Updating Data

T-SQL supports a standard *UPDATE* statement that allows you to update rows in a table. T-SQL also supports nonstandard uses of the *UPDATE* statement with joins and with variables. This section describes the various uses of the *UPDATE* statement.

The examples I provide in this section are against copies of the Orders and OrderDetails tables from the TSQLFundamentals2008 database created in the tempdb database. Run the following code to create and populate those tables:

```
USE tempdb;

IF OBJECT_ID('dbo.OrderDetails', 'U') IS NOT NULL DROP TABLE dbo.OrderDetails;
IF OBJECT_ID('dbo.Orders', 'U') IS NOT NULL DROP TABLE dbo.Orders;

SELECT * INTO dbo.Orders FROM TSQLFundamentals2008.Sales.Orders;
SELECT * INTO dbo.OrderDetails FROM TSQLFundamentals2008.Sales.OrderDetails;

ALTER TABLE dbo.Orders ADD
  CONSTRAINT PK_Orders PRIMARY KEY(orderid);
ALTER TABLE dbo.OrderDetails ADD
  CONSTRAINT PK_OrderDetails PRIMARY KEY(orderid, productid),
  CONSTRAINT FK_OrderDetails_Orders FOREIGN KEY(orderid)
    REFERENCES dbo.Orders(orderid);
```

The UPDATE Statement

The *UPDATE* statement is a standard statement that allows you to update a subset of rows in a table. To identify the subset of rows that are the target of the update, you specify a predicate in a WHERE clause. You specify the assignment of values or expressions to columns in a SET clause, separated by commas.

For example, the following *UPDATE* statement increases the discount of all order details with product 51 by 5 percent:

```
USE tempdb;

UPDATE dbo.OrderDetails
  SET discount = discount + 0.05
WHERE productid = 51;
```

Of course you can run a *SELECT* statement with the same filter before and after the update to see the changes. Later in the chapter, I'll show you another way to see the changes using a clause called OUTPUT that you can add to modification statements.

SQL Server 2008 introduces support for compound assignment operators: += (plus equal), −= (minus equal), *= (multiplication equal), /= (division equal,) and %= (modulo equal), allowing you to shorten assignment expressions such as the one in the preceding query. Instead of the expression discount = discount + 0.05, you can use the shorter form: discount += 0.05. The full *UPDATE* statement looks like this:

```
UPDATE dbo.OrderDetails
  SET discount += 0.05
WHERE productid = 51;
```

All-at-once operations are an important aspect of SQL that you should keep in mind when writing *UPDATE* statements. I explained the concept in Chapter 2, "Single-Table Queries," in the context of *SELECT* statements, but it's just as applicable with *UPDATE* statements. Remember the concept that says that all expressions in the same logical phase are evaluated as if at the same point in time. To understand the relevance of this concept, consider the following *UPDATE* statement:

```
UPDATE dbo.T1
  SET col1 = col1 + 10, col2 = col1 + 10;
```

Suppose that one row in the table has the values 100 in col1 and 200 in col2 prior to the update. Can you determine the values of those columns after the update?

If you do not consider the all-at-once concept, you would think that col1 will be set to 110 and col2 to 120, as if the assignments were performed from left to right. However, the assignments take place as if all at once, meaning that both assignments use the same value of col1—the value before the update. The result of this update is that both col1 and col2 will end up with the value 110.

With the concept of all-at-once in mind, can you figure out how to write an *UPDATE* statement that swaps the values in the columns col1 and col2? In most programming languages where expressions and assignments are evaluated in some order (typically left to right), you need a temporary variable. However, because in SQL all assignments take place as if at the same point in time, the solution is very simple:

```
UPDATE dbo.T1
  SET col1 = col2, col2 = col1;
```

In both assignments the source column values used are those prior to the update, so you don't need a temporary variable.

UPDATE Based on a Join

Similar to the *DELETE* statement, T-SQL also supports a nonstandard syntax for *UPDATE* statements based on joins. As with *DELETE* statements, the join serves a filtering purpose.

The syntax is very similar to a *SELECT* statement based on a join; that is, the FROM and WHERE clauses are the same, but instead of the SELECT clause you specify an UPDATE clause. The *UPDATE* keyword is followed by the alias of the table that is the target of the update (you can't update more than one table in the same statement), followed by the SET clause with the column assignments.

For example, the *UPDATE* statement in Listing 8-1 increases the discount of all order details of orders placed by customer 1 by 5 percent:

Listing 8-1. UPDATE Based on a Join

```
UPDATE OD
  SET discount = discount + 0.05
FROM dbo.OrderDetails AS OD
  JOIN dbo.Orders AS O
    ON OD.orderid = O.orderid
WHERE custid = 1;
```

To "read" or interpret the query, start with the FROM clause, move on to the WHERE clause, and finally go to the UPDATE clause. The query joins the OrderDetails table (aliased as OD) with the Orders table (aliased as O) based on a match between the order detail's order ID and the order's order ID. The query then filters only the rows where the order's customer ID is 1. The query then specifies in the UPDATE clause that OD (the alias of the OrderDetails table) is the target of the update, and increases the discount by 5 percent.

