCT_TYPE_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Modern Science</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Chemistry</td>
</tr>
</tbody>
</table>

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:

```sql
SELECT product_id, product_type_id, name
FROM products
MINUS
SELECT prd_id, prd_type_id, name
FROM more_products;
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRODUCT_TYPE_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>Supernova</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Tank War</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Z Files</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>2412: The Return</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>Space Force 9</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>From Another Planet</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>Classical Music</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>Pop 3</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>Creative Yell</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>My Front Line</td>
</tr>
</tbody>
</table>

10 rows selected.

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;
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRODUCT_TYPE_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Modern Science</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>New Chemistry</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Supernova</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>Lunar Landing</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>Submarine</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>Airplane</td>
</tr>
</tbody>
</table>

The next query does the following:

- 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.)
- 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.)
- 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;
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRODUCT_TYPE_ID</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Modern Science</td>
</tr>
</tbody>
</table>
The following query has the parentheses set so that the \texttt{INTERSECT} is performed first; notice the different results returned by the query compared with the previous example:

\begin{verbatim}
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);
\end{verbatim}

\begin{tabular}{lll}
PRODUCT_ID & PRODUCT_TYPE_ID & NAME \\
---------- & --------------- & --------------- \\
1         & 1               & Modern Science \\
2         & 1               & Chemistry \\
3         & 2               & Supernova \\
4         & 2               & Tank War \\
5         & 2               & Z Files \\
6         & 2               & 2412: The Return \\
7         & 3               & Space Force 9 \\
8         & 3               & From Another Planet \\
9         & 4               & Classical Music \\
10        & 4               & Pop 3 \\
11        & 4               & Creative Yell \\
12        &                 & My Front Line \\
\end{tabular}

This concludes the discussion of the set operators.

\section*{Using the \texttt{TRANSLATE()} Function}

\texttt{TRANSLATE(x, from_string, to_string)} converts the occurrences of characters in \texttt{from_string} found in \texttt{x} to corresponding characters in \texttt{to_string}. The easiest way to understand how \texttt{TRANSLATE()} works is to see some examples.

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

\begin{verbatim}
SELECT TRANSLATE('SECRET MESSAGE: MEET ME IN THE PARK',
    'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
    'EFGHIJKLMNOPQRSTUVWXYZABCD')
FROM dual;
\end{verbatim}

\begin{verbatim}
WIGVIX QIWEKI: QIIX QI MR XLI TEVO
\end{verbatim}

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:

\begin{verbatim}
SELECT TRANSLATE('WIGVIX QIWEKI: QIIX QI MR XLI TEVO',
    'EFGHIJKLMNOPQRSTUVWXYZABCD',
    'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
FROM dual;
\end{verbatim}

\begin{verbatim}
MEET ME IN THE PARK
\end{verbatim}
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, 'ABCDEFGHIJKLMNOPQRSTUVWXYZ', 'EFGHIJKLMNOPQRSTUVWXYZABCDE');
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>TRANSLATE(NAME)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Qshivr Wgmirgi</td>
</tr>
<tr>
<td>2</td>
<td>Gliqmwxvc</td>
</tr>
<tr>
<td>3</td>
<td>Wyтивраze</td>
</tr>
<tr>
<td>4</td>
<td>Xero Aeв</td>
</tr>
<tr>
<td>5</td>
<td>D Jmpiw</td>
</tr>
<tr>
<td>6</td>
<td>2412: Xli Vixyvr</td>
</tr>
<tr>
<td>7</td>
<td>Wтеги Jsvgi 9</td>
</tr>
<tr>
<td>8</td>
<td>Jvsq Erxliv Tperix</td>
</tr>
<tr>
<td>9</td>
<td>Gпewwmgep Qywmg</td>
</tr>
<tr>
<td>10</td>
<td>Tst 3</td>
</tr>
<tr>
<td>11</td>
<td>Gviexmzi Cipp</td>
</tr>
<tr>
<td>12</td>
<td>Qc Jvsrx Pmri</td>
</tr>
</tbody>
</table>

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;
```

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.
NOTE

DECODE() is an old Oracle proprietary function, and therefore you should use CASE expressions instead if you are using Oracle Database 9i 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;
```

```
PRD_ID  A DECODE(AVAILABLE,'Y','PR
---------- - ------------------------
1 Y Product is available
2 Y Product is available
3 N Product is not available
4 N Product is not available
5 Y Product is available
```

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 dual;
```

```
DECODE(PRODUCT_TYPE_ID,1,'Book',2,'Video',3,'DVD',4,'CD','Magazine')
---------------
Book
Video
DVD
CD
Magazine
```
Using the CASE Expression

The CASE expression performs if-then-else logic in SQL and is supported in Oracle Database 9i 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:

- Simple case expressions, which use expressions to determine the returned value
- 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:

```sql
CASE search_expression
  WHEN expression1 THEN result1
  WHEN expression2 THEN result2
  ...
  WHEN expressionN THEN resultN
  ELSE default_result
END
```
194  Oracle Database 11g SQL

where

- `search_expression` is the expression to be evaluated.
- `expression1, expression2, ..., expressionN` are the expressions to be evaluated against `search_expression`.
- `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.
- `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;
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRODUCT_TYPE_ID</th>
<th>CASEPROD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Book</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Book</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Video</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Video</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Video</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>Video</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>DVD</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>DVD</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>CD</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>CD</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>CD</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>Magazine</td>
</tr>
</tbody>
</table>

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

- \( \text{condition1, condition2, \ldots, conditionN} \) are the expressions to be evaluated.
- \( \text{result1, result2, \ldots, resultN} \) are the returned results (one for each possible condition). If \( \text{condition1} \) is true, \( \text{result1} \) is returned, and similarly for the other expressions.
- \( \text{default_result} \) is returned when there is no condition that returns true.

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

```sql
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;
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRODUCT_TYPE_ID</th>
<th>CASEPROD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Book</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Book</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Video</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>Video</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Video</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>Video</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>DVD</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>DVD</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>CD</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>CD</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>CD</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>Magazine</td>
</tr>
</tbody>
</table>

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

```sql
SELECT product_id, price,
CASE
  WHEN price > 15 THEN 'Expensive'
  ELSE 'Cheap'
END
FROM products;
```

<table>
<thead>
<tr>
<th>PRODUCT_ID</th>
<th>PRICE</th>
<th>CASEWHENEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19.95</td>
<td>Expensive</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>Expensive</td>
</tr>
<tr>
<td>3</td>
<td>25.99</td>
<td>Expensive</td>
</tr>
<tr>
<td>4</td>
<td>13.95</td>
<td>Cheap</td>
</tr>
<tr>
<td>5</td>
<td>49.99</td>
<td>Expensive</td>
</tr>
</tbody>
</table>
You will see more advanced examples of CASE expressions later in this chapter and in Chapter 16.

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:

```sql
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_employees table; manager_id indicates the manager of an employee (if any). The following query returns the rows from more_employees:

```sql
SELECT *  
FROM more_employees;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>MANAGER_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>TITLE</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>James</td>
<td>Smith</td>
<td>CEO</td>
<td>800000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Ron</td>
<td>Johnson</td>
<td>Sales Manager</td>
<td>600000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Fred</td>
<td>Hobbs</td>
<td>Sales Person</td>
<td>200000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Susan</td>
<td>Jones</td>
<td>Support Manager</td>
<td>500000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Rob</td>
<td>Green</td>
<td>Sales Person</td>
<td>40000</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Jane</td>
<td>Brown</td>
<td>Support Person</td>
<td>450000</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>John</td>
<td>Grey</td>
<td>Support Manager</td>
<td>300000</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Jean</td>
<td>Blue</td>
<td>Support Person</td>
<td>290000</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 7: Advanced Queries

As you can see, it’s difficult to pick out the employee relationships from this data. Figure 7-1 shows the relationships in a graphical form.

As you can see from Figure 7-1, the elements—or nodes—form a tree. Trees of nodes have the following technical terms associated with them:

- **Root node**  The root is the node at the top of the tree. In the example shown in Figure 7-1, the root node is James Smith, the CEO.
- **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.
- **Child node**  A child is a node that has one parent node above it. For example, Ron Johnson’s parent is James Smith.
- **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

```sql
SELECT [LEVEL], column, expression, ...
FROM table
[WHERE where_clause]
[[START WITH start_condition] [CONNECT BY PRIOR prior_condition]];
```

where

- **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.
- **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.
- **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:

```sql
SELECT employee_id, manager_id, first_name, last_name
FROM more_employees
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>MANAGER_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>James</td>
<td>Smith</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Ron</td>
<td>Johnson</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>Fred</td>
<td>Hobbs</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>Rob</td>
<td>Green</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Susan</td>
<td>Jones</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>Jane</td>
<td>Brown</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>Henry</td>
<td>Heyson</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>John</td>
<td>Grey</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>Jean</td>
<td>Blue</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>Kevin</td>
<td>Black</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>Keith</td>
<td>Long</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>Frank</td>
<td>Howard</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>Doreen</td>
<td>Penn</td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>EMPLOYEE_ID</th>
<th>MANAGER_ID</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>James</td>
<td>Smith</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>Ron</td>
<td>Johnson</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>Susan</td>
<td>Jones</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>1</td>
<td>Kevin</td>
<td>Black</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>Fred</td>
<td>Hobbs</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>4</td>
<td>John</td>
<td>Grey</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>10</td>
<td>Frank</td>
<td>Howard</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>10</td>
<td>Doreen</td>
<td>Penn</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>10</td>
<td>1</td>
<td>Long</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
<td>Rob</td>
<td>Green</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>4</td>
<td>Jane</td>
<td>Brown</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>6</td>
<td>Henry</td>
<td>Heyson</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>7</td>
<td>Jean</td>
<td>Blue</td>
</tr>
</tbody>
</table>

