developerWorks®

MERGE ahead

Introducing the DB2 for i SQL MERGE statement

Karl Hanson

April 12, 2011

DB2 for i 7.1 includes support for the MERGE SQL statement. This article gives an overview of MERGE and includes examples showing how it can be used to simplify applications.

Introduction

As any shade tree mechanic or home improvement handyman knows, you can never have too many tools. Sure, you can sometimes get by with inadequate tools on hand, but the right tools can help complete a job in a simpler, safer, and quicker way. The same is true in programming. New in DB2 for i 7.1, the MERGE statement is a handy tool to synchronize data in two tables. But as you will learn later, it can also do more. You might think of MERGE as doing the same thing you could do by writing a program, but with less work and with simpler notation.

Below is a simple example. Assume two tables, YEARSALES and MONTHSALES, both have columns SKU and QTY, where SKU is a unique key. The goal is to merge data from MONTHSALES into YEARSALES, so the statement might look like this:

```
MERGE INTO yearsales y                -- target
  USING monthsales   m                -- source (table-reference)
  ON ( m.sku = y.sku )                -- comparison
  WHEN MATCHED THEN                   –- matching-condition 1
  UPDATE SET y.qty = y.qty + m.qty    –- row operation 1
  WHEN NOT MATCHED THEN               -- matching-condition 2
  INSERT VALUES(m.sku, m.qty)         –- row operation 2
```

This statement could be paraphrased as: For each row in MONTHSALES, look for a row in YEARSALES with a matching SKU. If a matching row exists, update the QTY column using the MONTHSALES QTY column. If no matching row exists, insert a new row with the SKU and QTY column values of the MONTHSALES row.

In this example, the `Y` suffix on the INTO target and the `M` suffix on USING source are *correlation-names*. Using these to qualify column references improves readability and can avoid ambiguity.

Compare this to the following SQL procedure example that does approximately the same function. No doubt it could be simplified somewhat, but not to the level of the MERGE statement above.

```
CREATE PROCEDURE MRGEXMPL
  LANGUAGE SQL MODIFIES SQL DATA
MG: BEGIN ATOMIC
     DECLARE tgt_sku INT ;
     DECLARE end_tgt INT DEFAULT 0;
     DECLARE tgt CURSOR for
       SELECT sku,qty FROM yearsales ORDER BY sku
                      FOR UPDATE OF qty ;
     DECLARE CONTINUE HANDLER FOR NOT FOUND
       SET end_tgt = 1 ;
     OPEN tgt ;
     FETCH tgt INTO tgt_sku ;

 SC: FOR src CURSOR FOR
       SELECT sku as src_sku, qty as src_qty
         FROM monthsales ORDER BY sku
         DO
   EQ_CHK: IF src_sku = tgt_sku THEN
             WHILE src_sku = tgt_sku AND end_tgt = 0 DO
               UPDATE yearsales SET qty = qty + src_qty
                 WHERE CURRENT OF tgt ;
               FETCH tgt INTO tgt_sku ;
             END WHILE ;
           END IF ;

           IF src_sku > tgt_sku THEN
             WHILE src_sku > tgt_sku AND end_tgt = 0 DO
               FETCH tgt INTO tgt_sku ;
             END WHILE ;
             IF src_sku = tgt_sku AND end_tgt = 0 THEN
               GOTO EQ_CHK ;
             END IF ;
           END IF ;

           IF src_sku < tgt_sku OR end_tgt = 1 THEN
             INSERT INTO yearsales VALUES(src_sku,src_qty);
           END IF ;

     END FOR ;
     CLOSE tgt ;
END MG ;
```

Anatomy of a MERGE statement

A MERGE statement includes:

- One "target" table (or view), that can have rows changed by the MERGE.
- A data "source", called a *table-reference*, that along with the *search-condition*, determines what row operations are performed on the target. This is not limited to a single table; it could be a complex query involving multiple tables and views.
- A comparison, called a *search-condition*, that typically compares a column in the source *table-reference* to a column in the target, to control target row operations.
- One or more *matching-conditions*, that specify INSERT, UPDATE, or DELETE row operations on the target.

