2nd Edition

Covers SQL Server, DB2, MySQL, Oracle & PostgreSQL

SQL IN A NUTSHELL

A Desktop Quick Reference

Kevin E. Kline
with Daniel Kline & Brand Hunt

O'REILLY®

4

SQL Functions

A function is a special type of command word in the SQL command set, and each SQL dialect varies in its implementation of the command set. In effect, functions are one-word commands that return single values. The value of a function can be determined by input parameters, as with a function that averages a list of database values. But many functions do not use any type of input parameter, such as the function that returns the current system time, *CURRENT_TIME*.

The ANSI standard supports a number of useful functions. This chapter covers those functions, providing detailed descriptions and examples for each platform. In addition, each database maintains a long list of their own internal functions that are outside of the scope of the standard SQL. This chapter provides parameters and descriptions for each database implementation's internal functions.

In addition, most database platforms support the ability to create user-defined functions (UDFs). For more information on UDFs, refer to the "CREATE/ALTER FUNCTION/PROCEDURE Statements" in Chapter 3.

Types of Functions

There are different ways to categorize functions into groups. The following subsections describe distinctions that are critical to understand how functions work.

Deterministic and Nondeterministic Functions

Functions can be either *deterministic* or *nondeterministic*. A *deterministic* function always returns the same results if given the same input values. A *nondeterministic* function may return different results every time it is called, even when the same input values are provided.

Why is it important that a given input always returns the same output? It is important because of how functions may be used within views, in user-defined functions, and in stored procedures. Restrictions vary across implementations, but these objects sometimes allow only deterministic functions within their defining code. For example, SQL Server allows the creation of an index on a column expression—as long as the expression does not contain nondeterministic functions. Rules and restrictions vary between the platforms, so check the specific documentation when using functions.

Aggregate and Scalar Functions

Another way of categorizing functions is in terms of whether they operate on values from just one row at a time, on values from a collection, or on a set of rows. *Aggregate functions* operate against a collection of values and return a single summarizing value. *Scalar functions* return a single value based on scalar input arguments. Some scalar functions, such as *CURRENT_TIME*, do not require any arguments.

Window Functions

Window functions can be thought of as being similar to aggregate functions in that they operate over many rows at one time. The difference lies in how you define those rows. Aggregate functions operate over the sets of rows defined by a query's *GROUP BY* clause. With window functions, you specify the set of rows for each function call, and different invocations of a function within the same query can execute over different sets of rows.

ANSI SQL Aggregate Functions

Aggregate functions return a single value based upon a set of other values. If used among other expressions in the item list of a *SELECT* statement, the *SELECT* must have a *GROUP BY* or *HAVING* clause. No *GROUP BY* or *HAVING* clause is required if the aggregate function is the only value retrieved by the *SELECT* statement. The supported aggregate functions and their syntax are listed in Table 4-1.

Table 4-1. ANSI SQL aggregate functions

Function	Usage
`AVG(expression)`	Computes the average value of a column given by `expression`.
`CORR(dependent, independent)`	Computes a correlation coefficient.
`COUNT(expression)`	Counts the rows defined by the `expression`.
`COUNT(*)`	Counts all rows in the specified table or view.
`COVAR_POP (dependent, independent)`	Computes population covariance.
`COVAR_SAMP(dependent, independent)`	Computes sample covariance.

Table 4-1. ANSI SQL aggregate functions (continued)

Function	Usage
CUME_DIST(value_list) WITHIN GROUP (ORDER BY sort_list)	Computes the relative rank of a hypothetical row within a group of rows, where the rank is equal to the number of rows less than or equal to the hypothetical row divided by the number of rows in the group.
DENSE_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)	Generates a dense rank (no ranks are skipped) for a hypothetical row (*value_list*) in a group of rows generated by GROUP BY.
MIN(expression)	Finds the minimum value in a column given by *expression*.
MAX(expression)	Finds the maximum value in a column given by *expression*.
PERCENT_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)	Generates a relative rank for a hypothetical row by dividing that row's rank less 1 by the number of rows in the group.
PERCENTILE_CONT (percentile) WITHIN GROUP (ORDER BY sort_list)	Generates an interpolated value that, if added to the group, would correspond to the *percentile* given.
PERCENTILE_DISC (percentile) WITHIN GROUP (ORDER BY sort_list)	Returns the value with the smallest cumulative distribution value greater than or equal to *percentile*.
RANK(value_list) WITHIN GROUP (ORDER BY sort_list)	Generates a rank for a hypothetical row (*value_list*) in a group of rows generated by GROUP BY.
REGR_AVGX(dependent, independent)	Computes the average of the independent variable.
REGR_AVGY(dependent, independent)	Computes the average of the dependent variable.
REGR_COUNT(dependent, independent)	Counts the number of pairs remaining in the group after any pair with one or more NULL values has been eliminated.
REGR_INTERCEPT(dependent,independent)	Computes the y-intercept of the least-squares-fit linear equation.
REGR_R2(dependent, independent)	Squares the correlation coefficient.
REGR_SLOPE(dependent, independent)	Determines the slope of the least-squares-fit linear equation.
REGR_SXX(dependent, independent)	Sums the squares of the independent variables.
REGR_SXY(dependent, independent)	Sums the products of each pair of variables.
REGR_SYY(dependent, independent)	Sums the squares of the dependent variables.
STDDEV_POP(expression)	Computes the population standard deviation of all *expression* values in a group.
STDDEV_SAMP(expression)	Computes the sample standard deviation of all *expression* values in a group.
SUM(expression)	Computes the sum of column values given by *expression*.
VAR_POP(expression)	Computes the population variance of all *expression* values in a group.
VAR_SAMP(expression)	Computes the sample standard deviation of all *expression* values in a group.

