Chapter 20

DATABASE TRIGGERS

- What is a database trigger?
- Types of Triggers
- Creating a database trigger
- Correlation names
- Instead-of triggers
- Knowing which command fired the trigger
- Enabling and disabling trigger
- Dropping a trigger

What is a Database Trigger?

Database trigger is a PL/SQL block that is executed on an event in the database. The event is related to a particular data manipulation of a table such as inserting, deleting or updating a row of a table.

Triggers may be used for any of the following:

- To implement complex business rule, which cannot be implemented using integrity constraints.
- To audit the process. For example, to keep track of changes made to a table.
- To automatically perform an action when another concerned action takes place. For example, updating a table whenever there is an insertion or a row into another table.

Triggers are similar to stored procedures, but stored procedures are called explicitly and triggers are called implicitly by Oracle when the concerned event occurs.

Note: Triggers are automatically executed by Oracle and their execution is transparent to users.

Types of Triggers

Depending upon, when a trigger is fired, it may be classified as :

- Statement-level trigger
- Row-level trigger
- Before triggers
- After triggers

Statement-level Triggers

A statement trigger is fired only for once for a DML statement irrespective of the number of rows affected by the statement.

For example, if you execute the following UPDATE command STUDENTS table, statement trigger for UPDATE is executed only for once.

```
update students   set  bcode='b3'
where bcode = 'b2';
```

However, statements triggers cannot be used to access the data that is being inserted, updated or deleted. In other words, they do not have access to keywords NEW and OLD, which are used to access data.

Statement-level triggers are typically used to enforce rules that are not related to data. For example, it is possible to implement a rule that says "no body can modify BATCHES table after 9 P.M".

Statement-level trigger is the default type of trigger.

Row-level Trigger

A row trigger is fired once for each row that is affected by DML command. For example, if an UPDATE command updates 100 rows then row-level trigger is fired 100 times whereas a statement-level trigger is fired only for once.

Row-level trigger are used to check for the validity of the data. They are typically used to implement rules that cannot be implemented by integrity constraints.

Row-level triggers are implemented by using the option FOR EACH ROW in CREATE TRIGGER statement.

Before Triggers
While defining a trigger, you can specify whether the trigger is to be fired before the command (INSERT, DELETE, and UPDATE) is executed or after the command is executed.

Before triggers are commonly used to check the validity of the data before the action is performed. For instance, you can use before trigger to prevent deletion of row if deletion should not be allowed in the given case.

AFTER Triggers
After triggers are fired after the triggering action is completed. For example, If after trigger is associated with INSERT command then it is fired after the row is inserted into the table.

Possible Combinations
The following are the various possible combinations of database triggers.

- Before Statement
- Before Row
- After Statement
- After Row

***Note**: Each of the above triggers can be associated with INSERT, DELETE, and UPDATE commands resulting in a total of 12 triggers.*

In the next section, we will see how to create database triggers.

Creating a Database Trigger
CREATE TRIGGER command is used to create a database trigger. The following details are to be given at the time of creating a trigger.

- Name of the trigger
- Table to be associated with
- When trigger is to be fired - before or after
- Command that invokes the trigger - UPDATE, DELETE, or INSERT
- Whether row-level trigger or not
- Condition to filter rows.
- PL/SQL block that is to be executed when trigger is fired.

The following is the syntax of CREATE TRIGGER command.

```
CREATE  [OR REPLACE] TRIGGER trigername
{BEFORE | AFTER}
{DELETE | INSERT | UPDATE [OF columns]}
        [OR {DELETE | INSERT |UPDATE [OF columns]}]...
ON table
[FOR EACH ROW [WHEN condition]]
[REFERENCING  [OLD AS old] [NEW AS new]]
PL/SQL block
```

If FOR EACH ROW option is used then it becomes a row-level trigger otherwise it is a statement-level trigger.

WHEN is used to fire the trigger only when the given condition is satisfied. This clause can be used only with row triggers.

For example, the following trigger is fired only when AMOUNT is more than 1000.

```
create or replace trigger ..
before insert on payments
for each row
when  :new.amount > 1000
```

OF option allows you to specify updation of which columns will fire trigger. The list of columns can be given by separating column by comma.