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(DISTINCT LEVEL)
---------------------
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 employee
FROM more_employees
START WITH employee_id = 1
CONNECT BY PRIOR employee_id = manager_id;
```

```
LEVEL EMPLOYEE
----------- -------
1   James Smith
2    Ron Johnson
```
Oracle Database 11g SQL

3 Fred Hobbs
3 Rob Green
2 Susan Jones
3 Jane Brown
4 Henry Heyson
3 John Grey
4 Jean Blue
2 Kevin Black
3 Keith Long
3 Frank Howard
3 Doreen Penn

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 WITH clause. 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;
```

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Susan Jones</td>
</tr>
<tr>
<td>2</td>
<td>Jane Brown</td>
</tr>
<tr>
<td>3</td>
<td>Henry Heyson</td>
</tr>
<tr>
<td>2</td>
<td>John Grey</td>
</tr>
<tr>
<td>3</td>
<td>Jean Blue</td>
</tr>
</tbody>
</table>

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 (SELECT employee_id
              FROM employees
             WHERE first_name = 'Kevin' AND last_name = 'Black')
CONNECT BY PRIOR employee_id = manager_id;
```
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;

LEVEL EMPLOYEE
---------- ---------------
  1  Kevin Black
  2    Keith Long
  2    Frank Howard
  2    Doreen Penn

### 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 `LEVEL` returns 1 for Jean Blue, 2 for John Grey, and so on:

```sql
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;

LEVEL EMPLOYEE
---------- ------------------
  1 James Smith
  3     Fred Hobbs
```

### 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'`:

```sql
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;

LEVEL EMPLOYEE
---------- ------------------
  1 James Smith
  3     Fred Hobbs
```
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:

```sql
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';
```

<table>
<thead>
<tr>
<th>LEVEL</th>
<th>EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>James Smith</td>
</tr>
<tr>
<td>2</td>
<td>Susan Jones</td>
</tr>
<tr>
<td>3</td>
<td>Jane Brown</td>
</tr>
<tr>
<td>4</td>
<td>Henry Heyson</td>
</tr>
<tr>
<td>3</td>
<td>John Grey</td>
</tr>
<tr>
<td>4</td>
<td>Jean Blue</td>
</tr>
<tr>
<td>2</td>
<td>Kevin Black</td>
</tr>
<tr>
<td>3</td>
<td>Keith Long</td>
</tr>
<tr>
<td>3</td>
<td>Frank Howard</td>
</tr>
<tr>
<td>3</td>
<td>Doreen Penn</td>
</tr>
</tbody>
</table>

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:

```sql
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

- **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.

- **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:

- **divisions**, which stores the divisions within the company
- **jobs**, which stores the jobs within the company
- **employees2**, which stores the employees

These tables are created by the `store_schema.sql` script. The `divisions` table is created using the following statement:

```sql
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:

```sql
SELECT *  
FROM divisions;
```

<table>
<thead>
<tr>
<th>DIV</th>
<th>NAME</th>
<th>SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAL</td>
<td>Sales</td>
<td></td>
</tr>
</tbody>
</table>
The jobs table is created using the following statement:

```sql
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:

```sql
SELECT * 
FROM jobs;
```

The employees2 table is created using the following statement:

```sql
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:

```sql
SELECT * 
FROM employees2  
WHERE ROWNUM <= 5;
```

<table>
<thead>
<tr>
<th>EMPLOYEE_ID</th>
<th>DIV</th>
<th>JOB</th>
<th>FIRST_NAME</th>
<th>LAST_NAME</th>
<th>SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BUS</td>
<td>PRE</td>
<td>James</td>
<td>Smith</td>
<td>800000</td>
</tr>
<tr>
<td>2</td>
<td>SAL</td>
<td>MGR</td>
<td>Ron</td>
<td>Johnson</td>
<td>350000</td>
</tr>
<tr>
<td>3</td>
<td>SAL</td>
<td>WOR</td>
<td>Fred</td>
<td>Hobbs</td>
<td>140000</td>
</tr>
<tr>
<td>4</td>
<td>SUP</td>
<td>MGR</td>
<td>Susan</td>
<td>Jones</td>
<td>200000</td>
</tr>
<tr>
<td>5</td>
<td>SAL</td>
<td>WOR</td>
<td>Rob</td>
<td>Green</td>
<td>350000</td>
</tr>
</tbody>
</table>
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, 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;
```

\[
\begin{array}{l|l}
\text{DIV} & \text{SUM(SALARY)} \\
\hline
\text{BUS} & 1610000 \\
\text{OPE} & 1320000 \\
\text{SAL} & 4936000 \\
\text{SUP} & 1015000 \\
\end{array}
\]

### 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;
```

\[
\begin{array}{l|l}
\text{DIV} & \text{SUM(SALARY)} \\
\hline
\text{BUS} & 1610000 \\
\text{OPE} & 1320000 \\
\text{SAL} & 4936000 \\
\text{SUP} & 1015000 \\
\text{8881000} & \\
\end{array}
\]

### 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

```
SELECT division_id, SUM(salary)
FROM employees2
GROUP BY ROLLUP(division_id, job_id)
ORDER BY division_id, job_id;
```

\[
\begin{array}{l|l|l}
\text{DIV} & \text{JOB} & \text{SUM(SALARY)} \\
\hline
\text{BUS} & \text{SA} & 1335000 \\
\text{OPE} & \text{FI} & 1320000 \\
\text{SAL} & \text{AC} & 4936000 \\
\text{SUP} & \text{TE} & 1015000 \\
\text{8881000} & & \\
\end{array}
\]
Oracle Database 11g SQL

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;
```

<table>
<thead>
<tr>
<th>DIV</th>
<th>JOB</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS</td>
<td>MGR</td>
<td>530000</td>
</tr>
<tr>
<td>BUS</td>
<td>PRE</td>
<td>800000</td>
</tr>
<tr>
<td>BUS</td>
<td>WOR</td>
<td>280000</td>
</tr>
<tr>
<td>BUS</td>
<td></td>
<td>1610000</td>
</tr>
<tr>
<td>OPE</td>
<td>ENG</td>
<td>245000</td>
</tr>
<tr>
<td>OPE</td>
<td>MGR</td>
<td>805000</td>
</tr>
<tr>
<td>OPE</td>
<td>WOR</td>
<td>270000</td>
</tr>
<tr>
<td>OPE</td>
<td></td>
<td>1320000</td>
</tr>
<tr>
<td>SAL</td>
<td>MGR</td>
<td>4446000</td>
</tr>
<tr>
<td>SAL</td>
<td>WOR</td>
<td>490000</td>
</tr>
<tr>
<td>SAL</td>
<td></td>
<td>4936000</td>
</tr>
<tr>
<td>SUP</td>
<td>MGR</td>
<td>465000</td>
</tr>
<tr>
<td>SUP</td>
<td>TEC</td>
<td>115000</td>
</tr>
<tr>
<td>SUP</td>
<td>WOR</td>
<td>435000</td>
</tr>
<tr>
<td>SUP</td>
<td></td>
<td>1015000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8881000</td>
</tr>
</tbody>
</table>

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;
```

<table>
<thead>
<tr>
<th>JOB</th>
<th>DIV</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENG</td>
<td>OPE</td>
<td>245000</td>
</tr>
<tr>
<td>ENG</td>
<td></td>
<td>245000</td>
</tr>
<tr>
<td>MGR</td>
<td>BUS</td>
<td>530000</td>
</tr>
<tr>
<td>MGR</td>
<td>OPE</td>
<td>805000</td>
</tr>
<tr>
<td>MGR</td>
<td>SAL</td>
<td>4446000</td>
</tr>
<tr>
<td>MGR</td>
<td>SUP</td>
<td>465000</td>
</tr>
<tr>
<td>MGR</td>
<td></td>
<td>6246000</td>
</tr>
<tr>
<td>PRE</td>
<td>BUS</td>
<td>800000</td>
</tr>
<tr>
<td>PRE</td>
<td></td>
<td>800000</td>
</tr>
<tr>
<td>TEC</td>
<td>SUP</td>
<td>115000</td>
</tr>
<tr>
<td>TEC</td>
<td></td>
<td>115000</td>
</tr>
<tr>
<td>WOR</td>
<td>BUS</td>
<td>280000</td>
</tr>
<tr>
<td>WOR</td>
<td>OPE</td>
<td>270000</td>
</tr>
</tbody>
</table>
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 in Chapter 4). 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;
```

<table>
<thead>
<tr>
<th>DIV JOB AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS MGR 176666.667</td>
</tr>
<tr>
<td>BUS PRE 800000</td>
</tr>
<tr>
<td>BUS WOR 280000</td>
</tr>
<tr>
<td>BUS 322000</td>
</tr>
<tr>
<td>OPE ENG 245000</td>
</tr>
<tr>
<td>OPE MGR 201250</td>
</tr>
<tr>
<td>OPE WOR 135000</td>
</tr>
<tr>
<td>OPE 188571.429</td>
</tr>
<tr>
<td>SAL MGR 261529.412</td>
</tr>
<tr>
<td>SAL WOR 245000</td>
</tr>
<tr>
<td>SAL 259789.474</td>
</tr>
<tr>
<td>SUP MGR 232500</td>
</tr>
<tr>
<td>SUP TEC 115000</td>
</tr>
<tr>
<td>SUP WOR 145000</td>
</tr>
<tr>
<td>SUP 169166.667</td>
</tr>
<tr>
<td>240027.027</td>
</tr>
</tbody>
</table>

