**Triggers**

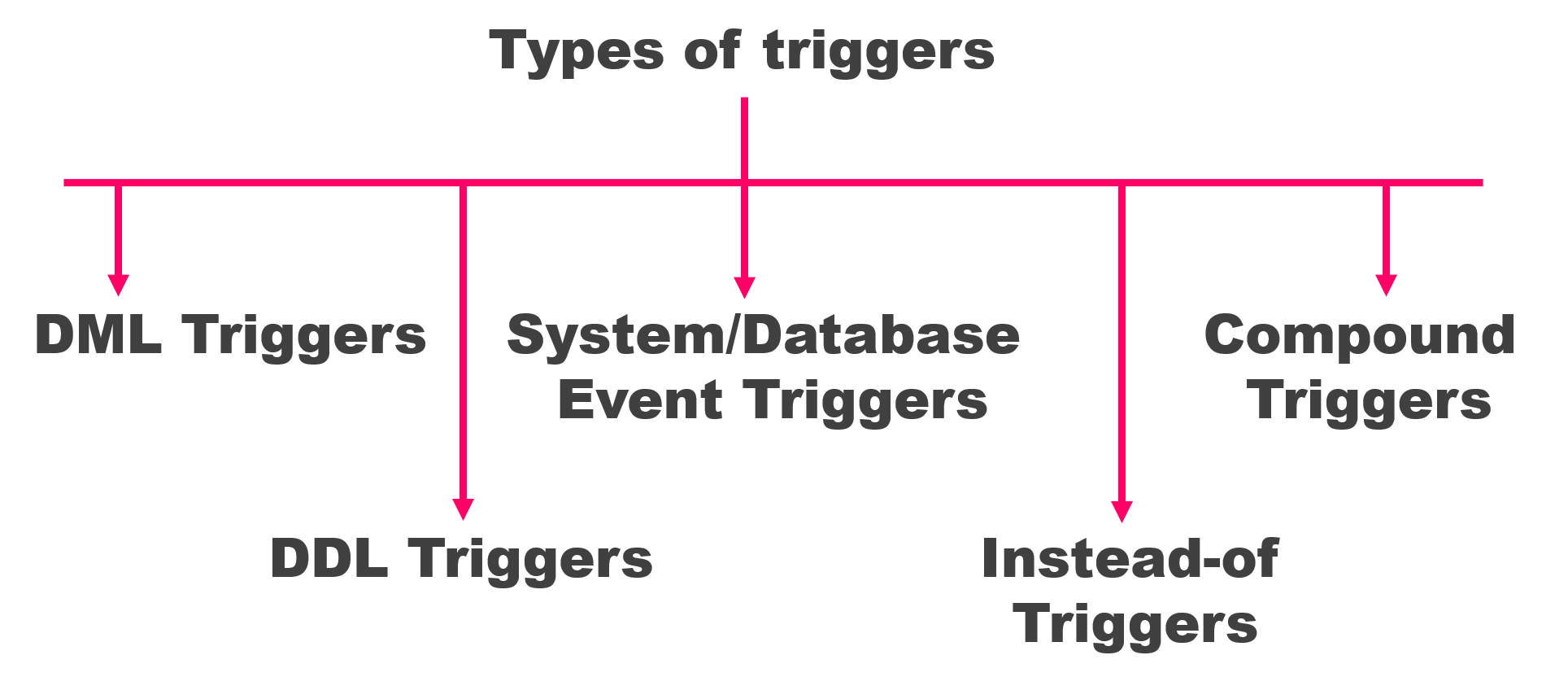
Triggers are named pl/sql blocks which are stored in the database.

Specialized stored programs which execute implicitly when a triggering event occurs.

**EVENTS**

* A DML statement
* A DDL statement
* A system event
* A user event

**Types of Triggers**

There are 5 types of triggers in oracle database in which 3 of them are based on the triggering event which are discussed in the previous section.  
  


1. **Data Manipulation Language Triggers or DML triggers**

As the name suggests these are the triggers which depend on DML statements such as Update, Insert or Delete and they get fired either before or after them. Using DML trigger you can control the behavior of your DML statements. You can audit, check, replace or save values before they are changed. Automatic Increment of your Numeric primary key is one of the most frequent tasks of these types of triggers.

1. **Data Definition Language Triggers or DDL triggers.**

Again as the name suggests these are the type of triggers which are created over DDL statements such as CREATE or ALTER and get fired either before or after execution of your DDL statements. Using this type of trigger you can monitor the behavior and force rules on your DDL statements.

