# Using Triggers

Russell Sinclair

2000 2002 DOWNLOAD

Last month, Russell Sinclair explained the anatomy of triggers and how they work. In this month's installment of "Working T-SQL," he shows you how to write triggers and how to put them to work in your database application.

LAST month, I explained what triggers are, how they work, and what functionality they provide. I didn't provide any code examples for how to use them. As I explained then, this is because the next version of SQL Server (currently codenamed "Yukon") is going to be programmable in any .NET-compliant language. You won't be limited to Transact-SQL for any code you need to write for your database.

That being said, Yukon is still a long way away from release. In fact, it hasn't even been released to public beta yet. Since it's unlikely that you want to wait for a year or two until Yukon is released, this month I'll take you through the syntax of writing triggers, and explain some of the more common problems you'll run into when you write them. I'll also show you how and when to use them to replace VBA code.

## Trigger syntax

The basic syntax for creating a trigger is:

```
CREATE TRIGGER trigger_name ON {table|view}
{FOR|AFTER|INSTEAD OF}
    {[INSERT] [,] [UPDATE] [,] [DELETE]}
AS
    sql_statement
```

The basic syntax requires that you define a name for the trigger, to which table or view it applies, what type of trigger it is, and what data modification statements will raise it. Once all of this is defined, you can define the behavior of the trigger in any number of SQL statements.

The SQL statements you define can use any number of data modification or manipulation calls and can use the UPDATE(column_name) function as necessary. You can make use of the inserted and deleted tables, and you can cancel the action that raised the trigger by calling ROLLBACK TRAN in your code. For a complete description of these features, see last month's "Working T-SQL" column (also available on the *Smart Access* Web site at www.vb123.com/kb).

In order to create or edit a trigger in Access, right-click a table and select the Triggers... menu item on the context menu. This will open the Triggers dialog shown in Figure 1.

You can use this dialog to create, edit, and delete triggers as necessary. When you choose to create a new trigger or edit an existing one, you'll be taken to a SQL editor window that will allow you to modify the trigger's definition.

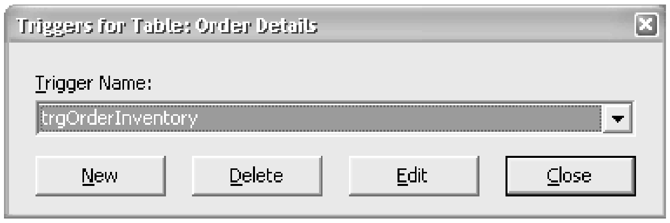
One of the most common examples used with triggers is inventory management. When a user places an order (that is, a new record is added to the Orders table), you want to automatically update inventory values (in other words, update the Inventory table by removing the ordered items from stock). This situation is the ideal situation in which to use an AFTER trigger. The Northwind database uses a Products table to keep track of inventory levels. This isn't the best example, but I'll use it because it's the one database that I know that you have a copy of. Two fields in the Products table need to be managed—UnitsInStock and UnitsOnOrder. When a user adds a new Order Detail, we want these values to be updated. A trigger that handles this situation would look like this:

```
CREATE TRIGGER trgOrderInventoryInsert
ON dbo.[Order Details]
AFTER INSERT
AS
    SET NOCOUNT ON

    UPDATE P SET
      UnitsInStock = UnitsInStock -
      (SELECT SUM(Quantity) FROM inserted AS I
       WHERE I.ProductID = P.ProductID
       GROUP BY I.ProductID),
      UnitsOnOrder = UnitsOnOrder +
      (SELECT SUM(Quantity) FROM inserted AS I
       WHERE I.ProductID = P.ProductID
       GROUP BY I.ProductID)
    FROM Products AS P INNER JOIN inserted AS I
        ON P.ProductID = I.ProductID

    SET NOCOUNT OFF
```

This trigger is defined as trgOrderInventory on the Order Details table, and handles INSERT statements only. The math it performs is a simple case of adding or subtracting the new ordered value from the Product fields. The new values are retrieved by summing the


Triggers for Table: Order Details

Trigger Name: trgOrderInventory

Figure 1. Access 2002 Triggers dialog.