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;
```

<table>
<thead>
<tr>
<th>DIV JOB SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS MGR 530000</td>
</tr>
<tr>
<td>BUS PRE 800000</td>
</tr>
<tr>
<td>BUS WOR 280000</td>
</tr>
<tr>
<td>BUS 1610000</td>
</tr>
<tr>
<td>SUP MGR 232500</td>
</tr>
<tr>
<td>SUP TEC 115000</td>
</tr>
<tr>
<td>SUP WOR 145000</td>
</tr>
<tr>
<td>SUP 169166.667</td>
</tr>
<tr>
<td>240027.027</td>
</tr>
</tbody>
</table>
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:

```sql
SELECT job_id, division_id, SUM(salary)
FROM employees2
GROUP BY CUBE(job_id, division_id)
ORDER BY job_id, division_id;
```

<table>
<thead>
<tr>
<th>JOB</th>
<th>DIV</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENG</td>
<td>OPE</td>
<td>245000</td>
</tr>
<tr>
<td>ENG</td>
<td></td>
<td>245000</td>
</tr>
<tr>
<td>MGR</td>
<td>BUS</td>
<td>530000</td>
</tr>
<tr>
<td>MGR</td>
<td>OPE</td>
<td>805000</td>
</tr>
<tr>
<td>MGR</td>
<td>SAL</td>
<td>4446000</td>
</tr>
<tr>
<td>MGR</td>
<td>SUP</td>
<td>465000</td>
</tr>
<tr>
<td>MGR</td>
<td></td>
<td>6246000</td>
</tr>
<tr>
<td>PRE</td>
<td>BUS</td>
<td>800000</td>
</tr>
<tr>
<td>PRE</td>
<td></td>
<td>800000</td>
</tr>
<tr>
<td>TEC</td>
<td>SUP</td>
<td>115000</td>
</tr>
<tr>
<td>TEC</td>
<td></td>
<td>115000</td>
</tr>
<tr>
<td>WOR</td>
<td>BUS</td>
<td>280000</td>
</tr>
<tr>
<td>WOR</td>
<td>OPE</td>
<td>270000</td>
</tr>
<tr>
<td>WOR</td>
<td>SAL</td>
<td>490000</td>
</tr>
<tr>
<td>WOR</td>
<td>SUP</td>
<td>435000</td>
</tr>
<tr>
<td>WOR</td>
<td></td>
<td>1475000</td>
</tr>
<tr>
<td>BUS</td>
<td></td>
<td>1610000</td>
</tr>
<tr>
<td>OPE</td>
<td></td>
<td>1320000</td>
</tr>
<tr>
<td>SAL</td>
<td></td>
<td>4936000</td>
</tr>
<tr>
<td>SUP</td>
<td></td>
<td>1015000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8881000</td>
</tr>
</tbody>
</table>
Using the GROUPING() Function

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

Using \( \text{GROUPING()} \) 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;
```

<table>
<thead>
<tr>
<th>DIV</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS</td>
<td>1610000</td>
</tr>
<tr>
<td>OPE</td>
<td>1320000</td>
</tr>
<tr>
<td>SAL</td>
<td>4936000</td>
</tr>
<tr>
<td>SUP</td>
<td>1015000</td>
</tr>
<tr>
<td></td>
<td>8881000</td>
</tr>
</tbody>
</table>

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

```
SELECT GROUPING(division_id), division_id, SUM(salary)
FROM employees2
GROUP BY ROLLUP(division_id)
ORDER BY division_id;
```

<table>
<thead>
<tr>
<th>GROUPING(DIVISION_ID)</th>
<th>DIV</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BUS</td>
<td>1610000</td>
</tr>
<tr>
<td></td>
<td>OPE</td>
<td>1320000</td>
</tr>
<tr>
<td></td>
<td>SAL</td>
<td>4936000</td>
</tr>
<tr>
<td></td>
<td>SUP</td>
<td>1015000</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>8881000</td>
</tr>
</tbody>
</table>

Using CASE to Convert the Returned Value from GROUPING()

You can use the \( \text{CASE} \) expression to convert the 1 in the previous example to a meaningful value. The following example uses \( \text{CASE} \) 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;
```
FROM employees2
GROUP BY ROLLUP(division_id)
ORDER BY division_id;

<table>
<thead>
<tr>
<th>DIV</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS</td>
<td>1610000</td>
</tr>
<tr>
<td>OPE</td>
<td>1320000</td>
</tr>
<tr>
<td>SAL</td>
<td>4936000</td>
</tr>
<tr>
<td>SUP</td>
<td>1015000</td>
</tr>
<tr>
<td>All divisions</td>
<td>8881000</td>
</tr>
</tbody>
</table>

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;
```

<table>
<thead>
<tr>
<th>DIV</th>
<th>JOB</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS</td>
<td>MGR</td>
<td>530000</td>
</tr>
<tr>
<td>BUS</td>
<td>PRE</td>
<td>800000</td>
</tr>
<tr>
<td>BUS</td>
<td>WOR</td>
<td>280000</td>
</tr>
<tr>
<td>BUS</td>
<td>All jobs</td>
<td>1610000</td>
</tr>
<tr>
<td>OPE</td>
<td>ENG</td>
<td>245000</td>
</tr>
<tr>
<td>OPE</td>
<td>MGR</td>
<td>805000</td>
</tr>
<tr>
<td>OPE</td>
<td>WOR</td>
<td>270000</td>
</tr>
<tr>
<td>OPE</td>
<td>All jobs</td>
<td>1320000</td>
</tr>
<tr>
<td>SAL</td>
<td>MGR</td>
<td>4446000</td>
</tr>
<tr>
<td>SAL</td>
<td>WOR</td>
<td>490000</td>
</tr>
<tr>
<td>SAL</td>
<td>All jobs</td>
<td>4936000</td>
</tr>
<tr>
<td>SUP</td>
<td>MGR</td>
<td>465000</td>
</tr>
<tr>
<td>SUP</td>
<td>TEC</td>
<td>115000</td>
</tr>
<tr>
<td>SUP</td>
<td>WOR</td>
<td>435000</td>
</tr>
<tr>
<td>SUP</td>
<td>All jobs</td>
<td>1015000</td>
</tr>
<tr>
<td>All divisions All jobs</td>
<td>8881000</td>
<td></td>
</tr>
</tbody>
</table>

Using GROUPING() with CUBE

You can use the GROUPING() function with CUBE, as in this example:
### Chapter 7: Advanced Queries

#### 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`:

```sql
SELECT division_id, job_id, SUM(salary)
FROM employees2
GROUP BY GROUPING_SETS(division_id, job_id)
ORDER BY division_id, job_id;
```

<table>
<thead>
<tr>
<th>DIV</th>
<th>JOB</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS</td>
<td>MGR</td>
<td>5300000</td>
</tr>
<tr>
<td>BUS</td>
<td>PRE</td>
<td>8000000</td>
</tr>
<tr>
<td>BUS</td>
<td>WOR</td>
<td>2800000</td>
</tr>
<tr>
<td>BUS</td>
<td>All jobs</td>
<td>1610000</td>
</tr>
<tr>
<td>OPE</td>
<td>ENG</td>
<td>2450000</td>
</tr>
<tr>
<td>OPE</td>
<td>MGR</td>
<td>8050000</td>
</tr>
<tr>
<td>OPE</td>
<td>WOR</td>
<td>2700000</td>
</tr>
<tr>
<td>OPE</td>
<td>All jobs</td>
<td>1320000</td>
</tr>
<tr>
<td>SAL</td>
<td>MGR</td>
<td>4446000</td>
</tr>
<tr>
<td>SAL</td>
<td>WOR</td>
<td>4900000</td>
</tr>
<tr>
<td>SAL</td>
<td>All jobs</td>
<td>4936000</td>
</tr>
<tr>
<td>SUP</td>
<td>MGR</td>
<td>4650000</td>
</tr>
<tr>
<td>SUP</td>
<td>TEC</td>
<td>1150000</td>
</tr>
<tr>
<td>SUP</td>
<td>WOR</td>
<td>4350000</td>
</tr>
<tr>
<td>SUP</td>
<td>All jobs</td>
<td>1015000</td>
</tr>
<tr>
<td>All divisions</td>
<td>ENG</td>
<td>2450000</td>
</tr>
<tr>
<td>All divisions</td>
<td>MGR</td>
<td>6246000</td>
</tr>
<tr>
<td>All divisions</td>
<td>PRE</td>
<td>8000000</td>
</tr>
<tr>
<td>All divisions</td>
<td>TEC</td>
<td>1150000</td>
</tr>
<tr>
<td>All divisions</td>
<td>WOR</td>
<td>1475000</td>
</tr>
<tr>
<td>All divisions</td>
<td>All jobs</td>
<td>8881000</td>
</tr>
</tbody>
</table>
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.

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:

- 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.
- 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.
- 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.
- 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.

