|  |
| --- |
| , RD Dep. |
| **Oracle SQL for Analysis and Reporting** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial Version | Volha Kutsevol | 14-Feb-2012 |  |  |
|  |  |  |  |  |  |

Contents

[1. Overview of SQL for Analysis and Reporting 3](#_Toc317047317)

[1.1. RANK and DENSE\_RANK Functions 3](#_Toc317047318)

[1.2. Treatment of NULLs 5](#_Toc317047319)

[1.3. CUME\_DIST Function 6](#_Toc317047320)

[1.4. PERCENT\_RANK Function 7](#_Toc317047321)

[1.5. NTILE Function 8](#_Toc317047322)

[1.6. Reporting 9](#_Toc317047323)

[1.7. ROW\_NUMBER Function 9](#_Toc317047324)

[1.8. RATIO\_TO\_REPORT Function 10](#_Toc317047325)

[1.9. LAG/LEAD Function 11](#_Toc317047326)

[1.10. FIRST/LAST Functions 11](#_Toc317047327)

# Overview of SQL for Analysis and Reporting

Oracle has enhanced SQL's analytical processing capabilities by introducing a new family of analytic SQL functions. These analytic functions enable you to calculate:

* Rankings and percentiles
* Lag/lead analysis
* First/last analysis
* Linear regression statistics

Ranking functions include cumulative distributions, percent rank, and N-tiles. Lag/lead analysis enables direct inter-row references so you can calculate period-to-period changes. First/last analysis enables you to find the first or last value in an ordered group.

## RANK and DENSE\_RANK Functions

The RANK and DENSE\_RANK functions allow you to rank items in a group. There are two functions that perform ranking, as shown by the following syntax:

RANK ( ) OVER ( [query\_partition\_clause] order\_by\_clause )

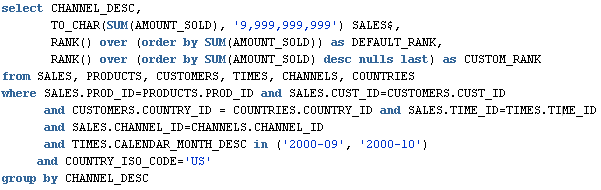
DENSE\_RANK ( ) OVER ( [query\_partition\_clause] order\_by\_clause )

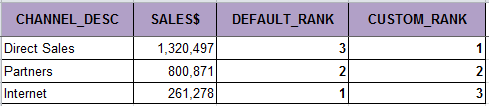
The difference between RANK and DENSE\_RANK is that DENSE\_RANK leaves no gaps in ranking sequence when there are ties.

The following are some relevant points about RANK:

* Ascending is the default sort order, which you may want to change to descending.
* The expressions in the optional PARTITION BY clause divide the query result set into groups within which the RANK function operates. That is, RANK gets reset whenever the group changes. In effect, the value expressions of the PARTITION BY clause define the reset boundaries.
* If the PARTITION BY clause is missing, then ranks are computed over the entire query result set.
* The ORDER BY clause specifies the measures (<value expression>) on which ranking is done and defines the order in which rows are sorted in each group (or partition). Once the data is sorted within each partition, ranks are given to each row starting from 1.
* The NULLS FIRST | NULLS LAST clause indicates the position of NULLs in the ordered sequence, either first or last in the sequence. The order of the sequence would make NULLs compare either high or low with respect to non-NULL values. If the sequence were in ascending order, then NULLS FIRST implies that NULLs are smaller than all other non-NULL values and NULLS LAST implies they are larger than non-NULL values. It is the opposite for descending order.
* If the NULLS FIRST | NULLS LAST clause is omitted, then the ordering of the null values depends on the ASC or DESC arguments. Null values are considered larger than any other values. If the ordering sequence is ASC, then nulls will appear last; nulls will appear first otherwise. Nulls are considered equal to other nulls and, therefore, the order in which nulls are presented is non-deterministic.

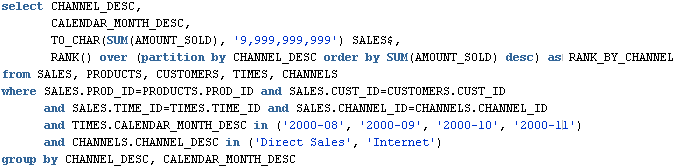
For example:

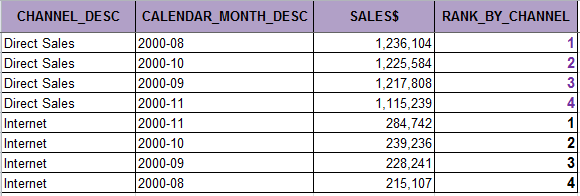




The RANK function can be made to operate within groups, that is, the rank gets reset whenever the group changes. This is accomplished with the **PARTITION BY** clause. The group expressions in the **PARTITION BY** subclause divide the data set into groups within which RANK operates.

