**Analytic Functions**

Introduced in Oracle 8i, analytic functions, also known as windowing functions, allow developers to perform tasks in SQL that were previously confined to procedural languages.

* [Introduction](http://www.oracle-base.com/articles/misc/analytic-functions.php#introduction)
* [Analytic Function Syntax](http://www.oracle-base.com/articles/misc/analytic-functions.php#syntax)
  + [query\_partition\_clause](http://www.oracle-base.com/articles/misc/analytic-functions.php#query_partition_clause)
  + [order\_by\_clause](http://www.oracle-base.com/articles/misc/analytic-functions.php#order_by_clause)
  + [windowing\_clause](http://www.oracle-base.com/articles/misc/analytic-functions.php#windowing_clause)
* [Using Analytic Functions](http://www.oracle-base.com/articles/misc/analytic-functions.php#using_analytic_functions)

Related articles.

* [RANK, DENSE\_RANK, FIRST and LAST Analytic Functions](http://www.oracle-base.com/articles/misc/rank-dense-rank-first-last-analytic-functions.php)
* [FIRST\_VALUE and LAST\_VALUE Analytic Functions](http://www.oracle-base.com/articles/misc/first-value-and-last-value-analytic-functions.php)
* [LAG and LEAD Analytic Functions](http://www.oracle-base.com/articles/misc/lag-lead-analytic-functions.php)
* [LISTAGG Analystic Function in 11g Release 2](http://www.oracle-base.com/articles/misc/string-aggregation-techniques.php#listagg)
* [Top-N Queries](http://www.oracle-base.com/articles/misc/top-n-queries.php)

**Introduction**

Probably the easiest way to understand analytic functions is to start by looking at aggregate functions. An aggregate function, as the name suggests, aggregates data from several rows into a single result row. For example, we might use the AVG aggregate function to give us an average of all the employee salaries in the EMP table.

SELECT AVG(sal)

FROM emp;

AVG(SAL)

----------

2073.21429

SQL>

The GROUP BY clause allows us to apply aggregate functions to subsets of rows. For example, we might want to display the average salary for each department.

SELECT deptno, AVG(sal)

FROM emp

GROUP BY deptno

ORDER BY deptno;

DEPTNO AVG(SAL)

---------- ----------

10 2916.66667

20 2175

30 1566.66667

SQL>

In both cases, the aggregate function reduces the number of rows returned by the query.

**Analytic functions** also operate on subsets of rows, similar to aggregate functions in GROUP BY queries, but they do not reduce the number of rows returned by the query. For example, the following query reports the salary for each employee, along with the average salary of the employees within the department.

SELECT empno, deptno, sal,

AVG(sal) OVER (PARTITION BY deptno) AS avg\_dept\_sal

FROM emp;

EMPNO DEPTNO SAL AVG\_DEPT\_SAL

---------- ---------- ---------- ------------

7782 10 2450 2916.66667

7839 10 5000 2916.66667

7934 10 1300 2916.66667

7566 20 2975 2175

7902 20 3000 2175

7876 20 1100 2175

7369 20 800 2175

7788 20 3000 2175

7521 30 1250 1566.66667

7844 30 1500 1566.66667

7499 30 1600 1566.66667

7900 30 950 1566.66667

7698 30 2850 1566.66667

7654 30 1250 1566.66667

SQL>

This time AVG is an analytic function, operating on the group of rows defined by the contents of the OVER clause. This group of rows is known as a window, which is why analytic functions are sometimes referred to as window[ing] functions. Notice how the AVG function is still reporting the departmental average, like it did in the GROUP BY query, but the result is present in each row, rather than reducing the total number of rows returned. This is because **analytic functions are performed on a result set after all join, WHERE, GROUP BY and HAVING clauses are complete, but before the final ORDER BY operation is performed.**

**Analytic Function Syntax**

There are some variations in the syntax of the individual analytic functions, but the basic syntax for an analytic function is as follows.

analytic\_function([ arguments ]) OVER (analytic\_clause)

avg(sal) over (partition by deptno order by depnto)

The analytic\_clause breaks down into the following optional elements.

[ query\_partition\_clause ] [ order\_by\_clause [ windowing\_clause ] ]

The sub-elements of the analytic\_clause each have their own syntax diagrams, shown [here](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions004.htm#SQLRF06174). Rather than repeat the syntax diagrams, the following sections describe what each section of the analytic\_clause is used for.

**query\_partition\_clause**

The query\_partition\_clause divides the result set into partitions, or groups, of data. The operation of the analytic function is restricted to the boundary imposed by these partitions, similar to the way a GROUP BY clause affects the action of an aggregate function. If the query\_partition\_clause is omitted, the whole result set is treated as a single partition. The following query uses an empty OVER clause, so the average presented is based on all the rows of the result set.

SELECT empno, deptno, sal,

AVG(sal) OVER () AS avg\_sal

FROM emp;

EMPNO DEPTNO SAL AVG\_SAL

---------- ---------- ---------- ----------

7369 20 800 2073.21429

7499 30 1600 2073.21429

7521 30 1250 2073.21429

7566 20 2975 2073.21429

7654 30 1250 2073.21429

7698 30 2850 2073.21429

7782 10 2450 2073.21429

7788 20 3000 2073.21429

7839 10 5000 2073.21429

7844 30 1500 2073.21429

7876 20 1100 2073.21429

7900 30 950 2073.21429

7902 20 3000 2073.21429

7934 10 1300 2073.21429

SQL>

If we change the OVER clause to include a query\_partition\_clause based on the department, the averages presented are specifically for the department the employee belongs too.