1. **System or Database Event triggers.**

Third type of triggers is system or database triggers. These are the type of triggers which come into action when some system event occurs such as database log on or log off. You can use these triggers for auditing purposes for example keeping an eye on information of system access like say who connects with your database and when. Most of the time System or Database Event triggers work as Swiss Knife for DBAs and help them in increasing the security of the data.

1. **Instead-of Trigger**

This is a type of trigger which enables you to stop and redirect the performance of a DML statement. Often this type of trigger helps you in managing the way you write to non-updatable views. You can also see the application of business rules by INSTEAD OF triggers where they insert, update or delete rows directly in tables that are defining updatable views. Alternatively, sometimes the INSTEAD OF triggers are also seen inserting, updating or deleting rows in designated tables that are otherwise unrelated to the view.

1. **Compound triggers**

These are multi-tasking triggers that act as both statement as well as row-level triggers when the data is inserted, updated or deleted from a table. You can capture information at four timing points using this trigger:   
(a) before the firing statement;   
(b) prior to change of each row from the firing statement;   
(c) post each row changes from the firing statement;   
(d) after the firing statement.   
All these types of triggers can be used to audit, check, save and replace the values even before they are changed right when there is a need to take action at the statement as well as row event levels.

**Syntax**

 CREATE [OR REPLACE] TRIGGER Ttrigger\_name  
 {BEFORE|AFTER} Triggering\_event ON table\_name  
 [FOR EACH ROW]  
 [FOLLOWS another\_trigger\_name]  
 [ENABLE/DISABLE]  
 [WHEN condition]  
 DECLARE  
  declaration statements  
 BEGIN  
  executable statements  
 EXCEPTION  
  exception-handling statements  
 END;

**Uses of triggers.**

1. Using triggers we can enforce business rules that can’t be defined by using integrity constants.
2. Using triggers we can gain strong control over the security.
3. We can also collect statistical information on the table access.
4. We can automatically generate values for derived columns such as auto increment numeric primary key.
5. Using triggers you can prevent invalid transaction.

**Restriction on Triggers**

1. Maximum size of the trigger body must not exceed 32,760 bytes because triggers’ bodies are stored in LONG datatypes columns.
2. A trigger may not issue transaction control statements or TCL statements such as COMMIT, ROLLBACK or SAVEPOINT. All operations performed when the trigger fires, become part of a transaction. Therefore whenever this transaction is rolled back or committed it leads to the respective rolling back or committing of the operations performed.
3. Any function or procedure called by a trigger may not issue a transactional control statement unless it contains an autonomous transaction.
4. Declaring LONG or LONG RAW variable is not permissible in the body of the trigger.

**Pseduo columns**

**:new** :- allows to access a row currently being processed. In other words , when a row is being inserted or updated into emp table.

**:old** :- allows to access a row which is already being either updated or deleted from emp table.

**Example**

create or replace trigger trig\_emp\_insert

before **insert** on emp

for each row

declare

v\_user varchar2(100);

begin

select user

into v\_user

from dual;

dbms\_output.put\_line('user '||v\_user||'has inserted data to emp table');

end;

similarly update and delete also we can perform

create or replace trigger trig\_emp\_insert

before **update** on emp

for each row

declare

v\_user varchar2(100);

begin

select user

into v\_user

from dual;

dbms\_output.put\_line('user '||v\_user||'has updated data to emp table');

end;

**we can perform all three in single DML statement**

CREATE OR REPLACE TRIGGER trig\_emp\_insert

BEFORE INSERT OR UPDATE OR DELETE ON employees

FOR EACH ROW

DECLARE

v\_user VARCHAR2(100);

BEGIN

SELECT USER

INTO v\_user

FROM dual;

IF INSERTING THEN

DBMS\_OUTPUT.PUT\_LINE('user '||v\_user||'has inserted data to emp table');

ELSIF UPDATING THEN

DBMS\_OUTPUT.PUT\_LINE('user '||v\_user||'has updated data to emp table');

ELSIF DELETING THEN

DBMS\_OUTPUT.PUT\_LINE('user '||v\_user||'has updated data to emp table');

END IF;

END;

**Statement level trigger**

CREATE OR REPLACE TRIGGER trg\_emp\_instead

AFTER UPDATE ON employee

DECLARE

PRAGMA AUTONOMOUS\_TRANSACTION;

BEGIN

INSERT INTO emp\_hist(action,action\_Date)

VALUES(‘UPDATING’,sysdate);

COMMIT;

END;

/

**Using WHEN Clause**

CREATE OR REPLACE TRIGGER trg\_emp\_sal\_ud

BEFORE UPDATE OF SALARY ON employee

FOR EACH ROW

WHEN(new.salary<1000)

BEGIN

RAISE\_APPLICATION\_ERROR(-20001,’Salary Is Less Than 1000’);

END;

UPDATE employee SET salary=970 WHERE empno=7900;

ORA-20001 Salary Is Less Than 1000

**Error While executing trigger** trg\_emp\_sal\_ud

Now we will capture data which is performed on employees table for audit.

