DML Trigger Basics and Templates

In this appendix, I cover a template for triggers that I use in all of the chapters where I need a trigger. Along with this, I cover how triggers are written. These templates are the ones I use for all the real TRIGGER objects I create in my career as well, because there are a lot of idiosycracies with triggers that are easily forgotten.

In this appendix, we will cover the two different types of DML triggers that are coded very similarly, but serve different purposes:

* The AFTER TRIGGER is made to be called after an INSERT, UPDATE, or a DELETE statement occurs, along with the any constraints on the table are called. It is used to apply complex checks and to add side effects to a DML operation as may be needed. AFTER triggers are usually used for handling rules that won’t fit into the mold of a constraint, for example, rules that require data to be stored, such as a logging mechanism. You may have a virtually unlimited number of AFTER triggers that fire on INSERT, UPDATE, and DELETE, or any combination of these actions.
* The INSTEAD OF type of trigger is called before a DML operation, even before any CONSTRAINT objects do their work, other than DEFAULT constraints. The actual operation on the table is not done, you have to re-perform the operation the caller executed. So, for an INSERT INSTEAD OF TRIGGER object, if you want the data to go in the table, you have to recode the INSERT statement. In this way, you can do whatever you want with the data, either doing exactly what was requested by the user or doing something completely different (you can even just ignore the operation altogether). You can have a maximum of one INSTEAD OF INSERT, UPDATE, and DELETE trigger of each type per table. It is allowed, but not a generally good idea in most cases, to combine all three into one and have a single trigger that fires for all three operations.

Note  In this appendix, I will primarily cover interpreted triggers that you can place on an on-disk table. In the final section of the Appendix, I will review a few of the differences between the interpreted triggers and natively compiled triggers you can place on memory optimized tables. The big difference between the natively compiled triggers and interpreted are the coding limitations of native compilation.

Coding at DML TRIGGER object is very much like coding a STORED PROCEDURE object, with some important differences. Instead of having data passed into the stored procedure, you use two special virtual tables that are instantiated for the life of the operation and are scoped strictly to the code executing directly in the trigger. These tables are called inserted and deleted. The inserted table contains new or updated rows for an INSERT or UPDATE operation, and the deleted table contains the row values from the affected rows as they were before the DML operation that occurred. It will be the rows that have been deleted for a DELETE statement execution or that have been modified by an UPDATE statement. It is important to note that these tables can have multiple rows in them if you modify more than one row in your DML statement. Also, they are tightly scoped to the executing trigger, if you call a stored procedure, the tables will not be accessible, and if your trigger causes another trigger to fire, the contents of the tables will be for the currently executing trigger.

This appendix introduces a set of templates that are used in several chapters of this book, as well as providing fairly exhaustive examples of using the template. (In Chapter 8, I presented the more “realistic” uses of triggers, while here in this appendix I am trying to show some of the more esoteric ways triggers can be used if you have a complex need that is not typical.)

While triggers are similar to procedures, there are some fairly major differences that are important. In the following bulleted list are overviews of some of the concepts you need to be cognizant of when writing triggers, including some settings that can be very useful:

* Multi-row operations: Triggers fire once for a DML operation, regardless of how many rows are affected. Hence, statements within triggers have to be coded with multiple rows in mind. This can be confusing, because unlike what seems to be natural, trigger code for validations typically needs to look for rows that don’t meet your criteria, instead of those that do. Hence, you have to code your triggers in a way that recognizes that more than one row in the table might be being modified and only one might be wrong.
* Performance: When a reasonable number of rows are dealt with in a TRIGGER object (certainly any amount that are part of typical OLTP operations), they are quite fast, but as the number of modified rows increases into the thousands or more, triggers can become tremendous performance drains. This is largely because the inserted and deleted tables aren’t real tables so they don’t have indexes that the optimizer can optimize for. Hence, the plans chosen for the queries can be fairly optimistic about the number of rows in inserted and deleted virtual tables. As said, because OLTP systems usually deal with small numbers of rows at a time, there’s rarely a major performance hit because of using triggers, but it is something you have to be cognizant of anytime you create a new TRIGGER object and particularly when you have to do a large data load/purge.
* Determining modified columns: For performance reasons, you may not want to validate data that’s in a column that isn’t affected by a DML statement. You can tell which columns were part of the INSERT or UPDATE statement by using the UPDATE(<columnName>) function to check the column to see whether it was involved in the DML operation. This does not indicate that a value has changed, just that the column was referenced in the INSERT or UPDATE statement. For example, given the simple statement UPDATE tableName SET column1=column1, the values would not change, but UPDATE(column1) would return true. (There is also another method using the function COLUMNS\_UPDATED(columnBitmask) to check the columns by their position in the table. 1+2+4 = 7 would mean the first three columns were updated, but it’s generally a bad practice to address columns in a table positionally, for future maintenance purposes.)

Having multiple AFTER triggers for the same action: It’s possible to have many different triggers on a table, which gives you the ability to add triggers to third-party systems without touching triggers that the third-party created. However, often the order of TRIGGER objects are called can be important, especially when you have to deal with validating data that another trigger might modify. You do get some minor control over the order in which triggers fire. Using the sp\_settriggerorder system stored procedure you can choose the first and the last trigger to fire. Usually this is all you need, because there are places where you want to set the first trigger (often the third-party trigger) and the last trigger (such as a TRIGGER object to implement an audit trail might fire last to audit how the data looks at the end of the process. Or you might make it first, if you want to capture the user’s input, not how it looks after transformation.)

* Nesting triggers: Take care when building AFTER triggers that modify data (the same table or other tables) because these updates could in turn cause other triggers to fire. INSTEAD OF triggers always cause other triggers to fire and that is usually not an issue. I am not going to go into deep detail with the concerns with nesting triggers, but it is important to know that unlike any DML statement, DML in TRIGGER objects may or may not cause additional triggers to fire based on the following settings. Make sure that you test your triggers and settings to ensure that what you expect to occur does occur. There are three important settings to be concerned with:  
    
  Server option—sp\_serveroption—nested triggers: When this setting is set to 1, it indicates that if you modify a different table, that table’s trigger will be fired. This setting is usually set to 1, because it allows for data validations to occur in the other tables without coding every business rule again.  
    
  Database option—ALTER DATABASE–RECURSIVE\_TRIGGERS: When set to ON, when an AFTER trigger modifies data in the same table, the triggers for that table execute again. This setting is usually set to OFF. Because it’s common practice to modify the same table in the trigger, it’s assumed that any modifications done in a trigger will meet all business rules for the same table.  
    
  Server option—sp\_serveroption—disallow results from triggers: It is a very bad practice to return results from a trigger during production code. However, when debugging a TRIGGER object it can be a very useful practice. Turn this setting on will ensure that any trigger that tries to return data to the client will get the following error message: Msg 524, Level 16, State 1, Procedure Test$InsertTrigger, A trigger returned a resultset and the server option 'disallow results from triggers' is true.

Because multi-row operations are the most frequently messed up aspect of trigger writing, it’s worth discussing this aspect in greater detail. If you insert a thousand rows, the inserted table will have a thousand rows. The deleted table will remain empty on an insert. When you delete rows, the deleted table is filled, and the inserted table remains empty. For an UPDATE, both tables are filled with the rows in the updated table that had been modified as they appeared before and after the update.

Because of this, writing validations must take this into consideration. For example, the following all-too-typical approach wouldn’t be a good idea:

SELECT @column1 = column1 FROM inserted;

IF @column1 < 0

BEGIN

--handle the error

This is wrong because only a single row would be checked—in this case, the last row that the SELECT statement came to. (There’s no order, but @column1 would be set to every value in the inserted table and would end up with only one value of the changed rows.) Instead, the proper way to code this would be as follows:

If EXISTS (SELECT \*

FROM inserted

WHERE column1 < 0)

BEGIN

--handle the error

This works because each row in the inserted table is checked against the criteria. If any rows do match the criteria, the EXISTS Boolean expression returns True, and the error block is started.

You’ll see this more in the example triggers, and it was covered in detail in Chapter 8. However, you need to make a conscious effort as you start to code triggers to consider what the effect of modifying more than one row would be on your code, because you certainly don’t want to miss an invalid value because of coding like the first wrong example.

If you need a full reference on the many details of triggers, refer to Microsoft Docs “CREATE TRIGGER (Transact-SQL)” ([docs.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql](https://docs.microsoft.com/en-us/sql/t-sql/statements/create-trigger-transact-sql)). In the following section, we’ll look at the different types of triggers, the basics of coding them. Luckily, for the most part triggers after you get past the need to do multi-row checks, triggers are straightforward, and the basic settings will work just fine.

Note I don't think I could stress nearly enough about the need to understand multi-row operations in triggers. Almost every time a question is raised on the forums about triggers, the code that gets posted contains code that will handle only one row.

* My recommendation for common places you may need to use triggers when you need to do the following:Check inter-row rules, where just looking at the current row isn’t enough for the constraints
* Check inter-table constraints, when rules require access to data in a different table
* Introduce desired side effects to your data-modification queries, such as maintaining required denormalizations
* Guaranteeing that no INSERT, UPDATE, or DELETE operations can be executed on a table, even if the user does have rights to perform the operation
* Perform non-standard referential integrity, like if you need cross-database integrity

Some of these operations could possibly be done in an application layer or even using security, but for the most part, these operations are far easier and safer, particularly for data integrity, when done automatically using a TRIGGER object.

