



O'REILLY®

Anthony Molinaro

Advanced Searching

In a very real sense, this entire book so far has been about searching. You've seen all sorts of queries that use joins and WHERE clauses and grouping techniques to search out and return the results that you need. Some types of searching operations, though, stand apart from others in that they represent a different way of thinking about searching. Perhaps you're displaying a result set one page at a time. Half of that problem is to identify (search for) the entire set of records that you want to display. The other half of that problem is to repeatedly search for the next page to display as a user cycles through the records on a display. Your first thought may not be to think of pagination as a searching problem, but it *can* be thought of that way, and it can be solved that way; that is the type of searching solution this chapter is all about.

Paginating Through a Result Set

Problem

You want to paginate or “scroll through” a result set. For example, you want to return the first five salaries from table EMP, then the next five, and so forth. Your goal is to allow a user to view five records at a time, scrolling forward with each click of a “Next” button.

Solution

Because there is no concept of first, last, or next in SQL, you must impose order on the rows you are working with. Only by imposing order can you accurately return ranges of records.

DB2, Oracle, and SQL Server

Use the window function `ROW_NUMBER OVER` to impose order, and specify the window of records that you want returned in your `WHERE` clause. For example, to return rows 1 through 5:

```
select sal
  from (
select row_number() over (order by sal) as rn,
       sal
  from emp
   ) x
 where rn between 1 and 5
```

```
SAL
----
800
950
1100
1250
1250
```

Then to return rows 6 through 10:

```
select sal
  from (
select row_number() over (order by sal) as rn,
       sal
  from emp
   ) x
 where rn between 6 and 10
```

```
SAL
-----
1300
1500
1600
2450
2850
```

You can return any range of rows that you wish simply by changing the `WHERE` clause of your query.

MySQL and PostgreSQL

Scrolling through a result set is particularly easy due to the `LIMIT` and `OFFSET` clauses that these products support. Use `LIMIT` to specify the number of rows to return, and use `OFFSET` to specify the number of rows to skip. For example, to return the first five rows in order of salary:

```
select sal
  from emp
 order by sal limit 5 offset 0
```

```
SAL
-----
800
950
1100
1250
1250
```

To return the next group of five rows:

```
select sal
  from emp
 order by sal limit 5 offset 5
```

```
SAL
-----
1300
1500
1600
2450
2850
```

LIMIT and OFFSET not only make the MySQL and PostgreSQL solutions easy to write, but they are quite readable, too.

Discussion

DB2, Oracle, and SQL Server

The window function ROW_NUMBER OVER in inline view X will assign a unique number to each salary (in increasing order starting from 1). Listed below is the result set for inline view X:

```
select row_number() over (order by sal) as rn,
       sal
  from emp
```

RN	SAL
1	800
2	950
3	1100
4	1250
5	1250
6	1300
7	1500
8	1600
9	2450
10	2850
11	2975
12	3000
13	3000
14	5000

Once a number has been assigned to a salary, simply pick the range you want to return by specifying values for RN.

For Oracle users, an alternative: you can use ROWNUM instead of ROW NUMBER OVER to generate sequence numbers for the rows:

```
select sal
  from (
select sal, rownum rn
  from (
select sal
  from emp
 order by sal
  )
 )
 where rn between 6 and 10

SAL
-----
1300
1500
1600
2450
2850
```

Using ROWNUM forces you into writing an extra level of subquery. The innermost subquery sorts rows by salary. The next outermost subquery applies row numbers to those rows, and, finally, the very outermost SELECT returns the data you are after.

MySQL and PostgreSQL

The OFFSET clause added to the SELECT clause makes scrolling through results intuitive and easy. Specifying OFFSET 0 will start you at the first row, OFFSET 5 at the sixth row, and OFFSET 10 at the eleventh row. The LIMIT clause restricts the number of rows returned. By combining the two clauses you can easily specify where in a result set to start returning rows and how many to return.

Skipping *n* Rows from a Table

Problem

You want a query to return every other employee in table EMP; you want the first employee, third employee, and so forth. For example, from the following result set:

```
ENAME
-----
ADAMS
ALLEN
BLAKE
CLARK
FORD
```

JAMES
JONES
KING
MARTIN
MILLER
SCOTT
SMITH
TURNER
WARD

you want to return:

```
ENAME
-----
ADAMS
BLAKE
FORD
JONES
MARTIN
SCOTT
TURNER
```

Solution

To skip the second or fourth or *n*th row from a result set, you must impose order on the result set, otherwise there is no concept of first or next, second, or fourth.

DB2, Oracle, and SQL Server

Use the window function `ROW_NUMBER OVER` to assign a number to each row, which you can then use in conjunction with the modulo function to skip unwanted rows. The modulo function is `MOD` for DB2 and Oracle. In SQL Server, use the percent (`%`) operator. The following example uses `MOD` to skip even-numbered rows:

```
1 select ename
2   from (
3 select row_number() over (order by ename) rn,
4        ename
5   from emp
6        ) x
7  where mod(rn,2) = 1
```

MySQL and PostgreSQL

Because there are no built-in functions for ranking or numbering rows, you need to use a scalar subquery to rank the rows (by name in this example). Then use modulus to skip rows:

```
1 select x.ename
2   from (
3 select a.ename,
4        (select count(*)
5         from emp b
```

```
6         where b.ename <= a.ename) as rn
7   from emp a
8     ) x
9   where mod(x.rn,2) = 1
```

Discussion

DB2, Oracle, and SQL Server

The call to the window function `ROW_NUMBER OVER` in inline view X will assign a rank to each row (no ties, even with duplicate names). The results are shown below:

```
select row_number() over (order by ename) rn, ename
from emp
```

```
RN ENAME
-----
1 ADAMS
2 ALLEN
3 BLAKE
4 CLARK
5 FORD
6 JAMES
7 JONES
8 KING
9 MARTIN
10 MILLER
11 SCOTT
12 SMITH
13 TURNER
14 WARD
```

The last step is to simply use modulus to skip every other row.

MySQL and PostgreSQL

With a function to rank or number rows, you can use a scalar subquery to first rank the employee names. Inline view X ranks each name and is shown below:

```
select a.ename,
       (select count(*)
        from emp b
        where b.ename <= a.ename) as rn
from emp a
```

```
ENAME          RN
-----
ADAMS           1
ALLEN           2
BLAKE           3
CLARK           4
FORD            5
JAMES           6
```

JONES	7
KING	8
MARTIN	9
MILLER	10
SCOTT	11
SMITH	12
TURNER	13
WARD	14

The final step is to use the modulo function on the generated rank to skip rows.

Incorporating OR Logic when Using Outer Joins

Problem

You want to return the name and department information for all employees in departments 10 and 20 along with department information for departments 30 and 40 (but no employee information). Your first attempt looks like this:

```
select e.ename, d.deptno, d.dname, d.loc
  from dept d, emp e
  where d.deptno = e.deptno
        and (e.deptno = 10 or e.deptno = 20)
  order by 2
```

ENAME	DEPTNO	DNAME	LOC
CLARK	10	ACCOUNTING	NEW YORK
KING	10	ACCOUNTING	NEW YORK
MILLER	10	ACCOUNTING	NEW YORK
SMITH	20	RESEARCH	DALLAS
ADAMS	20	RESEARCH	DALLAS
FORD	20	RESEARCH	DALLAS
SCOTT	20	RESEARCH	DALLAS
JONES	20	RESEARCH	DALLAS

Because the join in this query is an inner join, the result set does not include department information for DEPTNOs 30 and 40.

