ARTICLE 6

# Assertions, Triggers, Stored Procedures, and Functions

**Abhishek Sharma**
Loyola University Chicago

**Svetlozar Nestorov**
Loyola University Chicago

In this article, we will give a brief overview of assertions, triggers, stored procedures, and functions.

**Assertion** is one of the mechanisms for specifying user-defined constraints. To illustrate and describe this mechanism, we will use the example shown in Figure 1.

In this example, a user-defined constraint states that every professor can advise up to 10 students. The following is the SQL CREATE ASSERTION statement specifying this rule:

```
CREATE ASSERTION profsadvisingupto10students
CHECK (
(SELECT MAX( totaladvised )
FROM (SELECT count(*) AS totaladvised
   FROM student
   GROUP BY advisorid)) < 11);
```

The CHECK statement verifies that the highest number of students advised by any professor is less than 11. The comparison within the CREATE ASSERTION statement would return the value TRUE if there were no advisors who advise more than 10 students. In case there are any professors who do advise more than 10 students, the comparison would return the value FALSE.

Even though CREATE ASSERTION is part of the SQL standard, most RDBMS packages do not implement assertions using CREATE ASSERTION statements. Nevertheless, we chose to give an example of a CREATE ASSERTION statement, as it is very illustrative of the process of implementing user-defined constraints. Most RDBMS packages are capable of implementing the functionality of the assertion through different, more complex types of mechanisms, such as triggers.

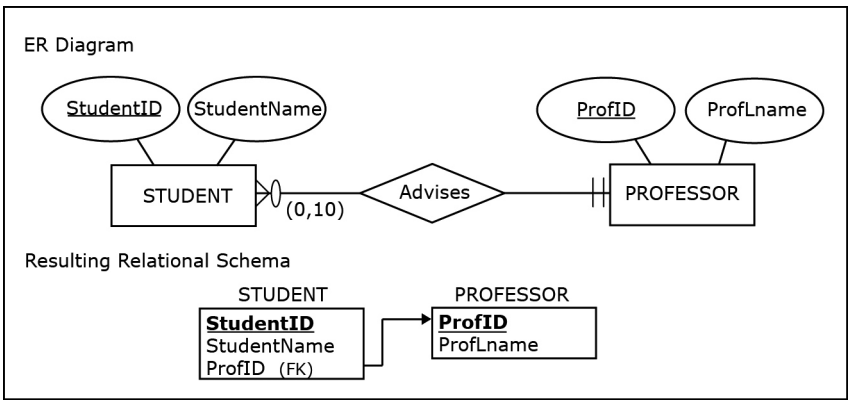


**Figure 1** Each professor can advise up to 10 students.

## Triggers

A **trigger** is a rule that is activated by a deletion of a record, an insertion of a record, or a modification (update) of a record in a relation. To illustrate triggers, consider how the assertion ProfsAdvisingUpTo10students could be implemented as the following two triggers written in Oracle:

```
CREATE OR REPLACE TRIGGER studentinserttrigger
       BEFORE INSERT ON student
FOR EACH ROW
    DECLARE totaladvised INT DEFAULT 0;
    BEGIN
       SELECT COUNT(*) INTO totaladvised
       FROM student
       WHERE advisorid = :NEW.advisorid;
       IF totaladvised >= 10
       THEN
            :NEW.advisorid := NULL;
       END IF;
    END;
CREATE OR REPLACE TRIGGER studentupdatetrigger
       BEFORE UPDATE ON student
FOR EACH ROW
    DECLARE totaladvised INT DEFAULT 0;
    BEGIN
       SELECT COUNT(*) INTO totaladvised
       FROM student
       WHERE advisorid = :NEW.advisorid;
       IF (totaladvised >= 10) THEN
            :NEW.advisorid := NULL;
       END IF;
    END;
```