When it comes to data protection, the primary advantage that triggers have over constraints is the ability to access other tables seamlessly and to operate on multiple rows at once. In a trigger, you can run almost every T-SQL command, except for the following (you also can’t execute statements that must be in their own batch, such as CREATE PROCEDURE, or CREATE TRIGGER, though admittedly using Dynamic SQL you can execute ANY statement that can be executed inside an active transaction.):

* ALTER DATABASE • RECONFIGURE
* CREATE DATABASE • RESTORE DATABASE
* DROP DATABASE • LOAD LOG
* RESTORE LOG • LOAD DATABASE

In addition, you cannot use the following commands on the table that the trigger protects:

* CREATE INDEX • ALTER PARTITION FUNCTION
* ALTER INDEX • DROP TABLE
* DROP INDEX • ALTER TABLE
* DBCC REINDEX

It wouldn’t be a very good design to change the schema of *any* table or do any of the things in this list in a trigger anyhow, much less the one that the trigger is built on, so these aren’t overly restrictive requirements at all.

When using triggers, it is important to keep them as lean as possible. Avoid using cursors, calling STORED PROCEDURE objects (unless it is like the error log stored procedure that is called only once, with one piece of data), or doing any sort of looping operation if at all possible, and instead keep triggers lean to get the operation done fast. If you need to do some extra processing, such as sending an e-mail for every row affected by the trigger, create a table that can be used as a queue for another process to work on. When your code is executing in a trigger, you will almost certainly be holding locks, unnecessarily forcing other users to wait, and you cannot be completely certain that the rows that were modified to fire the trigger will actually be committed, and if it gets rolled back, you probably won’t want something like an e-mail sent. You can’t enlist your cloud-based email system in your transaction, so if you send the e-mail directly via the trigger, it may have already sent the e-mail by the time the transaction is rolled back.

**Note** In SQL Server 2005 and later, the SQL-based mail object was changed to use Service Broker to implement mail, so if you roll back the transaction, it will roll back the mail command, but you would still have to use a cursor to call the send mail procedure.

Error handling for TRIGGER objects will use the simple error logging procedure that is implemented in Chapter 13 as part of the “Building Reusable Components” section. It will use the following code:

--[Error logging section]

DECLARE @ERROR\_NUMBER int = ERROR\_NUMBER(),  
 @ERROR\_PROCEDURE sysname = ERROR\_PROCEDURE(),

@ERROR\_MESSAGE varchar(4000) = ERROR\_MESSAGE()

EXEC ErrorHandling.ErrorLog$Insert

@ERROR\_NUMBER,@ERROR\_PROCEDURE,@ERROR\_MESSAGE;

It captures the error values from the ERROR\_ system functions values that are populated after an error occurs, and then calls the procedure to write the data to a table named ErrorHandling.ErrorLog. This can be commented out or removed from your template if you don’t care to log errors, and it can also be used in any of your other code to capture errors (noting it needs to come after a ROLLBACK TRANSACTION or the logged errors will vanish.) The table for logging is built with the following DML:

--Database I will use for examples in this appendix

CREATE DATABASE AppendixB;  
GO

USE AppendixB;

GO

CREATE SCHEMA ErrorHandling;

GO

CREATE TABLE ErrorHandling.ErrorLog(

ErrorLogId int NOT NULL IDENTITY CONSTRAINT PKErrorLog PRIMARY KEY,

Number int NOT NULL,

Location sysname NOT NULL,

Message varchar(4000) NOT NULL,

LogTime datetime2(3) NULL

CONSTRAINT DFLTErrorLog\_LogTime DEFAULT (SYSDATETIME()),

ServerPrincipal sysname NOT NULL

--use original\_login to capture the user name of the actual user

--not a user they have impersonated

CONSTRAINT DFLTErrorLog\_ServerPrincipal DEFAULT (ORIGINAL\_LOGIN())

);

It uses this STORED PROCEDURE object to write the log rows:

CREATE PROCEDURE ErrorHandling.ErrorLog$Insert

(

@ERROR\_NUMBER int,

@ERROR\_LOCATION sysname,

@ERROR\_MESSAGE nvarchar(4000)

) AS

------------------------------------------------------------------------

-- Writes a row to the error log. If an error occurs in the call (such as a

-- NULL value) It writes a row to the error table. If that call fails an

-- error will be returned

--

-- 2020 Louis Davidson – drsql@hotmail.com – drsql.org

------------------------------------------------------------------------

BEGIN

SET NOCOUNT ON;

BEGIN TRY

INSERT INTO ErrorHandling.ErrorLog(Number, Location,Message)

SELECT @ERROR\_NUMBER,

COALESCE(@ERROR\_LOCATION, N'No Object'),@ERROR\_MESSAGE;

END TRY

BEGIN CATCH

INSERT INTO ErrorHandling.ErrorLog(Number, Location, Message)

VALUES (-100, 'Utility.ErrorLog$insert',

'An invalid call was made to the error log procedure ' +

ERROR\_MESSAGE());

END CATCH;

END;

# AFTER Triggers

AFTER triggers fire after all the CONSTRAINT objects on the table pass their predicates. For instance, it wouldn’t be useful to insert rows in a child table, causing its entire trigger/constraint chain to fire, when the far cheaper operation of checking a FOREIGN KEY reference or a CHECK constraint might fail the operation. Equally, you wouldn’t want to check the status of all the rows in your table until you’ve completed all your changes to them; the same could be said for cascading DELETE operations.

All triggers I write in this book (other than short ones to show a concept) will use the following common template that sets up the code that is used over and over again (and is pretty tedious to set up the first time anyhow). I have done my best to comment the code here and, in all uses so it will be easy for you to reuse.

CREATE TRIGGER <schema>.<tablename>$<actions>[<purpose>]Trigger

ON <schema>.<tablename>

AFTER <comma delimited actions> AS

BEGIN

SET NOCOUNT ON; --to avoid the rowcount messages

SET ROWCOUNT 0; --in case the client has modified the rowcount

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

DECLARE @msg varchar(2000), --used to hold the error message

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

@rowsAffected int = (SELECT COUNT(\*) FROM inserted);

-- @rowsAffected int = (SELECT COUNT(\*) FROM deleted);

--no need to continue on if no rows affected

IF @rowsAffected = 0 RETURN;

BEGIN TRY

--[validation section]

--[modification section]

END TRY

BEGIN CATCH

IF @@trancount > 0

ROLLBACK TRANSACTION;

--[Error logging section]

DECLARE @ERROR\_NUMBER int = ERROR\_NUMBER(),

@ERROR\_PROCEDURE sysname = ERROR\_PROCEDURE(),

@ERROR\_MESSAGE varchar(4000) = ERROR\_MESSAGE();

EXEC ErrorHandling.ErrorLog$Insert

@ERROR\_NUMBER,@ERROR\_PROCEDURE,@ERROR\_MESSAGE;

THROW; --will halt the batch or be caught by the caller's catch block

END CATCH;

END;

Tip The AFTER keyword was introduced in the 2000 version of SQL Server when INSTEAD OF triggers were introduced. Prior to this, the keyword was FOR, since the trigger was for certain actions. Both are still allowed and common, but it is best to use AFTER in all new code.

I generally write triggers keeping things simple. When the first error occurs, an error will be raised, and I roll back the transaction to halt any further commands. You can get more complex and capture multiple errors, try not to rollback everything on a failure, but when it comes to error handling, it is generally best to just end the work you are attempting when something goes wrong.

In the TRIGGER template, there are three areas where code is added, denoted by comments headers:

* --[validation section]: In this section you will add validation logic that will be executed after the DML has been performed on the table. This would be used instead of a constraint when you need to code some complex validation that doesn’t fit the mold of a constraint well.
* --[modification section]: Used for DML statements to modify the contents of tables or to do some other operation that has side effects of some sort. In this section, you might modify the same table as the triggering table or any other table.
* --[error logging section]: This is the part of the trigger where you log any errors, either by inserting them into a table, or to the error log using xp\_logevent. As previously noted, in the trigger templates in this appendix, I use the ErrorHandling.ErrorLog$Insert procedure we created in Chapter 14 that writes to the ErrorHandling.ErrorLog table created earlier in the appendix.

The form I use for almost every [validation section] I build is similar to the following code. I typically try to code different messages for the case when one row was affected by the DML operation by checking the @rowsAffected variable that is set earlier in the trigger by checking the number of rows in the inserted or deleted tables. This allows for better error messages (like to include the invalid value, something that can really help when you have executed 200 statements and one failed) for more typical singleton case, and a more generic explanation when many rows were changed. (If you commonly get errors in multi-row operations, you can enhance the multi-row error message to use an aggregate to return a single invalid value, but this can be costly for large updates.

IF EXISTS (<some condition,

usually using inserted and/or deleted tables>)

BEGIN

IF @rowsAffected = 1 --custom error message for single row

SELECT @msg = CONCAT('<reason>', inserted.value)

FROM inserted; --and/or deleted, depending on action

ELSE

SELECT @msg = '<more generic reason>';

--in the TRY . . . CATCH block, this will redirect to the CATCH

THROW 50000, @msg, 16;

END;

The [modification section] section can simply be set up as simple INSERT, UPDATE, or DELETE statements thanks to the TRY-CATCH construct. Any errors raised because of the DML, such as from a CONSTRAINT object or another TRIGGER, will be caught and sent to a CATCH block.

It is necessary here to provide further explanation about how I capture the number of rows affected. It seems like @@ROWCOUNT would be sufficient, but in SQL Server versions after 2005, the MERGE statement changed that because every action clause in the MERGE statement calls the trigger, and the @@ROWCOUNT value is the one from the MERGE call, not the individual clauses. Hence, if your MERGE statement inserted 10 rows, but had branches for removing and modifying rows that did not modify any rows, all three could execute triggers, and @@ROWCOUNT would be 10 for all three calls.