You attempt to outer join EMP to DEPT with the following query, but you still do not get the correct results:

```
select e.ename, d.deptno, d.dname, d.loc
  from dept d left join emp e
    on (d.deptno = e.deptno)
  where e.deptno = 10
        or e.deptno = 20
  order by 2
```

ENAME	DEPTNO	DNAME	LOC
CLARK	10	ACCOUNTING	NEW YORK
KING	10	ACCOUNTING	NEW YORK
MILLER	10	ACCOUNTING	NEW YORK

SMITH	20	RESEARCH	DALLAS
ADAMS	20	RESEARCH	DALLAS
FORD	20	RESEARCH	DALLAS
SCOTT	20	RESEARCH	DALLAS
JONES	20	RESEARCH	DALLAS

Ultimately, you would like the result set to be:

ENAME	DEPTNO	DNAME	LOC
CLARK	10	ACCOUNTING	NEW YORK
KING	10	ACCOUNTING	NEW YORK
MILLER	10	ACCOUNTING	NEW YORK
SMITH	20	RESEARCH	DALLAS
JONES	20	RESEARCH	DALLAS
SCOTT	20	RESEARCH	DALLAS
ADAMS	20	RESEARCH	DALLAS
FORD	20	RESEARCH	DALLAS
	30	SALES	CHICAGO
	40	OPERATIONS	BOSTON

Solution

DB2, MySQL, PostgreSQL, and SQL Server

Move the OR condition into the JOIN clause:

```

1 select e.ename, d.deptno, d.dname, d.loc
2   from dept d left join emp e
3     on (d.deptno = e.deptno
4        and (e.deptno=10 or e.deptno=20))
5  order by 2
```

Alternatively, you can filter on EMP.DEPTNO first in an inline view and then outer join:

```

1 select e.ename, d.deptno, d.dname, d.loc
2   from dept d
3  left join
4     (select ename, deptno
5      from emp
6      where deptno in ( 10, 20 )
7      ) e on ( e.deptno = d.deptno )
8  order by 2
```

Oracle

If you are on Oracle9i Database or later, you can use either of the solutions for the other products. Otherwise, you need to use CASE or DECODE in a workaround. Following is a solution using CASE:

```

select e.ename, d.deptno, d.dname, d.loc
   from dept d, emp e
```

```
where d.deptno = e.deptno (+)
      and d.deptno = case when e.deptno(+) = 10 then e.deptno(+)
                          when e.deptno(+) = 20 then e.deptno(+)
                        end
order by 2
```

And next is the same solution, but this time using DECODE:

```
select e.ename, d.deptno, d.dname, d.loc
      from dept d, emp e
      where d.deptno = e.deptno (+)
            and d.deptno = decode(e.deptno(+),10,e.deptno(+),
                                20,e.deptno(+))
order by 2
```

When using the proprietary Oracle outer join syntax (+) along with an IN or OR predicate on an outer joined column, the query will return an error. The solution is to move the IN or OR predicate to an inline view:

```
select e.ename, d.deptno, d.dname, d.loc
      from dept d,
      ( select ename, deptno
        from emp
        where deptno in ( 10, 20 )
      ) e
      where d.deptno = e.deptno (+)
order by 2
```

Discussion

DB2, MySQL, PostgreSQL, and SQL Server

Two solutions are given for these products. The first moves the OR condition into the JOIN clause, making it part of the join condition. By doing that, you can filter the rows returned from EMP without losing DEPTNOs 30 and 40 from DEPT.

The second solution moves the filtering into an inline view. Inline view E filters on EMP.DEPTNO and returns EMP rows of interest. These are then outer joined to DEPT. Because DEPT is the anchor table in the outer join, all departments, including 30 and 40, are returned.

Oracle

Use the CASE and DECODE functions as a workaround for what seems to be a bug in the older outer-join syntax. The solution using inline view E works by first finding the rows of interest in table EMP, and then outer joining to DEPT.

Determining Which Rows Are Reciprocals

Problem

You have a table containing the results of two tests, and you want to determine which pair of scores are reciprocals. Consider the result set below from view V:

```
select *  
  from V
```

TEST1	TEST2
20	20
50	25
20	20
60	30
70	90
80	130
90	70
100	50
110	55
120	60
130	80
140	70

Examining these results, you see that a test score for TEST1 of 70 and TEST2 of 90 is a reciprocal (there exists a score of 90 for TEST1 and a score of 70 for TEST2). Likewise, the scores of 80 for TEST1 and 130 for TEST2 are reciprocals of 130 for TEST1 and 80 for TEST2. Additionally, the scores of 20 for TEST1 and 20 for TEST2 are reciprocals of 20 for TEST2 and 20 for TEST1. You want to identify only one set of reciprocals. You want your result set to be this:

TEST1	TEST2
20	20
70	90
80	130

not this:

TEST1	TEST2
20	20
20	20
70	90
80	130
90	70
130	80

Solution

Use a self join to identify rows where TEST1 equals TEST2 and vice versa:

```
select distinct v1.*
  from V v1, V v2
 where v1.test1 = v2.test2
       and v1.test2 = v2.test1
       and v1.test1 <= v1.test2
```

Discussion

The self-join results in a Cartesian product in which every TEST1 score can be compared against every TEST2 score and vice versa. The query below will identify the reciprocals:

```
select v1.*
  from V v1, V v2
 where v1.test1 = v2.test2
       and v1.test2 = v2.test1
```

TEST1	TEST2
20	20
20	20
20	20
20	20
90	70
130	80
70	90
80	130

The use of DISTINCT ensures that duplicate rows are removed from the final result set. The final filter in the WHERE clause (and V1.TEST1 <= V1.TEST2) will ensure that only one pair of reciprocals (where TEST1 is the smaller or equal value) is returned.

Selecting the Top *n* Records

Problem

You want to limit a result set to a specific number of records based on a ranking of some sort. For example, you want to return the names and salaries of the employees with the top five salaries.

Solution

The key to this solution is to make two passes: first rank the rows on whatever value you want to rank on; then limit the result set to the number of rows you are interested in.

DB2, Oracle, and SQL Server

The solution to this problem depends on the use of a window function. Which window function you will use depends on how you want to deal with ties. The following solution uses DENSE_RANK, so that each tie in salary will count as only one against the total:

```
1 select ename,sal
2   from (
3 select ename, sal,
4        dense_rank() over (order by sal desc) dr
5   from emp
6   ) x
7  where dr <= 5
```

The total number of rows returned may exceed five, but there will be only five distinct salaries. Use ROW_NUMBER OVER if you wish to return five rows regardless of ties (as no ties are allowed with this function).

MySQL and PostgreSQL

Use a scalar subquery to create a rank for each salary. Then restrict the results of that subquery by rank:

```
1 select ename,sal
2   from (
3 select (select count(distinct b.sal)
4        from emp b
5        where a.sal <= b.sal) as rnk,
6        a.sal,
7        a.ename
8   from emp a
9   )
10  where rnk <= 5
```

Discussion

DB2, Oracle, and SQL Server

The window function DENSE_RANK OVER in inline view X does all the work. The following example shows the entire table after applying that function:

```
select ename, sal,
       dense_rank() over (order by sal desc) dr
from emp
```

ENAME	SAL	DR
KING	5000	1
SCOTT	3000	2
FORD	3000	2
JONES	2975	3
BLAKE	2850	4

CLARK	2450	5
ALLEN	1600	6
TURNER	1500	7
MILLER	1300	8
WARD	1250	9
MARTIN	1250	9
ADAMS	1100	10
JAMES	950	11
SMITH	800	12

Now it's just a matter of returning rows where DR is less than or equal to five.

