**Triggers**

A trigger is an object that is compiled and stored in the database. It is associated with a SPECIFIC table for a SPECIFIC DML statement. If we create a trigger that is associated with INSERT operation on the STAFF table, the code in the trigger will execute every time an insert statement is executed on the staff table. They cannot be explicitly executed and only execute in response to a SPECIFIC DML statement executing on a SPECIFIC table.

The code inside a trigger may look very much like code in a Stored Procedure however unlike a Stored Procedure a trigger cannot accept parameters.

Stored Procedures can be use to:

1. Enforce referential integrity across databases. This concept is similar to a Foreign Key constraint but the two related tables are in different databases. Since a Foreign Key cannot reference a table outside the database it is in you must use a trigger for this purpose.
2. Enforce business rules that are too complex for a check constraint.
3. Automate an operation (logging, backup, archiving)
4. Create an audit trail (a record of the change history of data in a database)

Refer to the “DML STATEMENT LOGIC WITH TRIGGERS” handout to get an understanding how the server processes a DML statement.

By looking at this handout we see the following:

* If the DML operation (update) violates a constraint or a data type requirement the operation fails and the server issues an error message. The trigger does not execute.
* The server initiates a transaction as part of executing the DML operation.
* Two temporary tables are created and used by the server when executing the DML operation:
* Deleted: contains the before image of all rows affected by the DML operation.
* Inserted: contains the after image of all rows affected by the DML operation.
* The trigger executes after the DML operation has been done but before the transaction is committed (making the operation permanent). So, still able to perform a rollback.
* The trigger can use the Inserted and Deleted tables.
* The trigger can rollback the transaction and issue an error message if it deems the operation to be invalid. This prevents the changes from being saved to the table in the database.
* If the trigger does not issue a rollback, the DBMS will commit the transaction thus making the operation permanent.

**A trigger never begins or commits a transaction!**

The term base table refers to the table that the trigger is associated with. The contents of the three tables involves in a DML operation (Inserted, Deleted and the base table) when a trigger starts to execute is described below:

|  |  |  |  |
| --- | --- | --- | --- |
| DML Operation | Inserted | Deleted | Base |
|  |  |  |  |
| Insert | After image of new rows. | Empty | New rows and all previously existing rows. |
| Delete | Empty | Before image of deleted rows | All rows that were not deleted. |
| Update | After image of changed rows. | Before image of changed rows. | After image of changed rows and all other rows not affected by the operation. |

A trigger can execute under any of three fundamental conditions:

1. When the DML operation affects zero rows in the base table.
2. When the DML operation affects one row in the base table.
3. When the DML operation affects many rows in the base table.

The logic of a trigger must work under all three conditions.

**DML STATEMENT LOGIC WITH TRIGGERS**

Use the CREATE TRIGGER statement to create a trigger object.

Basic Syntax:

CREATE TRIGGER trigger\_name

ON table\_name

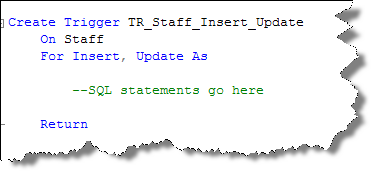
FOR {[update] [,] [insert] [,] [delete]} AS

SQL statements

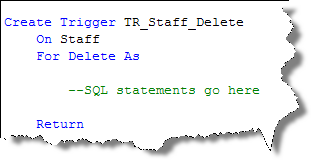
RETURN

Example:

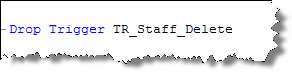
To create a trigger for the Insert and Update operations and associate it with the Staff table:



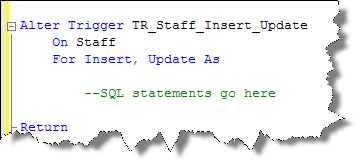
To create a trigger for the Delete operation and associate it with the Student table:



To drop a trigger you use DROP TRIGGER:



To alter a trigger you use ALTER TRIGGER in the same way you use ALTER PROCEDURE:



If you drop a table that has triggers associated with it, the triggers are dropped as well. A trigger cannot exist without the table it is associated with.