INTO clause

This identifies the MERGE target, which can be a table or a view that allows changes to underlying rows. One INTO clause is required, and it must identify an existing table or view. The requester must have authority to insert, update, or delete target rows.

USING clause

This specifies the source data, as a *table-reference*. Some common types of *table-reference* used in a MERGE are:

- A single table or view, as in the example above:
  ```
  USING sales AS src
  ```
- A SELECT statement, such as:
  ```
  USING ( SELECT qty FROM sales ) AS src
  ```
- A table function, such as:
  ```
  USING ( TABLE( salesfunc(1) ) ) AS src (col1, col2)
  ```

Regardless of the type of *table-reference* in the USING clause, it represents a logical result set of 0 – N rows.

ON clause

The ON clause contains a comparison (called *search-condition*) that controls how source *table-reference* rows affect target rows, as specified in WHEN clauses. A common comparison is between a column in the source and a column in the target, such as:

```
ON ( src.partnbr = tgt.partnbr )
```

However, there is no requirement to reference columns in either the source or target, and more elaborate expressions are possible, including subqueries.

For each source row, the ON comparison is evaluated. When it evaluates as true, a WHEN MATCHED clause may be executed. When false, a WHEN NOT MATCHED clause may be executed.

A WHEN MATCHED or WHEN NOT MATCHED clause may also return an error using the SIGNAL statement instead of performing a row operation. If a source row does not qualify for any WHEN MATCHED or WHEN NOT MATCHED clause, it is ignored and has no effect on the target.

WHEN clause

Each WHEN clause contains a *matching-condition*. The minimum required *matching-condition* is either MATCHED or NOT MATCHED followed by THEN followed by one row operation. Row operations allowed for each *matching-condition* are:

```
    MATCHED : UPDATE or DELETE
    NOT MATCHED : INSERT
```

The simplest forms of matching-conditions are:

```
    WHEN MATCHED THEN <operation>
    WHEN NOT MATCHED THEN <operation>
```

These simple forms are often adequate. However, an "AND *search-condition*" extension can be used for more specific or granular selection:

```
    WHEN MATCHED AND search-condition THEN <operation>
    WHEN NOT MATCHED AND search-condition THEN <operation>
```

This *search-condition* after an AND is in addition to, or augments, the comparison in the ON clause.

Although one WHEN MATCHED clause and one WHEN NOT MATCHED clause are often sufficient, any number of WHEN clauses are allowed. The following book list example shows how more than two WHEN clauses can be useful.

A WHEN MATCHED or WHEN NOT MATCHED instead of performing a row operation clause may also return an error using the SIGNAL statement. This can be useful to add coherence checks to a MERGE, and end the MERGE when a check fails. In the previous example, say a source row should never identify an obsolete SKU. You can augment the statement as shown below, to detect an obsolete SKU and return an error.

```
WHEN MATCHED AND y.status = ‘O’ THEN –- if obsolete SKU
 SIGNAL SQLSTATE VALUE ‘75002’ -- quit with specific error
        SET MESSAGE_TEXT = ‘Obsolete SKU’
    WHEN MATCHED THEN
      UPDATE SET y.qty = y.qty + m.qty
```

Row operations in WHEN clauses

A single row in the target table is affected each time a WHEN clause operation is performed. For the MATCHED case, an existing row may be updated or deleted. For the NOT MATCHED case, a new row may be inserted.

Only rows in the target table are affected, so the syntax of INSERT, UPDATE, and DELETE operations in a WHEN clause does not include the target table name, as in standalone INSERT, UPDATE, or DELETE statements. This is shown in the first MERGE example above.

