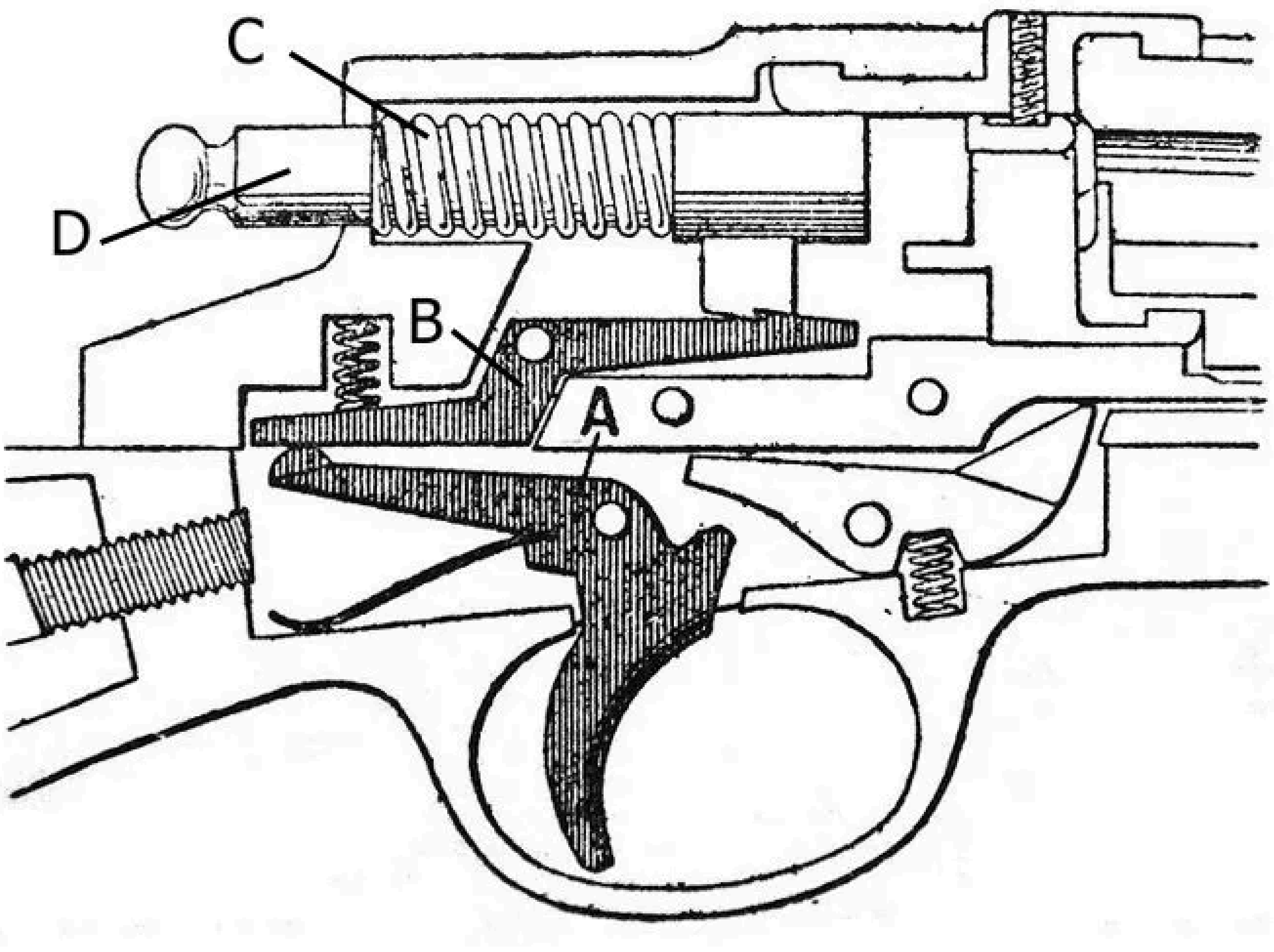
TRIGGERS Summary

In SQL, **Triggers** can be thought of as specialized stored procedures that are only invoked by the database in response to individual INSERT/UPDATE/DELETE statements. They have no parameters, cannot be explicitly invoked, and are tied to the database table on which they are created.