quantity ordered in the “inserted” table. A more efficient way to write this trigger would be to use an embedded query, like this:

```
CREATE TRIGGER trgOrderInventoryInsert
ON dbo.[Order Details]
AFTER INSERT
AS
    SET NOCOUNT ON

    UPDATE P SET
      UnitsInStock =
        UnitsInStock - I.IQuantity,
      UnitsOnOrder =
        UnitsOnOrder + I.IQuantity
    FROM Products AS P
        INNER JOIN
            (SELECT ProductID,
             SUM(Quantity) As IQuantity
             FROM inserted
             GROUP BY ProductID) AS I
          ON P.ProductID = I.ProductID

    SET NOCOUNT OFF
```

This trigger has a call to one of SQL Server’s SET statements: SET NOCOUNT. The SET NOCOUNT statement tells SQL Server not to tell any calling library that rows were modified between the lines where the feature is turned on then off again. ADO, the library through which Access connects to SQL Server, checks statements as they’re executed to determine how many rows they affected. Without the call to SET NOCOUNT, ADO will generate a message for the rows updated by the trigger, which are outside the scope of the original update. This will cause Access to generate an error. Wrapping SQL statements that select or modify data in a trigger in SET NOCOUNT calls will prevent this error from being raised.

I’ve handled inserts into the table, but what about when a user *deletes* an Order Detail from the table? The solution is surprisingly simple in this case. The only differences are that I use the deleted table instead of the inserted table, and subtract where I formerly added (and vice versa):

```
CREATE TRIGGER trgOrderInventoryDelete
ON dbo.[Order Details]
AFTER DELETE
AS
    SET NOCOUNT ON

    UPDATE P SET
      UnitsInStock =
        UnitsInStock + D.DQuantity,
      UnitsOnOrder =
        UnitsOnOrder - D.DQuantity
    FROM Products AS P
        INNER JOIN
            (SELECT ProductID,
             SUM(Quantity) As DQuantity
             FROM deleted
             GROUP BY ProductID) AS D
          ON P.ProductID = D.ProductID

    SET NOCOUNT OFF
```

I still have one situation that hasn’t been handled: What happens when a user updates a row and changes the quantity ordered? In order to handle this situation, you could create another trigger that handles updates only. However, I prefer to create a single trigger for all actions in cases where the code has the same purpose. Rewritten to handle all data modification statements, my final version of the trigger would look like this:

```
CREATE TRIGGER trgOrderInventory
ON dbo.[Order Details]
AFTER INSERT, DELETE, UPDATE
AS
    SET NOCOUNT ON

    UPDATE P SET
      UnitsInStock = UnitsInStock
        -  (ISNULL(I.IQuantity, 0)
         - ISNULL(D.DQuantity, 0)),
      UnitsOnOrder = UnitsOnOrder
        +  (ISNULL(I.IQuantity, 0)
         - ISNULL(D.DQuantity, 0))
    FROM Products AS P
        LEFT JOIN
            (SELECT ProductID,
             SUM(Quantity) As IQuantity
             FROM inserted
             GROUP BY ProductID) AS I
          ON P.ProductID = I.ProductID
        LEFT JOIN
            (SELECT ProductID,
             SUM(Quantity) As DQuantity
             FROM deleted
             GROUP BY ProductID) AS D
          ON P.ProductID = D.ProductID

    SET NOCOUNT OFF
```

This case seems much more complicated, but it really isn’t—it’s just a combination of the INSERT and DELETE triggers that I created previously. All I’ve done in this code is subtract the sum of the old quantities (available in the deleted table) from the new quantities (available from the inserted table), and then I used that value to determine what has to be added to or subtracted from the inventory levels.

You may have noticed that I also used a SQL Server function that you might not have seen before—ISNULL. This function is roughly equivalent to the Nz function in VBA. It takes a value to check as the first argument and (as the second argument) the value to return if the first value is Null. Since there’s a chance that there might not be matching records in the inserted and deleted tables (this can happen if an INSERT or DELETE took place), I needed to handle Null values properly by converting them to zeros.

