Triggers in Oracle

Triggers

The creation of procedures and functions through PL / SQL certainly represents a significant advance of the DBMS in the direction of programming languages, but their practical application is still limited to calls that however must be made explicitly by the user; the DBMS simply ***passively*** run the actions requested by the user.

Triggers

It would be preferable to have tools to execute PL / SQL code in a totally automatic way, in response to events that are to be managed by the DBMS (***reactive behavior*** of the database).

In this way, the operations on the db are no longer performed only by users, but can also be the result of a behavior that the db takes *to react* to certain situations.

The knowledge relative to the reactive behavior is subtracted to the application programs and coded in the form of ***active rules*** or ***triggers***.

Triggers

Oracle, like many other commercial DBMS, allows the definition of triggers through a broad and powerful definition syntax by which you can create triggers that can easily manage different categories of situations.

Triggers can be used to:

- Ensure security / consistency of data in the db
- Prevent invalid transactions
- Manage replicated information
- Create log tables

...but they are an excellent solution to many other problems.

Triggers

A trigger is nothing more than a procedure associated with a table and is automatically invoked by the system when a certain change (or *event*) takes place within the table.

The changes on the table that can ***activate*** a trigger are:

- insert
- update
- delete

Trigger types in Oracle

Triggers can be divided into two categories:

- **Trigger at row level**: it is executed once for each tuple affected by the event that fired the trigger
- **Trigger at statement level**: it is executed only once for the event that fired the trigger, regardless of how many tuples have been affected by the event

For example, if an update done on a table affects 10 tuples, a *trigger at row level* created on the *update* event will be activated 10 times, while a *trigger at statement level* will be activated only once.

Trigger types in Oracle

In addition, depending on its timing, a trigger can be of type:

- **before:** It is executed <u>before</u> the event that caused its activation
- **after:** It is executed <u>after</u> the event that caused its activation

Trigger types in Oracle

Ultimately, then, in Oracle we can define 4 different types of trigger, which execution takes place according to the following scheme:

1. Execution of before triggers at statement level
2. Execution of before triggers at row level
3. Execution of the row update
4. Execution of after triggers at row level
5. Execution of after triggers at statement level

Triggers: syntax

The definition of a trigger is based on the following syntax:

create [or replace] trigger <*Trigger_name*>

before | after

insert [or update [of <*Column (s)*>]] [Or delete] on <*Table*>

[For each row]

[when <*Condition*>]

<*PL / SQL block*>

Triggers: syntax

```
create [or replace] trigger <Trigger_name>
before | after
insert [or update [of <Column (s)>]] [Or delete] on <Table>
[For each row]
[when <Condition>]
<PL / SQL block>
```

The optional clause or replace re-creates (overwriting) a previous trigger definition, if there was one. Without such a clause, the system does not allow the trigger redefinition.

Triggers: syntax

create [or replace] trigger <*Trigger_name*>

before | after

insert [or update [of <*Column (s)*>]] [Or delete] on <*Table*>

[For each row]

[when <*Condition*>]

<*PL / SQL block*>

This clause establishes the timing of the trigger: A trigger can be invoked before or after the event that causes its activation.

Triggers: syntax

create [or replace] trigger <*Trigger_name*>

before | after

insert [or update [of <*Column (s)*>]] [Or delete] on <*Table*>

[For each row]

[when <*Condition*>]

<*PL / SQL block*>

This part defines the event that the trigger has to monitor. Such an event can be of type insert, update, delete or a combination (in OR) of these 3 types of event.

Triggers: syntax

create [or replace] trigger <*Trigger_name*>

before | after

insert [or update [of <*Column (s)*>]] [Or delete] on <*Table*>

[For each row]

[when <*Condition*>]

<*PL / SQL block*>

For the triggers that control update operations, it is possible to specify which attributes should be affected by the operation to be controlled through the of option. The trigger will be activated only when the specified attributes are modified.