For INSERT, the VALUES clause supplies column values for the new row. If one or more values are not supplied for target columns, the normal rules for assigning null or default values apply. Special values NULL or DEFAULT can be explicitly coded in the VALUES clause, the same as in a standalone INSERT statement. Likewise, if host variables or parameter markers are used, NULL values can be supplied the same as in a standalone INSERT statement.

Column names may also be specified to indicate which columns are being inserted into. In the first example above, the INSERT of the MERGE above could be:

```
WHEN NOT MATCHED THEN
      INSERT(SKU,QTY) VALUES(M.SKU, M.QTY)
```

An UPDATE modifies one existing row in the target. The SET clause supplies any new column values for that row. Like a standalone UPDATE statement, the syntax for each column to be modified is "`column-name = < value >`". See the MERGE statement section in the DB2 for i SQL Reference for details and options of the SET clause within MERGE.

A DELETE simply removes one existing row in the target. No other clauses are allowed for this row operation in a MERGE.

Possible INSERT and UPDATE column values

The VALUES and SET clauses specify new values for columns in MERGE target rows. Often column values come from the USING *table-reference* result set row, such as in the example above:

```
WHEN NOT MATCHED THEN INSERT VALUES(M.SKU, M.QTY)
```

But column values can be provided in other ways, and each column value is independent of all others in a given VALUES or SET clause. Here are a few alternative examples.

- Literals:
  ```
  INSERT VALUES( M.SKU, 100 )
  ```
- SQL built-in functions:
  ```
  INSERT VALUES( M.SKU, MAX(M.QTY,100) )
  ```
- SQL special registers:
  ```
  INSERT (SKU,UTC_TIME)
              VALUES( M.SKU, CURRENT TIME – CURRENT TIMEZONE )
  ```
- SQL expressions:
  ```
  INSERT (PROPERTY_ID,TIMES_SHOWN)
      VALUES( S.ID ,
      ( SELECT COUNT(*) FROM SALES_EVENTS
      WHERE PROPERTY_ID = S.ID ) )
  ```
- Host variables:
  ```
  INSERT (PROPERTY_ID,TIMES_SHOWN)
              VALUES( S.ID , :Shown )
  ```
- Parameter markers:
  ```
  INSERT (PROPERTY_ID,TIMES_SHOWN)
              VALUES( S.ID , CAST(? AS INT) )
  ```
- Global variables:
  ```
  INSERT (PROPERTY_ID,TIMES_SHOWN)
              VALUES( S.ID , SHOWNGVAR )
  ```

Just like for standalone INSERT or UPDATE statements, the data type of a supplied column value must be compatible with the corresponding column data type.

Atomicity options

MERGE supports three options:

1. `ATOMIC` (the default)
2. `NOT ATOMIC STOP ON SQLEXCEPTION`
3. `NOT ATOMIC CONTINUE ON SQLEXCEPTION`

For most MERGE applications, `ATOMIC` should be used. When running under commitment control, this means that if any error condition is detected, all changes made by the MERGE are undone.

The other options may be useful for unique situations. When using `NOT ATOMIC STOP ON SQLEXCEPTION`, if an error occurs during an individual row operation, the MERGE ends, but changes for all previous (successful) row operations remain in effect. When using `NOT ATOMIC CONTINUE ON SQLEXCEPTION`, if an error occurs during an individual row operation, the MERGE continues to process all remaining source rows, so any number of row operation errors are tolerated.

Selection of multiple WHEN clauses

Many MERGE statements have one WHEN MATCHED clause and one WHEN NOT MATCHED clause, but more WHEN MATCHED or WHEN NOT MATCHED clauses can be specified. In that case you might wonder which WHEN is picked for a particular source *table-reference* row? This works like a CASE control statement in an SQL routine or a switch statement in the C programming language.