You may also have noticed that, unlike the first two examples, I’ve made LEFT joins from the Products table to the two trigger tables. Again, this allowed me to handle cases in which there might not be matching values in one of the tables.

## Validation triggers

Triggers aren’t always designed to update related data. Sometimes they can be used to validate the data that a user enters. Although you can use constraints to validate data, constraints can only validate a record in relation to itself. You can’t check values in other tables to validate the entry that the user made. This is where you can use an AFTER trigger to solve a business problem.

As I mentioned earlier, a trigger has the ability to reject changes made by a SQL command by calling the ROLLBACK TRAN statement. When you do this, however, the users don't get any indication that the update failed. It may even appear to them that the data has been successfully updated. Because of this, you should raise an error along with the transaction rollback. This is achieved in SQL Server by calling the RAISERROR statement (please see the "RAISERROR" sidebar for further discussion).

The following trigger illustrates the use of rollbacks and raising errors. This trigger is an extension of the previous trigger in that it validates the inventory values before the order is allowed to be placed. If the UnitsInStock value is less than what's being ordered, an error will be raised.

```
CREATE TRIGGER trgOrderDetailsValidate
ON dbo.[Order Details]
AFTER INSERT
AS
    IF EXISTS(SELECT *
              FROM Products AS P
                INNER JOIN
                   (SELECT ProductID,
                    SUM(Quantity) As IQuantity
                    FROM inserted
                    GROUP BY ProductID) AS I
                ON P.ProductID = I.ProductID
              WHERE I.IQuantity > P.UnitsInStock)
        BEGIN
            ROLLBACK TRAN
            RAISERROR(50009, 16, 1)
        END
```

## RAISERROR

Raising errors in SQL Server is very similar to raising errors in VBA. However, instead of calling Err.Raise, you must call RAISERROR. RAISERROR allows you to raise errors predefined in SQL Server (either system or user-defined errors) and can cause those errors to be passed back to the client. The basic syntax for RAISERROR is

```
RAISERROR (msgid|msgtext, severity, state)
```

The first parameter is either the number for a predefined error, or your own error message text.

SQL Server defines a number of severities—set in the second parameter—that are described in more detail in the Books Online. If you want your error to be picked up by Access, the severity must be 10 or greater. Standard practice is to use the number 16, as it's a level sufficient to cause Access to see the error but is low enough that any user can use it without conflicting with higher error types that are reserved for specific purposes.

The final argument, the state, is an arbitrary integer between 1 and 127 that you can use to provide your application with some information that's meaningful to you. The value can be accessed through the SQLState property of the ADO Error object generated after the error is raised. If you're not sure what to use for state, stick with 1.

This trigger handles INSERT statements and also checks to see whether the total quantity of any item entered exceeds the number in stock. If any such records exist, the transaction is rolled back and an error is raised. Although this trigger is a separate object than the one that I programmed before, the two triggers work in tandem. If any trigger on a table rolls back a transaction, the actions of all triggers that are reacting to that action are rolled back. This means that the previous trigger (the one that updated the UnitsInStock in the Products table) will also be rolled back by this trigger. The ROLLBACK forces the database back to the state it was in before the trigger was fired.

This trigger deals only with inserts into the table. But how would you handle updates? Handling updates would be done in much the same way as this. However, instead of checking the values in the inserted table, you'd have to compare the values to the values in the deleted table, as in the previous examples. If you're interested in the solution to this particular question, feel free to e-mail me. I'd be happy to provide you with the answer. However, this trigger isn't really necessary in the Northwind database. If the inventory trigger drops the UnitsInStock or UnitsOnOrder values below zero, constraints on the Products table defined against these columns will raise an error and prevent the update. This is a case where constraints can actually be used to simplify triggers.

## INSTEAD OF triggers

INSTEAD OF triggers come into play during an admittedly unusual situation, one that usually involves updates to a view. In order to understand when you might use INSTEAD OF triggers, you first need to understand something about views.

