An Introduction to Triggers -- Part I

By Garth Wells on **30 April 2001** | 12 Comments | Tags: Triggers

Article Series Navigation: An Introduction to Triggers -- Part I ▼

This article, submitted by Garth , covers the basics of using triggers. "*A trigger is a database object that is attached to a table. In many aspects it is similar to a stored procedure.*" If you're a developer and not familiar with triggers this article is a great starting point.

A trigger is a database object that is *attached* to a table. In many aspects it is similar to a stored procedure. As a matter of fact, triggers are often referred to as a "special kind of stored procedure." The main difference between a trigger and a stored procedure is that the former is attached to a table and is only *fired* when an INSERT, UPDATE or DELETE occurs. You specify the modification action(s) that fire the trigger when it is created.

The following shows how to create a trigger that displays the current system time when a row is inserted into the table to which it is attached.

```
SET NOCOUNT ON

CREATE TABLE Source (Sou_ID int IDENTITY, Sou_Desc varchar(10))
go
CREATE TRIGGER tr_Source_INSERT
ON Source
FOR INSERT
AS
PRINT GETDATE()
go
INSERT Source (Sou_Desc) VALUES ('Test 1')

-- Results --

Apr 28 2001  9:56AM
```

This example is shown for illustrative purposes only. I'll cover an example later in the article that shows a real-world world use of triggers.

When to Use Triggers

There are more than a handful of developers who are not real clear when triggers should be used. I only use them when I need to perform a certain action as a result of an INSERT, UPDATE or DELETE and ad hoc SQL (aka SQL Passthrough) is used. I implement most of my data manipulation code via stored procedures and when you do this the trigger functionality can be moved into the procedure. For example, let's say you want to send an email to the Sales Manager when an order is entered whose priority is high. When ad hoc SQL is used to insert the Orders row, a trigger is used to determine the OrderPriority and send the email when the criteria is met. The following shows a partial code listing of what this looks like.

```
CREATE TABLE Orders (Ord_ID int IDENTITY, Ord_Priority varchar(10))
go
CREATE TRIGGER tr_Orders_INSERT
ON Orders
FOR INSERT
AS
IF (SELECT COUNT(*) FROM inserted WHERE Ord_Priority = 'High') = 1
 BEGIN
  PRINT 'Email Code Goes Here'
 END
go
INSERT Orders (Ord_Priority) VALUES ('High')
```

Resources

SQL Server Resources

Advertise on SQLTeam.com

SQL Server Books

SQLTeam.com Newsletter

Contact Us

About the Site

```
-- Results --

Email Code Goes Here
```

When the stored procedure approach is used you can move the trigger code into the procedure and it looks like this.

```
CREATE PROCEDURE ps_Orders_INSERT
@Ord_Priority varchar(10)
AS
BEGIN TRANSACTION
 INSERT Orders (Ord_Priority) VALUES (@Ord_Priority)

 IF @@ERROR <> 0
  GOTO ErrorCode

 IF @Ord_Priority = 'High'
  PRINT 'Email Code Goes Here'

COMMIT TRANSACTION

ErrorCode:
 IF @@TRANCOUNT <> 0
  PRINT 'Error Code'
go
```

Let's take a look at the trigger example. The first thing you probably noticed is that the SELECT references a table called *inserted*. Triggers make use of two special tables called **inserted** and **deleted**. The inserted table contains the data referenced in an INSERT before it is actually committed to the database. The deleted table contains the data in the underlying table referenced in a DELETE before it is actually removed from the database. When an UPDATE is issued both tables are used. More specifically, the *new* data referenced in the UPDATE statement is contained in inserted and the data that is being updated is contained in deleted.

The example makes an assumption about how data is going to be added to the table. The IF statement is looking for a count of 1. This means the trigger assumes only one row will be added to the table at a time. If more than one row is added to the table in a single statement you may miss an order with a High priority because the trigger only fires once for each associated statement. I realize this may sound a little confusing so let's take a look at two more examples. The following shows that the trigger fires for each INSERT statement executed.

```
INSERT Orders (Ord_Priority) VALUES ('High')
INSERT Orders (Ord_Priority) VALUES ('High')

-- Results --

Email Code Goes Here

Email Code Goes Here
```

Now we have three rows in Orders whose Ord_Priority is High. Let's insert new rows based on the current contents of Orders to show how a trigger behaves when a multi-row statement is executed.

```
INSERT Orders
SELECT Ord_Priority FROM Orders
```

The 'Email Code Here' message is not displayed even though three new rows were added with a priority of High because the IF statement criteria was not satisfied. A trigger fires only once per statement, so the actual COUNT(*) associated with the INSERT is 3. The following shows how to modify the code to handle a multi-row INSERT.

```
ALTER TRIGGER tr_Orders_INSERT
ON Orders
FOR INSERT
AS
IF EXISTS (SELECT * FROM inserted WHERE Ord_Priority = 'High')
 BEGIN
  DECLARE @Count tinyint
  SET @Count = (SELECT COUNT(*) FROM inserted WHERE Ord_Priority = 'High')
  PRINT CAST(@Count as varchar(3))+' row(s) with a priority of High were entered'
 END
go
```

We can test the code using the same INSERT with a SELECT as follows.

```
INSERT Orders
SELECT Ord_Priority FROM Orders

-- Results --

12 row(s) with a priority of High were entered
```

A Real-World Example

Those of you familiar with web site management know that counting the traffic on a site is key in determining which areas of the site are being used. Internet Information Server (IIS) has logging capabilities that tracks a number of attributes associated with each visitor. For example, every time a visitor accesses a page on a site that page and the user's information is logged. By default the data is logged in a text file, but you can alter the default behavior and log the data to an ODBC-compliant data store.