MySQL and PostgreSQL

The scalar subquery in inline view X ranks the salaries as follows:

```
select (select count(distinct b.sal)
        from emp b
        where a.sal <= b.sal) as rnk,
       a.sal,
       a.ename
from emp a
```

RNK	SAL	ENAME
1	5000	KING
2	3000	SCOTT
2	3000	FORD
3	2975	JONES
4	2850	BLAKE
5	2450	CLARK
6	1600	ALLEN
7	1500	TURNER
8	1300	MILLER
9	1250	WARD
9	1250	MARTIN
10	1100	ADAMS
11	950	JAMES
12	800	SMITH

The final step is to return only rows where RNK is less than or equal to five.

Finding Records with the Highest and Lowest Values

Problem

You want to find “extreme” values in your table. For example, you want to find the employees with the highest and lowest salaries in table EMP.

Solution

DB2, Oracle, and SQL Server

Use the window functions MIN OVER and MAX OVER to find the lowest and highest salaries, respectively:

```
1 select ename
2   from (
3 select ename, sal,
4        min(sal)over() min_sal,
5        max(sal)over() max_sal
6   from emp
7        ) x
8  where sal in (min_sal,max_sal)
```

MySQL and PostgreSQL

Write two subqueries, one each to return the MIN and MAX values of SAL:

```
1 select ename
2   from emp
3  where sal in ( (select min(sal) from emp),
4                (select max(sal) from emp) )
```

Discussion

DB2, Oracle, and SQL Server

The window functions MIN OVER and MAX OVER allow each row to have access to the lowest and highest salaries. The result set from inline view X is as follows:

```
select ename, sal,
       min(sal)over() min_sal,
       max(sal)over() max_sal
from emp
```

ENAME	SAL	MIN_SAL	MAX_SAL
SMITH	800	800	5000
ALLEN	1600	800	5000
WARD	1250	800	5000
JONES	2975	800	5000
MARTIN	1250	800	5000
BLAKE	2850	800	5000
CLARK	2450	800	5000
SCOTT	3000	800	5000
KING	5000	800	5000
TURNER	1500	800	5000
ADAMS	1100	800	5000
JAMES	950	800	5000
FORD	3000	800	5000
MILLER	1300	800	5000

Given this result set, all that's left is to return rows where SAL equals MIN_SAL or MAX_SAL.

MySQL and PostgreSQL

This solution uses two subqueries in one IN list to find the lowest and highest salaries from EMP. The rows returned by the outer query are the ones having salaries that match the values returned by either subquery.

Investigating Future Rows

Problem

You want to find any employees who earn less than the employee hired immediately after them. Based on the following result set:

ENAME	SAL	HIREDATE
SMITH	800	17-DEC-80
ALLEN	1600	20-FEB-81
WARD	1250	22-FEB-81
JONES	2975	02-APR-81
BLAKE	2850	01-MAY-81
CLARK	2450	09-JUN-81
TURNER	1500	08-SEP-81
MARTIN	1250	28-SEP-81
KING	5000	17-NOV-81
JAMES	950	03-DEC-81
FORD	3000	03-DEC-81
MILLER	1300	23-JAN-82
SCOTT	3000	09-DEC-82
ADAMS	1100	12-JAN-83

SMITH, WARD, MARTIN, JAMES, and MILLER earn less than the person hired immediately after they were hired, so those are the employees you wish to find with a query.

Solution

The first step is to define what “future” means. You must impose order on your result set to be able to define a row as having a value that is “later” than another.

DB2, MySQL, PostgreSQL, and SQL Server

Use subqueries to determine the following for each employee:

- The date of the first person subsequently hired with a greater salary
- The date of the next person to be hired

When the two dates match, you have what you are looking for:

```

1 select ename, sal, hiredate
2   from (
3 select a.ename, a.sal, a.hiredate,
4        (select min(hiredate) from emp b
5         where b.hiredate > a.hiredate
6         and b.sal      > a.sal ) as next_sal_grtr,
7        (select min(hiredate) from emp b
8         where b.hiredate > a.hiredate) as next_hire
9   from emp a
10  ) x
11 where next_sal_grtr = next_hire

```

Oracle

You can use the LEAD OVER window function to access the salary of the next employee that was hired. It's then a simple matter to check whether that salary is larger:

```

1 select ename, sal, hiredate
2   from (
3 select ename, sal, hiredate,
4        lead(sal)over(order by hiredate) next_sal
5   from emp
6  )
7  where sal < next_sal

```

Discussion

DB2, MySQL, PostgreSQL, and SQL Server

The scalar subqueries return, for each employee, the HIREDATE of the very next employee hired and the HIREDATE of the first, subsequently hired employee who earns more than the current employee. Here's a look at the raw data:

```

select a.ename, a.sal, a.hiredate,
       (select min(hiredate) from emp b
        where b.hiredate > a.hiredate
        and b.sal      > a.sal ) as next_sal_grtr,
       (select min(hiredate) from emp b
        where b.hiredate > a.hiredate) as next_hire
from emp a

```

ENAME	SAL	HIREDATE	NEXT_SAL_GRTR	NEXT_HIRE
SMITH	800	17-DEC-80	20-FEB-81	20-FEB-81
ALLEN	1600	20-FEB-81	02-APR-81	22-FEB-81
WARD	1250	22-FEB-81	02-APR-81	02-APR-81
JONES	2975	02-APR-81	17-NOV-81	01-MAY-81
MARTIN	1250	28-SEP-81	17-NOV-81	17-NOV-81
BLAKE	2850	01-MAY-81	17-NOV-81	09-JUN-81
CLARK	2450	09-JUN-81	17-NOV-81	08-SEP-81
SCOTT	3000	09-DEC-82		12-JAN-83

KING	5000	17-NOV-81		03-DEC-81
TURNER	1500	08-SEP-81	17-NOV-81	28-SEP-81
ADAMS	1100	12-JAN-83		
JAMES	950	03-DEC-81	23-JAN-82	23-JAN-82
FORD	3000	03-DEC-81		23-JAN-82
MILLER	1300	23-JAN-82	09-DEC-82	09-DEC-82

Someone hired subsequently may or may not have been hired immediately after the current employee was hired. The next (and last) step then is to return only rows where NEXT_SAL_GRTR (the earliest HIREDATE of an employee who earns more than the current employee) equals NEXT_HIRE (the HIREDATE of the very next employee relative to the current employee’s HIREDATE).

Oracle

The window function LEAD OVER is perfect for a problem such as this one. It not only makes for a more readable query than the solution for the other products, LEAD OVER also leads to a more flexible solution because an argument can be passed to it that will determine how many rows ahead it should look (by default 1). Being able to leap ahead more than one row is important in the case of duplicates in the column you are ordering by.

