Introduction to Analytic Functions

We begin by examining the name, department, and salary of all staff, presenting rows sorted by name within department. Our query will reveal details about individual employees. We separate rows for each department using the SQL*Plus BREAK command.

```
BREAK ON DEPT SKIP 1 DUPLICATES

SELECT ENAME, DEPT, ESALARY
FROM STAFF
ORDER BY DEPT, ENAME;
```

Assume the need to show the total salary for employees in each department. The following query will produce departmental salary summary information.

```
SELECT DEPT, SUM(ESALARY)
FROM STAFF
GROUP BY DEPT;
```

How can details of individuals be preserved and presented with departmental summaries? The SQL*Plus COMPUTE command can perform computations on individual column values within a break column.

```
COMPUTE SUM OF ESALARY ON DEPT;

SELECT DEPT, ENAME, ESALARY
FROM STAFF
ORDER BY DEPT, ENAME;
```
Running Total Example

But the total is a summary reflective of the entire group of rows in a particular department.

How can we show detail information of individuals together with the departmental summary while also revealing a running accumulation of this summary? That is, how do we show the cumulative effect of each new row encountered in the group?

In the following query, RTOT computes a running total for the entire query. DTOT computes a running total for each department. Rows in each group are sequentially numbered by the ROW_NUMBER function.

The ORDER BY clause of the query controls the order in which rows are returned to the user.

```
SELECT ENAME, DEPT, ESALARY,
     SUM(ESALARY) OVER
     (ORDER BY DEPT, ENAME) AS "RTOT",
     SUM(ESALARY) OVER
     (PARTITION BY DEPT
      ORDER BY ENAME) AS "DTOT",
     ROW_NUMBER() OVER
     (PARTITION BY DEPT
      ORDER BY ENAME) AS "SEQUENCE"
FROM STAFF
ORDER BY DEPT, ENAME;
```
OVER Keyword

An analytic function is applied "over" a partition of rows from the result set of the query. The call to the analytic function with its arguments is coded before the keyword OVER. Analytic functions take 0 to 3 arguments.

The presence of the OVER keyword identifies the function as an analytic function. OVER is followed by an expression that describes the data to which the analytic function will be applied, i.e., the data "over" which the function will be performed. This keyword defines the partitioning criteria as well as the ordering of rows within the partition.

If partitioning is not explicit, the entire result set will be treated as a single partition. Specifying an explicit partitioning value causes the result set to be "partitioned" into groups of rows against which the analytic function will be applied.

When a change is detected in the value of the partitioning field, a new group is defined and computations are reset. Thus, evaluation of analytic functions respects the boundaries of a group or partition.

When partitioning, an ORDER BY clause may be included to control the order in which rows are added to the partition. Although use of the ORDER BY clause is optional when defining a partition, functions that perform inter-row analysis (e.g., LAG, LEAD) depend on the order of rows within a partition. When using a windowing function, rows within a partition must be ordered.
Applying an Analytic Function

In the previous query, the RTOT column results from applying the SUM(ESALARY) over the rows in a partition. The OVER keyword is followed by a partition definition which does not specify a particular field. Thus, the SUM function treats the entire result set of the query as a single partition.

```
SUM(ESALARY) OVER
(ORDER BY DEPT, ENAME) AS "RTOT"
```

The DTOT column results from applying the SUM(ESALARY) over the rows in a partition defined using DEPT as the partitioning column. The result is computed for each department, so the total is reset when the value in DEPT changes.

```
SUM(ESALARY) OVER
(PARTITION BY DEPT
  ORDER BY ENAME) AS "DTOT"
```

The SEQUENCE column results also rely on a partition defined using DEPT as the partitioning column, so the sequential numbering is reset for each department.

```
ROW_NUMBER() OVER
(PARTITION BY DEPT
  ORDER BY ENAME) AS "SEQUENCE"
```

This example also illustrates that each analytic function in a query may use a different partitioning clause.
Ordering Rows in a Partition

The ORDER BY clause in a partition definition determines the order in which rows are processed within a partition. Analytic functions always operate on rows in the order specified in the ORDER BY clause of the function. However, the ORDER BY clause of the function does not guarantee the order of the result.

The ORDER BY clause in the query controls the order in which rows are returned to the user. It will often be the case that the ordering expressions of the partition and the query are the same. However, each analytic function in a query may use a different partitioning clause.

Each of the next three queries applies the same analytic function over a partition. Observe the ordering of rows within the partition and for the query result.

```sql
SELECT ENAME, ESALARY, AVG(ESALARY) OVER () AS "AVERAGE"
FROM STAFF;

SELECT ENAME, ESALARY, AVG(ESALARY) OVER (ORDER BY ENAME) AS "AVERAGE"
FROM STAFF;

SELECT ENAME, ESALARY, AVG(ESALARY) OVER (ORDER BY ENAME) AS "AVERAGE"
FROM STAFF
ORDER BY ENAME;
```
The Window Concept

Analytic functions return multiple rows for each group of rows, called a window. The window determines the range of rows used to perform the calculations for the current row. A window is defined by the analytic_clause.

Syntax:

```
analytic_function([args]) OVER
(analytic_clause)
```

For each row in a partition, a sliding window of data can be defined. Such a window determines the range of rows used to perform calculations for the current row. The current row serves as a reference point determining the start and end of the window.

For instance, a centered moving average calculation could be defined with a window holding the current row, the six preceding rows, and the following six rows - a sliding window of 13 rows.

