Triggers

Objective
After completing this lab, you will be able to:
- Create triggers to maintain data integrity.
- Create triggers to maintain business rules.
- Use conditional logic and operations on local variables in Transact-SQL.

Required Materials
- Northwind database.
- SQL Server Management Studio

Related Reading
- SQL Server Management Studio Help, topics:
  - "Triggers" and subtopics.
  - local variables: specifically, DECLARE to declare local variables and usage of SET and SELECT to assign values to local variables.
  - IF..ELSE blocks.

Assignment Details
You will only have one deliverable for this lab: a SQL script containing a trigger to maintain a business rule. We first introduce the problem. The numbered steps below will walk you through how to write triggers so that you can solve the problem by the end of the lab.

The trigger you will write will be a trigger in your copy of the **Northwind** database to reflect the notion of limited resources. Specifically, we want the database to handle the following behavior on its own:

> When an order is placed for *X* units of product *Y*, we must first check the **Products** table to ensure that there is sufficient stock to fill the order. This trigger will operate on the **Order Details** table.
>
> If sufficient stock exists, then fill the order and decrement *X* units from the **UnitsInStock** column in **Products**.
>
> If insufficient stock exists, then refuse the order (*i.e.* do not insert it) and notify the user that the order could not be filled because of insufficient stock.

To simplify matters, we will assume that this trigger will have to operate only on single-row inserts. (Triggers that operate on multiple deletes, updates, or inserts are possible, but they add an unnecessary layer of complexity for this lab.)

In an earlier lab, you implemented a stored procedure for inserting order details that provided similar functionality to the trigger that you'll implement in this lab. It is useful to know both techniques, because not every DBMS supports both triggers and stored procedures.

(1) First, we will become acquainted with the nature of triggers. Triggers use two logical tables, **inserted** and **deleted**, which are local to the trigger being executed. They contain rows that are to be inserted into or deleted from the table being operated on by the trigger, and have the same structure as the table being operated on by the trigger. They can be accessed by any valid SELECT statement inside the body of the trigger. For example: `SELECT COUNT(*) FROM inserted` will return the number of rows that are to be inserted into the table for which the trigger was defined.

To experiment with this behavior, create a basic trigger on **Order Details** that executes **after** insertion. Your trigger should **select** all rows and all columns in the logical table named "**inserted**".

The **CREATE TRIGGER** statement starts out with the format:

```
CREATE TRIGGER trigger_name
ON [table_name] {AFTER | INSTEAD OF | FOR}
{INSERT | UPDATE | DELETE}
AS …
```

To test your new trigger, write the SQL statement

```
INSERT INTO [Order Details] (OrderID, ProductID, UnitPrice,
Quantity, Discount) VALUES (11077, 22, 21.00, 5, 0.0)
```

If this trigger is working properly, you will receive the result of the SELECT query, which should look something like this:

OrderID	*ProductID*	*UnitPrice*	*Quantity*	*Discount*
11077	*22*	*21.00*	*5*	*0*

We will be using this **INSERT** statement again, so you will need to delete the row you just added out of the Order Details table:

```
DELETE FROM [Order Details] WHERE OrderID=11077 AND ProductID=22
```

(2) In this step, we will alter the trigger we created in step (1) to add the necessary functionality to automatically decrement the **UnitsInStock** column in **Products.**

To do this, it may be helpful to use local variables to store temporary values for decrementing. There are three primary constructs used for working with local variables: **DECLARE**, **SELECT**, and **SET**.

DECLARE simply declares variables as a certain type. For example:

```
DECLARE @units_remaining AS INTEGER
DECLARE @q AS INTEGER, @id AS INTEGER
```

(Variable names must always be prefixed with the @ symbol.)

To assign a value to this variable, we use the **SELECT** command. Here are two ways to assign values to local variables using SELECT.

You can use SELECT like a SET command:

```
SELECT @units_remaining = @units_remaining – 40
```

You can also use SELECT in conjunction with SQL statements:

```
SELECT @q=Quantity,@id=ProductID FROM inserted
```

Alter the trigger created in step (1) to: read the number of items, from **Products**, present for the product being ordered; decrement the amount requested by the order; and, finally, update the **Products** table to reflect the new number of units in stock.

To test your trigger, first check the **UnitsInStock** value for product ID 22 in the **Products** table, and make a note of this value. Then, issue the **INSERT** command given in step (1), and then, in **Products**, check the **UnitsInStock** value corresponding to ProductID 22. It should be 5 units lower than its original value. Remove the newly inserted entry in **Order Details** and revert the value of **UnitsInStock** in table **Products** for ProductID 22 to its original value.

(3) In this step, you will finish the trigger by implementing the conditional logic that either refuses or accepts the order based on the number of units in stock for the product being ordered.

First, the trigger must be set to execute in place of an **INSERT** statement being done on the trigger's table. To do this, the trigger must be set to execute *instead of* the **INSERT** command, rather than *after* the **INSERT** command. This can be done by starting an **ALTER TRIGGER** or **CREATE TRIGGER** command like

```
ALTER TRIGGER / CREATE TRIGGER trigger_name
ON [table_name] INSTEAD OF INSERT
AS
```

Note that issuing an **INSERT** on the trigger's table inside an **INSTEAD OF** trigger will not cause the trigger to fire again. Thus, the following code will insert values into TableA if the given condition is met:

```
CREATE TRIGGER trig1 ON TableA INSTEAD OF INSERT
AS
DECLARE @some_flag int
SET @some_flag = (RAND() * 10)
IF @some_flag = 4
BEGIN
INSERT INTO TableA SELECT * FROM inserted
END
ELSE
BEGIN
PRINT 'Oops!'
END
```

Alter the trigger to print out the message `Insufficient quantity to carry out this order` (or something conveying the same idea) if the number of units in stock for a requested product is less than the number of units requested, and to fill the order in the **Order Details** table and decrement the number of units in stock in **Products** otherwise. You may use the script `Triggers-checktrig.txt` (located in the ANGEL folder for this lab) to check for the proper operation of your trigger.

(4) Complete the ANGEL feedback survey for this lab.

Turn-in Instructions

Create a zip or rar archive containing the SQL script you created for your trigger. Submit your archive to the ANGEL dropbox for the lab.

Revision History

Jan. 19, 2007 Additional example code, Curt Clifton.
Jan. 20, 2006: Minor clarifications, Curt Clifton.
Jan 9, 2005: Correction to CREATE TRIGGER statement.
Jan 9, 2005: Created by David Yip.