REFERENCING is used to use new names instead of default correlation names OLD and NEW. See the section "Correlation Names "

The following is a simple database trigger that is used to check whether date of joining of the student is less than or equal to system date. Otherwise it raises an error.

```
create or replace trigger students_bi_row
before insert
on students
for each row
begin
    if :new.dj > sysdate then
       raise_application_error
           (-20002,'Date of joining cannot be after system date.');
    end if;
end;
```

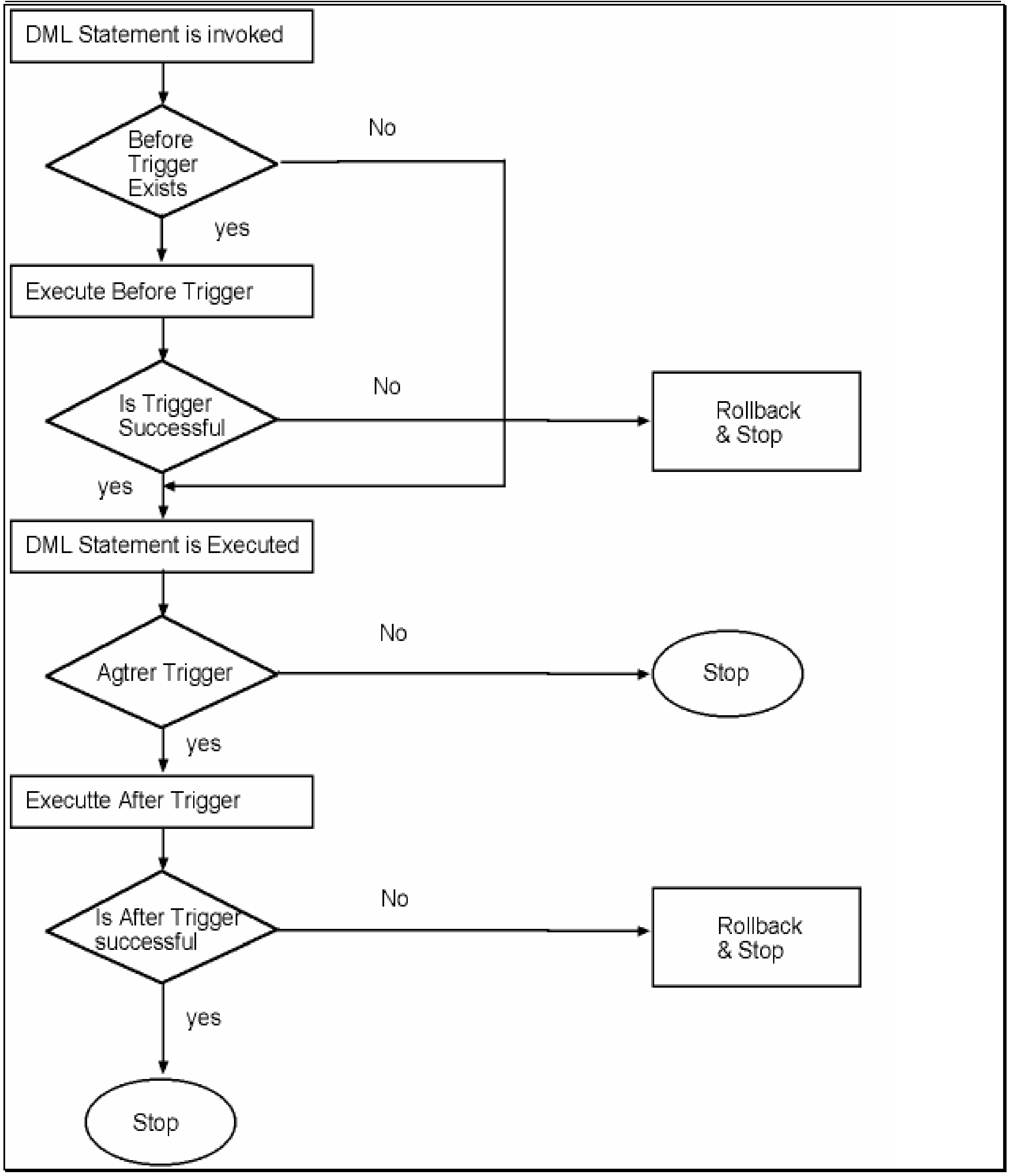

Figure 1: Execution sequence of database triggers.

STUDENTS_BI_ROW is the name of the trigger. It represents table name – STUDENTS, event of the trigger – BI (before insert) and type – ROW. Though trigger name can be anything, it is better you follow a convention while naming it.

FOR EACH ROW specifies that this trigger is a row-level trigger.