While a partition might define all rows in a department, a sliding window could refer to a subset of rows within the department. For example, relative to row X, we could examine rows X-2, X-1, X+1, and X+2. The rows within the window would change when X identified a different row.

Window size can be based on either a physical number of rows or a logical interval such as time. Each window has a starting row and an ending row.
The Window Concept

Depending on its definition, the window may move at one or both ends. For instance, a window defined for a cumulative sum function would have its starting row fixed at the first row of its partition, and its ending row would slide from the starting point all the way to the last row of the partition.

In contrast, a window defined for a moving average would have both its starting and end points slide so that they maintain a constant physical or logical range.

A window can be set as large as all the rows in a partition or just a sliding window of one row within a partition.

When a window is near a border, the function returns results for only the available rows, rather than issuing a warning.
The Window Concept

In addition to a sliding window, a window may be anchored. The default window is an anchored windowed, starting at the first row of the partition and continuing to the current row.

A sliding window is defined by specifying a distance before or after the current row. This distance can be specified as a fixed number of rows (an offset) or as a range based on some values.

Note that the use of a window requires the ordering of rows in a partition, i.e., the ORDER BY clause.

Windowing functions can be used to compute cumulative, moving, and centered aggregates. They return a value for each row in the table, which depends on other rows in the corresponding window.

With windowing aggregate functions, moving and cumulative versions of many functions such as SUM, AVERAGE, COUNT, MAX, and MIN can be calculated.

Windowing aggregate functions include FIRST_VALUE, which returns the first value in the window; and LAST_VALUE, which returns the last value in the window. These functions provide access to more than one row of a table without a self-join.

Note that the DISTINCT keyword is not supported in windowing functions except for MAX and MIN.
The Window Concept

The analytic_clause may contain a query_partition_clause, order_by_clause, and windowing_clause.

analytic_clause syntax:

    [query_partition_clause]
    [order_by_clause [windowing_clause]]

The query_partition_clause uses the PARTITION BY clause to partition the query result set into groups based on one or more valexpr. If this clause is not supplied, the function treats all rows of the query result set as a single group.

It is possible to specify multiple analytic functions in the same query, each with the same or different PARTITION BY keys.

query_partition_clause syntax:

    PARTITION BY
    {valexpr[,] valexpr]... |
    (valexpr[,] valexpr]...)}

The order_by_clause specifies how data are ordered within a partition. Within each function, it is possible to specify multiple ordering expressions.

NULLS LAST is the default for ascending order, and NULLS FIRST is the default for descending order.
**The Window Concept**

The windowing_clause defines the rows in the window.

**windowing_clause syntax:**

```
{ROWS | RANGE}
{BETWEEN
  {UNBOUNDED PRECEDING
   | CURRENT ROW
   | valexpr {PRECEDING | FOLLOWING}
  }
  AND
  {UNBOUNDED FOLLOWING
   | CURRENT ROW
   | valexpr {PRECEDING | FOLLOWING}
  }
 | {UNBOUNDED PRECEDING
   | CURRENT ROW
   | valexpr PRECEDING
  }
}
```

The NULL semantics of window functions matches the NULL semantics for SQL aggregate functions. All aggregate functions except COUNT(*), GROUPING, and GROUPING_ID ignore nulls. For the remaining aggregate functions, if the data set contains no rows, or contains only rows with nulls as arguments to the aggregate function, then the function returns null.
RANGE Windows

The value specified by the RANGE keyword indicates a *logical offset* from the current row. RANGE 100 PRECEDING will generate a sliding window that has the set of all preceding rows in the partition such that they are within 100 units of the current row. The number of units must be a constant or expression that evaluates to a positive numeric value or an interval literal.

RANGE allows only one expression in the ORDER BY clause.

- If the range value evaluates to a numeric value, the ORDER BY expression must be a numeric or DATE data type.

- If range value evaluates to an interval value, the ORDER BY expression must be a DATE data type.

An analytic function using RANGE can use multiple sort keys in its ORDER BY clause if it specifies any of the following windows:

- RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
- The short form: RANGE UNBOUNDED PRECEDING

- RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING

- RANGE BETWEEN CURRENT ROW AND CURRENT ROW

- RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
RANGE Windows

Window boundaries other than these can have only one sort key in the ORDER BY clause of the analytic function.

If the windowing clause is omitted, the default is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW.

The EMPLOYEE table will be use to illustrate the use of windows. Each row in the table includes a HIREDATE column.

SELECT *
FROM EMPLOYEE
ORDER BY HIREDATE;

The FIRST_VALUE function returns the value of the expression passed as its argument using the first row in the window. The LAST_VALUE function returns the value of the expression passed as its argument using the last row in the window. In the following example, LAST_VALUE would identify the current row.

SELECT ENAME, HIREDATE,
    HIREDATE-90 AS "BACKSPAN",
    FIRST_VALUE(ENAME) OVER
    (ORDER BY HIREDATE ASC
    RANGE 90 PRECEDING) AS "PRECEDING_EMPLOYEE",
    FIRST_VALUE(HIREDATE) OVER
    (ORDER BY HIREDATE ASC
    RANGE 90 PRECEDING) AS "PRECEDING_HIREDATE"
FROM EMPLOYEE
ORDER BY HIREDATE;
RANGE Windows

Rows in the partition are ordered by HIREDATE in ascending sequence. Applying FIRST_VALUE identifies the first ENAME and HIREDATE in the window. BACKSPAN is computed for each row in the partition to determine 90 days prior to the HIREDATE in the current row.

