Lab 7. Working with Numbers

This lab focuses on common operations involving numbers, including numeric computations. While SQL is not typically considered the first choice for complex computations, it is very efficient for day-to-day numeric chores.

TIP
Some recipes in this lab make use of aggregate functions and the GROUP BY clause.

7.1. Computing an Average

PROBLEM

You want to compute the average value in a column, either for all rows in a table or for some subset of rows. For example, you might want to find the average salary for all employees as well as the average salary for each department.

SOLUTION

When computing the average of all employee salaries, simply apply the AVG function to the column containing those salaries. By excluding a WHERE clause, the average is computed against all non-NULL values:

```
  select avg(sal) as avg_sal
    from emp

    AVG_SAL
 ----------
 2073.21429
```

To compute the average salary for each department, use the GROUP BY clause to create a group corresponding to each department:

```
 select deptno, avg(sal) as avg_sal
   from emp
  group by deptno

    DEPTNO     AVG_SAL
----------  ----------
        10  2916.66667
        20        2175
        30  1566.66667
```

DISCUSSION

When finding an average where the whole table is the group or window, simply apply the AVG function to the column you are interested in without using the GROUP BY clause. It is important to realize that the function AVG ignores NULLs. The effect of NULL values being ignored can be seen here:

```
create table t2(sal integer)
insert into t2 values (10)
insert into t2 values (20)
insert into t2 values (null)
select avg(sal)    select distinct 30/2
  from t2            from t2

  AVG(SAL)              30/2
----------         ----------
        15                 15


select avg(coalesce(sal,0))    select distinct 30/3
  from t2                        from t2

AVG(COALESCE(SAL,0))                30/3
--------------------           ----------
                  10                   10
```

The COALESCE function will return the first non-NULL value found in the list of values that you pass. When NULL SAL values are converted to zero, the average changes. When invoking aggregate functions, always give thought to how you want NULLs handled.

The second part of the solution uses GROUP BY (line 3) to divide employee records into groups based on department affiliation. GROUP BY automatically causes aggregate functions such as AVG to execute and return a result for each group. In this example, AVG would execute once for each department-based group of employee records.

It is not necessary, by the way, to include GROUP BY columns in your select list. For example:

```
select avg(sal)
  from emp
 group by deptno

  AVG(SAL)
----------
2916.66667
      2175
1566.66667
```

You are still grouping by DEPTNO even though it is not in the SELECT clause. Including the column you are grouping by in the SELECT clause often improves readability, but is not mandatory. It is mandatory, however, to avoid placing columns in your SELECT list that are not also in your GROUP BY clause.

7.2. Finding the Min/Max Value in a Column

PROBLEM

You want to find the highest and lowest values in a given column. For example, you want to find the highest and lowest salaries for all employees, as well as the highest and lowest salaries for each department.

SOLUTION

When searching for the lowest and highest salaries for all employees, simply use the functions MIN and MAX, respectively:

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

   MIN_SAL     MAX_SAL
----------  ----------
       800        5000
```

When searching for the lowest and highest salaries for each department, use the functions MIN and MAX with the GROUP BY clause:

```
select deptno, min(sal) as min_sal, max(sal) as max_sal
from emp
group by deptno

    DEPTNO     MIN_SAL     MAX_SAL
----------  ----------  ----------
        10        1300        5000
        20         800        3000
        30         950        2850
```

DISCUSSION

When searching for the highest or lowest values, and in cases where the whole table is the group or window, simply apply the MIN or MAX function to the column you are interested in without using the GROUP BY clause.

Remember that the MIN and MAX functions ignore NULLs, and that you can have NULL groups as well as NULL values for columns in a group. The following are examples that ultimately lead to a query using GROUP BY that returns NULL values for two groups (DEPTNO 10 and 20):

```
select deptno, comm
  from emp
 where deptno in (10,30)
 order by 1


    DEPTNO       COMM
 ---------- ----------
         10
         10
         10
         30        300
         30        500
         30
         30          0
         30       1300
         30


select min(comm), max(comm)
  from emp

 MIN(COMM)   MAX(COMM)
----------  ----------
         0        1300


select deptno, min(comm), max(comm)
  from emp
 group by deptno

    DEPTNO  MIN(COMM)   MAX(COMM)
 ---------- ----------  ----------
         10
         20
         30          0        1300
```

Remember, as Appendix A points out, even if nothing other than aggregate functions are listed in the SELECT clause, you can still group by other columns in the table; for example:

```
select min(comm), max(comm)
  from emp
 group by deptno

 MIN(COMM)   MAX(COMM)
----------  ----------
         0        1300
```

Here you are still grouping by DEPTNO even though it is not in the SELECT clause. Including the column you are grouping by in the SELECT clause often improves readability, but is not mandatory. It is mandatory, however, that any column in the SELECT list of a GROUP BY query also be listed in the GROUP BY clause.

7.3. Summing the Values in a Column

PROBLEM

You want to compute the sum of all values, such as all employee salaries, in a column.

SOLUTION

When computing a sum where the whole table is the group or window, simply apply the SUM function to the columns you are interested in without using the GROUP BY clause:

```
 select sum(sal)
  from emp

  SUM(SAL)
----------
     29025
```

When creating multiple groups or windows of data, use the SUM function with the GROUP BY clause. The following example sums employee salaries by department:

```
1 select deptno, sum(sal) as total_for_dept
2   from emp
3  group by deptno


    DEPTNO  TOTAL_FOR_DEPT
----------  --------------
        10            8750
        20           10875
        30            9400
```

DISCUSSION

When searching for the sum of all salaries for each department, you are creating groups or “windows” of data. Each employee’s salary is added together to produce a total for his respective department. This is an example of aggregation in SQL because detailed information, such as each individual employee’s salary, is not the focus; the focus is the end result for each department. It is

important to note that the SUM function will ignore NULLs, but you can have NULL groups, which can be seen here. DEPTNO 10 does not have any employees who earn a commission, thus grouping by DEPTNO 10 while attempting to SUM the values in COMM will result in a group with a NULL value returned by SUM:

```
select deptno, comm
  from emp
 where deptno in (10,30)
 order by 1

    DEPTNO       COMM
---------- ----------
        10
        10
        10
        30        300
        30        500
        30
        30          0
        30       1300
        30


select sum(comm)
  from emp

 SUM(COMM)
----------
      2100

select deptno, sum(comm)
  from emp
 where deptno in (10,30)
 group by deptno

    DEPTNO   SUM(COMM)
----------  ----------
        10
        30        2100
```

7.4. Counting Rows in a Table

PROBLEM

You want to count the number of rows in a table, or you wish to count the number of values in a column. For example, you want to find the total number of employees as well as the number of

employees in each department.

SOLUTION

When counting rows where the whole table is the group or window, simply use the COUNT function along with the “*” character:

```
 select count(*)
  from emp

  COUNT(*)
----------
        14
```

When creating multiple groups, or windows of data, use the COUNT function with the GROUP BY clause:

```
select deptno, count(*)
  from emp
 group by deptno

    DEPTNO    COUNT(*)
----------  ----------
        10           3
        20           5
        30           6
```

DISCUSSION

When counting the number of employees for each department, you are creating groups or “windows” of data. Each employee found increments the count by one to produce a total for her respective department. This is an example of aggregation in SQL because detailed information, such as each individual employee’s salary or job, is not the focus; the focus is the end result for each department. It is important to note that the COUNT function will ignore NULLs when passed a column name as an argument, but will include NULLs when passed the “*” character or any constant; consider:

```
select deptno, comm
  from emp

    DEPTNO        COMM
----------  ----------
        20
        30         300
        30         500
        20
        30        1300
        30
        10
        20
        10
        30           0
        20
        30
        20
        10


select count(*), count(deptno), count(comm), count('hello')
  from emp

  COUNT(*)  COUNT(DEPTNO)   COUNT(COMM)  COUNT('HELLO')
----------  -------------   -----------  --------------
        14             14             4              14



select deptno, count(*), count(comm), count('hello')
  from emp
 group by deptno

     DEPTNO   COUNT(*)  COUNT(COMM)  COUNT('HELLO')
 ---------- ----------  -----------  --------------
         10          3            0               3
         20          5            0               5
         30          6            4               6
```

If all rows are null for the column passed to COUNT or if the table is empty, COUNT will return zero. It should also be noted that, even if nothing other than aggregate functions are specified in the SELECT clause, you can still group by other columns in the table; for example:

```
select count(*)
  from emp
 group by deptno

   COUNT(*)
 ----------
          3
          5
          6
```

Notice that you are still grouping by DEPTNO even though it is not in the SELECT clause. Including the column you are grouping by in the SELECT clause often improves readability, but is not mandatory. If you do include it (in the SELECT list), it is mandatory that is it listed in the GROUP BY clause.

7.5. Counting Values in a Column

PROBLEM

You wish to count the number of non-NULL values in a column. For example, you'd like to find out how many employees are on commission.

SOLUTION

Count the number of non-NULL values in the EMP table's COMM column:

```
select count(comm)
  from emp

COUNT(COMM)
-----------
          4
```

DISCUSSION

When you "count star," as in COUNT(*), what you are really counting is rows (regardless of actual value, which is why rows containing NULL and non-NULL values are counted). But when you COUNT a column, you are counting the number of non-NULL values in that column. The previous recipe's discussion touches on this distinction. In this solution, COUNT(COMM) returns the number of non-NULL values in the COMM column. Since only commissioned employees have commissions, the result of COUNT(COMM) is the number of such employees.

7.6. Generating a Running Total

PROBLEM

You want to calculate a running total of values in a column.

SOLUTION

As an example, the following solutions show how to compute a running total of salaries for all employees. For readability, results are ordered by SAL whenever possible so that you can easily eyeball the progression of the running total.

DB2 and Oracle
Use the windowing version of the function SUM to compute a running total:

```
select ename, sal,
       sum(sal) over (order by sal,empno) as running_total
  from emp
  order by 2
```

```
ENAME              SAL  RUNNING_TOTAL
----------  ----------  -------------
SMITH              800            800
JAMES              950           1750
ADAMS             1100           2850
WARD              1250           4100
MARTIN            1250           5350
MILLER            1300           6650
TURNER            1500           8150
ALLEN             1600           9750
CLARK             2450          12200
BLAKE             2850          15050
JONES             2975          18025
SCOTT             3000          21025
FORD              3000          24025
KING              5000          29025
```

MySQL, PostgreSQL, and SQL Server
Use a scalar subquery to compute a running total (without the use of a window function such as SUM OVER, you cannot easily order the result set by SAL as in the DB2 and Oracle solution). Ultimately, the running total is correct (the final value is the same as the above recipe), but the intermediate values differ due to the lack of ordering:

```
select e.ename, e.sal,
       (select sum(d.sal) from emp d
         where d.empno <= e.empno) as running_total
  from emp e
 order by 3
```

ENAME	SAL	RUNNING_TOTAL
SMITH	800	800
ALLEN	1600	2400
WARD	1250	3650
JONES	2975	6625
MARTIN	1250	7875
BLAKE	2850	10725
CLARK	2450	13175
SCOTT	3000	16175
KING	5000	21175
TURNER	1500	22675
ADAMS	1100	23775
JAMES	950	24725
FORD	3000	27725
MILLER	1300	29025

DISCUSSION

Generating a running total is one of the tasks made simple by the new ANSI windowing functions. For DBMSs that do not yet support these windowing functions, a scalar subquery (joining on a field with unique values) is required.

DB2 and Oracle

The windowing function SUM OVER makes generating a running total a simple task. The ORDER BY clause in the solution includes not only the SAL column, but also the EMPNO column (which is the primary key) to avoid duplicate values in the running total. The column RUNNING_TOTAL2 in the following example illustrates the problem that you might otherwise have with duplicates:

```
select empno, sal,
       sum(sal)over(order by sal,empno) as running_total1,
       sum(sal)over(order by sal) as running_total2
  from emp
 order by 2
```

ENAME	SAL	RUNNING_TOTAL1	RUNNING_TOTAL2
SMITH	800	800	800
JAMES	950	1750	1750
ADAMS	1100	2850	2850
WARD	1250	4100	5350
MARTIN	1250	5350	5350
MILLER	1300	6650	6650
TURNER	1500	8150	8150
ALLEN	1600	9750	9750
CLARK	2450	12200	12200
BLAKE	2850	15050	15050
JONES	2975	18025	18025
SCOTT	3000	21025	24025
FORD	3000	24025	24025
KING	5000	29025	29025

The values in RUNNING_TOTAL2 for WARD, MARTIN, SCOTT, and FORD are incorrect. Their salaries occur more than once, and those duplicates are summed together and added to the running total. This is why EMPNO (which is unique) is needed to produce the (correct) results that you see in RUNNING_TOTAL1. Consider this: for ADAMS you see 2850 for RUNNING_TOTAL1 and RUNNING_TOTAL2. Add WARD's salary of 1250 to 2850 and you get 4100, yet RUNNING_TOTAL2 returns 5350. Why? Since WARD and MARTIN have the same SAL, their two 1250 salaries are added together to yield 2500, which is then added to 2850 to arrive at 5350 for both WARD and MARTIN. By specifying a combination of columns to order by that cannot result in duplicate values (e.g., any combination of SAL and EMPNO is unique), you ensure the correct progression of the running total.

MySQL, PostgreSQL, and SQL Server

Until windowing functions are fully supported for these DBMSs, you can use a scalar subquery to compute a running total. You must join on a column with unique values; otherwise the running total will have incorrect values in the event that duplicate salaries exist. The key to this recipe's solution is the join on D.EMPNO to E. EMPNO, which returns (sums) every D.SAL where D.EMPNO is less than or equal E.EMPNO. This can be understood easily by rewriting the scalar subquery as a join for a handful of the employees:

```
select e.ename as ename1, e.empno as empno1, e.sal as sal1,
         d.ename as ename2, d.empno as empno2, d.sal as sal2
  from emp e, emp d
 where d.empno <= e.empno
   and e.empno = 7566

ENAME           EMPNO1        SAL1  ENAME           EMPNO2        SAL2
----------  ----------  ----------  ----------  ----------  ----------
JONES             7566        2975  SMITH             7369         800
JONES             7566        2975  ALLEN             7499        1600
JONES             7566        2975  WARD              7521        1250
JONES             7566        2975  JONES             7566        2975
```

Every value in EMPNO2 is compared against every value in EMPNO1. For every row where the value in EMPNO2 is less than or equal to the value in EMPNO1, the value in SAL2 is included in the sum. In this snippet, the EMPNO values for employees Smith, Allen, Ward, and Jones are compared against the EMPNO of Jones. Since all four employees' EMPNOs meet the condition of being less than or equal to Jones' EMPNO, those salaries are summed. Any employee whose EMPNO is greater than Jones' is not included in the SUM (in this snippet). The way the full query works is by summing all the salaries where the corresponding EMPNO is less than or equal to 7934 (Miller's EMPNO), which is the highest in the table.

7.7. Generating a Running Product

PROBLEM

You want to compute a running product on a numeric column. The operation is similar to "Calculating a Running Total," but using multiplication instead of addition.

SOLUTION

By way of example, the solutions all compute running products of employee salaries. While a running product of salaries may not be all that useful, the technique can easily be applied to other, more useful domains.

DB2 and Oracle
Use the windowing function SUM OVER and take advantage of the fact that you can simulate multiplication by adding logarithms:

```
 select empno,ename,sal,
        exp(sum(ln(sal))over(order by sal,empno)) as running_prod
   from emp
  where deptno = 10

EMPNO ENAME          SAL             RUNNING_PROD
----- ----------    ----    --------------------
 7934 MILLER        1300                    1300
 7782 CLARK         2450                 3185000
 7839 KING          5000              15925000000
```

It is not valid in SQL to compute logarithms of values less than or equal to zero. If you have such values in your tables you need to avoid passing those invalid values to SQL's LN function. Precautions against invalid values and NULLs are not provided in this solution for the sake of readability, but you should consider whether to place such precautions in production code that you write. If you absolutely must work with negative and zero values, then this solution may not work for you.

An alternative, Oracle-only solution is to use the MODEL clause that became available in Oracle Database 10g. In the following example, each SAL is returned as a negative number to show that negative values will not cause a problem for the running product:

```
select empno, ename, sal, tmp as running_prod
  from (
select empno,ename,-sal as sal
  from emp
 where deptno=10
       )
model
  dimension by(row_number()over(order by sal desc) rn )
  measures(sal, 0 tmp, empno, ename)
  rules (
    tmp[any] = case when sal[cv()-1] is null then sal[cv()]
                    else tmp[cv()-1]*sal[cv()]
                end
  )

EMPNO ENAME       SAL         RUNNING_PROD
----- ---------- ---- --------------------
 7934 MILLER     -1300               -1300
 7782 CLARK      -2450             3185000
 7839 KING       -5000        -15925000000
```

MySQL, PostgreSQL, and SQL Server
You still use the approach of summing logarithms, but these platforms do not support windowing functions, so use a scalar subquery instead:

```
select e.empno,e.ename,e.sal,
       (select exp(sum(ln(d.sal)))
          from emp d
         where d.empno <= e.empno
           and e.deptno=d.deptno) as running_prod
from emp e
where e.deptno=10

EMPNO  ENAME         SAL             RUNNING_PROD
-----  ----------   ----    --------------------
 7782  CLARK        2450                    2450
 7839  KING         5000                12250000
 7934  MILLER       1300             15925000000
```

SQL Server users use LOG instead of LN.

DISCUSSION

Except for the MODEL clause solution, which is only usable with Oracle Database 10g or later, all the solutions take advantage of the fact that you can multiply two numbers by:

Computing their respective natural logarithms

Summing those logarithms

Raising the result to the power of the mathematical constant e (using the EXP function)

The one caveat when using this approach is that it doesn't work for summing zero or negative values, because any value less than or equal to zero is out of range for an SQL logarithm.

DB2 and Oracle
For an explanation of how the window function SUM OVER works, see the previous recipe "Generating a Running Total."

In Oracle Database 10g and later, you can generate running products via the MODEL clause. Using the MODEL clause along with the window function ROW_NUMBER allows you to easily access prior rows. Each item in the MEASURES list can be accessed like an array. The arrays can then be searched by using the items in the DIMENSIONS list (which are the values returned by ROW_NUMBER, alias RN):

```
select empno, ename, sal, tmp as running_prod,rn
  from (
select empno,ename,-sal as sal
  from emp
 where deptno=10
       )
 model
   dimension by(row_number()over(order by sal desc) rn )
   measures(sal, 0 tmp, empno, ename)
  rules ()

EMPNO   ENAME              SAL  RUNNING_PROD            RN
-----   ----------  ----------  ------------    ----------
 7934   MILLER           -1300             0             1
 7782   CLARK            -2450             0             2
 7839   KING             -5000             0             3
```

Observe that SAL[1] has a value of–1300. Because the numbers are increasing by one with no gaps, you can reference prior rows by subtracting one. The RULES clause:

```
rules (
   tmp[any] = case when sal[cv()-1] is null then sal[cv()]
                   else tmp[cv()-1]*sal[cv()]
              end
)
```

uses the built-in operator, ANY, to work through each row without hard-coding. ANY in this case will be the values 1, 2, and 3. TMP[n] is initialized to zero. A value is assigned to TMP[n] by evaluating the current value (the function CV returns the current value) of the corresponding SAL row. TMP[1] is initially zero and SAL[1] is–1300. There is no value for SAL[0] so TMP[1] is set to SAL[1]. After TMP[1] is set, the next row is TMP[2]. First SAL[1] is evaluated (SAL[CV()–1] is SAL[1] because the current value of ANY is now 2). SAL[1] is not null, it is–1300, so TMP[2] is set to the product of TMP[1] and SAL[2]. This is continued for all the rows.

MySQL, PostgreSQL, and SQL Server
See “Generating a Running Total” earlier in this lab for an explanation of the subquery approach used for the MySQL, PostgreSQL, and SQL Server solutions.

Be aware that the output of the subquery-based solution is slightly different from that of the Oracle and DB2 solutions due to the EMPNO comparison (the running product is computed in a different order). Like a running total, the summation is driven by the predicate of the scalar subquery; the ordering of rows is by EMPNO for this solution whereas the Oracle/DB2 solution order is by SAL.

7.8. Calculating a Running Difference

PROBLEM

You want to compute a running difference on values in a numeric column. For example, you want to compute a running difference on the salaries in DEPTNO 10. You would like to return the following result set:

```
ENAME             SAL RUNNING_DIFF
---------- ---------- ------------
MILLER           1300         1300
CLARK            2450        -1150
KING             5000        -6150
```

SOLUTION

DB2 and Oracle

Use the window function SUM OVER to create a running difference:

```
select ename,sal,
       sum(case when rn = 1 then sal else -sal end)
        over(order by sal,empno) as running_diff
  from (
select empno,ename,sal,
       row_number()over(order by sal,empno) as rn
  from emp
 where deptno = 10
       ) x
```

MySQL, PostgreSQL, and SQL Server

Use a scalar subquery to compute a running difference:

```
 select a.empno, a.ename, a.sal,
        (select case when a.empno = min(b.empno) then sum(b.sal)
                     else sum(-b.sal)
                end
           from emp b
          where b.empno <= a.empno
            and b.deptno = a.deptno ) as rnk
  from emp a
 where a.deptno = 10
```

DISCUSSION

The solutions are identical to those of “Generating a Running Total.” The only difference is that all values for SAL are returned as negative values with the exception of the first (you want the starting point to be the first SAL in DEPTNO 10).

7.9. Calculating a Mode

PROBLEM

You want to find the mode (for those of you who don’t recall, the mode in mathematics is the element that appears most frequently for a given set of data) of the values in a column. For example, you wish to find mode of the salaries in DEPTNO 20. Based on the following salaries:

```
select sal
  from emp
 where deptno = 20
 order by sal

       SAL
----------
       800
      1100
      2975
      3000
      3000
```

the mode is 3000.

SOLUTION

DB2 and SQL Server
Use the window function DENSE_RANK to rank the counts of the salaries to facilitate extracting the mode:

```
select sal
  from (
select sal,
       dense_rank()over( order by cnt desc) as rnk
  from (
select sal, count(*) as cnt
  from emp
 where deptno = 20
 group by sal
       ) x
       ) y
 where rnk = 1
```

Oracle

Users on Oracle8i Database can use the solution provided for DB2. If you are on Oracle9i Database and later, you can use the KEEP extension to the aggregate function MAX to find the mode SAL. One important note is that if there are ties, i.e., multiple rows that are the mode, the solution using KEEP will only keep one, and that is the one with the highest salary. If you want to see all modes (if more than one exists), you must modify this solution or simply use the DB2 solution presented above. In this case, since 3000 is the mode SAL in DEPTNO 20 and is also the highest SAL, this solution is sufficient:

```
 select max(sal)
        keep(dense_rank first order by cnt desc) sal
   from (
 select sal, count(*) cnt
   from emp
  where deptno=20
  group by sal
        )
```

MySQL and PostgreSQL

Use a subquery to find the mode:

```
select sal
  from emp
 where deptno = 20
 group by sal
having count(*) >= all ( select count(*)
                           from emp
                          where deptno = 20
                          group by sal )
```

DISCUSSION

DB2 and SQL Server
The inline view X returns each SAL and the number of times it occurs. Inline view Y uses the window function DENSE_RANK (which allows for ties) to sort the results.

The results are ranked based on the number of times each SAL occurs as is seen below:

```
  select sal,
         dense_rank()over(order by cnt desc) as rnk
    from (
  select sal,count(*) as cnt
    from emp
   where deptno = 20
   group by sal
         ) x
```

```
   SAL          RNK
-------   ----------
  3000            1
   800            2
  1100            2
  2975            2
```

The outermost portion of query simply keeps the row(s) where RNK is 1.

Oracle
The inline view returns each SAL and the number of times it occurs and is shown below:

```
select sal, count(*) cnt
from emp
where deptno=20
group by sal
```

```
   SAL          CNT
-------   ----------
   800            1
  1100            1
  2975            1
  3000            2
```

The next step is to use the KEEP extension of the aggregate function MAX to find the mode. If you analyze the KEEP clause shown below you will notice three subclauses, DENSE_RANK, FIRST, and ORDER BY CNT DESC:

```
keep(dense_rank first order by cnt desc)
```

What this does is extremely convenient for finding the mode. The KEEP clause determines which SAL will be returned by MAX by looking at the value of CNT returned by the inline view. Working from right to left, the values for CNT are ordered in descending order, then the first is kept of all the values for CNT returned in DENSE_RANK order. Looking at the result set from the inline view, you can see that 3000 has the highest CNT of 2. The MAX(SAL) returned is the greatest SAL that has the greatest CNT, in this case 3000.

SEE ALSO
lab 11, the section on “Finding Knight Values,” for a deeper discussion of Oracle’s KEEP extension of aggregate functions.

MySQL and PostgreSQL
The subquery returns the number of times each SAL occurs. The outer query returns any SAL that has a number of occurrences greater than or equal to all of the counts returned by the subquery (or to put it another way, the outer query returns the most common salaries in DEPTNO 20).

7.10. Calculating a Median

PROBLEM

You want to calculate the median (for those of who do not recall, the median is the value of the middle member of a set of ordered elements) value for a column of numeric values. For example, you want to find the median of the salaries in DEPTNO 20. Based on the following salaries:

```
select sal
  from emp
 where deptno = 20
 order by sal

       SAL
----------
       800
      1100
      2975
      3000
      3000
```

the median is 2975.

SOLUTION

Other than the Oracle solution (which uses supplied functions to compute a median), all of the solutions are based on the method described by Rozenshtein, Abramovich, and Birger in Optimizing

Transact-SQL: Advanced Programming Techniques (SQL Forum Press, 1997). The introduction of window functions allows for a more efficient solution compared to the traditional self join.

DB2

Use the window functions COUNT(*) OVER and ROW_NUMBER to find the median:

```
select avg(sal)
  from (
select sal,
       count(*) over() total,
       cast(count(*) over() as decimal)/2 mid,
       ceil(cast(count(*) over() as decimal)/2) next,
       row_number() over ( order by sal) rn
  from emp
 where deptno = 20
       ) x
where ( mod(total,2) = 0
        and rn in ( mid, mid+1 )
      )
   or ( mod(total,2) = 1
        and rn = next
      )
```

MySQL and PostgreSQL

Use a self join to find the median:

```
select avg(sal)
  from (
select e.sal
  from emp e, emp d
 where e.deptno = d.deptno
   and e.deptno = 20
 group by e.sal
having sum(case when e.sal = d.sal then 1 else 0 end)
                          >= abs(sum(sign(e.sal - d.sal)))
       )
```

Oracle

Use the functions MEDIAN (Oracle Database 10g) or PERCENTILE_CONT (Oracle9i Database):

```
select median(sal)
  from emp
 where deptno=20
select percentile_cont(0.5)
        within group(order by sal)
  from emp
 where deptno=20
```

Use the DB2 solution for Oracle8i Database. For versions prior to Oracle8i Database you can use the PostgreSQL/MySQL solution.

SQL Server

Use the window functions COUNT(*) OVER and ROW_NUMBER to find the median:

```
select avg(sal)
  from (
select sal,
       count(*)over() total,
       cast(count(*)over() as decimal)/2 mid,
       ceiling(cast(count(*)over() as decimal)/2) next,
row_number()over( order by sal) rn
  from emp
 where deptno = 20
       ) x
 where ( total%2 = 0
         and rn in ( mid, mid+1 )
       )
    or ( total%2 = 1
         and rn = next
       )
```

DISCUSSION

DB2 and SQL Server

The only difference between the DB2 and SQL Server solutions is a small point of syntax: SQL Server uses “%” for modulo and DB2 uses the function MOD; otherwise they are the same. Inline view X returns three different counts, TOTAL, MID, and NEXT, along with RN, generated by ROW_NUMBER. These additional columns help determine how to find the median. Examine the result set for inline view X to see what these columns represent:

```
select sal,
       count(*)over() total,
       cast(count(*)over() as decimal)/2 mid,
       ceil(cast(count(*)over() as decimal)/2) next,
       row_number()over(order by sal) rn
  from emp
 where deptno = 20


SAL   TOTAL  MID NEXT   RN
----  ----- ---- ---- ----
 800      5  2.5    3    1
1100      5  2.5    3    2
2975      5  2.5    3    3
3000      5  2.5    3    4
3000      5  2.5    3    5
```

To find the median, the values for SAL must be ordered from lowest to highest. Since DEPTNO 20 has an odd number of employees, the median is simply the SAL that is located in the position where RN equals NEXT (the position that represents the smallest whole number larger than the total number of employees divided by two).

The first part of the WHERE clause (lines 11–13) is not satisfied if there are an odd number of rows returned by the result set. If you know that the result set will always be odd, you can simplify to:

```
select avg(sal)
  from (
select sal,
       count(*)over() total,
       ceil(cast(count(*)over() as decimal)/2) next,
       row_number()over(order by sal) rn
  from emp
 where deptno = 20
       ) x
 where rn = next
```

Unfortunately, if you have an even number of rows in the result set, the simplified solution will not work. The original solution handles even-numbered rows by using the values in the column MID. Consider what the results from inline view X would look like for DEPTNO 30, which has six employees:

```
select sal,
       count(*)over() total,
       cast(count(*)over() as decimal)/2 mid,
       ceil(cast(count(*)over() as decimal)/2) next,
       row_number()over(order by sal) rn
  from emp
 where deptno = 30

 SAL   TOTAL   MID  NEXT    RN
 ----  -----  ----  ----  ----
  950      6     3     3     1
 1250      6     3     3     2
 1250      6     3     3     3
 1500      6     3     3     4
 1600      6     3     3     5
 2850      6     3     3     6
```

Since there are an even number of rows returned, the median is computed by taking the average of two rows; the row where RN equals MID and the row where RN equals MID + 1.

MySQL and PostgreSQL

The median is computed by first self joining table EMP, which returns a Cartesian product for all the salaries (but the GROUP BY on E.SAL will prevent duplicates from being returned). The HAVING clause uses the function SUM to count the number of times E.SAL equals D.SAL; if this count is greater than or equal to the number of times E.SAL is greater than D.SAL then that row is the median. You can observe this by moving the SUM into the SELECT list:

```
select e.sal,
       sum(case when e.sal=d.sal
                then 1 else 0 end) as cnt1,
       abs(sum(sign(e.sal - d.sal))) as cnt2
  from emp e, emp d
 where e.deptno = d.deptno
   and e.deptno = 20
 group by e.sal

 SAL CNT1 CNT2
---- ---- ----
 800    1    4
1100    1    2
2975    1    0
3000    4    6
```

Oracle

If you are on Oracle Database 10g or Oracle9i Database, you can leave the work of computing a median to functions supplied by Oracle. If you are running Oracle8i Database, you can use the DB2

solution. Otherwise you must use the PostgreSQL solution. While the MEDIAN function obviously computes a median, it may not be at all obvious that PERCENTILE_CONT does so as well. The argument passed to PERCENTILE_CONT, 0.5, is a percentile value. The clause, WITHIN GROUP (ORDER BY SAL), determines which sorted rows PERCENTILE_CONT will search (remember, a median is the middle value from a set of ordered values). The value returned is the value from the sorted rows that falls into the given percentile (in this case, 0.5, which is the middle because the boundary values are 0 and 1).

7.11. Determining the Percentage of a Total

PROBLEM

You want to determine the percentage that values in a specific column represent against a total. For example, you want to determine what percentage of all salaries are the salaries in DEPTNO 10 (the percentage that DEPTNO 10 salaries contribute to the total).

SOLUTION

In general, computing a percentage against a total in SQL is no different than doing so on paper; simply divide, then multiply. In this example you want to find the percentage of total salaries in table EMP that come from DEPTNO 10. To do that, simply find the salaries for DEPTNO 10, and then divide by the total salary for the table. As the last step, multiply by 100 to return a value that represents a percent.

MySQL and PostgreSQL
Divide the sum of the salaries in DEPTNO 10 by the sum of all salaries:

```
select (sum(
         case when deptno = 10 then sal end)/sum(sal)
        )*100 as pct
  from emp
```

DB2, Oracle, and SQL Server
Use an inline view with the window function SUM OVER to find the sum of all salaries along with the sum of all salaries in DEPTNO 10. Then do the division and multiplication in the outer query:

```
select distinct (d10/total)*100 as pct
  from (
select deptno,
       sum(sal)over() total,
       sum(sal)over(partition by deptno) d10
  from emp
       ) x
 where deptno=10
```

DISCUSSION

MySQL and PostgreSQL

The CASE statement conveniently returns only the salaries from DEPTNO 10. They are then summed and divided by the sum of all the salaries. Because NULLs are ignored by aggregates, an ELSE clause is not needed in the CASE statement. To see exactly which values are divided, execute the query without the division:

```
select sum(case when deptno = 10 then sal end) as d10,
       sum(sal)
  from emp

D10  SUM(SAL)
---- ---------
8750     29025
```

Depending on how you define SAL, you may need to include explicit casts when performing division. For example, on DB2, SQL Server, and PostgreSQL, if SAL is stored as an integer, you can cast to decimal to get the correct answer, as seen below:

```
select (cast(
         sum(case when deptno = 10 then sal end)
            as decimal)/sum(sal)
        )*100 as pct
  from emp
```

DB2, Oracle, and SQL Server

As an alternative to the traditional solution, this solution uses window functions to compute a percentage relative to the total. For DB2 and SQL Server, if you've stored SAL as an integer, you'll need to cast before dividing:

```
select distinct
       cast(d10 as decimal)/total*100 as pct
  from (
select deptno,
       sum(sal)over() total,
       sum(sal)over(partition by deptno) d10
  from emp
       ) x
 where deptno=10
```

It is important to keep in mind that window functions are applied after the WHERE clause is evaluated. Thus, the filter on DEPTNO cannot be performed in inline view X. Consider the results of inline view X without and with the filter on DEPTNO. First without:

```
select deptno,
       sum(sal)over() total,
       sum(sal)over(partition by deptno) d10
  from emp

DEPTNO      TOTAL       D10
------- --------- ---------
     10     29025      8750
     10     29025      8750
     10     29025      8750
     20     29025     10875
     20     29025     10875
     20     29025     10875
     20     29025     10875
     20     29025     10875
     30     29025      9400
     30     29025      9400
     30     29025      9400
     30     29025      9400
     30     29025      9400
     30     29025      9400
```

and now with:

```
select deptno,
       sum(sal)over() total,
       sum(sal)over(partition by deptno) d10
  from emp
 where deptno=10

DEPTNO     TOTAL       D10
------ --------- ---------
    10      8750      8750
    10      8750      8750
    10      8750      8750
```

Because window functions are applied after the WHERE clause, the value for TOTAL represents the sum of all salaries in DEPTNO 10 only. But to solve the problem you want the TOTAL to represent the sum of all salaries, period. That's why the filter on DEPTNO must happen outside of inline view X.

7.12. Aggregating Nullable Columns

PROBLEM

You want to perform an aggregation on a column, but the column is nullable. You want the accuracy of your aggregation to be preserved, but are concerned because aggregate functions ignore NULLs. For example, you want to determine the average commission for employees in DEPTNO 30, but there are some employees who do not earn a commission (COMM is NULL for those employees). Because NULLs are ignored by aggregates, the accuracy of the output is compromised. You would like to somehow include NULL values in your aggregation.

SOLUTION

Use the COALESCE function to convert NULLs to 0, so they will be included in the aggregation:

```
select avg(coalesce(comm,0)) as avg_comm
  from emp
 where deptno=30
```

DISCUSSION

When working with aggregate functions, keep in mind that NULLs are ignored. Consider the output of the solution without using the COALESCE function:

```
select avg(comm)
  from emp
 where deptno=30

 AVG(COMM)
 ---------
       550
```

This query shows an average commission of 550 for DEPTNO 30, but a quick examination of those rows:

```
select ename, comm
  from emp
 where deptno=30
order by comm desc

ENAME           COMM
---------- ---------
BLAKE
JAMES
MARTIN          1400
WARD             500
ALLEN            300
TURNER             0
```

shows that only four of the six employees can earn a commission. The sum of all commissions in DEPTNO 30 is 2200, and the average should be 2200/6, not 2200/4. By excluding the COALESCE function, you answer the question, “What is the average commission of employees in DEPTNO 30 who can earn a commission?” rather than “What is the average commission of all employees in DEPTNO 30?” When working with aggregates, remember to treat NULLs accordingly.

7.13. Computing Averages Without High and Low Values

PROBLEM

You want to compute an average, but you wish to exclude the highest and lowest values in order to (hopefully) reduce the effect of skew. For example, you want to compute the average salary of all employees excluding the highest and lowest salaries.

SOLUTION

MySQL and PostgreSQL
Use subqueries to exclude high and low values:

```
select avg(sal)
  from emp
 where sal not in (
    (select min(sal) from emp),
    (select max(sal) from emp)
 )
```

DB2, Oracle, and SQL Server
Use an inline view with the windowing functions MAX OVER and MIN OVER to generate a result set from which you can easily eliminate the high and low values:

```
select avg(sal)
  from (
select sal, min(sal)over() min_sal, max(sal)over() max_sal
  from emp
       ) x
 where sal not in (min_sal,max_sal)
```

DISCUSSION

MySQL and PostgreSQL
The subqueries return the highest and lowest salaries in the table. By using NOT IN against the values returned, you exclude the highest and lowest salaries from the average. Keep in mind that if there are duplicates (if multiple employees have the highest or lowest salaries), they will all be excluded from the average. If your goal is to exclude only a single instance of the high and low values, simply subtract them from the SUM and then divide:

```
select (sum(sal)-min(sal)-max(sal))/(count(*)-2)
  from emp
```

DB2, Oracle, and SQL Server
Inline view X returns each salary along with the highest and lowest salary:

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

      SAL   MIN_SAL   MAX_SAL
--------- --------- ---------
      800       800      5000
     1600       800      5000
     1250       800      5000
     2975       800      5000
     1250       800      5000
     2850       800      5000
     2450       800      5000
     3000       800      5000
     5000       800      5000
     1500       800      5000
     1100       800      5000
      950       800      5000
     3000       800      5000
     1300       800      5000
```

You can access the high and low salary at every row, so finding which salaries are highest and/or lowest is trivial. The outer query filters the rows returned from inline view X such that any salary that matches either MIN_SAL or MAX_SAL is excluded from the average.

7.14. Converting Alphanumeric Strings into Numbers

PROBLEM

You have alphanumeric data and would like to return numbers only. You want to return the number 123321 from the string “paul123f321”.

SOLUTION

DB2

Use the functions TRANSLATE and REPLACE to extract numeric characters from an alphanumeric string:

```
select cast(
       replace(
     translate( 'paul123f321',
                repeat('#',26),
                'abcdefghijklmnopqrstuvwxyz'),'#','')
       as integer ) as num
  from t1
```

Oracle and PostgreSQL
Use the functions TRANSLATE and REPLACE to extract numeric characters from an alphanumeric string:

```
select cast(
       replace(
     translate( 'paul123f321',
                'abcdefghijklmnopqrstuvwxyz',
                rpad('#',26,'#')),'#','')
       as integer ) as num
  from t1
```

MySQL and SQL Server
As of the time of this writing, neither vendor supports the TRANSLATE function, thus a solution will not be provided.

