

# Chapter 7

## Triggers

A trigger is an element of the database schema that has the following structure:

```
On event If precondition Then action
```

where `event` is a request for the execution of a particular database operation, e.g., add in a row in the table; `precondition` is a boolean expression and `action` is a statement of what needs done when this trigger is fired.

A concrete example could be that when a student registers for a course (event), if the class is already full (the associated condition), a message should be displayed and this insertion will be declined (action).

## Where a trigger is used?

1. It can be used to maintain constraints. The most common form of a trigger in this kind uses the `On Delete` and `On Update` clauses, which are attached to foreign key clauses, so that, e.g., when something is to be deleted, but some other rows are still referring to it, a trigger can be fired to prevent this deletion from happening.

2. A business rule is a concise statement about a basic principle underlying a business process.

For example, a trigger could state that if an international money transfer is made into a client's account, and it is not suspicious, then an Email message should be sent to notify the client.

3. complex physical objects are monitored by sensor networks, which will add in the measurement into a database. Since such insertion is a database event, we can use a trigger to take some action when needed.

For example, if too much  $CO_2$  is detected when the record is being inserted, a ventilation system can be turned on as a result.

4. A trigger can also be used to update a view, e.g., whenever a change is made to the base tables.

5. When exceptions need to be handled in data base applications, triggers are ideal.

For example, when divided by zero.

## Issues with triggers

1. There are quite a few types of triggers, which might be useful for different kinds of applications. Which one should be used?

2. At a certain point, several triggers are ready to be fired. Which one should be picked up first? and more generally, what is the order of the execution of those triggers?

3. Execution of one trigger might cause the execution of other triggers, sort of a chain of reaction. To prevent an infinite loop, DBMS has to put a limit on the length of such a chain.

When such a limit is reached, all the changes by the original update and those triggers will be rolled back.

## Trigger consideration

A trigger is *activated* when the triggering event is requested. The *consideration* of a trigger refers to *when* the precondition is checked after a trigger is activated.

If the precondition is checked out to be true, then the trigger should be fired, i.e., the associated action should be taken. But, moments later, this precondition could become false, maybe caused by the update made by this very requested event; or something else.

Thus, *when the precondition is checked* becomes an issue.

## An example

Consider the following trigger:

```
On inserting a row in CourseRegistration table  
If over course capacity  
Then abort registration transaction
```

When a student attempts to add a course in the registration table, the course might be full, when this insertion should be rejected.

But, it might be the case that a bit later, another student is withdrawing from this course, or the capacity is increased. Thus, if the condition is checked at a later time, this insertion might be doable.

It all depends....

Apparently, in this case, deferring the consideration of this update might be a good idea. Sort of putting it into a waiting list.

However, if the database is to monitor the pressure in a nuclear power plant and the trigger event is pressure increase while the precondition is that the pressure not to exceed a preset level, then an immediate check of the condition is definitely mandatory.

Hence, there are at least two approaches as when the precondition should be checked: immediate or deferred until the triggering transaction is *committed*, i.e., when everything is ready so that the triggering event has been executed.

## Trigger execution

If trigger consideration is deferred, its execution is necessarily deferred as well until the end of the triggering transaction.

On the other hand, if the precondition is immediately checked, we can either execute the trigger immediately after the check of the precondition; or we can defer execution of the trigger until the end of the triggering transaction.

It again depends on the application. For the nuclear power plant case, it should be done immediately.

## More possibilities

Normally, the action as defined for the trigger is executed after the triggering event, which causes the firing of the trigger, and is referred to as an “after trigger”. But, it can also be executed before the event “before trigger”; e.g., check the capacity before a student is added.

We can also do something else, instead of the event; i.e., instead of the triggering event, the trigger’s action will be taken instead.

As an example, when a tuple is inserted into a view (a triggering event), a trigger can be executed to *insert the row into the defining table of the view* (the action of the trigger) instead of inserting into the view itself.

## Trigger granularity

The issue is what can be called an event? *row-level granularity* assumes that a change to a row is an event, and changes to different rows are viewed as separate events that might cause the trigger to be executed multiple times.