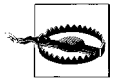
Technically speaking, *ANY*, *EVERY*, and *SOME* are considered aggregate functions. However, they have been discussed as range search criteria since they are most often used that way. Refer to "ALL/ANY/SOME Operators" in Chapter 3 for more information on these functions.

The number of values processed by an aggregate function varies depending on the number of rows queried from the table. This behavior makes aggregate functions different from scalar functions, which can only operate on the values of a single row per invocation.

The general syntax of an aggregate function is:

```
aggregate_function_name ( [ALL | DISTINCT] expression )
```

The aggregate function name may be *AVG*, *COUNT*, *MAX*, *MIN*, or *SUM*, as listed in Table 4-1. The *ALL* keyword, which specifies the default behavior, evaluates all rows when aggregating the value of the function. The *DISTINCT* keyword uses only distinct values when evaluating the function.



All aggregate functions except *COUNT(*)* will ignore NULL values when computing their results.

AVG and SUM

The *AVG* function computes the average of values in a column or an expression. *SUM* computes the sum. Both functions work with numeric values and ignore NULL values. Use the *DISTINCT* keyword to compute the average or sum of all distinct values of a column or expression.

SQL Standard Syntax

```
AVG ([ALL | DISTINCT] expression )
SUM ([ALL | DISTINCT] expression )
```

MySQL, PostgreSQL, and SQL Server

All these platforms support the SQL2003 syntax of *AVG* and *SUM*.

DB2 and Oracle

DB2 and Oracle support the ANSI syntax and the following analytic syntax:

```
AVG ([ALL | DISTINCT] expression ) OVER (window_clause)
SUM ([ALL | DISTINCT] expression ) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section "ANSI SQL Window Functions" later in this chapter.

Examples

The following query computes average year-to-date sales for each type of book:

```
SELECT   type, AVG( ytd_sales ) AS "average_ytd_sales"
FROM     titles
GROUP BY type;
```

This query returns the sum of year-to-date sales for each type of book:

```
SELECT   type, SUM( ytd_sales )
FROM     titles
GROUP BY type;
```

CORR

The *CORR* function returns the correlation coefficient between a set of dependent and independent variables.

SQL2003 Syntax

Calls the function with two variables, one dependent and the other independent:

```
CORR( dependent, independent )
```

Any pair in which either the dependent variable, independent variable, or both are NULL is ignored. The result of the function is NULL when none of the input pairs consist of two non-NULL values.

Oracle

Oracle supports the SQL2003 syntax, and the following analytic syntax:

```
CORR (dependent, independent) OVER (window_ clause)
```

For an explanation of the *window_clause*, see the section "ANSI SQL Window Functions" later in this chapter.

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not support any form of the *CORR* function.

Example

The following *CORR* example uses the data shown by the first *SELECT*:

```
SELECT * FROM test2;

         Y          X
---------- ----------
         1          3
         2          2
         3          1

SELECT CORR(y,x) FROM test2;

 CORR(Y,X)
----------
        -1
```


COUNT

The *COUNT* function is used to compute the number of rows in an expression.

SQL2003 Syntax

```
COUNT(*)
COUNT( [ALL|DISTINCT] expression )
```

COUNT(*)
: Counts all the rows in the target table whether or not they include NULLs.

COUNT([ALL|DISTINCT] expression)
: Computes the number of rows with non-NULL values in a specific column or expression. When the keyword *DISTINCT* is used, duplicate values will be ignored and a count of the distinct values is returned. *ALL* returns the number of non-NULL values in the expression and is implicit when *DISTINCT* is not used.

MySQL, PostgreSQL, and SQL Server

All of these platforms support the SQL2003 syntax of *COUNT*.

DB2 and Oracle

DB2 and Oracle support the ANSI syntax and the following analytic syntax:

```
COUNT ({*|[DISTINCT] expression}) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section later in this chapter titled "ANSI SQL Window Functions."

Examples

This query counts all rows in a table:

```
SELECT COUNT(*) FROM publishers;
```

The following query finds the number of different countries where publishers are located:

```
SELECT COUNT(DISTINCT country) "Count of Countries"
FROM   publishers
```

COVAR_POP

The *COVAR_POP* function returns the population covariance of a set of dependent and independent variables.

SQL2003 Syntax

Call the function with two variables, one dependent and the other independent:

```
COVAR_POP( dependent, independent)
```

The function disregards any pair in which either the dependent variable, independent variable, or both are NULL. If no rows remain in the group after NULL elimination, then the result of the function is NULL.

Oracle

Oracle supports the SQL2003 syntax and implements the following analytic syntax:

```
COVAR_POP ( dependent, independent ) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section later in this chapter titled "ANSI SQL Window Functions."

DB2

In DB2, the function is named *CORRELATION*.

MySQL, PostgreSQL, and SQL Server

These platforms do not support any form of the *COVAR_POP* function.

Example

The following *COVAR_POP* example uses the data shown by the first *SELECT*:

```
SELECT * FROM test2;

         Y          X
---------- ----------
         1          3
         2          2
         3          1
```

```
SELECT COVAR_POP(y,x) FROM test2;

COVAR_POP(Y,X)
--------------
    -.66666667
```

COVAR_SAMP

The *COVAR_SAMP* function returns the sample covariance of a set of dependent and independent variables.

SQL2003 Syntax

Call the function with two variables, one dependent and the other independent:

```
COVAR_SAMP( dependent, independent )
```

The function disregards any pair in which either the dependent variable, independent variable, or both are NULL. The result of the function is NULL when none of the input pairs consist of two non-NULL values.

Oracle

Oracle supports the SQL2003 syntax and implements the following analytic syntax:

```
COVAR_SAMP ( dependent, independent ) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section later in this chapter titled "ANSI SQL Window Functions."

