One-Step Insert and Update

SQL Server provides a way to update some records, while inserting or even deleting others, with a single command.

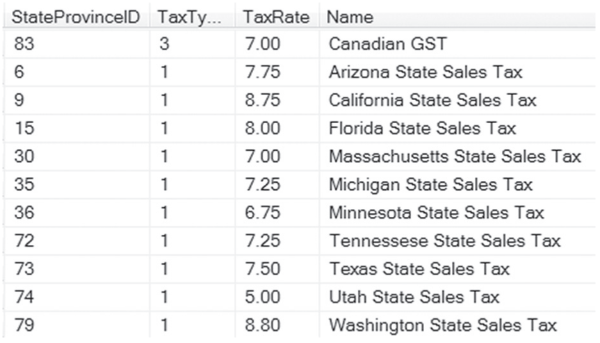
Tamar E. Granor, Ph.D.

This series of articles looks at features of SQL Server's T-SQL that make tasks easier than they are with VFP's SQL sub-language. This time, we look at the MERGE command that lets you process complex updates with a single statement.

One of the questions I often see in VFP forums is how to update some records in a table while inserting others. The typical situation is that there's a second table of new and updated information and the goal is to update the records that already exist and add those that don't.

You can do this in VFP, but you can't do it with a single command. SQL Server 2008 and later, however, has a command that handles not only this task (sometimes called "upsert"), but a variety of similar tasks.

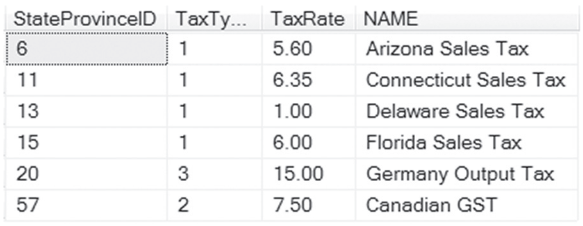
To explore this problem, we're going to consider a table of tax rates. For the SQL Server examples, we'll use a copy of the SalesTaxRate table in the AdventureWorks 2008 database. For the VFP examples, we'll create a simplified version of the same table to work on. Figure 1 shows the fields we'll look at and some of the data.



StateProvinceID	TaxTy...	TaxRate	Name
83	3	7.00	Canadian GST
6	1	7.75	Arizona State Sales Tax
9	1	8.75	California State Sales Tax
15	1	8.00	Florida State Sales Tax
30	1	7.00	Massachusetts State Sales Tax
35	1	7.25	Michigan State Sales Tax
36	1	6.75	Minnesota State Sales Tax
72	1	7.25	Tennessese State Sales Tax
73	1	7.50	Texas State Sales Tax
74	1	5.00	Utah State Sales Tax
79	1	8.80	Washington State Sales Tax

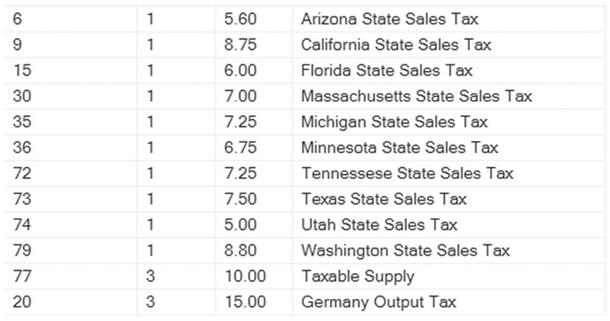
Figure 1. To demonstrate "upserts," we'll make changes to this sales tax date.

We'll also have a second table (or cursor) containing the new tax rates, as in Figure 2. The problem then is to update the TaxRate column in the original table for any items that already exist and add a new record for any that don't. Partial results are shown in Figure 3.



StateProvinceID	TaxTy...	TaxRate	NAME
6	1	5.60	Arizona Sales Tax
11	1	6.35	Connecticut Sales Tax
13	1	1.00	Delaware Sales Tax
15	1	6.00	Florida Sales Tax
20	3	15.00	Germany Output Tax
57	2	7.50	Canadian GST

Figure 2. A similarly-structured table shows the new data for some records.