Table to capture data

CREATE TABLE emp\_audit

(new\_value VARCHAR2(1000),

old\_value VARCHAR2(1000),

user\_name VARCHAR2(100),

entry\_date DATE,

operation VARCHAR2(200)

);

create or replace TRIGGER trig\_Emp\_audit

BEFORE INSERT OR UPDATE OR DELETE ON employees

FOR EACH ROW

DECLARE

v\_user VARCHAR2(100);

v\_date DATE;

l\_error\_msg VARCHAR2(4000);

BEGIN

l\_error\_msg:= 'Error While Selecting Data Of User And Date';

SELECT user , sysdate

INTO v\_user,v\_date

FROM dual;

l\_error\_msg:= 'Error While Capturing Data For Audit';

IF INSERTING THEN

INSERT INTO emp\_audit(new\_value,old\_value,user\_name,entry\_date,operation) VALUES (:new.employee\_id,null,v\_user,v\_date,'Insert');

ELSIF UPDATING THEN

INSERT INTO emp\_audit(new\_value,old\_value,user\_name,entry\_date,operation) VALUES (:new.employee\_id,:old.employee\_id,v\_user,v\_date,'Update');

ELSIF DELETING THEN

INSERT INTO emp\_audit(new\_value,old\_value,user\_name,entry\_date,operation) VALUES (null,:old.employee\_id,v\_user,v\_date,'Delete');

END IF;

EXCEPTION

WHEN OTHERS THEN

l\_Error\_msg:= substr(l\_error\_msg||sqlcode||sqlerrm,1,3000);

RAISE\_APPLICATION\_ERROR(-20001,l\_error\_msg);

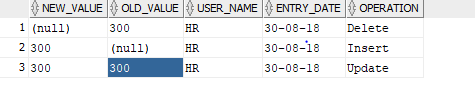
END;

/

delete from EMPLOYEES where EMPLOYEE\_ID>=300;

insert into employees values(300,'ramya','hk','ramya','515.123.4567',sysdate-100,'IT\_PROG',80000,null,100,90);

update employees set salary=9000 where EMPLOYEE\_ID=300;



**DDL Triggers**

**Using DDL triggers we can track changes to database.**

CREATE TABLE schema\_audit

(

ddl\_date DATE,

ddl\_user VARCHAR2(100),

object\_Created VARCHAR2(50),

object\_name VARCHAR2(50),

ddl\_operation VARCHAR2(100)

);

CREATE TABLE schema\_audit

(

ddl\_date DATE,

ddl\_user VARCHAR2(100),

object\_Created VARCHAR2(50),

object\_name VARCHAR2(50),

ddl\_operation VARCHAR2(100)

);

CREATE OR REPLACE TRIGGER trig\_hr\_audit

AFTER DDL ON SCHEMA

BEGIN

INSERT INTO schema\_audit

VALUES

(sysdate,

sys\_context('USERENV','CURRENT\_USER'),

ora\_dict\_obj\_type,

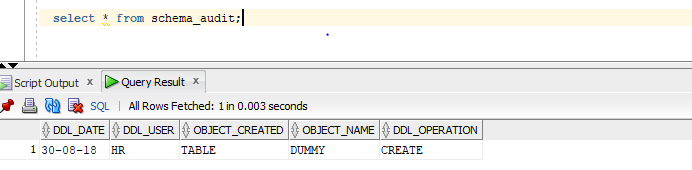
ora\_dict\_obj\_name,

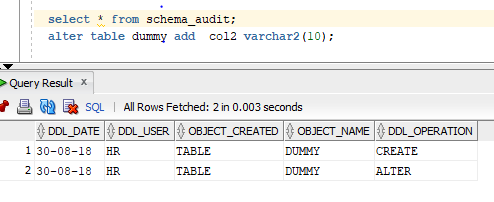
ora\_sysevent

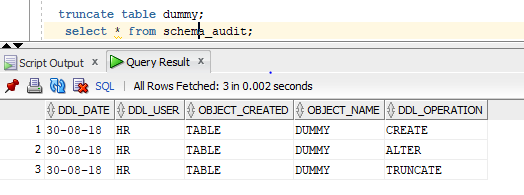
);

