1 Triggers

Triggers are simply *stored procedures* that are ran automatically by the database whenever some event (usually a table update) happens. We won't spend a great deal of time talking about *how to* write triggers, because if you know how to write stored procedures, you already know how to write triggers. What we will discuss is how to set them up to be executed whenever something happens to the table.

2 PL/SQL Triggers

Triggers are basically PL/SQL procedures that are associated with tables, and are called whenever a certain modification (event) occurs. The modification statements may include `INSERT`, `UPDATE`, and `DELETE`.

The general structure of triggers is:

```
CREATE [OR REPLACE]
TRIGGER trigger_name
BEFORE (or AFTER)
INSERT OR UPDATE [OF COLUMNS] OR DELETE
ON tablename
[FOR EACH ROW [WHEN (condition)]]
BEGIN
...
END;
```

The usual `CREATE OR REPLACE` we have already seen with procedures and functions... `TRIGGER` specifies just what type of object we are creating.

The `BEFORE` (or `AFTER`) in the trigger definition refers to when you want to run the trigger, either before the actual database modification (update, delete, insert) or after.

The list of various statements, `INSERT OR UPDATE [OF COLUMNS] OR DELETE` refers to statements that trigger this trigger. You can specify all three, or just one. Let's say you wanted the trigger to fire only when you do a delete; well, then you'd only specify a `DELETE` in the list.

On some table specifies that the trigger is associated with such table. As we shall see later, this does not necessarily has to be a *table*, but could also be a *view*.

There are several types of triggers; ones *for each row* and others *per statement*. For example, when you're doing an update, you can have a trigger fire once for each thing being updated (if you update 20 rows, the thing would fire 20 times), or you can have it fire just once per statement (if a single update statement is updating 20 rows, the trigger would fire just once). This is what that `FOR EACH ROW` in the trigger definition means.

The PL/SQL block (between `BEGIN` and `END`) is a usual code block where you can place PL/SQL commands. The only limitation is that you cannot use `COMMIT` (or `ROLLBACK`) for obvious reasons.

2.1 Permissions

Just like with procedures and functions, creating triggers requires certain privileges which are not part of the default privilege set. If you cannot create triggers from these notes because of permissions, you (or the admin) has to `GRANT CREATE TRIGGER` privilege on your username.

For example, to allow user 'alex' to create triggers, I may do something like this:

```
GRANT CREATE TRIGGER TO alex;
```

Note that if you are accessing a public Oracle server you must ask the admin to setup these things for you.

2.2 Sample Table to be Triggered

Before we begin playing with triggers, let's create a simple table with which we can experiment:

```
CREATE TABLE PERSON (
    ID   INT,
    NAME VARCHAR(30),
    DOB  DATE,
    PRIMARY KEY(ID)
);
```

The above creates a `PERSON` table with an `ID`, a `NAME` and a `DOB` columns (fields). Whatever triggers we define in these notes will relate to this table.

Also, let's not forget to setup: `SET SERVEROUTPUT ON;`

2.3 Before Insert Trigger

Let's start out our quest to learn triggers with the simplest case. We have nothing in the database (our `PERSON` table is empty). Before we insert any data, we'd like to perform some operation (let's say for logging purposes, or whatever). We write a trigger to fire *before* the insert takes place.

```
CREATE OR REPLACE
TRIGGER PERSON_INSERT_BEFORE
BEFORE
INSERT
ON PERSON
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('BEFORE INSERT OF ' || :NEW.NAME);
END;
```

Now let us test it out:

```
INSERT INTO PERSON(ID,NAME,DOB) VALUES (1,'JOHN DOE',SYSDATE);
```

The single INSERT statement fires the trigger. When we run it, we get the print out of 'BEFORE INSERT OF JOHN DOE'. Ie:

```
SQL> INSERT INTO PERSON(ID,NAME,DOB) VALUES (1,'JOHN DOE',SYSDATE);
BEFORE INSERT OF JOHN DOE

1 row created.
```

2.4 After Insert Trigger

Can you guess what the trigger would look like that would fire AFTER the insert? Well?

```
CREATE OR REPLACE
TRIGGER PERSON_INSERT_AFTER
AFTER
INSERT
ON PERSON
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('AFTER INSERT OF ' || :NEW.NAME);
END;
```

And with our 2nd test INSERT:

```
INSERT INTO PERSON(ID,NAME,DOB) VALUES (2,'JANE DOE',SYSDATE);
```

For a total result of:

```
SQL> INSERT INTO PERSON(ID,NAME,DOB) VALUES (2,'JANE DOE',SYSDATE);
BEFORE INSERT OF JANE DOE
AFTER INSERT OF JANE DOE

1 row created.
```

Notice that *both* triggers have fired. One *before* the INSERT the other one *after*.