6	1	5.60	Arizona State Sales Tax
9	1	8.75	California State Sales Tax
15	1	6.00	Florida State Sales Tax
30	1	7.00	Massachusetts State Sales Tax
35	1	7.25	Michigan State Sales Tax
36	1	6.75	Minnesota State Sales Tax
72	1	7.25	Tennessese State Sales Tax
73	1	7.50	Texas State Sales Tax
74	1	5.00	Utah State Sales Tax
79	1	8.80	Washington State Sales Tax
77	3	10.00	Taxable Supply
20	3	15.00	Germany Output Tax

Figure 3. After adding the two new taxes and updating the others, the results include these records.

Before jumping into solutions, it's probably worth pointing out that in a real application, you'd probably track tax rates with dates they apply. In that case, you'd add a new record every time a tax rate changes and not modify or delete existing records. But we'll ignore that reality for demonstration purposes.

Upsert with VFP

In VFP, this is a two-step process. You can't update and insert records in a single command. So we use UPDATE first to update the records that exist and then use INSERT to add the new ones. Listing 1 shows the code; the complete code, including that to create the two cursors is included in this month's downloads as UpdateInsert.PRG.

Listing 1. In VFP, you can't "upsert" in a single command.

```
* First update existing records
UPDATE TaxRates ;
  SET TaxRate = NewRates.TaxRate ;
  from NewRates ;
  WHERE TaxRates.StateProvinceID = ;
        NewRates.StateProvinceID ;
    AND TaxRates.TaxType = NewRates.TaxType
```

```
* Next insert new records
INSERT INTO TaxRates ;
  SELECT * ;
    FROM NewRates ;
    WHERE NOT exists (;
      SELECT * ;
        FROM TaxRates TR2 ;
        WHERE TR2.StateProvinceID = ;
              NewRates.StateProvinceID ;
          AND TR2.TaxType = NewRates.TaxType)
```

The UPDATE command is fairly straightforward. It matches records in the two tables (cursors) based on StateProvinceID and TaxType. If the record exists in both tables, the TaxRate from the NewRates table replaces the one in TaxRates.

The INSERT is a little more complex because we need to figure out which records to insert. The SELECT looks for records in NewRates that have no match in TaxRates, based on StateProvinceID and TaxType. Those records are then inserted into TaxRates.

Upsert with SQL Server

SQL Server offers a way to handle this problem in a single command. MERGE lets you synchronize data from two tables. You specify the target table (into which the data is merged), the source table (from which data is drawn), the relationship between the tables (that is, the join conditions) and what to do in each of several cases.

Upsert, as in the example here, is the simplest form of MERGE. Listing 2 shows the T-SQL code equivalent to Listing 1; full code, including creating and populating the two tables, is included in this month's downloads as MergeUpdateInsert.SQL. Note that MERGE must be terminated with a semi-colon.

Listing 2. In T-SQL, you can specify the update and the insert in one MERGE command.

```
MERGE INTO #TaxRates TR
  USING #NewRates NR
  ON TR.StateProvinceID = NR.StateProvinceID
  AND TR.TaxType = NR.TaxType
  WHEN MATCHED THEN
    UPDATE SET TaxRate = NR.TaxRate
  WHEN NOT MATCHED THEN
    INSERT (StateProvinceID, TaxType,
            TaxRate, NAME )
      VALUES (NR.StateProvinceID, NR.TaxType,
              NR.TaxRate, NR.NAME );
```

Here, INTO indicates that #TaxRates (with alias TR) is the target table, the one we want to update. USING says that #NewRates (with alias NR) is the source table, the one that contains the new data. The ON clause specifies the join condition; as in the VFP example, we match records based on StateProvinceID and TaxType.

The interesting part here is the series of WHEN clauses. WHEN MATCHED indicates what to do when we find a matching record. In this case, we use UPDATE to change the tax rate. WHEN NOT MATCHED says what to do when there's no matching record. In this case, we INSERT one.

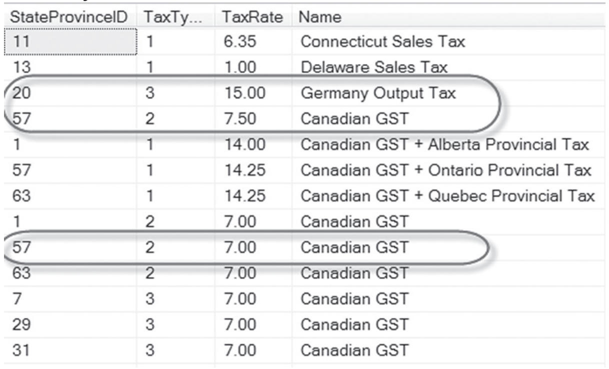
Note the variant format of both the UPDATE and INSERT commands here. We omit both the name of the table to operate on and the WHERE clause that indicates how to match records, because we've already specified them.