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not support any form of the *COVAR_SAMP* function.

Example

The following *COVAR_SAMP* example uses the data shown by the first *SELECT*:

```
SELECT * FROM test2;

         Y          X
---------- ----------
         1          3
         2          2
         3          1

SQL> SELECT COVAR_SAMP(y,x) FROM test2;

COVAR_SAMP(Y,X)
---------------
             -1
```

CUME_DIST

Computes the relative rank of a hypothetical row within a group of rows, where that relative rank is computed as follows:

```
(rows_preceding_hypothetical + rows_peered_with_hypothetical) / rows_in_
group
```

Bear in mind that the *rows_in_group* value includes the hypothetical row that you are proposing when you call the function.

SQL2003 Syntax

In the following syntax, items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

```
CUME_DIST(value_list) WITHIN GROUP (ORDER BY sort_list)

value_list ::= expression [,expression...]

sort_list ::= sort_item [,sort_item...]

sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle follows the SQL2003 syntax and implements the following analytic syntax:

```
CUME_DIST OVER ([partioning] ordering )
```

For an explanation of the *partioning* and *order* clauses, see the section later in this chapter titled "ANSI SQL Window Functions."

DB2, MySQL, PostgreSQl, and SQL Server

These platforms do not implement the *CUME_DIST* aggregate function.

Example

The following example determines the relative rank of the hypothetical new row (`num=4, odd=1`) within each group of rows from **test4**, where groups are distinguished by the values in the odd column:

```
SELECT * FROM test4;

       NUM        ODD
---------- ----------
         0          0
         1          1
         2          0
         3          1
         3          1
         4          0
         5          1

SELECT odd, CUME_DIST(4,1) WITHIN GROUP (ORDER BY num, odd)
FROM test4
GROUP BY odd;

       ODD CUME_DIST(4,1)WITHINGROUP(ORDERBYNUM,ODD)
---------- ----------------------------------------
         0                                        1
         1                                       .8
```

In group `odd=0`, the new row comes after the three rows: `(0,0)`, `(2,0)`, and `(4,0)`. It will peer with itself. The total number of rows in the group will be four, which includes the hypothetical row. The relative rank, therefore, is computed as follows:

```
(3 rows preceding + 1 peering) / (3 in group + 1 hypothetical)
= 4 / 4 = 1
```

In group `odd=1`, the new row follows the three rows `(1,1)`, `(3,1)`, and a duplicate `(3,1)`. Again, there is one peer, the hypothetical row itself. The number of rows in the group is five, which includes the hypothetical row. The relative rank is then:

```
(3 rows preceding + 1 peering) / (4 in group + 1 hypothetical)
= 4 / 5 = .8
```

DENSE_RANK

Computes a rank in a group for a hypothetical row that you supply. This is a *dense rank*. Rankings are never skipped, even when a group contains rows that rank identically.

SQL2003 Syntax

In the following syntax, items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

```
DENSE_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)

value_list ::= expression [,expression...]

sort_list ::= sort_item [,sort_item...]

sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle follows the SQL2003 syntax and implements the following analytic syntax:

```
DENSE_RANK() OVER ([partioning] ordering )
```

For an explanation of the *partioning* and *order* clauses, see the section later in this chapter titled "ANSI SQL Window Functions."

DB2, MySQL, PostgreSQl, and SQL Server

These platforms do not implement the *DENSE_RANK* aggregate function. However, DB2 does support *DENSE_RANK* as an analytic function. See the section later in this chapter titled "ANSI SQL Window Functions."

Example

The following example determines the dense rank of the hypothetical new row (`num=4, odd=1`) within each group of rows from **test4**, where groups are distinguished by the values in the odd column:

```
SELECT * FROM test4;

       NUM         ODD
---------- ----------
         0           0
         1           1
         2           0
         3           1
         3           1
```

```
         4          0
         5          1

SELECT odd, DENSE_RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
FROM test4
GROUP BY odd;

       ODD DENSE_RANK(4,1)WITHINGROUP(ORDERBYNUM,ODD)
---------- ----------------------------------------
         0                                        4
         1                                        3
```

In group `odd=0`, the new row comes after `(0,0)`, `(2,0)`, and `(4,0)`, and thus it is position 4. In group `odd=1`, the new row follows `(1,1)`, `(3,1)`, and a duplicate `(3,1)`. In that case, the duplicate occurrences of `(3,1)` both rank #2, so the new row is ranked #3. Compare this behavior with *RANK*, which gives a different result.

MIN and MAX

MIN(expression) and *MAX(expression)* find the minimum and maximum value of *expression* (string, datetime, or numeric) in a set of rows. *DISTINCT* or *ALL* may be used with these functions, but do not affect the result.

SQL2003 Syntax

```
MIN( [ALL | DISTINCT] expression )
MAX( [ALL | DISTINCT] expression )
```

PostgreSQL and SQL Server

These platforms support the SQL2003 syntax of *MIN* and *MAX*.