Triggers: syntax

create [or replace] trigger <*Trigger_name*>

before | after

insert [or update [of <*Column (s)*>]] [Or delete] on <*Table*>

[For each row]

[when <*Condition*>]

<*PL / SQL block*>

This clause specifies that we are defining a ***row-level trigger***. If this clause is omitted, it is assumed that the trigger is at ***statement level***.

Triggers: syntax

create [or replace] trigger <*Trigger_name*>

before | after

insert [or update [of <*Column (s)*>]] [Or delete] on <*Table*>

[For each row]

[when <*Condition*>]

<*PL / SQL block*>

The when clause can only be used with the for each row clause and it specifies additional conditions that must be satisfied by the tuples so that the trigger is activated.

Triggers: syntax

create [or replace] trigger <*Trigger_name*>

before | after

insert [or update [of <*Column (s)*>]] [Or delete] on <*Table*>

[For each row]

[when <*Condition*>]

<*PL / SQL block*>

In this clause it is possible to refer to old / new values of the current row attributes through **old** and **new** specifiers (without the colon ":").

Triggers: syntax

create [or replace] trigger <*Trigger_name*>

before | after

insert [or update [of <*Column (s)*>]] [Or delete] on <*Table*>

[For each row]

[when <*Condition*>]

<*PL / SQL block*>

This section is the body of the trigger and consists of PL / SQL code (with the sections of declarations, executable commands and exception handling). The only commands that can not be used in this section are commit and rollback.

Triggers: syntax

create [or replace] trigger <*Trigger_name*>

before | after

insert [or update [of <*Column (s)*>]] [Or delete] on <*Table*>

[For each row]

[when <*Condition*>]

<*PL / SQL block*>

Moreover, in this section it is possible to use the additional constructs if as a selection statement. In particular, it is possible to control the operation that has activated the trigger through the constructs if inserting, if updating [('<column>')] and if deleting.

A clarification on row-level triggers

Row level triggers allow to access the values of the affected row attributes, before (with the specifier :old) and after the change (with the specifier :new).

For example it is possible to find in the body of the trigger:

```
if (:new.price < :old.price) ...
```

Obviously for an update trigger both values of the row (old and new) can be accessed, while for an insert trigger only the new values (:new) are available, and for a delete trigger only the old values (: old) are available.

A clarification on row-level triggers

The new value of the row can also be modified by the trigger, but such changes can only be made if the temporal context of the trigger is *before*.

For example, in the body of a *before* trigger, it is possible to find the following code:

```
if (:new.price > :old.price * 1.5) then
    :new.price := :old.price * 1.3;
End if;
```

This operation is not permitted for *after* triggers.

Triggers: selection criteria

How to choose between different types of triggers?

In general, the following rules are taken into account to define the trigger type:

- If you want to access the value of the tuples that activate the trigger, you have to define a ***row-level trigger***, otherwise a ***statement level trigger*** is preferred
- If you want to change the value of some attributes in the tuples that activate the trigger, it is necessary to define a ***before*** trigger, otherwise, an ***after*** trigger is preferred, since it is handled more efficiently by Oracle.

Triggers and computations not ending

CAUTION:

Triggers typically perform operations on the database. This means that the execution of a trigger may cause the activation of another trigger, or even the activation of itself.

It can be observed immediately that an incorrect formulation of a trigger can lead to non-terminating computations. How to avoid them?

Reached the execution of 32 triggers in cascade, the system assumes a situation of infinite execution and suspends the execution, raising a particular exception.

Further operations on triggers

In addition to creating and replacing a trigger, you can delete an existing trigger using the command:

Drop trigger <*TriggerName*>

After the creation of a trigger, this is automatically compiled and, if there are no errors, enabled. To disable a trigger the following command is used:

Alter trigger <*TriggerName*> disable

Further operations on triggers

Finally, it is possible to enable / disable all triggers defined on a table in a single command:

Alter table <*TableName*> enable all trigger;

Alter table <*TableName*> disable all trigger;

Examples of triggers for Yellowcom

1) Creation of a trigger that avoids to insert/update of an operation if the credit is less than 0

2) Creation of a trigger that keep updated the credit associated with a contract (rechargeable) for an insertion or modification in the table operation (in the lab)

