9

Analytical Processing with SQL

Oracle has enhanced its ability to develop business intelligence. It has placed a variety of analytical functions in its database. These functions became available with the release of Oracle 8.1.6. The functions allow the business analyst to perform rankings, moving window calculations (i.e., moving averages), and lag/lead analysis. Oracle has also included families of statistic and regression functions in the database. These functions are part of the core SQL processing. Previous to this release, these functions could be obtained only through the use of purchased software. Incorporating these capabilities allows Oracle to enhance scalability, database performance, and simplicity.

Analytic functions are well suited for the data warehouses that are common in the workplace. The purpose of these warehouses is to derive business information. Data warehouses look at data derived from OLTP systems in different ways. One way is to look at the data across different dimensions. For example, a user of our employee data may want to view the data in the following ways:

- Total costs by current position
- Total costs by department
- Total costs by state
- Number of employees by state
- Number of employees by department
- Number of employees by current position

As you can see, there is virtually an unlimited number of ways to look at or analyze the same basic data. Each of these different ways of looking at the data is a dimension. Data warehouses generally have the ability to provide information across different dimensions, which makes them multidimensional databases. You will see that analytic functions like ROLLUP and CUBE are ideally suited for computing values across multidimensions.

ROLLUP

ROLLUP is an extension or option available for the GROUP BY clause. It allows the user to compute subtotals and totals for groups of data. It is highly efficient and adds little overhead to the SELECT statement. A syntax template of the option is as follows:

Group By rollup *(expression1, expression2)*

The ROLLUP keyword will cause a cumulative subtotal row to be created for each of the expressions. When multiple expressions are placed in the parameter list, Oracle will create a grand total for the right-most expression and a subtotal for each of the left-most expressions.

The ROLLUP rows can be identified by null values. The grand-total row will have only the grand-total values. All other row values will be null. If a subtotal is computed, all expressions to the left of the ROLLUP expression will contain nulls. Listing 9.1 illustrates the ROLLUP option. In this query, Wages values are summed for each gender and for each department. The ROLLUP option is used to create a subtotal of female and male wages and a grand total of all wages.

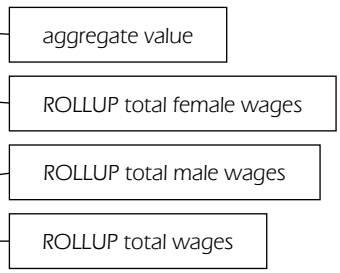
Listing 9.1 *Using the ROLLUP Option to Compute Subtotals and Totals*

```
SQL> select gender, department, sum(wages)
  2  from department, employee
  3  where department = fk_department
  4  group by rollup(gender, department)
  5  order by 1,2;
```

```
G DEPA SUM(WAGES)
- ---- ----------
F POL        9800
F WEL        7000
F           16800
M INT       65000
M POL       77900
M WEL       45000
M          187900
           204700

8 rows selected.

End Listing
```



Partial ROLLUPs can also be computed. If you place GROUP BY expressions outside the ROLLUP option, Oracle will aggregate values based on these expressions. The ROLLUP or subtotals will be based on the expressions within the ROLLUP parameter list. A value will be computed for each unique occurrence of a ROLLUP expression value within the aggregate value. Listing 9.2 depicts this feature.

The example query used in Listing 9.1 was changed. The Department column was moved outside the ROLLUP function. This caused Oracle to compute an aggregate (SUM) value for each department. The gender was left inside the ROLLUP function; therefore, Oracle also computed a rolled-up value for each type of gender for each department. The ROLLUP option no longer computes the total wages for each gender.

Listing 9.2 *Computing a Partial ROLLUP of Wages Per Gender within a Department*

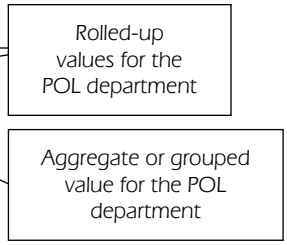
```
SQL> select gender, department, sum(wages)
  2  from department, employee
  3  where department = fk_department
  4  group by rollup(gender), department
  5  order by 2;

G DEPA SUM(WAGES)
- ---- ----------
M INT       65000
```

```
  INT       65000
F POL        9800
M POL       77900
  POL       87700
F WEL        7000
M WEL       45000
  WEL       52000

8 rows selected.

End listing
```



Prior to the development of the ROLLUP function, the only way to compute subtotal values was to use several SELECT statements by using the UNION ALL keyword. Replacing these combined SELECT statements with ROLLUPs substantially reduced the amount of work Oracle had to perform. ROLLUPs are very helpful when the data need to be subtotaled along a hierarchical dimension such as geography or time. It also is useful for the maintenance of summary tables in the data warehouse.

CUBE

The ROLLUPs that we have seen are a small percentage of the numerous possible combinations. For example, the ROLLUP option in Listing 9.1 computed subtotals for gender. Suppose the business analyst wanted to get subtotals for Department while still rolling up the costs of gender per department. This would require the GROUP BY clause to change to the following:

Group by rollup (department, gender)

Suppose the business analyst also wanted to subtotal costs for gender and department. This cannot be done with the ROLLUP option because the SELECT clause can contain only one GROUP BY clause (and only one ROLLUP option). The CUBE option must be used instead.

The CUBE option will compute subtotals for all expressions placed within the parameter list. Listing 9.3 depicts this. The SELECT statement from Listing 9.1

is modified. The ROLLUP option is replaced by the CUBE option, and the SELECT statement will now return subtotals for departments and for gender.

Listing 9.3 *Using the CUBE Option*

```
SQL> select gender, department, sum(wages)
  2  from department, employee
  3  where department = fk_department
  4  group by cube(gender, department)
  5  order by 1,2;

G DEPA SUM(WAGES)
- ---- ----------
F POL        9800
F WEL        7000
F           16800
M INT       65000
M POL       77900
M WEL       45000
M          187900
  INT       65000
  POL       87700
  WEL       52000
           204700

11 rows selected.

End Listing
```

This option should be used with care. It will cause all possible combinations of subtotals to occur, which will place a significant load on the computing environment. The CUBE option can be used in situations that require cross-tabular reports.