DB2 and Oracle

DB2 and Oracle support the ANSI syntax and implements the following analytic syntax:

```
MIN ({ALL|[DISTINCT] expression}) OVER (window_clause)
MAX ({ALL|[DISTINCT] expression}) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section later in this chapter titled "ANSI SQL Window Functions."

MySQL

MySQL supports the SQL2003 syntax of *MIN* and *MAX*. MySQL also supports the functions *LEAST( )* and *GREATEST( )*, providing the same capabilities.

Examples

The following query finds the best and worst sales for any title on record:

```
SELECT  MIN(ytd_sales), MAX(ytd_sales)
FROM    titles;
```

Aggregate functions are used often in the *HAVING* clause of queries with *GROUP BY*. The following query selects all categories (types) of books that have an average price for all books in the category higher than $15.00:

```
SELECT  type 'Category', AVG( price ) 'Average Price'
FROM    titles
GROUP BY type
HAVING AVG(price) > 15
```

PERCENT_RANK

Generates a relative rank for a hypothetical row by dividing that row's rank less 1 by the number of rows in the group.

SQL2003 Syntax

In the following syntax, items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

```
PERCENT_RANK(value_list) WITHIN GROUP (ORDER BY sort_list)

value_list ::= expression [,expression...]

sort_list ::= sort_item [,sort_item...]

sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle follows the SQL2003 syntax and implements the following syntax:

```
PERCENT_RANK() OVER ([partioning] ordering)
```

For an explanation of the *partioning* and *order* clauses, see the section later in this chapter titled "ANSI SQL Window Functions."

DB2, MySQL, PostgreSQl, and SQL Server

These platforms do not implement the *PERCENT_RANK* aggregate function.

Example

The following example determines the percentage rank of the hypothetical new row (`num=4, odd=1`) within each group of rows from **test4**, where groups are distinguished by the values in the odd column:

```
SELECT * FROM test4;

       NUM         ODD
---------- ----------
         0           0
         1           1
         2           0
         3           1
         3           1
         4           0
         5           1

SELECT odd, PERCENT_RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
FROM test4
GROUP BY odd;
```

```
       ODD PERCENT_RANK(4,1)WITHINGROUP(ORDERBYNUM,ODD)
---------- ---------------------------------------------
         0                                             1
         1                                           .75
```

In group `odd=0`, the new row comes after `(0,0)`, `(2,0)`, and `(4,0)`, and thus it is position 4. The rank computation is: `(4th rank - 1)/3 rows = 100%`. In group `odd=1`, the new row follows `(1,1)`, `(3,1)`, and a duplicate `(3,1)`, and is again ranked at #4. The rank computation for `odd=1` is: `(4th rank - 1)/4 rows = 3/4 = 75%`.

PERCENTILE_CONT

Generates an interpolated value corresponding to a percentile that you specify.

SQL2003 Syntax

In the following syntax, *percentile* is a number between zero and one:

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY sort_list)

sort_list ::= sort_item [,sort_item...]

sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle allows only one expression in the *ORDER BY* clause:

```
PERCENTILE_CONT(percentile) WITHIN GROUP (ORDER BY expression)
```

Oracle also allows some use of windowing syntax:

```
PERCENTILE_CONT (percentile)  WITHIN GROUP
(ORDER BY sort_list) OVER (partitioning)
```

See "ANSI SQL Window Functions" later in this chapter for a description of partitioning.

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement *PERCENTILE_CONT*.

Example

The following example groups the data in **test4** by the column named **odd**, and invokes *PERCENTILE_CONT* to return a 50th percentile value for each group:

```
SELECT * FROM test4;

       NUM         ODD
---------- ----------
         0           0
         1           1
         2           0
         3           1
         3           1
         4           0
         5           1

SELECT odd, PERCENTILE_CONT(0.50) WITHIN GROUP (ORDER BY NUM)
```

```
FROM test4
GROUP BY odd;

       ODD PERCENTILE_CONT(0.50)WITHINGROUP(ORDERBYNUM)
---------- ---------------------------------------------
         0                                           2
         1                                           3
```

PERCENTILE_DISC

Determines the value in a group with the smallest cumulative distribution greater than or equal to a percentile that you specify.

SQL2003 Syntax

In the following syntax, *percentile* is a number between zero and one:

```
PERCENTILE_DISC(percentile) WITHIN GROUP (ORDER BY sort_list)

sort_list ::= sort_item [,sort_item...]

sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle allows only one expression in the *ORDER BY* clause:

```
PERCENTILE_DISC(percentile) WITHIN GROUP (ORDER BY expression)
```

Oracle also allows some use of windowing syntax:

```
PERCENTILE_DISC (percentile)  WITHIN GROUP
(ORDER BY sort_list) OVER (partitioning)
```

See "ANSI SQL Window Functions" later in this chapter for a description of partitioning.

DB2, MySQL, PostgreSQL, and SQL Server

These platforms do not implement *PERCENTILE_DISC*.

Example

The following example is similar to that for *PERCENTILE_CONT*, except that it returns, for each group, the value closest, but not exceeding, the 60th percentile:

```
SELECT * FROM test4;

       NUM         ODD
---------- ----------
         0           0
         1           1
         2           0
         3           1
         3           1
         4           0
         5           1

SELECT odd, PERCENTILE_DISC(0.60) WITHIN GROUP (ORDER BY NUM)
```

```
FROM test4
GROUP BY odd;

PERCENTILE_CONT(0.50)WITHINGROUP(ORDERBYNUM)
--------------------------------------------
                                           2
                                           3
```

RANK

Computes a rank in a group for a hypothetical row that you supply. This is not a dense rank. If the group contains rows that rank identically, then it's possible for ranks to be skipped. If you want a dense rank, use the *DENSE_RANK* function.

SQL2003 Syntax

In the following syntax, items in the *value_list* correspond by position to items in the *sort_list*. Therefore, both lists must have the same number of expressions.