I used this approach for a client a while back because they wanted a simple way to track the activity for each major area of their site. A major area was defined as the sections listed on the site's main navigation bar (e.g., Home, About Us, Services, ...). The goal was to produce a report that showed the number of visits to each of the main areas of the site on a per month basis. A few of you may be wondering why a trigger is needed to implement this solution. After all, a SELECT with a WHERE clause to filter the date range and GROUP BY to count the instances per page will do the trick and no triggers are needed.

The reason I decided to use a trigger-based solution had to do with the unacceptable execution time of the report. Even on a low-traffic site the number of rows in the logging table grows at a staggering rate. For every page accessed by a visitor, there is at least one row added to the table. When a page contains a reference to a graphic (e.g., .gifs or .jpgs), there is another row created. If a page contains five references to graphics, there are six rows created in the logging table every time it is accessed.

The bottom-line is that because of the size of the table the report took too long to execute. In order to reduce the time it took to execute the report I decided to use a summary (aka aggregate) table to count the page views as they were entered into the logging table. Since there were only eight main areas on the site, the summary table contained eight rows and the report ran in less than one second.

The following uses a dummied-down schema to show how this technique works. For the sake of brevity, I will only use two main areas of the site.

```
CREATE TABLE InetLog (ClientHost varchar(255), LogTime datetime, Target
varchar(255))
go
CREATE TABLE LogSummary (LogSum_Category varchar(30), LogSum_Count int)
go
INSERT LogSummary VALUES ('About Us',0)
INSERT LogSummary VALUES ('Services',0)
```

InetLog is the main logging table and LogSummary is the summary table. The two main areas of the site are About Us and Services. The goal of the trigger is to update the value in LogSum_Count every time the AboutUs.htm and Services.htm pages are accessed. The trigger used to do this is shown here.

```
CREATE TRIGGER tr_InetLog_INSERT
ON InetLog
FOR INSERT
AS

IF EXISTS (SELECT * FROM inserted WHERE Target = 'AboutUs.htm')
 BEGIN
  UPDATE LogSummary
  SET LogSum_Count = (SELECT COUNT(*) FROM InetLog WHERE Target = 'AboutUs.htm')
  WHERE LogSum_Category = 'About Us'
 END

IF EXISTS (SELECT * FROM inserted WHERE Target = 'Services.htm')
 BEGIN
  UPDATE LogSummary
  SET LogSum_Count = (SELECT COUNT(*) FROM InetLog WHERE Target = 'Services.htm')
  WHERE LogSum_Category = 'Services'
 END
go
```

The trigger simply extends on the examples presented earlier and when the IF criteria is met the associated row in LogSummary is updated. The following shows the trigger works as expected.

```
INSERT InetLog VALUES ('111.111.111.111', '4/1/29 12:00:50','Default.htm')
INSERT InetLog VALUES ('111.111.111.111', '4/1/29 12:01:01','AboutUs.htm')
INSERT InetLog VALUES ('111.111.111.111', '4/1/29 12:02:01','Services.htm')
INSERT InetLog VALUES ('111.111.111.111', '4/1/29 12:03:01','Products.htm')
INSERT InetLog VALUES ('111.111.111.111', '4/1/29 12:04:50','Default.htm')
INSERT InetLog VALUES ('111.111.111.111', '4/1/29 12:05:01','AboutUs.htm')
INSERT InetLog VALUES ('111.111.111.111', '4/1/29 12:06:01','Services.htm')
INSERT InetLog VALUES ('111.111.111.111', '4/1/29 12:07:01','Products.htm')
go
SELECT * FROM LogSummary

-- Results --

LogSum_Category                LogSum_Count
------------------------------ ------------
About Us                       2
Services                       2
```

Before I leave this section I must mention that this *homemade* solution is not the preferred way to monitor web site traffic. I certainly had fun researching ODBC-logging and writing the code, but I actually suggested the client buy a commercial software package like WebTrends to implement this functionality. WebTrends is a great product and allows you to perform detailed analysis of a site's traffic. The client did not want to spend any money on *real* software, but instead on me:))

Part II

In the next installment of this article I will discuss UPDATE, DELETE and INSTEAD OF Triggers. For those of you who have not worked with SQL Server 2000, you probably have not heard of INSTEAD OF triggers. They were added in the latest version of product to help with updatable views. Part II is posted!

Garth
www.SQLBook.com

< Previous | Next: An Introduction to Triggers -- Part II >

Related Articles

Using DDL Triggers in SQL Server 2005 to Capture Schema Changes (13 August 2007)

Audit Triggers for SQL Server (8 May 2002)

Code to find out the statement that caused the trigger to fire! (16 April 2002)

An Introduction to Triggers -- Part II (4 November 2001)

Replicating Triggers (14 August 2000)

Other Recent Forum Posts

Cost Tracker in SSRS 08 (1 Reply)

UDFs GETWORDCOUNT, GETWORDNUM (24 Replies)

SSRS need help using inscope with a matrix (2 Replies)

only alphabat show (2 Replies)

show changes from and changes to in a history tabl (4 Replies)

Deleted Database (6 Replies)

WHERE clause statement problem (4 Replies)

Detect LDF corruption (4 Replies)