In the previous listings, the SUM group function was used. The ROLLUP and CUBE options can be used with the other group functions (i.e., COUNT, AVG, MIN, MAX, STDDEV, and VARIANCE). Finally, just as with the ROLLUP function, partial CUBE operations can be computed. Place outside the parameter list any expressions that you do not want subtotals on.

The GROUPING Function

A problem with the ROLLUP and CUBE options is the difficulty in identifying the rows that are subtotals. One way to identify subtotal rows is to identify the rows that contain null values. Expressions that are subtotaled will have a value in the column that determines the ROLLUP. The other expressions will contain null values. However, if the database contains nulls, this technique will not work because Oracle will also roll up null value records. Oracle developed the GROUPING function to help the identification of ROLLUP records.

GROUPING functions accept only one parameter. The GROUPING function returns a value of 1 if the row is a subtotal row for the target expression. The function will return a value of 0 if the row is not a subtotal row for the target expression. A syntax template follows:

Grouping *(expression)*

Listing 9.4 illustrates the GROUPING function, which was used to identify the rows that are a subtotal (or grand total) of the gender or department.

Listing 9.4 *Using the GROUPING Function*

```
SQL> select gender, department, sum(wages),
  2         grouping(gender) as gdr, grouping(department) as dpt
  3  from department, employee
  4  where department = fk_department
  5  group by rollup(gender, department)
  6  order by 1,2;

G DEPA SUM(WAGES)       GDR       DPT
- ---- ---------- --------- ---------
F POL        9800         0         0
F WEL        7000         0         0
F           16800         0         1
M INT       65000         0         0
M POL       77900         0         0
M WEL       45000         0         0
M          187900         0         1
           204700         1         1

8 rows selected.

End Listing
```

One excellent use of the GROUPING function is to create descriptions for the aggregated rows. It is possible to populate one of the SELECT statement columns with a text literal when the row has a rolled-up value. The DECODE and CASE functions can be used. The CASE function is discussed later in this chapter. Listing 9.5 illustrates this functionality. The GROUPING and DECODE functions are used to populate the subtotal and grand-total rows. The DECODE function will change the column value to the text literal if the GROUPING function returns a value of 1. Otherwise, the original value will be displayed.

Listing 9.5 *Using the DECODE and GROUPING Functions to Create Descriptive Prompts*

```
SQL> column gdr noprint
SQL> break on gender
SQL> select decode(grouping(gender),1, 'Total Department Wages',
  2         gender) gender,
  3         decode(grouping(department),1,
  4                decode(grouping(gender),0,'Total Per Gender', ' '),
  5                department)department,
  6         sum(wages),
  7         grouping(gender) as gdr
  8  from department, employee
  9  where department = fk_department
 10  group by rollup(gender, department)
 11  order by 1,4;

GENDER                 DEPARTMENT       SUM(WAGES)
---------------------- ---------------- ----------
F                      POL                    9800
                       WEL                    7000
                       Total Per Gender      16800
M                      INT                   65000
                       POL                   77900
                       WEL                   45000
                       Total Per Gender     187900
Total Department Wages                      204700

8 rows selected.

SQL>

End listing
```

The GROUPING function has two other uses: It can be used as a filtering argument in the HAVING clause, and it can be used as sort criteria in the ORDER BY clause.

Ranking Functions

Oracle has provided several functions for ranking rows. These functions will calculate rankings, percentiles, and n-tiles of the values. These functions are performed after the SELECT statement assembles the result set and prior to the ordering of the results.

There is an Oracle feature, PARTITIONS, that is used with analytic functions: This feature is totally unrelated to the physical partitioning of data stored within tables, which was discussed in Chapter 2. This partitioning occurs to only the result set of the SELECT statement and is not permanent. The feature allows the user to place into subsets the records that a SELECT statement returns. For example, assume that you would like to determine the top-two wage earners in each department. The PARTITION option can be used to assemble or collate the records by department. The RANK function can then be used to provide the proper rank.

The RANK Function

The RANK function allows the business analyst to compute the rank of a set of values, which is done by comparing all of the values in a set. The following is a syntax template of the function:

```
Rank( ) over (
        [ partition by expression, expression ]
        order by expression
        [ collate clause ] [asc|desc]
        [ nulls first | nulls last ]   )
```

The following describe these clauses:

- The PARTITION clause is an optional clause that will cause the results to be segregated into subsets. The rank value will be reset each time the partition group changes. If this clause is omitted, the entire group will form the set.

- The ORDER BY clause is mandatory. In order to rank values, the values must be sorted. The ORDER BY clause identifies the expression that will be evaluated for the ranking. This clause is executed after the data is partitioned. Ranks start with a value of 1. The default sort order is ascending. This can be changed to descending with the DESC keyword.
- The NULLS FIRST | NULLS LAST clause determines the position of null values in the ranking. The setting can determine whether the null values appear at the top or the bottom of the ranking.
- If the NULLS FIRST | NULLS LAST clause is omitted, null values will be considered larger than any other value in the list.

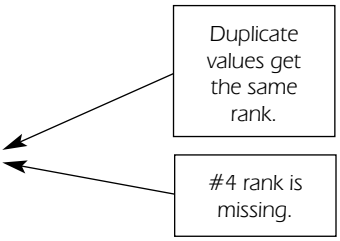
Listing 9.6 illustrates the RANK function. It is used to rank the total department wages. The department with the highest wages is ranked 1. The NVL function is used to change null values to 90000. This was done for illustrative purposes only. By making certain that two rows have the same value, I can call attention to the facts that equivalent values will have the same rank and sequential ranks are omitted for all duplicate ranks.

Listing 9.6 *Using the RANK Function*

```
SQL> select department, sum(nvl(wages,90000)),
  2         rank() over (order by sum(nvl(wages,90000)) desc)
  3         as rank_all
  4  from department, employee
  5  where department = fk_department(+)
  6  group by department;

DEPA SUM(NVL(WAGES,90000))  RANK_ALL
---- -------------------- ---------
INT                155000         1
WEL                142000         2
CEN                 90000         3
TRF                 90000         3
POL                 87700         5

End listing
```



Null values can cause havoc in ranking and ordering. A null value is essentially an unknown value. Normally nulls appear at the end of a value list that is

in ascending order. They will appear at the top of a value list that is in descending order. This may not be the ranking order you desire. You may want null values ranked at the top. THE NULLS FIRST | NULLS LAST option can be used to change how the null values are ranked. Table 9.1 describes how the null values will be ranked.