So instead of @@ROWCOUNT, I use the following (comments from TRIGGER object code removed for clarity):

DECLARE @msg varchar(2000), --used to hold the error message

@rowsAffected int = (SELECT COUNT(\*) FROM inserted)

--@rowsAffected int = (SELECT COUNT(\*) FROM deleted)

;

For the insert and update triggers, I count rows in inserted, and for the delete operation, it will use the deleted virtual table.

**Note** Be careful with this counting. If you count inserted values in the DELETE trigger, it will succeed EVERY TIME, and do nothing, making you very angry when you realize you have a lot of bad data to clean up, and even in your development server this is frustrating.

As an example of the issue, I take the following simple table (In the downloaded code, this code continues in the database called AppendixB):

CREATE SCHEMA Demo;

GO

CREATE TABLE Demo.Test

(

TestId int

);

Next I add the following simple TRIGGER object that covers all actions. All the trigger does is to output a row that tells us the value of @@ROWCOUNT global variable, the number of rows in each of the virtual table, and then a calculated value that tells you if the operation was an insert, update, or a delete.

CREATE TRIGGER Demo.Test$InsertUpdateDeleteTrigger

ON Demo.Test

AFTER INSERT, UPDATE, DELETE AS

BEGIN

--stores the number of rows affected

DECLARE @rowcount int = @@ROWCOUNT,

@rowcountInserted int = (SELECT COUNT(\*) FROM inserted),

@rowcountDeleted int = (SELECT COUNT(\*) FROM deleted);

SELECT @rowcount as [@@ROWCOUNT],

@rowcountInserted as [@rowcountInserted],

@rowcountDeleted as [@rowcountDeleted],

CASE WHEN @rowcountInserted = 0 THEN 'DELETE'

WHEN @rowcountDeleted = 0 THEN 'INSERT'

ELSE 'UPDATE' END AS Operation;

END;

Remember, if you have set the disallow results from triggers; you will need to turn that off for this to work. Be sure and turn it back on after our demonstrations, as while this configuration value is an excellent way to allow yourself to use results for testing, in practice you do not want to have results returned from your triggers.

EXEC sp\_configure 'show advanced options',1;

RECONFIGURE;

GO

EXEC sp\_configure 'disallow results from triggers',0;

RECONFIGURE;

So now, let’s add two rows to the table:

INSERT INTO Demo.test

VALUES (1),

(2);

From the trigger, you will see the following output:

@@ROWCOUNT @rowcountInserted @rowcountDeleted Operation  
----------- ----------------- ---------------- ---------  
2 2 0 INSERT

As expected, @@ROWCOUNT was 2, and the count of rows from the inserted table is also 2. Now, execute this simple MERGE that uses a CTE that will give us three operations, a delete for testId = 1, an update for testId = 3, and an insert for testId = 3.

WITH testMerge AS (SELECT \*

FROM (VALUES(2),(3)) AS testMerge (TestId))

MERGE Demo.Test

USING (SELECT TestId FROM testMerge) AS source (TestId)

ON (Test.TestId = source.TestId)

WHEN MATCHED THEN

UPDATE SET TestId = source.TestId

WHEN NOT MATCHED THEN

INSERT (TestId) VALUES (Source.TestId)

WHEN NOT MATCHED BY SOURCE THEN

DELETE;

Now you will see the following output, which appears as if 9 rows have been modified, even though what actually happened was that 1 row was inserted, 1 row was updated, and 1 more row was deleted:

@@ROWCOUNT @rowcountInserted @rowcountDeleted Operation  
----------- ----------------- ---------------- ---------  
3 1 0 INSERT

@@ROWCOUNT @rowcountInserted @rowcountDeleted Operation  
----------- ----------------- ---------------- ---------  
3 1 1 UPDATE

@@ROWCOUNT @rowcountInserted @rowcountDeleted Operation  
----------- ----------------- ---------------- ---------  
3 0 1 DELETE

In my normal triggers, I typically do not mix any DML operations (so I have an insert trigger, an update trigger, and a separate delete trigger), and if I used the @rowcount value for anything (such as we usually quit the trigger if 0 rows were affected), we would have treated each trigger as having multiple rows. In either case, in this TRIGGER object, I had three calls with a single row modification, but @@ROWCOUNT indicated three rows.

**Note** Having a TRIGGER object that handles multiple actions is useful for simple tasks, like if you are logging changes in an operation, but gets messy when you have to do validations that can change based on the action, which is generally typical.

In Chapter 8, AFTER triggers are used to implement triggers to solve realistic sorts of problems:

* Range checks on multiple rows: Make sure that a summation of values of column, usually over some grouping, is within some specific range of values.
* Maintaining summary values (only as necessary): Basically, updating one value whenever one or more values change in a different table.
* Cascading inserts: After a row is inserted into a table, one or more other new rows are automatically inserted into other tables. This is frequently done when you need to initialize a row in another table, quite often a status of some sort.
* Child-to-parent cascades: Performing cascading operations that cannot be done using a typical foreign key constraint.
* Relationships that span databases and servers: Basic referential integrity only works within the confines of a database.

In this appendix, I present an extended, rather abstract example that demonstrates some of the power and possible uses of TRIGGER objects. I create a TRIGGER object that makes sure that data, grouped on a given value always stays > 0. The triggers in this example are some of the most complex triggers I have written (since the advent of constraints, at least), and I did this to make the TRIGGER object a bit complex to show some of the power that is seldom needed (but certainly interesting when needed).

I take the data from a table called Example.AfterTriggerExample, which has a simple integer key; a column called GroupingValue, which serves as a kind of Account to group on; and Example.AfterTriggerExampleGroupingBalance, which holds the running balance.

CREATE SCHEMA Example;

GO

--this is the “transaction” table

CREATE TABLE Example.AfterTriggerExample

(

AfterTriggerExampleId int

CONSTRAINT PKAfterTriggerExample PRIMARY KEY,

GroupingValue varchar(10) NOT NULL,

Value int NOT NULL

);

GO

--this is the table that holds the summary data

CREATE TABLE Example.AfterTriggerExampleGroupBalance

(

GroupingValue varchar(10) NOT NULL

CONSTRAINT PKAfterTriggerExampleGroupBalance PRIMARY KEY,

Balance int NOT NULL

);

Then I create the following AFTER INSERT TRIGGER object. The code is commented as to what is occurring in there, but the basics are that there are two major sections, one to validate data, which does a summation on the items in the table and makes sure the sums are greater than 0, and another to write the denormalization/summary data. I could have implemented the non-negative requirement by putting a constraint on the Example.AfterTriggerExampleGroupBalance to require a **Balance >= 0**, but I am trying to show triggers with a validation and a cascading/modification action.

CREATE TRIGGER Example.AfterTriggerExample$InsertTrigger

ON Example.AfterTriggerExample

AFTER INSERT AS

BEGIN

SET NOCOUNT ON; --to avoid the rowcount messages

SET ROWCOUNT 0; --in case the client has modified the rowcount

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

DECLARE @msg varchar(2000), --used to hold the error message

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

@rowsAffected int = (SELECT COUNT(\*) FROM inserted);

-- @rowsAffected int = (SELECT COUNT(\*) FROM deleted);

--no need to continue on if no rows affected

IF @rowsAffected = 0 RETURN;

BEGIN TRY

--[validation section]

--Use a WHERE EXISTS to inserted to make sure not

--to duplicate rows in the set if > 1 row is modified for the

--same grouping value

IF EXISTS (SELECT AfterTriggerExample.GroupingValue

FROM Example.AfterTriggerExample

WHERE EXISTS (SELECT \*

FROM Inserted

WHERE AfterTriggerExample.GroupingValue

= Inserted.Groupingvalue)

GROUP BY AfterTriggerExample.GroupingValue

HAVING SUM(Value) < 0)

BEGIN

IF @rowsAffected = 1

SELECT @msg = CONCAT('Grouping Value "', GroupingValue,

'" balance value after operation must be greater than 0')

FROM inserted;

ELSE

SELECT @msg = CONCAT('The total for grouping values ',

' must be greater than 0');

THROW 50000, @msg, 16;

END;

--[modification section]

--get the balance for any Grouping Values used

--in the DML statement

WITH GroupBalance AS

(SELECT AfterTriggerExample.GroupingValue,

SUM(Value) AS NewBalance

FROM Example.AfterTriggerExample

WHERE EXISTS (SELECT \*

FROM Inserted

WHERE AfterTriggerExample.GroupingValue =

Inserted.Groupingvalue)

GROUP BY AfterTriggerExample.GroupingValue )

--use merge because there may not be an existing balance row for

--the grouping value

MERGE Example.AfterTriggerExampleGroupBalance

USING (SELECT GroupingValue, NewBalance FROM GroupBalance)

AS source (GroupingValue, NewBalance)

ON (AfterTriggerExampleGroupBalance.GroupingValue =

source.GroupingValue)

WHEN MATCHED THEN --a grouping value already existed

UPDATE SET Balance = source.NewBalance

WHEN NOT MATCHED THEN --this is a new grouping value

INSERT (GroupingValue, Balance)

VALUES (Source.GroupingValue, Source.NewBalance);

END TRY

BEGIN CATCH

IF @@trancount > 0

ROLLBACK TRANSACTION;

--[error logging section]

DECLARE @ERROR\_NUMBER int = ERROR\_NUMBER(),

@ERROR\_PROCEDURE sysname = ERROR\_PROCEDURE(),

@ERROR\_MESSAGE varchar(4000) = ERROR\_MESSAGE()

EXEC ErrorHandling.ErrorLog$Insert

@ERROR\_NUMBER,@ERROR\_PROCEDURE,@ERROR\_MESSAGE;

THROW; --will halt the batch or be caught by the caller's catch block

END CATCH

END;

After adding the trigger, we can test it. Let’s try to add two new rows, each as single row. The first two inserts will work, and data will be added. In the first insert, it will cause the MERGE statement to add a new row to the Example.AfterTriggerExampleGroupBalance table, the second updating that row:

INSERT INTO Example.AfterTriggerExample

(AfterTriggerExampleId,GroupingValue,Value)

VALUES (1,'Group A',100);

GO

INSERT INTO Example.AfterTriggerExample

(AfterTriggerExampleId,GroupingValue,Value)

VALUES (2,'Group A',-50);

Before we look at the summary table, let’s check the case where the balance ends up being less than 0 with this row with -100 added the 100 and -50 from earlier causing a negative balance:

INSERT INTO Example.AfterTriggerExample

(AfterTriggerExampleId,GroupingValue,Value)

VALUES (3,'Group A',-10);

This will cause the following error. Note that it uses the single row error message we set up, telling us the group that caused the error to help make error tracking easier:

Msg 50000, Level 16, State 16, Procedure AfterTriggerExample$InsertTrigger, Line 39  
Grouping Value "Group A" balance value after operation must be greater than 0

Next, to show a multi-row error, I try to add two new rows, but not enough to make the sum greater than 0.