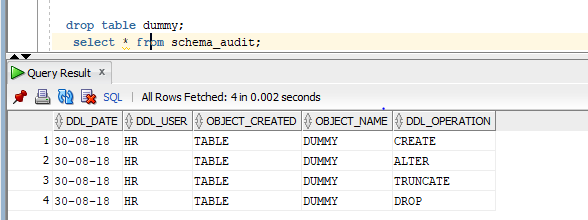
end;

create table dummy(col1 number);









In case if we need to track on truncate or drop etc then we can

create or replace TRIGGER trig\_hr\_audit

AFTER **truncate or alter** ON SCHEMA

BEGIN

INSERT INTO schema\_audit

VALUES

(sysdate,

sys\_context('USERENV','CURRENT\_USER'),

ora\_dict\_obj\_type,

ora\_dict\_obj\_name,

ora\_sysevent

);

end;

above will capture only schema level , if we need to capture database level then

create or replace TRIGGER trig\_hr\_audit

AFTER DDL ON **databse**

BEGIN

INSERT INTO schema\_audit

VALUES

(sysdate,

sys\_context('USERENV','CURRENT\_USER'),

ora\_dict\_obj\_type,

ora\_dict\_obj\_name,

ora\_sysevent

);

end;

**Database event triggers**

Database event triggers come into action when some system event occurs such as

* Database log on
* Log off
* Start up
* Shut down

Database event triggers can be created to monitor the system even activities of either a specific

* User or
* A Whole database

Syntax

CREATE or REPLACE TRIGGER trigger\_name

BEFORE |AFTER databse\_Event ON database/schema

BEGIN

Pl/sql code

END;

Example

To create logon trigger

CREATE TABLE hr\_Event\_Audit

(

event\_type VARCHAR2(30),

logon\_Date DATE,

logon\_time VARCHAR2(30),

logof\_date DATE,

logof\_time VARCHAR2(30)

);

CREATE OR REPLACE TRIGGER trig\_logon\_Audit

AFTER LOGON on schema / **AFTER LOGON on database(for database level)**

BEGIN

INSERT INTO hr\_Event\_audit

VALUES

(ora\_sysevent,

sysdate,

to\_char(sysdate,'hh24:mi:ss'),

null,

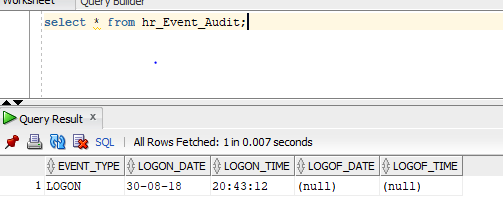
null

);

end;

now I am disconnecting from HR schema

and connected again



For log off

CREATE OR REPLACE TRIGGER trig\_logoff\_Audit

BEFORE LOGOFF on schema

BEGIN

INSERT INTO hr\_Event\_audit

VALUES

(ora\_sysevent,

null,

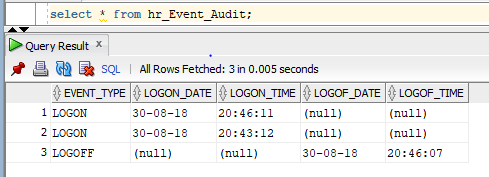
null,

sysdate,

to\_char(sysdate,'hh24:mi:ss')

);

end;



Startup and shutdown triggers

CREATE OR REPLACE TRIGGER trig\_startup\_Audit

AFTER STARTUP on database

BEGIN

INSERT INTO startup\_audit

VALUES

(ora\_sysevent,

sysdate,

to\_char(sysdate,'hh24:mi:ss')

);

end;

/

CREATE OR REPLACE TRIGGER trig\_shutdown\_Audit

BEFORE shutdown on database

BEGIN

INSERT INTO startup\_audit

VALUES

(ora\_sysevent,

sysdate,

to\_char(sysdate,'hh24:mi:ss')

);

end;

/

**INSTEAD OF TRIGGER**

Using instead of trigger we can control the default behavior of insert ,update,delete and merge operations on views but not on tables.

**Use of instead of trigger**

* We can use them to make a non-updatable view updatable
* Override the default behavior of views that are updatable

CREATE [OR REPLACE ] TRIGGER trigger\_name

INSTEAD OF operation

On view\_name

FOR each row

Begin

Plsql code

End;

create table trainer(full\_name varchar2(100));

create table subject(subject\_name varchar2(100));

create or replace view vw\_subject

as

select full\_name,subject\_name

from trainer,subject;

trying to insert to view

insert into vw\_subject values('Ramya','oracle');

QL Error: ORA-01779: cannot modify a column which maps to a non key-preserved table

01779. 00000 - "cannot modify a column which maps to a non key-preserved table"

\*Cause: An attempt was made to insert or update columns of a join view which

map to a non-key-preserved table.