Listing 9.7 illustrates the various options available using the NULLS option. The first use of the option is to rank the nulls as the lowest value in an ascending sort order. The second use is to rank the nulls as the highest value in an ascending sort order. The third use causes the nulls to have the highest values in a descending sort order. The final use causes the nulls to have the lowest value in a descending sort order.

Listing 9.7 *Using the NULLS FIRST | NULLS LAST Option*

```
SQL> select last_name, first_name, wages,
  2    rank() over (order by wages asc nulls first) as ranking_1,
  3    rank() over (order by wages asc nulls last) as ranking_1,
  4    rank() over (order by wages desc nulls first) as ranking_1,
  5    rank() over (order by wages desc nulls last) as ranking_1
  6  from employee;

LAST_NAME       FIRST_NAMEWAGES RANKING_1 RANKING_1 RANKING_1 RANKING_1
--------------- --------- --------- --------- --------- --------- -----
CLINTON         WILLIAM   15000         21        19         3         1
BUSH            GEORGE    14000         20        18         4         2
REAGAN          RONALD    13500         19        17         5         3
CARTER          JIMMY     13000         17        15         6         4
FORD            GERALD    13000         17        15         6         4
NIXON           RICHARD   12500         16        14         8         6
JOHNSON         LYNDON    12000         15        13         9         7
...
TAFT            WILLIAM    8500          6         4        18        16
ROOSEVELT       THEODORE   8000          5         3        19        17
JOHNSON         ANDREW     7500          4         2        20        18
ANTHONY         SUSANNE    7000          3         1        21        19
ROOSEVELT       ELEANOR                  1        20         1        20
EISENHOWER      DWIGHT                   1        20         1        20

21 rows selected.

End listing
```

Table 9.1 *NULLS FIRST | NULLS LAST Ranking Descriptions*

	Ascending Sort	Descending Sort
NULLS FIRST	Lowest Value/Highest Rank	Highest Value/Highest Rank
NULLS LAST	Highest Value/Lowest Rank	Lowest Value/Lowest Rank

The DENSE_RANK Function

The DENSE_RANK function is similar to the RANK function except for one major difference: It does not skip sequential ranking numbers. In Listing 9.6, two departments had the same total Wages value. The RANK function gave these rows the same rank (3). The rank that would have been used if the fourth row were different (4) was omitted from the ranking scheme. A rank of 5 was given to the first different row. The omitted rank never appeared.

The DENSE_RANK function will ensure that all ranks are used. This function will give duplicate values the same ranking but will use the next sequential rank value for the first nonduplicate value. Listing 9.8 illustrates this function. Unlike Listing 9.6, rank 4 is now used.

Listing 9.8 *Using the DENSE_RANK Function*

```
SQL> select department, sum(nvl(wages, 90000)),
  2         dense_rank() over (order by sum(nvl(wages,90000)) desc)
  3         as rank_dense
  4  from department, employee
  5  where department = fk_department(+)
  6  group by department;

DEPA SUM(NVL(WAGES,90000)) RANK_DENSE
---- -------------------- ----------
INT                155000          1
WEL                142000          2
CEN                 90000          3
TRF                 90000          3
POL                 87700          4

SQL>

End listing
```

Top-N and Bottom-N Queries

The RANK and RANK_DENSE functions can also be used to create Top-N and Bottom-N queries. These types of queries display only a part of the overall ranking. The Top-N query displays a specific number of the highest-ranked values. The Bottom-N query displays a specific number of the lowest-ranked values. To create these types of queries, two steps are required:

1. Create an inline view to develop the data and the rankings.
2. Use the RANK expression in the WHERE clause of the SELECT statement to identify the number of Top and Bottom ranked records.

Listing 9.9 demonstrates a Top-N query. An inline view is used to develop a ranking of employee wages. The NVL function is used to change null values to 0. The ranking expression *emp_wage_rank* is used in the WHERE clause to limit the number of ranks. This query displays the highest-three wage earners.

Listing 9.9 *Top-N Query*

```
SQL> select last_name, first_name, wages, emp_wage_rank
  2  from (select last_name, first_name, wages,
  3        rank() over(order by nvl(wages,0) desc) as emp_wage_rank
  4        from employee)
  5  where emp_wage_rank <= 3;

LAST_NAME       FIRST_NAME          WAGES EMP_WAGE_RANK
--------------- --------------- --------- -------------
CLINTON         WILLIAM             15000             1
BUSH            GEORGE              14000             2
REAGAN          RONALD              13500             3
```

It is very easy to change the query in Listing 9.9 into a Bottom-N query: Simply change the sort order to ASC (ascending). The bottom-three wage earners will then be displayed.

The PARTITION option is an especially effective tool to use in Top-N and Bottom-N queries. This option allows the business analyst to segment the data and to limit the records for each segment. Examples of uses are result sets that display the top-two selling products among a large set of product groups or, in the case of Listing 9.10, the top-two wage earners in each department. In

this example, the data is partitioned by Department_name, each partition is ordered descending, and the top-two ranks are displayed.

Listing 9.10 *Using the PARTITION Option*

```
SQL> select department_name, last_name, first_name, wages,
  2         emp_wage_rank
  3  from (select department_name, last_name, first_name, wages,
  4               rank() over(partition by department_name
  5               order by nvl(wages,0) desc) as emp_wage_rank
  6        from department, employee
  7        where department = fk_department)
  8  where emp_wage_rank <= 2;

DEPARTMENT_NAME LAST_NAME       FIRST_NAME          WAGES EMP_WAGE_RANK
--------------- --------------- --------------- --------- -------------
INTERIOR DESIGN BUSH            GEORGE              14000             1
INTERIOR DESIGN FORD            GERALD              13000             2
POLITICAL SCIEN CLINTON         WILLIAM             15000             1
POLITICAL SCIEN NIXON           RICHARD             12500             2
WELFARE BUREAU  REAGAN          RONALD              13500             1
WELFARE BUREAU  CARTER          JIMMY               13000             2

6 rows selected

End listing
```

The PERCENT_RANK Function

The PERCENT_RANK function computes the percentile of the ranking. The percent rank represents the relative position of the ranking. The ranking is determined by the target value, but the percent rank is not. The percent rank is based on this formula:

(rank of row in its partition – 1) / (number of rows in the partition – 1)

If the partition contains five ranks, the highest-ranked value will be at the 100th (or 1) percent rank. The lowest-ranked value will be at the 0 percent rank. The remainder of the ranked values will be at the 75th, 50th, and 25th percent rank.