The following example shows how easy it is to use LEAD OVER to look at the salary of the “next” employee hired:

```
select ename, sal, hiredate,
       lead(sal)over(order by hiredate) next_sal
from emp
```

ENAME	SAL	HIREDATE	NEXT_SAL
SMITH	800	17-DEC-80	1600
ALLEN	1600	20-FEB-81	1250
WARD	1250	22-FEB-81	2975
JONES	2975	02-APR-81	2850
BLAKE	2850	01-MAY-81	2450
CLARK	2450	09-JUN-81	1500
TURNER	1500	08-SEP-81	1250
MARTIN	1250	28-SEP-81	5000
KING	5000	17-NOV-81	950
JAMES	950	03-DEC-81	3000
FORD	3000	03-DEC-81	1300
MILLER	1300	23-JAN-82	3000
SCOTT	3000	09-DEC-82	1100
ADAMS	1100	12-JAN-83	

The final step is to return only rows where SAL is less than NEXT_SAL. Because of LEAD OVER’s default range of one row, if there had been duplicates in table EMP, in particular, multiple employees hired on the same date, their SAL would be compared. This may or may not have been what you intended. If your goal is to compare the SAL

of each employee with SAL of the next employee hired, excluding other employees hired on the same day, you can use the following solution as an alternative:

```
select ename, sal, hiredate
  from (
select ename, sal, hiredate,
       lead(sal,cnt-rn+1)over(order by hiredate) next_sal
  from (
select ename,sal,hiredate,
       count(*)over(partition by hiredate) cnt,
       row_number()over(partition by hiredate order by empno) rn
  from emp
   )
   )
 where sal < next_sal
```

The idea behind this solution is to find the distance from the current row to the row it should be compared with. For example, if there are five duplicates, the first of the five needs to leap five rows to get to its correct LEAD OVER row. The value for CNT represents, for each employee with a duplicate HIREDATE, how many duplicates there are in total for their HIREDATE. The value for RN represents a ranking for the employees in DEPTNO 10. The rank is partitioned by HIREDATE so only employees with a HIREDATE that another employee has will have a value greater than one. The ranking is sorted by EMPNO (this is arbitrary). Now that you now how many total duplicates there are and you have a ranking of each duplicate, the distance to the next HIREDATE is simply the total number of duplicates minus the current rank plus one (CNT-RN+1).

See Also

For additional examples of using LEAD OVER in the presence of duplicates (and a more thorough discussion of the technique above): Chapter 8, the section on “Determining the Date Difference Between the Current Record and the Next Record” and Chapter 10, the section on “Finding Differences Between Rows in the Same Group or Partition.”

Shifting Row Values

Problem

You want to return each employee’s name and salary along with the next highest and lowest salaries. If there are no higher or lower salaries, you want the results to wrap (first SAL shows last SAL and vice versa). You want to return the following result set:

ENAME	SAL	FORWARD	REWIND
SMITH	800	950	5000
JAMES	950	1100	800
ADAMS	1100	1250	950

WARD	1250	1250	1100
MARTIN	1250	1300	1250
MILLER	1300	1500	1250
TURNER	1500	1600	1300
ALLEN	1600	2450	1500
CLARK	2450	2850	1600
BLAKE	2850	2975	2450
JONES	2975	3000	2850
SCOTT	3000	3000	2975
FORD	3000	5000	3000
KING	5000	800	3000

Solution

For Oracle users, the window functions LEAD OVER and LAG OVER make this problem easy to solve and the resulting queries very readable. With other RDBMSs you can use scalar subqueries, though ties will present a problem. Because of the problem with ties, the RDBMSs without support for window functions enable only an approximate solution to this problem.

DB2, SQL Server, MySQL, and PostgreSQL

Use a scalar subquery to find next and prior salaries relative to each salary:

```
1 select e.ename, e.sal,
2     coalesce(
3         (select min(sal) from emp d where d.sal > e.sal),
4         (select min(sal) from emp)
5     ) as forward,
6     coalesce(
7         (select max(sal) from emp d where d.sal < e.sal),
8         (select max(sal) from emp)
9     ) as rewind
10    from emp e
11   order by 2
```

Oracle

Use the window functions LAG OVER and LEAD OVER to access prior and next rows relative to the current row:

```
1 select ename,sal,
2     nvl(lead(sal)over(order by sal),min(sal)over()) forward,
3     nvl(lag(sal)over(order by sal),max(sal)over()) rewind
4    from emp
```

Discussion

DB2, SQL Server, MySQL, and PostgreSQL

The scalar subquery solution is not a true solution to the problem. It's an approximation that will fail in the event any two records contain the same value for SAL. It's the best you can do without having window functions available.

Oracle

The window functions LAG OVER and LEAD OVER will (by default and unless otherwise specified) return values from the row before and after the current row, respectively. You define what “before” or “after” means in the ORDER BY portion of the OVER clause. If you examine the solution, the first step is to return the next and prior rows relative to the current row, ordered by SAL:

```
select ename,sal,
       lead(sal)over(order by sal) forward,
       lag(sal)over(order by sal) rewind
from emp
```

ENAME	SAL	FORWARD	REWIND
SMITH	800	950	
JAMES	950	1100	800
ADAMS	1100	1250	950
WARD	1250	1250	1100
MARTIN	1250	1300	1250
MILLER	1300	1500	1250
TURNER	1500	1600	1300
ALLEN	1600	2450	1500
CLARK	2450	2850	1600
BLAKE	2850	2975	2450
JONES	2975	3000	2850
SCOTT	3000	3000	2975
FORD	3000	5000	3000
KING	5000		3000

Notice that REWIND is NULL for employee SMITH and FORWARD is NULL for employee KING; that is because those two employees have the lowest and highest salaries, respectively. The requirement in the problem section should NULL values exist in FORWARD or REWIND is to “wrap” the results meaning that, for the highest SAL, FORWARD should be the value of the lowest SAL in the table, and for the lowest SAL, REWIND should be the value of the highest SAL in the table. The window functions MIN OVER and MAX OVER with no partition or window specified (i.e., an empty parenthesis after the OVER clause) will return the lowest and highest salaries in the table, respectively. The results are shown below:

```
select ename,sal,
       nvl(lead(sal)over(order by sal),min(sal)over()) forward,
       nvl(lag(sal)over(order by sal),max(sal)over()) rewind
from emp
```

ENAME	SAL	FORWARD	REWIND
SMITH	800	950	5000
JAMES	950	1100	800
ADAMS	1100	1250	950
WARD	1250	1250	1100
MARTIN	1250	1300	1250
MILLER	1300	1500	1250

TURNER	1500	1600	1300
ALLEN	1600	2450	1500
CLARK	2450	2850	1600
BLAKE	2850	2975	2450
JONES	2975	3000	2850
SCOTT	3000	3000	2975
FORD	3000	5000	3000
KING	5000	800	3000

Another useful feature of LAG OVER and LEAD OVER is the ability to define how far forward or back you would like to go. In the example for this recipe, you go only one row forward or back. If want to move three rows forward and five rows back, doing so is simple. Just specify the values 3 and 5 as shown below:

```
select ename,sal,
       lead(sal,3)over(order by sal) forward,
       lag(sal,5)over(order by sal) rewind
from emp
```

ENAME	SAL	FORWARD	REWIND
SMITH	800	1250	
JAMES	950	1250	
ADAMS	1100	1300	
WARD	1250	1500	
MARTIN	1250	1600	
MILLER	1300	2450	800
TURNER	1500	2850	950
ALLEN	1600	2975	1100
CLARK	2450	3000	1250
BLAKE	2850	3000	1250
JONES	2975	5000	1300
SCOTT	3000		1500
FORD	3000		1600
KING	5000		2450

Ranking Results

Problem

You want to rank the salaries in table EMP while allowing for ties. You want to return the following result set:

RNK	SAL
1	800
2	950
3	1100
4	1250
4	1250
5	1300
6	1500
7	1600

8	2450
9	2850
10	2975
11	3000
11	3000
12	5000

Solution

Window functions make ranking queries extremely simple. Three window functions are particularly useful for ranking: `DENSE_RANK OVER`, `ROW_NUMBER OVER`, and `RANK OVER`.

DB2, Oracle, and SQL Server

Because you want to allow for ties, use the window function `DENSE_RANK OVER`:

```
1 select dense_rank() over(order by sal) rnk, sal
2   from emp
```

MySQL and PostgreSQL

Until window functions are introduced, use a scalar subquery to rank the salaries:

```
1 select (select count(distinct b.sal)
2         from emp b
3         where b.sal <= a.sal) as rnk,
4        a.sal
5   from emp a
```

Discussion

DB2, Oracle, and SQL Server

The window function `DENSE_RANK OVER` does all the legwork here. In parentheses following the `OVER` keyword you place an `ORDER BY` clause to specify the order in which rows are ranked. The solution uses `ORDER BY SAL`, so rows from `EMP` are ranked in ascending order of salary.