The window defined by FIRST_VALUE includes every row in the sorted partition that precedes the current row where HIREDATE is between the HIREDATE of the current row and BACKSPAN computed for the current row.

The first value of ENAME and HIREDATE are returned as PRECEDING_EMPLOYEE and PRECEDING_HIREDATE, respectively.

Using the COUNT function, we can determine the number of rows that precede the current row in the partition such that these rows have a HIREDATE within 90 days of the HIREDATE in the current row.

\[
\text{COUNT(*) OVER (ORDER BY HIREDATE ASC RANGE 90 PRECEDING)}
\]

Because this partition orders rows in an ascending sequence of HIREDATE values, rows contributing to the count will be those rows in the current partition with a HIREDATE value less than that of the current row and within 100 days of that HIREDATE.
RANGE Windows

```sql
SELECT ENAME, HIREDATE,
    HIREDATE-90 AS "BACKSPAN",
    FIRST_VALUE(ENAME) OVER
    (ORDER BY HIREDATE ASC
     RANGE 90 PRECEDING) AS "PRECEDING_EMPLOYEE",
    FIRST_VALUE(HIREDATE) OVER
    (ORDER BY HIREDATE ASC
     RANGE 90 PRECEDING) AS "PRECEDING_HIREDATE",
    COUNT(*) OVER
    (ORDER BY HIREDATE ASC
     RANGE 90 PRECEDING) AS "HOWMANY_BEFORE"
FROM EMPLOYEE
ORDER BY HIREDATE;
```

If a descending order (DESC) was specified for rows in this partition, the rows preceding the current row would have a HIREDATE value greater than the HIREDATE of the current row and within 100 days of that HIREDATE.

RANGE windows are subject to the limitation that they work only on numbers and dates because it is not possible to subtract from a character string.

RANGE windows are also limited to a single column in the ORDER BY clause.
Row Windows

Row windows use a *physical offset*, the physical number of rows to include in the window.

A window defined as ROWS 5 PRECEDING will consist of up to 6 rows, the current row and up to 5 rows precede it.

The preceding rows are determined by the ORDER BY clause specified in the partition.

```sql
SELECT ENAME, HIREDATE,
       FIRST_VALUE(ENAME) OVER
       (ORDER BY HIREDATE ASC
        ROWS 5 PRECEDING) AS "PRECEDING_ENAME",
       COUNT(*) OVER
       (ORDER BY HIREDATE ASC
        ROWS 5 PRECEDING) AS "PRECEDING_ENAME"
FROM EMPLOYEE
ORDER BY HIREDATE;
```

Row partitions are not subject to the data type restrictions of range partitions and the ORDER BY is not limited to a single column.
Defining Windows

ROWS and RANGE keywords define for each row a window (a physical or logical set of rows) used for calculating the function result. The function is then applied to all rows in the window. The window moves through the query result set or partition from top to bottom.

Use the BETWEEN ... AND clause to specify a start point and end point for the window. The first expression (before AND) defines the start point and the second expression (after AND) defines the end point. If BETWEEN is not used and only one point is supplied, Oracle considers it the start point, and the end point defaults to the current row.

Specify UNBOUNDED PRECEDING to indicate that the window starts at the first row of the partition. This is the start point specification and cannot be used as an end point specification.

Specify UNBOUNDED FOLLOWING to indicate that the window ends at the last row of the partition. This is the end point specification and cannot be used as a start point specification.

As a start point, CURRENT ROW specifies that the window begins at the current row or value (depending on whether you have specified ROW or RANGE, respectively). In this case the end point cannot be value_expr PRECEDING.
Defining Windows

As an end point, CURRENT ROW specifies that the window ends at the current row or value (depending on whether you have specified ROW or RANGE, respectively). In this case the start point cannot be value_expr FOLLOWING.

value_expr PRECEDING or value_expr FOLLOWING

For RANGE or ROW:

If value_expr FOLLOWING is the start point, then the end point must be value_expr FOLLOWING.

If value_expr PRECEDING is the end point, then the start point must be value_expr PRECEDING.

If you are defining a logical window defined by an interval of time in numeric format, then you may need to use conversion functions.

If you specified ROWS:

value_expr is a physical offset. It must be a constant or expression and must evaluate to a positive numeric value.

If value_expr is part of the start point, then it must evaluate to a row before the end point.
RANK and DENSE_RANK Functions

The RANK and DENSE_RANK functions provide the ability to rank items in a group.

The difference between RANK and DENSE_RANK is that DENSE_RANK leaves no gaps in ranking sequence when there are ties.

Suppose five people competed in a race and they finished with the following times:

- Runner1 5 minutes
- Runner2 5 minutes + 10 seconds
- Runner3 5 minutes + 10 seconds
- Runner4 5 minutes + 10 seconds
- Runner5 5 minutes + 30 seconds

The awards for the race would be as follows:

<table>
<thead>
<tr>
<th>Person</th>
<th>DENSE_RANK</th>
<th>RANK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runner1</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; place</td>
<td>1&lt;sup&gt;st&lt;/sup&gt; place</td>
</tr>
<tr>
<td>Runner2</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; place</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; place</td>
</tr>
<tr>
<td>Runner3</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; place</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; place</td>
</tr>
<tr>
<td>Runner4</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; place</td>
<td>2&lt;sup&gt;nd&lt;/sup&gt; place</td>
</tr>
<tr>
<td>Runner5</td>
<td>3&lt;sup&gt;rd&lt;/sup&gt; place</td>
<td>5&lt;sup&gt;th&lt;/sup&gt; place</td>
</tr>
</tbody>
</table>
RANK and DENSE_RANK Functions

The following points are relevant for ranks:

- Ascending is the default sort order, but often descending is needed for analysis.
- Expressions in the PARTITION BY clause divide the query result set into group within which the RANK function operates. RANK gets reset when the group changes.
- If the PARTITION BY clause is not specified, ranks are computed over the entire query result set.
- The ORDER BY clause specifies the measures on which ranking is done and defines the order in which rows are stored in each group. Once data are sorted within each partition, ranks are given to each row starting from 1.
- NULLS FIRST or NULLS LAST indicates the position of NULLs in the ordered sequence.