Also, be aware that the ON clause in MERGE doesn't filter out any records from the source; it just determines which case a record falls into. That is, every record from the source table will be processed. You might expect that the query in Listing 3 (called MergeUpdateFilterInOn.SQL in this month's downloads) would let you process only the sales tax items, that is, those with TaxType = 1.

Listing 3. This query looks like it would update or insert only the items with TaxType=1, but it creates records it shouldn't.

```
MERGE INTO #TaxRates TR
  using #NewRates NR
  ON TR.StateProvinceID = NR.StateProvinceID
  AND TR.TaxType = NR.TaxType
  AND TR.TaxType = 1
  WHEN MATCHED THEN
    UPDATE SET TaxRate = NR.TaxRate
  WHEN NOT MATCHED THEN
    INSERT (StateProvinceID, TaxType,
            TaxRate, NAME )
      VALUES (NR.StateProvinceID, NR.TaxType,
              NR.TaxRate, NR.NAME );
```

However, as Figure 4 shows, it adds new rows for the records in #NewRates with TaxType set to 2 or 3. Why does it behave this way? Because the condition in the ON clause is used only to sort the records into "matches" and "doesn't match." So, the records where TaxType is 2 or 3 get sorted into the "doesn't match" bucket, and new records are added. The next section of this paper shows how to have more choices than just "matches" and "doesn't match," but for a case like this one, there's actually an even easier solution.



StateProvinceID	TaxTy...	TaxRate	Name
11	1	6.35	Connecticut Sales Tax
13	1	1.00	Delaware Sales Tax
20	3	15.00	Germany Output Tax
57	2	7.50	Canadian GST
1	1	14.00	Canadian GST + Alberta Provincial Tax
57	1	14.25	Canadian GST + Ontario Provincial Tax
63	1	14.25	Canadian GST + Quebec Provincial Tax
1	2	7.00	Canadian GST
57	2	7.00	Canadian GST
63	2	7.00	Canadian GST
7	3	7.00	Canadian GST
29	3	7.00	Canadian GST
31	3	7.00	Canadian GST

Figure 4. The flawed query in Listing 3 adds new rows for the records with TaxType something other than 1, rather than ignoring them.

To process only some records from the source table, use a CTE before the MERGE command. Listing 4 (MergeUpdateFilterCTE.SQL in this month's downloads) shows how to use a CTE to process only the sales tax records.

Listing 4. When you want to process only some of the records in the source table, use a CTE to select those records before the MERGE command.

```
WITH csrSalesTaxOnly
  (StateProvinceID ,TaxType ,TaxRate ,NAME)
AS
  (SELECT * FROM #NewRates WHERE TaxType = 1)

MERGE INTO #TaxRates TR
  using csrSalesTaxOnly NR
  ON TR.StateProvinceID = NR.StateProvinceID
  AND TR.TaxType = NR.TaxType
  WHEN MATCHED THEN
    UPDATE SET TaxRate = NR.TaxRate
  WHEN NOT MATCHED THEN
    INSERT (StateProvinceID, TaxType,
            TaxRate, NAME )
      VALUES (NR.StateProvinceID, NR.TaxType,
              NR.TaxRate, NR.NAME );
```

You might also use a CTE to do some aggregation before the MERGE. For example, say you're tracking the maximum monthly sales for a group of products. If the new data shows individual sales for the month, you might compute the totals by product in a CTE and then use MERGE to update those already there while adding products not previously tracked.

The MERGE command can actually do more than simple upsert. In some cases, you can DELETE rather than UPDATE or INSERT. In addition, you can test for more than the simple "does it match a record or not?"

Filtering matches

You can add a filter condition in WHEN MATCHED so that the action applies to only some of the matched records. This provides an alternative solution to the problem of filtering out some source records. Listing 5 shows how to use this approach to change only those records representing sales tax.