INSERT INTO Example.AfterTriggerExample(AfterTriggerExampleId,GroupingValue,Value)

VALUES (3,'Group A',10),

(4,'Group A',-100);

This causes the following error far more generic answer:

Msg 50000, Level 16, State 16, Procedure AfterTriggerExample$InsertTrigger, Line 39  
The total for the grouping value must be greater than 0

Of course, if you need better messages, you can clearly build a more interesting error handler, but since almost all code that may cause an error in most systems is going to be row at a time, it is probably not worth it. While it is fairly easy to build a single row error message the multi-row message is not going to be quite so easy to do, because you would need to know which rows were wrong, and that would make the cost of validation a lot costlier and a lot more coding.

For the error, that was raised, you will be able to see the error in the ErrorHandling.ErrorLog table, if you implemented it:

SELECT ErrorLog.Message,Logtime, ErrorLog.ServerPrincipal

FROM ErrorHandling.ErrorLog

WHERE ErrorLog.Location = 'AfterTriggerExample$InsertTrigger';

This will return at least two row, depending on what other calls you have made:

Message Logtime ServerPrincipal

--------------------------------- ------------------------ ---------------

Grouping Value "Group A" balan... 2020-07-31 16:08:32.847 Domain\drsql

The total for grouping values ... 2020-07-31 16:08:54.978 Domain\drsql

Next, we will do another multi-row update that does not fail validation:

INSERT INTO Example.AfterTriggerExample(AfterTriggerExampleId,GroupingValue,Value)

VALUES (5,'Group A',100),

(6,'Group B',200),

(7,'Group B',150);

Now, let’s look at the data that has been created:

SELECT \*

FROM Example.AfterTriggerExample;

SELECT \*

FROM Example.AfterTriggerExampleGroupBalance;

This returns:

AfterTriggerExampleId GroupingValue Value  
--------------------- ------------- -----------  
1 Group A 100  
2 Group A -50  
5 Group A 100  
6 Group B 200  
7 Group B 150

GroupingValue Balance  
------------- -----------  
Group A 150  
Group B 350

Take a look at the data, and make sure that the numbers add up. Test your code in as many ways as you possibly can. One of the reasons we don’t generally write such messy code in triggers is that they will need a lot of testing because of the many ways they can be called. Also, all of the triggers that we are building basically maintain summary data that replaces optimizing a fairly simple query:

SELECT GroupingValue, SUM(Value) AS Balance  
FROM Example.AfterTriggerExample

GROUP BY GroupingValue;

**Note** Sometimes it is useful/interesting to do an exercise like this to learn the difficulties in using code in certain ways to help you when you do need to do some complex code, and to show you why not to do this in your own code.

Next we move on to the UPDATE trigger. It is very similar in nature, and the validation section will be the same with the slight change to the FROM clause of the subquery to use a UNION of a query to the inserted and deleted table. You could make one validation trigger for INSERT, UPDATE, and DELETE, but I prefer to stick with one trigger with a bit of duplicated code, to make things easier to manage at the DBA level since multiple triggers complicates management.

Logically, an update is a delete and an insert of a new row. This is also how it is sometimes implemented internally, depending on the structure that is being updated, but you can always think of the operation as an insert and delete operation. So, when you change a row value, it deletes the old (represented in the deleted table) and creates a new row (in the inserted table). The MERGE statement for the UPDATE trigger has to deal with one additional branch to delete a group that has been added, so when you get a value, it could be a new group that was created by the update, a delete because the group from the deleted table no longer exists.

CREATE TRIGGER Example.AfterTriggerExample$UpdateTrigger

ON Example.AfterTriggerExample

AFTER UPDATE AS

BEGIN

SET NOCOUNT ON; --to avoid the rowcount messages

SET ROWCOUNT 0; --in case the client has modified the rowcount

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

DECLARE @msg varchar(2000), --used to hold the error message

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

@rowsAffected int = (SELECT COUNT(\*) FROM inserted);

-- @rowsAffected int = (SELECT COUNT(\*) FROM deleted);

--no need to continue on if no rows affected

IF @rowsAffected = 0 RETURN;

BEGIN TRY

--[validation section]

--Use a WHERE EXISTS to inserted to make sure

--not to duplicate rows in the set

--if > 1 row is modified for the same grouping value

IF EXISTS (SELECT AfterTriggerExample.GroupingValue

FROM Example.AfterTriggerExample

--need to check total on any rows that were

--modified, even if key change

WHERE EXISTS (SELECT \*

FROM Inserted

WHERE AfterTriggerExample.GroupingValue =

Inserted.Groupingvalue

UNION ALL

SELECT \*

FROM Deleted

WHERE AfterTriggerExample.GroupingValue =

Deleted.Groupingvalue)

GROUP BY AfterTriggerExample.GroupingValue

HAVING SUM(Value) < 0)

BEGIN

IF @rowsAffected = 1

SELECT @msg = CONCAT('Grouping Value "',

COALESCE(inserted.GroupingValue,

deleted.GroupingValue),

'" balance value after operation must',

' be greater than 0')

--only one row could be returned...

FROM inserted

CROSS JOIN deleted;

ELSE

SELECT @msg = CONCAT('The total for the grouping value ',

'must be greater than 0');

THROW 50000, @msg, 16;

END

--[modification section]

--get the balance for any Grouping Values used in the DML statement

--we know we will be summing on a NULL, with no better way

SET ANSI\_WARNINGS OFF;

WITH GroupBalance AS

(SELECT ChangedRows.GroupingValue, SUM(Value) as NewBalance

FROM Example.AfterTriggerExample

--the right outer join makes sure that we get all groups,   
 --even if no data remains in the table for a set

RIGHT OUTER JOIN

(SELECT GroupingValue

FROM Inserted

UNION

SELECT GroupingValue

FROM Deleted ) as ChangedRows

--the join make sure we only get rows for changed   
 --grouping values

ON ChangedRows.GroupingValue =

AfterTriggerExample.GroupingValue

GROUP BY ChangedRows.GroupingValue )

--use merge because the user may change the grouping value, and   
 --That could even cause a row in the balance table to need to be deleted

MERGE Example.AfterTriggerExampleGroupBalance

USING (SELECT GroupingValue, NewBalance FROM GroupBalance)

AS source (GroupingValue, NewBalance)

ON (AfterTriggerExampleGroupBalance.GroupingValue =

source.GroupingValue)

--should only happen with changed key

WHEN MATCHED and Source.NewBalance IS NULL

THEN DELETE

WHEN MATCHED THEN --normal case, where an amount was updated

UPDATE SET Balance = source.NewBalance

WHEN NOT MATCHED THEN --should only happen with changed

--key that didn't previously exist

INSERT (GroupingValue, Balance)

VALUES (Source.GroupingValue, Source.NewBalance);

SET ANSI\_WARNINGS ON;

END TRY

BEGIN CATCH

IF @@trancount > 0

ROLLBACK TRANSACTION;

--[Error logging section]

DECLARE @ERROR\_NUMBER int = ERROR\_NUMBER(),  
 @ERROR\_PROCEDURE sysname = ERROR\_PROCEDURE(),

@ERROR\_MESSAGE varchar(4000) = ERROR\_MESSAGE()

EXEC ErrorHandling.ErrorLog$Insert

@ERROR\_NUMBER,@ERROR\_PROCEDURE,@ERROR\_MESSAGE;

THROW; --will halt the batch or be caught by the caller's catch block

END CATCH;

END;

As a reminder, this is the where the balance is after the INSERT statement section:

GroupingValue Balance  
------------- -----------  
Group A 150  
Group B 350

So now, we update one row, setting a row that used to be 100 down to 50:

UPDATE Example.AfterTriggerExample

SET Value = 50 --Was 100

WHERE AfterTriggerExampleId = 5;

Looking at the balance again:

SELECT \*

FROM Example.AfterTriggerExampleGroupBalance;

You will see that the balance has gone down to 100 for Group A.

GroupingValue Balance  
------------- -----------  
Group A 100  
Group B 350

The next thing to test is changing the value that is being grouped on. This will cause a row in the balance table to be deleted, and a new one to be added by the MERGE statement.