```
RANK(value_list) WITHIN GROUP (ORDER BY sort_list)

value_list ::= expression [,expression...]

sort_list ::= sort_item [,sort_item...]

sort_item ::= expression [ASC|DESC] [NULLS FIRST|NULLS LAST]
```

Oracle

Oracle follows the SQL2003 syntax and implements the following analytic syntax:

```
RANK() OVER ([partitioning] ordering)
```

For an explanation of the *partioning* and *order* clauses, see the section later in this chapter titled "ANSI SQL Window Functions."

DB2, MySQL, PostgreSQl, and SQL Server

These platforms do not implement the *RANK* aggregate function.

Example

The following example determines the rank of the hypothetical new row (`num=4, odd=1`) within each group of rows from **test4**, where groups are distinguished by the values in the **odd** column:

```
SELECT * FROM test4;

       NUM         ODD
---------- ----------
         0           0
         1           1
         2           0
         3           1
         3           1
         4           0
         5           1
```

```
SELECT odd, RANK(4,1) WITHIN GROUP (ORDER BY num, odd)
FROM test4
GROUP BY odd;

       ODD RANK(4,1)WITHINGROUP(ORDERBYNUM,ODD)
---------- -----------------------------------
         0                                   4
         1                                   4
```

In both cases, the rank of the hypothetical new row is 4. In group `odd=0`, the new row comes after: `(0,0)`, `(2,0)`, and `(4,0)`, and thus it is position 4. In group `odd=1`, the new row follows `(1,1)`, `(3,1)`, and a duplicate `(3,1)`. In that case, the new row is preceding by three rows, so it is ranked #4. Compare this behavior with *DENSE_RANK*.

The REGR Family of Functions

SQL2003 defines a family of functions, having names beginning with *REGR_*, that relate to different aspects of linear regression. The functions work in the context of a least-squares regression line.

SQL2003 Syntax

Following is the syntax and a brief description of each *REGR_* function:

REGR_AVGX(*dependent*, *independent*)
: Averages (as in *AVG(x)*) the *independent* variable values.

REGR_AVGY(*dependent*, *independent*)
: Averages (as in *AVG(y)*) the *dependent* variable values.

REGR_COUNT(*dependent*, *independent*)
: Counts the number of non-NULL number pairs.

REGR_INTERCEPT(*dependent*, *independent*)
: Computes the y-intercept of the regression line.

REGR_R2(*dependent*, *independent*)
: Computes the coefficient of determination.

REGR_SLOPE(*dependent*, *independent*)
: Computes the slope of the regression line.

REGR_SXX(*dependent*, *independent*)
: Sums the squares of the independent variable values.

REGR_SXY(*dependent*, *independent*)
: Sums the products of each pair of values.

REGR_SYY(*dependent*, *independent*)
: Sums the squares of the dependent variable values.

The *REGR_* functions only work on number pairs containing two non-NULL values. Any number pair with one or more NULL values will be ignored.

DB2 and Oracle

DB2 and Oracle support the SQL2003 syntax for all *REGR_* functions. In addition, DB2 allows the shortened name *REGR_ICPT* in place of *REGR_INTERCEPT*.

Oracle supports the following analytic syntax:

```
REGR_function ( dependent, independent ) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section later in this chapter titled "ANSI SQL Window Functions."

MySQL, PostgreSQL, and SQL Server

These platforms do not implement the *REGR* family of functions.

Example

The following *REGEXP_COUNT* example demonstrates that any pair with one or more NULL values is ignored. The table **test3** contains three non-NULL number pairs, and three other pairs having at least one NULL:

```
SQL> SELECT * FROM test3;

         Y          X
---------- ----------
         1          3
         2          2
         3          1
         4       NULL
      NULL          4
      NULL       NULL
```

The *REGR_COUNT* function ignores the pairs having NULLs, counting only those pairs with non-NULL values:

```
SELECT REGR_COUNT(y,x) FROM test3;

REGR_COUNT(Y,X)
---------------
              3
```

Likewise, all other *REGR_* functions filter out any pairs having NULL values before performing their respective computations.

STDDEV_POP

Use *STDDEV_POP* to find the *population standard deviation* within a group of numeric values.

SQL2003 Syntax

```
STDDEV_POP( numeric_expression )
```

DB2 and MySQL

Use the *STDDEV* function. In DB2 and MySQL, *STDDEV* returns the population standard deviation.

PostgreSQL

This platform does not provide a function to compute population standard deviation.

Oracle

Oracle supports the standard syntax and the following analytic syntax:

```
STDDEV_POP (numeric_expression) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section later in this chapter titled "ANSI SQL Window Functions."

SQL Server

Use the *STDEVP* function.

Example

The following example computes the population standard deviation for the values 1, 2, and 3:

```
SELECT * FROM test;

          X
----------
          1
          2
          3

SELECT STDDEV_POP(x) FROM test;

STDDEV_POP(X)
-------------
   .816496581
```

STDDEV_SAMP

Use *STDDEV_SAMP* to find the *sample standard deviation* within a group of numeric values.

SQL2003 Syntax

```
STDDEV_SAMP( numeric_expression )
```

Oracle

Oracle supports the standard syntax. Oracle also provides the *STDDEV* function, which operates similar to *STDDEV_SAMP* except that it returns zero as not NULL when there is only one value in the set.

Oracle also supports analytic syntax:

```
STDDEV_SAMP (numeric_expression) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section later in this chapter titled "ANSI SQL Window Functions."

DB2

This platform does not provide a function to compute sample standard deviation.

MySQL

MySQL does not provide a function to compute sample standard deviation. MySQL does provide a function named *STDDEV*, but it returns the *population* standard deviation.

PostgreSQL

Use *STDDEV*.

SQL Server

Use *STDEV* (with only one *D*!).

Example

The following example computes the sample standard deviation for the values 1, 2, and 3:

```
SELECT * FROM test;

         X
----------
         1
         2
         3

SELECT STDDEV_SAMP(x) FROM test;

STDDEV_SAMP(X)
--------------
             1
```

VAR_POP

Use *VAR_POP* to compute the population variance of a set of values.

SQL2003 Syntax

```
VAR_POP( numeric_expression )
```

DB2 and PostgreSQL

These platforms do not provide a function to compute population variance.