<table>
<thead>
<tr>
<th>division_id</th>
<th>job_id</th>
<th>Bit Vector</th>
<th>GROUPING_ID() Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-null</td>
<td>non-null</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>non-null</td>
<td>null</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>null</td>
<td>non-null</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
<td>11</td>
<td>3</td>
</tr>
</tbody>
</table>
Chapter 7: Advanced Queries

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:

```sql
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;
```

<table>
<thead>
<tr>
<th>DIV</th>
<th>JOB</th>
<th>DIV_GRP</th>
<th>JOB_GRP</th>
<th>GRP_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
<td>-------</td>
<td>-------</td>
<td>------</td>
<td>-----------</td>
</tr>
<tr>
<td>BUS MGR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>530000</td>
<td></td>
</tr>
<tr>
<td>BUS PRE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>800000</td>
<td></td>
</tr>
<tr>
<td>BUS WOR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>280000</td>
<td></td>
</tr>
<tr>
<td>BUS</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1610000</td>
<td></td>
</tr>
<tr>
<td>OPE ENG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>245000</td>
<td></td>
</tr>
<tr>
<td>OPE MGR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>805000</td>
<td></td>
</tr>
<tr>
<td>OPE WOR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>270000</td>
<td></td>
</tr>
<tr>
<td>OPE</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1320000</td>
<td></td>
</tr>
<tr>
<td>SAL MGR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4446000</td>
<td></td>
</tr>
<tr>
<td>SAL WOR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>490000</td>
<td></td>
</tr>
<tr>
<td>SAL</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>4936000</td>
<td></td>
</tr>
<tr>
<td>SUP MGR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>465000</td>
<td></td>
</tr>
<tr>
<td>SUP TEC</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>115000</td>
<td></td>
</tr>
<tr>
<td>SUP WOR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>435000</td>
<td></td>
</tr>
<tr>
<td>SUP</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1015000</td>
<td></td>
</tr>
<tr>
<td>ENG</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>245000</td>
<td></td>
</tr>
<tr>
<td>MGR</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>6246000</td>
<td></td>
</tr>
<tr>
<td>PRE</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>800000</td>
<td></td>
</tr>
<tr>
<td>TEC</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>115000</td>
<td></td>
</tr>
<tr>
<td>WOR</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1475000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>8881000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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:

```sql
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;

<table>
<thead>
<tr>
<th>DIV</th>
<th>JOB</th>
<th>GRP_ID</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS</td>
<td></td>
<td>1</td>
<td>1610000</td>
</tr>
<tr>
<td>OPE</td>
<td></td>
<td>1</td>
<td>1320000</td>
</tr>
<tr>
<td>SAL</td>
<td></td>
<td>1</td>
<td>4936000</td>
</tr>
<tr>
<td>SUP</td>
<td></td>
<td>1</td>
<td>1015000</td>
</tr>
<tr>
<td></td>
<td>ENG</td>
<td>2</td>
<td>245000</td>
</tr>
<tr>
<td></td>
<td>MGR</td>
<td>2</td>
<td>6246000</td>
</tr>
<tr>
<td></td>
<td>PRE</td>
<td>2</td>
<td>800000</td>
</tr>
<tr>
<td></td>
<td>TEC</td>
<td>2</td>
<td>115000</td>
</tr>
<tr>
<td></td>
<td>WOR</td>
<td>2</td>
<td>1475000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>8881000</td>
</tr>
</tbody>
</table>

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);
```

<table>
<thead>
<tr>
<th>DIV</th>
<th>JOB</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS</td>
<td>MGR</td>
<td>530000</td>
</tr>
<tr>
<td>BUS</td>
<td>PRE</td>
<td>800000</td>
</tr>
<tr>
<td>BUS</td>
<td>WOR</td>
<td>280000</td>
</tr>
<tr>
<td>OPE</td>
<td>ENG</td>
<td>245000</td>
</tr>
<tr>
<td>OPE</td>
<td>MGR</td>
<td>805000</td>
</tr>
<tr>
<td>OPE</td>
<td>WOR</td>
<td>270000</td>
</tr>
<tr>
<td>SAL</td>
<td>MGR</td>
<td>4446000</td>
</tr>
<tr>
<td>SAL</td>
<td>WOR</td>
<td>490000</td>
</tr>
<tr>
<td>SUP</td>
<td>MGR</td>
<td>465000</td>
</tr>
<tr>
<td>SUP</td>
<td>TEC</td>
<td>115000</td>
</tr>
<tr>
<td>SUP</td>
<td>WOR</td>
<td>435000</td>
</tr>
<tr>
<td>BUS</td>
<td></td>
<td>1610000</td>
</tr>
<tr>
<td>OPE</td>
<td></td>
<td>1320000</td>
</tr>
<tr>
<td>SAL</td>
<td></td>
<td>4936000</td>
</tr>
<tr>
<td>SUP</td>
<td></td>
<td>1015000</td>
</tr>
</tbody>
</table>

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.
Chapter 7: Advanced Queries

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:

```sql
SELECT division_id, job_id, GROUP_ID(), SUM(salary)
FROM employees2
GROUP BY division_id, ROLLUP(division_id, job_id);
```

<table>
<thead>
<tr>
<th>DIV</th>
<th>JOB</th>
<th>GROUP_ID()</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS</td>
<td>MGR</td>
<td>0</td>
<td>530000</td>
</tr>
<tr>
<td>BUS</td>
<td>PRE</td>
<td>0</td>
<td>800000</td>
</tr>
<tr>
<td>BUS</td>
<td>WOR</td>
<td>0</td>
<td>280000</td>
</tr>
<tr>
<td>OPE</td>
<td>ENG</td>
<td>0</td>
<td>245000</td>
</tr>
<tr>
<td>OPE</td>
<td>MGR</td>
<td>0</td>
<td>805000</td>
</tr>
<tr>
<td>OPE</td>
<td>WOR</td>
<td>0</td>
<td>270000</td>
</tr>
<tr>
<td>SAL</td>
<td>MGR</td>
<td>0</td>
<td>4446000</td>
</tr>
<tr>
<td>SAL</td>
<td>WOR</td>
<td>0</td>
<td>490000</td>
</tr>
<tr>
<td>SUP</td>
<td>MGR</td>
<td>0</td>
<td>465000</td>
</tr>
<tr>
<td>SUP</td>
<td>TEC</td>
<td>0</td>
<td>115000</td>
</tr>
<tr>
<td>SUP</td>
<td>WOR</td>
<td>0</td>
<td>435000</td>
</tr>
<tr>
<td>BUS</td>
<td></td>
<td>0</td>
<td>1610000</td>
</tr>
<tr>
<td>OPE</td>
<td></td>
<td>0</td>
<td>1320000</td>
</tr>
<tr>
<td>SAL</td>
<td></td>
<td>0</td>
<td>4936000</td>
</tr>
<tr>
<td>SUP</td>
<td></td>
<td>0</td>
<td>1015000</td>
</tr>
<tr>
<td>BUS</td>
<td></td>
<td>1</td>
<td>1610000</td>
</tr>
<tr>
<td>OPE</td>
<td></td>
<td>1</td>
<td>1320000</td>
</tr>
<tr>
<td>SAL</td>
<td></td>
<td>1</td>
<td>4936000</td>
</tr>
<tr>
<td>SUP</td>
<td></td>
<td>1</td>
<td>1015000</td>
</tr>
</tbody>
</table>

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

```sql
SELECT division_id, job_id, GROUP_ID(), SUM(salary)
FROM employees2
GROUP BY division_id, ROLLUP(division_id, job_id)
HAVING GROUP_ID() = 0;
```

<table>
<thead>
<tr>
<th>DIV</th>
<th>JOB</th>
<th>GROUP_ID()</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS</td>
<td>MGR</td>
<td>0</td>
<td>530000</td>
</tr>
<tr>
<td>BUS</td>
<td>PRE</td>
<td>0</td>
<td>800000</td>
</tr>
<tr>
<td>BUS</td>
<td>WOR</td>
<td>0</td>
<td>280000</td>
</tr>
<tr>
<td>OPE</td>
<td>ENG</td>
<td>0</td>
<td>245000</td>
</tr>
<tr>
<td>OPE</td>
<td>MGR</td>
<td>0</td>
<td>805000</td>
</tr>
<tr>
<td>OPE</td>
<td>WOR</td>
<td>0</td>
<td>270000</td>
</tr>
</tbody>
</table>

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

- **Ranking functions** enable you to calculate ranks, percentiles, and n-tiles (tertiles, quartiles, and so on).
- **Inverse percentile functions** enable you to calculate the value that corresponds to a percentile.
- **Window functions** enable you to calculate cumulative and moving aggregates.
- **Reporting functions** enable you to calculate things like market share.
- **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.
- **First and last functions** enable you to get the first and last values in an ordered group.
- **Linear regression functions** enable you to fit an ordinary-least-squares regression line to a set of number pairs.
- **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:

```sql
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
```

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAL MGR</td>
<td>Salary Manager</td>
<td>4446000</td>
</tr>
<tr>
<td>SAL WOR</td>
<td>Salary Worker</td>
<td>490000</td>
</tr>
<tr>
<td>SUP MGR</td>
<td>Supervisor Manager</td>
<td>465000</td>
</tr>
<tr>
<td>SUP TEC</td>
<td>Supervisor Technical</td>
<td>115000</td>
</tr>
<tr>
<td>SUP WOR</td>
<td>Supervisor Worker</td>
<td>435000</td>
</tr>
<tr>
<td>BUS</td>
<td>Business</td>
<td>1610000</td>
</tr>
<tr>
<td>OPE</td>
<td>Operations</td>
<td>1320000</td>
</tr>
<tr>
<td>SAL</td>
<td>Sales</td>
<td>4936000</td>
</tr>
<tr>
<td>SUP</td>
<td>Supervisor</td>
<td>1015000</td>
</tr>
</tbody>
</table>
As you can see, the all_sales table contains five columns, which are as follows:

- **YEAR** stores the year the sales took place.
- **MONTH** stores the month the sales took place (1 to 12).
- **PRD_TYPE_ID** stores the product_type_id of the product.
- **EMP_ID** stores the employee_id of the employee who handled the sales.
- **AMOUNT** stores the total dollar amount of the sales.