SELECT empno, deptno, sal,

AVG(sal) OVER (PARTITION BY deptno) AS avg\_dept\_sal

FROM emp;

EMPNO DEPTNO SAL AVG\_DEPT\_SAL

---------- ---------- ---------- ------------

7782 10 2450 2916.66667

7839 10 5000 2916.66667

7934 10 1300 2916.66667

7566 20 2975 2175

7902 20 3000 2175

7876 20 1100 2175

7369 20 800 2175

7788 20 3000 2175

7521 30 1250 1566.66667

7844 30 1500 1566.66667

7499 30 1600 1566.66667

7900 30 950 1566.66667

7698 30 2850 1566.66667

7654 30 1250 1566.66667

SQL>

**order\_by\_clause**

The order\_by\_clause is used to order rows, or siblings, within a partition. So if an analytic function is sensitive to the order of the siblings in a partition you should include an order\_by\_clause. The following query uses the FIRST\_VALUE function to return the first salary reported in each department. Notice we have partitioned the result set by the department, but there is no order\_by\_clause.

SELECT empno, deptno, sal,

FIRST\_VALUE(sal IGNORE NULLS) OVER (PARTITION BY deptno) AS first\_sal\_in\_dept

FROM emp;

EMPNO DEPTNO SAL FIRST\_SAL\_IN\_DEPT

---------- ---------- ---------- -----------------

7782 10 2450 2450

7839 10 5000 2450

7934 10 1300 2450

7566 20 2975 2975

7902 20 3000 2975

7876 20 1100 2975

7369 20 800 2975

7788 20 3000 2975

7521 30 1250 1250

7844 30 1500 1250

7499 30 1600 1250

7900 30 950 1250

7698 30 2850 1250

7654 30 1250 1250

SQL>

Now compare the values of the FIRST\_SAL\_IN\_DEPT column when we include an order\_by\_clause to order the siblings by ascending salary.

SELECT empno, deptno, sal,

FIRST\_VALUE(sal IGNORE NULLS) OVER (PARTITION BY deptno ORDER BY sal ASC NULLS LAST) AS first\_val\_in\_dept

FROM emp;

EMPNO DEPTNO SAL FIRST\_VAL\_IN\_DEPT

---------- ---------- ---------- -----------------

7934 10 1300 1300

7782 10 2450 1300

7839 10 5000 1300

7369 20 800 800

7876 20 1100 800

7566 20 2975 800

7788 20 3000 800

7902 20 3000 800

7900 30 950 950

7654 30 1250 950

7521 30 1250 950

7844 30 1500 950

7499 30 1600 950

7698 30 2850 950

SQL>

**In this case the "ASC NULLS LAST" keywords are unnecessary as ASC is the default for an order\_by\_clause and NULLS LAST is the default for ASC orders. When ordering by DESC, the default is NULLS FIRST**.

It is important to understand how the order\_by\_clause affects display order. The order\_by\_clause is guaranteed to affect the order of the rows as they are processed by the analytic function, but it may not always affect the display order. As a result, you must always use a conventional ORDER BY clause in the query if display order is important. Do not rely on any implicit ordering done by the analytic function. Remember, the conventional ORDER BY clause is performed after the analytic processing, so it will always take precedence.

**windowing\_clause**

We have seen previously the query\_partition\_clause controls the window, or group of rows, the analytic operates on. The **windowing\_clause** gives some analytic functions a further degree of control over this window **within the current partition**. The windowing\_clause is an extension of the order\_by\_clause and as such, it can only be used if an order\_by\_clause is present. The windowing\_clause has two basic forms.

RANGE BETWEEN start\_point AND end\_point

ROWS BETWEEN start\_point AND end\_point

Possible values for "start\_point" and "end\_point" are:

* UNBOUNDED PRECEDING : The window **starts at the first row of the partition**. Only available for start points.
* UNBOUNDED FOLLOWING : The window **ends at the last row of the partition**. Only available for end points.
* CURRENT ROW : The window **starts** or **ends** at the current row. Can be used as start or end point.
* value\_expr PRECEDING : A physical or logical offset **before** the current row using a constant or expression that evaluates to a positive numerical value. When used with RANGE, it can also be an interval literal if the order\_by\_clause uses a DATE column.
* value\_expr FOLLOWING : As above, but an offset **after** the current row.

The documentation states the start point must always be before the end point, but this is not true, as demonstrated by this rather silly, but valid, query.

SELECT empno, deptno, sal,

AVG(sal) OVER (PARTITION BY deptno ORDER BY sal ROWS BETWEEN 0 PRECEDING AND 0 PRECEDING) AS avg\_of\_current\_sal

FROM emp;

EMPNO DEPTNO SAL AVG\_OF\_CURRENT\_SAL

---------- ---------- ---------- ------------------

7934 10 1300 1300

7782 10 2450 2450

7839 10 5000 5000

7369 20 800 800

7876 20 1100 1100

7566 20 2975 2975

7788 20 3000 3000

7902 20 3000 3000

7900 30 950 950

7654 30 1250 1250

7521 30 1250 1250

7844 30 1500 1500

7499 30 1600 1600

7698 30 2850 2850

SQL>

**In fact, the start point must be before or equal to the end point.**

For analytic functions that support the windowing\_clause, the default action is "RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW". The following query is similar to one used previously to report the employee salary and average department salary, but now we have included an order\_by\_clause so we also get the default windowing\_clause. Notice how the average salary is now calculated using only the employees from the same department up to and including the current row.

SELECT empno, deptno, sal,

AVG(sal) OVER (PARTITION BY deptno ORDER BY sal) AS avg\_dept\_sal\_sofar

FROM emp;

EMPNO DEPTNO SAL AVG\_DEPT\_SAL\_SOFAR

---------- ---------- ---------- ------------------

7934 10 1300 1300

7782 10 2450 1875

7839 10 5000 2916.66667

7369 20 800 800

7876 20 1100 950

7566 20 2975 1625

7788 20 3000 2175

7902 20 3000 2175

7900 30 950 950

7654 30 1250 1150

7521 30 1250 1150

7844 30 1500 1237.5

7499 30 1600 1310

7698 30 2850 1566.66667

SQL>

The following query shows one method for accessing data from previous and following rows within the current row using the windowing\_clause. This can also be accomplished with [LAG and LEAD](http://www.oracle-base.com/articles/misc/lag-lead-analytic-functions.php).