Listing 9.11 illustrates the PERCENT_RANK function. The query computes the tool costs for each employee. These costs are then ranked within the respective departments. The PERCENT_RANK function is then used to compute the percent rank of each ranking.

Listing 9.11 *Using the PERCENT_RANK Function*

```
SQL> select department_name, last_name||', '||first_name name,
  2         sum(tool_cost) "Tool Costs",
  3         percent_rank() over
  4         (partition by department_name
  5          order by sum(tool_cost)) percent
  6  from department, employee, emp_tools
  7  where department = fk_department
  8    and payroll_number = fk_payroll_number
  9  group by department_name, last_name, first_name
 10  order by 1, 3 desc;

DEPARTMENT_NAME NAME                             Tool Costs   PERCENT
--------------- -------------------------------- ---------- ---------
INTERIOR DESIGN EISENHOWER, DWIGHT                      375         1
INTERIOR DESIGN ROOSEVELT, THEODORE                     324       .75
INTERIOR DESIGN BUSH, GEORGE                           46.2        .5
INTERIOR DESIGN COOLIDGE, CALVIN                         35       .25
INTERIOR DESIGN FORD, GERALD                             12         0
POLITICAL SCIEN WILSON, WOODROW                      116.95         1
POLITICAL SCIEN ROOSEVELT, FRANKLIN                      20 .66666667
POLITICAL SCIEN NIXON, RICHARD                         18.5 .33333333
POLITICAL SCIEN JOHNSON, ANDREW                        16.7         0
WELFARE BUREAU  ANTHONY, SUSANNE                       88.85         1
WELFARE BUREAU  ROOSEVELT, ELEANOR                     61.95       .75
WELFARE BUREAU  REAGAN, RONALD                          28.7        .5
WELFARE BUREAU  HOOVER, HERBERT                           24       .25
WELFARE BUREAU  TAFT, WILLIAM                             23         0

14 rows selected.

SQL>

End Listing
```

The CUME_DIST Function

CUME_DIST is similar to the PERCENT_RANK function. It has been called the inverse of percentile. It computes the position of a specified value relative to a set of values. The CUME_DIST function does not compute a 0 percentile as does the PERCENT_RANK. If the partition contains five ranks, the highest-ranked value is at the 100th percentile. The lowest-ranked value has a percent rank of .2. The remaining ranks have values of .8, .6, and .4. The formula used in this calculation is as follows:

Cume_dist(x) =
Number of values (different from, or equal to, x)
In the set coming before x in the specified order / size of set

Listing 9.12 is a modification of Listing 9.11. The PERCENT_RANK function was replaced by the CUME_DIST function.

Listing 9.12 *Using the CUME_DIST Function*

```
SQL> select department_name, last_name||', '||first_name name,
  2         sum(tool_cost) "Tool Costs",
  3         cume_dist() over
  4         (partition by department_name
  5          order by sum(tool_cost)) percent
  6  from department, employee, emp_tools
  7  where department = fk_department
  8    and payroll_number = fk_payroll_number
  9  group by department_name, last_name, first_name
 10  order by 1, 3 desc;

DEPARTMENT_NAME NAME                            Tool Costs   PERCENT
--------------- ------------------------------ ---------- ---------
INTERIOR DESIGN EISENHOWER, DWIGHT                     375         1
INTERIOR DESIGN ROOSEVELT, THEODORE                    324        .8
INTERIOR DESIGN BUSH, GEORGE                          46.2        .6
INTERIOR DESIGN COOLIDGE, CALVIN                        35        .4
INTERIOR DESIGN FORD, GERALD                            12        .2
POLITICAL SCIEN WILSON, WOODROW                     116.95         1
POLITICAL SCIEN ROOSEVELT, FRANKLIN                     20       .75
POLITICAL SCIEN NIXON, RICHARD                        18.5        .5
```

```
POLITICAL SCIEN JOHNSON, ANDREW                          16.7        .25
WELFARE BUREAU  ANTHONY, SUSANNE                        88.85          1
WELFARE BUREAU  ROOSEVELT, ELEANOR                      61.95         .8
WELFARE BUREAU  REAGAN, RONALD                           28.7         .6
WELFARE BUREAU  HOOVER, HERBERT                            24         .4
WELFARE BUREAU  TAFT, WILLIAM                              23         .2

14 rows selected.

SQL>

End Listing
```

The NTILE Function

The NTILE function is used to calculate tertiles, quartiles, and deciles. The function contains a parameter that allows the business analyst to specify the number of segments the distribution will contain. If the value 4 is specified, the values will be placed into one of four quartiles. It may be easier to think of the segments as buckets. The NTILE parameter specifies the number of buckets. The function then places the record into one of the buckets.

Listing 9.13 depicts the NTILE function. The SELECT statement computes the total tool purchases for each employee. The NTILE function is then used to place each of the values into one of four quartiles. The employees with the highest cost of purchases are placed in the first quartile. The employees with no tool purchases are placed in the third and fourth quartiles. The NULLS LAST option was used to force the employees with no tool purchases to the bottom.