The StudentInsertTrigger uses a variable (local to the trigger) TotalAdvised that is initially set to zero. This trigger is executed before each INSERT INTO Student statement is executed. The NEW.AdvisorID refers to the AdvisorID value of the student record being inserted. For example, let us assume that the following INSERT INTO statement is being issued for the relation STUDENT:

```
INSERT INTO student VALUES ('1111', 'Mark', 'P11');
```

The AdvisorID value for the new record being inserted is P11, and therefore the NEW. AdvisorID value in this case is P11. Before each insert, the SELECT query counts how many records in the STUDENT relation have the same value of AdvisorID as one in the record being inserted (P11 in this case). This count value is then placed into the TotalAdvised variable. The IF statement in the trigger, following the SELECT query, checks if the count has already reached the value of 10. If it has, the insert would cause the new count to exceed the value of 10. This is not allowed, and the trigger will cause the INSERT INTO statement to be executed as

```
INSERT INTO student VALUES ('1111', 'Mark', null);
```

thereby leaving the new student without an advisor. However, in that case, since the AdvisorID is a NOT NULL column due to the mandatory participation of the STUDENT entity in the Advises relationship, the entire INSERT INTO statement will be rejected by the RDBMS. If the count has not already reached the value of 10, then the insert statement will be executed as originally written:

```
INSERT INTO student VALUES ('1111', 'Mark', 'P11');
```

The StudentUpdateTrigger would work exactly the same as the StudentInsertTrigger, except that it would be triggered by the update (modify) operation rather than by the insert operation. The StudentDeleteTrigger is not necessary in this case, since we are making sure that the number of advised students does not exceed a certain limit. A delete operation would never increase the number of students advised by any advisor.

## Stored Procedures and Functions

Users often have to accomplish repetitive tasks in the databases that involve many sequential steps. SQL cannot handle such requirements, because SQL statements issued against a database are independent of one another and have no memory or recollection of one another. Stored procedures and functions are created to meet these requirements.

Stored procedures and functions are the collections of sequential actions that can be created and saved in the database as database objects. These database objects are written in a programming language that is a procedural extension to SQL, such as PL/SQL (Oracle), T-SQL (MS SQL), PLpgSQL (PostgreSQL), and so on. The examples and discussion here are based on Oracle but the same principles can be applied to almost every other database.

**Stored procedures** contain recorded sequential database steps or actions that are executed and that do not return any value. On the other hand, **functions** contain recorded sequential database steps or actions that are executed and that provide a return value.

A stored procedure has following structure:

```
CREATE OR REPLACE PROCEDURE procedure_name(arg1 data_type, . . .) AS
BEGIN
   . . .
END procedure_name;
```

Procedure_name is a distinct name given to a procedure. Executing the CREATE OR REPLACE clause overwrites an existing stored procedure definition with a new stored procedure definition. Depending on the arguments supplied at the innovation of the stored procedure or function, different steps are executed. This aspect of creating multiple functions with the same name but different definitions is called *function overloading*.

Arg1 is an example input/argument given to the stored procedure followed by its data type. There can be multiple arguments for a stored procedure, but the use of arguments is optional (i.e., a procedure does not have to have an argument).

The BEGIN clause starts the definition of the stored procedure and details the steps that will be executed at the invocation of the procedure.

AS keywords initiate the definition of the function or procedure. Local variables that will be used by the function or procedure are declared after the AS keyword. Each variable is followed by its data type (shown in a later example).

BEGIN—END encapsulates the body of the function or procedure. This body contains all the steps that the procedure or function will perform and uses all salient methods of programming, including loops, case statements, if statements, and so on.