2.5 Before Update Statement Trigger

Now that we have some data in the table, we can create an update trigger, that would fire whenever someone tries to update any person (or persons).

```
CREATE OR REPLACE
TRIGGER PERSON_UPDATE_S_BEFORE
BEFORE UPDATE
ON PERSON
BEGIN
    DBMS_OUTPUT.PUT_LINE('BEFORE UPDATING SOME PERSON(S)');
END;
```

Now, let's run an update...

```
UPDATE PERSON SET DOB = SYSDATE;
```

Which produces the result:

```
SQL> UPDATE PERSON SET DOB = SYSDATE;
BEFORE UPDATING SOME PERSON(S)

2 rows updated.
```

Note that is says `2 rows updated` but we've only seen one `BEFORE UPDATING SOME PERSON(S)`, meaning that our trigger only fired once. This is because we did not specify `FOR EACH ROW` (which we'll do next).

Btw, from now on, we'll leave out a few details (I'll assume you can figure out how to write `PERSON_UPDATE_S_BEFORE` trigger, and such, etc.)

2.6 `FOR EACH ROW` Before Update Trigger

Right now, all we are doing is adding a `FOR EACH ROW` to last example:

```
CREATE OR REPLACE
TRIGGER PERSON_UPDATE_BEFORE
BEFORE UPDATE
ON PERSON
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('BEFORE UPDATING ' ||
       TO_CHAR(:OLD.DOB,'HH:MI:SS') || ' TO ' ||
       TO_CHAR(:NEW.DOB,'HH:MI:SS'));
END;
```

We're also printing out (displaying) the old value of `PERSON.DOB` and the new value. Now, let's run our update statement:

```
UPDATE PERSON SET DOB = SYSDATE;
```

Which gives the results:

```
SQL> UPDATE PERSON SET DOB = SYSDATE;
BEFORE UPDATING SOME PERSON(S)
BEFORE UPDATING 10:54:06 TO 11:10:01
BEFORE UPDATING 10:54:06 TO 11:10:01

2 rows updated.
```

Notice that we still get the firing of the initial 'non' per row trigger (that's the first one), then the `FOR EACH ROW` trigger fires, which actually does run for each row that is updated (in this case, twice).

2.7 Special IF statements

Inside the PL/SQL block of a trigger we can use if statements to determine what statement caused the firing of the trigger. These are generally of the form: `IF inserting THEN...` where besides "inserting" you can also use `updating` and `deleting`. An example would be something like:

```
CREATE OR REPLACE
TRIGGER PERSON_BIUD
BEFORE INSERT OR UPDATE OR DELETE ON PERSON
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE('INSERTING PERSON: ' || :NEW.NAME);
    ELSIF UPDATING THEN
        DBMS_OUTPUT.PUT_LINE('UPDATING PERSON: ' ||
            :OLD.NAME || ' TO ' || :NEW.NAME);
    ELSIF DELETING THEN
        DBMS_OUTPUT.PUT_LINE('DELETING PERSON: ' || :OLD.NAME);
    END IF;
END;
```

Notice that we only have one `TRIGGER`, and we are using `IF` statements to determine what statement invoked it, and display an appropriate message in various cases. For example, when we do an insert:

```
INSERT INTO PERSON(ID,NAME,DOB) VALUES (3,'SUPERMAN',TO_DATE('09/05/1950','MM/DD/YYYY'))
```

Then we get output like:

```
INSERTING PERSON: SUPERMAN
```

If we go ahead and modify that person:

```
UPDATE PERSON SET NAME = 'BATMAN' WHERE NAME = 'SUPERMAN';
```

Then we get an output like:

```
UPDATING PERSON: SUPERMAN TO BATMAN
```

And finally, if we go ahead and delete that person:

```
DELETE PERSON WHERE NAME = 'BATMAN';
```

Then we would get output like:

```
DELETING PERSON: BATMAN
```

Please note that you will have to run `SET SERVEROUTPUT ON;` in SQL*Plus order to see the output.