SELECT empno, deptno, sal,

FIRST\_VALUE(sal) OVER (ORDER BY sal ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) AS previous\_sal,

LAST\_VALUE(sal) OVER (ORDER BY sal ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING) AS next\_sal

FROM emp;

EMPNO DEPTNO SAL PREVIOUS\_SAL NEXT\_SAL

---------- ---------- ---------- ------------ ----------

7369 20 800 800 950

7900 30 950 800 1100

7876 20 1100 950 1250

7521 30 1250 1100 1250

7654 30 1250 1250 1300

7934 10 1300 1250 1500

7844 30 1500 1300 1600

7499 30 1600 1500 2450

7782 10 2450 1600 2850

7698 30 2850 2450 2975

7566 20 2975 2850 3000

7788 20 3000 2975 3000

7902 20 3000 3000 5000

7839 10 5000 3000 5000

SQL>

**Using Analytic Functions**

The best way to understand what analytic functions are capable of is to play around with them. This article contains links to other articles I've written about specific analytic functions and the following documentation links list all analytic functions available in Oracle 11g Release 2. The "\*" indicates that these functions allow for the full analytic syntax, including the windowing\_clause.

* [**AVG \***](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions018.htm#i82074)
* [CORR](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions035.htm#i82637) \*
* [COUNT](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions039.htm#i82697) \*
* [COVAR\_POP](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions040.htm#i1008854) \*
* [COVAR\_SAMP](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions041.htm#i82820) \*
* [CUME\_DIST](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions043.htm#i82886)
* [**DENSE\_RANK**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions052.htm#i1064409)
* [**FIRST**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions065.htm#i1000901)
* [**FIRST\_VALUE**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions066.htm#i83212) **\***
* [**LAG**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions082.htm#i1327527)
* [**LAST**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions083.htm#i1000905)
* [**LAST\_VALUE**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions085.htm#i83648) **\***
* [**LEAD**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions086.htm#i83834)
* [**LISTAGG**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions089.htm#CJABDFBD)
* [**MAX**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions098.htm#i89072) **\***
* [**MIN**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions100.htm#i1280029) **\***
* [**NTH\_VALUE**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions114.htm#CJAFEJBE) **\***
* [**NTILE**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions115.htm#i85619)
* [PERCENT\_RANK](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions126.htm#i1043951)
* [PERCENTILE\_CONT](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions127.htm#i1000909)
* [PERCENTILE\_DISC](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions128.htm#i1000913)
* [**RANK**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions141.htm#i1269223)
* [RATIO\_TO\_REPORT](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions142.htm#i85800)
* [REGR\_ (Linear Regression) Functions](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions151.htm#i85922) \*
* [**ROW\_NUMBER**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions156.htm#i86310)
* [STDDEV](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions178.htm#i89108) \*
* [STDDEV\_POP](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions179.htm#i86639) \*
* [STDDEV\_SAMP](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions180.htm#i86697) \*
* [**SUM**](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions182.htm#i89126) **\***
* [VAR\_POP](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions230.htm#i87119) \*
* [VAR\_SAMP](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions231.htm#i87169) \*
* [VARIANCE](http://docs.oracle.com/cd/E11882_01/server.112/e17118/functions232.htm#i89144) \*

**RANK, DENSE\_RANK, FIRST and LAST Analytic Functions**

This article gives and overview of the RANK, DENSE\_RANK, FIRST and LAST analytic functions.

* [RANK](http://www.oracle-base.com/articles/misc/rank-dense-rank-first-last-analytic-functions.php#rank)
* [DENSE\_RANK](http://www.oracle-base.com/articles/misc/rank-dense-rank-first-last-analytic-functions.php#dense_rank)
* [FIRST and LAST](http://www.oracle-base.com/articles/misc/rank-dense-rank-first-last-analytic-functions.php#first_and_last)

**RANK**

Let's assume we want to assign a sequential order, or rank, to people within a department based on salary, we might use the RANK function like.

SELECT empno,

deptno,

sal,

RANK() OVER (PARTITION BY deptno ORDER BY sal) "rank"

FROM emp;

EMPNO DEPTNO SAL rank

---------- ---------- ---------- ----------

7934 10 1300 1

7782 10 2450 2

7839 10 5000 3

7369 20 800 1

7876 20 1100 2

7566 20 2975 3

7788 20 3000 4

7902 20 3000 4

7900 30 950 1

**7654 30 1250 2**

**7521 30 1250 2**

**7844 30 1500 4**

7499 30 1600 5

7698 30 2850 6

SQL>

What we see here is where two people have the same salary they are assigned the same rank. When multiple rows share the same rank the next rank in the sequence is not consecutive.

**DENSE\_RANK**

The DENSE\_RANK function acts like the RANK function except that it assigns consecutive ranks.

SELECT empno,

deptno,

sal,

DENSE\_RANK() OVER (PARTITION BY deptno ORDER BY sal) "rank"

FROM emp;

EMPNO DEPTNO SAL rank

---------- ---------- ---------- ----------

7934 10 1300 1

7782 10 2450 2

7839 10 5000 3

7369 20 800 1

7876 20 1100 2

7566 20 2975 3

7788 20 3000 4

7902 20 3000 4

7900 30 950 1

**7654 30 1250 2**

**7521 30 1250 2**

**7844 30 1500 3**

7499 30 1600 4

7698 30 2850 5

SQL>

**FIRST and LAST**

**The FIRST and LAST functions can be used to return the first or last value from an ordered sequence**. Say we want to display the salary of each employee, along with the lowest and highest within their department we may use something like.

SELECT empno,

deptno,

sal,

MIN(sal) KEEP (DENSE\_RANK FIRST ORDER BY sal) OVER (PARTITION BY deptno) "Lowest",

MAX(sal) KEEP (DENSE\_RANK LAST ORDER BY sal) OVER (PARTITION BY deptno) "Highest"

FROM emp

ORDER BY deptno, sal;

EMPNO DEPTNO SAL Lowest Highest

---------- ---------- ---------- ---------- ----------

7934 10 1300 1300 5000

7782 10 2450 1300 5000

7839 10 5000 1300 5000

7369 20 800 800 3000

7876 20 1100 800 3000

7566 20 2975 800 3000

7788 20 3000 800 3000

7902 20 3000 800 3000

7900 30 950 950 2850

7654 30 1250 950 2850

7521 30 1250 950 2850

7844 30 1500 950 2850

7499 30 1600 950 2850

7698 30 2850 950 2850

SQL>

**FIRST\_VALUE and LAST\_VALUE Analytic Functions**

This article gives an overview of the FIRST\_VALUE and LAST\_VALUE analytic functions.