--Changing the group key

UPDATE Example.AfterTriggerExample

SET GroupingValue = 'Group C'

WHERE GroupingValue = 'Group B';

Looking at the balance again:

GroupingValue Balance  
------------- -----------  
Group A 100  
Group C 350

You can see that the GroupValue has now changed from B to C, because the source data has all been changed. To set up the next example, let’s change all of the rows to 10:

--all rows

UPDATE Example.AfterTriggerExample

SET Value = 10 ;

This changes our data to look like:

SELECT \*

FROM Example.AfterTriggerExample;

SELECT \*

FROM Example.AfterTriggerExampleGroupBalance;

Returning:

AfterTriggerExampleId GroupingValue Value  
--------------------- ------------- -----------  
1 Group A 10  
2 Group A 10  
5 Group A 10  
6 Group C 10  
7 Group C 10

GroupingValue Balance  
------------- -----------  
Group A 30  
Group C 20

Check to make sure a multi-statement failure works:

--violate business rules

UPDATE Example.AfterTriggerExample

SET Value = -10;

This returns:

Msg 50000, Level 16, State 16, Procedure AfterTriggerExample$UpdateTrigger, Line 45  
The total for the grouping value must be greater than 0

Finally, we work on the DELETE trigger. It very much resembles the other triggers, except in this TRIGGER object we use only the deleted table as our primary table, and we don’t have to deal with the cases where new data is introduced, so we have one less matching criteria in the MERGE statement.

CREATE TRIGGER Example.AfterTriggerExample$DeleteTrigger

ON Example.AfterTriggerExample

AFTER DELETE AS

BEGIN

SET NOCOUNT ON; --to avoid the rowcount messages

SET ROWCOUNT 0; --in case the client has modified the rowcount

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

DECLARE @msg varchar(2000), --used to hold the error message

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

-- @rowsAffected int = (SELECT COUNT(\*) FROM inserted);

@rowsAffected int = (SELECT COUNT(\*) FROM deleted);

--no need to continue on if no rows affected

IF @rowsAffected = 0 RETURN;

BEGIN TRY

--[validation section]

--Use a WHERE EXISTS to inserted to make sure not to

--duplicate rows in the set if > 1 row is modified for the same

--grouping value

IF EXISTS (SELECT AfterTriggerExample.GroupingValue

FROM Example.AfterTriggerExample

WHERE EXISTS (SELECT \* --delete trigger needs only check

FROM deleted -- deleted rows

WHERE AfterTriggerExample.GroupingValue =

deleted.Groupingvalue)

GROUP BY AfterTriggerExample.GroupingValue

HAVING SUM(Value) < 0)

BEGIN

IF @rowsAffected = 1

SELECT @msg = CONCAT('Grouping Value "', GroupingValue,

'" balance value after operation must ',

'be greater than 0')

FROM deleted; --use deleted for deleted trigger

ELSE

SELECT @msg = CONCAT('The total for the grouping value ',

'must be greater than 0');

THROW 50000, @msg, 16;

END

--[modification section]

--get the balance for any Grouping Values used in the DML

--statement

SET ANSI\_WARNINGS OFF; --we know we will be summing on a NULL

WITH GroupBalance AS

(SELECT ChangedRows.GroupingValue, SUM(Value) as NewBalance

FROM Example.AfterTriggerExample

--the right outer join makes sure that we get all groups,

--even if no data remains in the table for a set

RIGHT OUTER JOIN

(SELECT GroupingValue

FROM Deleted ) as ChangedRows

--the join make sure we only get rows for

--changed grouping values

ON ChangedRows.GroupingValue =

AfterTriggerExample.GroupingValue

GROUP BY ChangedRows.GroupingValue)

--using merge because the delete may or may not remove the last

--row for a group which could even cause a row in the balance

--table to need to be deleted

MERGE Example.AfterTriggerExampleGroupBalance

USING (SELECT GroupingValue, NewBalance FROM GroupBalance)

AS source (GroupingValue, NewBalance)

ON (AfterTriggerExampleGroupBalance.GroupingValue =

source.GroupingValue)

WHEN MATCHED and Source.NewBalance IS NULL -- deleted the last key

THEN DELETE

WHEN MATCHED THEN --there were still rows left after the delete

UPDATE SET Balance = source.NewBalance;

SET ANSI\_WARNINGS ON; --restore proper setting

END TRY

BEGIN CATCH

IF @@trancount > 0

ROLLBACK TRANSACTION;

--[Error logging section]

DECLARE @ERROR\_NUMBER int = ERROR\_NUMBER(),  
 @ERROR\_PROCEDURE sysname = ERROR\_PROCEDURE(),

@ERROR\_MESSAGE varchar(4000) = ERROR\_MESSAGE()

EXEC ErrorHandling.ErrorLog$Insert

@ERROR\_NUMBER,@ERROR\_PROCEDURE,@ERROR\_MESSAGE;

THROW; --will halt the batch or be caught by the caller's catch block

END CATCH;

END;

To test this code, I set up all of the balances to be 0 by setting several rows to -5 to balance out a 10, and one -10 to balance out the other 10.

UPDATE Example.AfterTriggerExample

SET Value = -5

WHERE AfterTriggerExampleId in (2,5);

UPDATE Example.AfterTriggerExample

SET Value = -10

WHERE AfterTriggerExampleId = 6;

This leaves the data in the following state:

AfterTriggerExampleId GroupingValue Value  
--------------------- ------------- -----------  
1 Group A 10  
2 Group A -5  
5 Group A -5  
6 Group C -10  
7 Group C 10

GroupingValue Balance  
------------- -----------  
Group A 0  
Group C 0

First, we will try to delete the positive value from Group A:

DELETE FROM Example.AfterTriggerExample

WHERE AfterTriggerExampleId = 1;

This gives you the following message:

Msg 50000, Level 16, State 16, Procedure AfterTriggerExample$DeleteTrigger, Line 40  
Grouping Value "Group A" balance value after operation must be greater than 0

Next, we will try deleting both of the positive values:

DELETE FROM Example.AfterTriggerExample

WHERE AfterTriggerExampleId in (1,7);

This returns the generic multi-row error message that we created:

Msg 50000, Level 16, State 16, Procedure AfterTriggerExample$DeleteTrigger, Line 38  
The total for the grouping value must be greater than 0

Finally, we will systematically unload the table:

DELETE FROM Example.AfterTriggerExample

WHERE AfterTriggerExampleId = 6;

Now you can see that the C group is 10 because we deleted the negative value:

GroupingValue Balance  
------------- -----------  
Group A 0  
Group C 10

Now, we add back a Group B row:

INSERT INTO Example.AfterTriggerExample

VALUES (8, 'Group B',10);

Now our data looks like the following:

AfterTriggerExampleId GroupingValue Value  
--------------------- ------------- -----------  
1 Group A 10  
2 Group A -5  
5 Group A -5  
7 Group C 10  
8 Group B 10

GroupingValue Balance  
------------- -----------  
Group A 0  
Group B 10  
Group C 10

Delete the entire Group A:

DELETE FROM Example.AfterTriggerExample

WHERE AfterTriggerExampleId in (1,2,5);

Now the summary table looks like:

GroupingValue Balance  
------------- -----------  
Group B 10  
Group C 10

Finally, just clear the table:

DELETE FROM Example.AfterTriggerExample;

Leaving the tables both empty:

AfterTriggerExampleId GroupingValue Value  
--------------------- ------------- -----------

GroupingValue Balance  
------------- -----------

**Tip** All of this code to test the structures may seem like overkill but I wanted to stress to you that you need to do the same tasks on your own tables with TRIGGER objects attached. I had to fix a lot of errors in my code that didn’t seem obvious when just writing them, and I worked through this example because in many cases, you have to deal with all of the various things a user can do. For example, changing a PRIMARY KEY column value (or a UNIQUE key value, depending your implementation) is one that is pretty easy to forget.

At this point, we have a set of TRIGGER objects to maintain a summary table, but we haven’t as yet covered the entire case. To deal with this completely you would need to add triggers to the Example.AfterTriggerExampleGroupBalance table to make sure that the row couldn’t be modified unless it meets the criteria of summing up to values in the Example.AfterTriggerExample table or find a way to lock the table down completely, although any measures that prevent the dbo from changing the data would prevent the triggers we created from doing its job. I won’t present that work here, but it is basically the same problem that is covered in Chapter 8 in the AFTER trigger section covering range checks on multiple rows.

# INSTEAD OF Triggers

INSTEAD OF TRIGGER objects are different from AFTER TRIGGER objects in that they fire prior to the DML action being affected by the SQL operation. In fact, when you have an INSTEAD OF trigger on a table, it’s the first thing that’s done when you execute an INSERT, UPDATE, or DELETE statement on that table other than DEFAULT constraints. These triggers are named INSTEAD OF because they replace the native action the user executed. Inside the trigger code, you perform the action manually—either the action that the user performed or some other action.

INSTEAD OF triggers have a fairly narrow set of use cases, though they are very useful. The most typical use is to automatically populate a value such as the last time a change to a row occurred because since you perform the action in your code, no DML operation can override your action. They can be used to do some validations prior to the operation happening, but most of the time, it is easier to do validations in the AFTER TRIGGER, first because the entire process has completed when that code starts, and second because INSTEAD OF triggers require you to redo the operation,which can easily be a source of issue when someone forgets to include a new column in the trigger.

Another value of INSTEAD OF TRIGGER objects is that they can be created on a view to make a view editable in a straightforward manner for the client. While VIEW objects are generally editable (changing the underlying data), you are only allowed to update a single table at a time. By applying an instead of trigger to the view, you control the DML operation and you can insert to multiple tables in the background. Doing this, you encapsulate calls to all the affected tables in the TRIGGER object , much like you would a STORED PROCEDURE object, except now the view has all the properties of a physical table, hiding the actual implementation from users.