Listing 9.13 *Using the NTILE Function to Determine Quartiles*

```
SQL> select last_name||', '||first_name, sum(tool_cost) tool_cost,
  2    NTILE(4) over (order by sum(tool_cost) desc nulls last) quartile
  3  from employee, emp_tools
  4  where payroll_number = fk_payroll_number(+)
  5  group by last_name||', '||first_name;

LAST_NAME||','||FIRST_NAME       TOOL_COST  QUARTILE
-------------------------------- ---------- ---------
EISENHOWER, DWIGHT                      375         1
ROOSEVELT, THEODORE                     324         1
```

```
WILSON, WOODROW                    116.95         1
ANTHONY, SUSANNE                    88.85         1
ROOSEVELT, ELEANOR                  61.95         1
BUSH, GEORGE                         46.2         1
COOLIDGE, CALVIN                       35         2
REAGAN, RONALD                       28.7         2
HOOVER, HERBERT                        24         2
TAFT, WILLIAM                          23         2
ROOSEVELT, FRANKLIN                    20         2
NIXON, RICHARD                       18.5         3
JOHNSON, ANDREW                      16.7         3
FORD, GERALD                           12         3
CARTER, JIMMY                                     3
CLINTON, WILLIAM                                  3
KENNEDY, JOHN                                     4
TRUMAN, HAROLD                                    4
MILLER, KEVIN                                     4
JOHNSON, LYNDON                                   4
DWORCZAK, ALICE                                   4

21 rows selected.

SQL>

End listing
```

The ROW_NUMBER Function

The ROW_NUMBER function is the last of the ranking functions. It is somewhat similar to the ROWNUM pseudocolumn. It numbers the rows sequentially as they are fetched. The difference is that the PARTITION option can be used, which allows the row numbers to be used repeatedly in the result set.

Listing 9.14 illustrates the function. The ROW_NUMBER function is used to number the employees in each department, and its values return to 1 each time a new department in encountered.

Listing 9.14 *Using the ROW_NUMBER Function*

```
SQL> select department_name, last_name||', '||first_name,
  2         row_number() over
  3           (partition by department_name
```

```
  4            order by last_name||', '||first_name nulls last)
  5              "Row Number"
  6  from department, employee
  7  where department = fk_department(+);

DEPARTMENT_NAME LAST_NAME||','||FIRST_NAME       Row Number
--------------- -------------------------------- ----------
CENSUS DEPT     ,                                         1
INTERIOR DESIGN BUSH, GEORGE                              1
INTERIOR DESIGN COOLIDGE, CALVIN                          2
INTERIOR DESIGN EISENHOWER, DWIGHT                        3
INTERIOR DESIGN FORD, GERALD                              4
INTERIOR DESIGN MILLER, KEVIN                             5
INTERIOR DESIGN ROOSEVELT, THEODORE                       6
INTERIOR DESIGN TRUMAN, HAROLD                            7
POLITICAL SCIEN CLINTON, WILLIAM                          1
POLITICAL SCIEN DWORCZAK, ALICE                           2
POLITICAL SCIEN JOHNSON, ANDREW                           3
POLITICAL SCIEN JOHNSON, LYNDON                           4
POLITICAL SCIEN KENNEDY, JOHN                             5
POLITICAL SCIEN NIXON, RICHARD                            6
POLITICAL SCIEN ROOSEVELT, FRANKLIN                       7
POLITICAL SCIEN WILSON, WOODROW                           8
TRESURY DEPAR   ,                                         1
WELFARE BUREAU  ANTHONY, SUSANNE                          1
WELFARE BUREAU  CARTER, JIMMY                             2
WELFARE BUREAU  HOOVER, HERBERT                           3
WELFARE BUREAU  REAGAN, RONALD                            4
WELFARE BUREAU  ROOSEVELT, ELEANOR                        5
WELFARE BUREAU  TAFT, WILLIAM                             6

23 rows selected.

SQL>
```

End Listing

Windowing

Oracle has added to its database some functionality that allows you to calculate values based on a window. The window is a period of time. The functions in this class can be used to compute moving, cumulative, and centered aggregates. They include moving averages, moving sums, moving MIN/MAX, cumulative SUM, and LAG/LEAD. These functions create a value that is based on values that precede or follow the record. The windowing functions can be used in the SELECT and ORDER BY clauses.

The following is a syntax template that can be used for the functions:

```
{Sum|Avg|Max|Min|Count|Stddev|Variance|
    First_value|Last_value} ({value expression1> | * })
Over ({partition by <value expression2>[, ...]]
Order by <value expression3>[collate clause>] [asc | desc]
    [nulls first | nulls last] [,...]
rows | range {{unbounded preceding | <value expression4>
    preceding } | between {unbounded preceding |
    <value expression4> preceding }
and {current row | <value expression4> following }}
```

The first clause in the template is used to identify the type of calculation. The bulk of the calculations is self-explanatory. The FIRST_VALUE calculation returns the first value in the window. The LAST_VALUE calculation returns the last value in the window. The following describe the other options:

- OVER — Tells Oracle that the function will operate over a query result set.
- PARTITION BY — Determines how the data will be segregated for analysis.
- ORDER BY — Determines how the data will be sorted within the partition. Options are ASC (default), DESC, NULLS FIRST, or NULLS LAST.
- ROWS | RANGE — These keywords determine the window used for the calculation. The ROWS keyword is used to specify the

	window as a set of rows. RANGE sets the window as a logical offset. This option cannot be used unless the ORDER BY clause is used.
■ BETWEEN … AND	Determines the starting point and endpoint of the window. Omitting the BETWEEN keyword and specifying only one endpoint will cause Oracle to consider the endpoint as the starting point. The current row will then consist of the current row.
■ UNBOUNDED PRECEDING	Sets the first row of the partition as the window starting point.
■ UNBOUNDED FOLLOWING	Sets the last row of the partition as the endpoint of the window.
■ CURRENT ROW	Sets the current row as the starting point or as the endpoint of the window.

The Cumulative Aggregate Function

A cumulative aggregate function is used to create a checkbook-style running balance. The function computes the cumulative balance after the current value is subtracted or added to the aggregate. Listing 9.15 is an example of this function, which is used to develop a cumulative value of the tool purchases. The records are ordered by tool purchase date, which causes the tool purchase balances to be arranged in chronological order. The UNBOUNDED PRECEDING option was used to set the first row returned as the window starting point.