MySQL

Use the *VARIANCE* function, which in MySQL returns the population variance.

Oracle

Oracle supports the standard syntax and the following analytic syntax:

```
VAR_POP (numeric_expression) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section later in this chapter titled "ANSI SQL Window Functions."

SQL Server

Use the *VARP* function.

Example

The following example computes the population variance for the values 1, 2, and 3:

```
SELECT * FROM test;

         X
----------
         1
         2
         3

SELECT VAR_POP(x) FROM test;

VAR_POP(X)
----------
.666666667
```

VAR_SAMP

Use *VAR_SAMP* to compute the sample variance of a set of values.

SQL2003 Syntax

```
VAR_SAMP( numeric_expression )
```

DB2 and PostgreSQL

Use *VARIANCE*(*numeric_expression*) to compute sample variance.

MySQL

MySQL provides no function for computing sample variance. There is the *VARIANCE* function, but in MySQL that function returns the *population* variance.

Oracle

Oracle supports the standard syntax. You may also use the *VARIANCE* function, which differs from *VAR_SAMP* by returning zero (and not NULL) for sets having only a single value.

Oracle also supports analytic syntax:

```
VAR_SAMP (numeric_expression) OVER (window_clause)
```

For an explanation of the *window_clause*, see the section later in this chapter titled "ANSI SQL Window Functions."

SQL Server

Use the *VAR* function.

Example

The following example computes the sample variance for the values 1, 2, and 3:

```
SELECT * FROM test;

         X
----------
         1
         2
         3
```

```
SELECT VAR_SAMP(x) FROM test;

VAR_SAMP(X)
-----------
          1
```

ANSI SQL Window Functions

SQL2003 allows for a *window_clause* in aggregate function calls, the addition of which makes those functions into window functions. Both Oracle and DB2 support this window function syntax. This section describes how to use the *window_clause* within Oracle and DB2.

Oracle tends to refer to window functions as *analytic functions*.

Window, or analytic, functions are similar to standard aggregate functions in that they operate on multiple rows, or groups of rows, within the result set returned from a query. However, the groups of rows that a window function operates on are defined not by a GROUP BY clause, but by partitioning and windowing clauses. Furthermore, the order within these groups is defined by an ordering clause, but that order only affects function evaluation, and has no effect on the order in which rows are returned by the query.

Window functions are the last items in a query to be evaluated except for the *ORDER BY* clause. Because of this late evaluation, window functions *cannot* be used within the *WHERE*, *GROUP BY*, or *HAVING* clauses.

SQL2003's Window Syntax

SQL2003 specifies the following syntax for window functions:

```
FUNCTION_NAME(expr) OVER {window_name|(window_specification)}

window_specification ::= [window_name][partitioning][ordering][framing]

partitioning ::= PARTITION BY value [, value...] [COLLATE collation_name]

ordering ::= ORDER [SIBLINGS] BY rule [, rule...]

rule ::= {value|position|alias} [ASC|DESC] [NULLS {FIRST|LAST}]

framing ::= {ROWS|RANGE} {start|between} [exclusion]

start ::= {UNBOUNDED PRECEDING|unsigned-integer PRECEDING|CURRENT ROW}

between ::= BETWEEN bound AND bound

bound ::= {start|UNBOUNDED FOLLOWING|unsigned-integer FOLLOWING}
```

```
exclusion ::= {EXCLUDE CURRENT ROW|EXCLUDE GROUP
              |EXCLUDE TIES|EXCLUDE NO OTHERS}
```

Oracle's Window Syntax

Oracle's window function syntax is as follows:

```
FUNCTION_NAME(expr) OVER (window_clause)

window_clause ::= [partitioning] [ordering [framing]]

partitioning ::= PARTITION BY value [, value...]

ordering ::= ORDER [SIBLINGS] BY rule [, rule...]

rule ::= {value|position|alias} [ASC|DESC]
         [NULLS {FIRST|LAST}]

framing ::= {ROWS|RANGE} {not_range|begin AND end}

not_range ::= {UNBOUNDED PRECEDING
              |CURRENT ROW|
              |value PRECEDING}

begin ::= {UNBOUNDED PRECEDING
          |CURRENT ROW|
          |value {PRECEDING|FOLLOWING}}

end ::= {UNBOUNDED FOLLOWING
        |CURRENT ROW|
        |value {PRECEDING|FOLLOWING}}
```

DB2's Window Syntax

DB2's syntax is similar to Oracle's. For OLAP, ranking, and numbering functions, DB2 allows the following syntax:

```
FUNCTION_NAME(expr) OVER (window_clause)

window_clause ::= [partitioning] [ordering]

partitioning ::= PARTITION BY (value [, value...])

ordering ::= {ORDER BY rule [, rule...] | ORDER OF table_name}

rule ::= {value|position|alias} [ASC|DESC [NULLS {FIRST|LAST}]]
```

When aggregate functions (e.g. *AVG*) are used as window functions, DB2 allows the addition of a *framing* clause:

```
FUNCTION_NAME(expr) OVER (window_clause)

window_clause ::= [partitioning] [ordering [framing]] [all|framing]

all ::= RANGE UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
```

```
partitioning ::= PARTITION BY (value [, value...])