Listing 5. Another way to handle filtering of some source rows is to put the filter condition in the WHEN MATCHED clause.

```
MERGE INTO #TaxRates TR
  using #NewRates NR
  ON TR.StateProvinceID = NR.StateProvinceID
  AND TR.TaxType = NR.TaxType
  WHEN MATCHED AND TR.TaxType = 1 THEN
    UPDATE SET TaxRate = NR.TaxRate
  WHEN NOT MATCHED AND NR.TaxType = 1 THEN
    INSERT (StateProvinceID, TaxType,
            TaxRate, NAME )
      VALUES (NR.StateProvinceID, NR.TaxType,
              NR.TaxRate, NR.NAME );
```

In fact, MERGE supports up to two WHEN MATCHED cases; when there are two, the first must add an additional condition. That is, we can say "If we find a matching record and this condition is true, do this, but if we find a matching record and this condition is not true, do that."

We can modify the earlier example so that if a tax rate is updated to be 0, we delete the record. Listing 6 shows the modified MERGE command. (MergeUpdateDelete.SQL in this month's downloads contains the complete program.)

Listing 6. You can have two WHEN MATCHED clauses, one with a condition and one without. Here, the first is used to delete tax rates that have become zero.

```
MERGE INTO #TaxRates TR
  USING #NewRates NR
  ON TR.StateProvinceID = NR.StateProvinceID
  AND TR.TaxType = NR.TaxType
  WHEN MATCHED AND NR.TaxRate = 0 THEN
    DELETE
  WHEN MATCHED THEN
    UPDATE SET TaxRate = NR.TaxRate
  WHEN NOT MATCHED THEN
    INSERT (StateProvinceID, TaxType,
            TaxRate, NAME )
      VALUES (NR.StateProvinceID, NR.TaxType,
              NR.TaxRate, NR.NAME );
```

The command here is identical to that one in Listing 2, except for the first WHEN MATCHED clause. That one checks whether the new tax rate is 0, and if so, deletes the record.

We can do something similar with VFP, but doing the exact same thing is actually tricky. The similar thing is to simply add a DELETE command at the end of **Listing 1** that removes any record with a tax rate of 0. However, the T-SQL command in **Listing 2** allows you to insert a record with a 0 tax rate and doesn't delete it. To do that in VFP, you have to play with the DELETED setting, so that you can delete an existing record without adding it back. (Presumably, of course, you don't actually want to add 0 tax rate records at all, so the VFP version would be more accurate. You can accomplish the same thing in T-SQL by removing those records from the source cursor before the MERGE command.)

Unmatched target records

The WHEN NOT MATCHED clause refers to records in the source for which there's no match in the target. In fact, you can add the optional BY TARGET clause to clarify, as in Listing 7.

Listing 7. WHEN NOT MATCHED is equivalent to WHEN NOT MATCHED BY TARGET

```
MERGE INTO #TaxRates TR
  USING #NewRates NR
  ON TR.StateProvinceID = NR.StateProvinceID
  AND TR.TaxType = NR.TaxType
  WHEN MATCHED THEN
    UPDATE SET TaxRate = NR.TaxRate
  WHEN NOT MATCHED BY TARGET THEN
    INSERT (StateProvinceID, TaxType,
            TaxRate, NAME )
      VALUES (NR.StateProvinceID, NR.TaxType,
              NR.TaxRate, NR.NAME );
```

You can also include a case for when a record appears in the target table, but isn't matched in the source table. For the example we've been looking at, that isn't useful, since we specified that the source table contains only changed records. But suppose instead a complete list of current sales tax values is provided annually, and the task is to update the built-in sales tax table. In that case, we want to do three things:

Update records that already exist;

- Add new records to the master table;
- Remove records from the master table that aren't in the new list.

The WHEN NOT MATCHED BY SOURCE clause makes that possible. It lets you specify what to do with items in the target table that aren't in the source table. Listing 8 shows code that implements the strategy above. The complete example, included in this month's downloads as MergeUpdateDeleteBySource.SQL, takes a new approach to creating the #NewRates table of new tax rates; it copies the existing table and then makes a series of changes to represent a complete list of current tax rates.

Listing 8. In some cases, you want to take action when there's a record in the target that has no match in the source. Here, such records are deleted.

```
MERGE INTO #TaxRates TR
  USING #NewRates NR
  ON TR.StateProvinceID = NR.StateProvinceID
  AND TR.TaxType = NR.TaxType
  WHEN MATCHED AND NR.TaxRate = 0 THEN
    DELETE
  WHEN MATCHED THEN
    UPDATE SET TaxRate = NR.TaxRate
  WHEN NOT MATCHED BY TARGET THEN
    INSERT (StateProvinceID, TaxType,
            TaxRate, NAME )
      VALUES (NR.StateProvinceID, NR.TaxType,
              NR.TaxRate, NR.NAME )
  WHEN NOT MATCHED BY SOURCE THEN
    DELETE;
```

Of course, in this case, you might simply delete all records in the existing table and replace them with the contents of the new table. For this example, with just a handful of records, that's probably not a bad choice. But in a production application, where the source and target tables might have thousands or even millions of records, simply making the necessary changes seems far more efficient.

What did I do?

MERGE has an optional OUTPUT clause that lets you see exactly what it did. When you use it, MERGE returns one row for each row updated, inserted or deleted by the command. You can use the special $action identifier to include the action taken on each affected row, as well as the special Inserted and Deleted tables.

For example, Listing 9 (included in this month's downloads as MergeUpdateInsertOutput.SQL) shows the query in Listing 2 with an OUTPUT clause. In this case, we show the action taken on the row, any values inserted and any deleted. Figure 5 shows the result.

Listing 9. Use the OUTPUT clause to see what MERGE actually did.

```
MERGE INTO #TaxRates TR
  using #NewRates NR
  ON TR.StateProvinceID = NR.StateProvinceID
  AND TR.TaxType = NR.TaxType
  WHEN MATCHED THEN
    UPDATE SET TaxRate = NR.TaxRate
  WHEN NOT MATCHED THEN
    INSERT (StateProvinceID, TaxType,
            TaxRate, NAME )
      VALUES (NR.StateProvinceID, NR.TaxType,
              NR.TaxRate, NR.NAME )
  OUTPUT $ACTION, INSERTED.*, DELETED.*;
```

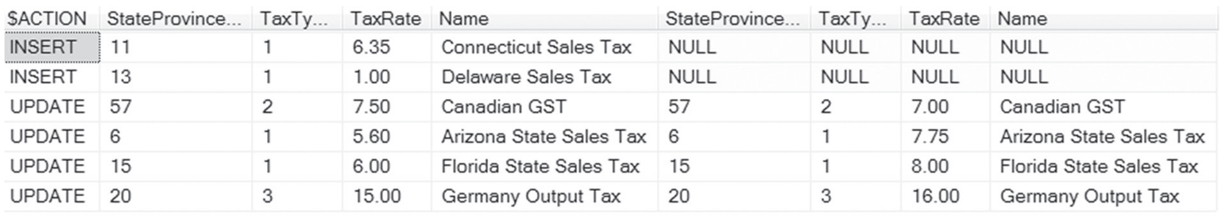
Final thoughts

MERGE provides an elegant solution to a common problem—the need to update some records, add others and delete yet others. But like any complex command, using it correctly requires some practice.

Before closing, I must add that there are some known issues with MERGE. Before committing to using MERGE, you may want to review this article by longtime SQL Server MVP Aaron Bertrand: http://tinyurl.com/mtq2kfx.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. Tamar is author or co-author of a dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with Visual FoxPro *and* Taming Visual FoxPro's SQL. *Her latest collaboration is* VFPX: Open Source Treasure for the VFP Developer, *available at www.foxrockx.com. Her other books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar was a Microsoft Support Most Valuable Professional from the program's inception in 1993 until 2011. She is one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.*



$ACTION	StateProvince...	TaxTy...	TaxRate	Name	StateProvince...	TaxTy...	TaxRate	Name
INSERT	11	1	6.35	Connecticut Sales Tax	NULL	NULL	NULL	NULL
INSERT	13	1	1.00	Delaware Sales Tax	NULL	NULL	NULL	NULL
UPDATE	57	2	7.50	Canadian GST	57	2	7.00	Canadian GST
UPDATE	6	1	5.60	Arizona State Sales Tax	6	1	7.75	Arizona State Sales Tax
UPDATE	15	1	6.00	Florida State Sales Tax	15	1	8.00	Florida State Sales Tax
UPDATE	20	3	15.00	Germany Output Tax	20	3	16.00	Germany Output Tax

Figure 5. The OUTPUT clause in Listing 9 shows that two rows were inserted and four were modified.