9.2.1 Triggers

A **trigger** is a piece of SQL code consisting of declarative and/or procedural instructions and stored in the catalog of the RDBMS. It is automatically activated and run (also called fired) by the RDBMS whenever a specific event (e.g., insert, update, delete) occurs and a specific condition is evaluated as true. In contrast to CHECK constraints discussed in Chapter 7, triggers can also reference attribute types in other tables. Therefore, one of their applications is to enforce complex semantic constraints that cannot be captured in the basic relational model. Triggers are defined in SQL using this syntax:

CREATE TRIGGER trigger-name
BEFORE | **AFTER** trigger-event **ON** table-name
[**REFERENCING** old-or-new-values-alias-list]
[**FOR EACH** { **ROW** | **STATEMENT** }]
[**WHEN** (trigger-condition)]
trigger-body;

Let's illustrate this with a few examples. Assume we have these two relational tables:

EMPLOYEE(SSN, ENAME, SALARY, BONUS, JOBCODE, *DNR*)
DEPARTMENT(DNR, DNAME, TOTAL-SALARY, *MGNR*)

Remember that the foreign key DNR in EMPLOYEE refers to DNR in DEPARTMENT and the foreign key MGNR refers to the SSN of the department manager in EMPLOYEE. The wage of an employee consists of both a fixed salary and a variable bonus. The JOBCODE attribute type refers to the type of job the employee is assigned to. The attribute type TOTAL-SALARY in DEPARTMENT contains the total salary of all employees working in a department and should be updated whenever a new employee is assigned to a particular department. This can be accomplished with the following SQL trigger:[1]

```
CREATE TRIGGER SALARYTOTAL
AFTER INSERT ON EMPLOYEE
FOR EACH ROW
WHEN (NEW.DNR IS NOT NULL)
UPDATE DEPARTMENT
SET TOTAL-SALARY = TOTAL-SALARY + NEW.SALARY
WHERE DNR = NEW.DNR
```

This is an example of an **after trigger**, since it first inserts the employee tuple(s) and then executes the trigger body, which adjusts the attribute type TOTAL-SALARY in DEPARTMENT. The trigger is executed for each row or tuple affected by the INSERT and first verifies if the DNR of the new employee tuple is NULL or not before the update is performed.

A **before trigger** is always executed before the triggering event (in this case the INSERT operation on EMPLOYEE) can take place. Assume we now also have a relational table WAGE defined as follows:

WAGE(JOBCODE, BASE_SALARY, BASE_BONUS)

For each value of JOBCODE, this table stores the corresponding base salary and bonus. We can now define the following before trigger:

```
CREATE TRIGGER WAGEDEFAULT
BEFORE INSERT ON EMPLOYEE
REFERENCING NEW AS NEWROW
FOR EACH ROW
SET (SALARY, BONUS) =
(SELECT BASE_SALARY, BASE_BONUS
FROM WAGE
WHERE JOBCODE = NEWROW.JOBCODE)
```

This before trigger first retrieves the BASE_SALARY and BASE_BONUS values for each new employee tuple and then inserts the entire tuple in the EMPLOYEE table. Triggers have various advantages:

- automatic monitoring and verification if specific events occur (e.g., generate message if bonus is 0);
- modeling extra semantics and/or integrity rules without changing the user front-end or application code (e.g., salary should be > 0, bonus cannot be bigger than salary);
- assign default values to attribute types for new tuples (e.g., assign default bonus);
- synchronic updates if data replication occurs;
- automatic auditing and logging, which may be hard to accomplish in any other application layer;
- automatic exporting of data (e.g., to the web).

However, they should also be approached with care and oversight because they may cause:

- hidden functionality, which may be hard to follow-up and manage;
- cascade effects leading to an infinite loop of a trigger triggering another trigger, etc.;
- uncertain outcomes if multiple triggers for the same database object and event are defined;

- deadlock situations (e.g., if the event causing the trigger and the action in the trigger body pertain to different transactions attempting to access the same data – see Chapter 14);
- debugging complexities since they do not reside in an application environment;
- maintainability and performance problems.

Given the above considerations, it is very important to extensively test triggers before deploying them in a production environment.

Some RDBMS vendors also support **schema-level triggers** (also called DDL triggers) which are fired after changes are made to the DBMS schema (such as creating, dropping, or altering tables, views, etc.). Most RDBMS vendors offer customized implementations of triggers. It is recommended to check the manual and explore the options provided.