Listing 9.15 *Using the Cumulative Aggregate Function*

```
SQL> select department, last_name, first_name, tool_cost,
  2         sum(tool_cost)
  3         over (order by purchase_date rows unbounded preceding)
            balance
  3  from department, employee, emp_tools
  4  where department = fk_department
```

```
  5    and payroll_number = fk_payroll_number;

DEPA LAST_NAME       FIRST_NAME      TOOL_COST   BALANCE
---- --------------- --------------- --------- ---------
INT  ROOSEVELT       THEODORE               34        34
INT  ROOSEVELT       THEODORE              290       324
WEL  TAFT            WILLIAM                23       347
POL  WILSON          WOODROW              4.95    351.95
POL  WILSON          WOODROW               100    451.95
POL  WILSON          WOODROW                12    463.95
INT  COOLIDGE        CALVIN                 25    488.95
INT  COOLIDGE        CALVIN                 10    498.95
WEL  HOOVER          HERBERT                 8    506.95
WEL  HOOVER          HERBERT                16    522.95
WEL  ROOSEVELT       ELEANOR                55    577.95
POL  ROOSEVELT       FRANKLIN               12    589.95
WEL  ROOSEVELT       ELEANOR              1.95     591.9
POL  ROOSEVELT       FRANKLIN                8     599.9
WEL  ROOSEVELT       ELEANOR                 5     604.9
WEL  ANTHONY         SUSANNE             43.95    648.85
WEL  ANTHONY         SUSANNE             34.95     683.8
WEL  ANTHONY         SUSANNE              9.95    693.75
INT  EISENHOWER      DWIGHT                 25    718.75
INT  EISENHOWER      DWIGHT                200    918.75
INT  EISENHOWER      DWIGHT                150   1068.75
POL  JOHNSON         ANDREW               5.95    1074.7
POL  JOHNSON         ANDREW              10.75   1085.45
POL  NIXON           RICHARD             12.75    1098.2
POL  NIXON           RICHARD              5.75   1103.95
INT  FORD            GERALD                 12   1115.95
INT  FORD            GERALD                  0   1115.95
INT  FORD            GERALD                  0   1115.95
WEL  REAGAN          RONALD                  4   1119.95
WEL  REAGAN          RONALD               7.95    1127.9
WEL  REAGAN          RONALD              16.75   1144.65
INT  BUSH            GEORGE               2.75    1147.4
INT  BUSH            GEORGE              35.95   1183.35
INT  BUSH            GEORGE                7.5   1190.85

34 rows selected.

End Listing
```

Moving Averages

The cumulative aggregate function can be used to compute moving averages. A moving average is a calculation based on a number of values. A three-day moving average consists of averaged values from three separate days. The purpose of a moving average is to remove the volatility of the values. Individual values often differ from each other. When an individual value is averaged with neighboring values, the variance is reduced.

Moving averages are computed when several keywords are added to the cumulative aggregate function:

- The RANGE INTERVAL value keywords are needed to identify the number of values to be averaged.
- A time unit value is needed. Common time units consist of the keywords Year, Month, and Day.
- The PRECEDING and/or FOLLOWING keywords must also be included. They indicate which records in the ordered list will be used in the calculation.

The SELECT statement must contain a date/time value. This value is used in the ORDER BY clause of the function to set the proper order of the records.

Listing 9.16 contains a SELECT statement that computes a moving average. The cumulative aggregate function averages tool records for the preceding 20 years. The Purchase_date expression is used to order the records. The RANGE INTERVAL is set to 20 years. This will cause the function to average the current row with any preceding row that has a Purchase_date value within 20 years of the current row.

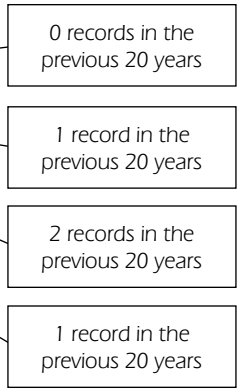
Listing 9.16 *Computing a 20-Year Moving Average of Tool Purchases*

```
SQL> select department, purchase_date, tool_cost,
  2         avg(tool_cost) over
  3           (partition by department
  4            order by purchase_date
  5           range interval '20' year preceding) balance
  6  from department, employee, emp_tools
  7  where department = fk_department
  8    and payroll_number = fk_payroll_number;
```

```
DEPA PURCHASE_ TOOL_COST   BALANCE
---- --------- --------- ---------
INT  01-FEB-03        34        34
INT  10-MAR-05       290       162
INT  01-OCT-22        25 116.33333
INT  01-FEB-23        10     89.75
INT  01-MAR-53        25        25
INT  31-MAR-53       200       125
...
...
WEL  30-NOV-34         5     17.19
WEL  31-MAY-40     43.95     21.65
WEL  12-DEC-41     34.95     23.55
WEL  03-MAY-43      9.95     21.85
WEL  04-JUN-80         4         4
WEL  24-APR-81      7.95     5.975
WEL  06-NOV-82     16.75 9.5666667

34 rows selected.

End Listing
```



Changing the PRECEDING keyword to FOLLOWING will cause the function to average records that follow the current record. Some moving-average calculations require records that precede and follow the current value to be averaged. These types of calculations are called centered moving averages. Centered calculations can be developed by using the BETWEEN and AND keywords. Listing 9.17 contains a query that averages records that exist on both sides of the current record. The RANGE statement contains two interval statements: The first allows records from the preceding 20 years to be included in the average, and the second allows records from the following 20 years to be included.

Listing 9.17 *Computing a 40-Year Centered Moving Average*

```
SQL> select department, purchase_date, tool_cost,
  2         avg(tool_cost) over
  3           (partition by department
  4            order by purchase_date
  5           range interval '20' year preceding) balance
```

```
  6  from department, employee, emp_tools
  7  where department = fk_department
  8    and payroll_number = fk_payroll_number;

DEPA PURCHASE_ TOOL_COST   BALANCE
---- --------- --------- ---------
INT  01-FEB-03        34        34
INT  10-MAR-05       290       162
INT  01-OCT-22        25 116.33333
INT  01-FEB-23        10     89.75
INT  01-MAR-53        25        25
INT  31-MAR-53       200       125
INT  31-MAR-53       150       125
...
WEL  12-MAR-29         8      15.5
WEL  31-MAY-29        16 15.666667
WEL  01-MAY-33        55 26.333333
WEL  06-SEP-34      1.95   20.2375
WEL  30-NOV-34         5     17.19
WEL  31-MAY-40     43.95     21.65
WEL  12-DEC-41     34.95     23.55
WEL  03-MAY-43      9.95     21.85
WEL  04-JUN-80         4         4
WEL  24-APR-81      7.95     5.975
WEL  06-NOV-82     16.75 9.5666667

34 rows selected.

SQL>

End listing
```

The RATIO_TO_REPORT Function

The RATIO_TO_REPORT function computes the ratio of the current row's value to the sum of the result set's values. The following is a syntax template of the function:

Ratio_to_report (<*value expression1*>) over
([partition by <*value expression2*> [, . . .]])