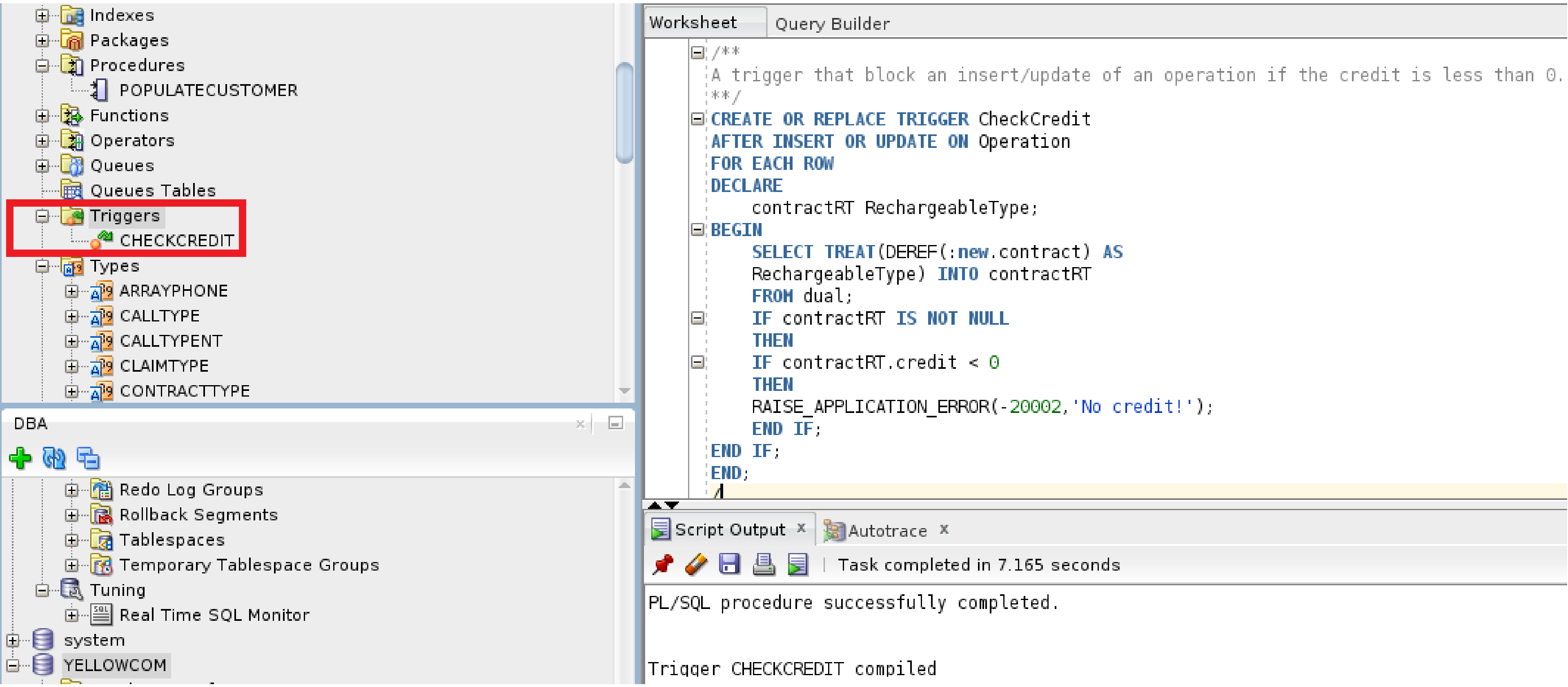
NOTE: In general, you can access attributes using dot notation. To access attributes of a subtype of a row or column's declared type, you can use the *TREAT* function.