2.8 Working with Views

For our next example, we will need to create a view (of PERSON table):

```
CREATE OR REPLACE
VIEW PERSON_VIEW AS
SELECT NAME FROM PERSON;
```

Now, we know that updating (or inserting) into a view is kind of pointless; however, we can provide this functionality using a trigger! For example:

```
CREATE OR REPLACE
TRIGGER PERSON_VIEW_INSERT
INSTEAD OF INSERT ON PERSON_VIEW
FOR EACH ROW
BEGIN
    DBMS_OUTPUT.PUT_LINE('INSERTING: ' || :NEW.NAME);

    -- we can also do
    -- INSERT INTO PERSON(ID,NAME,DOB) VALUES (N,:NEW.NAME,SYSDATE);
END;
```

When we do an insert statement on `PERSON_VIEW`:

```
INSERT INTO PERSON_VIEW(NAME) VALUES ('SUPERMAN');
```

Which produces the result:

```
INSERTING: SUPERMAN
```

So, what did just happen??? Did we insert a value into a view? No, not really. What we did was fire a trigger when someone tried to insert a value into a `VIEW`.

Now, as the comment in the code indicates, we can actually simulate the insertion statement (by inserting the value into the `PERSON` table ourselves).

2.9 Trigger Exceptions (introduction)

Triggers become part of the transaction of a statement, which implies that it causes (or raises) any exceptions (which we'll talk about later), the whole statement is rolled back. Think of an exception as a flag that is raised when an error occurs. Sometimes, an error (or exception) is raised for a good reason. For example, to prevent some action that improperly modifies the database. Let's say that our database should not allow anyone to modify their `DOB` (after the person is in the database, their `DOB` is assumed to be static). Anyway, we can create a trigger that would prevent us from updating the `DOB`:

```
CREATE OR REPLACE
TRIGGER PERSON_DOB
BEFORE UPDATE OF DOB ON PERSON
FOR EACH ROW
```

```
BEGIN
    RAISE_APPLICATION_ERROR(-20000,'CANNOT CHANGE DATE OF BIRTH');
END;
```

Notice the format of the trigger declaration. We explicitly specify that it will be called `BEFORE UPDATE OF DOB ON PERSON`.

The next thing you should notice is the procedure call `RAISE_APPLICATION_ERROR`, which accepts an error code, and an explanation string. This effectively halts our trigger execution, and raises an error, preventing our `DOB` from being modified. An error (exception) in a trigger stops the code from updating the `DOB`.

When we do the actual update for example:

```
UPDATE PERSON SET DOB = SYSDATE;
```

We end up with an error, that says we `CANNOT CHANGE DATE OF BIRTH`.

```
SQL> UPDATE PERSON SET DOB = SYSDATE;
UPDATE PERSON SET DOB = SYSDATE
       *
ERROR at line 1:
ORA-20000: CANNOT CHANGE DATE OF BIRTH
ORA-06512: at "PARTICLE.PERSON_DOB", line 2
ORA-04088: error during execution of trigger 'PARTICLE.PERSON_DOB'
```

You should also notice the error code of `ORA-20000`. This is our `-20000` parameter to `RAISE_APPLICATION_ERROR`.

2.10 Viewing Triggers

You can see all your user defined triggers by doing a select statement on `USER_TRIGGERS`. For example:

```
SELECT TRIGGER_NAME FROM USER_TRIGGERS;
```

Which produces the names of all triggers. You can also select more columns to get more detailed trigger information. You can do that at your own leisure, and explore it on your own.

2.11 Dropping Triggers

You can `DROP` triggers just like anything. The general format would be something like:

```
DROP TRIGGER trigger_name;
```

2.12 Altering Triggers

If a trigger seems to be getting in the way, and you don't want to drop it, just disable it for a little while, you can alter it to disable it. Note that this is not the same as dropping a trigger; after you drop a trigger, it is gone.

The general format of an alter would be something like this:

```
ALTER TRIGGER trigger_name [ENABLE|DISABLE];
```

For example, let's say that with all our troubles, we still need to modify the DOB of 'JOHN DOE'. We cannot do this since we have a trigger on that table that prevents just that! So, we can disable it...

```
ALTER TRIGGER PERSON_DOB DISABLE;
```

Now, we can go ahead and modify the DOB :-)

```
UPDATE PERSON SET DOB = SYSDATE WHERE NAME = 'JOHN DOE';
```

We can then re-ENABLE the trigger.

```
ALTER TRIGGER PERSON_DOB ENABLE;
```

If we then try to do the same type of modification, the trigger kicks and prevents us from modifying the `DOB`.

3 T-SQL Triggers

T-SQL triggers are pretty much just like the PL/SQL triggers. For example, to raise an error when someone tries to update/insert records into `Person` table, you'd create something like:

```
CREATE TRIGGER sometrigname
ON [Person]
FOR INSERT, UPDATE
AS RAISERROR (50009, 16, 10)
```

The format is similar enough to PL/SQL that it doesn't need explanation. Obviously, after the `AS`, you can have a `BEGIN` and continue that with a whole huge procedure.

i.e.: Everything that you can do in PL/SQL, you can also do in T-SQL, except with possibly slightly different format.