Probably the most obvious limitation of INSTEAD OF triggers is that you can have only one for each action (INSERT, UPDATE, and DELETE) on the table, or you can combine them just as you can for AFTER triggers, which I even more strongly advise against for INSTEAD OF triggers. We’ll use pretty much the same trigger template that we used for the T-SQL AFTER triggers, with only the modification that now you have to add a step to perform the action that the user was trying to do, which I comment as <perform action>. This tag indicates where we will put the DML operations to make modifications.

CREATE TRIGGER <schema>.<tablename>$InsteadOf<actions>Trigger

ON <schema>.<tablename>

INSTEAD OF <comma delimited actions> AS

BEGIN

SET NOCOUNT ON; --to avoid the rowcount messages

SET ROWCOUNT 0; --in case the client has modified the rowcount

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

DECLARE @msg varchar(2000), --used to hold the error message

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

@rowsAffected = (SELECT COUNT(\*) FROM inserted);

--@rowsAffected = (SELECT COUNT(\*) FROM deleted);

--no need to continue on if no rows affected

IF @rowsAffected = 0 RETURN;

BEGIN TRY

--[validation section]

--[modification section]

**--<perform action>**

END TRY

BEGIN CATCH

IF @@trancount > 0

ROLLBACK TRANSACTION;

--[Error logging section]

DECLARE @ERROR\_NUMBER int = ERROR\_NUMBER(),

@ERROR\_PROCEDURE sysname = ERROR\_PROCEDURE(),

@ERROR\_MESSAGE varchar(4000) = ERROR\_MESSAGE();

EXEC ErrorHandling.ErrorLog$Insert

@ERROR\_NUMBER,@ERROR\_PROCEDURE,@ERROR\_MESSAGE;

THROW; --will halt the batch or be caught by the caller's catch block

END CATCH;

END;

The most difficult part of working with INSTEAD OF TRIGGER objects is that coding the trigger to perform the operation, which also means you have to maintain the triggers anytime the table needs to change. Technically, performing the action is optional, and in the examples in Chapter 8, I used INSTEAD OF TRIGGER objects to prevent a DML operation from occurring altogether.

I most often use INSTEAD OF TRIGGER objects to set or modify values in my statements automatically so that the values are set to what I want, no matter what the client sends in a statement. A good example is a column to record the last time the row was modified. If you record last update times through client calls, it can be problematic if one of the client’s clock is a minute, a day, or even a year off. (You see this all the time in applications. My favorite example was in one system where phone calls appeared to be taking negative amounts of time because the client was providing the time when the call started and the server was recording when it stopped.) In Chapter 8, I demonstrated four ways you can use INSTEAD OF TRIGGER objects, and in Chapter 7 INSTEAD OF INSERT and UPDATE triggers were used to set the RowCreatedTime and RowLastModifiedTime column values.

**Note** INSTEAD OF triggers tend to really annoy some developers that want complete control over the changing of data, even more than an AFTER trigger that usually does validation or cascading that the non-data layer might not be able to do. However, because of how they work, they get the certain tasks done in the safest manner possible.

I will do an example very similar to the automatically maintained columns example for this appendix. The table follows, with an integer key, a column that will be formatted, and a couple of columns included in some tables to tell when they were last modified.

CREATE TABLE Example.InsteadOfTriggerExample

(

InsteadOfTriggerExampleId int NOT NULL

CONSTRAINT PKInsteadOfTriggerExample PRIMARY KEY,

FormatUpper varchar(30) NOT NULL,

RowCreatedTime datetime2(3) NOT NULL,

RowLastModifiedTime datetime2(3) NOT NULL

);

This example is a lot simpler than the AFTER trigger example, simply replacing the INSERT statement with one that does a bit of formatting on the incoming data. Generally speaking, the operations you will need to do in INSTEAD OF triggers are going to be very simple and straightforward. The example here will not only set the RowCreatedTime and RowLastModifiedTime values, it will format the FormatUpper column as uppercase in all cases.

CREATE TRIGGER Example.InsteadOfTriggerExample$InsteadOfInsertTrigger

ON Example.InsteadOfTriggerExample

INSTEAD OF INSERT AS

BEGIN

SET NOCOUNT ON; --to avoid the rowcount messages

SET ROWCOUNT 0; --in case the client has modified the rowcount

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

DECLARE @msg varchar(2000), --used to hold the error message

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

@rowsAffected int = (SELECT COUNT(\*) FROM inserted);

-- @rowsAffected int = (SELECT COUNT(\*) FROM deleted);

--no need to continue on if no rows affected

IF @rowsAffected = 0 RETURN;

BEGIN TRY

--[validation section]

--[modification section]

--<perform action> --this is all I change other than the name and

--table in the trigger declaration/heading

INSERT INTO Example.InsteadOfTriggerExample

(InsteadOfTriggerExampleId,FormatUpper,

RowCreatedTime,RowLastModifiedTime)

--uppercase the FormatUpper column, set the %time columns to

--system time

SELECT InsteadOfTriggerExampleId, UPPER(FormatUpper),

SYSDATETIME(),SYSDATETIME()

FROM inserted;

END TRY

BEGIN CATCH

IF @@trancount > 0

ROLLBACK TRANSACTION;

--[Error logging section]

DECLARE @ERROR\_NUMBER int = ERROR\_NUMBER(),

@ERROR\_PROCEDURE sysname = ERROR\_PROCEDURE(),

@ERROR\_MESSAGE varchar(4000) = ERROR\_MESSAGE();

EXEC ErrorHandling.ErrorLog$Insert

@ERROR\_NUMBER,@ERROR\_PROCEDURE,@ERROR\_MESSAGE;

THROW; --will halt the batch or be caught by the caller's catch block

END CATCH;

END;

Start out with a simple INSERT operation, referencing only two of the columns and not the row modification columns:

INSERT INTO Example.InsteadOfTriggerExample

(InsteadOfTriggerExampleId,FormatUpper)

VALUES (1,'not upper at all');

Now, checking the data:

SELECT FormatUpper, RowCreatedTime, RowLastModifiedTime

FROM Example.InsteadOfTriggerExample;

You can see that the FormatUpper value has been set to all uppercase, and the RowCreatedTime and RowLastModify time values have been set:

FormatUpper RowCreatedTime RowLastModifiedTime

------------------ --------------------------- -------------------------

NOT UPPER AT ALL 2020-08-02 17:23:58.037 2020-08-02 17:23:58.037

Now add two rows at a time to make sure that multi-row operations work:

INSERT INTO Example.InsteadOfTriggerExample

(InsteadOfTriggerExampleId,FormatUpper)

VALUES (2,'UPPER TO START'),(3,'UpPeRmOsT tOo!');

This will also be formatted as expected. Check the data:

FormatUpper RowCreatedTime RowLastModifiedTime

------------------ ------------------------- ---------------------------

NOT UPPER AT ALL 2020-08-02 17:23:58.037 2020-08-02 17:23:58.037

UPPER TO START 2020-08-02 17:25:45.385 2020-08-02 17:25:45.385

UPPERMOST TOO! 2020-08-02 17:25:45.385 2020-08-02 17:25:45.385

Now, check the error handler. Unlike AFTER triggers, you shouldn’t really expect any errors since almost any error you might check with an instead of trigger would generally be better served in a constraint. However, when you have an INSTEAD OF TRIGGER object, you do get the constraint errors being trapped by the trigger (thankfully, with the error re-throwing capabilities of THROW the error will be the native error that occurs):

--causes an error

INSERT INTO Example.InsteadOfTriggerExample

(InsteadOfTriggerExampleId,FormatUpper)

VALUES (4,NULL) ;

This returns the following error, which you can see claims to be coming from the InsteadOfTriggerExample$InsteadOfTrigger, line 23 in case you need to debug:

Msg 515, Level 16, State 2, Procedure InsteadOfTriggerExample$InsteadOfInsertTrigger, Line 23  
Cannot insert the value NULL into column 'FormatUpper', table 'AppendixB.Example.InsteadOfTriggerExample'; column does not allow nulls. INSERT fails.

Finally, we write the UPDATE version of the trigger. In this case, the UPDATE statement again forces the values of the RowLastModifiedTime column to ignore whatever is passed in, and ensures that the RowCreatedTime never changes.

CREATE TRIGGER Example.InsteadOfTriggerExample$InsteadOfUpdateTrigger

ON Example.InsteadOfTriggerExample

INSTEAD OF UPDATE AS

BEGIN

SET NOCOUNT ON; --to avoid the rowcount messages

SET ROWCOUNT 0; --in case the client has modified the rowcount

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

DECLARE @msg varchar(2000), --used to hold the error message

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

@rowsAffected int = (SELECT COUNT(\*) FROM inserted);

-- @rowsAffected int = (SELECT COUNT(\*) FROM deleted);

--no need to continue on if no rows affected

IF @rowsAffected = 0 RETURN;

BEGIN TRY

--[validation section]

--[modification section]

--<perform action>

--note, this trigger assumes non-editable keys. Consider

--adding an non-editable surrogate key (even non-pk)

--if you need to be able to modify the primary key values

--as you need something to correlate inserted rows to

--existing or deleted rows.