Uses of Triggers

Triggers have many uses, but the key ones are to:

1. **Create an audit trail (a record of the change history of data in a database).**

2. **Automate processes where changes to one table will cause changes to some other table(s).**
3. **Enforce business rules too complex to enforce with a `CHECK` constraint.**
4. **Prevent an Insert, Update, and/or Delete from happening on a table.**
5. Enforce table relationships when a foreign key constraint does not exist.
6. Enforce referential integrity across databases, or even servers.

Types of Triggers

There are two general categories of DML triggers

- **`FOR`** / **`AFTER`** - These triggers run after the database server has performed the INSERT/UPDATE/DELETE statement. These triggers have the ability to reverse (`ROLLBACK`) the DML statement.
- `INSTEAD OF` - (*We will **not** be using these in this course.*) These triggers are replacements for the standard INSERT/UPDATE/DELETE DML actions.

The Execution Context of Triggers

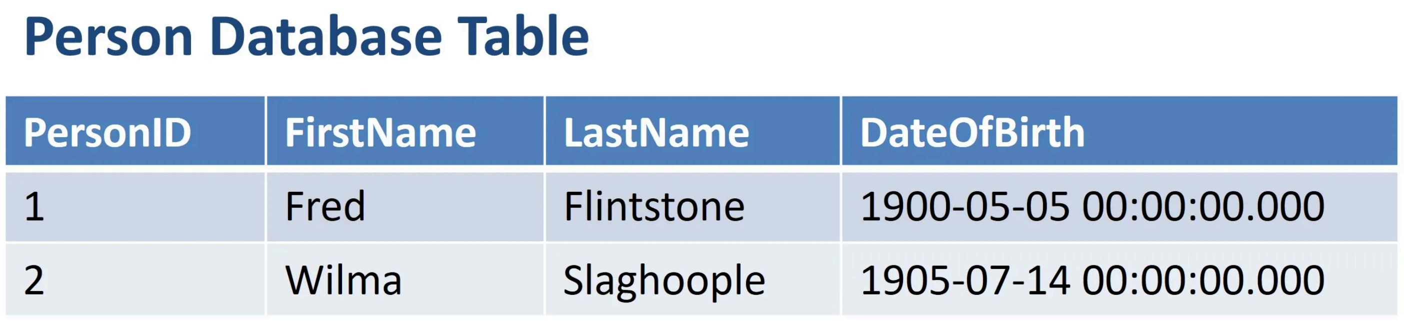
There are a number of facts to keep in mind regarding triggers in SQL.

- Each trigger is *attached* to a single table. Each table can have multiple triggers (any combination of INSERT, UPDATE, and DELETE).
- You *cannot* explicitly invoke (call) triggers; they are called by the DBMS in response to an INSERT/UPDATE/DELETE statement.
- Triggers *do not* have parameters.

When a trigger is executed, the database's table has **ALREADY** changed (within a transaction started by SQL Server). Also, SQL Server creates two temporary tables named "Inserted" and "Deleted"; these tables will have *exactly* the same column names/types as the trigger's table. The content of these two tables will depend on which DML statement is executed.

DML Operation	`inserted` Table Contents	`deleted` Table Contents	Trigger (Target) Table
`INSERT`	Newly inserted rows	Empty	New rows and all previously existing rows.
`DELETE`	Empty	Copy of deleted rows	All rows that were not deleted.
`UPDATE`	"After update" values of changed rows	"Before update" values of affected rows	After version of changed rows and all other rows not affected by the operation

Take, for example, the following **Person** table.