Views are stored queries that provide data in a format that's usually different, in some way, from the data in the system's table. Views might summarize data or otherwise manipulate data that's displayed to a user. SQL Server provides information to ADO data consumers (Access) about the underlying tables that comprise the view. This can include what tables and fields make up the view, and whether or not columns can be updated. As a result, when you update data in a view, ADO knows where the underlying tables and fields that make up the view are stored and uses the information about the underlying tables when you update the view. ADO tells SQL Server to make changes to the underlying tables at the time of the update, instead of sending the data back to the view itself.

For example, if you create a view called vwOrders that references the Orders table, and you update the CustomerID field in this view, ADO will, by default, send a statement to SQL Server to update the underlying table:

```
UPDATE Orders SET CustomerID = 'ALFKI'
WHERE Orders.OrderID = 123
```

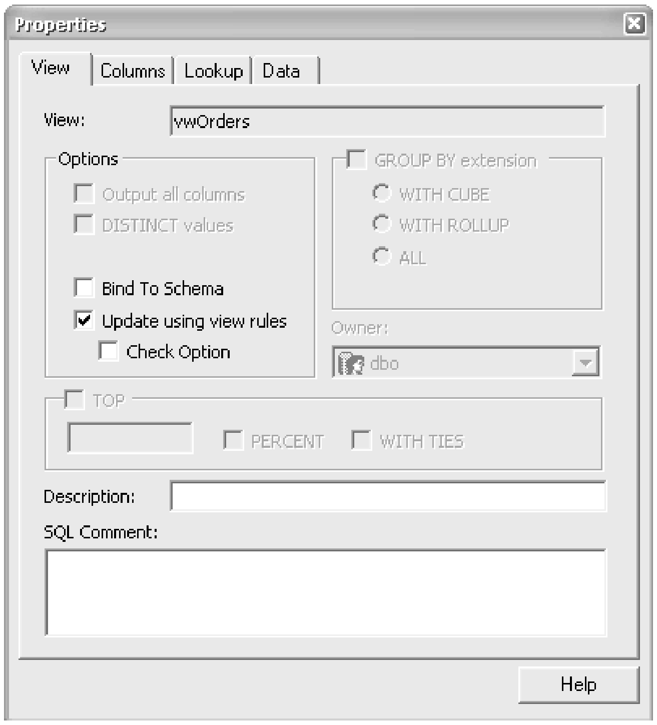
You can see that the underlying Orders table itself is referenced in the update, not the view. If you have any calculated or aggregate columns in your view, ADO and Access will prevent you from updating these fields because SQL Server will correctly tell ADO that they aren't updateable. However, you can bypass this behavior by telling SQL Server to directly update the view itself. This is done in Access by opening a view in design mode, choosing View | Properties from the menu, and checking the "Update using view rules" check box, as shown in Figure 2.

With the "Update using view rules" option checked, updates to data in the view are directed to the view itself, and not to its underlying tables. (For those of you familiar with creating views using T-SQL, checking this option is the equivalent of setting the WITH VIEW_METADATA option on a view—see the Books Online for more information.) The update to the CustomerID that I made before would now be sent to SQL Server as:

```
UPDATE vwOrders SET CustomerID = 'ALFKI'
WHERE vwOrders.OrderID = 123
```

Now that you understand this feature of views, you can make use of INSTEAD OF triggers. INSTEAD OF triggers are most commonly created on views. In fact, AFTER triggers can't be used on views at all.

Consider this situation: You need to create a view that shows all the Orders in the database and the effective discount for the entire order based on all items in the order (the actual discount is stored on a line-by-line basis). Now, wouldn't it be nice if you could not only



View: vwOrders

Options
- [ ] Output all columns
- [ ] DISTINCT values
- [ ] Bind To Schema
- [✓] Update using view rules
- [ ] Check Option

- [ ] GROUP BY extension
  - WITH CUBE
  - WITH ROLLUP
  - ALL

Owner: dbo

- [ ] TOP
- [ ] PERCENT
- [ ] WITH TIES

Description:

SQL Comment:

Figure 2. View options.

present that value to the users, but also provide them with a method of forcing a single discount for the entire order? With an updateable view and an INSTEAD OF trigger, this can easily be achieved.