* [FIRST\_VALUE](http://www.oracle-base.com/articles/misc/first-value-and-last-value-analytic-functions.php#first-value)
* [LAST\_VALUE](http://www.oracle-base.com/articles/misc/first-value-and-last-value-analytic-functions.php#last-value)

**FIRST\_VALUE**

The FIRST\_VALUE analytic function is similar to the FIRST analytic function, allowing you to return the first result from an ordered set.

SELECT empno,

deptno,

sal,

FIRST\_VALUE(sal) IGNORE NULLS

OVER (PARTITION BY deptno ORDER BY sal) AS lowest\_in\_dept

FROM emp;

EMPNO DEPTNO SAL LOWEST\_IN\_DEPT

---------- ---------- ---------- --------------

7934 10 1300 1300

7782 10 2450 1300

7839 10 5000 1300

7369 20 800 800

7876 20 1100 800

7566 20 2975 800

7788 20 3000 800

7902 20 3000 800

7900 30 950 950

7654 30 1250 950

7521 30 1250 950

7844 30 1500 950

7499 30 1600 950

7698 30 2850 950

SQL>

The "{RESPECT | IGNORE} NULLS" clause indicates if NULLs are considered when determining results.

The windowing clause can be used to alter the window of operation. The following example uses "ROWS 1 PRECEDING" to give a result similar, but not quite the same, to a LAG of 1 row.

SELECT empno,

deptno,

sal,

FIRST\_VALUE(sal) IGNORE NULLS

OVER (PARTITION BY deptno ORDER BY sal ROWS 1 PRECEDING) AS preceding\_in\_dept

FROM emp;

EMPNO DEPTNO SAL PRECEDING\_IN\_DEPT

---------- ---------- ---------- -----------------

7934 10 1300 1300

7782 10 2450 1300

7839 10 5000 2450

7369 20 800 800

7876 20 1100 800

7566 20 2975 1100

7788 20 3000 2975

7902 20 3000 3000

7900 30 950 950

7654 30 1250 950

7521 30 1250 1250

7844 30 1500 1250

7499 30 1600 1500

7698 30 2850 1600

SQL>

**LAST\_VALUE**

The LAST\_VALUE analytic function is similar to the LAST analytic function, allowing you to return the last result from an ordered set. Using the default windowing clause the result can be a little unexpected.

SELECT empno,

deptno,

sal,

LAST\_VALUE(sal) IGNORE NULLS

OVER (PARTITION BY deptno ORDER BY sal) AS highest\_in\_dept

FROM emp;

EMPNO DEPTNO SAL HIGHEST\_IN\_DEPT

---------- ---------- ---------- ---------------

7934 10 1300 1300

7782 10 2450 2450

7839 10 5000 5000

7369 20 800 800

7876 20 1100 1100

7566 20 2975 2975

7788 20 3000 3000

7902 20 3000 3000

7900 30 950 950

7654 30 1250 1250

7521 30 1250 1250

7844 30 1500 1500

7499 30 1600 1600

7698 30 2850 2850

SQL>

This is because the default windowing clause is "RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW", which in this example means the current row will always be the last value. Altering the windowing clause to "RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING" gives us the result we probably expected.

SELECT empno,

deptno,

sal,

LAST\_VALUE(sal) IGNORE NULLS

OVER (PARTITION BY deptno ORDER BY sal RANGE BETWEEN

UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS highest\_in\_dept

FROM emp;

EMPNO DEPTNO SAL HIGHEST\_IN\_DEPT

---------- ---------- ---------- ---------------

7934 10 1300 5000

7782 10 2450 5000

7839 10 5000 5000

7369 20 800 3000

7876 20 1100 3000

7566 20 2975 3000

7788 20 3000 3000

7902 20 3000 3000

7900 30 950 2850

7654 30 1250 2850

7521 30 1250 2850

7844 30 1500 2850

7499 30 1600 2850

7698 30 2850 2850

SQL>

As with the previous function, the "{RESPECT | IGNORE} NULLS" clause indicates if NULLs are considered when determining results. The default action is RESPECT NULLS.

# LAG and LEAD Analytic Functions

The LAG and LEAD analytic functions were introduced in 8.1.6 to give access to multiple rows within a table, without the need for a self-join.

* [Introduction](http://www.oracle-base.com/articles/misc/lag-lead-analytic-functions.php#introduction)
* [LAG](http://www.oracle-base.com/articles/misc/lag-lead-analytic-functions.php#lag)
* [LEAD](http://www.oracle-base.com/articles/misc/lag-lead-analytic-functions.php#lead)

## Introduction

Both LAG and LEAD functions have the same usage, as shown below.

LAG (value\_expression [,offset] [,default]) OVER ([query\_partition\_clause] order\_by\_clause)

LEAD (value\_expression [,offset] [,default]) OVER ([query\_partition\_clause] order\_by\_clause)

* value\_expression - Can be a column or a built-in function, except for other analytic functions.
* offset - The number of rows preceeding/following the current row, from which the data is to be retrieved. The default value is 1.
* default - The value returned if the offset is outside the scope of the window. The default value is NULL.

Looking at the EMP table, we query the data in salary (SAL) order.