If you want to achieve the same task using standard code you would need to use a subquery instead of a join, like so:

```
UPDATE dbo.OrderDetails
  SET discount = discount + 0.05
WHERE EXISTS
  (SELECT * FROM dbo.Orders AS O
   WHERE O.orderid = OrderDetails.orderid
     AND custid = 1);
```

The query's WHERE clause filters only order details in which a related order is placed by customer 1. With this particular task, SQL Server will most likely interpret both versions the same way; therefore, you shouldn't expect performance differences between the two. Again, the version you feel more comfortable with probably depends on whether you feel more comfortable with joins or subqueries. But as I mentioned earlier in regards to the *DELETE* statement, I recommend sticking to standard code unless you have a compelling reason to do otherwise. With our current task, I do not see a compelling reason.

However, in some cases, the join version will have a performance advantage over the subquery version. In addition to filtering, the join also gives you access to attributes from other tables that you can use in the column assignments in the SET clause. The same access to the other table can serve both the filtering purpose and obtaining attribute values from the other table for the assignments. However, with the subquery approach, each subquery involves separate access to the other table—that's at least the way subqueries are processed today by SQL Server's engine.

For example, consider the following nonstandard *UPDATE* statement based on a join:

```
UPDATE T1
  SET col1 = T2.col1,
      col2 = T2.col2,
      col3 = T2.col3
FROM dbo.T1 JOIN dbo.T2
  ON T2.keycol = T1.keycol
WHERE T2.col4 = 'ABC';
```

This statement joins the tables T1 and T2 based on a match between T1.keycol and T2.keycol. The WHERE clause filters only rows where T2.col4 is equal to 'ABC'. The *UPDATE* statement marks the T1 table as the target for the UPDATE, and the SET clause sets the values of the columns col1, col2, and col3 in T1 to the values of the corresponding columns from T2.

An attempt to express this task using standard code with subqueries yields the following lengthy query:

```
UPDATE dbo.T1
  SET col1 = (SELECT col1
              FROM dbo.T2
              WHERE T2.keycol = T1.keycol),

      col2 = (SELECT col2
              FROM dbo.T2
              WHERE T2.keycol = T1.keycol),

      col3 = (SELECT col3
              FROM dbo.T2
              WHERE T2.keycol = T1.keycol)
WHERE EXISTS
  (SELECT *
   FROM dbo.T2
   WHERE T2.keycol = T1.keycol
     AND T2.col4 = 'ABC');
```

Not only is this version convoluted (unlike the join version), but each subquery also involves separate access to table T2. So this version is less efficient than the join version.

ANSI SQL has support for row constructors (also known as vector expressions) that were only implemented partially in SQL Server 2008 as I explained earlier. Still, many aspects of row constructors have not yet been implemented in SQL Server, including the ability to use those in the SET clause of an *UPDATE* statement like so:

```
UPDATE dbo.T1

  SET (col1, col2, col3) =

      (SELECT col1, col2, col3
       FROM dbo.T2
       WHERE T2.keycol = T1.keycol)

WHERE EXISTS
  (SELECT *
   FROM dbo.T2
   WHERE T2.keycol = T1.keycol
     AND T2.col4 = 'ABC');
```

But as you can see, this version would still be more complicated than the join version because it requires separate subqueries for the filtering part and for obtaining the attributes from the other table for the assignments.

Assignment UPDATE

T-SQL supports a proprietary UPDATE syntax that both updates data in a table and assigns values to variables at the same time. This syntax saves you the need to use separate *UPDATE* and *SELECT* statements to achieve the same task.

One of the common cases where you can use this syntax is in maintaining a custom sequence/auto-numbering mechanism when, for whatever reason, the identity column property doesn't work for you. The idea is to keep the last used value in a table, and to use this special UPDATE syntax to increment the value in the table and assign the new value to a variable.

Run the following code to first create the Sequence table with the column val, and then populate it with a single row with the value 0—one less than the first value that you want to use:

```
USE tempdb;
IF OBJECT_ID('dbo.Sequence', 'U') IS NOT NULL DROP TABLE dbo.Sequence;
CREATE TABLE dbo.Sequence(val INT NOT NULL);
INSERT INTO dbo.Sequence VALUES(0);
```

Now, whenever you need to obtain a new sequence value use the following code:

```
DECLARE @nextval AS INT;
UPDATE Sequence SET @nextval = val = val + 1;
SELECT @nextval;
```

The code declares a local variable called *@nextval*. Then it uses the special UPDATE syntax to increment the column value by 1, assigns the updated column value to the variable, and presents the value in the variable. The assignments in the SET clause take place from right to left. That is, first val is set to val + 1, then the result (val + 1) is set to the variable *@nextval*.

The specialized UPDATE syntax is run as an atomic operation, and it is more efficient than using separate *UPDATE* and *SELECT* statements because it accesses the data only once.