The following query retrieves the first 12 rows from the all_sales table:

```sql
SELECT * FROM all_sales WHERE ROWNUM <= 12;
```

<table>
<thead>
<tr>
<th>YEAR</th>
<th>MONTH</th>
<th>PRD_TYPE_ID</th>
<th>EMP_ID</th>
<th>AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>2003</td>
<td>1</td>
<td>1</td>
<td>21</td>
<td>10034.84</td>
</tr>
<tr>
<td>2003</td>
<td>2</td>
<td>1</td>
<td>21</td>
<td>15144.65</td>
</tr>
<tr>
<td>2003</td>
<td>3</td>
<td>1</td>
<td>21</td>
<td>20137.83</td>
</tr>
<tr>
<td>2003</td>
<td>4</td>
<td>1</td>
<td>21</td>
<td>25057.45</td>
</tr>
<tr>
<td>2003</td>
<td>5</td>
<td>1</td>
<td>21</td>
<td>17214.56</td>
</tr>
<tr>
<td>2003</td>
<td>6</td>
<td>1</td>
<td>21</td>
<td>15564.64</td>
</tr>
<tr>
<td>2003</td>
<td>7</td>
<td>1</td>
<td>21</td>
<td>12654.84</td>
</tr>
<tr>
<td>2003</td>
<td>8</td>
<td>1</td>
<td>21</td>
<td>17434.82</td>
</tr>
<tr>
<td>2003</td>
<td>9</td>
<td>1</td>
<td>21</td>
<td>19854.57</td>
</tr>
<tr>
<td>2003</td>
<td>10</td>
<td>1</td>
<td>21</td>
<td>21754.19</td>
</tr>
<tr>
<td>2003</td>
<td>11</td>
<td>1</td>
<td>21</td>
<td>13029.73</td>
</tr>
<tr>
<td>2003</td>
<td>12</td>
<td>1</td>
<td>21</td>
<td>10034.84</td>
</tr>
</tbody>
</table>

**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.

Let’s examine the RANK() and DENSE_RANK() functions first.
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:

```sql
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;
```

<table>
<thead>
<tr>
<th>PRD_TYPE_ID</th>
<th>SUM(AMOUNT)</th>
<th>RANK</th>
<th>DENSE_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>905081.84</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>186381.22</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>478270.91</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>402751.16</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

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.

---

**TABLE 7-2  The Ranking Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RANK()</code></td>
<td>Returns the rank of items in a group. <code>RANK()</code> leaves a gap in the sequence of rankings in the event of a tie.</td>
</tr>
<tr>
<td><code>DENSE_RANK()</code></td>
<td>Returns the rank of items in a group. <code>DENSE_RANK()</code> doesn't leave a gap in the sequence of rankings in the event of a tie.</td>
</tr>
<tr>
<td><code>CUME_DIST()</code></td>
<td>Returns the position of a specified value relative to a group of values. <code>CUME_DIST()</code> is short for cumulative distribution.</td>
</tr>
<tr>
<td><code>PERCENT_RANK()</code></td>
<td>Returns the percent rank of a value relative to a group of values.</td>
</tr>
<tr>
<td><code>NTILE()</code></td>
<td>Returns n-tiles: tertiles, quartiles, and so on.</td>
</tr>
<tr>
<td><code>ROW_NUMBER()</code></td>
<td>Returns a number with each row in a group.</td>
</tr>
</tbody>
</table>
The all_sales table actually contains nulls in the AMOUNT column for all rows whose PRD_TYPE_ID column 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 WHERE clause:

```sql
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;
```

<table>
<thead>
<tr>
<th>PRD_TYPE_ID</th>
<th>SUM(AMOUNT)</th>
<th>RANK</th>
<th>DENSE_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>905081.84</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>186381.22</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>478270.91</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>402751.16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>NULL</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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:

```sql
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;
```

<table>
<thead>
<tr>
<th>PRD_TYPE_ID</th>
<th>SUM(AMOUNT)</th>
<th>RANK</th>
<th>DENSE_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>905081.84</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>186381.22</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>478270.91</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>402751.16</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>NULL</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Notice that the rank of null values is the lowest in the above query.
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:

```sql
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;
```

<table>
<thead>
<tr>
<th>PRD_TYPE_ID</th>
<th>SUM(AMOUNT)</th>
<th>RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>905081.84</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>186381.22</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>478270.91</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>402751.16</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>1972485.13</td>
<td>2</td>
</tr>
</tbody>
</table>

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

```sql
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;
```

<table>
<thead>
<tr>
<th>PRD_TYPE_ID</th>
<th>EMP_ID</th>
<th>SUM(AMOUNT)</th>
<th>RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td>197916.96</td>
<td>19</td>
</tr>
<tr>
<td>1</td>
<td>22</td>
<td>214216.96</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>23</td>
<td>98896.96</td>
<td>26</td>
</tr>
<tr>
<td>1</td>
<td>24</td>
<td>207216.96</td>
<td>18</td>
</tr>
<tr>
<td>1</td>
<td>25</td>
<td>93416.96</td>
<td>28</td>
</tr>
<tr>
<td>1</td>
<td>26</td>
<td>93417.04</td>
<td>27</td>
</tr>
</tbody>
</table>
The next query uses `GROUPING SETS` and `RANK()` to get just the sales amount subtotal rankings:

```sql
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:

```sql
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(bucket) 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:

```sql
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 \texttt{ROW\_NUMBER()} Function

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

\begin{verbatim}
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;
\end{verbatim}

\begin{table}
\begin{tabular}{lrr}
\hline
PRD\_TYPE\_ID & SUM(AMOUNT) & NTILE \\
\hline
1 & 905081.84 & 1 \\
2 & 186381.22 & 4 \\
3 & 478270.91 & 2 \\
4 & 402751.16 & 3 \\
\hline
\end{tabular}
\end{table}

This concludes the discussion of ranking functions.

Using the Inverse Percentile Functions

In the section “Using the \texttt{CUME\_DIST()} and \texttt{PERCENT\_RANK()} Functions,” you saw that \texttt{CUME\_DIST()} is used to calculate the position of a specified value relative to a group of values. You also saw that \texttt{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: \texttt{PERCENTILE\_DISC(x)} and \texttt{PERCENTILE\_CONT(x)}. They operate in a manner the reverse of \texttt{CUME\_DIST()} and \texttt{PERCENT\_RANK()}. \texttt{PERCENTILE\_DISC(x)} examines the cumulative distribution values in each group until it finds one that is greater than or equal to \(x\). \texttt{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 \texttt{PERCENTILE\_CONT()} and \texttt{PERCENTILE\_DISC()} to get the sum of the amount whose percentile is greater than or equal to 0.6:

\begin{verbatim}
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;
\end{verbatim}
PERCENTILE_CONT   PERCENTILE_DISC
------------------- -------------------
 417855.11       402751.16

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. 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:

```sql
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;
```

<table>
<thead>
<tr>
<th>MONTH</th>
<th>MONTH_AMOUNT</th>
<th>CUMULATIVE_AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>95525.55</td>
<td>95525.55</td>
</tr>
<tr>
<td>2</td>
<td>116671.6</td>
<td>212197.15</td>
</tr>
<tr>
<td>3</td>
<td>160307.92</td>
<td>372505.07</td>
</tr>
<tr>
<td>4</td>
<td>175998.8</td>
<td>548503.87</td>
</tr>
<tr>
<td>5</td>
<td>154349.44</td>
<td>702853.31</td>
</tr>
<tr>
<td>6</td>
<td>124951.36</td>
<td>827804.67</td>
</tr>
<tr>
<td>7</td>
<td>170296.16</td>
<td>998100.83</td>
</tr>
<tr>
<td>8</td>
<td>212735.68</td>
<td>1210836.51</td>
</tr>
<tr>
<td>9</td>
<td>199609.68</td>
<td>1410446.19</td>
</tr>
<tr>
<td>10</td>
<td>264480.79</td>
<td>1674926.98</td>
</tr>
<tr>
<td>11</td>
<td>160221.98</td>
<td>1835148.96</td>
</tr>
<tr>
<td>12</td>
<td>137336.17</td>
<td>1972485.13</td>
</tr>
</tbody>
</table>

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

\[
\text{SUM(SUM(amount)) OVER (ORDER BY month ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cumulative_amount}
\]

Let's break down this expression:

- **SUM(amount)** computes the sum of an amount. The outer **SUM()** computes the cumulative amount.
- **ORDER BY month** orders the rows read by the query by month.
- **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 ROW** represents 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):

```sql
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;
```

<table>
<thead>
<tr>
<th>MONTH</th>
<th>MONTH_AMOUNT</th>
<th>CUMULATIVE_AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>124951.36</td>
<td>124951.36</td>
</tr>
<tr>
<td>7</td>
<td>170296.16</td>
<td>295247.52</td>
</tr>
</tbody>
</table>
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;
```