MySQL and PostgreSQL

The output from the scalar subquery solution is similar to that of `DENSE_RANK` because the driving predicate in the scalar subquery is on `SAL`.

Suppressing Duplicates

Problem

You want to find the different job types in table `EMP` but do not want to see duplicates. The result set should be:

JOB

ANALYST
CLERK
MANAGER
PRESIDENT
SALESMAN

Solution

All of the RDBMSs support the keyword `DISTINCT`, and it arguably is the easiest mechanism for suppressing duplicates from the result set. However, this recipe will also cover two additional methods for suppressing duplicates.

DB2, Oracle, and SQL Server

The traditional method of using `DISTINCT` and sometimes `GROUP BY` (as seen next in the MySQL/PostgreSQL solution) certainly works for these RDBMSs. The solution below is an alternative that makes use of the window function `ROW_NUMBER OVER`:

```
1 select job
2   from (
3 select job,
4        row_number()over(partition by job order by job) rn
5   from emp
6  ) x
7  where rn = 1
```

MySQL and PostgreSQL

Use the `DISTINCT` keyword to suppress duplicates from the result set:

```
select distinct job
  from emp
```

Additionally, it is also possible to use `GROUP BY` to suppress duplicates:

```
select job
  from emp
 group by job
```

Discussion

DB2, Oracle, and SQL Server

This solution depends on some outside-the-box thinking about partitioned window functions. By using `PARTITION BY` in the `OVER` clause of `ROW_NUMBER`, you can reset the value returned by `ROW_NUMBER` to 1 whenever a new job is encountered. The results below are from inline view X:

```
select job,
       row_number()over(partition by job order by job) rn
  from emp
```

JOB	RN
ANALYST	1
ANALYST	2
CLERK	1
CLERK	2
CLERK	3
CLERK	4
MANAGER	1
MANAGER	2
MANAGER	3
PRESIDENT	1
SALESMAN	1
SALESMAN	2
SALESMAN	3
SALESMAN	4

Each row is given an increasing, sequential number, and that number is reset to 1 whenever the job changes. To filter out the duplicates, all you must do is keep the rows where RN is 1.

An ORDER BY clause is mandatory when using ROW_NUMBER OVER (except in DB2) but doesn't affect the result. Which job is returned is irrelevant so long as you return one of each job.

MySQL and PostgreSQL

The first solution shows how to use the keyword DISTINCT to suppress duplicates from a result set. Keep in mind that DISTINCT is applied to the whole SELECT list; additional columns can and will change the result set. Consider the difference between the two queries below:

<code>select distinct job from emp</code>	<code>select distinct job, deptno from emp</code>
JOB	JOB DEPTNO
-----	-----
ANALYST	ANALYST 20
CLERK	CLERK 10
MANAGER	CLERK 20
PRESIDENT	CLERK 30
SALESMAN	MANAGER 10
	MANAGER 20
	MANAGER 30
	PRESIDENT 10
	SALESMAN 30

By adding DEPTNO to the SELECT list, what you return is each DISTINCT pair of JOB/DEPTNO values from table EMP.

The second solution uses GROUP BY to suppress duplicates. While using GROUP BY this way is not uncommon, keep in mind that GROUP BY and DISTINCT are

two very different clauses that are not interchangeable. I've included GROUP BY in this solution for completeness, as you will no doubt come across it at some point.

Finding Knight Values

Problem

You want return a result set that contains each employee's name, the department they work in, their salary, the date they were hired, and the salary of the last employee hired, in each department. You want to return the following result set:

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
10	MILLER	1300	23-JAN-1982	1300
10	KING	5000	17-NOV-1981	1300
10	CLARK	2450	09-JUN-1981	1300
20	ADAMS	1100	12-JAN-1983	1100
20	SCOTT	3000	09-DEC-1982	1100
20	FORD	3000	03-DEC-1981	1100
20	JONES	2975	02-APR-1981	1100
20	SMITH	800	17-DEC-1980	1100
30	JAMES	950	03-DEC-1981	950
30	MARTIN	1250	28-SEP-1981	950
30	TURNER	1500	08-SEP-1981	950
30	BLAKE	2850	01-MAY-1981	950
30	WARD	1250	22-FEB-1981	950
30	ALLEN	1600	20-FEB-1981	950

The values in LATEST_SAL are the “Knight values” because the path to find them is analogous to a knight’s path in the game of chess. You determine the result the way a knight determines a new location: by jumping to a row then turning and jumping to a different column (see Figure 11-1). To find the correct values for LATEST_SAL, you must first locate (jump to) the row with the latest HIREDATE in each DEPTNO, and then you select (jump to) the SAL column of that row.

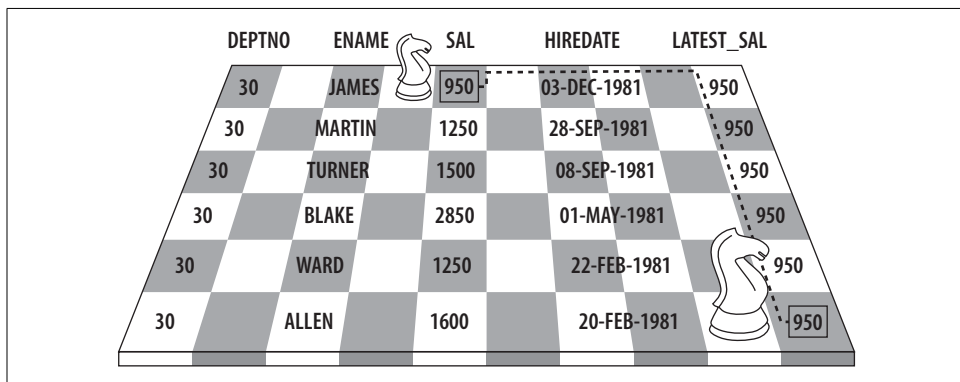


Figure 11-1. A knight value comes from “up and over”



The term “Knight value” was coined by a very clever coworker of mine, Kay Young. After having him review the recipes for correctness I admitted to him that I was stumped and could not come up with a good title. Because you need to initially evaluate one row then “jump” and take a value from another, he came up with the term “Knight value.”

Solution

DB2 and SQL Server

Use a CASE expression in a subquery to return the SAL of the last employee hired in each DEPTNO; for all other salaries, return zero. Use the window function MAX OVER in the outer query to return the non-zero SAL for each employee’s department:

```
1 select deptno,
2     ename,
3     sal,
4     hiredate,
5     max(latest_sal)over(partition by deptno) latest_sal
6   from (
7 select deptno,
8     ename,
9     sal,
10    hiredate,
11    case
12      when hiredate = max(hiredate)over(partition by deptno)
13      then sal else 0
14    end latest_sal
15   from emp
16  ) x
17  order by 1, 4 desc
```

MySQL and PostgreSQL

Use a scalar subquery nested two levels deep. First, find the HIREDATE of the last employee in each DEPTO. Then use the aggregate function MAX (in case there are duplicates) to find the SAL of the last employee hired in each DEPTNO:

```
1 select e.deptno,
2     e.ename,
3     e.sal,
4     e.hiredate,
5     (select max(d.sal)
6      from emp d
7      where d.deptno = e.deptno
8      and d.hiredate =
9          (select max(f.hiredate)
10         from emp f
11         where f.deptno = e.deptno)) as latest_sal
12   from emp e
```