SELECT empno,

ename,

job,

sal

FROM emp

ORDER BY sal;

EMPNO ENAME JOB SAL

---------- ---------- --------- ----------

7369 SMITH CLERK 800

7900 JAMES CLERK 950

7876 ADAMS CLERK 1100

7521 WARD SALESMAN 1250

7654 MARTIN SALESMAN 1250

7934 MILLER CLERK 1300

7844 TURNER SALESMAN 1500

7499 ALLEN SALESMAN 1600

7782 CLARK MANAGER 2450

7698 BLAKE MANAGER 2850

7566 JONES MANAGER 2975

7788 SCOTT ANALYST 3000

7902 FORD ANALYST 3000

7839 KING PRESIDENT 5000

SQL>

## LAG

The LAG function is used to access data from a previous row. The following query returns the salary from the previous row to calculate the difference between the salary of the current row and that of the previous row. Notice that the ORDER BY of the LAG function is used to order the data by salary.

SELECT empno,

ename,

job,

sal,

LAG(sal, 1, 0) OVER (ORDER BY sal) AS sal\_prev,

sal - LAG(sal, 1, 0) OVER (ORDER BY sal) AS sal\_diff

FROM emp;

EMPNO ENAME JOB SAL SAL\_PREV SAL\_DIFF

---------- ---------- --------- ---------- ---------- ----------

7369 SMITH CLERK 800 0 800

7900 JAMES CLERK 950 800 150

7876 ADAMS CLERK 1100 950 150

7521 WARD SALESMAN 1250 1100 150

7654 MARTIN SALESMAN 1250 1250 0

7934 MILLER CLERK 1300 1250 50

7844 TURNER SALESMAN 1500 1300 200

7499 ALLEN SALESMAN 1600 1500 100

7782 CLARK MANAGER 2450 1600 850

7698 BLAKE MANAGER 2850 2450 400

7566 JONES MANAGER 2975 2850 125

7788 SCOTT ANALYST 3000 2975 25

7902 FORD ANALYST 3000 3000 0

7839 KING PRESIDENT 5000 3000 2000

SQL>

## LEAD

The LEAD function is used to return data from the next row. The following query returns the salary from the next row to calulate the difference between the salary of the current row and the following row.

SELECT empno,

ename,

job,

sal,

LEAD(sal, 1, 0) OVER (ORDER BY sal) AS sal\_next,

LEAD(sal, 1, 0) OVER (ORDER BY sal) - sal AS sal\_diff

FROM emp;

EMPNO ENAME JOB SAL SAL\_NEXT SAL\_DIFF

---------- ---------- --------- ---------- ---------- ----------

7369 SMITH CLERK 800 950 150

7900 JAMES CLERK 950 1100 150

7876 ADAMS CLERK 1100 1250 150

7521 WARD SALESMAN 1250 1250 0

7654 MARTIN SALESMAN 1250 1300 50

7934 MILLER CLERK 1300 1500 200

7844 TURNER SALESMAN 1500 1600 100

7499 ALLEN SALESMAN 1600 2450 850

7782 CLARK MANAGER 2450 2850 400

7698 BLAKE MANAGER 2850 2975 125

7566 JONES MANAGER 2975 3000 25

7788 SCOTT ANALYST 3000 3000 0

7902 FORD ANALYST 3000 5000 2000

7839 KING PRESIDENT 5000 0 -5000

SQL>

# String Aggregation Techniques

On occasion it is necessary to aggregate data from a number of rows into a single row, giving a list of data associated with a specific value. Using the SCOTT.EMP table as an example, we might want to retrieve a list of employees for each department. Below is a list of the base data and the type of output we would like to return from an aggregate query.

Base Data:

DEPTNO ENAME

---------- ----------

20 SMITH

30 ALLEN

30 WARD

20 JONES

30 MARTIN

30 BLAKE

10 CLARK

20 SCOTT

10 KING

30 TURNER

20 ADAMS

30 JAMES

20 FORD

10 MILLER

Desired Output:

DEPTNO EMPLOYEES

---------- --------------------------------------------------

10 CLARK,KING,MILLER

20 SMITH,FORD,ADAMS,SCOTT,JONES

30 ALLEN,BLAKE,MARTIN,TURNER,JAMES,WARD

This article is based on a thread from [asktom.oracle.com](http://asktom.oracle.com/pls/apex/f?p=100:11:0::::P11_QUESTION_ID:2196162600402) and contains several methods to achieve the desired results.

* [LISTAGG Analytic Function in 11g Release 2](http://www.oracle-base.com/articles/misc/string-aggregation-techniques.php#listagg)
* [WM\_CONCAT Built-in Function (Not Supported)](http://www.oracle-base.com/articles/misc/string-aggregation-techniques.php#wm_concat)
* [User-Defined Aggregate Function](http://www.oracle-base.com/articles/misc/string-aggregation-techniques.php#user_defined_aggregate_function)
* [Specific Function](http://www.oracle-base.com/articles/misc/string-aggregation-techniques.php#specific_function)
* [Generic Function using Ref Cursor](http://www.oracle-base.com/articles/misc/string-aggregation-techniques.php#generic_function_using_ref_cursor)
* [ROW\_NUMBER() and SYS\_CONNECT\_BY\_PATH functions in Oracle 9i](http://www.oracle-base.com/articles/misc/string-aggregation-techniques.php#row_number)
* [COLLECT function in Oracle 10g](http://www.oracle-base.com/articles/misc/string-aggregation-techniques.php#collect)

## LISTAGG Analystic Function in 11g Release 2

The LISTAGG analytic function was introduced in Oracle 11g Release 2, making it very easy to aggregate strings. The nice thing about this function is it also allows us to order the elements in the concatenated list. If you are using 11g Release 2 you should use this function for string aggregation.

COLUMN employees FORMAT A50

SELECT deptno, LISTAGG(ename, ',') WITHIN GROUP (ORDER BY ename) AS employees

FROM emp

GROUP BY deptno;

DEPTNO EMPLOYEES

---------- --------------------------------------------------

10 CLARK,KING,MILLER

20 ADAMS,FORD,JONES,SCOTT,SMITH

30 ALLEN,BLAKE,JAMES,MARTIN,TURNER,WARD

3 rows selected.

## WM\_CONCAT Built-in Function (Not Supported)

If you are not running 11g Release 2, but are running a version of the database where the WM\_CONCAT function is present, then it is a zero effort solution as it performs the aggregation for you. It is actually an example of a user defined aggregate function described below, but Oracle have done all the work for you.