UPDATE InsteadOfTriggerExample

SET FormatUpper = UPPER(inserted.FormatUpper),

--RowCreatedTime, Leave this value out

RowLastModifiedTime = SYSDATETIME() --force this value

FROM inserted

JOIN Example.InsteadOfTriggerExample

ON inserted.InsteadOfTriggerExampleId =

InsteadOfTriggerExample.InsteadOfTriggerExampleId;

END TRY

BEGIN CATCH

IF @@trancount > 0

ROLLBACK TRANSACTION;

--[Error logging section]

DECLARE @ERROR\_NUMBER int = ERROR\_NUMBER(),

@ERROR\_PROCEDURE sysname = ERROR\_PROCEDURE(),

@ERROR\_MESSAGE varchar(4000) = ERROR\_MESSAGE();

EXEC ErrorHandling.ErrorLog$Insert

@ERROR\_NUMBER,@ERROR\_PROCEDURE,@ERROR\_MESSAGE;

THROW; --will halt the batch or be caught by the caller's catch block

END CATCH;

END;

Simple, really; just formatting data. Now update our data, two rows set to final test, and not that I set the modify times in the UPDATE statement, though they won’t be honored in the final results:

UPDATE Example.InsteadOfTriggerExample

SET RowCreatedTime = '1900-01-01',

RowLastModifiedTime = '1900-01-01',

FormatUpper = 'final test'

WHERE InsteadOfTriggerExampleId in (1,2);

Now check the data:

FormatUpper RowCreatedTime RowLastModifiedTime

------------------ ------------------------- ---------------------------

FINAL TEST 2020-08-02 17:23:58.037 2020-08-02 17:30:26.814

FINAL TEST 2020-08-02 17:25:45.385 2020-08-02 17:30:26.814

UPPERMOST TOO! 2020-08-02 17:25:45.385 2020-08-02 17:25:45.385

The RowCreatedTime values are the same as they were, and the RowLastModifiedTime values for the two rows are set to the time when the rows were updated (at the time I wrote this example).

It’s important to note that if you use a column with the identity property for a surrogate key, using an INSTEAD OF TRIGGER makes the SCOPE\_IDENTITY() function cease to work because the actual modification statement is not in the same scope. For example, take the following small table:

CREATE TABLE Example.TestIdentity

(

TestIdentityId int IDENTITY CONSTRAINT PKestIdentity PRIMARY KEY,

Value varchar(30) CONSTRAINT AKtestIdentity UNIQUE,

);

Without an instead of trigger, you can do the following:

INSERT INTO Example.TestIdentity(Value)

VALUES ('without trigger');

SELECT SCOPE\_IDENTITY() AS NewTestIdentityId;

And this will return:

NewTestIdentityId  
-------------  
1

But add a trigger such as the following (which does nothing but insert the data as-is, for the example):

CREATE TRIGGER TestIdentity$InsteadOfInsertTrigger

ON Example.TestIdentity

INSTEAD OF INSERT AS

BEGIN

SET NOCOUNT ON; --to avoid the rowcount messages

SET ROWCOUNT 0; --in case the client has modified the rowcount

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

DECLARE @msg varchar(2000), --used to hold the error message

--use inserted for insert or update trigger, deleted for update or

--delete trigger count instead of @@ROWCOUNT due to merge behavior that

--sets @@ROWCOUNT to a number that is equal to number of merged rows,

--not rows being checked in trigger

@rowsAffected int = (SELECT COUNT(\*) FROM inserted);

-- @rowsAffected int = (SELECT COUNT(\*) FROM deleted);

--no need to continue on if no rows affected

IF @rowsAffected = 0 RETURN;

BEGIN TRY

--[validation section]

--[modification section]

--<perform action>

INSERT INTO TestIdentity(Value)

SELECT Value

FROM inserted;

END TRY

BEGIN CATCH

IF @@trancount > 0

ROLLBACK TRANSACTION;

--[Error logging section]

DECLARE @ERROR\_NUMBER int = ERROR\_NUMBER(),

@ERROR\_PROCEDURE sysname = ERROR\_PROCEDURE(),

@ERROR\_MESSAGE varchar(4000) = ERROR\_MESSAGE();

EXEC ErrorHandling.ErrorLog$Insert

@ERROR\_NUMBER,@ERROR\_PROCEDURE,@ERROR\_MESSAGE;

THROW; --will halt the batch or be caught by the caller's catch block

END CATCH;

END;

And you will see that running very similar code:

INSERT INTO Example.TestIdentity(Value)

VALUES ('with trigger');

SELECT SCOPE\_IDENTITY() AS NewTestIdentityId;

Results in a NULL value:

NewTestIdentityId   
-----------------  
NULL

My typical solution is to use my knowledge of the data structures to simply use the natural key (something you should already have in the INSERT statement, generally):

INSERT INTO Example.TestIdentity(Value)

VALUES ('with trigger two');

SELECT testIdentityId AS NewTestIdentityId

FROM Example.TestIdentity

WHERE value = 'with trigger two'; --use an alternate key

And this returns:

NewTestIdentityId  
-------------  
3

Alternatively, use a SEQUENCE object-based key (see Chapter 7), which can be defaulted, or the caller can fetch the next value and pass it into the insert. In any case, the problem is often that the tool the developer uses expects the SCOPE\_IDENTITY function to work, so it can obviate the ability to use an instead of insert trigger.

# Triggers On Memory Optimized Tables

In this section, we are going to focus just briefly on natively compiled triggers by replicating the two triggers we have already created. At a high level, not a lot will change. The template is the same, the trigger types are the same, and the purpose is the same. However, the limitations of natively compiled objects is going to make our job a lot different.

I will demonstrate a natively compiled trigger by translating the AFTER TRIGGER example that we did earlier in this appendix. (For an example of an INSTEAD OF TRIGGER on a memory optimized table, see the in the downloads check the file: “Chapter 07\Chapter 7 - Database Create Memory Optimized.sql”.)

Translating this code was quite complex, because there are a bunch of limitations that you have to work with. The ones that came up in my translation of the AFTER TRIGGER from this appendix:

* No use of SET commands, so SET NOCOUNT and SET ROWCOUNT removed
* No way to rollback the transaction before the end of the object, so no error handler needed.
* Can’t use COALESCE or CONCAT
* Can’t use a subquery other than in a SELECT statement.
* No use of CTEs
* Natively compiled tables cannot be the target of a MERGE statement
* No use of BREAK in a WHILE loop
* Can’t use table variables unless they are pre-defined
* No use of the: UPDATE Table FROM … with a JOIN construct (Or DELETE either.)

Of the limitations, the biggest problem was UPDATE Table FROM. In order to get data from the inserted and deleted virtual tables in an UPDATE statement, we will end up using a loop. In the interpreted AFTER TRIGGER I built earlier, it used MERGE, and without UPDATE …FROM, things are a little tricky. It may seem clunky, and it definitely is. The coding you do is going to be much more like procedural coding than the declarative programming style that is favored by anyone with a modicum of SQL skills. However, because the code is natively compiled, and you will be working with small sets of data, it should not be a big deal.

Just like for the interpreted triggers, I am including a template that I work from. It has a few things added, and few removed, notably the SET options, and the error handling stuff. You could enhance the error hander to tell you what statement is being executed, the big thing you can’t do is anything that will be saved if you raise an error:

CREATE TRIGGER <schema>.<tablename>$<actions>[<purpose>]Trigger

ON <schema>.<tablename>

WITH NATIVE\_COMPILATION, SCHEMABINDING

<AFTER or INSTEAD OF> <comma delimited actions> AS

BEGIN ATOMIC WITH

(TRANSACTION ISOLATION LEVEL = SNAPSHOT, LANGUAGE = N'us\_english')

--use inserted for insert or update trigger, deleted for update

--or delete trigger count instead of @@ROWCOUNT due to merge behavior

--that sets @@ROWCOUNT to a number that is equal to number of merged

--rows, not rows being checked in trigger

DECLARE @msg varchar(2000), --used to hold the error message

--Natively compiled objects can't be the target of a MERGE currently,

--so could use @@ROWCOUNT, but this is safer for the future

@rowsAffected int = (SELECT COUNT(\*) FROM inserted);

-- @rowsAffected int = (SELECT COUNT(\*) FROM deleted);

--no need to continue on if no rows affected

IF @rowsAffected = 0 RETURN;

BEGIN TRY

--[validation section]

--[modification section]

--[perform action] --INSTEAD OF ONLY

END TRY

BEGIN CATCH

--will halt the batch or be caught by the caller's catch block

THROW;

END CATCH;

END;

We start the example by setting our database to allow memory optimized objects. Change the directories to whatever works in your environment.

ALTER DATABASE AppendixB ADD FILEGROUP AppendixB\_MemoryOptimized   
 CONTAINS MEMORY\_OPTIMIZED\_DATA;

GO

ALTER DATABASE AppendixB ADD FILE (

NAME = N'v\_MemoryOptimized',   
 FILENAME = N'C:\temp\AppendixB\_MemoryOptimized'

) TO FILEGROUP AppendixB\_MemoryOptimized;

To keep things simple, I am going to mirror the AfterTriggerExample table exactly, other than placing everything in a schema named Example\_InMem.

CREATE SCHEMA Example\_InMem;

Next, I create the two tables for our example, one for the value rows, the other for the summary data:

--this is the “transaction” table

CREATE TABLE Example\_InMem.AfterTriggerExample

(

AfterTriggerExampleId int

CONSTRAINT PKAfterTriggerExample PRIMARY KEY NONCLUSTERED,

GroupingValue varchar(10) NOT NULL,

Value int NOT NULL

)WITH (MEMORY\_OPTIMIZED = ON, DURABILITY = SCHEMA\_AND\_DATA);

GO

--this is the table that holds the summary data

CREATE TABLE Example\_InMem.AfterTriggerExampleGroupBalance

(

GroupingValue varchar(10) NOT NULL

CONSTRAINT PKAfterTriggerExampleGroupBalance

PRIMARY KEY NONCLUSTERED,

Balance int NOT NULL

)WITH (MEMORY\_OPTIMIZED = ON, DURABILITY = SCHEMA\_AND\_DATA);

The table is pretty simple, and I have simply used Bw-Tree indexes for the two PRIMARY KEY constraints. I didn’t include a FOREIGN KEY constraint just to keep it a bit simpler. The next thing we need is a table type that we will use to hold intermediate results to loop though. Each of the triggers you need to create may need to set the new balance for more than one grouping value.