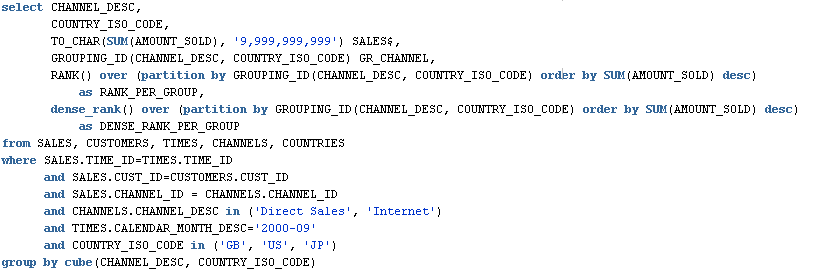
For example:

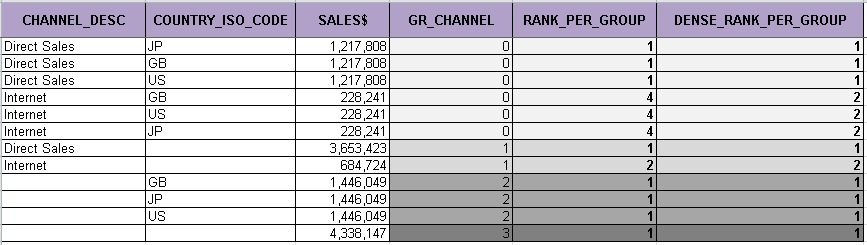




Analytic functions, RANK for example, can be reset based on the groupings provided by a **CUBE, ROLLUP**, or **GROUPING SETS** operator. It is useful to assign ranks to the groups created by CUBE, ROLLUP, and GROUPING SETS queries.

For example:

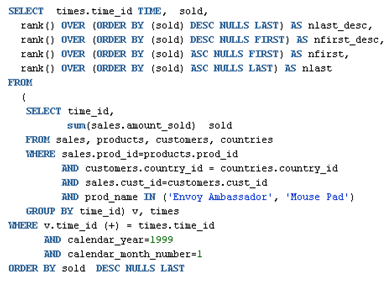


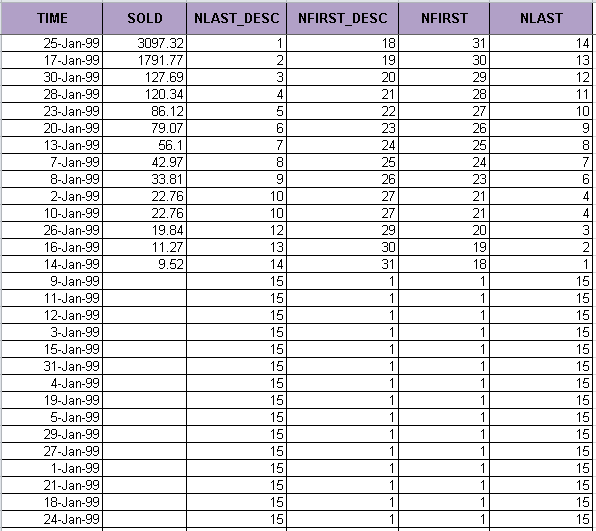


## Treatment of NULLs

NULLs are treated like normal values. Also, for rank computation, a NULL value is assumed to be equal to another NULL value. Depending on the ASC | DESC options provided for measures and the NULLS FIRST | NULLS LAST clause, NULLs will either sort low or high and hence, are given ranks appropriately.