COLUMN employees FORMAT A50

SELECT deptno, wm\_concat(ename) AS employees

FROM emp

GROUP BY deptno;

DEPTNO EMPLOYEES

---------- --------------------------------------------------

10 CLARK,KING,MILLER

20 SMITH,FORD,ADAMS,SCOTT,JONES

30 ALLEN,BLAKE,MARTIN,TURNER,JAMES,WARD

3 rows selected.

Note. WM\_CONCAT is an undocumented function and as such is not supported by Oracle for user applications ([MOS Note ID 1336219.1](https://support.oracle.com/CSP/main/article?cmd=show&type=NOT&doctype=HOWTO&id=1336219.1)). If this concerns you, use a User-Defined Aggregate Function described below.

## User-Defined Aggregate Function

The WM\_CONCAT function described above is an example of a user-defined aggregate function that Oracle have already created for you. If you don't want to use WM\_CONCAT, you can create your own user-defined aggregate function as described at [asktom.oracle.com](http://asktom.oracle.com/pls/apex/f?p=100:11:0::::P11_QUESTION_ID:2196162600402).

CREATE OR REPLACE TYPE t\_string\_agg AS OBJECT

(

g\_string VARCHAR2(32767),

STATIC FUNCTION ODCIAggregateInitialize(sctx IN OUT t\_string\_agg)

RETURN NUMBER,

MEMBER FUNCTION ODCIAggregateIterate(self IN OUT t\_string\_agg,

value IN VARCHAR2 )

RETURN NUMBER,

MEMBER FUNCTION ODCIAggregateTerminate(self IN t\_string\_agg,

returnValue OUT VARCHAR2,

flags IN NUMBER)

RETURN NUMBER,

MEMBER FUNCTION ODCIAggregateMerge(self IN OUT t\_string\_agg,

ctx2 IN t\_string\_agg)

RETURN NUMBER

);

/

SHOW ERRORS

CREATE OR REPLACE TYPE BODY t\_string\_agg IS

STATIC FUNCTION ODCIAggregateInitialize(sctx IN OUT t\_string\_agg)

RETURN NUMBER IS

BEGIN

sctx := t\_string\_agg(NULL);

RETURN ODCIConst.Success;

END;

MEMBER FUNCTION ODCIAggregateIterate(self IN OUT t\_string\_agg,

value IN VARCHAR2 )

RETURN NUMBER IS

BEGIN

SELF.g\_string := self.g\_string || ',' || value;

RETURN ODCIConst.Success;

END;

MEMBER FUNCTION ODCIAggregateTerminate(self IN t\_string\_agg,

returnValue OUT VARCHAR2,

flags IN NUMBER)

RETURN NUMBER IS

BEGIN

returnValue := RTRIM(LTRIM(SELF.g\_string, ','), ',');

RETURN ODCIConst.Success;

END;

MEMBER FUNCTION ODCIAggregateMerge(self IN OUT t\_string\_agg,

ctx2 IN t\_string\_agg)

RETURN NUMBER IS

BEGIN

SELF.g\_string := SELF.g\_string || ',' || ctx2.g\_string;

RETURN ODCIConst.Success;

END;

END;

/

SHOW ERRORS

CREATE OR REPLACE FUNCTION string\_agg (p\_input VARCHAR2)

RETURN VARCHAR2

PARALLEL\_ENABLE AGGREGATE USING t\_string\_agg;

/

SHOW ERRORS

The aggregate function is implemented using a type and type body, and is used within a query.

COLUMN employees FORMAT A50

SELECT deptno, string\_agg(ename) AS employees

FROM emp

GROUP BY deptno;

DEPTNO EMPLOYEES

---------- --------------------------------------------------

10 CLARK,KING,MILLER

20 SMITH,FORD,ADAMS,SCOTT,JONES

30 ALLEN,BLAKE,MARTIN,TURNER,JAMES,WARD

3 rows selected.

## Specific Function

One approach is to write a specific function to solve the problems. The get\_employees function listed below returns a list of employees for the specified department.

CREATE OR REPLACE FUNCTION get\_employees (p\_deptno in emp.deptno%TYPE)

RETURN VARCHAR2

IS

l\_text VARCHAR2(32767) := NULL;

BEGIN

FOR cur\_rec IN (SELECT ename FROM emp WHERE deptno = p\_deptno) LOOP

l\_text := l\_text || ',' || cur\_rec.ename;

END LOOP;

RETURN LTRIM(l\_text, ',');

END;

/

SHOW ERRORS

The function can then be incorporated into a query as follows.

COLUMN employees FORMAT A50

SELECT deptno,

get\_employees(deptno) AS employees

FROM emp

GROUP by deptno;

DEPTNO EMPLOYEES

---------- --------------------------------------------------

10 CLARK,KING,MILLER

20 SMITH,JONES,SCOTT,ADAMS,FORD

30 ALLEN,WARD,MARTIN,BLAKE,TURNER,JAMES

3 rows selected.

To reduce the number of calls to the function, and thereby improve performance, we might want to filter the rows in advance.

COLUMN employees FORMAT A50

SELECT e.deptno,

get\_employees(e.deptno) AS employees

FROM (SELECT DISTINCT deptno

FROM emp) e;

DEPTNO EMPLOYEES

---------- --------------------------------------------------

10 CLARK,KING,MILLER

20 SMITH,JONES,SCOTT,ADAMS,FORD

30 ALLEN,WARD,MARTIN,BLAKE,TURNER,JAMES

3 rows selected.

## Generic Function using Ref Cursor

An alternative approach is to write a function to concatenate values passed using a ref cursor. This is essentially the same as the previous example, except that the cursor is passed in making it generic, as shown below.

CREATE OR REPLACE FUNCTION concatenate\_list (p\_cursor IN SYS\_REFCURSOR)

RETURN VARCHAR2

IS

l\_return VARCHAR2(32767);

l\_temp VARCHAR2(32767);

BEGIN

LOOP

FETCH p\_cursor

INTO l\_temp;

EXIT WHEN p\_cursor%NOTFOUND;

l\_return := l\_return || ',' || l\_temp;

END LOOP;

RETURN LTRIM(l\_return, ',');

END;

/

SHOW ERRORS

The CURSOR function is used to allow a query to be passed to the function as a ref cursor, as shown below.