For example: SELECT name, TREAT(VALUE(p) AS Student_typ).major major FROM persons p;

Trigger 1

```
CREATE OR REPLACE
TRIGGER CheckCredit
AFTER INSERT OR UPDATE ON Operation
FOR EACH ROW
DECLARE
 contractRT RechargeableType;
BEGIN
 SELECT TREAT(DEREF(:new.contract) AS RechargeableType) INTO contractRT
 FROM dual;
 IF contractRT IS NOT NULL
 THEN
  IF contractRT.credit < 0
  THEN
   RAISE_APPLICATION_ERROR(-20002,'No credit!');
  END IF;
 END IF;
END;
/
```

Trigger 1



```
/**
A trigger that block an insert/update of an operation if the credit is less than 0.
**/
CREATE OR REPLACE TRIGGER CheckCredit
AFTER INSERT OR UPDATE ON Operation
FOR EACH ROW
DECLARE
    contractRT RechargeableType;
BEGIN
    SELECT TREAT(DEREF(:new.contract) AS
    RechargeableType) INTO contractRT
    FROM dual;
    IF contractRT IS NOT NULL
    THEN
    IF contractRT.credit < 0
    THEN
    RAISE_APPLICATION_ERROR(-20002,'No credit!');
    END IF;
END IF;
END;
/
```

Task completed in 7.165 seconds

```
PL/SQL procedure successfully completed.


Trigger CHECKCREDIT compiled
```

Trigger 2 (page 1)

```
CREATE OR REPLACE
TRIGGER BalanceCredit
AFTER INSERT ON Operation
FOR EACH ROW
DECLARE
 contractRT RechargeableType;
 stp SubscriptionTariffPlanType;
 tp TariffPlanType;
 callOperation CallType;
 textOperation TextType;
 internetOperation InternetConnectionType;
```

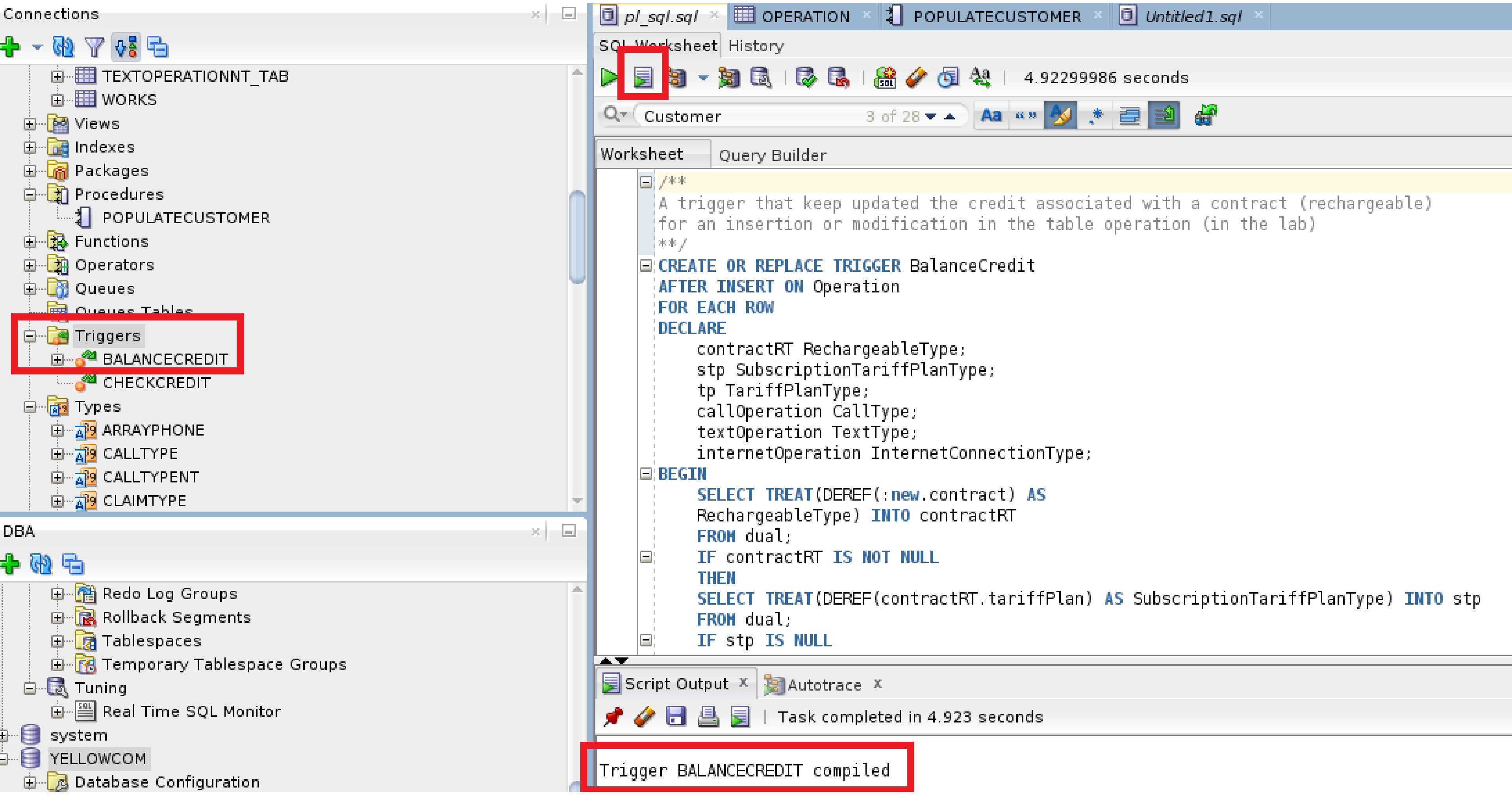
Trigger 2 (page 2)

```
BEGIN
 SELECT TREAT(DEREF(:new.contract) AS RechargeableType) INTO contractRT
 FROM dual;
 IF contractRT IS NOT NULL
 THEN
  SELECT TREAT(DEREF(contractRT.tariffPlan) AS SubscriptionTariffPlanType) INTO stp
  FROM dual;
   IF stp IS NULL
   THEN
    SELECT TREAT(DEREF(contractRT.tariffPlan) AS TariffPlanType) INTO tp
    FROM dual;
    SELECT TREAT(:new.details AS CallType) INTO callOperation --check if it is a call
    FROM dual;
     IF callOperation IS NOT NULL
     THEN
      contractRT.credit := contractRT.credit - (tp.callPrice * callOperation.durationCall);
      UPDATE Contract ct SET value(ct) = contractRT WHERE ct.codeContract = contractRT.codeContract;
     END IF;
```

