KIMBALL GROUP KIMBALL UNIVERSITY

www.kimballgroup.com Number 107, November 6, 2008

Design Tip #107 Using the SQL MERGE Statement for Slowly Changing Dimension Processing

By Warren Thornthwaite

Most ETL tools provide some functionality for handling slowly changing dimensions. Every so often, when the tool isn't performing as needed, the ETL developer will use the database to identify new and changed rows, and apply the appropriate inserts and updates. I've shown examples of this code in the Data Warehouse Lifecycle in Depth class using standard INSERT and UPDATE statements. A few months ago, my friend Stuart Ozer suggested the new MERGE command in SQL Server 2008 might be more efficient, both from a code and an execution perspective. His reference to a blog by Chad Boyd on MSSQLTips.com gave me some pointers on how it works. MERGE is a combination INSERT, UPDATE and DELETE that provides significant control over what happens in each clause.

This example handles a simple customer dimension with two attributes: first name and last name. We are going to treat first name as a Type 1 and last name as a Type 2. Remember, Type 1 is where we handle a change in a dimension attribute by overwriting the old value with the new value; Type 2 is where we track history by adding a new row that becomes effective when the new value appears.

Step 1: Overwrite the Type 1 Changes

I tried to get the entire example working in a single MERGE statement, but the function is deterministic and only allows one update statement, so I had to use a separate MERGE for the Type 1 updates. This could also be handled with an update statement since Type 1 is an update by definition.

```
MERGE INTO  dbo.Customer_Master AS CM
USING Customer_Source AS CS
ON (CM.Source_Cust_ID = CS.Source_Cust_ID)
WHEN MATCHED AND -- Update all existing rows for Type 1 changes
        CM.First_Name <> CS.First_Name
        THEN UPDATE SET CM.First_Name = CS.First_Name
```

This is a simple version of the MERGE syntax that says merge the Customer_Source table into the Customer_Master dimension by joining on the business key, and update all matched rows where First_Name in the master table does not equal the First_Name in the source table.

Step 2: Handle the Type 2 Changes

Now we'll do a second MERGE statement to handle the Type 2 changes.
This is where things get a little tricky because there are several steps involved in tracking Type 2 changes. Our code will need to:

1. Insert brand new customer rows with the appropriate effective and end dates
2. Expire the old rows for those rows that have a Type 2 attribute change by setting the appropriate end date and current_row flag = 'n'
3. Insert the changed Type 2 rows with the appropriate effective and end dates and current_row

flag = ‘y’

The problem with this is it’s one too many steps for the MERGE syntax to handle. Fortunately, the MERGE can stream its output to a subsequent process. We’ll use this to do the final insert of the changed Type 2 rows by INSERTing into the Customer_Master table using a SELECT from the MERGE results. This sounds like a convoluted way around the problem, but it has the advantage of only needing to find the Type 2 changed rows once, and then using them multiple times.

The code starts with the outer INSERT and SELECT clause to handle the changed row inserts at the end of the MERGE statement. This has to come first because the MERGE is nested inside the INSERT. The code includes several references to getdate; the code presumes the change was effective yesterday (getdate()-1) which means the prior version would be expired the day before (getdate()-2). Finally, following the code, there are comments that refer to the line numbers

```
1       INSERT INTO Customer_Master
2       SELECT Source_Cust_ID, First_Name, Last_Name, Eff_Date, End_Date, Current_Flag
3       FROM
4            (  MERGE Customer_Master    CM
5               USING Customer_Source    CS
6              ON (CM.Source_Cust_ID = CS.Source_Cust_ID)
7             WHEN NOT MATCHED THEN
8               INSERT VALUES (CS.Source_Cust_ID, CS.First_Name, CS.Last_Name,
                convert(char(10), getdate()-1, 101), '12/31/2199', 'y')
9            WHEN MATCHED AND CM.Current_Flag = 'y'
10              AND   (CM.Last_Name <> CS.Last_Name )  THEN
11                  UPDATE SET CM.Current_Flag = 'n', CM.End_date = convert(char(10), getdate()-
                    2, 101)
12           OUTPUT $Action Action_Out, CS.Source_Cust_ID, CS.First_Name, CS.Last_Name,
             convert(char(10), getdate()-1, 101) Eff_Date, '12/31/2199' End_Date, 'y'Current_Flag
13           ) AS MERGE_OUT
14      WHERE  MERGE_OUT.Action_Out = 'UPDATE';
```

Code Comments

Lines 1-3 set up a typical INSERT statement. What we will end up inserting are the new values of the Type 2 rows that have changed.

Line 4 is the beginning of the MERGE statement which ends at line 13. The MERGE statement has an OUTPUT clause that will stream the results of the MERGE out to the calling function. This syntax defines a common table expression, essentially a temporary table in the FROM clause, called MERGE_OUT.

Lines 4-6 instruct the MERGE to load Customer_Source data into the Customer_Master dimension table.

Line 7 says when there is no match on the business key, we must have a new customer, so Line 8 does the INSERT. You could parameterize the effective date instead of assuming yesterday’s date.

Lines 9 and 10 identify a subset of the rows with matching business keys, specifically, where it’s the current row in the Customer_Master AND any one of the Type 2 columns is different.

Line 11 expires the old current row in the Customer_Master by setting the end date and current row flag to ‘n’.

Line 12 is the OUTPUT clause which identifies what attributes will be output from the MERGE, if any. This is what will feed into the outer INSERT statement. The $Action is a MERGE function that tells us what part of the MERGE each row came from. Note that the OUTPUT can draw from both the source and the master. In this case, we are outputting source attributes because they contain the new Type 2 values.

Line 14 limits the output row set to only the rows that were updated in Customer_Master. These correspond to the expired rows in Line 11, but we output the current values from Customer_Source in Line 12.

Summary

The big advantage of the MERGE statement is being able to handle multiple actions in a single pass of the data sets, rather than requiring multiple passes with separate inserts and updates. A well tuned optimizer could handle this extremely efficiently.