13 order by 1, 4 desc

Oracle

Use the window function `MAX OVER` to return the highest SAL for each DEPTNO. Use the functions `DENSE_RANK` and `LAST`, while ordering by `HIREDATE`, in the `KEEP` clause to return the highest SAL for the latest `HIREDATE` in a given DEPTNO:

```
1 select deptno,
2     ename,
3     sal,
4     hiredate,
5     max(sal)
6     keep(dense_rank last order by hiredate)
7     over(partition by deptno) latest_sal
8 from emp
9 order by 1, 4 desc
```

Discussion

DB2 and SQL Server

The first step is to use the window function `MAX OVER` in a `CASE` expression to find the employee hired last, or most recently, in each DEPTNO. If an employee's `HIREDATE` matches the value returned by `MAX OVER`, then use a `CASE` expression to return that employee's SAL; otherwise return 0. The results of this are shown below:

```
select deptno,
       ename,
       sal,
       hiredate,
       case
         when hiredate = max(hiredate)over(partition by deptno)
         then sal else 0
       end latest_sal
from emp
```

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
10	CLARK	2450	09-JUN-1981	0
10	KING	5000	17-NOV-1981	0
10	MILLER	1300	23-JAN-1982	1300
20	SMITH	800	17-DEC-1980	0
20	ADAMS	1100	12-JAN-1983	1100
20	FORD	3000	03-DEC-1981	0
20	SCOTT	3000	09-DEC-1982	0
20	JONES	2975	02-APR-1981	0
30	ALLEN	1600	20-FEB-1981	0
30	BLAKE	2850	01-MAY-1981	0
30	MARTIN	1250	28-SEP-1981	0
30	JAMES	950	03-DEC-1981	950

```

30 TURNER          1500 08-SEP-1981      0
30 WARD            1250 22-FEB-1981      0

```

Because the value for LATEST_SAL will be either 0 or the SAL of the employee(s) hired most recently, you can wrap the above query in an inline view and use MAX OVER again, but this time to return the greatest non-zero LATEST_SAL for each DEPTNO:

```

select deptno,
       ename,
       sal,
       hiredate,
       max(latest_sal)over(partition by deptno) latest_sal
from (
select deptno,
       ename,
       sal,
       hiredate,
       case
         when hiredate = max(hiredate)over(partition by deptno)
         then sal else 0
       end latest_sal
from emp
) x
order by 1, 4 desc

```

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
10	MILLER	1300	23-JAN-1982	1300
10	KING	5000	17-NOV-1981	1300
10	CLARK	2450	09-JUN-1981	1300
20	ADAMS	1100	12-JAN-1983	1100
20	SCOTT	3000	09-DEC-1982	1100
20	FORD	3000	03-DEC-1981	1100
20	JONES	2975	02-APR-1981	1100
20	SMITH	800	17-DEC-1980	1100
30	JAMES	950	03-DEC-1981	950
30	MARTIN	1250	28-SEP-1981	950
30	TURNER	1500	08-SEP-1981	950
30	BLAKE	2850	01-MAY-1981	950
30	WARD	1250	22-FEB-1981	950
30	ALLEN	1600	20-FEB-1981	950

MySQL and PostgreSQL

The first step is to use a scalar subquery to find the HIREDATE of the last employee hired in each DEPTNO:

```

select e.deptno,
       e.ename,
       e.sal,
       e.hiredate,
       (select max(f.hiredate)
        from emp f
        where f.deptno = e.deptno) as last_hire
from emp e

```

order by 1, 4 desc

DEPTNO	ENAME	SAL	HIREDATE	LAST_HIRE
10	MILLER	1300	23-JAN-1982	23-JAN-1982
10	KING	5000	17-NOV-1981	23-JAN-1982
10	CLARK	2450	09-JUN-1981	23-JAN-1982
20	ADAMS	1100	12-JAN-1983	12-JAN-1983
20	SCOTT	3000	09-DEC-1982	12-JAN-1983
20	FORD	3000	03-DEC-1981	12-JAN-1983
20	JONES	2975	02-APR-1981	12-JAN-1983
20	SMITH	800	17-DEC-1980	12-JAN-1983
30	JAMES	950	03-DEC-1981	03-DEC-1981
30	MARTIN	1250	28-SEP-1981	03-DEC-1981
30	TURNER	1500	08-SEP-1981	03-DEC-1981
30	BLAKE	2850	01-MAY-1981	03-DEC-1981
30	WARD	1250	22-FEB-1981	03-DEC-1981
30	ALLEN	1600	20-FEB-1981	03-DEC-1981

The next step is to find the SAL for the employee(s) in each DEPTNO hired on LAST_HIRE. Use the aggregate function MAX to keep the highest (if there are multiple employees hired on the same day):

```
select e.deptno,
       e.ename,
       e.sal,
       e.hiredate,
       (select max(d.sal)
        from emp d
        where d.deptno = e.deptno
          and d.hiredate =
             (select max(f.hiredate)
              from emp f
              where f.deptno = e.deptno)) as latest_sal
from emp e
order by 1, 4 desc
```

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
10	MILLER	1300	23-JAN-1982	1300
10	KING	5000	17-NOV-1981	1300
10	CLARK	2450	09-JUN-1981	1300
20	ADAMS	1100	12-JAN-1983	1100
20	SCOTT	3000	09-DEC-1982	1100
20	FORD	3000	03-DEC-1981	1100
20	JONES	2975	02-APR-1981	1100
20	SMITH	800	17-DEC-1980	1100
30	JAMES	950	03-DEC-1981	950
30	MARTIN	1250	28-SEP-1981	950
30	TURNER	1500	08-SEP-1981	950
30	BLAKE	2850	01-MAY-1981	950
30	WARD	1250	22-FEB-1981	950
30	ALLEN	1600	20-FEB-1981	950

Oracle

Users on Oracle8i Database can use the DB2 solution. For users on Oracle9i Database and later, you can use the solution presented below. The key to the Oracle solution is to take advantage of the KEEP clause. The KEEP clause allows you to rank the rows returned by a group/partition and work with the first or last row in the group. Consider what the solution looks like without KEEP:

```
select deptno,  
       ename,  
       sal,  
       hiredate,  
       max(sal) over(partition by deptno) latest_sal  
from emp  
order by 1, 4 desc
```

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
10	MILLER	1300	23-JAN-1982	5000
10	KING	5000	17-NOV-1981	5000
10	CLARK	2450	09-JUN-1981	5000
20	ADAMS	1100	12-JAN-1983	3000
20	SCOTT	3000	09-DEC-1982	3000
20	FORD	3000	03-DEC-1981	3000
20	JONES	2975	02-APR-1981	3000
20	SMITH	800	17-DEC-1980	3000
30	JAMES	950	03-DEC-1981	2850
30	MARTIN	1250	28-SEP-1981	2850
30	TURNER	1500	08-SEP-1981	2850
30	BLAKE	2850	01-MAY-1981	2850
30	WARD	1250	22-FEB-1981	2850
30	ALLEN	1600	20-FEB-1981	2850

Rather than returning the SAL of the latest employee hired, MAX OVER without KEEP simply returns the highest salary in each DEPTNO. KEEP, in this recipe, allows you to order the salaries by HIREDATE in each DEPTNO by specifying ORDER BY HIREDATE. Then, the function DENSE_RANK assigns a rank to each HIREDATE in ascending order. Finally, the function LAST determines which row to apply the aggregate function to: the “last” row based on the ranking of DENSE_RANK. In this case, the aggregate function MAX is applied to the SAL column for the row with the “last” HIREDATE. In essence, keep the SAL of the HIREDATE ranked last in each DEPTNO.