Let us consider the following example of a stored procedure without an argument. Assume we want to add to the table PRODUCT a column pricecategory, which can have three values, such as “average,” “high,” and “low.” We want to update the table every day after adding new products to reflect these updated price categories. First, the table is altered as follows:

```
ALTER TABLE product ADD pricecategory CHAR(10);
```

Upon execution of the statement, the data inside the PRODUCT table is shown in Figure 2.

| ProductID | ProductName | ProductPrice | VendorID | CategoryID | PriceCategory |
|---|---|---|---|---|---|
| 1X1 | Zzz Bag | 100 | PG | CP | |
| 2X2 | Easy Boot | 70 | MK | FW | |
| 3X3 | Cosy Sock | 15 | MK | FW | |
| 4X4 | Dura Boot | 90 | PG | FW | |
| 5X5 | Tiny Tent | 150 | MK | CP | |
| 6X6 | Biggy Tent | 250 | MK | CP | |

**Figure 2** Altered table PRODUCT.

We will use a stored procedure to accomplish the repetitive task of adding price category values to products. The procedure will have following steps:

1. Create or replace a procedure named zagi_retail_store.
2. Declare three local variables:
   - avg_price of the data type number
   - pricecategory of data type that matches with the data type of the price category column of the product table
   - product_r of data type that could hold a row of table product
3. Declare a cursor (it is a way to pool a set of rows in the memory of the database server) named c1.
4. BEGIN starts the body of the procedure.
5. SELECT average price of all products from the product table into the local variable avg_price.
6. Create a FOR loop to iterate through all the rows in cursor c1 and assign that row to local variable product_r.
7. For each row, if the product price is 110% of the average price, set the local variable price-category to 'high'; if the price is less than 90% of the average price, set the local variable pricecategory to 'low'; otherwise set the pricecategory to 'average.'
8. Update the product table and set the pricecategory column to local variable price category where productid matches.
9. Commit the transaction.
10. End the loop after iterating through all the rows.
11. End the stored procedure.

The following is the syntax of this stored procedure:

```
CREATE OR REPLACE PROCEDURE zagi_retail_store
AS
avg_price NUMBER;
pricecategory product.pricecategory%type;
product_r product%rowtype;
CURSOR c1 is SELECT * FROM product;
BEGIN
SELECT avg(productprice) INTO avg_price FROM product;
FOR product_r IN c1
LOOP
   IF product_r.productprice > 1.1 * avg_price THEN pricecategory := 'high';
   ELSIF product_r.productprice < 0.9 * avg_price THEN pricecategory := 'low';
   ELSE pricecategory := 'average';
   END IF;
   UPDATE product t      SET t.pricecategory = pricecategory
   WHERE t.productid = product_r.productid ;
   COMMIT;
END LOOP;
END zagi_retail_store;
```

Once the stored procedure is created, it can be executed with the following command:

```
EXEC zagi_retail_store;
```

After executing the stored procedure, the values of the pricecategory column have been set to appropriate flags (high, low, average), as shown in Figure 3.

| ProductID | ProductName | ProductPrice | VendorID | CategoryID | PriceCategory |
|---|---|---|---|---|---|
| 1X1 | Zzz Bag | 100 | PG | CP | Low |
| 2X2 | Easy Boot | 70 | MK | FW | Low |
| 3X3 | Cosy Sock | 15 | MK | FW | Low |
| 4X4 | Dura Boot | 90 | PG | FW | Low |
| 5X5 | Tiny Tent | 150 | MK | CP | High |
| 6X6 | Biggy Tent | 250 | MK | CP | High |

