DB2's outer join. (SQL enhancements in DB2 for MVS/ESA Version 4.1)

From:
Enterprise Systems Journal
Date:
December 1, 1996
Author:
Favero, Willie
More results for:
Willie Favero

DB2 Version 4.1 has a number of SQL enhancements over the previous version, including the addition of a true outer join. The enhancements make outer joins easier to code and more reliable than earlier versions of DB2. The new outer joins perform better than the old commands for tables including those with large amounts of data. An example is shown to illustrate the ease in coding of outer joins with Version 4.1. Three types of outer join, left, right and full, are available. DB2 Version 4.1 has an added keyword JOIN and operators to code an outer join. Specific rules are provided to use the JOIN keyword for an outer join. The DB2 for MVS/ESA coding techniques discussed are applicable to DB2 Common Server releases.

DB2's Outer Join

Enhancements To SQL In DB2 For MVS/VSE Version 4.1 Make Life Easier

With each new release of DB2, there is much discussion about the improvements in performance, data access, recovery, utilities, etc. Seldom is there much ado about improvements in SQL. DB2 for MVS/ESA Version 4.1 changes all that. A number of significant enhancements to DB2's SQL are in this release -- everything from outer joins to table check constraints. Many of the changes are in direct support of the SQL92 ANSI standard. This article discusses the outer join and the SQL enhancements that can be of benefit when used with the outer join.

In The Old Days

To truly understand the significance of an outer join, you must first step back in time, all the way back to DB2 Version 3, and see what there was. In Version 3, only an inner join was available. In most cases, it did all that was needed. An inner join returns only rows that have an equal condition on the join predicate in both tables. If a value exists in only one of the tables, nothing is returned in the join result. Table 1 contains the data from three tables used in the SQL examples. The COLOR table contains a list of popular colors with a reference value. The FAVORITE table contains a list of people with the reference number for their favorite color. The CARS table is a list of cars and the reference value for the colors they come in.

A simple inner join can be run between the COLOR and FAVORITE tables using the column COLNO, the color reference number, as the join predicate. The desired result is a list of everyone in the FAVORITE table and their favorite color. Example 1 shows the query and the result. Although there are four individuals with favorite colors in the FAVORITE table, the query result has only three rows. Is this the correct answer? In this example, no. The original statement asked for everyone in the FAVORITE table. However, the answer contains only the rows where the join predicate found a matching row in both tables. There is a missing row because someone has a

favorite color that is not in the COLOR table. This could signify a data error. If the FAVORITE table had contained millions of rows, it might never be known that rows were missing from the answer set. So how can the rows be found that do not have a match in both tables?

A query can be formulated to return a row for every individual in the FAVORITE table. But it is not as straightforward a query as you might hope. Example 2 contains the query, with its result, coded in DB2 Version 3. To get all the rows from both the FAVORITE and COLOR tables, a second UNION to a SELECT with similar subselect and NOT EXISTS would have to be added to the query. Example 3 shows the modified SQL statement to return all unmatched rows from both tables and its result. These examples of outer joins before DB2 Version 4.1 are complex, difficult to read and error-prone. Look at Example 2 again. This is a full outer join of two tables. Can you imagine what this query might look like if you were trying to perform an outer join of three or more tables?

A True Outer Join

DB2 Version 4.1 enhances its SQL language with the addition of a true outer join. Example 4 shows a Version 4.1 left outer join. It gives the same result as the SQL statement in Example 2. However, comparing the SQL statements in Examples 2 and 4, the outer join in Example 4 is considerably easier to code and will probably be less prone to errors. Now that you've seen the simplicity of coding the new outer join, consider its components and how they work.

An outer join comes in three flavors: left, right and full. To code an outer join, a new keyword and clause have been added. First, there is the join connector on the FROM clause. Rather than using a comma as the connector between two or more tables being joined, a new keyword, JOIN, has been introduced. Specifying the JOIN keyword without a join operator defaults to an inner join. However, if the join operator LEFT, RIGHT or FULL is combined with the JOIN keyword, DB2 performs an outer join. The SQL example in Example 4 shows that OUTER and INNER can be explicitly specified although they are optional. When they are not coded, DB2 determines whether it should perform an inner or outer join by the omission or inclusion of the LEFT, RIGHT or FULL keywords.