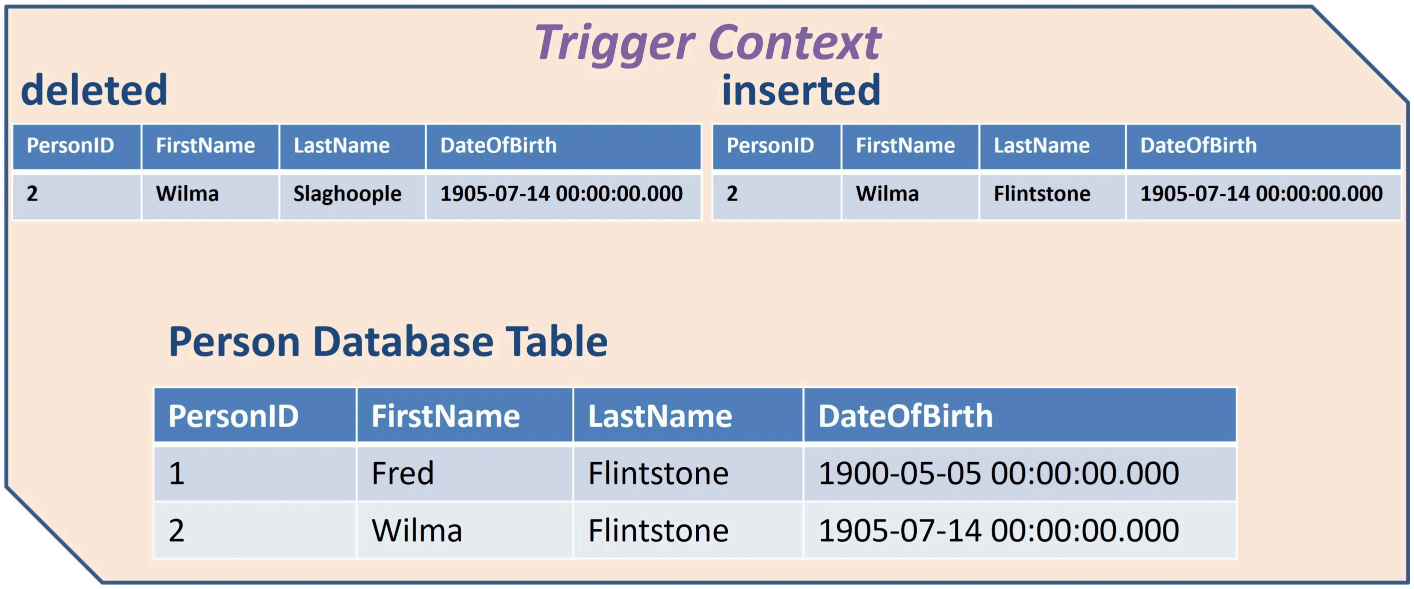
Person Database Table

PersonID	FirstName	LastName	DateOfBirth
1	Fred	Flintstone	1900-05-05 00:00:00.000
2	Wilma	Slaghoople	1905-07-14 00:00:00.000

Imagine that the following statement is issued against that table.

```
UPDATE Person SET LastName = 'Flintstone'
WHERE  FirstName = 'Wilma' AND LastName = 'Slaghoople'
```

If a trigger was added to that table for the `UPDATE` operation, then the trigger's context would have the following data in the deleted, inserted, and Person tables.



deleted

PersonID	FirstName	LastName	DateOfBirth
2	Wilma	Slaghoople	1905-07-14 00:00:00.000

inserted

PersonID	FirstName	LastName	DateOfBirth
2	Wilma	Flintstone	1905-07-14 00:00:00.000

Person Database Table

PersonID	FirstName	LastName	DateOfBirth
1	Fred	Flintstone	1900-05-05 00:00:00.000
2	Wilma	Flintstone	1905-07-14 00:00:00.000

Processing Inside a Trigger

Since it is quite possible that a INSERT, UPDATE, or DELETE statement will not affect any rows, it's important to account for that in the logic of your trigger.

- The number of rows affected by the current DML is in `@@Rowcount`
- The logic of the trigger **MUST** account for **0** rows OR affects **1** row OR affects **Many** rows.

It's also important to recognize that the trigger runs inside of a transaction that was created by the RDBMS when the DML statement was started. This means that you should **not** try to begin or commit any transactions. You are allowed, however, to `ROLLBACK` the transaction started by the RDBMS for DML statement. In fact, that's a common objective of trigger processing for those triggers that exist to enforce business rules or prevent DML actions on tables.