<table>
<thead>
<tr>
<th>MONTH</th>
<th>MONTH_AMOUNT</th>
<th>MOVING_AVERAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>95525.55</td>
<td>95525.55</td>
</tr>
<tr>
<td>2</td>
<td>116671.6</td>
<td>106098.575</td>
</tr>
<tr>
<td>3</td>
<td>160307.92</td>
<td>124168.357</td>
</tr>
<tr>
<td>4</td>
<td>175998.8</td>
<td>137125.968</td>
</tr>
<tr>
<td>5</td>
<td>154349.44</td>
<td>151831.94</td>
</tr>
<tr>
<td>6</td>
<td>124951.36</td>
<td>153901.88</td>
</tr>
<tr>
<td>7</td>
<td>170296.16</td>
<td>156398.94</td>
</tr>
<tr>
<td>8</td>
<td>212735.68</td>
<td>165583.16</td>
</tr>
<tr>
<td>9</td>
<td>199609.68</td>
<td>176898.22</td>
</tr>
<tr>
<td>10</td>
<td>264480.79</td>
<td>211780.578</td>
</tr>
<tr>
<td>11</td>
<td>160221.98</td>
<td>209262.033</td>
</tr>
<tr>
<td>12</td>
<td>137336.17</td>
<td>190412.155</td>
</tr>
</tbody>
</table>

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:

- **SUM(amount)** computes the sum of an amount. The outer **AVG()** computes the average.
- **ORDER BY month** orders the rows read by the query by month.
- **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:

```sql
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;
```

<table>
<thead>
<tr>
<th>MONTH</th>
<th>MONTH_AMOUNT</th>
<th>MOVING_AVERAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>95525.55</td>
<td>106098.575</td>
</tr>
<tr>
<td>2</td>
<td>116671.6</td>
<td>124168.357</td>
</tr>
<tr>
<td>3</td>
<td>160307.92</td>
<td>150992.773</td>
</tr>
<tr>
<td>4</td>
<td>175998.8</td>
<td>163552.053</td>
</tr>
<tr>
<td>5</td>
<td>154349.44</td>
<td>151766.533</td>
</tr>
<tr>
<td>6</td>
<td>124951.36</td>
<td>149865.653</td>
</tr>
<tr>
<td>7</td>
<td>170296.16</td>
<td>169327.733</td>
</tr>
<tr>
<td>8</td>
<td>212735.68</td>
<td>194213.84</td>
</tr>
<tr>
<td>9</td>
<td>199609.68</td>
<td>225608.717</td>
</tr>
<tr>
<td>10</td>
<td>264480.79</td>
<td>208104.15</td>
</tr>
<tr>
<td>11</td>
<td>160221.98</td>
<td>187346.313</td>
</tr>
<tr>
<td>12</td>
<td>137336.17</td>
<td>148779.075</td>
</tr>
</tbody>
</table>

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

```sql
AVG(SUM(amount)) OVER
  (ORDER BY month ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
AS moving_average
```

Let's break down this expression:

- `SUM(amount)` computes the sum of an amount. The outer `AVG()` computes the average.
- `ORDER BY month` orders the rows read by the query by month.
- `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;
```

<table>
<thead>
<tr>
<th>MONTH</th>
<th>MONTH_AMOUNT</th>
<th>PREVIOUS_MONTH_AMOUNT</th>
<th>NEXT_MONTH_AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>95525.55</td>
<td>95525.55</td>
<td>116671.6</td>
</tr>
<tr>
<td>2</td>
<td>116671.6</td>
<td>95525.55</td>
<td>160307.92</td>
</tr>
<tr>
<td>3</td>
<td>160307.92</td>
<td>116671.6</td>
<td>175998.8</td>
</tr>
<tr>
<td>4</td>
<td>175998.8</td>
<td>160307.92</td>
<td>154349.44</td>
</tr>
<tr>
<td>5</td>
<td>154349.44</td>
<td>175998.8</td>
<td>124951.36</td>
</tr>
<tr>
<td>6</td>
<td>124951.36</td>
<td>154349.44</td>
<td>170296.16</td>
</tr>
<tr>
<td>7</td>
<td>170296.16</td>
<td>124951.36</td>
<td>212735.68</td>
</tr>
<tr>
<td>8</td>
<td>212735.68</td>
<td>170296.16</td>
<td>199609.68</td>
</tr>
<tr>
<td>9</td>
<td>199609.68</td>
<td>212735.68</td>
<td>264480.79</td>
</tr>
<tr>
<td>10</td>
<td>264480.79</td>
<td>199609.68</td>
<td>160221.98</td>
</tr>
<tr>
<td>11</td>
<td>160221.98</td>
<td>264480.79</td>
<td>137336.17</td>
</tr>
<tr>
<td>12</td>
<td>137336.17</td>
<td>160221.98</td>
<td>137336.17</td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>MONTH</th>
<th>MONTH_AMOUNT</th>
<th>CURR_DIV_PREV</th>
<th>CURR_DIV_NEXT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>95525.55</td>
<td>1</td>
<td>.818755807</td>
</tr>
<tr>
<td>2</td>
<td>116671.6</td>
<td>1.22136538</td>
<td>.727796855</td>
</tr>
<tr>
<td>3</td>
<td>160307.92</td>
<td>1.37400978</td>
<td>.910846665</td>
</tr>
<tr>
<td>4</td>
<td>175998.8</td>
<td>1.09787963</td>
<td>1.14026199</td>
</tr>
<tr>
<td>5</td>
<td>154349.44</td>
<td>.876991434</td>
<td>1.23527619</td>
</tr>
<tr>
<td>6</td>
<td>124951.36</td>
<td>.809535558</td>
<td>.737329756</td>
</tr>
<tr>
<td>7</td>
<td>170296.16</td>
<td>1.36289961</td>
<td>.80055867</td>
</tr>
<tr>
<td>8</td>
<td>212735.68</td>
<td>1.24921008</td>
<td>1.06575833</td>
</tr>
<tr>
<td>9</td>
<td>199609.68</td>
<td>.93829902</td>
<td>.754722791</td>
</tr>
<tr>
<td>10</td>
<td>264480.79</td>
<td>1.3249898</td>
<td>1.65071478</td>
</tr>
<tr>
<td>11</td>
<td>160221.98</td>
<td>.605798175</td>
<td>1.16664081</td>
</tr>
<tr>
<td>12</td>
<td>137336.17</td>
<td>.857161858</td>
<td>1</td>
</tr>
</tbody>
</table>

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

- The total sum of all sales for all three months (labeled as total_month_amount).
- The total sum of all sales for all product types (labeled as total_product_type_amount).