You are ranking the rows in each DEPTNO based on one column (HIREDATE), but then applying the aggregation (MAX) on another column (SAL). This ability to rank in one dimension and aggregate over another is convenient as it allows you to avoid extra joins and inline views as are used in the other solutions. Finally, by adding the OVER clause after the KEEP clause you can return the SAL “kept” by KEEP for each row in the partition.

Alternatively, you can order by HIREDATE in descending order and “keep” the first SAL. Compare the two queries below, which return the same result set:

```
select deptno,
       ename,
       sal,
       hiredate,
       max(sal)
         keep(dense_rank last order by hiredate)
         over(partition by deptno) latest_sal
from emp
order by 1, 4 desc
```

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
10	MILLER	1300	23-JAN-1982	1300
10	KING	5000	17-NOV-1981	1300
10	CLARK	2450	09-JUN-1981	1300
20	ADAMS	1100	12-JAN-1983	1100
20	SCOTT	3000	09-DEC-1982	1100
20	FORD	3000	03-DEC-1981	1100
20	JONES	2975	02-APR-1981	1100
20	SMITH	800	17-DEC-1980	1100
30	JAMES	950	03-DEC-1981	950
30	MARTIN	1250	28-SEP-1981	950
30	TURNER	1500	08-SEP-1981	950
30	BLAKE	2850	01-MAY-1981	950
30	WARD	1250	22-FEB-1981	950
30	ALLEN	1600	20-FEB-1981	950

```
select deptno,
       ename,
       sal,
       hiredate,
       max(sal)
         keep(dense_rank first order by hiredate desc)
         over(partition by deptno) latest_sal
from emp
order by 1, 4 desc
```

DEPTNO	ENAME	SAL	HIREDATE	LATEST_SAL
10	MILLER	1300	23-JAN-1982	1300
10	KING	5000	17-NOV-1981	1300
10	CLARK	2450	09-JUN-1981	1300
20	ADAMS	1100	12-JAN-1983	1100
20	SCOTT	3000	09-DEC-1982	1100
20	FORD	3000	03-DEC-1981	1100
20	JONES	2975	02-APR-1981	1100
20	SMITH	800	17-DEC-1980	1100
30	JAMES	950	03-DEC-1981	950
30	MARTIN	1250	28-SEP-1981	950
30	TURNER	1500	08-SEP-1981	950

30 BLAKE	2850 01-MAY-1981	950
30 WARD	1250 22-FEB-1981	950
30 ALLEN	1600 20-FEB-1981	950

Generating Simple Forecasts

Problem

Based on current data, you want to return addition rows and columns representing future actions. For example, consider the following result set:

ID	ORDER_DATE	PROCESS_DATE
1	25-SEP-2005	27-SEP-2005
2	26-SEP-2005	28-SEP-2005
3	27-SEP-2005	29-SEP-2005

You want to return three rows per row returned in your result set (each row plus two additional rows for each order). Along with the extra rows you would like to return two additional columns providing dates for expected order processing.

From the result set above you can see that an order takes two days to process. For the purposes of this example, let's say the next step after processing is verification, and the last step is shipment. Verification occurs one day after processing and shipment occurs one day after verification. You want to return a result set expressing the whole procedure. Ultimately you want to transform the result set above to the following result set:

ID	ORDER_DATE	PROCESS_DATE	VERIFIED	SHIPPED
1	25-SEP-2005	27-SEP-2005		
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	29-SEP-2005
2	26-SEP-2005	28-SEP-2005		
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	30-SEP-2005
3	27-SEP-2005	29-SEP-2005		
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	01-OCT-2005

Solution

The key is to use a Cartesian product to generate two additional rows for each order then simply use CASE expressions to create the required column values.

DB2 and SQL Server

Use the recursive WITH clause to generate rows needed for your Cartesian product. The DB2 and SQL Server solutions are identical except for the function used to

retrieve the current date. DB2 uses CURRENT_DATE and SQL Server uses GET-DATE. The SQL Server solution is shown below:

```
1 with nrows(n) as (  
2 select 1 from t1 union all  
3 select n+1 from nrows where n+1 <= 3  
4 )  
5 select id,  
6     order_date,  
7     process_date,  
8     case when nrows.n >= 2  
9         then process_date+1  
10        else null  
11     end as verified,  
12     case when nrows.n = 3  
13         then process_date+2  
14         else null  
15     end as shipped  
16 from (  
17 select nrows.n id,  
18        getdate()+nrows.n as order_date,  
19        getdate()+nrows.n+2 as process_date  
20 from nrows  
21 ) orders, nrows  
22 order by 1
```

Oracle

Use the hierarchical CONNECT BY clause to generate the three rows needed for the Cartesian product. Use the WITH clause to allow you to reuse the results returned by CONNECT BY without having to call it again:

```
1 with nrows as (  
2 select level n  
3 from dual  
4 connect by level <= 3  
5 )  
6 select id,  
7     order_date,  
8     process_date,  
9     case when nrows.n >= 2  
10        then process_date+1  
11        else null  
12     end as verified,  
13     case when nrows.n = 3  
14        then process_date+2  
15        else null  
16     end as shipped  
17 from (  
18 select nrows.n id,  
19        sysdate+nrows.n as order_date,  
20        sysdate+nrows.n+2 as process_date  
21 from nrows  
22 ) orders, nrows
```

PostgreSQL

You can create a Cartesian product many different ways; this solution uses the PostgreSQL function `GENERATE_SERIES`:

```
1  select id,
2     order_date,
3     process_date,
4     case when gs.n >= 2
5         then process_date+1
6         else null
7     end as verified,
8     case when gs.n = 3
9         then process_date+2
10        else null
11    end as shipped
12  from (
13  select gs.id,
14         current_date+gs.id  as order_date,
15         current_date+gs.id+2 as process_date
16  from generate_series(1,3) gs (id)
17       ) orders,
18       generate_series(1,3)gs(n)
```

MySQL

MySQL does not support a function for automatic row generation.

Discussion

DB2 and SQL Server

The result set presented in the problem section is returned via inline view `ORDERS` and is shown below:

```
with nrows(n) as (
select 1 from t1 union all
select n+1 from nrows where n+1 <= 3
)
select nrows.n id,
       getdate()+nrows.n  as order_date,
       getdate()+nrows.n+2 as process_date
from nrows
```

ID	ORDER_DATE	PROCESS_DATE
1	25-SEP-2005	27-SEP-2005
2	26-SEP-2005	28-SEP-2005
3	27-SEP-2005	29-SEP-2005

The query above simply uses the `WITH` clause to make up three rows representing the orders you must process. `NROWS` returns the values 1, 2, and 3, and those

numbers are added to GETDATE (CURRENT_DATE for DB2) to represent the dates of the orders. Because the problem section states that processing time takes two days, the query above also adds two days to the ORDER_DATE (adds the value returned by NROWS to GETDATE, then adds two more days).

Now that you have your base result set, the next step is to create a Cartesian product because the requirement is to return three rows for each order. Use NROWS to create a Cartesian product to return three rows for each order:

```
with nrows(n) as (
  select 1 from t1 union all
  select n+1 from nrows where n+1 <= 3
)
select nrows.n,
       orders.*
  from (
select nrows.n id,
       getdate()+nrows.n as order_date,
       getdate()+nrows.n+2 as process_date
  from nrows
) orders, nrows
order by 2,1
```

N	ID	ORDER_DATE	PROCESS_DATE
1	1	25-SEP-2005	27-SEP-2005
2	1	25-SEP-2005	27-SEP-2005
3	1	25-SEP-2005	27-SEP-2005
1	2	26-SEP-2005	28-SEP-2005
2	2	26-SEP-2005	28-SEP-2005
3	2	26-SEP-2005	28-SEP-2005
1	3	27-SEP-2005	29-SEP-2005
2	3	27-SEP-2005	29-SEP-2005
3	3	27-SEP-2005	29-SEP-2005

Now that you have three rows for each order, simply use a CASE expression to create the addition column values to represent the status of verification and shipment.