The order in which the tables are specified on the JOIN keyword can affect the results of the join. A LEFT OUTER JOIN retains the unmatched rows from the composite or outer table. This is the table specified on the left-hand side of the expression. The RIGHT OUTER JOIN retains the unmatched rows from the new or inner table, the table specified on the right-hand side of the join expression. A FULL OUTER JOIN contains the unmatched rows from both tables. The JOIN keyword can be used to control the order in which the tables are evaluated.

When you examine the output from EXPLAIN, left and right outer joins can be either nested loop joins or merge-scan joins. A full outer join usually uses a merge-scan join technique. An outer join also discourages the choice of a hybrid join. A new column, JOIN--TYPE, has been added to the PLAN--TABLE to identify the type of join to be processed. If JOIN--TYPE is blank, an inner join is being used. If JOIN--TYPE is set to F, it's a FULL OUTER JOIN. However, if JOIN--TYPE is set to L, LEFT OUTER JOIN or RIGHT OUTER JOIN may have been specified in the SQL statement. DB2 does not perform a RIGHT OUTER JOIN. The optimizer does not have the concept of a right outer join. It reverses the table order and performs a left outer join.

TABLE1 RIGHT OUTER JOIN TABLE2

becomes

TABLE2 LEFT OUTER JOIN TABLE1

The order of the tables being evaluated in the PLAN--TABLE is also reversed. In the previous example, TABLE2 becomes the outer table. Also, if performing a multiple table join and one of the join methods is an outer join, DB2 creates a temporary work file. If a correlation name is not used, DB2 assigns a name to this work file and the work file name appears in the TNAME column of the PLAN--TABLE. DB2 uses DSNWFQB(xx) as the name for the work file, where "xx" is the number of the query block (QBLOCKNO) that produced the work file.

The second join change from previous releases of DB2 is the addition of the ON clause. When you use the JOIN keyword to perform an inner or outer join, you must use the ON clause to specify the join predicate. If the clause is omitted, the statement will receive a -104 SQLCODE. The ON clause also has a side benefit: It makes analyzing or debugging a join a little simpler. When using the ON clause, the join predicates immediately follow the two tables being joined. It is no longer necessary to search a long list of predicates to try to determine which are being used by the join and associate those predicates with the right tables. Only two tables can be joined with a single JOIN keyword and that join definition must be followed by the ON clause. If a third table needs to be added to the outer join, the JOIN keyword is repeated, followed by the third table and another ON clause for each additional join predicate. Notice the improved readability in Example 5, a three-way outer join.

The ON clause does not preclude the use of the WHERE clause for specifying additional local predicates. Remember, though, that the join is always performed before applying the WHERE predicates.

There are some rules that must be followed when using the JOIN keyword for an outer join:

* The join conditions specified with the ON clause can only be ANDed together for the outer join. OR and NOT keywords are not allowed.

* Although a left and right outer join can use comparison operators, a full outer join can use only the equal (=) operator.

* Except for a full outer join, only column names can be used in comparisons. A full outer join can use the COALESCE scalar function.

Violations of any of these rules will result in the SQL statement receiving a -338 SQLCODE.

Even though the original tables in the example were created specifying NOT NULL, the result of the outer join can still contain nulls. All three types of outer joins will supply nulls for the columns referenced in the SELECT clause that do not have values assigned to them because of the unmatched condition. An application program needs to be particularly careful when a result contains nulls. There are two possible solutions. First, the program can be coded to detect and handle a returned null value. Second, the scalar function COALESCE can be used to ensure the column with the nonnull value is used. COALESCE is the ANSI SQL92 name given to the VALUE scalar function introduced in DB2 Version 2.3. COALESCE uses a list of columns as an operand and returns the first nonnull value from any column in the list. The result of COALESCE can be assigned to a label using the new SQL AS keyword. The AS keyword allows the assignment of a column, scalar (as in the result of a COALESCE) or function to a label. That label can then be used in the SQL statement in the same way an actual column can be used. Example 6 shows a full outer join using the COALESCE scalar and the AS keyword.