Condition :NEW.DJ > SYSDATE checks whether the value of DJ of the row being inserted is greater than system date. If the condition is true then it raises an error using RAISE_APPLICATION_ERROR procedure.

Note*: Use data dictionary view USER_TRIGGERS to list the existing triggers.*

The following command can be used to display the name and body of each trigger in the current account.

```
select trigger_name, trigger_body from user_trigger;
```

Column TRIGGER_BODY is of LONG type. How many characters of this column will be displayed is determined by variable LONG in SQL*Plus. Set variable LONG to a larger value to see the complete body of the trigger. The default value of LONG is 80.

The following is another trigger used to implement a rule that cannot be implemented by integrity constraints. The trigger checks whether payment made by the student is more than what the student is supposed to pay.

```
create or replace trigger payments_bi_row
before insert
on payments
for each row
declare
  v_dueamt number(5);
begin
   v_dueamt := getdueamt(:new.rollno);
   if  :new.amount > v_dueamt then
          raise_application_error(-20112,'Amount being paid is more than what
is to be paid');
   end if;
end;
```

The above trigger makes use of GETDUEAMT function that we created in the previous chapter. It first gets the due amount of the given roll number. If the amount being paid – values of AMOUNT column – is more than the due amount then trigger fails.

Statement-level trigger example

The following is an example of statement-level trigger. As we have seen in the previous section, a statement trigger is fired only for once for the entire statement irrespective of the number of rows affected by the statement.

```
create or replace trigger payments_biud
before insert or update or delete
on payments
begin
  if to_char(sysdate,'DY') = 'SUN' then
    raise_application_error(-20111,'No changes can be made on sunday.');
  end if;
end;
```

The above trigger is fired whenever an UPDATE or DELETE or INSERT command is executed on PAYMENTS table. Trigger checks whether day of week of the system date is Sunday. If condition is true then it raises an application error. Since the trigger is not concerned with data, it uses a statement-level trigger.

Valid Statements in trigger body

The following are the only valid statements in trigger body. Trigger can make use of existing stored procedures, functions and packages (as shown in the previous example).

- DML commands
- SELECT INTO command

Correlation Names

The default correlation names are NEW for new values and OLD for old values of the row. So, in order to access the values of new row use NEW and to access the values of old (existing) row, use OLD.

It is also possible to change these correlation names using REFERENCING option of CREATE TRIGGER command. This is done mainly to avoid names conflicts between table names and correlation names.

See the table 1, to understand the availability of correlation name with triggering commands.

Command	**NEW**	**OLD**
DELETE	No	Yes
INSERT	Yes	No
UPDATE	Yes	Yes

Table 1: Availability of correlation names

NEW is not available with DELETE command because there are no new values. In the same way, OLD is not available with INSERT because a new row is being inserted and it doesn't contain old values. Triggers based on UPDATE command can access both NEW and OLD correlation names. OLD refers to values that refer to values before the change and NEW refers to values that are after the change.

The following trigger prevents any change to AMOUNT column of PAYMENTS table. This is done by comparing old value with new value and if they differ that means the value is changed and trigger fails.

```
create or replace trigger payments_bu_row
before update
on payments
for each row
begin
   if  :new.amount <> :old.amount then
          raise_application_error(-20113,'Amount cannot be changed. Please
delete and reinsert the row, if required');
   end if;
end;
```

Instead-of Trigger

These trigger are defined on relation-views and object-views. These triggers are used to modify views that cannot be directly modified by DML commands. Unlike normal trigger, which are fired during the execution of DML commands, these triggers are fired instead of execution of DML commands. That means instead of executing DML command on the view, Oracle invokes the corresponding INSTEAD-OF trigger.

The following is a view based on STUDENTS and PAYMENTS tables.

```
create view  newstudent
as
select  s.rollno, name, bcode,gender, amount
from students s, payments p
where  s.rollno = p.rollno;
```

```
If you try to insert data into NEWSTUDENT table then Oracle displays the
following error.

SQL> insert into newstudent values (15,'Joe','b2','m',2000);
insert into newstudent values (15,'Joe','b2','m',2000)
*
ERROR at line 1:
ORA-01779: cannot modify a column which maps to a non key-preserved table
```

But, we want the data supplied to NEWSTUDENT view to be inserted into STUDENTS and PAYMENTS table. This can be done with an instead of trigger as follows:

```
create or replace trigger newstudent_it_bi_row
instead of insert
on newstudent
for each row
begin
     -- insert into STUDENTS table first

     insert into students(rollno,bcode,name,gender,dj)
        values(:new.rollno, :new.bcode, :new.name, :new.gender,sysdate);
     -- insert a row into PAYMENTS table

     insert into payments
           values(:new.rollno, sysdate, :new.amount);
end;
```

The above INSTEAD-OF trigger is used to insert one row into STUDENTS table with roll number, name, batch code and gender of the student. It then inserts a row into PAYMENTS table with roll number, date of payment - sysdate, and amount.