To see a list of triggers in your data base:

Select Name From Sysobjects Where Type = 'TR'

You can also use sp\_helptext to retrieve the source code of your trigger

sp\_helptext tr\_student\_update

**Activity**

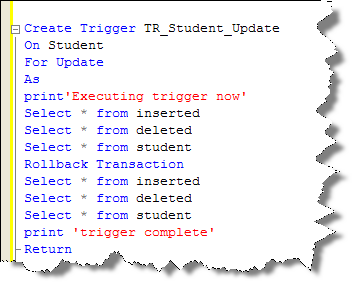
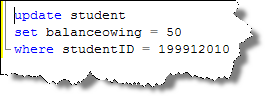
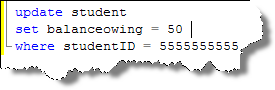
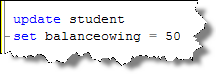
To become familiar with the inserted and delete table and the affect of rolling back transactions from a trigger, wrote a script that will create a trigger for use with an UPDATE to the students BALANCE on the STUDENT table. The trigger will:

* issue an informational message(print) to indicate the trigger is executing
* display the contents of the inserted table
* display the contents of the deleted table
* display the contents of the base table (student)
* issue a ROLLBACK TRANSACTION
* display the contents of the inserted table after the rollback
* display the contents of the deleted table after the rollback
* display the contents of the base table (Student) after the rollback
* issue an informational message(print) to indicate completion of the trigger

To test the trigger:

* issue an update to the BalanceOwing column that violates the data type of the column being updated
* issue a valid update to the character wages that affects one row
* issue a valid update to the character wages that affects multiple rows
* issue a valid update to the character wages that affects zero rows

***Does the output of the trigger make sense for each test condition?***



* Examine the contents of inserted and deleted
* Did the rollback execute?
* Did it work for each test condition?
* Note that we will be enhancing our triggers beyond the syntax in this example!

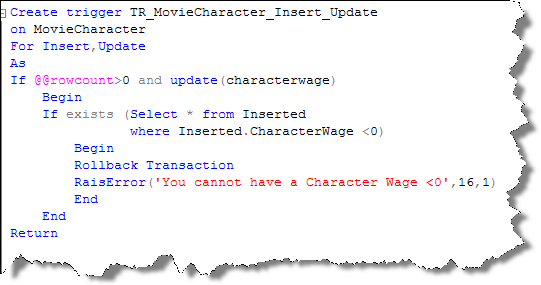
To explore some common uses of triggers we will use the MovieCharacter database. Run the MovieCharacter.sql script to create the tables and insert a few records.



\*AgentID and CharacterID are both identity (1,1) fields

**Create a trigger that enforces a rule that a CharactersWage must be >=0.**

NOTE: This simple business rule example can be done with a simple check constraint as well.



**@@RowCount ­**

Number of rows affected by the DML statement that caused the trigger to execute. If the insert or update did not affect any rows there is no need to evaluate the business rule so we branch around that logic in the trigger.

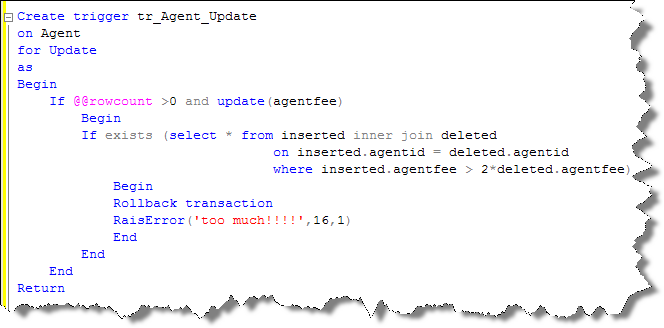
**Update (characterwage)**

The update function takes a column name as a parameter. If that column is being updated in the update that caused the trigger to execute then the function returns TRUE. Therefore, if we are not updating the characterwage column the update function returns false and there is no need to evaluate the business rule, since it only involves the characterwage column, so we branch around that logic in the trigger. When you insert new record the update function always returns true since you are updating the characterwage column from ‘nothing’ to ‘something’.

**Why use exists?**

The basic logic is we check for the existence of one or more records that break the business rule in the records being affected by the DML statement. To check only the records that we are actually Inserting/Updating , we check the records in the inserted table. If any of those records break the business rule the *if exists* statement is TRUE and we rollback the transaction to reverse the changes to the database and Raise an Error letting the user know the DML statement failed.

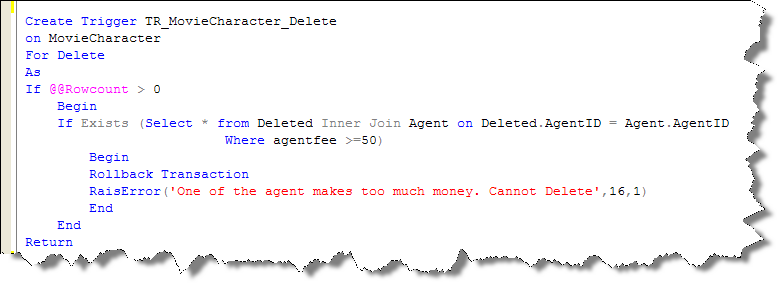
**Create a trigger that enforces a rule that an AgentsFee cannot be increased by more than double in one update.**

****

@@rowcount and update() serve the same purpose as the characterwages >0= example. Branching around the business rule check if the update statement affects 0 rows or the update statement was not updating the agentfee column.