RANK Syntax:
RANK () OVER
([query_partition_clause] order_by_clause)

DENSE_RANK Syntax:
DENSE_RANK () OVER
([query_partition_clause] order_by_clause)
RANK and DENSE_RANK Functions

The first ranking example demonstrates the general concept of the RANK and DENSE_RANK functions. There are no ties in the query result set, so the results of these functions are the same.

```sql
SELECT DEPT, ENAME, ESALARY,
       RANK() OVER
       (PARTITION BY DEPT
       ORDER BY ESALARY DESC) AS "R",
       DENSE_RANK() OVER
       (PARTITION BY DEPT
       ORDER BY ESALARY DESC) AS "DR"
FROM STAFF;
```

Modify the salary of one employee to create a tie and then evaluate RANK and DENSE_RANK again. The output of the functions will now differ due to the tie.

```sql
UPDATE STAFF
SET ESALARY = 54
WHERE ENAME = 'MARK';

SELECT DEPT, ENAME, ESALARY,
       RANK() OVER
       (PARTITION BY DEPT
       ORDER BY ESALARY DESC) AS "R",
       DENSE_RANK() OVER
       (PARTITION BY DEPT
       ORDER BY ESALARY DESC) AS "DR"
FROM STAFF;
```
**RANK and DENSE_RANK Functions**

A single query block can contain more than one ranking function, each partitioning the data into different groups. This means that the functions are reset on different boundaries.

The groups may be mutually exclusive.

```sql
SELECT DEPT, ENAME, ETITLE, ESALARY,
    RANK() OVER
    (PARTITION BY DEPT
      ORDER BY ESALARY DESC) AS "R1",
    RANK() OVER
    (PARTITION BY ETITLE
      ORDER BY ESALARY DESC) AS "R2"
FROM STAFF
ORDER BY DEPT;
```

<table>
<thead>
<tr>
<th>DEPT</th>
<th>ENAME</th>
<th>ETITLE</th>
<th>ESALARY</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENG</td>
<td>ARCHIMEDES</td>
<td>LAB ASSIST</td>
<td>200</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>MATH</td>
<td>EUCLID</td>
<td>LAB ASSIST</td>
<td>1000</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PHIL</td>
<td>DICK NIX</td>
<td>CROOK</td>
<td>25001</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PHIL</td>
<td>HANK KISS</td>
<td>JESTER</td>
<td>25000</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>THEO</td>
<td>JOHN</td>
<td>EVANGLIST4</td>
<td>54</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>THEO</td>
<td>LUKE</td>
<td>EVANGLIST3</td>
<td>53</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>THEO</td>
<td>MARK</td>
<td>EVANGLIST2</td>
<td>52</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>THEO</td>
<td>MATTHEW</td>
<td>EVANGLIST1</td>
<td>51</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DA VINCI</td>
<td>LAB ASSIST</td>
<td>500</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
RANK and DENSE_RANK Functions

NULLs are treated like normal values. For rank computation, a NULL value is assumed to be equal to another NULL value.

Placement of NULL values depends on the keyword ASC or DESC in the ORDER BY clause and the use of NULLS FIRST or NULLS LAST in the clause. NULLS FIRST is the default.

```
INSERT INTO STAFF
VALUES ('TIM','DBA',NULL, 'PHIL');

SELECT DEPT, ENAME, ETITLE, ESALARY,
     RANK() OVER
     (PARTITION BY DEPT
      ORDER BY ESALARY DESC) AS "R"
FROM STAFF
ORDER BY DEPT;
```

<table>
<thead>
<tr>
<th>DEPT</th>
<th>ENAME</th>
<th>ETITLE</th>
<th>ESALARY</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENG</td>
<td>ARCHIMEDES</td>
<td>LAB ASSIST</td>
<td>200</td>
<td>1</td>
</tr>
<tr>
<td>MATH</td>
<td>EUCLID</td>
<td>LAB ASSIST</td>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>PHIL</td>
<td>TIM</td>
<td>DBA</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>PHIL</td>
<td>DICK NIX</td>
<td>CROOK</td>
<td>25001</td>
<td>2</td>
</tr>
<tr>
<td>PHIL</td>
<td>HANK KISS</td>
<td>JESTER</td>
<td>25000</td>
<td>3</td>
</tr>
<tr>
<td>THEO</td>
<td>JOHN</td>
<td>EVANGLIST4</td>
<td>54</td>
<td>1</td>
</tr>
<tr>
<td>THEO</td>
<td>LUKE</td>
<td>EVANGLIST3</td>
<td>53</td>
<td>2</td>
</tr>
<tr>
<td>THEO</td>
<td>MARK</td>
<td>EVANGLIST2</td>
<td>52</td>
<td>3</td>
</tr>
<tr>
<td>THEO</td>
<td>MATTHEW</td>
<td>EVANGLIST1</td>
<td>51</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>DA VINCI</td>
<td>LAB ASSIST</td>
<td>500</td>
<td>1</td>
</tr>
</tbody>
</table>
RANK and DENSE_RANK Functions