On the other hand, *statement-level granularity* assumes that all events are statements, such as `Insert`, `Delete`, `Update`, but not the tuple-level changes they make.

Thus, an `Update` statement that does nothing can also cause a trigger to be executed. An `Insertion` into a view might cause the trigger to be fired several times, depending on into how many tables it will be inserted.

## A consequence

At the row-level granularity, a trigger might need to know the *before* and *after* value of a tuple to check the associated precondition before deciding if an action can be taken. For example, in case of a salary increase, the old tuple contains old figure and the new one the new figure. When both are available, a trigger can verify the change is indeed no more than 10%; or a corrective action will be executed.

For the statement-level granularity, it cannot check for the old/new values for all the rows involved in between every row-level action, thus updates are often collected in temporary structures such as *old table* and *new table* so that the trigger can get access to the before and after data.

## Trigger conflicts

If it is possible that an event can trigger several triggers at once. For example, when a student signs up for a course, both of the following triggers can be fired:

```
On inserting a row in registration table  
If over capacity  
Then notify registrar about unmet demands
```

```
On Inserting a row in registration table  
If over capacity  
Then put on waiting list
```

In such a situation, an important issue is which trigger should be considered first.

## Options

1. With the *ordered trigger resolution*, triggers will be evaluated in turn according to a pre-decided order. During the process, if the precondition is found to be true, its action will be executed. Upon its completion, the next trigger in the order will be considered.

For example, in the registration case, the student might accept an alternative course and abandon her request. Thus, when the second trigger is considered, its precondition is no longer true.

2. With the *group conflict resolution*, all the triggers can be evaluated at once, then an execution schedule will be made for those triggers whose condition is checked out. This approach might increase efficiency if triggers can be executed in parallel, actually concurrently.

## Triggers in SQL

In SQL 99, an event can be either Delete, Insert, or Update.

Any conditions allowed in the Where part can be used as a triggering precondition.

A triggering action can be Delete, Insert, Update, Rollback, a Signal statement, or even a program.

SQL follows the ordered resolution, and always consider triggers as immediate, and allow both tuple-level and statement-level granularity.

Triggers are not supported in *MySQL* until in version 5.0.

## *Before triggers*

Execution of a trigger can be specified as either before or after the triggering event. All *Before triggers* execute, often a test, entirely before the triggering events complete, without changing the database.

Another interesting point is that we can reference the old table and/or the new table using the Referencing Old As or Referencing New As clauses, which are the pre-update and post-update data as contained in the affected table.

The *New As* and *Old As* specify only those tuples that are affected. Thus, e.g., for an insertion event, *New As* refers to the tuple being inserted.

## Examples

Assume that our registration database includes a `CrsLimits(CrsCode, Semester, Limit)` table, we could have the following trigger that enforces the course limit by monitoring insertions into the `Transcript` table.

```
Create Trigger RoomCapacityCheck
  Before Insert On Transcript
  Referencing New As N
  For each row
  When ((Select Count(T.StudId) From Transcript T
        Where T.CrsCode=N.CrsCode A
           And T.Semester=N.Semester)
        >=
        Select.Limit From CrsLimits L
        Where L.CrsCode=N.CrsCode
           And L.Semester=N.Semester))
  Rollback
```

Notice that an insertion can add in several tuples, since it is specified as row-level granularity, thus, this trigger will be fired for each row added.

## *After triggers*

All *After triggers* execute entirely after the triggering event has applied all its changes to the database. They are allowed to change the database, and thus can cause other triggers to fire as well.

Essentially implementing a conditional structure, an *after trigger* can be perceived as an extension of the application logic, which takes care of various events automatically, thus relieving programmers of the need to code all these pieces for all cases.

## An example

The following caps any salary change by 5%, assuming the Employee table has an attribute of Salary.

```
Create Trigger LimitSalaryRaise
  After Update Of Salary On Employee
  Referencing Old as O
             New As N
  For each row
  When (N.Salary-O.Salary > 0.05*O.Salary)
    Update Employee
    Set Salary =1.05*O.Salary
  Where Id=O.Id
```

Notice that the tuples O and N always refer to the same tuple that is being updated.