What is new is the use of a join between the inserted and deleted tables. What purpose do you think that serves? Think about the business rule we are trying to enforce. It is a comparison between the original agentfee and the updated agentfee. Remember as well that the update could update more than one record and if any of the records updated violate the business rule they all should be rolled back. By creating a join between the inserted and deleted tables we are guarantee that we will compare the correct original value with the correct updated value . It also compares only the records that are in the inserted and deleted tables (ones that are being updated) and not the other records in the table that are not being updated.

**Create a trigger that enforces a rule that a MovieCharacter cannot be deleted if their Agents Agentfee is >=50.**



**@@Rowcount**

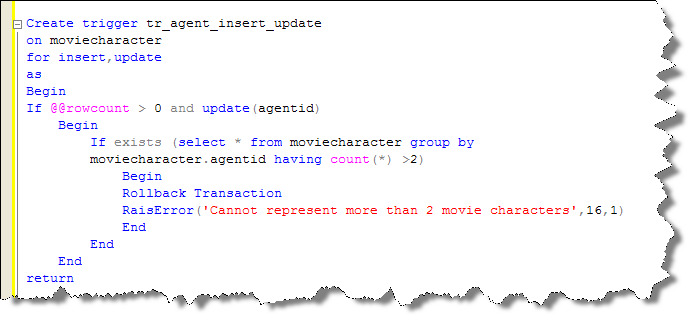
Did the delete that caused the trigger to execute delete any rows? If not then no need to check the business rule.

**Exists**

Check to see if the records being deleted have an AgentFee >=50. The join between the Deleted table and the Agent table only checks the records (agents) that have their related moviecharacters being deleted. If any exists that violate the business rule Exists is true and all the records that were deleted are rolled back.

Note that we cannot pass in a parameter identifying the record(s) we need to check. This is one of the differences between triggers and stored procedures. We often perform inner joins to the inserted or deleted tables to identify the records we need to evaluate in the trigger.

**Create a trigger that enforces a rule that an Agent cannot represent more than 2 movie characters.**



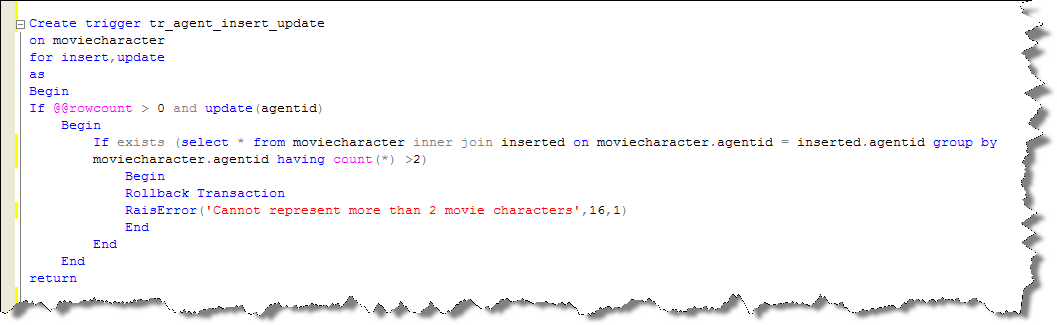
@@rowcount and update() are used to branch around the business rule check if no rows were actually changed or the agendid fields was not changed.

Exists checks if any agent has a count of related moviecharacter records >2. Remember, by the time the trigger has executed the DML statement has affected the base table (moviecharacter). This means we are actually counting the number moviecharacters per agent in the base table.

Now, while there is a small issue in the preceding example. Which agents are being checked to see if they violate the business rule being checked in the trigger? They all are. A trigger should check records that are being updated/inserted/deleted but not check records that are already in the database to see if they violate the business rule in the trigger. In other words, your trigger should only enforce the business rule from the time the trigger is created on. If an agent already exists with >2 movie characters then that is fine. Don’t go back in time!

So, how can we adjust the previous example to only check the agents that are being affected by the Insert/Update statement that caused the trigger to execute?

Perform an inner join to the inserted table to only check the records that are in both the inserted table (records being updated/inserted) and the moviecharacter table. The records that are not being affected by the insert/update will not be checked against the business rule.



We can use triggers to create a record of changes to other records. This is often called Logging and involves entering a record in a logging table when changes are made to another table. This example will demonstrate using a trigger in this respect.

**Create a Trigger to Log when changes are made to the coursecost in the course table. The changes will be inserted in to the following Logging Table. LogID will be an identity(1,1) column.**



First we need to create the Course Changes table!