```sql
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;
```

<table>
<thead>
<tr>
<th>MONTH</th>
<th>PRD_TYPE_ID</th>
<th>TOTAL_MONTH_AMOUNT</th>
<th>TOTAL_PRODUCT_TYPE_AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>95525.55</td>
<td>201303.92</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>95525.55</td>
<td>44503.92</td>
</tr>
</tbody>
</table>
Notice that the query uses the following expression to report the total sum of all sales for all months (labeled as \( \text{total\_month\_amount} \)):

\[
\text{SUM(SUM(amount)) OVER (PARTITION BY month)} \\
\text{AS total\_month\_amount}
\]

Let’s break down this expression:
- \( \text{SUM(amount)} \) computes the sum of an amount. The outer \( \text{SUM()} \) computes the total sum.
- \( \text{OVER (PARTITION BY month)} \) causes the outer \( \text{SUM()} \) 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 \( \text{total\_product\_type\_amount} \)):

\[
\text{SUM(SUM(amount)) OVER (PARTITION BY prd\_type\_id)} \\
\text{AS total\_product\_type\_amount}
\]

Let’s break down this expression:
- \( \text{SUM(amount)} \) computes the sum of an amount. The outer \( \text{SUM()} \) computes the total sum.
- \( \text{OVER (PARTITION BY prd\_type\_id)} \) causes the outer \( \text{SUM()} \) to compute the sum for each product type.

**Using the RATIO_TO_REPORT() Function**

You use the \( \text{RATIO\_TO\_REPORT()} \) 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:

- The sum of the sales amount by product type for each month (labeled as \( \text{prd\_type\_amount} \)).
- The ratio of the product type’s sales amount to the entire month’s sales (labeled as \( \text{prd\_type\_ratio} \)), which is computed using \( \text{RATIO\_TO\_REPORT()} \).

\[
\text{SELECT} \\
\quad \text{month, prd\_type\_id,} \\
\quad \text{SUM(amount) AS prd\_type\_amount,} \\
\quad \text{RATIO\_TO\_REPORT(SUM(amount)) OVER (PARTITION BY month) AS prd\_type\_ratio}
\]
FROM all_sales
WHERE year = 2003
AND month <= 3
GROUP BY month, prd_type_id
ORDER BY month, prd_type_id;

<table>
<thead>
<tr>
<th>MONTH</th>
<th>PRD_TYPE_ID</th>
<th>PRD_TYPE_AMOUNT</th>
<th>PRD_TYPE_RATIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>38909.04</td>
<td>.40731553</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>14309.04</td>
<td>.149792804</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>24909.04</td>
<td>.260757881</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>17398.43</td>
<td>.182133785</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>70567.9</td>
<td>.604842138</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>13367.9</td>
<td>.114577155</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>15467.9</td>
<td>.132576394</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>17267.9</td>
<td>.148004313</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>91826.98</td>
<td>.57281624</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>16826.98</td>
<td>.104966617</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>20626.98</td>
<td>.128670998</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>31026.98</td>
<td>.193546145</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notice that the query uses the following expression to compute the ratio (labeled as prd_type_ratio):

\[
RATIO\_TO\_REPORT(SUM(amount))\ \text{OVER\ (PARTITION\ BY\ month)}\ \text{AS\ prd\_type\_ratio}
\]

Let's break down this expression:

- \( \text{SUM(amount)} \) computes the sum of the sales amount.
- \( \text{OVER (PARTITION BY month)} \) causes the outer \( \text{SUM()} \) to compute the sum of the sales amount for each month.
- 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:

\[
\text{SELECT month, SUM(amount) AS month_amount, LAG(SUM(amount), 1) OVER (ORDER BY month) AS previous_month_amount, LEAD(SUM(amount), 1) OVER (ORDER BY month) AS next_month_amount FROM all_sales WHERE year = 2003}
\]
Chapter 7: Advanced Queries

GROUP BY month
ORDER BY month;

<table>
<thead>
<tr>
<th>MONTH</th>
<th>MONTH_AMOUNT</th>
<th>PREVIOUS_MONTH_AMOUNT</th>
<th>NEXT_MONTH_AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>95525.55</td>
<td>116671.6</td>
<td>116671.6</td>
</tr>
<tr>
<td>2</td>
<td>116671.6</td>
<td>95525.55</td>
<td>160307.92</td>
</tr>
<tr>
<td>3</td>
<td>160307.92</td>
<td>116671.6</td>
<td>175998.8</td>
</tr>
<tr>
<td>4</td>
<td>175998.8</td>
<td>160307.92</td>
<td>154349.44</td>
</tr>
<tr>
<td>5</td>
<td>154349.44</td>
<td>175998.8</td>
<td>124951.36</td>
</tr>
<tr>
<td>6</td>
<td>124951.36</td>
<td>154349.44</td>
<td>170296.16</td>
</tr>
<tr>
<td>7</td>
<td>170296.16</td>
<td>124951.36</td>
<td>199609.68</td>
</tr>
<tr>
<td>8</td>
<td>212735.68</td>
<td>170296.16</td>
<td>264480.79</td>
</tr>
<tr>
<td>9</td>
<td>199609.68</td>
<td>212735.68</td>
<td>160221.98</td>
</tr>
<tr>
<td>10</td>
<td>264480.79</td>
<td>199609.68</td>
<td>137336.17</td>
</tr>
<tr>
<td>11</td>
<td>160221.98</td>
<td>264480.79</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>137336.17</td>
<td>160221.98</td>
<td></td>
</tr>
</tbody>
</table>

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

\[
\text{LAG(SUM(amount), 1) OVER (ORDER BY month) AS previous\_month\_amount,}
\text{LEAD(SUM(amount), 1) OVER (ORDER BY month) AS next\_month\_amount}
\]

\[
\text{LAG(SUM(amount), 1) gets the previous row's sum of the amount. LEAD(SUM(amount), 1) gets the next row's sum of the amount.}
\]

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 FIRST and 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:

\[
\text{SELECT MIN(month) KEEP (DENSE_RANK FIRST ORDER BY SUM(amount)) AS highest\_sales\_month,}
\text{MIN(month) KEEP (DENSE_RANK LAST ORDER BY SUM(amount)) AS lowest\_sales\_month}
\]

FROM all_sales
WHERE year = 2003
GROUP BY month
ORDER BY month;

<table>
<thead>
<tr>
<th>HIGHEST_SALES_MONTH</th>
<th>LOWEST_SALES_MONTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

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 \( x \).

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REGR_AVGX(( y, x ))</td>
<td>Returns the average of ( x ) after eliminating ( x ) and ( y ) pairs where either ( x ) or ( y ) is null</td>
</tr>
<tr>
<td>REGR_AVGY(( y, x ))</td>
<td>Returns the average of ( y ) after eliminating ( x ) and ( y ) pairs where either ( x ) or ( y ) is null</td>
</tr>
<tr>
<td>REGR_COUNT(( y, x ))</td>
<td>Returns the number of non-null number pairs that are used to fit the regression line</td>
</tr>
<tr>
<td>REGR_INTERCEPT(( y, x ))</td>
<td>Returns the intercept on the ( y )-axis of the regression line</td>
</tr>
<tr>
<td>REGR_R2(( y, x ))</td>
<td>Returns the coefficient of determination (R-squared) of the regression line</td>
</tr>
<tr>
<td>REGR_SLOPE(( y, x ))</td>
<td>Returns the slope of the regression line</td>
</tr>
<tr>
<td>REGR_SXX(( y, x ))</td>
<td>Returns ( \text{REG_COUNT}(y, x) \times \text{VAR_POP}(x) )</td>
</tr>
<tr>
<td>REGR_SXY(( y, x ))</td>
<td>Returns ( \text{REG_COUNT}(y, x) \times \text{COVAR_POP}(y, x) )</td>
</tr>
<tr>
<td>REGR_SYY(( y, x ))</td>
<td>Returns ( \text{REG_COUNT}(y, x) \times \text{VAR_POP}(y) )</td>
</tr>
</tbody>
</table>

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

```sql
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;
```

<table>
<thead>
<tr>
<th>PRD_TYPE_ID</th>
<th>AVGX</th>
<th>AVGY</th>
<th>COUNT</th>
<th>INTER</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SLOPE</td>
<td>SXX</td>
<td>SXY</td>
<td>SYY</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6.5</td>
<td>12570.5811</td>
<td>72</td>
<td>13318.4543</td>
<td>.003746289</td>
</tr>
<tr>
<td>-115.05741</td>
<td>858</td>
<td>-98719.26</td>
<td>3031902717</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6.5</td>
<td>2588.62806</td>
<td>72</td>
<td>2608.11268</td>
<td>.0000508</td>
</tr>
<tr>
<td>-2.997634</td>
<td>858</td>
<td>-2571.97</td>
<td>151767392</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>6.5</td>
<td>6642.65153</td>
<td>72</td>
<td>2154.23119</td>
<td>.126338815</td>
</tr>
</tbody>
</table>
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()**
- **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:

```sql
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;
```

<table>
<thead>
<tr>
<th>PRD_TYPE_ID</th>
<th>SUM(AMOUNT)</th>
<th>RANK</th>
<th>PERCENT_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>905081.84</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>186381.22</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>478270.91</td>
<td>2</td>
<td>.333333333</td>
</tr>
<tr>
<td>4</td>
<td>402751.16</td>
<td>3</td>
<td>.666666667</td>
</tr>
</tbody>
</table>

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

```sql
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;
```

<table>
<thead>
<tr>
<th>RANK</th>
<th>PERCENT_RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>.25</td>
</tr>
</tbody>
</table>

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 10g 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)
ORDER BY prd_type_id, year, month;
```

Let's break down this query:

- **PARTITION BY (prd_type_id)** specifies that the results are partitioned by prd_type_id.
- **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.
- **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.
- After MEASURES come three lines that compute the future sales for January, February, and March of 2004:

■ 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):

<table>
<thead>
<tr>
<th>PRD_TYPE_ID</th>
<th>YEAR</th>
<th>MONTH</th>
<th>SALES_AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2003</td>
<td>1</td>
<td>10034.84</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>2</td>
<td>15144.65</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>3</td>
<td>20137.83</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>4</td>
<td>25057.45</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>5</td>
<td>17214.56</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>6</td>
<td>15564.64</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>7</td>
<td>12654.84</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>8</td>
<td>17434.82</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>9</td>
<td>19854.57</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>10</td>
<td>21754.19</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>11</td>
<td>13029.73</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>12</td>
<td>10034.84</td>
</tr>
<tr>
<td>1</td>
<td>2004</td>
<td>1</td>
<td>10034.84</td>
</tr>
<tr>
<td>1</td>
<td>2004</td>
<td>2</td>
<td>35282.48</td>
</tr>
<tr>
<td>1</td>
<td>2004</td>
<td>3</td>
<td>25172.29</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>1</td>
<td>1034.84</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>2</td>
<td>1544.65</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>3</td>
<td>2037.83</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>4</td>
<td>2557.45</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>5</td>
<td>1714.56</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>6</td>
<td>1564.64</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>7</td>
<td>1264.84</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>8</td>
<td>1734.82</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>9</td>
<td>1854.57</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>10</td>
<td>2754.19</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>11</td>
<td>1329.73</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>12</td>
<td>1034.84</td>
</tr>
<tr>
<td>2</td>
<td>2004</td>
<td>1</td>
<td>1034.84</td>
</tr>
<tr>
<td>2</td>
<td>2004</td>
<td>2</td>
<td>3582.48</td>
</tr>
<tr>
<td>2</td>
<td>2004</td>
<td>3</td>
<td>2547.29</td>
</tr>
</tbody>
</table>

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:

```sql
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],
}
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:

```sql
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:

```sql
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:

```sql
sales_amount[1, 2004] = 
ROUND(sales_amount[CURRENTV(), 2003] * 1.25, 2)
```

The following query shows the use of this expression:

```sql
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):

<table>
<thead>
<tr>
<th>PRD_TYPE_ID</th>
<th>YEAR</th>
<th>MONTH</th>
<th>SALES_AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2003</td>
<td>1</td>
<td>10034.84</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>2</td>
<td>15144.65</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>3</td>
<td>20137.83</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>4</td>
<td>25057.45</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>5</td>
<td>17214.56</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>6</td>
<td>15564.64</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>7</td>
<td>12654.84</td>
</tr>
</tbody>
</table>
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:

\[
\text{sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004]} = \\
\text{ROUND(sales_amount[CURRENTV(), 2003] \times 1.25, 2)}
\]