\*Action: Modify the underlying base tables directly.

We can do It by using instead of trigger

create or replace trigger trig\_train\_sub

instead of insert on vw\_subject

for each row

begin

insert into trainer values(:new.full\_name);

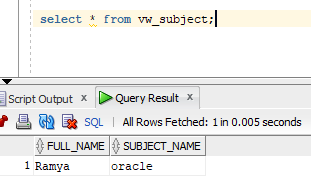
insert into subject values(:new.subject\_name);

end;

now same insert

insert into vw\_subject values('Ramya','oracle');

I row inserted



**Instead of update**

create or replace trigger trig\_train\_sub

instead of update on vw\_subject

for each row

begin

update trainer set full\_name=:new.full\_name where full\_name=:old.full\_name;

update subject set subjecT\_name =:new.subject\_name where subject\_name=:old.subject\_name;

end;

/

**Instead of delete**

create or replace trigger trig\_train\_sub

instead of delete on vw\_subject

for each row

begin

delete from trainer where full\_name=:old.full\_name;

delete from subject where subject\_name=:old.subject\_name;

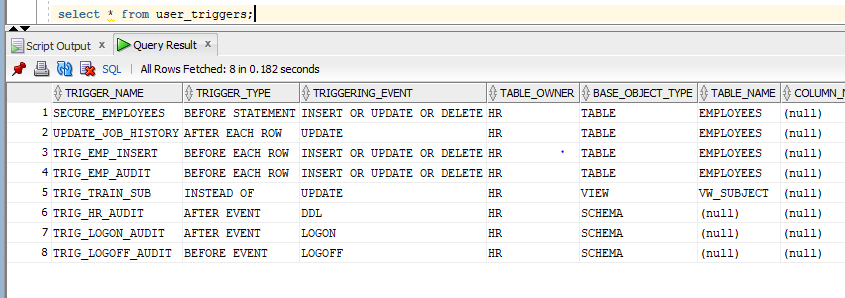
end;

/

DD(data dictionary)

User\_triggers

All\_triggers



select \* from all\_source where type='TRIGGER';

to drop trigger

drop trigger trigger\_name;

to get code of trigger

select

'create or replace '|| listagg (text, ' ')

WITHIN GROUP

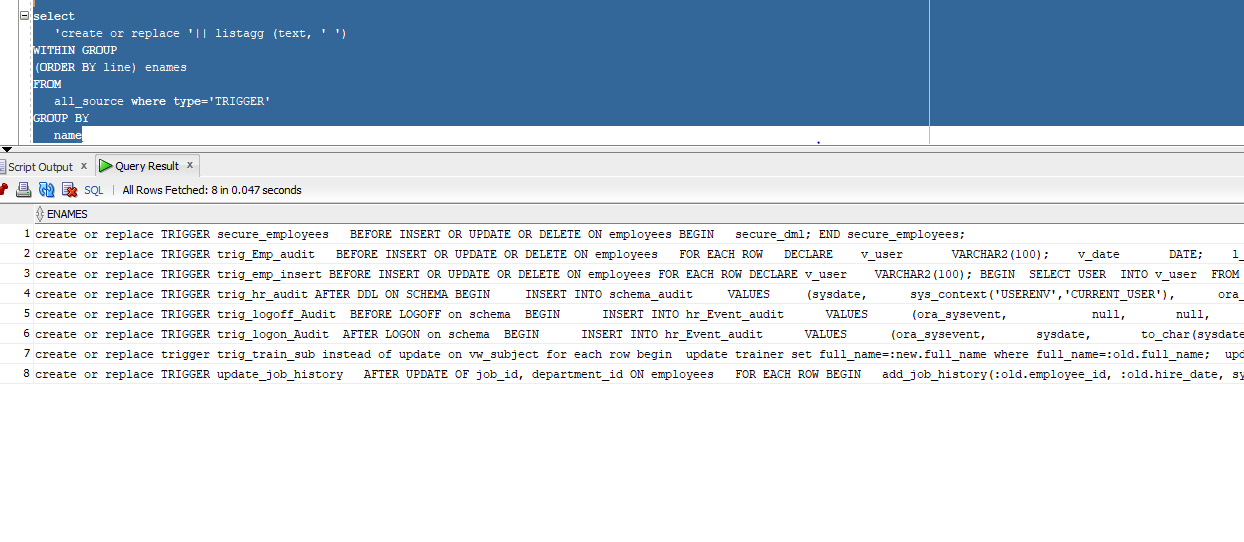
(ORDER BY line) enames

FROM

all\_source where type='TRIGGER'