With NULLS LAST specified in the ORDER BY clause, the row with NULL for ESALARY is presented last, receiving the highest rank among rows in the department.

```sql
SELECT DEPT, ENAME, ETITLE, ESALARY,
       RANK() OVER
       (PARTITION BY DEPT
          ORDER BY ESALARY DESC NULLS LAST) AS "R"
FROM STAFF
ORDER BY DEPT;
```

<table>
<thead>
<tr>
<th>DEPT</th>
<th>ENAME</th>
<th>ETITLE</th>
<th>ESALARY</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENG</td>
<td>ARCHIMEDES</td>
<td>LAB ASSIST</td>
<td>200</td>
<td>1</td>
</tr>
<tr>
<td>MATH</td>
<td>EUCLID</td>
<td>LAB ASSIST</td>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>PHIL</td>
<td>DICK NIX</td>
<td>CROOK</td>
<td>25001</td>
<td>1</td>
</tr>
<tr>
<td>PHIL</td>
<td>HANK KISS</td>
<td>JESTER</td>
<td>25000</td>
<td>2</td>
</tr>
<tr>
<td>PHIL</td>
<td>TIM</td>
<td>DBA</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>THEO</td>
<td>JOHN</td>
<td>EVANGLIST4</td>
<td>54</td>
<td>1</td>
</tr>
<tr>
<td>THEO</td>
<td>LUKE</td>
<td>EVANGLIST3</td>
<td>53</td>
<td>2</td>
</tr>
<tr>
<td>THEO</td>
<td>MARK</td>
<td>EVANGLIST2</td>
<td>52</td>
<td>3</td>
</tr>
<tr>
<td>THEO</td>
<td>MATTHEW</td>
<td>EVANGLIST1</td>
<td>51</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>DA VINCI</td>
<td>LAB ASSIST</td>
<td>500</td>
<td>1</td>
</tr>
</tbody>
</table>

DELETE
FROM STAFF
WHERE ENAME = 'TIM';
RANK and DENSE_RANK Functions

CUME_DIST in the inverse of percentile. It computes the position of a specified value relative to a set of values. The range of values for CUME_DIST is from greater than 0 to 1.

The following formula is used to compute the CUME_DIST of a value x in a set S of size N:

$$CUME\_DIST(x) = \frac{\text{number of values in S before and including x in the specified order}}{N}$$

Syntax:

```sql
CUME\_DIST() OVER
([query_partition_clause] order_by_clause)
```

The semantics of CUME_DIST are similar to those in the RANK function. The default order is ascending, implying the lowest values get the lowest CUME_DIST.

NULLS are treated the same as in the RANK function.

```sql
SELECT ETITLE, DEPT, SUM(ESALARY) AS "SALARY",
       CUME\_DIST() OVER
           (PARTITION BY ETITLE
            ORDER BY SUM(ESALARY)) AS "CUME\_DIST"
FROM STAFF
GROUP BY ETITLE, DEPT;
```
RANK and DENSE_RANK Functions

There are 3 rows with ETITLE of LAB ASSIST, so N is equal to 3.

The first row displayed for LAB ASSIST has no rows preceding it, so the numerator in the computation is 0 + the current row, or 1. CUME_DIST evaluates to 1 / 3.

The second row displayed for LAB ASSIST has 1 row preceding it, so the numerator in the computation is 1 + the current row, or 2. CUME_DIST evaluates to 2 / 3.

The third row displayed for LAB ASSIST has 2 rows preceding it, so the numerator in the computation is 2 + the current row, or 3. CUME_DIST evaluates to 3 / 3.

<table>
<thead>
<tr>
<th>ETITLE</th>
<th>DEPT</th>
<th>SALARY</th>
<th>CUME_DIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>CROOK</td>
<td>PHIL</td>
<td>25001</td>
<td>1</td>
</tr>
<tr>
<td>EVANGLIST1</td>
<td>THEO</td>
<td>51</td>
<td>1</td>
</tr>
<tr>
<td>EVANGLIST2</td>
<td>THEO</td>
<td>52</td>
<td>1</td>
</tr>
<tr>
<td>EVANGLIST3</td>
<td>THEO</td>
<td>53</td>
<td>1</td>
</tr>
<tr>
<td>EVANGLIST4</td>
<td>THEO</td>
<td>54</td>
<td>1</td>
</tr>
<tr>
<td>JESTER</td>
<td>PHIL</td>
<td>25000</td>
<td>1</td>
</tr>
<tr>
<td>LAB ASSIST</td>
<td>ENG</td>
<td>200</td>
<td>.333333333</td>
</tr>
<tr>
<td>LAB ASSIST</td>
<td></td>
<td>500</td>
<td>.666666667</td>
</tr>
<tr>
<td>LAB ASSIST</td>
<td>MATH</td>
<td>1000</td>
<td>1</td>
</tr>
</tbody>
</table>