ordering ::= {ORDER BY rule [, rule...] | ORDER OF table_name}

rule ::= {value|position|alias} [ASC|DESC [NULLS {FIRST|LAST}]]

framing ::= {ROWS|RANGE} {group_start|group_between|group_end}

group_start ::= {UNBOUNDED PRECEDING|unsigned-integer PRECEDING
                |CURRENT ROW}

group_between ::= BETWEEN {UNBOUNDED PRECEDING|unsigned_integer PRECEDING
                          |unsigned_integer FOLLOWING|CURRENT ROW}
                  AND {UNBOUNDED FOLLOWING|unsigned_integer PRECEDING
                          |unsigned_integer FOLLOWING|CURRENT ROW}

group_end ::= UNBOUNDED FOLLOWING|unsigned-integer FOLLOWING}
```

Partitioning

Partitioning the rows operated on by the partition clause is similar to using the *GROUP BY* expression on a standard *SELECT* statement. The partitioning clause takes a list of expressions that will be used to divide the result set into groups. We'll use the following table as the basis for some examples:

```
SELECT * FROM odd_nums;

       NUM         ODD
---------- ----------
         0           0
         1           1
         2           0
         3           1
```

The following results illustrate the effects of partitioning by **ODD**. The sum of the even numbers is 2 (0+2), and the sum of the odd numbers is 4 (1+3). The second column of the result set reports the sum of all values in the partition to which *that row* belongs. Yet all the detail rows are returned. The query provides summary results in the context of detail rows:

```
SELECT NUM, SUM(NUM) OVER (PARTITION BY ODD) S
FROM ODD_NUMS;

NUM            S
---------      ----------
0              2
2              2
1              4
3              4
```

Not using a partitioning clause at all will sum all of the numbers in the **NUM** column for each row returned by the query. In effect, the entire result set is treated as a single, large partition:

```
SELECT NUM, SUM(NUM) OVER () S FROM ODD_NUMS;

NUM            S
---------      ----------
0              6
1              6
2              6
3              6
```

Ordering

You specify the order of the rows on which an analytic function operates using the *ordering* clause. However, this analytic ordering clause does not define the result set ordering. To define the overall result set ordering, you must use the query's *ORDER BY* clause. The following use of Oracle's *FIRST_VALUE* function illustrates the effect of different orderings of the partitions:

```
SELECT NUM,
       SUM(NUM) OVER (PARTITION BY ODD) S,
       FIRST_VALUE(NUM) OVER (PARTITION BY ODD ORDER BY NUM ASC) first_asc,
       FIRST_VALUE(NUM) OVER (PARTITION BY ODD ORDER BY NUM DESC) first_desc
FROM ODD_NUMS;

       NUM          S  FIRST_ASC FIRST_DESC
---------- ---------- ---------- ----------
         0          2          0          2
         2          2          0          2
         1          4          1          3
         3          4          1          3
```

As you can see, the *ORDER BY* clauses in the window function invocations affect the ordering of the rows in the respective partitions when those functions are evaluated. Thus, *ORDER BY NUM ASC* orders partitions in ascending order, resulting in `0` for the first value in the even-number partition and `1` for the first value in the odd-number partition. *ORDER BY NUM DESC* has the opposite effect.

The preceding query also illustrates an important point: using window functions, you can summarize and order many different ways in the same query.

Grouping or Windowing

Many analytic functions also allow you to specify a virtual, moving window surrounding a row within a partition. You do this using the *framing* clause. Such moving windows are useful for running calculations such as a running total.

The following, Oracle-based example uses the *framing* clause on the analytic variant of *SUM* to calculate a running sum of the values in the first column. No partitioning clause is used, so each invocation of *SUM* operates over the entire result set. However, the *ORDER BY* clause sorts the rows for *SUM* in ascending order of **NUM**'s value, and the *BETWEEN* clause (which is the windowing clause) causes each invocation of *SUM* to include values for **NUM** only up through the

current row. Each successive invocation of *SUM* includes yet another value for **NUM**, in order, from the lowest value of **NUM** to the greatest:

```
SELECT NUM, SUM(NUM) OVER (ORDER BY NUM ROWS
    BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) S FROM ODD_NUMS;
NUM            S
---------      ----------
0              0
1              1
2              3
3              6
```

This example's a bit too easy, as the order of the final result set happens to match the order of the running total. That doesn't need to be the case. The following example generates the same results, but in a different order. You can see that the running total values are appropriate for each value of **NUM**, but the rows are presented in a different order than before. The result set ordering is completely independent of the ordering used for window function calculations:

```
SELECT NUM, SUM(NUM) OVER (ORDER BY NUM ROWS
    BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) S FROM ODD_NUMS
ORDER BY NUM DESC;

       NUM          S
---------- ----------
         3          6
         2          3
         1          1
         0          0
```

List of Window Functions

SQL2003 specifies that any aggregate function may also be used as a window function. Both Oracle and DB2 largely follow the standard in that respect, so you'll find that you can take just about any aggregate function (certainly the standard ones) and apply to it the window function syntax described in the preceding sections.

In addition to the aggregate functions, SQL2003 defines the window functions described in the following sections. Only Oracle and DB2 currently implement these functions. All examples use the following table and data, which is a variation on the **ODD_NUMS** table used earlier to illustrate the concepts of partitioning, ordering, and grouping:

```
SELECT * FROM test4;

       NUM        ODD
---------- ----------
         0          0
         1          1
         2          0
         3          1
         3          1
         4          0
         5          1
```

Platform-specific window functions for Oracle (there are none for DB2) are included in the lists found under "Platform-Specific Extensions" later in this chapter.

CUME_DIST( )

Calculates the cumulative distribution, or relative rank, of the current row to other rows in the same partition. The calculation for a given row is as follows:

```
number of peer or preceding rows / number of rows in partition
```

Because the result for a given row depends on the number of rows preceding that row in the same partition, it's important to always specify an *ORDER BY* clause when invoking this function.

SQL2003 Syntax

```
CUME_DIST( ) OVER {window_name|(window_specification)}
```

DB2

DB2 does not support the *CUME_DIST( )* window function.

Oracle

Oracle does not allow the framing portion of the windowing syntax. Oracle requires the *ordering* clause:

```
CUME_DIST( ) OVER ([partitioning] ordering)
```

Example

The following Oracle-based example uses *CUME_DIST( )* to generate a relative rank for each row, ordering by **NUM**, after partitioning the data by **ODD**:

```
SELECT NUM, ODD, CUME_DIST( ) OVER
      (PARTITION BY ODD ORDER BY NUM) cumedist
FROM test4;

       NUM        ODD   CUMEDIST
---------- ---------- ----------
         0          0 .333333333
         2          0 .666666667
         4          0          1
         1          1        .25
         3          1        .75
         3          1        .75
         5          1          1
```

Following is an explanation of the calculation behind the rank for the row in which **NUM**=0:

1. Because of the *ORDER BY* clause, the rows in the partition are ordered as follows:

   NUM=0
   NUM=2
   NUM=4

2. There are no rows preceding **NUM**=0.

3. There is one row that is a peer of NUM=0, and that is the **NUM**=0 row itself. Thus, the divisor is 1.
4. There are three rows in the partition as a whole, making the dividend 3.
5. The result of 1/3 is.33 repeating, as shown in the example output.

DENSE_RANK( )

Assigns a rank to each row in a partition, which should be ordered in some manner. The rank for a given row is computed by counting the number of rows preceding the row in question, and then adding 1 to the result. Rows with duplicate *ORDER BY* values will rank the same. Unlike the case with *RANK()*, gaps in rank numbers will not result from two rows sharing the same rank.

SQL2003 Syntax

```
DENSE_RANK( ) OVER {window_name|(window_specification)}
```

DB2

DB2 requires the *ordering* clause and does not allow the *framing* clause:

```
DENSE_RANK( ) OVER ([partitioning] ordering)
```

Oracle

Oracle also requires the *ordering* clause and does not allow the *framing* clause:

```
DENSE_RANK( ) OVER ([partitioning] ordering)
```

Example

Compare the results from the following Oracle-based example to those shown in the section on the *RANK()* function:

```
SELECT NUM, DENSE_RANK( ) OVER (ORDER BY NUM) rank
FROM test4;

       NUM       RANK
---------- ----------
         0          1
         1          2
         2          3
         3          4
         3          4
         4          5
         5          6
```

The two rows where NUM=3 are both ranked at #3. The next higher row is ranked at #4. Rank numbers are not skipped, hence the term "dense."

RANK( )

Assigns a rank to each row in a partition, which should be ordered in some manner. The rank for a given row is computed by counting the number of rows preceding the row in question, and then adding 1 to the result. Rows with duplicate *ORDER BY* values will rank the same, and will lead to subsequent gaps in rank numbers.

SQL2003 Syntax

```
RANK( ) OVER {window_name|(window_specification)}
```

DB2

DB2 requires the *ordering* clause and does not allow the *framing* clause:

```
RANK( ) OVER ([partitioning] ordering)
```

Oracle

Oracle also requires the *ordering* clause and does not allow the *framing* clause:

```
RANK( ) OVER ([partitioning] ordering)
```

Example

The following Oracle-based example uses the **NUM** column to rank the rows in the **test4** table:

```
SELECT NUM, RANK( ) OVER (ORDER BY NUM) rank
FROM test4;

       NUM       RANK
---------- ----------
         0          1
         1          2
         2          3
         3          4
         3          4
         4          6
         5          7
```

Because both rows where **NUM**=3 rank the same at #4, the next higher row will be ranked at #6. The #5 rank is skipped.

PERCENT_RANK

Computes the relative rank of a row by dividing that row's rank less 1 by the number of rows in the partition, also less 1:

```
(rank - 1) / (rows - 1)
```

Compare this calculation to that used for *CUME_DIST*.

SQL2003 Syntax

```
PERCENT_RANK( ) OVER ({window_name|(window_specification)}
```

DB2

DB2 does not support the *PERCENT_RANK()* window function.

Oracle

Oracle requires the *ordering* clause and does not allow the *framing* clause:

```
PERCENT_RANK( ) OVER ([partitioning] ordering)
```

Example

The following, Oracle-based example assigns a relative rank to values of **NUM**, partitioning the data on the **ODD** column:

```
SELECT NUM, ODD, PERCENT_RANK( ) OVER
      (PARTITION BY ODD ORDER BY NUM) cumedist
FROM test4;

       NUM        ODD   CUMEDIST
---------- ---------- ----------
         0          0          0
         2          0         .5
         4          0          1
         1          1          0
         3          1 .333333333
         3          1 .333333333
         5          1          1
```

Following is an explanation of the calculation behind the rank for the row in which **NUM**=2:

1. Row **NUM**=2 is the second row in its partition; thus, it ranks #2.
2. Subtract 1 from 2 to get a divisor of 1.
3. The dividend is the total number of rows in the partition, or 3.
4. Subtract 1 from 3 to get a dividend of 2.
5. The result of 1/3 is.33 repeating, as shown in the example.

ROW_NUMBER

Assigns a unique number to each row in a partition.

SQL2003 Syntax

```
ROW_NUMBER( ) OVER ({window_name|(window_specification)}
```

DB2

DB2 does not allow the *framing* clause, and it makes the *ordering* clause optional:

```
ROW_NUMBER( ) OVER ([partitioning] [ordering])
```

Oracle

Oracle requires the *ordering* clause and does not allow the *framing* clause:

```
ROW_NUMBER( ) OVER ([partitioning] ordering)
```

Example

```
SELECT NUM, ODD, ROW_NUMBER( ) OVER
      (PARTITION BY ODD ORDER BY NUM) cumedist
FROM test4;

       NUM        ODD   CUMEDIST
---------- ---------- ----------
         0          0          1
         2          0          2
```