The PARTITION BY keywords define subgroups of the data based on an expression. The ratio will then be computed for each of the subgroups. If the keywords are omitted, the function computes ratios based on the sum of the entire record set's values. Listing 9.18 illustrates this function, which is used to compute the ratio of each department's wages compared to the total employee wages.

Listing 9.18 *Using the RATIO_TO_REPORT Function*

```
SQL> select department_name, sum(wages) "Total Wages",
  2         ratio_to_report(sum(wages)) over() as wage_ratio
  3  from department, employee
  4  where department = fk_department
  5  group by department_name;

DEPARTMENT_NAME Total Wages WAGE_RATIO
--------------- ----------- ----------
INTERIOR DESIGN       65000  .31753786
POLITICAL SCIEN       87700  .42843185
WELFARE BUREAU        52000  .25403029

End Listing
```

The LAG and LEAD Functions

The LAG and LEAD functions return the value of a preceding or following row to the current row. These functions are useful for computing the difference between values in different rows. The following is a syntax template of the functions:

Lag (*expression, record offset*)
Lead (*expression, record offset*)

EXPRESSION refers to one of values contained in the SELECT clause. RECORD OFFSET refers to a row. The value identifies the particular row. A value of 1 indicates the previous or following row, depending on the use of LAG/LEAD. A value of 2 indicates the second row that precedes or follows the current row. If this parameter is omitted, it will default to 1. The function can

be used with the standard PARTITION BY, ORDER BY, and NULLS FIRST | NULLS LAST options.

Listing 9.19 illustrates the LAG function, which is used to retrieve the value of the previous Wages expression. This value is then subtracted from the current value of Wages. The result is the difference between the Wages value in consecutive rows.

Listing 9.19 *Using the LAG Function*

```
SQL> select last_name, wages,
  2         wages - lag(wages, 1) over (order by employment_date)
  3         as "Wage Differences"
  4  from employee;

LAST_NAME           WAGES Wage Differences
--------------- --------- ----------------
ROOSEVELT            8000
TAFT                 8500              500
WILSON               9000              500
COOLIDGE             9500              500
HOOVER              10000              500
ROOSEVELT
ROOSEVELT           10400
ANTHONY              7000            -3400
TRUMAN              11000             4000
EISENHOWER
KENNEDY             11500
JOHNSON             12000              500
JOHNSON              7500            -4500
NIXON               12500             5000
FORD                13000              500
CARTER              13000                0
REAGAN              13500              500
BUSH                14000              500
CLINTON             15000             1000
DWORCZAK             9800            -5200
MILLER               9500             -300

21 rows selected.

End Listing
```

Listing 9.20 depicts the LEAD function. This query performs the same basic function as the previous query. The difference is that the LEAD function retrieves the following row for the calculation.

Listing 9.20 *Using the LEAD Function*

```
LAST_NAME           WAGES Wage Differences
--------------- --------- ----------------
ROOSEVELT            8000              500
TAFT                 8500              500
WILSON               9000              500
COOLIDGE             9500              500
HOOVER              10000
ROOSEVELT
ROOSEVELT           10400            -3400
ANTHONY              7000             4000
TRUMAN              11000
EISENHOWER
KENNEDY             11500              500
JOHNSON             12000            -4500
JOHNSON              7500             5000
NIXON               12500              500
FORD                13000                0
CARTER              13000              500
REAGAN              13500              500
BUSH                14000             1000
CLINTON             15000            -5200
DWORCZAK             9800             -300
MILLER               9500

21 rows selected.

End listing
```

Another use of the LEAD and LAG functions is to avoid the use of the self join. Listing 5.6 in Chapter 5 illustrated a SELECT statement that had a self join, the purpose of which was to display the employees hired before a given employee. Listing 9.21 computes similar results using the LAG function. Using the LEAD and LAG functions rather than a self join can offer two important advantages:

1. Enhanced performance. Using the functions allows Oracle to avoid the work of joining records.
2. Reduces the complexity of the statements. Listing 5.6 was substantially more complex. The developer had to perform outer joins and determine how to join the records. Even so, it did not quite display the values correctly because some of the employees in the Employee table did not have sequential payroll numbers. The previous employee was not displayed if the employee did not have the previous sequential payroll number. The SELECT statement in Listing 9.21 avoids this problem.

Listing 9.21 *Using the LAG Function to Determine the Previously Hired Employee and to Avoid Performing a Self Join*

```
SQL> select payroll_number, last_name,
  2         lag(payroll_number, 1) over
  3               (order by payroll_number) prev_payroll_number,
  4         lag(last_name, 1) over (order by payroll_number)
  5         prev_last_name
  4  from employee;

PAYROLL_NUMBER LAST_NAME       PREV_PAYROLL_NUMBER PREV_LAST_NAME
-------------- --------------- ------------------- ---------------
            19 ROOSEVELT
            20 ANTHONY                          19 ROOSEVELT
            21 JOHNSON                          20 ANTHONY
            22 ROOSEVELT                        21 JOHNSON
            23 TAFT                             22 ROOSEVELT
            24 WILSON                           23 TAFT
            25 COOLIDGE                         24 WILSON
            26 HOOVER                           25 COOLIDGE
            27 ROOSEVELT                        26 HOOVER
            28 TRUMAN                           27 ROOSEVELT
            29 EISENHOWER                       28 TRUMAN
            30 KENNEDY                          29 EISENHOWER
            31 JOHNSON                          30 KENNEDY
            32 NIXON                            31 JOHNSON
            33 FORD                             32 NIXON
            34 CARTER                           33 FORD
            35 REAGAN                           34 CARTER
            36 BUSH                             35 REAGAN
```

```
            37 CLINTON                      36 BUSH
            45 DWORCZAK                     37 CLINTON
            46 MILLER                       45 DWORCZAK

21 rows selected.

SQL>

End listing
```

Statistical Functions

Oracle has added a variety of statistical functions to its database. Statistics such as correlation, covariance, and linear regression can be computed. The functions operate on unordered result sets, and they can be used with the windowing function. Table 9.2 describes the available functions

The use of the various functions is depicted in Listing 9.22. The functions are used to analyze the employee Wages values.