For example:





## CUME\_DIST Function

The CUME\_DIST function (defined as the inverse of percentile in some statistical books) computes the position of a specified value relative to a set of values. The order can be ascending or descending. Ascending is the default. The range of values for CUME\_DIST is from greater than 0 to 1. To compute the CUME\_DIST of a value x in a set S of size N, you use the formula:

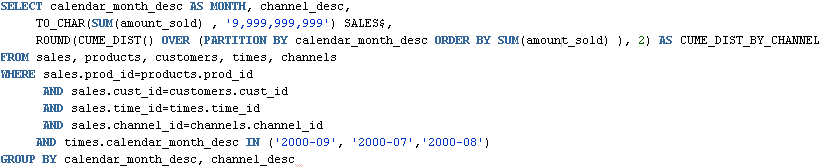
CUME\_DIST(x) = number of values in S coming before and including x in the specified order / N

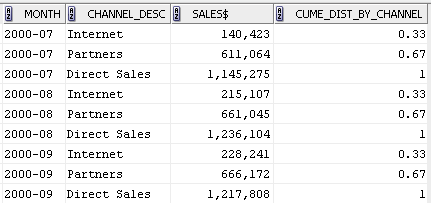
Its syntax is:

CUME\_DIST ( ) OVER ( [query\_partition\_clause] order\_by\_clause )

The semantics of various options in the CUME\_DIST function are similar to those in the RANK function. The default order is ascending, implying that the lowest value gets the lowest CUME\_DIST (as all other values come later than this value in the order). NULLs are treated the same as they are in the RANK function.

For example:





## PERCENT\_RANK Function

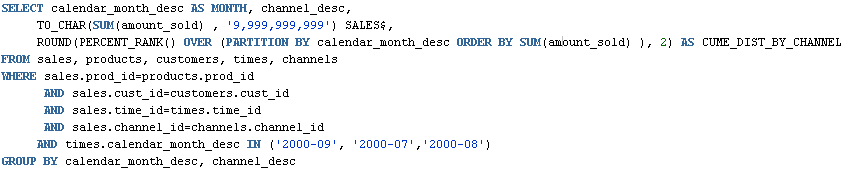
PERCENT\_RANK is similar to CUME\_DIST, but it uses rank values rather than row counts in its numerator. Therefore, it returns the percent rank of a value relative to a group of values. The function is available in many popular spreadsheets. PERCENT\_RANK of a row is calculated as:

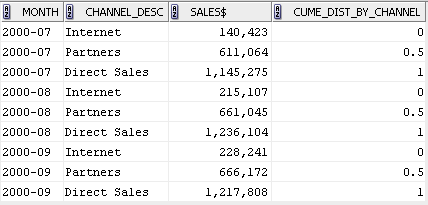
(rank of row in its partition - 1) / (number of rows in the partition - 1)

PERCENT\_RANK returns values in the range zero to one. The row(s) with a rank of 1 will have a PERCENT\_RANK of zero. Its syntax is:

PERCENT\_RANK () OVER ([query\_partition\_clause] order\_by\_clause)

For example:





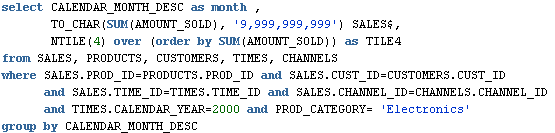
## NTILE Function

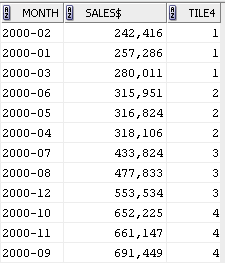
NTILE allows easy calculation of tertiles, quartiles, deciles and other common summary statistics. This function divides an ordered partition into a specified number of groups called buckets and assigns a bucket number to each row in the partition. NTILE is a very useful calculation because it lets users divide a data set into fourths, thirds, and other groupings.

The NTILE function has the following syntax:

NTILE (expr) OVER ([query\_partition\_clause] order\_by\_clause)

For example:





## Reporting

After a query has been processed, aggregate values like the number of resulting rows or an average value in a column can be easily 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. Their behavior with respect to NULLs is the same as the SQL aggregate functions. The syntax is:

{SUM | AVG | MAX | MIN | COUNT | STDDEV | VARIANCE ... }

([ALL | DISTINCT] {value expression1 [,...] | \*})

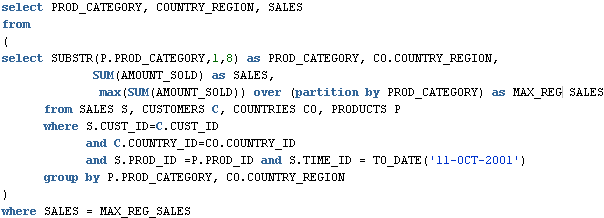
OVER ([PARTITION BY value expression2[,...]])

