Craig S. Mullins

*Database Performance Management*

Return to Home Page

DB2 update

June 2005

## Some SQL Tricks for the DB2 Developer

*By Craig S. Mullins*

It is always a good idea to keep your bag of SQL tricks loaded with techniques to help solve some of the more common or troubling application development problems. I do not know if the following SQL techniques match up to your most pressing development issues, but I'm sure they might come in handy in certain situations.

### Sorting Days of the Week

Here is a sorting trick that you can use when you are dealing with temporal data. Assume that you have a table containing transactions, or some other type of interesting facts. The table has a CHAR(3) column containing the name of the day on which the transaction happened; let’s call this column DAY_NAME. Now, let’s further assume that we want to write queries against this table that orders the results by DAY_NAME. We’d want Sunday first, followed by Monday, Tuesday, Wednesday, and so on. How can this be done?

Well, if we write the first query that comes to mind the results will obviously be sorted improperly:

```
SELECT   DAY_NAME, COL1, COL2 . . .
FROM     TXN_TABLE
ORDER BY DAY_NAME;
```

The results from this query would be ordered alphabetically; in other words

```
FRI
MON
SAT
SUN
THU
TUE
WED
```

One solution would be to design the table with an additional numeric or alphabetic column that would sort properly. By this I mean that we could add a DAY_NUM column that would be 1 for Sunday, 2 for Monday, and so on. But this requires a database design change, and it becomes possible for the DAY_NUM and DAY_NAME to get out of sync.

A better solution uses just SQL and requires no change to the database structures. All you need is an understanding of SQL and SQL functions – in this case, the LOCATE function.  Here is the SQL:

```
SELECT   DAY_NAME, COL1, COL2 . . .
FROM     TXN_TABLE
ORDER BY
LOCATE(DAY_NAME,'SUNMONTUEWEDTHUFRISAT');
```

The trick here is to understand how the LOCATE function works: it returns the starting position of the first occurrence of one string within another string. So, in our example, LOCATE finds the position of the DAY_NAME value within the string 'SUNMONTUEWEDTHUFRISAT', and returns the integer value of that position. If DAY_NAME is WED, the LOCATION function in the above SQL statement returns 10. (Note: Some other database systems have a similar function called INSTR.) Sunday would return 1, Monday 4, Tuesday 7, Wednesday 10, Thursday 13, Friday 16, and Saturday 19. This means that our results would be in the order we require.

Of course, you can go one step further if you'd like. Some queries may need to actually return the day of week. You can use the same technique with a twist to return the day of week value given only the day's name. To turn this into the appropriate day of the week number (that is, a value of 1 through 7), we divide by three, use the INT function on the result to return only the integer portion of the result, and then add one:

```
INT(LOCATE('SUNMONTUEWEDTHUFRISAT',DAY_NAME)/3)
                                  + 1;
```

Let’s use our previous example of Wednesday again. The LOCATION function returns the value 10. So, INT(10/3) = 3 and add 1 to get 4. And sure enough, Wednesday is the fourth day of the week.

**Removing Superfluous Spaces**

We all can relate to dealing with systems that have data integrity problems. But some data integrity problems can be cleaned up using a touch of SQL. Consider the common data entry problem of extraneous spaces (or blanks) inserted into a name field. Not only is it annoying, sometimes it can cause the system to ignore relationships between data elements because the names do not match. For example, “Craig Mullins” is not equivalent to “Craig Mullins”; the first one has two spaces between the first and last name whereas the second one only has one.

You can write an SQL UPDATE statement to clean up the problem, if you know how to use the REPLACE function. REPLACE does what it sounds like it would do: it reviews a source string and replaces all occurrences of a one string with another. For example, to replace all occurrences of Z with A in the string BZNZNZ you would code:

```
          REPLACE(‘BZNZNZ’,’Z’,’A’)
```

And the result would be BANANA. So, let’s create a SQL statement using REPLACE to get rid of any unwanted spaces in the NAME column of our EMPLOYEE table:

```
UPDATE EMPLOYEE
   SET NAME = REPLACE(
               REPLACE(
                REPLACE(NAME, SPACE(1), '<>')
                '><', SPACE(0))
               '<>', SPACE(1));
```

Wait-a-minute, you might be saying. What are all of those left and right carats and why do I need them? Well, let’s go from the inside out. The inside REPLACE statement takes the NAME column and converts every occurrence of a single space into a left/right carat. The next REPLACE (working outward), takes the string we just created, and removes every occurrence of a right/left carat combination by replacing it with a zero length string. The final REPLACE function takes that string and replaces any left/right carats with a single space. The reversal of the carats is the key to removing all spaces except one – remember, we want to retain a single space anywhere there was a single space as well as anywhere that had multiple spaces. Try it, it works.

Of course, you can use any two characters you like, but the left and right carat characters work well visually. Be sure that you do not choose to use characters that occur naturally in the string that you are acting upon.

Finally, the SPACE function was used for clarity. You could have used strings encased in single quotes, but the SPACE function is easier to read. It simply returns a string of spaces the length of which is specified as the integer argument. So, SPACE(12) would return a string of twelve spaces.

## Aggregating Aggregates

The final SQL trick we'll uncover in this article allows us to perform aggregations of aggregates. For example, you might want to compute the average of a sum. This comes up frequently in applications that are built around sales amounts. Let's assume that we have a table containing sales information. Each sales amount has additional information indicating the salesman, region, district, product, date, etc.

A common requirement is to produce a report of the average sales by region for a particular period, say the first quarter of 2005. But the data in the table is at a detail level, meaning we have a row for each specific sale.

A novice SQL coder might try to write a query with a function inside a function, like AVG(SUM(SALE_AMT)). Of course, this is invalid SQL syntax. DB2 will not permit the nesting of aggregate functions. But we can use nested table expressions and our knowledge of SQL functions to build the correct query.

Let's start by creating a query to return the sum of all sales by region for the time period in question. That query should look something like this:

```
SELECT REGION, SUM(SALE_AMT)
FROM   SALES
WHERE SALE_DATE BETWEEN DATE('2005-01-01')
                AND     DATE('2005-03-31')
GROUP BY REGION;
```

Now that we have the total sales by region for the period in question, we can embed this query into a nested table expression in another query like so:

```
SELECT NTE.REGION, AVE(NTE.TOTAL_SALES)
FROM (SELECT REGION, SUM(SALE_AMT)
      FROM   SALES
      WHERE SALE_DATE BETWEEN DATE('2005-01-
01')
                      AND     DATE('2005-03-
31')
```

```
        GROUP BY REGION) AS NTE
GROUP BY NTE.REGION;
```

## Summary

In this article we examined several techniques to solve application problems using only SQL. With a sound understanding of SQL, and particularly SQL functions and expressions, you can often times find novel ways of solving thorny problems using nothing but SQL.

From DB2 Update, June 2005.