Since we created an INSTEAD-OF trigger for INSERT command, Oracle invokes this trigger when an INSERT command is executed on NEWSTUDENTS view.

```
SQL> insert into newstudent values (15,'Joe','b2','m',2000);
```

So the above command inserts two rows – one into STUDENTS table and another into PAYMENTS table. The following two queries will ratify that.

```
select rollno, bcode,name, gender, dj
from students where rollno = 15;

   ROLLNO BCODE NAME                           G DJ
--------- ----- ------------------------------ - ---------
       15 b2    Joe                            m 30-Oct-01



select * from payments where rollno = 15;

   ROLLNO DP           AMOUNT
--------- --------- ---------
       15 30-OCT-01      2000
```

Knowing which command fired the trigger

When a trigger may be fired by more than one DML command, it may be required to know which DML command actually fired the trigger.

Use the following conditional predicates in the body of the trigger to check which command fired the trigger.

- INSERTING
- DELETING
- UPDATING

The following example is used to log information about the change made to table COURSES. The trigger makes use of a COURSES_LOG table, which is created with the following structure.

```
create table courses_log
(  cmd  number(1),
   pk   varchar2(20),
   dc   date,
   un   varchar2(20)
);
```

Now a row level trigger is used to insert a row into COURSES_LOG table whenever there is a change to COURSES table.

```
create or replace trigger  courses_biud_row
before insert or delete or update
on courses
for each row
begin
    if  inserting then
            insert into courses_log values (1, :new.ccode,sysdate, user);
    elsif  deleting then
            insert into courses_log values(2,:old.ccode,sysdate,user);
    else
            insert into courses_log values(3,:old.ccode,sysdate,user);
    end if;
end;
```

After the trigger is created, if you execute an UPDATE command as follows:

```
update courses set fee = fee * 1.1 where ccode = 'ora';
```

it will fire COURSES_BIUD_ROW trigger, which will insert a row into COURSES_LOG table as follows:

```
SQL> select * from courses_log;

      CMD PK                   DC        UN
--------- -------------------- --------- --------------------
        3 ora                  30-OCT-01 BOOK
```

Enabling and disabling triggers

A database trigger may be either in enabled state or disabled state.

Disabling trigger my improve performance when a large amount of table data is to be modified, because the trigger code is not executed.

For example,

```
UPDATE students set name = upper(name);
```

will run faster if all the triggers fired by UPDATE on STUDENTS table are disabled.

Use ALTER TRIGGER command to enable or disable triggers as follows:

```
alter trigger  payments_bu_row disable;
```

A disabled trigger is not fired until it is enabled. Use the following command to enable a disabled trigger.

```
alter trigger payments_bu_row enable;
```

The following command disables all triggers of STUDENTS table.

```
alter table  students disable all triggers;
```

Dropping a Trigger

When a trigger is no longer needed it can be dropped using DROP TRIGGER command as follows.

```
drop trigger payments_bu_row;
```

Summary

Database triggers are used to implement complex business rules that cannot be implemented using integrity constraints. Triggers can also be used for logging and to take actions that should be automatically performed.

Trigger may be fired before the DML operation or after DML operation. Trigger may be fired for each row affected by the DML operation or for the entire statement.

INSTEAD-OF trigger are called instead of executing the given DML command. They are defined on relational-views and object-views (will be discussed later in this book).

A trigger can be disabled to increase the performance when a lot of data manipulations are to be done on the table.

Exercises

1. Which data dictionary view contains information about triggers?_______________
2. How many before trigger can we create?__________
3. Is it possible to create two or more trigger for the same event(BEFORE INSERT)?_____.
4. What is the default type of trigger?[Statement/row]___
5. Create a trigger to prevent any changes to FEE column of COURSES table in such a way that increase is more than the half of the existing course fee.
6. Create a trigger to prevent all deletions between 9p.m to 9 a.m.