In Example 6, it is possible that only one of the COLNO columns in the two tables being joined will actually have a value. If you look at the sample data in Table 1, FAVORITE.COLNO = 99 has no equivalent row in the COLOR table. If the value from COLOR.COLNO were used, a null would be returned. In contrast, COLOR.COLNO = 70 has no equivalent row in the FAVORITE

table. In this case, if the value from COLOR.COLNO were used, again a null would be returned. So how do you guarantee a real value will always be returned? COALESCE will return the first nonnull value from its list of columns. That value can then be assigned to the label COLOR--NUMBER using the AS clause. The label COLOR--NUMBER can be referenced in the WHERE and ORDER BY clauses and as a column in the final answer set.

Nested Table Expressions

As mentioned earlier, only one table expression can be specified on each side of the join operator. However, there are situations when you want to perform an outer join against the results of another join. No problem. DB2 Version 4.1 introduces the nested table expression, a subselect that replaces the table name on the FROM clause.

Because the nested table expression is treated as a subselect, anything valid in a subselect is valid in the nested table expression. Joins (both outer and inner), WHERE predicates, other subselects, GROUP BY and HAVING are allowed. Just like a subselect, ORDER BY is not allowed.

There are only a few rules to follow when coding a nested table expression. It must be completely enclosed in parentheses (see Example 7) and the result of the nested table expression must be assigned to a correlation name. In the example, the correlation names are TEMPA and TEMPB. The correlation name can be assigned with or without the AS clause. If the columns in the result of a nested table expression are not unique, they must be prefixed with the correlation name. In the example, a nested table expression is used on both sides of the join operator in the FROM clause. Because the nested table expressions are evaluated first, their result sets can then be joined. This overcomes the restriction of being able to specify only one table expression on each side of the join operator. The results can also be used anywhere a qualified column name can be specified.

Nested table expressions can improve the readability of a complex join. Specifying local predicates within the nested table expression makes a three-or-more table join easier to debug. The local predicates for the table are declared where the table name is specified. The join predicate is specified for each table pair with the ON clause, again where the table name is specified. This allows the query to be written in "complete" executable parts.

If the results from two or more nested table expressions are to be joined together using an inner join, the JOIN keyword must be used. Nested table expressions cannot be separated by commas. This is an additional reason for using the JOIN keyword to perform an inner join.

There are more ways nested table expressions can improve your SQL. Replacing a view name with the view definition within the SELECT and building summary data within a query are only two examples. Because the focus here is the outer join, discussing all their uses is beyond the scope of this article. In fact, all the SQL enhancements mentioned in this article have more uses than only enhancing the outer join.

Conclusion

Outer joins are here and they are wonderful. Although they have taken a while to finally arrive, their implementation is easy and straightforward. In most cases, they should perform better than the old SQL JOIN-UNION-NOT EXISTS combination for tables with any sizable amount of data. They are definitely less error-prone. So get rid of the old contrived method of achieving an outer join-like result and start using the real thing: full, left and right outer joins. In fact, you may want to start using the new JOIN keyword when coding inner joins, too.

One final thought: This article is written for DB2 for MVS/ESA Version 4.1. However, outer joins, nested table expressions, COALESCE and the AS clause are all available on DB2 Common Server Version 2.1. The optimizers are different and the EXPLAIN information is not presented the same way, but everything else discussed here is the same. Any DB2 for MVS/ESA SQL coding techniques discussed in this article should port to the current releases of DB2 Common Server.

Table 1: Sample Data

```
COLOR Table FAVORITE Table CARS Table


COLNO  COLORCOLNO  NAMECOLNO  CAR--NAME

10 RED  20 KAREN   50 THUNDERBIRD

20 BLUE 30 CATHERINE   10 CORVETTE

30 GREEN40 LAUREN  60 JEEP GRAND CHEROKEE

40 YELLOW   99 WILLIE

50 MAROON

60 DRIFTWOOD

70 WHITE
```

Example 1: Simple SQL Inner Join

```
SELECT COLOR.COLNO, COLOR, NAME

FROM COLOR, FAVORITE

WHERE COLOR.COLNO = FAVORITE.COLNO


COLNO  COLORNAME

20 BLUE KAREN

30 GREENCATHERINE

40 YELLOW   LAUREN

DSNE610I NUMBER OF ROWS DISPLAYED IS 3
```