Now add three rows with an ETITLE value of LAB ASSIST. Two of the rows are added to the same department, CIS.
### RANK and DENSE_RANK Functions

```sql
INSERT INTO STAFF
VALUES ('TIM', 'LAB ASSIST', 999, 'CIS');
INSERT INTO STAFF
VALUES ('MOE', 'LAB ASSIST', 1500, 'CIS');
INSERT INTO STAFF
VALUES ('SHEMP', 'LAB ASSIST', 1500, 'MGT');

SELECT ETITLE, DEPT, SUM(ESALARY) AS "SALARY",
      CUME_DIST() OVER
          (PARTITION BY ETITLE
           ORDER BY SUM(ESALARY)) AS "CUME_DIST"
FROM STAFF
GROUP BY ETITLE, DEPT;
```

<table>
<thead>
<tr>
<th>ETITLE</th>
<th>DEPT</th>
<th>SALARY</th>
<th>CUME_DIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>CROOK</td>
<td>PHIL</td>
<td>25001</td>
<td>1</td>
</tr>
<tr>
<td>EVANGLIST1</td>
<td>THEO</td>
<td>51</td>
<td>1</td>
</tr>
<tr>
<td>EVANGLIST2</td>
<td>THEO</td>
<td>52</td>
<td>1</td>
</tr>
<tr>
<td>EVANGLIST3</td>
<td>THEO</td>
<td>53</td>
<td>1</td>
</tr>
<tr>
<td>EVANGLIST4</td>
<td>THEO</td>
<td>54</td>
<td>1</td>
</tr>
<tr>
<td>JESTER</td>
<td>PHIL</td>
<td>25000</td>
<td>1</td>
</tr>
<tr>
<td>LAB ASSIST</td>
<td>ENG</td>
<td>200</td>
<td>.2</td>
</tr>
<tr>
<td>LAB ASSIST</td>
<td></td>
<td>500</td>
<td>.4</td>
</tr>
<tr>
<td>LAB ASSIST</td>
<td>MATH</td>
<td>1000</td>
<td>.6</td>
</tr>
<tr>
<td>LAB ASSIST</td>
<td>MGT</td>
<td>1500</td>
<td>.8</td>
</tr>
<tr>
<td>LAB ASSIST</td>
<td>CIS</td>
<td>2499</td>
<td>1</td>
</tr>
</tbody>
</table>

---

*Introduction to Analytic Functions © Tim Hartley and Associates LLC, 2011*
Hypothetical Rank

The rank functions can be used provide functionality for what-if questions. For example, consider the rank of a row that was inserted into a set of other rows.

In this context, a rank function includes as an argument an expression that evaluates to a constant.

The WITHIN GROUP clause specifies the column values and the ordering of those values for the specified function.

```sql
SELECT DEPT, ESALARY,
       RANK() OVER
       (ORDER BY ESALARY DESC) AS "R"
FROM STAFF;
```

<table>
<thead>
<tr>
<th>DEPT</th>
<th>ESALARY</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHIL</td>
<td>25001</td>
<td>1</td>
</tr>
<tr>
<td>PHIL</td>
<td>25000</td>
<td>2</td>
</tr>
<tr>
<td>CIS</td>
<td>1500</td>
<td>3</td>
</tr>
<tr>
<td>MGT</td>
<td>1500</td>
<td>3</td>
</tr>
<tr>
<td>MATH</td>
<td>1000</td>
<td>5</td>
</tr>
<tr>
<td>CIS</td>
<td>999</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>7</td>
</tr>
<tr>
<td>ENG</td>
<td>200</td>
<td>8</td>
</tr>
<tr>
<td>THEO</td>
<td>54</td>
<td>9</td>
</tr>
<tr>
<td>THEO</td>
<td>53</td>
<td>10</td>
</tr>
<tr>
<td>THEO</td>
<td>52</td>
<td>11</td>
</tr>
<tr>
<td>THEO</td>
<td>51</td>
<td>12</td>
</tr>
</tbody>
</table>
**Hypothetical Rank**

Consider how an employee with a salary of $1200 would rank against existing employees.

Specify the hypothetical salary of 1200 as an argument to the RANK function. Use the ORDER BY clause to identify ESALARY as the column associated with the RANK function argument.

```
SELECT RANK(1200) WITHIN GROUP
    (ORDER BY ESALARY DESC) AS "R"
FROM STAFF;
```

```
R
---
5
```

Consider how an employee with a salary of $500 would rank against existing employees.

```
SELECT RANK(500) WITHIN GROUP
    (ORDER BY ESALARY DESC) AS "R"
FROM STAFF;
```

```
R
---
7
```

If the ranking function includes multiple values, the ORDER BY clause must identify multiple columns. Values and columns are separated by a comma.
NTILE Function

The NTILE function divides an ordered data set into a specified number of buckets indicated by expr and assigns the appropriate bucket number to each row in the partition.

Syntax:

```
NTILE(expr) OVER
([query_partition_clause] order_by_clause)
```

Buckets are numbered 1 through \( expr \), a value that must resolve to a positive constant for each partition. An integer is expected, so if \( expr \) is a noninteger constant, the value will be truncated to an integer.

The return value of NTILE is a NUMBER.

Buckets are calculated so that each bucket has exactly the same number of rows assigned to it or at most 1 row more than the others.

For example, if there are 100 rows in a partition and the NTILE function is specified with four buckets, 25 rows will be assigned a value of 1, 25 rows will have value 2, and so on. These buckets are referred to as *equiheight buckets*. 
NTILE Function

If the number of rows in the partition does not divide evenly into the number of buckets, the number of rows assigned for each bucket will differ by one at most. Extra rows will be distributed one for each bucket starting from the lowest bucket number.