Trigger 2 (page 3)

```
SELECT TREAT(:new.details AS TextType) INTO textOperation --check if it is a text
    FROM dual;
     IF textOperation IS NOT NULL
     THEN
      contractRT.credit := contractRT.credit - tp.smsPrice;
      UPDATE Contract ct SET value(ct) = contractRT WHERE ct.codeContract = contractRT.codeContract;
     END IF;
    SELECT TREAT(:new.details AS InternetConnectionType) INTO internetOperation --check if it is a
internet operation
    FROM dual;
     IF internetOperation IS NOT NULL
     THEN
      contractRT.credit := contractRT.credit - (tp.internetPrice * internetOperation.amountOfData);
      UPDATE Contract ct SET value(ct) = contractRT WHERE ct.codeContract = contractRT.codeContract;
     END IF;
   END IF;
 END IF;
END;
/
```

Trigger 2



```
/**
A trigger that keep updated the credit associated with a contract (rechargeable)
for an insertion or modification in the table operation (in the lab)
**/
CREATE OR REPLACE TRIGGER BalanceCredit
AFTER INSERT ON Operation
FOR EACH ROW
DECLARE
    contractRT RechargeableType;
    stp SubscriptionTariffPlanType;
    tp TariffPlanType;
    callOperation CallType;
    textOperation TextType;
    internetOperation InternetConnectionType;
BEGIN
    SELECT TREAT(DEREF(:new.contract) AS
    RechargeableType) INTO contractRT
    FROM dual;
    IF contractRT IS NOT NULL
    THEN
    SELECT TREAT(DEREF(contractRT.tariffPlan) AS SubscriptionTariffPlanType) INTO stp
    FROM dual;
    IF stp IS NULL
```

```
Trigger BALANCECREDIT compiled
```