CREATE TYPE Example\_InMem.AfterTriggerExampleIntermediateSet AS TABLE(

GroupingValue varchar(10) NULL,

NewBalance int NULL,

INDEX TT\_AfterTriggerExampleIntermediateSet NONCLUSTERED

(

GroupingValue,

NewBalance

)

)

WITH ( MEMORY\_OPTIMIZED = ON );

Next, I code the following AFTER TRIGGER object. I have commented the code to help you see the basics of what is done. It is extremely complex in comparison to the interpreted trigger:

CREATE TRIGGER Example\_InMem.AfterTriggerExample$InsertTrigger

ON Example\_InMem.AfterTriggerExample

WITH NATIVE\_COMPILATION, SCHEMABINDING

AFTER INSERT

AS

BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = REPEATABLE READ,

LANGUAGE = N'us\_english')

--use inserted for insert or update trigger, deleted for update

--or delete trigger count instead of @@ROWCOUNT due to merge behavior

--that sets @@ROWCOUNT to a number that is equal to number of merged

--rows, not rows being checked in trigger

DECLARE @msg varchar(2000), --used to hold the error message

--Natively compiled objects can't be the target of a MERGE currently,

--so could use @@ROWCOUNT, but this is safer for the future

@rowsAffected int = (SELECT COUNT(\*) FROM inserted);

-- @rowsAffected int = (SELECT COUNT(\*) FROM deleted);

--no need to continue on if no rows affected

IF @rowsAffected = 0 RETURN;

BEGIN TRY

--[validation section]

--Use a WHERE EXISTS to inserted to make sure not to duplicate

--rows in the set if > 1 row is modified for the same grouping value

DECLARE @Exists BIT --can't use subquery, so use this to set

--a value if a row exists

SELECT @Exists = 1

FROM Example\_InMem.AfterTriggerExample

WHERE EXISTS (SELECT 1 --select \* prohibited

FROM Inserted

WHERE AfterTriggerExample.GroupingValue =

Inserted.Groupingvalue)

GROUP BY AfterTriggerExample.GroupingValue

HAVING SUM(Value) < 0

IF @Exists = 1 --replaces the subquery

BEGIN

IF @rowsAffected = 1

SELECT @msg = 'Grouping Value "' + GroupingValue +

'" balance value after operation must be greater than 0'

FROM inserted;

ELSE

SELECT @msg = 'The total for the grouping value must ' +

'be greater than 0';

THROW 50000, @msg, 16;

END;

--[modification section]

--get the balance for any Grouping Values used in the DML

--statement temp table to loop through

DECLARE @GroupBalance

Example\_InMem.AfterTriggerExampleIntermediateSet

--set up the set of balances to modify

INSERT INTO @GroupBalance (GroupingValue, NewBalance)

SELECT ChangedRows.GroupingValue, SUM(Value) AS NewBalance

FROM ( --Only one row per grouping set

SELECT DISTINCT GroupingValue

FROM Inserted

) as ChangedRows

JOIN Example\_InMem.AfterTriggerExample

ON AfterTriggerExample.GroupingValue =

ChangedRows.GroupingValue

GROUP BY ChangedRows.GroupingValue;

DECLARE @GroupingValue varchar(10),

@NewBalance int, @ContinueLoop int = 1;

-- Prehistoric cursor, grab a value from temp table, use it,

-- delete it from temp table, get another.

-- Can’t use BREAK so we start by fetching one value first

SELECT TOP(1) @GroupingValue = GroupingValue,

@NewBalance = NewBalance

FROM @GroupBalance;

WHILE (@ContinueLoop=1)

BEGIN

--update the row if it exists, if it doesn’t the @@ROWCOUNT   
 --will signal insert

UPDATE Example\_InMem.AfterTriggerExampleGroupBalance

SET Balance = @NewBalance

WHERE @GroupingValue = GroupingValue

AND @NewBalance IS NOT NULL;

IF @@ROWCOUNT = 0

INSERT INTO Example\_InMem.AfterTriggerExampleGroupBalance   
 (GroupingValue, Balance)

VALUES (@GroupingValue, @NewBalance);

--Manage loop variables

DELETE FROM @GroupBalance

WHERE @GroupingValue = GroupingValue;

SELECT TOP(1) @GroupingValue = GroupingValue,

@NewBalance = NewBalance

FROM @GroupBalance;

IF @@ROWCOUNT = 0 SET @ContinueLoop = 0;

END

END TRY

BEGIN CATCH

THROW; --will halt the batch or be caught by the caller's catch block

END CATCH

END;

I won’t go through the test steps again; the code is in the download. But it is clear things are quite a bit more complex. This took me several more hours to hone (along with the UPDATE and DELETE than I had expected. I want to include the AFTER UPDATE TRIGGER as well, because it is the most complex since now, we have to worry with groups possibly going away. The DELETE trigger, along with the code to test it, will be in the download.

Here is the UPDATE trigger, again with comments to help clarify what is going on in ways that is different than the INSERT trigger:

CREATE TRIGGER [Example\_InMem].[AfterTriggerExample$UpdateTrigger]

ON [Example\_InMem].[AfterTriggerExample]

WITH NATIVE\_COMPILATION, SCHEMABINDING

AFTER UPDATE

AS

BEGIN ATOMIC WITH (TRANSACTION ISOLATION LEVEL = REPEATABLE READ, LANGUAGE = N'us\_english')

--use inserted for insert or update trigger, deleted for update

--or delete trigger count instead of @@ROWCOUNT due to merge behavior

--that sets @@ROWCOUNT to a number that is equal to number of merged

--rows, not rows being checked in trigger

DECLARE @msg varchar(2000), --used to hold the error message

--Natively compiled objects can't be the target of a MERGE,

--so could use @@ROWCOUNT

@rowsAffected int = (SELECT COUNT(\*) FROM inserted);

-- @rowsAffected int = (SELECT COUNT(\*) FROM deleted);

--no need to continue on if no rows affected

IF @rowsAffected = 0 RETURN;

BEGIN TRY

--[validation section]

--Use a WHERE EXISTS to inserted to make sure not to duplicate

--rows in the set if > 1 row is modified for the same grouping value

DECLARE @Exists BIT

SELECT @Exists = 1

FROM Example\_InMem.AfterTriggerExample

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

--added UNION ALL and the deleted part

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

WHERE EXISTS (SELECT 1

FROM Inserted

WHERE AfterTriggerExample.GroupingValue =

Inserted.GroupingValue

UNION ALL

SELECT 1

FROM Deleted

WHERE AfterTriggerExample.GroupingValue =

Deleted.Groupingvalue)

GROUP BY AfterTriggerExample.GroupingValue

HAVING SUM(Value) < 0;

IF @Exists = 1

BEGIN

IF @rowsAffected = 1

SELECT @msg = 'Grouping Value "' + GroupingValue +

'" balance value after operation must be greater than 0'

FROM inserted;

ELSE

SELECT @msg = 'The total for the grouping value must ' +

'be greater than 0';

THROW 50000, @msg, 16;

END;

--[modification section]

--get the balance for any Grouping Values used in the DML statement

DECLARE @GroupBalance

Example\_InMem.AfterTriggerExampleIntermediateSet;

INSERT INTO @GroupBalance (GroupingValue, NewBalance)

SELECT ChangedRows.GroupingValue, SUM(Value) AS NewBalance

FROM (SELECT GroupingValue --now also need deleted rows

FROM Inserted

UNION --need distinct values, from inserted and deleted

SELECT GroupingValue

FROM Deleted

) as ChangedRows

--left outer join, because the primary key could change,

--looking like a delete of all rows

LEFT OUTER JOIN Example\_InMem.AfterTriggerExample

ON AfterTriggerExample.GroupingValue =

ChangedRows.GroupingValue

GROUP BY ChangedRows.GroupingValue;

DECLARE @GroupingValue varchar(10), @NewBalance int,

@ContinueLoop int = 1;

SELECT TOP(1) @GroupingValue = GroupingValue,

@NewBalance = NewBalance  
 FROM @GroupBalance;

WHILE (@ContinueLoop=1 )

BEGIN

UPDATE Example\_InMem.AfterTriggerExampleGroupBalance

SET Balance = @NewBalance

WHERE @GroupingValue = GroupingValue

--If no rows for GroupingValue, balance NULL

AND @NewBalance IS NOT NULL;

--null balance is a delete

IF @@ROWCOUNT = 0 and @NewBalance IS NOT NULL

INSERT INTO

Example\_InMem.AfterTriggerExampleGroupBalance   
 (GroupingValue, Balance)

VALUES (@GroupingValue, @NewBalance);

--this means no rows for grouping, then delete

IF @NewBalance IS NULL

DELETE FROM

Example\_InMem.AfterTriggerExampleGroupBalance

WHERE @GroupingValue = GroupingValue;

--Manage loop variables

DELETE FROM @GroupBalance

WHERE @GroupingValue = GroupingValue;

SELECT TOP(1) @GroupingValue = GroupingValue,

@NewBalance = NewBalance

FROM @GroupBalance;

IF @@ROWCOUNT = 0 SET @ContinueLoop = 0;

END;

END TRY

BEGIN CATCH

THROW; --will halt the batch or be caught by the caller's catch block

END CATCH

END;

Definitely not the easiest code to read (or write!) but will execute fast enough for typical uses of memory optimized tables.