Triggers

Triggers are similar to procedures and functions, but triggers are executed implicitly whenever the triggering event happens, and they don't accept arguments. The act of executing a trigger is known as *firing* the trigger. The triggering event event can be:

- a DML operation on a database or certain kinds of view
- some DDL operations (such as table creation)
- some system operations like (a DB startup or shutdown)

Using triggers we can

- maintain a complex integrity constraints not possible through declarative constraints enabled at table creation;
- audit information in a table by recording the changes made and who made them;
- automatically signal other programs that action needs to take place when changes are made to a table;
- publish information about various events in a publish-subscribe environment.

Creating triggers

The general syntax to create a trigger is

```
create [or replace] trigger trigger_name
 {before | after | instead of} triggering_event
 [referencing_clause]
 [when trigger_condition]
 [for each row]
 trigger_body;
```

Creating DML triggers

A DML trigger is fired on an `insert`, `update`, or `delete` operations on a database. It can be fired before or after the statement executes, and can be fired once per row, or once per statement. The table below summarizes the various options:

Category	Values	Comments
Statement	`insert`, `update`, or `delete`	Defines which kind of DML statement causes the trigger to fire.
Timing	`before` or `after`	Defines whether the trigger fires before or after the statement is executed.
Level	row or statement	If the trigger is a row-level trigger, it fires once for each row affected by the triggering statement. If the trigger is a statement-level trigger, it fires once, either before or after the statement. A row-level triggers are identified by the **`for each row`** clause.

`instead of` triggers

Oracle 8*i* (and higher) provides an additional kind of trigger. **`instead of`** trigger can be defined on views only. Unlike DML triggers, which execute in addition to the DML

operation, an instead-of trigger will execute *instead* of the DML statement that fired it. Instead-of trigger must be row-level. These triggers are useful in situations when we need to insert into a join view. It's illegal to `insert` into a view which is a join view of two or more tables and insert requires that both underlying tables were modified. Instead we can use an instead-of trigger on the `insert` event on that view and modify the underlying tables separately.

Order of DML trigger firing

Triggers are fired as the DML statement is executed. The algorithm for executing a DML statement is given here:

1. Execute the `before` statement-level triggers, if present.
2. For each row affected by the statement:
    1. Execute the `before` row-level triggers, if present.
    2. Execute the statement itself.
    3. Execute the `after` row-level triggers, if present.
3. Execute the `after` statement-level triggers, if present.

A very good example from Scott Urman's *PL/SQL Programming* illustrates this algorithm.

Correlation identifiers in row-level triggers

Inside a row-level trigger (which executes once per row processed by the triggering statement) we can access the data in the row that is currently being processed. This is accomplished through two correlation identifiers - **`:old`** and **`:new`**. The colon in front of each indicates that they re bind variables and not regular PL/SQL variables. The PL/SQL compiler treats them as records of type ***`triggering_table`*`%rowtype`**, where the *triggering table* is the table for which the trigger is defined. Thus, the reference such as `:new.field` will be valid only if there is a column *field* in the triggering table. The meanings of `:old` and `:new` are explained in the table:

Statement	**`:old`**	**`:new`**
`insert`	undefined - all fields are `NULL`	Values that will be inserted when the statement is complete
`update`	Original values for the row before the update	New values that will be updated when the statement is complete
`delete`	Original values before the row is deleted	undefined - all fields are `NULL`

If desired, we can use the *referencing* clause to specify a different name for `:old` and `:new`. This clause has the following syntax:

```
referencing [old as old_name] [new as new_name]
```

For example,

```
create or replace trigger GenerateProfID
 before insert on Professors
 referencing new as new_prof
```

```
 for each row
begin
  if :new_prof.pid is null then
    select profids.nextval into :new_prof.pid from dual;
  end if;
end GenerateProfID;
```

The `when` clause

The **`when`** clause is valid for row-level triggers only. If present, the trigger body will be executed only for those rows that meet the condition specified. The syntax of the `when` clause is

```
when trigger_condition
```

where *trigger_condition* is a Boolean expression. The `:new` and `:old` records can be referenced inside the expression, but like `referencing`, the colon is *not* used there. In the following example the trigger will be executed only for 'DB' department:

```
create trigger InsertStudent
 after insert or update on Student
 for each row
 when (new.mid = 'DB')
begin
  /* put code here */
end InsertStudent;
```

Trigger predicates

If we use one trigger that can be fired by several statements (for example, `insert` and `update`), we may need to find out inside the trigger which statement fired the trigger. Oracle provides three Boolean functions that we can use to determine this. These functions are:

- **`inserting`** - is `true` if the triggering statement is `insert`; `false` otherwise.
- **`updating`** - is `true` if the triggering statement is `update`; `false` otherwise.
- **`deleting`** - is `true` if the triggering statement is `delete`; `false` otherwise.

System triggers

System triggers, unlike DML triggers, are fired on two different kind of events: DDL or database. DDL events include: `create`, `alter`, or `drop` statements, whereas database events include startup/shutdown of the server, logon/logof of a user, and a server error. The general syntax for creating a system trigger is

```
create [or replace] trigger [schema.]trigger_name
  { before | after }
  { DDL_event_list | DB_event_list}
  on { database | [schema.]schema }
  [when condition]
  trigger_body
```

Event	Timing allowed	Description
`startup`	`after`	Fired when an instance is started up.

shutdown	before	Fired when an instance is shut down. This trigger may not fire if the DB is shutdown abnormally.
servererror	after	Fired whenever an error occurs.
logon	after	Fired after a user has successfully connected to the database.
logoff	before	Fired at the start of the logoff.
create	before, after	Fired before or after a schema object is created.
drop	before, after	Fired before or after a schema object is dropped.
alter	before, after	Fired before or after a schema object is altered.