GROUP BY

Name



If trigger is create on one table and if u try to replace it with another table name then will get error

Example

Below trigger is on employees table ,

create or replace TRIGGER trig\_emp\_insert

BEFORE INSERT OR UPDATE OR DELETE ON **employees**

FOR EACH ROW

DECLARE

v\_user VARCHAR2(100);

BEGIN

SELECT USER

INTO v\_user

FROM dual;

IF INSERTING THEN

DBMS\_OUTPUT.PUT\_LINE('user '||v\_user||'has inserted data to emp table');

ELSIF UPDATING THEN

DBMS\_OUTPUT.PUT\_LINE('user '||v\_user||'has updated data to emp table');

ELSIF DELETING THEN

DBMS\_OUTPUT.PUT\_LINE('user '||v\_user||'has updated data to emp table');

END IF;

END;

Now trying to replace same for emp table

create or replace TRIGGER trig\_emp\_insert

BEFORE INSERT OR UPDATE OR DELETE ON **emp**

FOR EACH ROW

DECLARE

v\_user VARCHAR2(100);

BEGIN

SELECT USER

INTO v\_user

FROM dual;

IF INSERTING THEN

DBMS\_OUTPUT.PUT\_LINE('user '||v\_user||'has inserted data to emp table');

ELSIF UPDATING THEN

DBMS\_OUTPUT.PUT\_LINE('user '||v\_user||'has updated data to emp table');

ELSIF DELETING THEN

DBMS\_OUTPUT.PUT\_LINE('user '||v\_user||'has updated data to emp table');

END IF;

END;

**ORA-04095: trigger 'TRIG\_EMP\_INSERT' already exists on another table, cannot replace it**

**04095. 00000 - "trigger '%s' already exists on another table, cannot replace it"**

**\*Cause: Cannot replace a trigger which already exists on a different table**

**than the one being replaced.**

**\*Action: Drop the trigger with the same name and re-create it.**

**To enable and disable triggers**

Alter trigger trigger\_name enable;

Alter trigger trigger\_name disable;

To enable and disable all triggers on table

Alter table table\_name enable all triggers;

Alter table table\_name disable all triggers;

**compound trigger**

A compound trigger is a single trigger on a table that enables you to specify actions for each of four timing points:

1. Before the firing statement
2. Before each row that the firing statement affects
3. After each row that the firing statement affects
4. After the firing statement

With the compound trigger, both the statement-level and row-level action can be put up in a single trigger. Plus there is an added advantage: it allows sharing of common state between all the trigger-points using variable. This is because compound trigger in oracle 11g has a declarative section where one can declare variable to be used within trigger. This common state is established at the start of triggering statement and is destroyed after completion of trigger (regardless of trigger being in error or not). If same had to be done without compound-trigger, it might have been required to share data using packages.

## When to use Compound Triggers

The compound trigger is useful when you want to accumulate facts that characterize the “for each row” changes and then act on them as a body at “after statement” time. Two popular reasons to use compound trigger are:

1. To accumulate rows for bulk-insertion
2. To avoid the infamous ORA-04091: mutating-table error.

## Syntax

CREATE OR REPLACE TRIGGER compound\_trigger\_name

FOR [INSERT|DELETE]UPDATE [OF column] ON table

COMPOUND TRIGGER

-- Declarative Section (optional)

-- Variables declared here have firing-statement duration.

--Executed before DML statement

BEFORE STATEMENT IS

BEGIN

NULL;

END BEFORE STATEMENT;

--Executed before each row change- :NEW, :OLD are available

BEFORE EACH ROW IS

BEGIN

NULL;

END BEFORE EACH ROW;

--Executed aftereach row change- :NEW, :OLD are available

AFTER EACH ROW IS

BEGIN

NULL;

END AFTER EACH ROW;

--Executed after DML statement

AFTER STATEMENT IS

BEGIN

NULL;

END AFTER STATEMENT;

END compound\_trigger\_name;