One small problem—you can't create triggers on views in Access. Someone along the way decided that we Access programmers didn't deserve to use this feature. Such is life. In order to create this view, you'll need to use a tool like Query Analyzer that can run SQL statements directly against a database.

There are also a few tricks that need to be used in order to make the view updates behave the way that you want them to. The first thing you need to do is create a new text scalar function. This function takes the OrderID and returns the effective discount for the order. The definition of this function is in the file udfEffectiveDiscount.sql in this month's Download (as are other triggers that I've mentioned in this article).

You then need to create a view that uses the Orders table and a calculated field based on the udfEffectiveDiscount function (see vwOrderDiscount.sql). You also need to check the "Update using view rules" option in the view properties. Your view is now almost ready for prime time. The one thing left to do is to define the trigger itself.

You can see the full definition of this trigger in trgOrdersDiscount.sql, but here's the code for the portion of the trigger that handles changes to the EffectiveDiscount column in the view:

```
CREATE  TRIGGER trgOrdersDiscount
ON dbo.vwOrderDiscount
INSTEAD OF INSERT, UPDATE, DELETE
AS
    SET NOCOUNT ON

    /*   **CODE NOT SHOWN** INSERT,
         DELETE, UPDATE passed onto
         base table              */

    --Now update the child records
    UPDATE OD SET Discount = I.EffectiveDiscount
    FROM [Order Details] AS OD
        INNER JOIN inserted AS I
        ON OD.OrderID = I.OrderID
        INNER JOIN deleted AS D
        ON OD.OrderID = D.OrderID
         WHERE D.EffectiveDiscount <> I.EffectiveDiscount


    SET NOCOUNT OFF
```

This code changes the discount values of all child records to match the value of the EffectiveDiscount field in the inserted table. Note that this field is a calculated field and it never physically existed. It was created only for use in the view.

The code that I've cut out of this listing handles the main portion of the deletes, updates, and inserts and is relatively simple to create. All you have to do is explicitly pass on the changes that were made by the user, mapping fields in the view to fields in the base tables where appropriate. You should keep in mind that, in this type of trigger, if you add fields to the base table or view later on, you'll need to add these fields to your statements that pass on the changes in your trigger. If you don't do that, you may frustrate yourself and others trying to figure out why changes aren't being committed to new fields.

As I said earlier, in order to add this trigger to your view, you'll need to use a query tool like Query Analyzer that can bypass Access's shortcomings. If you don't have access to any such tool, you can use code to create this trigger. All you have to do is load the text file into a string variable and execute the string against the database connection, like this:

```
CurrentProject.Connection.Execute strFileContents
```

To be honest, I rarely use INSTEAD OF triggers. If I do, it's usually in a Visual Basic application where I need to fool ADO into thinking that the view is updateable. Even then, I usually don't pass the data back to the server through the view—I let code handle the updates. But knowing how to write this kind of trigger can come in useful when you run into a development problem with your data.

## When to use triggers

Now that I've told you most of what you need to know to get started using triggers, you might see a lot of places where you can use them to replace VBA code. Keep a few things in mind when doing so:

- The best code to replace with triggers can usually be found in form BeforeUpdate and AfterUpdate events. The code in these two events is often ripe for a move to triggers.
- Also keep in mind that triggers should only be used for code that's directly related to the structure of the database. Don't move code that defines some business logic that has nothing to do with the data. You could overload your server with unnecessary code.
- Finally, don't overuse triggers. Triggers offer some very strong features for data validation—but so do constraints. If your validation only needs to handle a single record at a time, you can most likely use a constraint instead of a trigger, as was the case in the inventory validation trigger I showed you earlier. ▲

DOWNLOAD ADPTRIG2.ZIP at www.vb123.com/kb

Russell Sinclair is an MSCD and is the owner of Synthesystems, a technology consulting firm specializing in Visual Basic, SQL Server, and Microsoft Access development. He's the author of *From Access to SQL Server*, an Access developer's guide to migrating to SQL Server; a senior programmer with Questica, Inc., a company specializing in software for custom-design manufacturers; and a *Smart Access* ContributingEditor

HALF PAGE AD:
APPDEV