Table 9.2 *Statistical Functions*

Function Name	Description
VAR_POP	Computes the population variance of a number set after discarding all null values
VAR_SAMP	Computes the sample variance of a number set after discarding the null values (This function is similar to the VARIANCE function. The difference occurs when the function takes a single argument. When this occurs, the VARIANCE function returns 0, and the VAR_SAMP function returns a null.)
STDDEV_POP	Computes the standard deviation of the population
STDDEV_SAMP	Computes the standard deviation of the sample
COVAR_POP	Computes the population covariance of a set of number pairs (Oracle will eliminate all pairs that contain a null value.)
COVAR_SAMP	Computes the covariance of the sample set of number pairs (Oracle will eliminate all pairs that contain a null value.)

Listing 9.22 *Using Statistical Functions to Analyze Wages*

```
SQL> select var_pop(wages) var_pop, var_samp(wages) var_samp,
  7          stddev_pop(wages) stddev_pop,
  8          stddev_samp(wages) stddev_samp,
  4          covar_pop(wages, wages * 1.07) covar_pop,
  5          covar_samp(wages, wages * 1.07) covar_samp
  6  from employee;

  VAR_POP  VAR_SAMP STDDEV_POP STDDEV_SAMP COVAR_POP COVAR_SAMP
--------- --------- ---------- ----------- --------- ----------
5082991.7 5365380.1  2254.5491    2316.329 5438801.1  5740956.7

End listing
```

Oracle also has developed a number of linear regression functions, which are detailed in Table 9.3.

The uses of the various regression functions are depicted in Listing 9.23. This query groups and summarizes tool costs by year within a decade (i.e., 1, 2, 3, etc.). This was done in order to reduce the number of data points.

Listing 9.23 *Using Regression Functions to Analyze Wages*

```
SQL> select regr_count(trunc(substr(purchase_date, 8, 2),-1),
  2                    sum(tool_cost)) regr_count,
  3         regr_avgx(trunc(substr(purchase_date, 8, 2),-1),
  4                    sum(tool_cost)) regr_avgx,
  5         regr_avgy(trunc(substr(purchase_date, 8, 2),-1),
  6                    sum(tool_cost)) regr_avgy,
  4         regr_slope(trunc(substr(purchase_date, 8, 2),-1),
  5                    sum(tool_cost)) regr_slope,
  6         regr_intercept(trunc(substr(purchase_date, 8, 2),-1),
  7                    sum(tool_cost)) regr_intercept,
  8         regr_r2(trunc(substr(purchase_date, 8, 2),-1),
  9                    sum(tool_cost)) regr_r2,
  10        regr_sxx(trunc(substr(purchase_date, 8, 2),-1),
  11                   sum(tool_cost)) regr_sxx,
  12        regr_sxy(trunc(substr(purchase_date, 8, 2),-1),
  13                   sum(tool_cost)) regr_sxy
  17 from emp_tools
  18 group by trunc(substr(purchase_date, 8, 2),-1);
```

```
RG_CNT RG_AVGX RG_AVGY RG_SLP RG_INT   REGR_R2  REGR_SXX  REGR_SXY
------ ------- ------- ------ ------ --------- --------- ---------
     9 132.316      40  -.081 50.771 .15708558 142226.18    -11578

SQL>

End Listing
```

Table 9.3 *Linear Regression Functions*

Function Name	Description
REGR_AVGX	Computes the average of the independent variable of the regression line. This is the average of the second argument after nulls are eliminated.
REGR_AVGY	Computes the average of the dependent variable of the regression line. This is the average of the first argument after nulls are eliminated.
REGR_COUNT	Computes the number of not null number pairs that are used to fit the regression line.
REGR_INTERCEPT	Computes the y-intercept of the regression line.
REGR_R2	Computes the coefficient of determination for the regression line.
REGR_SLOPE	Computes the slope of the regression line. It is fitted to not null pairs.
REGR_SXX	Computes a diagnostic statistic for regression analysis. The following is the formula: REGR_COUNT(e1, e2) * VAR_POP(e2)
REGR_SXY	Computes a diagnostic statistic for regression analysis. The following is the formula: REGR_COUNT(e1, e2) * COVAR_POP(e1,e2)
REGR_SYY	Computes a diagnostic statistic for regression analysis. The following is the formula: REGR_COUNT(e1, e2) * VAR_POP(e1)

What's Next?

The next chapter will cover views and sequences. A view is an important tool for the business analyst. It is a SELECT statement that resides in the database. Views are a very common tool for creating virtual record sets. Sequences are a database tool that returns unique and unused numbers to the calling object. It is a great tool for generating artificial primary keys, payroll numbers, or account numbers.

PRACTICE

1. Determine the cost of tools per classification within gender. Subtotal the costs for each gender.
2. Rank all employees by their total cost of eyeglasses and tool purchases. The employee with the lowest cost should be ranked first.
3. Determine the two employees in each department who had the largest cost of eyeglasses purchases. Include employees who have not purchased eyeglasses.
4. Create a checkbook-style cumulative cost of eyeglasses purchases.
5. Compute the ratio of the total cost of each department's eyeglasses purchases to the cost of all eyeglasses purchases.
6. Determine whether the price of tools per tool is increasing. Use the LAG function to calculate the difference in the costs of each tool.
7. Create a SELECT statement that counts the number of eyeglasses within one of four cost classes: Less than $100, $100 to $125, $126 to $150, and Above $150.