Create Table CourseChanges

(

LogID int identity(1,1) not null constraint pk\_CourseChanges Primary Key Clustered,

ChangeDate datetime not null,

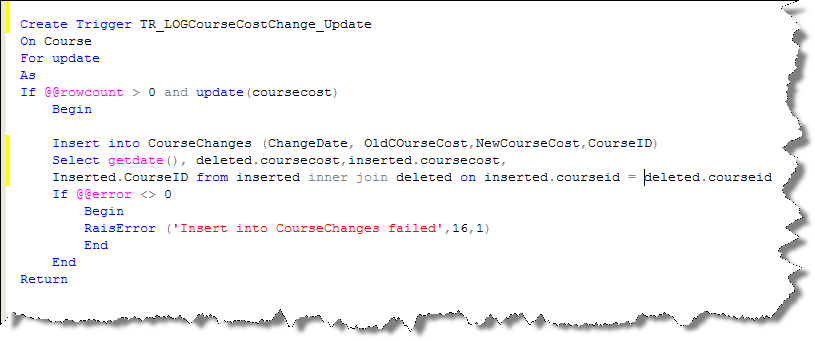
OldCourseCost money not null,

NewCourseCost money not null,

CourseID char(7) not null

)

Then we create the trigger.



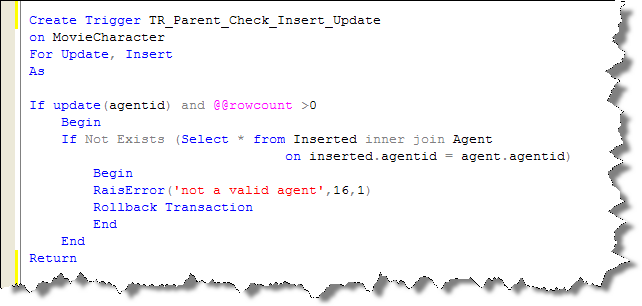
Not too much logic here. Just checking to see if the coursecost was being updated and that at least one record changed. Then inserting the correct record into the CourseChanges table to log the changes. Note the subquery used in the insert statement to insert the correct columns from the inserted and deleted tables into CourseChanges. The inner join is required to relate the correct old record(deleted) with the correct new record(inserted).

We can also use a trigger to enforce referential integrity (Foreign Key Constraints) instead of Foreign Keys if we wish. We would normally only do this if we need to ensure a value in a table has a matching “parent” value in another table that is in another database. A foreign key constraint can only enforce that a foreign key field has a matching value in its parent table from the same database. So, if we needed to ensure a value has a matching value in a table from a different database we can use a trigger.

However, for the scope of this course we will demonstrate it by replacing a foreign key constraint with a trigger checking referential integrity in the same database.

To do this we will need to use ALTER TABLE to disable the foreign key constraint in the moviecharacter table. We will replace it with a trigger to enforce that the AgentID in the MovieCharacter table has a matching AgentID in the Agent table.

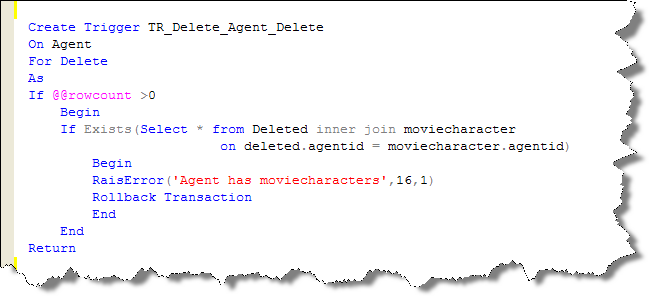
**Create a trigger to enforce referential integrity between the Agent and MovieCharacter table.**



The logic of this trigger checks to see if the AgentID trying to be inserted in the MovieCharacter table does NOT exists in the Agent table (parent table). If it does not that means the agented trying to be inserted/updated in the MovieCharacter (child) table does not have a matching value in the Agent (parent) table. Again, the inner join only returns records where there is a match so if there is no record returned there is no matching parent record.

**WAIT! This trigger will only work if only one record is affected! Why is that? What must you change in this trigger to work if many records are affected and to rollback if any one of them is not a valid agent ID? The correct solution will be gone over in class ☺**

Let’s check the referential integrity from the other side. We cannot delete a parent record if there are child records so that means that we cannot delete an Agent record if that agent has related MovieCharacter records. Again, this would normally be enforced by the foreign key constraint but we will code a constraint to enforce referential integrity as a trigger.



The logic of the trigger checks to see if a record with the same AgentID as the one being deleted exists in the MovieCharacter (child). If it does you cannot delete the Agent record.

**So, when to use the @@rowcount and update()????**

There is a pattern here…..Honestly!

Whenever you execute any DML statement the potential exists that it may affect 0 records. Therefore, you will always check @@rowcount so you can branch around the logic of the trigger if there was no change to the database.

Whenever you execute an update you may not be updating the column the that the trigger is concerned with. If that is the case you want to branch around the logic of the trigger that does not involve that column.

Remember that when you insert a record that you are actually updating data (from nonexistent to being there) so update() will always return true for an insert.

That’s it!