Assume a MERGE statement with several WHEN clauses:

```
MERGE INTO trgtbl AS t
  USING ( SELECT partno, cost, … FROM srctbl ) AS s
  ON ( s.partno = t.partno )
  WHEN MATCHED AND c.cost > 100 THEN
  UPDATE SET …
  WHEN MATCHED THEN
  UPDATE SET …
  WHEN NOT MATCHED AND s.cost > 100 THEN
  INSERT VALUES( … )
  WHEN NOT MATCHED THEN
  INSERT VALUES( … )
```

Two WHEN MATCHED clauses and two WHEN NOT MATCHED clauses are present. When a row from the USING *table-reference* result set is processed, logically each WHEN clause is evaluated in the order they are written. The first one that evaluates completely true is picked. So if the condition in the ON clause is true (that is, the PARTNO column values are equal), then all WHEN MATCHED clauses are candidates for selection. Logically the first WHEN MATCHED that evaluates true is picked and all others are bypassed. In this example if the input COST column has a value of 101, the AND *search-condition* of the first WHEN MATCHED clause is true, so the UPDATE for that WHEN clause is run. The same holds true for multiple WHEN NOT MATCHED clauses.

A MERGE can be coded such that one or more rows from the USING *table-reference* result set do not qualify for **any** WHEN clause. All such rows are ignored. In fact the MERGE statement has an optional ELSE IGNORE clause that can be specified after all WHEN clauses. The ELSE IGNORE clause is for readability only – MERGE processing is identical whether or not the clause is present.

A simple MERGE example

Assume two tables exist, and we need to merge data from one into the other.

The part inventory table INVENTORY table has these columns:

```
PART_NO INTEGER UNIQUE KEY
QTY_ONHAND INTEGER
DESCRIPTION VARCHAR(400)
DATE_ADDED   DATE
```

The new parts received NEW_RECEIVED table has these columns:

```
PART_NO INTEGER UNIQUE KEY
QTY_RCVD INTEGER
DESCRIPTION  VARCHAR(400)
```

The NEW_RECEIVED table contains information about new inventory received from suppliers. The INVENTORY table can be updated with this information using a MERGE statement like this:

```
MERGE INTO inventory AS i
 USING ( SELECT part_no, qty_rcvd, description FROM new_received
 WHERE qty_rcvd > 0 ) AS r
 ON ( r.part_no = i.part_no)
 WHEN MATCHED THEN
  UPDATE SET qty_onhand = qty_onhand + r.qty_rcvd
 WHEN NOT MATCHED THEN
  INSERT VALUES( r.part_no, r.qty_rcvd, r.description,
                     CURRENT DATE )
```

When this statement is run, each row from the USING result set (from NEW_RECEIVED table) is compared to rows in the target INVENTORY table, using the PART_NO column of both. If a matching INVENTORY row is found, the QTY_ONHAND column is incremented by the value of QTY_RCVD from the NEW_RECEIVED table. If no matching INVENTORY row is found, a new row is inserted with column values from the USING result set, from the NEW_RECEIVED table. Note that the new row's column value for DATE_ADDED is supplied via the CURRENT DATE special register.

A non-trivial practical example

Consider a book club where the members read a book and meet regularly to discuss the book. The book members also discuss future book candidates and select the next one to read. A simple spreadsheet is used to record a list of names and authors of the books read in the past, as well as books recommended but not yet read. Periodically the list is updated with books read or proposed. This example shows how the book list could be stored in DB2 for i tables and managed using a MERGE statement. The goals are:

1. Minimize data entry of book titles and authors
2. Avoid duplicate book entries
3. Simplify updates to the book list

Creating the book list database

As shown in SQL DDL (Data Definition Language) below, objects used for the book list are:

1. Book registry table—for title and author
2. Book log table—for accumulating activity over time
3. Meeting log table—for book activity from the most recent book club meeting

For each new book, a row with the book's title and author is inserted into the book registry table. The BookID column is an IDENTITY column, so a unique number is automatically assigned for each book when its row is inserted.

Once a book's row exists in the registry table, ongoing activity can be accumulated in the book log table using its unique BookID value, rather than re-entering title and author strings. Rows in the meeting log table also contain the BookID, and the idea is to merge meeting log data into accumulated data in the book log table. Both the meeting log and book log tables use referential integrity constraints linked to the book registry BookID column, ensuring every row in the log tables has a corresponding row in the registry table.