Another useful item when processing within a trigger is the **Update *Function***. This is not to be confused with the `UPDATE` DML statement. The **`UPDATE(column_name)`** function is a special function which can only be used in triggers, and its purpose is to determine if the value has changed for a specific column as a result of the DML statement. During an INSERT or DELETE statement, all the columns are modified for the affected row(s). But for an `UPDATE` statement, it is possible that only *some* of the column values have been changed. The **Update Function** is useful for testing to see if a particular column value has changed.

A Deeper Look at Triggers

Let’s revisit the Person table involving Fred and Wilma. But this time, we’ll have an extra column - `MaritalStatus` - to give us an excuse for writing a trigger. We’ll also include a couple more people.

Person Table

PersonID	MaritalStatus	FirstName	LastName	DateOfBirth
1	Single	Fred	Flintstone	2000-05-05 00:00:00.000
2	Single	Wilma	Slaghoople	2005-07-14 00:00:00.000
3	Single	Barney	Rubble	2001-03-19 00:00:00.000
4	Single	Betty	McBricker	2002-05-25 00:00:00.000

Now, let’s craft an `UPDATE` statement so that Fred and Wilma can get married. We’ll throw in the added aspect of Wilma taking on Fred’s last name.

```sql
UPDATE Person
SET LastName = 'Flintstone',
    MaritalStatus = 'Married'
WHERE  (FirstName = 'Wilma' AND LastName = 'Slaghoople')
   OR  (FirstName = 'Fred' AND LastName = 'Flintstone')
```

Here’s where we make use of that `MaritalStatus` column we added. We can build a trigger that makes sure a person is single before changing their status to ‘Married’.

```sql
CREATE OR ALTER TRIGGER Person_MaritalStatusRule
ON Person
FOR UPDATE
AS
    IF  @@ROWCOUNT > 0 AND UPDATE('MaritalStatus')
        AND EXISTS(SELECT * FROM 'deleted' WHERE MaritalStatus = 'Married')
    BEGIN
        RAISERROR('Cannot change marital status because they are already married', 16, 1)
        ROLLBACK TRANSACTION
    END
```

Caveat

You might be thinking, "Hey, what if they are changing the status from 'Married' to 'Single'?" I'll leave that as an exercise for you to figure out how the `IF` statement in the trigger should be changed to accommodate that.

But how does the database server process the `UPDATE` statement internally? It goes through the following steps:

1. Start a **transaction** internally.

2. Copy the rows to be replaced *(updated)* to a table called `deleted`. This table will have the same structure as the `Person` table.

   deleted

PersonID	MaritalStatus	FirstName	LastName	DateOfBirth
1	Single	Fred	Flintstone	2000-05-05 00:00:00.000
2	Single	Wilma	Slaghoople	2005-07-14 00:00:00.000

3. Make the changes to the `Person` table

```sql
UPDATE Person
SET LastName = 'Flintstone',
    MaritalStatus = 'Married'
WHERE  (FirstName = 'Wilma' AND LastName = 'Slaghoople')
   OR  (FirstName = 'Fred' AND LastName = 'Flintstone')
```

   Person Table

PersonID	MaritalStatus	FirstName	LastName	DateOfBirth
1	***Married***	*Fred*	***Flintstone***	2000-05-05 00:00:00.000
2	***Married***	*Wilma*	***Flintstone***	* 2005-07-14 00:00:00.000*
3	Single	Barney	Rubble	2001-03-19 00:00:00.000
4	Single	Betty	McBricker	2002-05-25 00:00:00.000

4. Check for any errors (e.g.: `CHECK` constraint violations). If any are found, it will do a `ROLLBACK TRANSACTION` and return.

5. If no problems were found, the server will copy the rows that were changed to a table called `inserted`. Again, this table will have the same structure as the `Person` table.

   inserted

PersonID	MaritalStatus	FirstName	LastName	DateOfBirth
1	Married	Fred	Flintstone	2000-05-05 00:00:00.000
2	Married	Wilma	Flintstone	2005-07-14 00:00:00.000

6. Lastly, the server will invoke (or *`EXECUTE`*) any triggers that were attached to the `Person` table - in our case, the **Person_MaritalStatusRule**. That trigger will have access to the `inserted` and `deleted` tables.