DISCUSSION

The only difference between the two solutions is syntax; DB2 uses the function REPEAT rather than RPAD and the parameter list for TRANSLATE is in a different order. The following explanation uses the Oracle/PostgreSQL solution but is relevant to DB2 as well. If you run query inside out (starting with TRANSLATE only), you'll see this is very simple. First, TRANSLATE converts any non-numeric character to an instance of "#":

```
select translate( 'paul123f321',
                  'abcdefghijklmnopqrstuvwxyz',
                  rpad('#',26,'#')) as num
  from t1

NUM
-----------
####123#321
```

Since all non-numeric characters are now represented by "#", simply use REPLACE to remove them, then cast the result to a number. This particular example is extremely simple because the data is alphanumeric. If additional characters can be stored, rather than fishing for those characters, it is easier to approach this problem differently: rather than finding non-numeric characters and then removing them, find all numeric characters and remove anything that is not amongst them. The following example will help clarify this technique:

```
select replace(
     translate('paul123f321',
       replace(translate( 'paul123f321',
                          '0123456789',
                          rpad('#',10,'#')),'#',''),
                rpad('#',length('paul123f321'),'#')),'#','') as num
  from t1

NUM
-----------
123321
```

This solution looks a bit more convoluted than the original but is not so bad once you break it down. Observe the innermost call to TRANSLATE:

```
select translate( 'paul123f321',
                  '0123456789',
                  rpad('#',10,'#'))
  from t1

TRANSLATE('
-----------
paul###f###
```

So, the initial approach is different; rather than replacing each non-numeric character with an instance of “#”, you replace each numeric character with an instance of “#”. The next step removes all instances of “#”, thus leaving only non-numeric characters:

```
select replace(translate( 'paul123f321',
                          '0123456789',
                          rpad('#',10,'#')),'#','')
  from t1

REPLA
-----
paulf
```

The next step is to call TRANSLATE again, this time to replace each of the non-numeric characters (from the query above) with an instance of “#” in the original string:

```
select translate('paul123f321',
       replace(translate( 'paul123f321',
                          '0123456789',
                          rpad('#',10,'#')),'#',''),
                rpad('#',length('paul123f321'),'#'))
  from t1

TRANSLATE('
__________
####123#321
```

At this point, stop and examine the outermost call to TRANSLATE. The second parameter to RPAD (or the second parameter to REPEAT for DB2) is the length of the original string. This is convenient to use since no character can occur enough times to be greater than the string it is part of. Now that all non-numeric characters are replaced by instances of “#”, the last step is to use REPLACE to remove all instances of “#”. Now you are left with a number.

7.15. Changing Values in a Running Total

PROBLEM

You want to modify the values in a running total depending on the values in another column. Consider a scenario where you want to display the transaction history of a credit card account along with the current balance after each transaction. The following view, V, will be used in this example:

```
create view V (id,amt,trx)
as
select 1, 100, 'PR' from t1 union all
select 2, 100, 'PR' from t1 union all
select 3, 50,  'PY' from t1 union all
select 4, 100, 'PR' from t1 union all
select 5, 200, 'PY' from t1 union all
select 6, 50,  'PY' from t1

select * from V

ID         AMT  TR
--  ----------  --
 1         100  PR
 2         100  PR
 3          50  PY
 4         100  PR
 5         200  PY
 6          50  PY
```

The ID column uniquely identifies each transaction. The AMT column represents the amount of money involved in each transaction (either a purchase or a payment). The TRX column defines the type of transaction; a payment is “PY” and a purchase is “PR.” If the value for TRX is PY, you want the current value for AMT subtracted from the running total; if the value for TRX is PR, you want the current value for AMT added to the running total. Ultimately you want to return the following result set:

```
TRX_TYPE        AMT    BALANCE
-------- ---------- ----------
PURCHASE        100        100
PURCHASE        100        200
PAYMENT          50        150
PURCHASE        100        250
PAYMENT         200         50
PAYMENT          50          0
```

SOLUTION

DB2 and Oracle

Use the window function SUM OVER to create the running total along with a CASE expression to determine the type of transaction:

```
select case when trx = 'PY'
            then 'PAYMENT'
            else 'PURCHASE'
        end trx_type,
        amt,
        sum(
         case when trx = 'PY'
            then -amt else amt
         end
       ) over (order by id,amt) as balance
from V
```

MySQL, PostgreSQL, and SQL Server

Use a scalar subquery to create the running total along with a CASE expression to determine the type of transaction:

```
select case when v1.trx = 'PY'
            then 'PAYMENT'
            else 'PURCHASE'
        end as trx_type,
        v1.amt,
       (select sum(
                case when v2.trx = 'PY'
                     then -v2.amt else v2.amt
                end
               )
          from V v2
         where v2.id <= v1.id) as balance
  from V v1
```

DISCUSSION

The CASE expression determines whether the current AMT is added or deducted from the running total. If the transaction is a payment, the AMT is changed to a negative value, thus reducing the amount of the running total. The result of the CASE expression is seen below:

```
select case when trx = 'PY'
            then 'PAYMENT'
            else 'PURCHASE'
       end trx_type,
       case when trx = 'PY'
            then -amt else amt
       end as amt
  from V

TRX_TYPE       AMT
-------- ---------
PURCHASE       100
PURCHASE       100
PAYMENT        -50
PURCHASE       100
PAYMENT       -200
PAYMENT        -50
```

After evaluating the transaction type, the values for AMT are then added to or subtracted from the running total. For an explanation on how the window function, SUM OVER, or the scalar subquery creates the running total see recipe “Calculating a Running Total.”