A system trigger can be defined at the database level trigger or a schema level trigger (keywords **database** and **schema**). A database-level trigger will fire whenever the triggering event occurs, whereas a schema-level trigger will fire only when the triggering event occurs for the specified schema. If the schema name is not specified with the schema keyword, it defaults to the schema that owns the trigger. The following trigger monitors all logons to the current schema

```
create or replace trigger MonitorLogons
  after logon on schema
begin
  insert into danil.logons values (user, sysdate);
end MonitorLogons;
```

This trigger will monitor only the connections to the schema that owns this trigger, but if we change the keyword schema to database we will monitor all connections to the database. Don't forget that all users have to have the insert privilege on the table used in the trigger and also the person who creates the trigger should have enough privileges to do so. Please see more detailed example here. Please also read this chapter from Oracle documentation to better understand the difference between schema and database.

Privilege	Description
create trigger	Allows the grantee to create a trigger in his or her own schema.
create any trigger	Allows the grantee to create triggers in any schema except SYS.
alter any trigger	Allows the grantee to enable, disable, or compile database triggers in any schema except SYS.
drop any trigger	Allows the grantee to drop database triggers in any schema except SYS.
administer database trigger	Allows the grantee to create or alter a system trigger on the database. The grantee must also have either create trigger or create any trigger privilege.

System privileges related to triggers

Event attribute function

There are several event attribute functions that allow in system trigger body to get information about the triggering event. We already saw trigger predicates (inserting, deleteing, updating). These functions work similar to these, but may return

string or number values not only boolean. Unlike these trigger predicates, event attribute functions are stand-alone PL/SQL functions that are owned by **sys**. There are no synonyms defined for them by default, so they must be prefixed by `sys.` in order to resolve them.

Function	Datatype	Event applicable for	Description
`sysevent`	`varchar2(20)`	All events	Returns the system event that fired the trigger
`instance_num`	`number`	All events	Returns the current instance number. This will always be 1 unless you are running with Oracle Real Application Cluster
`database_name`	`varchar2(50)`	All events	Returns the current database name.
`server_error`	`number`	`servererror`	Takes a single `number` argument. Returns the error at the position on the error stack indicated by the argument. The position 1 is the top of the stack.
`is_servererror`	`boolean`	`servererror`	Takes an error number as an argument, and returns `true` if the Oracle indicated is on the error stack.
`login_user`	`varchar2(30)`	All events	Returns the user ID of the user that fired the trigger.
`dictionary_obj_type`	`varchar2(20)`	`create, alter, drop`	Returns the type of the dictionary object on which the DDL operation that fired the trigger occurred.
`dictionary_obj_name`	`varchar2(30)`	`create, alter, drop`	Returns the name of the dictionary object on which the DDL operation that fired the trigger occurred.
`dictionary_obj_owner`	`varchar2(30)`	`create, alter, drop`	Returns the owner of the dictionary object on which the DDL operation that fired the trigger occurred.
`des_encrypted_password`	`varchar2(30)`	`create` or `alter` user	Returns the DES encrypted password of the user being created or altered.

Event attribute functions

The following example illustrates how to use these functions. In this example we create a trigger that monitors all `create` operations against a database. If such an operation happens, the trigger check what kind of object was created. If the created object is a user object, then information about who and when created a new user as well as new user ID is inserted into the table *example.new_users*

```
create or replace trigger MonitorCreation
 after create on database
```

```
declare
  msg TestCreateTrig.event%type;
begin
  if sys.dictionary_obj_type = 'USER' then
    msg := 'User ' || sys.login_user;
    msg := msg || ' created user ' || sys.dictionary_obj_name;
        msg := msg || ' identified by ' || sys.des_encrypted_password;
    insert into system.TestCreateTrig values (user, sysdate, msg);
  end if;
end MonitorCreation;
```

Restrictions on triggers

The body of the trigger is a PL/SQL block. That is, any statement legal in PL/SQL block is legal in a trigger body, with the following restrictions apply:

- A trigger may not issue any **transaction control statements**.
- That also means that any procedure or a function called in the body of the trigger may not issue any transaction control statements.
- The trigger body cannot declare `long` or `long raw` variables.
  Also, `:new` and `:old` variables cannot refer to a `long` or `long raw` columns in the table for which trigger is defined.
- In Oracle8 and higher, code in a trigger body may reference and use `lob` columns, but it may not modify the values of the columns.

Just like DML triggers, system triggers may use th e`when` clause to specify a condition on the trigger firing. However, there are restrictions on the type of conditions that may be specified for each type of system trigger namely:

- `startup` and `shutdown` triggers cannot have any conditions
- `servererror` triggers can use `errno` test to check for a specific error only
- `logon` and `logoff` triggers can check the user ID or username with the `userid` or `username` tests
- DDL triggers can check the type and name of the object being modified, and can check the useris or user name

Other trigger issues

Once a trigger created, its source code and all other information about the trigger is stored into the system dictionary table **`all_triggers`**. Also there is the view **`user_triggers`** based on that table that shows all information about the current user triggers.

To drop a trigger user needs to execute the following command:

```
drop trigger trigger_name;
```

This command permanently removes the trigger from the data dictionary. Of course, the user has to have enough privileges to successfully execute this command. Unlike

procedures and packages, a trigger may be disabled without removing it. If a trigger is disables it still is in the data dictionary, but it cannot be fired. To disable a trigger, a user should execute the comand

```
alter trigger trigger_name disable;
```

To enable the trigger back use

```
alter trigger trigger_name enable;
```

All triggers for a particular table can be enable or disable with one `alter table` command:

```
alter table table_name  {disable | enable} all triggers;
```