For instance, if there are 103 rows in a partition which has an NTILE(5) function, first 21 rows > 1st bucket, next 21 > 2nd bucket, next 21 > 3rd bucket, next 20 > 4th bucket and final 20 > 5th bucket.

The example divides values in the salary column of rows from department 100 into 4 buckets. There are six salary values for this department. The two extra values (remainder of 6 / 4) are placed in buckets 1 and 2.

```sql
SELECT LAST_NAME, SALARY,
       NTILE(4) OVER (ORDER BY SALARY DESC) AS "QUARTILE"
FROM HR.EMPLOYEES
WHERE DEPARTMENT_ID = 100
ORDER BY LAST_NAME, SALARY, QUARTILE;
```

<table>
<thead>
<tr>
<th>LAST_NAME</th>
<th>SALARY</th>
<th>QUARTILE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chen</td>
<td>8200</td>
<td>2</td>
</tr>
<tr>
<td>Faviet</td>
<td>9000</td>
<td>1</td>
</tr>
<tr>
<td>Greenberg</td>
<td>12008</td>
<td>1</td>
</tr>
<tr>
<td>Popp</td>
<td>6900</td>
<td>4</td>
</tr>
<tr>
<td>Sciarra</td>
<td>7700</td>
<td>3</td>
</tr>
<tr>
<td>Urman</td>
<td>7800</td>
<td>2</td>
</tr>
</tbody>
</table>
Creating Histograms

Use a CASE expression to obtain histograms with user-defined buckets (both in number of buckets and width of each bucket). The following examples show how histograms can be created with CASE statements.

```sql
SELECT SUM(CASE WHEN ESALARY BETWEEN 0 AND 99 THEN 1 ELSE 0 END) AS "0-99",
       SUM(CASE WHEN ESALARY BETWEEN 100 AND 499 THEN 1 ELSE 0 END) AS "100-499",
       SUM(CASE WHEN ESALARY BETWEEN 500 AND 999 THEN 1 ELSE 0 END) AS "500-999",
       SUM(CASE WHEN ESALARY >= 1000 THEN 1 ELSE 0 END) AS "1000"
FROM STAFF;
```

<table>
<thead>
<tr>
<th></th>
<th>0-99</th>
<th>100-499</th>
<th>500-999</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Count</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>
Creating Histograms

SELECT (CASE WHEN ESALARY BETWEEN 0 AND 99 THEN '1: 0-99'
    WHEN ESALARY BETWEEN 100 AND 499 THEN '2: 100-499'
    WHEN ESALARY BETWEEN 500 AND 999 THEN '3: 500-999'
    WHEN ESALARY >= 1000 THEN '4: 1000+
    END) AS "BUCKET",
    COUNT(*) AS "GROUP_COUNT"
FROM STAFF
GROUP BY
    (CASE WHEN ESALARY BETWEEN 0 AND 99 THEN '1: 0-99'
        WHEN ESALARY BETWEEN 100 AND 499 THEN '2: 100-499'
        WHEN ESALARY BETWEEN 500 AND 999 THEN '3: 500-999'
        WHEN ESALARY >= 1000 THEN '4: 1000+
        END)
ORDER BY "BUCKET";

<table>
<thead>
<tr>
<th>BUCKET</th>
<th>GROUP_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: 0-99</td>
<td>4</td>
</tr>
<tr>
<td>2: 100-499</td>
<td>1</td>
</tr>
<tr>
<td>3: 500-999</td>
<td>2</td>
</tr>
<tr>
<td>4: 1000+</td>
<td>5</td>
</tr>
</tbody>
</table>

---

*Introduction to Analytic Functions © Tim Hartley and Associates LLC, 2011*
Reporting Functions

After a query has been processed, aggregate values can be computed within a partition and made available to other reporting functions.

Reporting aggregate functions return the same aggregate value for every row in a partition.

Syntax:

\[{\text{SUM | AVG | MAX | MIN | COUNT | STDDEV ...}}\]
\([\text{[ALL | DISTINCT] \{value expr1 [, ...] | *\}}\]
\(\text{OVER ([PARTITION BY value expr2[, ...]])}\)

The PARTITION BY clause defines groups on which the windowing functions would be computed. If the PARTITION BY clause is not present, the function is computed over the entire query result set.

Reporting functions can appear only in the SELECT clause or the ORDER BY clause. Reporting functions efficiently support multiple passes of data in a single query block.

```
SELECT DEPT, ETITLE, SUM(ESALARY) AS "SALARY",
     MAX(SUM(ESALARY)) OVER
        (PARTITION BY DEPT) AS "MAX_SALARY"
FROM STAFF
GROUP BY DEPT, ETITLE;
```
Reporting Functions

The aggregate function MAX(ESALARY) computes the total salary of employees with the same ETITLE who work in the same DEPT. The reporting aggregate function MAX(SUM(ESALARY)) returns the highest total salary for a DEPT, the partitioning field.

