select USER\_ID,prog\_code,dept\_code,user\_name

, ROW\_NUMBER() over(partition by prog\_code order by dept\_code)

, DENSE\_RANK() over(partition by prog\_code order by dept\_code)

, rank() over(partition by prog\_code order by dept\_code)

from user\_mst where user\_type='S'

* ROW\_NUMBER(): This one generates a new row number for every row, regardless of duplicates within a partition.
* RANK(): This one generates a new row number for every distinct row, leaving gaps between groups of duplicates within a partition.
* DENSE\_RANK(): This one generates a new row number for every distinct row, leaving no gaps between groups of duplicates within a partition.

As always, the above is much easier to understand by example. Let’s assume the following PostgreSQL schema containing a table with 8 records, some of which are duplicates:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | CREATE TABLE t AS  SELECT 'a' v UNION ALL  SELECT 'a'   UNION ALL  SELECT 'a'   UNION ALL  SELECT 'b'   UNION ALL  SELECT 'c'   UNION ALL  SELECT 'c'   UNION ALL  SELECT 'd'   UNION ALL  SELECT 'e' |

Now, let’s select each value, along with the three aforementioned ranking functions. And for kicks and giggles, we’ll use the SQL standard [WINDOW clause](http://www.postgresql.org/docs/9.3/static/sql-select.html#SQL-WINDOW)! Yay, it saved us 15 characters of repetitive SQL code. Note that the WINDOW clause is hardly implemented apart by PostgreSQL and Sybase SQL Anywhere…

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | SELECT    v,    ROW\_NUMBER() OVER (window) row\_number,    RANK()       OVER (window) rank,    DENSE\_RANK() OVER (window) dense\_rank  FROM t  WINDOW window AS (ORDER BY v)  ORDER BY v |

And the above results in:

+---+------------+------+------------+

| V | ROW\_NUMBER | RANK | DENSE\_RANK |

+---+------------+------+------------+

| a | 1 | 1 | 1 |

| a | 2 | 1 | 1 |

| a | 3 | 1 | 1 |

| b | 4 | 4 | 2 |

| c | 5 | 5 | 3 |

| c | 6 | 5 | 3 |

| d | 7 | 7 | 4 |

| e | 8 | 8 | 5 |

+---+------------+------+------------+

(See also this[SQLFiddle](http://sqlfiddle.com/#!12/b4124/4/0))

**How DENSE\_RANK() can help when writing SELECT DISTINCT**

No doubt, ROW\_NUMBER() is the most useful ranking function among the above, specifically when you need to emulate LIMIT .. OFFSET clauses as in DB2, Oracle (11g or less), Sybase SQL Anywhere (prior to version 12), SQL Server (2008 and less). [Read on here](http://www.jooq.org/doc/3.1/manual/sql-building/sql-statements/select-statement/limit-clause/) about how [jOOQ](http://www.jooq.org/) emulates this SQL clause in various SQL dialects.

[](http://www.jooq.org/)

But using ROW\_NUMBER() has a subtle problem when used along withDISTINCT or UNION. It prevents the database from being able to remove duplicates, because ROW\_NUMBER will always produce distinct values within a partition. In the above example, duplicate values for T.V were added on purpose. How can we first remove the duplicates and only then enumerate row numbers? Clearly, we can no longer use ROW\_NUMBER(). The following query:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT DISTINCT    v,    ROW\_NUMBER() OVER (window) row\_number  FROM t  WINDOW window AS (ORDER BY v)  ORDER BY v, row\_number |

… yields

+---+------------+

| V | ROW\_NUMBER |

+---+------------+

| a | 1 |

| a | 2 |

| a | 3 |

| b | 4 |

| c | 5 |

| c | 6 |

| d | 7 |

| e | 8 |

+---+------------+

(See also this[SQLFiddle](http://sqlfiddle.com/#!12/b4124/6))

But we can use DENSE\_RANK(), instead! With DENSE\_RANK() the ranking is applied in a way that duplicate records will receive the same rank. And there are no gaps between ranks. Hence:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | SELECT DISTINCT    v,    DENSE\_RANK() OVER (window) row\_number  FROM t  WINDOW window AS (ORDER BY v)  ORDER BY v, row\_number |

… which yields:

+---+------------+

| V | ROW\_NUMBER |

+---+------------+

| a | 1 |

| b | 2 |

| c | 3 |

| d | 4 |

| e | 5 |

+---+------------+

===

RANK gives you the ranking within your ordered partition. Ties are assigned the same rank, with the next ranking(s) skipped. So, if you have 3 items at rank 2, the next rank listed would be ranked 5.

DENSE\_RANK again gives you the ranking within your ordered partition, but the ranks are consecutive. No ranks are skipped if there are ranks with multiple items.

As for nulls, it depends on the ORDER BY clause. Here is a simple test script you can play with to see what happens:

with q as (

select 10 deptno, 'rrr' empname, 10000.00 sal from dual union all

select 11, 'nnn', 20000.00 from dual union all

select 11, 'mmm', 5000.00 from dual union all

select 12, 'kkk', 30000 from dual union all

select 10, 'fff', 40000 from dual union all

select 10, 'ddd', 40000 from dual union all

select 10, 'bbb', 50000 from dual union all

select 10, 'xxx', null from dual union all

select 10, 'ccc', 50000 from dual)

select empname, deptno, sal

, rank() over (partition by deptno order by sal nulls first) r

, dense\_rank() over (partition by deptno order by sal nulls first) dr1

, dense\_rank() over (partition by deptno order by sal nulls last) dr2

from q;

EMP DEPTNO SAL R DR1 DR2

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

xxx 10 1 1 4

rrr 10 10000 2 2 1

fff 10 40000 3 3 2

ddd 10 40000 3 3 2

ccc 10 50000 5 4 3

bbb 10 50000 5 4 3

mmm 11 5000 1 1 1

nnn 11 20000 2 2 2

kkk 12 30000 1 1 1

9 rows selected.

====

select dept\_code,prog\_code,user\_id,

ROW\_NUMBER() OVER(partition by prog\_code ORDER BY dept\_code) AS 'RowN'

,DENSE\_RANK() OVER(partition by prog\_code ORDER BY dept\_code) AS 'RowN'

,RANK() OVER(partition by prog\_code ORDER BY dept\_code) AS 'RowN'

from user\_mst

where user\_type ='S'

order by prog\_code,RANK() OVER(partition by prog\_code ORDER BY dept\_code),dept\_code

output

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

dept prog Dens\_Rank() Rank()

1 1 1(common) 1(common)

1 2 2(common) 439(common)

1 4 3(common) 756(common)

1 5 4(common) 915(common)

2 1 1(common) 1(common)

2 2 2(common) 277(common)

2 3 3(common) 394(common)

2 4 4(common) 450(common)

2 5 5(common) 781(common)

3 1 1(common) 1(common)

3 4 2(common) 6(common)