COLUMN employees FORMAT A50

SELECT e1.deptno,

concatenate\_list(CURSOR(SELECT e2.ename FROM emp e2 WHERE e2.deptno = e1.deptno)) employees

FROM emp e1

GROUP BY e1.deptno;

DEPTNO EMPLOYEES

---------- --------------------------------------------------

10 CLARK,KING,MILLER

20 SMITH,JONES,SCOTT,ADAMS,FORD

30 ALLEN,WARD,MARTIN,BLAKE,TURNER,JAMES

3 rows selected.

Once again, the total number of function calls can be reduced by filtering the distinct values, rather than calling the function for each row.

COLUMN employees FORMAT A50

SELECT deptno,

concatenate\_list(CURSOR(SELECT e2.ename FROM emp e2 WHERE e2.deptno = e1.deptno)) employees

FROM (SELECT DISTINCT deptno

FROM emp) e1;

DEPTNO EMPLOYEES

---------- --------------------------------------------------

10 CLARK,KING,MILLER

20 SMITH,JONES,SCOTT,ADAMS,FORD

30 ALLEN,WARD,MARTIN,BLAKE,TURNER,JAMES

3 rows selected.

## ROW\_NUMBER() and SYS\_CONNECT\_BY\_PATH functions in Oracle 9i

An example on [williamrobertson.net](http://www.williamrobertson.pwp.blueyonder.co.uk/documents/one_row.html) uses the ROW\_NUMBER() and SYS\_CONNECT\_BY\_PATH functions to achieve the same result without the use of PL/SQL or additional type definitions.

SELECT deptno,

LTRIM(MAX(SYS\_CONNECT\_BY\_PATH(ename,','))

KEEP (DENSE\_RANK LAST ORDER BY curr),',') AS employees

FROM (SELECT deptno,

ename,

ROW\_NUMBER() OVER (PARTITION BY deptno ORDER BY ename) AS curr,

ROW\_NUMBER() OVER (PARTITION BY deptno ORDER BY ename) -1 AS prev

FROM emp)

GROUP BY deptno

CONNECT BY prev = PRIOR curr AND deptno = PRIOR deptno

START WITH curr = 1;

DEPTNO EMPLOYEES

---------- --------------------------------------------------

10 CLARK,KING,MILLER

20 ADAMS,FORD,JONES,SCOTT,SMITH

30 ALLEN,BLAKE,JAMES,MARTIN,TURNER,WARD

3 rows selected.

## COLLECT function in Oracle 10g

An example on [oracle-developer.net](http://www.oracle-developer.net/display.php?id=306) uses the COLLECT function in Oracle 10g to get the same result. This method requires a table type and a function to convert the contents of the table type to a string. I've altered his method slightly to bring it in line with this article.

CREATE OR REPLACE TYPE t\_varchar2\_tab AS TABLE OF VARCHAR2(4000);

/

CREATE OR REPLACE FUNCTION tab\_to\_string (p\_varchar2\_tab IN t\_varchar2\_tab,

p\_delimiter IN VARCHAR2 DEFAULT ',') RETURN VARCHAR2 IS

l\_string VARCHAR2(32767);

BEGIN

FOR i IN p\_varchar2\_tab.FIRST .. p\_varchar2\_tab.LAST LOOP

IF i != p\_varchar2\_tab.FIRST THEN

l\_string := l\_string || p\_delimiter;

END IF;

l\_string := l\_string || p\_varchar2\_tab(i);

END LOOP;

RETURN l\_string;

END tab\_to\_string;

/

The query below shows the COLLECT function in action.

COLUMN employees FORMAT A50

SELECT deptno,

tab\_to\_string(CAST(COLLECT(ename) AS t\_varchar2\_tab)) AS employees

FROM emp

GROUP BY deptno;

DEPTNO EMPLOYEES

---------- --------------------------------------------------

10 CLARK,KING,MILLER

20 SMITH,JONES,SCOTT,ADAMS,FORD

30 ALLEN,WARD,MARTIN,BLAKE,TURNER,JAMES

3 rows selected.

# Top-N Queries

Top-N queries provide a method for limiting the number of rows returned from ordered sets of data. They are extremely useful when you want to return the top or bottom "N" number of rows from a set or when you are paging through data. This article presents several methods to implement Top-N queries.

* [Setup](http://www.oracle-base.com/articles/misc/top-n-queries.php#setup)
* [What not to do!](http://www.oracle-base.com/articles/misc/top-n-queries.php#mistake)
* [Row Limiting Clause (12c onward)](http://www.oracle-base.com/articles/misc/top-n-queries.php#row-limiting-clause)
* [Inline View and ROWNUM](http://www.oracle-base.com/articles/misc/top-n-queries.php#rownum)
* [WITH Clause and ROWNUM](http://www.oracle-base.com/articles/misc/top-n-queries.php#with)
* [RANK](http://www.oracle-base.com/articles/misc/top-n-queries.php#rank)
* [DENSE\_RANK](http://www.oracle-base.com/articles/misc/top-n-queries.php#dense_rank)
* [ROW\_NUMBER](http://www.oracle-base.com/articles/misc/top-n-queries.php#row_number)

**Setup**

First we must create and populate a test table.

DROP TABLE rownum\_order\_test;

CREATE TABLE rownum\_order\_test (

val NUMBER

);

INSERT ALL

INTO rownum\_order\_test

INTO rownum\_order\_test

SELECT level

FROM dual

CONNECT BY level <= 10;

COMMIT;

The following query shows we have 20 rows with 10 distinct values.

SELECT val

FROM rownum\_order\_test

ORDER BY val;

VAL

----------

1

1

2

2

3

3

4

4

5

5

6

VAL

----------

6

7

7

8

8

9

9

10

10

20 rows selected.