## Some Restriction/Catches to note

1. The body of a compound trigger must be a compound trigger block.
2. A compound trigger must be a DML trigger.
3. A compound trigger must be defined on either a table or a view.
4. The declarative part cannot include PRAGMA AUTONOMOUS\_TRANSACTION.
5. A compound trigger body cannot have an initialization block; therefore, it cannot have an exception section. This is not a problem, because the BEFORE STATEMENT section always executes exactly once before any other timing-point section executes.
6. An exception that occurs in one section must be handled in that section. It cannot transfer control to another section.
7. If a section includes a GOTO statement, the target of the GOTO statement must be in the same section.
8. OLD, :NEW, and :PARENT cannot appear in the declarative part, the BEFORE STATEMENT section, or the AFTER STATEMENT section.
9. Only the BEFORE EACH ROW section can change the value of :NEW.
10. If, after the compound trigger fires, the triggering statement rolls back due to a DML exception:
    * Local variables declared in the compound trigger sections are re-initialized, and any values computed thus far are lost.
    * Side effects from firing the compound trigger are not rolled back.
11. The firing order of compound triggers is not guaranteed. Their firing can be interleaved with the firing of simple triggers.
12. If compound triggers are ordered using the FOLLOWS option, and if the target of FOLLOWS does not contain the corresponding section as source code, the ordering is ignored.

## Example: Using Compound Triggers in Table Auditing

Hopefully this example with make things more clear. Lets create a compound trigger for auditing a large table called ’employees’. Any changes made in any field of ’employees’ table needs to be logged in as a separate row in audit table ‘aud\_empl’.  
Since each row update in employees table needs to make multiple inserts in the audit table, we should consider using a compound trigger so that batching of inserts can be performed.

But before that we need to create our Tables:

--Target Table

CREATE TABLE employees(

emp\_id varchar2(50) NOT NULL PRIMARY KEY,

name varchar2(50) NOT NULL,

salary number NOT NULL

);

--Audit Table

CREATE TABLE aud\_emp(

upd\_by varchar2(50) NOT NULL,

upd\_dt date NOT NULL,

field varchar2(50) NOT NULL,

old\_value varchar2(50) NOT NULL,

new\_value varchar2(50) NOT NULL);

Now the trigger…  
On update of each row instead of performing an insert operation for each field, we store (buffer) the required attributes in a Arrays of type aud\_emp. Once a threshold is reached (say 1000 records), we flush the buffered data into audit table and reset the counter for further buffering.  
And at last, as part of AFTER STATEMENT we flush any remaining data left in buffer.

--Trigger

CREATE OR REPLACE TRIGGER aud\_emp

FOR INSERT OR UPDATE

ON employees

COMPOUND TRIGGER

TYPE t\_emp\_changes IS TABLE OF aud\_emp%ROWTYPE INDEX BY SIMPLE\_INTEGER;

v\_emp\_changes t\_emp\_changes;

v\_index SIMPLE\_INTEGER := 0;

v\_threshhold CONSTANT SIMPLE\_INTEGER := 1000; --maximum number of rows to write in one go.

v\_user VARCHAR2(50); --logged in user

PROCEDURE flush\_logs

IS

v\_updates CONSTANT SIMPLE\_INTEGER := v\_emp\_changes.count();

BEGIN

FORALL v\_count IN 1..v\_updates

INSERT INTO aud\_emp

VALUES v\_emp\_changes(v\_count);

v\_emp\_changes.delete();

v\_index := 0; --resetting threshold for next bulk-insert.

END flush\_logs;

AFTER EACH ROW

IS

BEGIN

IF INSERTING THEN

v\_index := v\_index + 1;

v\_emp\_changes(v\_index).upd\_dt := SYSDATE;

v\_emp\_changes(v\_index).upd\_by := SYS\_CONTEXT ('USERENV', 'SESSION\_USER');

v\_emp\_changes(v\_index).emp\_id := :NEW.emp\_id;

v\_emp\_changes(v\_index).action := 'Create';

v\_emp\_changes(v\_index).field := '\*';

v\_emp\_changes(v\_index).from\_value := 'NULL';

v\_emp\_changes(v\_index).to\_value := '\*';

ELSIF UPDATING THEN

IF ( (:OLD.EMP\_ID <> :NEW.EMP\_ID)

OR (:OLD.EMP\_ID IS NULL AND :NEW.EMP\_ID IS NOT NULL)

OR (:OLD.EMP\_ID IS NOT NULL AND :NEW.EMP\_ID IS NULL)

)