**Figure 3** Altered table PRODUCT with price category values.

Now let us consider an example of a stored procedure with an argument. Let us assume the scenario that a user wants to set the pricecategory column based on a variable percentage of the average price. This requirement can be met by adding arguments to the previous stored procedure. In the following example of stored procedure, we add two arguments (highbound and lowbound) and use them in the calculation and comparison for inputting the price category column. In the previous example, the highbound was 1.1 and lowbound was 0.9, but now we can change them by changing the argument.

```
CREATE OR REPLACE
PROCEDURE zagi_retail_store_parameter (highbound number, lowbound number)
AS
avg_price NUMBER;
pricecategory product.pricecategory%TYPE;
product_r product%ROWTYPE;
CURSOR c1 is SELECT * FROM product;
BEGIN
SELECT avg(productprice) INTO avg_price FROM product;
FOR product_r IN c1
LOOP
   IF product_r.productprice > highbound * avg_price THEN pricecategory := 'high';
   ELSIF product_r.productprice < lowbound * avg_price THEN pricecategory := 'low';
   ELSE pricecategory := 'average';
   END IF;
   UPDATE product t SET t.pricecategory = pricecategory
   WHERE t.productid = product_r.productid ;
   COMMIT;
END LOOP;
END zagi_retail_store_parameter;
```

In this example, we supply highbound = 1.4 and lowbound = 0.7 and execute the stored procedure by using following command:

```
EXEC zagi_retail_store_parameter(1.4,0.7);
```

Now the data in the product table looks as shown in Figure 4.

| ProductID | ProductName | ProductPrice | VendorID | CategoryID | PriceCategory |
|---|---|---|---|---|---|
| 1X1 | Zzz Bag | 100 | PG | CP | Average |
| 2X2 | Easy Boot | 70 | MK | FW | Low |
| 3X3 | Cosy Sock | 15 | MK | FW | Low |
| 4X4 | Dura Boot | 90 | PG | FW | Average |
| 5X5 | Tiny Tent | 150 | MK | CP | Average |
| 6X6 | Biggy Tent | 250 | MK | CP | High |

**Figure 4** Result of executing a stored procedure with an argument.

Functions are written in a similar fashion as procedures, but they contain an additional syntax for the return of a value. In the declaration section, we declare the data type of the returned value. The example shown below uses three parameters/arguments (highbound, lowbound, and productid or pid). The function returns the imputed pricecategory for a specific productid supplied in pid argument using the same logic as shown in the previous example, but it does not update the table. This function can then be called in an SQL statement to update the table or just display the calculated expression.

```
CREATE OR REPLACE FUNCTION zagi_retail_store_function (highbound
            NUMBER, lowbound NUMBER, pid product.productid%TYPE)
RETURN product.pricecateogry%TYPE
AS
avg_price number;
pricecategory product.PRICE_CATEOGRY%type;
pprice product.productprice%type;
BEGIN
   SELECT avg(productprice) INTO avg_price FROM product;
   SELECT productprice INTO pprice FROM product WHERE productid =
            pid;
   IF pprice > highbound * avg_price THEN pricecategory := 'high';
   ELSIF pprice < lowbound * avg_price THEN pricecategory := 'low';
   ELSE pricecategory := 'average';
   END IF;
RETURN(pricecategory);
END zagi_retail_store_function;
```

Once the function is created, we can now call it like any other regular SQL function, and it will display the output as shown below. The pricecategory column reflects values created by the stored procedure that used highbound = 1.1 and lowbound = 0.9, but the function (as shown below using the alias pricecategoryupdated) shows the price category based on highbound = 1.4 and lowbound = 0.7.

```
SELECT p.*,
zagi_retail_store_function(1.4, 0.7, p.productid) AS
           pricecategoryupdated
FROM product p;
```

The result of this query is shown in Figure 5.

| ProductID | ProductName | ProductPrice | VendorID | CategoryID | PriceCategory | PriceCategoryUpdated |
|---|---|---|---|---|---|---|
| 1X1 | Zzz Bag | 100 | PG | CP | Low | Average |
| 2X2 | Easy Boot | 70 | MK | FW | Low | Low |
| 3X3 | Cosy Sock | 15 | MK | FW | Low | Low |
| 4X4 | Dura Boot | 90 | PG | FW | Low | Average |
| 5X5 | Tiny Tent | 150 | MK | CP | High | Average |
| 6X6 | Biggy Tent | 250 | MK | CP | High | High |

**Figure 5** Result of executing a query using a function.