The first row for each order should have a NULL value for VERIFIED and SHIPPED. The second row for each order should have a NULL value for SHIPPED. The third row for each order should have non-NULL values for each column. The final result set is shown below:

```
with nrows(n) as (
  select 1 from t1 union all
  select n+1 from nrows where n+1 <= 3
)
select id,
       order_date,
       process_date,
       case when nrows.n >= 2
           then process_date+1
           else null
```

```

end as verified,
case when nrows.n = 3
    then process_date+2
    else null
end as shipped
from (
select nrows.n id,
    getdate()+nrows.n as order_date,
    getdate()+nrows.n+2 as process_date
from nrows
) orders, nrows
order by 1

```

ID	ORDER_DATE	PROCESS_DATE	VERIFIED	SHIPPED
1	25-SEP-2005	27-SEP-2005		
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	29-SEP-2005
2	26-SEP-2005	28-SEP-2005		
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	30-SEP-2005
3	27-SEP-2005	29-SEP-2005		
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	01-OCT-2005

The final result set expresses the complete order process from the day the order was received to the day it should be shipped.

Oracle

The result set presented in the problem section is returned via inline view ORDERS and is shown below:

```

with nrows as (
select level n
from dual
connect by level <= 3
)
select nrows.n id,
    sysdate+nrows.n order_date,
    sysdate+nrows.n+2 process_date
from nrows

```

ID	ORDER_DATE	PROCESS_DATE
1	25-SEP-2005	27-SEP-2005
2	26-SEP-2005	28-SEP-2005
3	27-SEP-2005	29-SEP-2005

The query above simply uses CONNECT BY to make up three rows representing the orders you must process. Use the WITH clause to refer to the rows returned by CONNECT BY as NROWS.N. CONNECT BY returns the values 1, 2, and 3, and

those numbers are added to SYSDATE to represent the dates of the orders. Since the problem section states that processing time takes two days, the query above also adds two days to the ORDER_DATE (adds the value returned by GENERATE_SERIES to SYSDATE, then adds two more days).

Now that you have your base result set, the next step is to create a Cartesian product because the requirement is to return three rows for each order. Use NROWS to create a Cartesian product to return three rows for each order:

```
with nrows as (
  select level n
    from dual
   connect by level <= 3
)
select nrows.n,
       orders.*
  from (
select nrows.n id,
       sysdate+nrows.n order_date,
       sysdate+nrows.n+2 process_date
  from nrows
) orders, nrows
```

N	ID	ORDER_DATE	PROCESS_DATE
1	1	25-SEP-2005	27-SEP-2005
2	1	25-SEP-2005	27-SEP-2005
3	1	25-SEP-2005	27-SEP-2005
1	2	26-SEP-2005	28-SEP-2005
2	2	26-SEP-2005	28-SEP-2005
3	2	26-SEP-2005	28-SEP-2005
1	3	27-SEP-2005	29-SEP-2005
2	3	27-SEP-2005	29-SEP-2005
3	3	27-SEP-2005	29-SEP-2005

Now that you have three rows for each order, simply use a CASE expression to create the addition column values to represent the status of verification and shipment.

The first row for each order should have a NULL value for VERIFIED and SHIPPED. The second row for each order should have a NULL value for SHIPPED. The third row for each order should have non-NULL values for each column. The final result set is shown below:

```
with nrows as (
  select level n
    from dual
   connect by level <= 3
)
select id,
       order_date,
       process_date,
       case when nrows.n >= 2
           then process_date+1
```

```

        else null
    end as verified,
    case when nrows.n = 3
        then process_date+2
        else null
    end as shipped
from (
select nrows.n id,
       sysdate+nrows.n order_date,
       sysdate+nrows.n+2 process_date
from   nrows
) orders, nrows

```

ID	ORDER_DATE	PROCESS_DATE	VERIFIED	SHIPPED
1	25-SEP-2005	27-SEP-2005		
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	29-SEP-2005
2	26-SEP-2005	28-SEP-2005		
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	30-SEP-2005
3	27-SEP-2005	29-SEP-2005		
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	01-OCT-2005

The final result set expresses the complete order process from the day the order was received to the day it should be shipped.

PostgreSQL

The result set presented in the problem section is returned via inline view ORDERS and is shown below:

```

select gs.id,
       current_date+gs.id as order_date,
       current_date+gs.id+2 as process_date
from   generate_series(1,3) gs (id)

```

ID	ORDER_DATE	PROCESS_DATE
1	25-SEP-2005	27-SEP-2005
2	26-SEP-2005	28-SEP-2005
3	27-SEP-2005	29-SEP-2005

The query above simply uses the GENERATE_SERIES function to make up three rows representing the orders you must process. GENERATE_SERIES returns the values 1, 2, and 3, and those numbers are added to CURRENT_DATE to represent the dates of the orders. Since the problem section states that processing time takes two days, the query above also adds two days to the ORDER_DATE (adds the value returned by GENERATE_SERIES to CURRENT_DATE, then adds two more days).

Now that you have your base result set, the next step is to create a Cartesian product because the requirement is to return three rows for each order. Use the GENERATE_SERIES function to create a Cartesian product to return three rows for each order:

```
select gs.n,
       orders.*
  from (
select gs.id,
       current_date+gs.id  as order_date,
       current_date+gs.id+2 as process_date
  from generate_series(1,3) gs (id)
       ) orders,
       generate_series(1,3)gs(n)
```

N	ID	ORDER_DATE	PROCESS_DATE
1	1	25-SEP-2005	27-SEP-2005
2	1	25-SEP-2005	27-SEP-2005
3	1	25-SEP-2005	27-SEP-2005
1	2	26-SEP-2005	28-SEP-2005
2	2	26-SEP-2005	28-SEP-2005
3	2	26-SEP-2005	28-SEP-2005
1	3	27-SEP-2005	29-SEP-2005
2	3	27-SEP-2005	29-SEP-2005
3	3	27-SEP-2005	29-SEP-2005

Now that you have three rows for each order, simply use a CASE expression to create the additional column values to represent the status of verification and shipment.

The first row for each order should have a NULL value for VERIFIED and SHIPPED. The second row for each order should have a NULL value for SHIPPED. The third row for each order should have non-NULL values for each column. The final result set is shown below:

```
select id,
       order_date,
       process_date,
       case when gs.n >= 2
            then process_date+1
            else null
       end as verified,
       case when gs.n = 3
            then process_date+2
            else null
       end as shipped
  from (
select gs.id,
       current_date+gs.id  as order_date,
       current_date+gs.id+2 as process_date
  from generate_series(1,3) gs(id)
       ) orders,
       generate_series(1,3)gs(n)
```

ID	ORDER_DATE	PROCESS_DATE	VERIFIED	SHIPPED
1	25-SEP-2005	27-SEP-2005		
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	
1	25-SEP-2005	27-SEP-2005	28-SEP-2005	29-SEP-2005
2	26-SEP-2005	28-SEP-2005		
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	
2	26-SEP-2005	28-SEP-2005	29-SEP-2005	30-SEP-2005
3	27-SEP-2005	29-SEP-2005		
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	
3	27-SEP-2005	29-SEP-2005	30-SEP-2005	01-OCT-2005

The final result set expresses the complete order process from the day the order was received to the day it should be shipped.