Example 2: Pre-V4.1 Equivalent To Left Outer Join

```
SELECT COLOR.COLNO, COLOR, NAME
FROM FAVORITE, COLOR
WHERE COLOR.COLNO = FAVORITE.COLNO
UNION ALL
SELECT COLNO, 'COLOR NOT FOUND', NAME
FROM FAVORITE B
WHERE NOT EXISTS
(SELECT * FROM COLOR
WHERE B.COLNO = COLNO)

COLNO  COLOR  NAME
20 BLUE   KAREN
30 GREEN  CATHERINE
40 YELLOW LAUREN
99 COLOR NOT FOUNDWILLIE
DSNE610I NUMBER OF ROWS DISPLAYED IS 4
```

Example 3: Pre-V4.1 SQL Statement Equivalent To Full Outer Join

```
SELECT COLOR.COLNO, COLOR, NAME
FROM COLOR, FAVORITE
WHERE COLOR.COLNO = FAVORITE.COLNO
UNION ALL
SELECT COLNO, COLOR, 'NO NAME FOR COLOR'
FROM COLOR A
WHERE NOT EXISTS (SELECT * FROM FAVORITE
 WHERE A.COLNO = COLNO)
UNION ALL
```

```
SELECT COLNO, 'COLOR NOT FOUND', NAME
FROM FAVORITE B
   WHERE NOT EXISTS (SELECT * FROM COLOR
 WHERE B.COLNO = COLNO)

COLNO  COLOR NAME
10 RED   NO NAME FOR COLOR
20 BLUE  KAREN
30 GREEN CATHERINE
40 YELLOWLAUREN
50 MAROONNO NAME FOR COLOR
60 DRIFTWOOD NO NAME FOR COLOR
70 WHITE NO NAME FOR COLOR
99 COLOR NOT FOUND WILLIE
DSNE610I NUMBER OF ROWS DISPLAYED IS 8
```

Example 4: DB2 V4.1 Left Outer Join

```
SELECT *
FROM FAVORITE LEFT OUTER JOIN COLOR
ON COLOR.COLNO = FAVORITE.COLNO

COLNO  NAMECOLNO  COLOR
20 KAREN   20 BLUE
30 CATHERINE   30 GREEN
40 LAUREN  40 YELLOW
99 WILLIE  ---------
DSNE610I NUMBER OF ROWS DISPLAYED IS 4
```

Example 5: Three-Way Outer Join

```
SELECT *

FROM COLOR LEFT OUTER JOIN FAVORITE

ON COLOR.COLNO = FAVORITE.COLNO

LEFT OUTER JOIN CARS

ON COLOR.COLNO = CARS.COLNO

WHERE ...
```

Example 6: DB2 V4.1 Full Outer Join Using COALESCE And AS

```
SELECT COALESCE (COLOR.COLNO, FAVORITE.COLNO) AS COLOR--NUMBER,

COLOR, NAME

FROM COLOR FULL OUTER JOIN FAVORITE

  ON COLOR.COLNO = FAVORITE.COLNO


COLOR--NUMBERCOLORNAME

10  RED  ---------

20  BLUE KAREN

30  GREENCATHERINE

40  YELLOW   LAUREN

50  MAROON   ---------

60  DRIFTWOOD---------

70  WHITE---------

99  ---------WILLIE

DSNE610I NUMBER OF ROWS DISPLAYED IS 8
```

Example 7: Nested Table Expression

```
SELECT TEMPA.COLNO, COLOR, NAME, CAR--NAME

FROM (SELECT COLOR.COLNO, COLOR, NAME
```

```
  FROM COLOR LEFT OUTER JOIN FAVORITE

ON COLOR.COLNO = FAVORITE.COLNO) AS TEMPA

LEFT OUTER JOIN

 (SELECT COLOR.COLNO, CAR--NAME

  FROM CARS RIGHT OUTER JOIN COLOR

ON CARS.COLNO = COLOR.COLNO) AS TEMPB

ON TEMPA.COLNO = TEMPB.COLNO
```

About The Author

Willie Favero conducts education and consulting for BMC Software. **Favero** has been a database professional for more than 20 years, dealing primarily with DB2 in the last 12 years. Most of this time was spent as a senior instructor for IBM. He is the author of numerous articles and regularly speaks at national technical conferences. He can be reached via e-mail at 73577.656@compuserve.com.