The following query shows the use of this expression:

\[
\text{SELECT prd_type_id, year, month, sales_amount} \\
\text{FROM all_sales} \\
\text{WHERE prd_type_id BETWEEN 1 AND 2} \\
\text{AND emp_id = 21} \\
\text{MODEL} \\
\text{PARTITION BY (prd_type_id)} \\
\text{DIMENSION BY (month, year)} \\
\text{MEASURES (amount sales_amount)} \\
\{ \\
\text{sales_amount[FOR month FROM 1 TO 3 INCREMENT 1, 2004]} = \\
\text{ROUND(sales_amount[CURRENTV(), 2003] \times 1.25, 2)}
\}
\]

ORDER BY prd_type_id, year, month;

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

\[
\begin{array}{cccc}
\text{PRD_TYPE_ID} & \text{YEAR} & \text{MONTH} & \text{SALES_AMOUNT} \\
\hline
1 & 2003 & 1 & 10034.84 \\
1 & 2003 & 2 & 15144.65 \\
1 & 2003 & 3 & 20137.83 \\
1 & 2003 & 4 & 25057.45 \\
\end{array}
\]
Handling Null and Missing Values

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

\textbf{Using IS PRESENT}

\begin{verbatim}
IS PRESENT returns true if the row specified by the cell reference existed prior to the execution
of the \texttt{MODEL} clause. For example:

\begin{verbatim}
sales_amount[\texttt{CURRENTV()}, 2003] \texttt{IS PRESENT}
\end{verbatim}

will return true if \texttt{sales_amount[\texttt{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:

\begin{verbatim}
sales_amount[\texttt{FOR month FROM 1 TO 3 INCREMENT 1, 2004}] =
\begin{verbatim}
CASE WHEN sales_amount[\texttt{CURRENTV()}, 2003] \texttt{IS PRESENT} THEN
ROUND(sales_amount[\texttt{CURRENTV()}, 2003] * 1.25, 2)
ELSE
0
END
\end{verbatim}
\end{verbatim}

The following query shows the use of this expression:

\begin{verbatim}
SELECT prd_type_id, year, month, sales_amount
FROM all_sales
WHERE prd_type_id BETWEEN 1 AND 2
\end{verbatim}
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 \texttt{sales\_amount[CURRENTV(), 2003]} exists and is not null; otherwise 0 will be returned.

**Using \texttt{IGNORE NAV} and \texttt{KEEP NAV}**

\texttt{IGNORE NAV} returns

- 0 for null or missing numeric values.
- An empty string for null or missing string values.
- 01-JAN-2000 for null or missing date values.
- Null for all other database types.

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

The following query shows the use of \texttt{IGNORE NAV}:

```sql
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 \texttt{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 \texttt{RULES UPDATE}:

```sql
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:

<table>
<thead>
<tr>
<th>PRD_TYPE_ID</th>
<th>YEAR</th>
<th>MONTH</th>
<th>SALES_AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2003</td>
<td>1</td>
<td>10034.84</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>2</td>
<td>15144.65</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>3</td>
<td>20137.83</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>4</td>
<td>25057.45</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>5</td>
<td>17214.56</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>6</td>
<td>15564.64</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>7</td>
<td>12654.84</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>8</td>
<td>17434.82</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>9</td>
<td>19854.57</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>10</td>
<td>21754.19</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>11</td>
<td>13029.73</td>
</tr>
<tr>
<td>1</td>
<td>2003</td>
<td>12</td>
<td>10034.84</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>1</td>
<td>1034.84</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>2</td>
<td>1544.65</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>3</td>
<td>2037.83</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>4</td>
<td>2557.45</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>5</td>
<td>1714.56</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>6</td>
<td>1564.64</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>7</td>
<td>1264.84</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>8</td>
<td>1734.82</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>9</td>
<td>1854.57</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>10</td>
<td>2754.19</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>11</td>
<td>1329.73</td>
</tr>
<tr>
<td>2</td>
<td>2003</td>
<td>12</td>
<td>1034.84</td>
</tr>
</tbody>
</table>

Using the PIVOT and UNPIVOT Clauses

The PIVOT clause is new for Oracle Database 11g 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 11g 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;

<table>
<thead>
<tr>
<th>PRD_TYPE_ID</th>
<th>JAN</th>
<th>FEB</th>
<th>MAR</th>
<th>APR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38909.04</td>
<td>70567.9</td>
<td>91826.98</td>
<td>120344.7</td>
</tr>
<tr>
<td>2</td>
<td>14309.04</td>
<td>13367.9</td>
<td>16826.98</td>
<td>15664.7</td>
</tr>
<tr>
<td>3</td>
<td>24909.04</td>
<td>15467.9</td>
<td>20626.98</td>
<td>23844.7</td>
</tr>
</tbody>
</table>

Starting with the first line of output, you can see there was
- $38,909.04 of product type #1 sold in January.
- $70,567.90 of product type #1 sold in February.
- ...and so on for the rest of the first line.

The second line of output shows there was
- $14,309.04 of product type #2 sold in January.
- $13,367.90 of product type #2 sold in February.
- ...and so on for the rest of the output.

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:

```sql
SELECT * 
FROM ( 
    inner_query 
) 
PIVOT ( 
    aggregate_function FOR pivot_column IN (list_of_values) 
) 
ORDER BY ...;
```
Oracle Database 11g SQL

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

- 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.
- \( \text{SUM(amount)} \) for month IN (1 AS JAN, 2 AS FEB, 3 AS MAR, 4 AS APR) is the line in the PIVOT clause.
  - The \( \text{SUM()} \) function adds up the sales amounts for the product types in the first four months (the months are listed in the \text{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.
  - 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.
  - At the very end of the example, the \text{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 \text{FOR} part of the PIVOT. The following example pivots on both the month and prd_type_id columns, which are referenced in the \text{FOR} part; notice that the list of values in the \text{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  
   )  
);
```

<table>
<thead>
<tr>
<th>JAN_PRDTYPE2</th>
<th>FEB_PRDTYPE3</th>
<th>MAR_PRDTYPE1</th>
<th>APR_PRDTYPE2</th>
</tr>
</thead>
<tbody>
<tr>
<td>14309.04</td>
<td>15467.9</td>
<td>91826.98</td>
<td>15664.7</td>
</tr>
</tbody>
</table>

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 \text{IN} part). As you can see from the query output, there were the following sales amounts:

- $14,309.04 of product type #2 in January
$15,467.90 of product type #3 in February
$91,826.98 of product type #1 in March
$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:

```sql
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
));
```

<table>
<thead>
<tr>
<th>JAN_PRDTYPE1</th>
<th>FEB_PRDTYPE2</th>
<th>MAR_PRDTYPE3</th>
<th>APR_PRDTYPE1</th>
</tr>
</thead>
<tbody>
<tr>
<td>38909.04</td>
<td>13367.9</td>
<td>20626.98</td>
<td>120344.7</td>
</tr>
</tbody>
</table>

As you can see from this output, there were the following sales amounts:

- $38,909.04 of product type #1 in January
- $13,367.90 of product type #2 in February
- $20,626.98 of product type #3 in March
- $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:

```sql
SELECT *
FROM (
    SELECT month, prd_type_id, amount
    FROM all_sales
    WHERE year = 2003
    AND prd_type_id IN (1, 2, 3)
);
Oracle Database 11g SQL

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;

PRD_TYPE_ID JAN_SUM_AMOUNT JAN_AVG_AMOUNT FEB_SUM_AMOUNT FEB_AVG_AMOUNT
----------- -------------- -------------- -------------- --------------
1       38909.04        6484.84        70567.9     11761.3167
2       14309.04        2384.84        13367.9     2227.98333
3       24909.04     4151.50667        15467.9     2577.98333

As you can see, the first line of output shows for product type #1:

- A total of $38,909.04 and an average of $6,484.84 sold in January
- 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:

- A total of $14,309.04 and an average of $2,384.84 sold in January
- 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:

```sql
SELECT *
FROM pivot_sales_data
UNPIVOT (
    amount FOR month IN (JAN, FEB, MAR, APR)
)
ORDER BY prd_type_id;
```

<table>
<thead>
<tr>
<th>PRD_TYPE_ID</th>
<th>JAN</th>
<th>FEB</th>
<th>MAR</th>
<th>APR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38909.04</td>
<td>70567.9</td>
<td>91826.98</td>
<td>120344.7</td>
</tr>
<tr>
<td>2</td>
<td>14309.04</td>
<td>13367.9</td>
<td>16826.98</td>
<td>15664.7</td>
</tr>
<tr>
<td>3</td>
<td>24909.04</td>
<td>15467.9</td>
<td>20626.98</td>
<td>23844.7</td>
</tr>
</tbody>
</table>

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.

**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:

- The set operators (UNION ALL, UNION, INTERSECT, and MINUS) allow you to combine rows returned by two or more queries.
- `TRANSLATE(x, from_string, to_string)` translates characters in one string to characters in another string.
- `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.
Oracle Database 11g SQL

- **CASE** is similar to **DECODE()**. You should use **CASE** because it is ANSI-compliant.
- Queries may be run against data that is organized into a hierarchy.
- **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.
- **CUBE** extends the **GROUP BY** clause to return rows containing a subtotal for all combinations of columns, plus a row containing the grand total.
- 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 **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.
- The **Oracle Database 11g PIVOT** and **UNPIVOT** clauses are useful for seeing overall trends in large amounts of data.

In the next chapter, you’ll learn about changing the contents of a table.