SQL>

**What not to do!**

The following example shows a common trap people fall into when they don't understand the way the ROWNUM pseudocolumn and ORDER BY clause interact. Let's assume we wanted to return the top 5 values in the ID column. We might decide to order the data by descending ID and pick off the first five rows. That sounds correct, so we go ahead and issue the following query.

SELECT val

FROM rownum\_order\_test

WHERE rownum <= 5

ORDER BY val DESC;

VAL

----------

5

4

3

2

1

5 rows selected.

SQL>

That didn't do what we wanted!

The problem is that the ROWNUM assignment is performed prior to the ORDER BY operation, resulting in potentially random data being returned.

**Row Limiting Clause (12c onward)**

If you are using Oracle 12c onward, Top-N queries just got a whole lot easier with the introduction of the [Row Limiting Clause](http://www.oracle-base.com/articles/12c/row-limiting-clause-for-top-n-queries-12cr1.php). If backwards compatibility is not your concern, switch to using this new clause.

**Inline View and ROWNUM**

The classic Top-N style query uses an ordered inline view to force the data into the correct order, then uses the ROWNUM check to limit the data returned.

SELECT val

FROM (SELECT val

FROM rownum\_order\_test

ORDER BY val DESC)

WHERE ROWNUM <= 5;

VAL

----------

10

10

9

9

8

5 rows selected.

SQL>

As the data is in the desired order before the ROWNUM check is performed, we get the result we wanted. Notice that we asked for 5 rows and we got five, even though there is a second row with the value "8".

We can return the 5 smallest values by altering the ORDER BY clause to ascending.

SELECT val

FROM (SELECT val

FROM rownum\_order\_test

ORDER BY val)

WHERE rownum <= 5;

VAL

----------

1

1

2

2

3

5 rows selected.

SQL>

This method can also be used for paging through data, like paged web reports.

SELECT val

FROM (SELECT val, rownum AS rnum

FROM (SELECT val

FROM rownum\_order\_test

ORDER BY val)

WHERE rownum <= 8)

WHERE rnum >= 4;

VAL

----------

2

3

3

4

4

5 rows selected.

SQL>

This looks like it might not perform well, but we can see from the execution plan that Oracle can push the predicates down into the inline views to make them much more efficient, so this is the best way to page through data using SQL.

Execution Plan

----------------------------------------------------------

Plan hash value: 2927523340

----------------------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | Cost (%CPU)| Time |

----------------------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 8 | 208 | 4 (25)| 00:00:01 |

|\* 1 | VIEW | | 8 | 208 | 4 (25)| 00:00:01 |

|\* 2 | COUNT STOPKEY | | | | | |

| 3 | VIEW | | 100 | 1300 | 4 (25)| 00:00:01 |

|\* 4 | SORT ORDER BY STOPKEY| | 100 | 1300 | 4 (25)| 00:00:01 |

| 5 | TABLE ACCESS FULL | ROWNUM\_ORDER\_TEST | 100 | 1300 | 3 (0)| 00:00:01 |

----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

1 - filter("RNUM">=4)

2 - filter(ROWNUM<=8)

4 - filter(ROWNUM<=8)

**WITH Clause and ROWNUM**

The previous example can be rewritten to use a WITH clause in place of the inline view.

WITH ordered\_query AS

(SELECT val

FROM rownum\_order\_test

ORDER BY val DESC)

SELECT val

FROM ordered\_query

WHERE rownum <= 5;

VAL

----------

10

10

9

9

8

5 rows selected.

SQL>

**RANK**

The RANK analytic function assigns a sequential rank for each distinct value in the specified window.

SELECT val

FROM (SELECT val,

RANK() OVER (ORDER BY val DESC) AS val\_rank

FROM rownum\_order\_test)

WHERE val\_rank <= 5;

VAL

----------

10

10

9

9

8

8

6 rows selected.

SQL>

At first glance this looks like there may be a problem, but displaying the rank information shows us what is happening.

SELECT val, val\_rank

FROM (SELECT val,

RANK() OVER (ORDER BY val DESC) AS val\_rank

FROM rownum\_order\_test)

WHERE val\_rank <= 5;

VAL VAL\_RANK

---------- ----------

10 1

10 1

9 3

9 3

8 5

8 5

6 rows selected.

SQL>

From this we can see that duplicate rows are assigned the same rank, followed by a skip in the sequence to keep the rank consistent. Similar to Olympic medal places. This means the RANK function doesn't give us the "top N rows" or the "top N distinct values". The number of rows returned is dependent on the number of duplicates in the data.

**DENSE\_RANK**

The DENSE\_RANK analytic function is similar to the RANK analytic function in that it assigns a sequential rank for each distinct value in the specified window. The difference being the ranks are compacted, so there are no gaps.

SELECT val

FROM (SELECT val,

DENSE\_RANK() OVER (ORDER BY val DESC) AS val\_rank

FROM rownum\_order\_test)

WHERE val\_rank <= 5;

VAL

----------

10

10

9

9

8

8

7

7

6

6

10 rows selected.

SQL>

Displaying the rank information shows us what is happening.

SELECT val, val\_rank

FROM (SELECT val,

DENSE\_RANK() OVER (ORDER BY val DESC) AS val\_rank

FROM rownum\_order\_test)

WHERE val\_rank <= 5;

VAL VAL\_RANK

---------- ----------

10 1

10 1

9 2

9 2

8 3

8 3

7 4

7 4

6 5

6 5

10 rows selected.

SQL>

Once again, duplicate values are assigned the same rank, but there is no gap in the rank sequence. As a result DENSE\_RANK always gives us a "top N distinct values" result.

**ROW\_NUMBER**

The ROW\_NUMBER analytic function is similar to the ROWNUM pseudocolumn in that it assigns a unique number for each row returned, but like all analytic functions its action can be limited to a specific window of data in the result set and based on the order of data in that window. In this simple example using a window of the whole result set it functions the same as the ROWNUM psuedocolumn.

SELECT val

FROM (SELECT val,

ROW\_NUMBER() OVER (ORDER BY val DESC) AS val\_row\_number

FROM rownum\_order\_test)

WHERE val\_row\_number <= 5;

VAL

----------

10

10

9

9

8

5 rows selected.

SQL>