<table>
<thead>
<tr>
<th>DEPT</th>
<th>ETITLE</th>
<th>SALARY</th>
<th>MAX_SALARY</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIS</td>
<td>LAB ASSIST</td>
<td>2499</td>
<td>2499</td>
</tr>
<tr>
<td>ENG</td>
<td>LAB ASSIST</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>MATH</td>
<td>LAB ASSIST</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>MGT</td>
<td>LAB ASSIST</td>
<td>1500</td>
<td>1500</td>
</tr>
<tr>
<td>PHIL</td>
<td>CROOK</td>
<td>25001</td>
<td>25001</td>
</tr>
<tr>
<td>PHIL</td>
<td>JESTER</td>
<td>25000</td>
<td>25001</td>
</tr>
<tr>
<td>THEO</td>
<td>EVANGLIST1</td>
<td>51</td>
<td>54</td>
</tr>
<tr>
<td>THEO</td>
<td>EVANGLIST2</td>
<td>52</td>
<td>54</td>
</tr>
<tr>
<td>THEO</td>
<td>EVANGLIST3</td>
<td>53</td>
<td>54</td>
</tr>
<tr>
<td>THEO</td>
<td>EVANGLIST4</td>
<td>54</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>LAB ASSIST</td>
<td>500</td>
<td>500</td>
</tr>
</tbody>
</table>

Using data from this query, we can reveal for each department, the job title responsible for the highest total salary.

```sql
SELECT DEPT, ETITLE, SALARY
FROM (SELECT DEPT, ETITLE,
        SUM(ESALARY) AS "SALARY",
        MAX(SUM(ESALARY)) OVER
        (PARTITION BY DEPT) AS "MAX_SALARY"
        FROM STAFF
        GROUP BY DEPT, ETITLE)
WHERE SALARY = MAX_SALARY;
```
RATIO_TO_REPORT Function

The RATIO_TO_REPORT function computes the ratio of a value to the sum of a set of values.

Syntax:

    RATIO_TO_REPORT (expr)
    OVER ([query_partition_clause])

The PARTITION BY clause defines the groups on which the function is to be computed. If this clause is not present, the function is computed over the entire query result set.

The following query determines the proportion of each department's salary taken up by its individual employees.

```
SELECT DEPT, ENAME, ESALARY, DEPT_SAL,
    ROUND(EMPLOYEES_DEPT_RATIO*100,2) AS EMPS_PROPORTION
FROM (SELECT DEPT, ENAME, ESALARY,
    SUM(ESALARY) OVER
    (PARTITION BY DEPT) AS DEPT_SAL,
    RATIO_TO_REPORT(ESALARY) OVER
    (PARTITION BY DEPT) AS EMPLOYEES_DEPT_RATIO
    FROM STAFF)
ORDER BY DEPT;
```
LAG and LEAD Functions

The LAG and LEAD functions are useful for comparing values when the relative positions can be known reliably. They work by specifying the count of rows which separate the target row from the current row.

These functions provide the ability to perform inter-row analysis more efficiently than available through a self-join.

LAG() and LEAD() provide access to rows around the current row.

Each function takes three arguments: (arg1, arg2, arg3)

arg1 - expression to be returned from the other row

arg2 - postive integer indicating offset from current row

arg3 - default returned when index (arg2) is outside window

The next example determines the order by which employees joined their respective departments. Each row in the output of the query includes the employees who preceded and succeeded the current employee.
LAG and LEAD Functions

SELECT DEPTNO, ENAME, HIREDATE,
    LAG(ENAME, 1, NULL) OVER
    (PARTITION BY DEPTNO
    ORDER BY HIREDATE ASC
    NULLS LAST) AS PREV_EMP,
    LEAD(ENAME, 1, NULL) OVER
    (PARTITION BY DEPTNO
    ORDER BY HIREDATE ASC
    NULLS LAST) AS NEXT_EMP
FROM SCOTT.EMP
ORDER BY DEPTNO;

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>ENAME</th>
<th>HIREDATE</th>
<th>PREV_EMP</th>
<th>NEXT_EMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>CLARK</td>
<td>09-JUN-81</td>
<td></td>
<td>KING</td>
</tr>
<tr>
<td>10</td>
<td>KING</td>
<td>17-NOV-81</td>
<td>CLARK</td>
<td>MILLER</td>
</tr>
<tr>
<td>10</td>
<td>MILLER</td>
<td>23-JAN-82</td>
<td>KING</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>SMITH</td>
<td>17-DEC-80</td>
<td></td>
<td>JONES</td>
</tr>
<tr>
<td>20</td>
<td>JONES</td>
<td>02-APR-81</td>
<td>SMITH</td>
<td>FORD</td>
</tr>
<tr>
<td>20</td>
<td>FORD</td>
<td>03-DEC-81</td>
<td>JONES</td>
<td>SCOTT</td>
</tr>
<tr>
<td>20</td>
<td>SCOTT</td>
<td>19-APR-87</td>
<td>FORD</td>
<td>ADAMS</td>
</tr>
<tr>
<td>20</td>
<td>ADAMS</td>
<td>23-MAY-87</td>
<td>SCOTT</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>ALLEN</td>
<td>20-FEB-81</td>
<td></td>
<td>WARD</td>
</tr>
<tr>
<td>30</td>
<td>WARD</td>
<td>22-FEB-81</td>
<td>ALLEN</td>
<td>BLAKE</td>
</tr>
<tr>
<td>30</td>
<td>BLAKE</td>
<td>01-MAY-81</td>
<td>WARD</td>
<td>TURNER</td>
</tr>
<tr>
<td>30</td>
<td>TURNER</td>
<td>08-SEP-81</td>
<td>BLAKE</td>
<td>MARTIN</td>
</tr>
<tr>
<td>30</td>
<td>MARTIN</td>
<td>28-SEP-81</td>
<td>TURNER</td>
<td>JAMES</td>
</tr>
<tr>
<td>30</td>
<td>JAMES</td>
<td>03-DEC-81</td>
<td>MARTIN</td>
<td></td>
</tr>
</tbody>
</table>