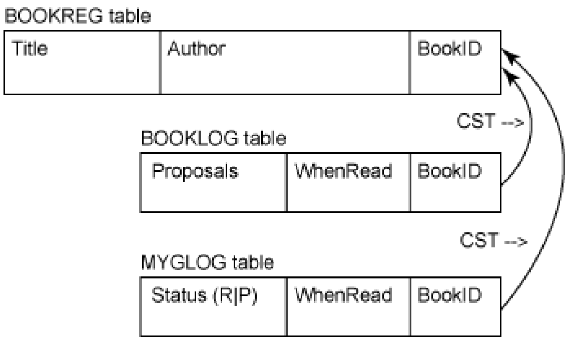
To see the entire list of book activity, a SQL view (not shown) can be used that joins book title and author columns from the registry with activity from the log.

Below is the SQL DDL to implement the book list database.

```
create schema bookinf ;

create table bookinf.bookreg  -- book register
( title  char(60),          -- cover title
  author char(40),          -- author
bookID integer as identity unique ) ;

create table bookinf.booklog  -- book log
( bookID integer  references bookinf.bookreg (bookID),
  proposals smallint,       -- no. times proposed as candidate
whenread date ) ; -- when book was actually read by club

create table bookinf.mtglog  -- meeting log
( bookID integer references bookinf.bookreg (bookID),
status char(1)
whenread date ) ;
```

Figure 1 shows these objects and their relationships.

Figure 1. Book list database



Book list data

Below is an example of rows in the book registry table and a meeting log table. The unique BOOKID column value, generated whenever a new book (row) is inserted into the book registry table, is used in the meeting log table to correlate book status with the registry.

```
BOOKREG Table:
TITLE AUTHOR BOOKID
 In the Heart of the Sea Nathaniel Philbrick 1
 The River of Doubt Candice Millard 2
 Isaac's Storm Erik Larson 3
 The Shipping News Annie Proulx 4

MTGLOG Table:
 BOOKID  STATUS   WHENREAD
 1         P    <null>
 2         P    <null>
 3         P    <null>
 4         R     2011-01-31
```

Updating book log with meeting log data

Finally, the MERGE statement below can be used to merge data from the meeting log table into the book log table, containing book information accumulated over time.

```
MERGE INTO bookinf.booklog  log
 USING ( SELECT bookID, status, whenread FROM bookinf.mtglog ) mtg
 ON mtg.bookID = log.bookID )
 WHEN MATCHED AND mtg.status = 'R' THEN
 UPDATE SET log.whenread = mtg.whenread
 WHEN MATCHED AND mtg.status = 'P' THEN
 UPDATE SET log.proposals = log.proposals + 1
 WHEN NOT MATCHED AND mtg.status = 'P' THEN
 INSERT ( bookID, mentions, proposals, whenread )
 VALUES( mtg.bookID, 0, 1, null )
 WHEN NOT MATCHED AND mtg.status = 'R' THEN
 INSERT ( bookID, mentions, proposals, whenread )
        VALUES( mtg.bookID, 0, 1, mtg.whenread ) ;
```

1. The query in the USING could be more complex if desired, including a WHERE clause. Column values (select list) could be derived from other tables, user defined or built-in functions, special registers, and so on.
2. Column values used in UPDATE SET and INSERT VALUES are derived from other column values from the USING result set.
3. Extra conditions are used on both WHEN MATCHED and WHEN NOT MATCHED clauses to determine which UPDATE or INSERT to perform.

Conclusion

As you can see, the MERGE statement can be a valuable addition to your SQL tools and techniques. At first glance it might appear complex, but after trying it out, you may find it simpler and more convenient than alternative solutions.

Related topics

- The ON clause contains a comparison (called a *search-condition*)
- DB2 for i manuals