THEN

v\_index := v\_index + 1;

v\_emp\_changes(v\_index).upd\_dt := SYSDATE;

v\_emp\_changes(v\_index).upd\_by := SYS\_CONTEXT ('USERENV', 'SESSION\_USER');

v\_emp\_changes(v\_index).emp\_id := :NEW.emp\_id;

v\_emp\_changes(v\_index).field := 'EMP\_ID';

v\_emp\_changes(v\_index).from\_value := to\_char(:OLD.EMP\_ID);

v\_emp\_changes(v\_index).to\_value := to\_char(:NEW.EMP\_ID);

v\_emp\_changes(v\_index).action := 'Update';

END IF;

IF ( (:OLD.NAME <> :NEW.NAME)

OR (:OLD.NAME IS NULL AND :NEW.NAME IS NOT NULL)

OR (:OLD.NAME IS NOT NULL AND :NEW.NAME IS NULL)

)

THEN

v\_index := v\_index + 1;

v\_emp\_changes(v\_index).upd\_dt := SYSDATE;

v\_emp\_changes(v\_index).upd\_by := SYS\_CONTEXT ('USERENV', 'SESSION\_USER');

v\_emp\_changes(v\_index).emp\_id := :NEW.emp\_id;

v\_emp\_changes(v\_index).field := 'NAME';

v\_emp\_changes(v\_index).from\_value := to\_char(:OLD.NAME);

v\_emp\_changes(v\_index).to\_value := to\_char(:NEW.NAME);

v\_emp\_changes(v\_index).action := 'Update';

END IF;

IF ( (:OLD.SALARY <> :NEW.SALARY)

OR (:OLD.SALARY IS NULL AND :NEW.SALARY IS NOT NULL)

OR (:OLD.SALARY IS NOT NULL AND :NEW.SALARY IS NULL)

)

THEN

v\_index := v\_index + 1;

v\_emp\_changes(v\_index).upd\_dt := SYSDATE;

v\_emp\_changes(v\_index).upd\_by := SYS\_CONTEXT ('USERENV', 'SESSION\_USER');

v\_emp\_changes(v\_index).emp\_id := :NEW.emp\_id;

v\_emp\_changes(v\_index).field := 'SALARY';

v\_emp\_changes(v\_index).from\_value := to\_char(:OLD.SALARY);

v\_emp\_changes(v\_index).to\_value := to\_char(:NEW.SALARY);

v\_emp\_changes(v\_index).action := 'Update';

END IF;

END IF;

IF v\_index >= v\_threshhold THEN

flush\_logs();

END IF;

END AFTER EACH ROW;

-- AFTER STATEMENT Section:

AFTER STATEMENT IS

BEGIN

flush\_logs();

END AFTER STATEMENT;

END aud\_emp;

/

INSERT INTO employees VALUES (1, 'emp1', 10000);

INSERT INTO employees VALUES (2, 'emp2', 20000);

INSERT INTO employees VALUES (3, 'emp3', 16000);

UPDATE employees

SET salary = 2000

WHERE salary > 15000;

SELECT \* FROM aud\_emp;

Result:

EMP\_ID,UPD\_BY,UPD\_DT,ACTION,FIELD,FROM\_VALUE,TO\_VALUE

1,Aditya,1/22/2014 10:59:33 AM,Create,\*,NULL,\*

2,Aditya,1/22/2014 10:59:34 AM,Create,\*,NULL,\*

3,Aditya,1/22/2014 10:59:35 AM,Create,\*,NULL,\*

2,Aditya,1/22/2014 10:59:42 AM,Update,SALARY,20000,2000

3,Aditya,1/22/2014 10:59:42 AM,Update,SALARY,16000,2000

Now any changes in any field of employees will to be written in aud\_emp table. A beauty of this approach is we were able to access same data ‘v\_emp\_changes’ between statement and row triggering events.

With this in mind, one can see that it make sense to move v\_emp\_changes(v\_index).upd\_by := SYS\_CONTEXT ('USERENV', 'SESSION\_USER'); inside declarative(or BEFORE STATEMENT if complex computation) section as a pre-processing step. To do so, v\_user variable declared in trigger body can be used and assigned value of logged in user in the declarative section itself. So that same computation is not made during after-each-row section, and is computed and stored in a variable just once before row-level execution begins.

--declarative section

v\_user VARCHAR2(50) := SYS\_CONTEXT ('USERENV', 'SESSION\_USER');

Similarly any such pre-processing if required can be performed on that source table (mutating table), doing so will avoid any possible mutating-error. For e.g., consider the same example with another restriction, “Any update of salary should be such that it is not less than 1/12th of maximum salary of any employee, or else an error is raised”. To do this, it will be needed to get the maximum value of salary in the ’employees’ table, and such calculation can be made in BEFORE STATEMENT section and stored in variable.