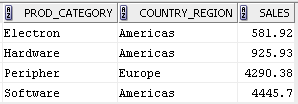
In addition, the following conditions apply:

* An asterisk (\*) is only allowed in COUNT(\*)
* DISTINCT is supported only if corresponding aggregate functions allow it.
* Value expression1 and value expression2 can be any valid expression involving column references or aggregates.
* The PARTITION BY clause defines the groups on which the windowing functions would be computed. If the PARTITION BY clause is absent, then the function is computed over the whole query result set.

Reporting functions can appear only in the SELECT clause or the ORDER BY clause. The major benefit of reporting functions is their ability to do multiple passes of data in a single query block and speed up query performance.

For example, for each product category, find the region in which it had maximum sales:



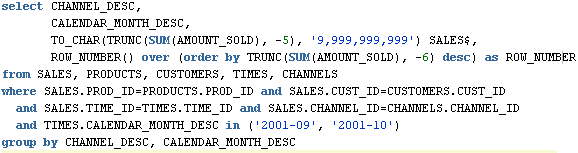


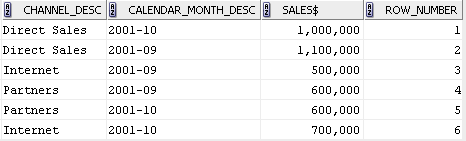
## ROW\_NUMBER Function

The ROW\_NUMBER function assigns a unique number (sequentially, starting from 1, as defined by ORDER BY) to each row within the partition. It has the following syntax:

ROW\_NUMBER ( ) OVER ( [query\_partition\_clause] order\_by\_clause )

For example:





## RATIO\_TO\_REPORT Function

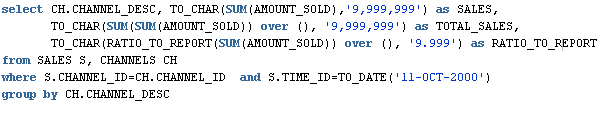
The RATIO\_TO\_REPORT function computes the ratio of a value to the sum of a set of values. If the expression value expression evaluates to NULL, RATIO\_TO\_REPORT also evaluates to NULL, but it is treated as zero for computing the sum of values for the denominator. Its syntax is:

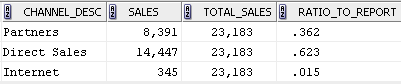
RATIO\_TO\_REPORT ( expr ) OVER ( [query\_partition\_clause] )

In this, the following applies:

* Expr can be any valid expression involving column references or aggregates.
* The PARTITION BY clause defines the groups on which the RATIO\_TO\_REPORT function is to be computed. If the PARTITION BY clause is absent, then the function is computed over the whole query result set.

For example:





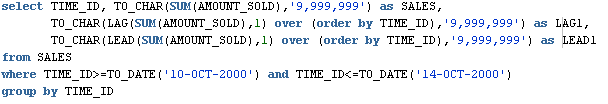
## LAG/LEAD Function

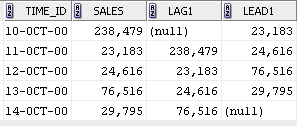
The LAG and LEAD functions are useful for comparing values when the relative positions of rows can be known reliably. They work by specifying the count of rows which separate the target row from the current row. Because the functions provide access to more than one row of a table at the same time without a self-join, they can enhance processing speed. The LAG function provides access to a row at a given offset prior to the current position, and the LEAD function provides access to a row at a given offset after the current position.These functions have the following syntax:

{LAG | LEAD} ( value\_expr [, offset] [, default] ) [RESPECT NULLS|IGNORE NULLS]

OVER ( [query\_partition\_clause] order\_by\_clause )

For example:





## FIRST/LAST Functions

The FIRST\_VALUE and LAST\_VALUE functions allow you to select the first and last rows from a window. These rows are especially valuable because they are often used as the baselines in calculations.

If the IGNORE NULLS option is used with FIRST\_VALUE, it will return the first non-null value in the set, or NULL if all values are NULL. If IGNORE NULLS is used with LAST\_VALUE, it will return the last non-null value in the set, or NULL if all values are NULL. The IGNORE NULLS option is particularly useful in populating an inventory table properly.

These functions have syntax as follows:

FIRST\_VALUE|LAST\_VALUE ( <expr> ) [RESPECT NULLS|IGNORE NULLS]

OVER (analytic clause )

For example:

