**A Hazard of Using the SQL Update Statement**

**By** [**Dwain Camps**](http://www.sqlservercentral.com/Authors/Articles/Dwain_Camps/1444841/)**, 2015/04/06 (first published: 2013/09/09)**

**FROM:** [**http://www.sqlservercentral.com/articles/T-SQL/101464/**](http://www.sqlservercentral.com/articles/T-SQL/101464/)

A few months back, I submitted an article I called [A Hazard of Using the SQL Merge Statement](http://www.sqlservercentral.com/articles/MERGE/97867/).  That article generated a bit of [controversy](http://www.sqlservercentral.com/Forums/FindPost1438203.aspx) at the time, and subsequently generated [more](http://www.sqlservercentral.com/Forums/FindPost1475161.aspx) and [more](http://www.sqlservercentral.com/Forums/FindPost1476254.aspx).  Since, as they say, there’s no such thing as bad press so long as you spell my name right, naturally I was compelled to write a SeQueL.

Today’s topic is the SQL UPDATE statement, so before we can get into what hazards may be afoot, we’ll need to set up some sample data.  Admittedly, this will seem a bit contrived, but hopefully it will ultimately make our point.

CREATE TABLE #Test1 (

ID INT IDENTITY(1,1) PRIMARY KEY

,Value VARCHAR(10)

);

CREATE TABLE #Test2 (

ID INT

,Row\_No INT

,Value VARCHAR(10)

,PRIMARY KEY (ID, Row\_No)

);

INSERT INTO #Test1

SELECT 'Dwain' UNION ALL SELECT 'Brandie';

INSERT INTO #Test2

SELECT 1, 1, 'Tom'

UNION ALL

SELECT 1, 2, 'Chris'

UNION ALL

SELECT 2, 1, 'Jeff'

UNION ALL

SELECT 2, 2, 'Lynn';

SELECT a.ID, b.Row\_No, a.Value, b.Value

FROM #Test1 a

JOIN #Test2 b ON a.ID = b.ID;

The result from our small sample of data appears as follows.

ID   Row\_No   Value    Value

1    1        Dwain    Tom

1    2        Dwain    Chris

2    1        Brandie  Jeff

2    2        Brandie  Lynn

What would happen in this case, should we use the results of this query to update the Value column of our #Test1 table?  Let’s give it a go and find out.

UPDATE a

SET Value = b.Value

FROM #Test1 a

JOIN #Test2 b ON a.ID = b.ID;

SELECT \*

FROM #Test1;

The query runs without issue and the SELECT returns these results.

ID Value

-- -----

1  Tom

2  Jeff

This is interesting. We updated the Value column of our #Test1 table twice for each row. Or did we?

Obviously only one result can end up being stored there. In this case, it happens to be the first value as per the clustered index on ID, Row\_No of the #Test2 table. Since we always hear that SQL does not guarantee row ordering without an explicit ORDER BY clause, we assume that the same case is true here.

Let’s make a slight modification to our UPDATE and try it again.

UPDATE a

SET Value = b.Value

OUTPUT DELETED.\*, INSERTED.\*

FROM #Test1 a

JOIN #Test2 b ON a.ID = b.ID;

SELECT \*

FROM #Test1;

The results from the OUTPUT statement display this:

ID   Value    ID   Value

1    Dwain    1    Tom

2    Brandie  2    Jeff

In reality we find that somehow the SQL Server Query Optimizer was smart enough to decide that it only needed to update each row in our #Test1 table once. How it chose the row to use to make that update will ultimately remain its own dirty little secret.

Herein lies that hazard. The results of a query that returns multiple rows for the target row to be updated, will update that row with unpredictable results.

Often times I’ve seen developers that fail to properly analyze the queries they write, most specifically when it comes to determining whether sufficient criteria has been included to ensure the uniqueness of the rows in the output set. I can’t tell you how many times I’ve seen junior developers throw in a DISTINCT to simply resolve the non-uniqueness, without attempting to understand the root cause.

In this particularly contrived example, there’s no way to ensure the distinctness of the rows you’re applying as the source to the target table for the update.  Understandably, it is unlikely that you’d try to update the target table using such a contrived source table. I ask you to use your imagination to think of a case where the source might be something much more complex and just happens to return a couple of duplicate rows somewhere along the way.

Is it a good thing to have a piece of code like this running in Production, where the results of the update may be unpredictable?

Let’s refactor our update just a little to help in illustrating another approach.

WITH JoinedTables AS (

    SELECT ID=a.ID, Value=a.Value, ID2=b.ID, Value2=b.Value

     FROM #Test1 a

    JOIN #Test2 b ON a.ID = b.ID)

UPDATE JoinedTables

SET Value = Value2;

SELECT \*

FROM #Test1;

Nothing new so far. A quick run of this code shows that the results it yields are identical to our last update (assuming you’ve restored the original data to the tables).  Even an update through a Common Table Expression (CTE) yields the same unpredictable behavior.

The SQL MERGE statement can also be used to do an update, although more commonly it is used to insert and update at the same time.  Let’s look at a piece of code that does the same thing as the update prior.

WITH JoinedTables AS (

    SELECT ID=a.ID, Value=a.Value, ID2=b.ID, Value2=b.Value

    FROM #Test1 a

    JOIN #Test2 b ON a.ID = b.ID)

MERGE #Test1 t

USING JoinedTables s

ON s.ID = t.ID

WHEN MATCHED THEN

UPDATE SET Value = s.Value2;

When we run this code, we get the following error message:

The MERGE statement attempted to UPDATE or DELETE the same row more than once. This happens when a target row matches more than one source row. A MERGE statement cannot UPDATE/DELETE the same row of the target table multiple times. Refine the ON clause to ensure a target row matches at most one source row, or use the GROUP BY clause to group the source rows.

Now that’s something that makes a whole lot of sense. Why couldn’t our UPDATE statement throw a nice clear warning like that?

**Conclusion**

I won’t go so far as to suggest that every time you need to do an UPDATE, you should use a MERGE instead.  I will suggest that perhaps it’s better for your SQL to throw an error (that you could trap in a TRY/CATCH construct), rather than have it unpredictably update rows in a manner you may not be expecting. The latter occurring in a Production environment could manifest itself as a nightmare of troubleshooting.

It should be clear, that anytime you have the potential of a JOIN generating duplicates, you may want to consider what may happen if it does and how you want to guard against that occurrence.

OK.  I’ve said my peace.  Let the controversy begin.

Dwain Camps