Creating Database Triggers

Creating Database Triggers

- Learn About Database Triggers
- Develop Statement-Level Triggers
- Develop Row-Level Triggers
- Consider Several Trigger Examples
- Develop INSTEAD OF Triggers
- Learn to Employ Triggers Within Applications

What Is A Database Trigger?

A Database Trigger is a PL/SQL program stored within the database.

- Very similar to a PL/SQL code.
- Triggers are unique because of they fire as the result of a database event.

ⓘ This may sometimes be referred to as the trigger firing **timing point**.

Database Triggers Illustrated

The following demonstrates the timing of two different database triggers.

```
UPDATE employee
SET lname = 'Smythe'
WHERE lname = 'Smith';
```

ⓘ All table activity is done within the context of a single transaction.

Before Update Event

After Update Event

SSN	LName	Super_SSN	DNO
123456789	Smythe	333445555	5
453453453	English	333445555	5
…			
888665555	Borg		1

Cascading Triggers

As you can see, there can be many triggers associated with different DML events for a single table.

Just like a trigger may contain SQ her table in the database.

ⓘ At this time, Oracle is limited to 32 levels of cascading triggers.

In this situation, you can have one trigger event fire multiple different trigger events for other objects.

Trigger Types

There are several different types, and even subtypes of triggers.

The first and primary distinction between trigger types are:

- Triggers are attached to Table DML events
- Triggers are attached to View DML events
- Triggers are associated with predefined database system events.

Trigger Subtypes

Within the category of DML event triggers, there are the following subtypes of triggers:

- Statement-Level Triggers: These fire once for the entire triggering statement.
- Row-Level Triggers: These fire once for each row affected by the triggering statement.
- INSTEAD OF Triggers: Used to execute in place of any update operations on a particular View.

Database Trigger Example

```
CREATE OR REPLACE TRIGGER TriggerName
  TriggerEvent
  ON TableName

DECLARE
…
BEGIN
…
EXCEPTION
…
END;
```

- The Trigger Name must be unique
- The Trigger Event defines when the trigger will fire relative to the DML event.

Database Trigger Information

Triggers are unique in their purpose and when they are executed, but they are similar to every other program unit.

- Share the same space in the PL/SQL cache
- Can become invalid if changes are made to dependent objects
- Can invoke stored procedures

Using Triggers For Security

Oracle provides standard object security features used to restrict which users may access or the type of access to database objects.

But, what if we want to restrict access based on the Day and Time of their access?

Security Trigger Example

```
CREATE OR REPLACE TRIGGER security_time_check
  BEFORE DELETE OR UPDATE ON employee
DECLARE
  dy_of_week      CHAR(3);
  hh_of_day       NUMBER(2);
BEGIN
  --Set variables
  dy_
  hh_

  -Te
  IF
  OR hh_of_day      BETWEEN 8 AND 17 THEN
     RAISE_APPLICATION_ERROR(-20600,
         'Transaction rejected for security reasons');
  END IF;

END;
```

ⓘ Notice the RAISE_APPLICATION_ERROR() function.

Security Check SQL

```
UPDATE EMPLOYEE
SET SALARY = 70000
WHERE SSN = '123456789';
```

See It In Action

Statement-Level Triggers

Statement-Level Triggers fire once for the entire triggering statement.

- The system doesn't care how many rows are effected by the DML event.

The DML Event

- The Trigger Declaration allows you to specify when it should fire
  - BEFORE
  - AFTER
- It also specifies exactly which DML event should cause the execution
  - INSERT
  - UPDATE
  - DELETE

The DML Event (cont)

- Additionally, you can qualify the UPDATE specification to list individual columns within the table.
  - When omitted, any update will fire the Trigger
  - When included, only updates to those fields will fire the Trigger

```
CREATE OR REPLACE TRIGGER security_time_check
  BEFORE DELETE OR UPDATE OF Salary ON employee
DECLARE
  dy_of_week  CH
  hh_of_day   NU
BEGIN
…
EXCEPTION
…
END;
```

ⓘ This can greatly improve performance.

When To Use Trigger Events

The challenge when creating a Trigger is deciding how to use the feature as part of the overall design.

- BEFORE Trigger Event
  - Might be good for creating journal entries
  - Enforcing application security
- AFTER Trigger Event
  - Might be good for logging journal entries for successful operations

Clarifying The Event Within The Implementation

When the trigger has been defined for multiple DML events, the logic might require different actions depending upon which specific event fired it.

- INSERTING
- DELETING
- UPDATING
- UPDATING ('Column

ⓘ The UPDATING function includes the option to specify the column as a parameter.

Clarifying The Event Within The Implementation (Cont)

The Boolean functions might be referenced within the implementation logic as shown.

```
BEGIN
  IF INSERTING THEN
    …
  END IF;

  IF DELETING THEN
    …
  END IF;

  IF UPDATING ('Column1') THEN
    …
  END IF;
END;
```

Using RAISE_APPLICATION_ERROR()

The RAISE_APPLICATION_ERROR() system-supplied procedure may be invoked within your trigger.

The effect of calling this procedure for application errors are:

- A user-defined database error number and text is generated
- The trigger execution is aborted and the exception is raised
- If the exception is not handled and it persists, the transaction is rolled back to the automatic savepoint.

Using RAISE_APPLICATION_ERROR()

```
 --Test the day and time
  IF dy_of_week IN ('FRI', 'SUN')
     OR hh_of_day NOT BETWEEN 8 AND 17 THEN

    --Raise an application error to stop the process
    RAISE_APPLICATION_ERROR(-20600,
         'Transaction rejected for security reasons');

  END IF;
END;
```

Using RAISE_APPLICATION_ERROR()

The parameters available when calling the RAISE_APPLICATION_ERROR() procedure are:

Parameter	Description
user_error_number	May be any number between -20000 and -20999.
user_error_text	which is
keep_error_stack	Optional Boolean value which indicates whether the error message should be added to others within the stack. Default = FALSE.

ⓘ Several system-supplied packages use the error numbers from -20000 to -20005.

About The Error Stack

For you to better understand the Error Stack and the keep_error_stack parameter, review the code below:

```
BEGIN
  RAISE_APPLICATION_ERROR(-20000, 'Sample message');
EXCEPTION
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20001,
                            'Additional sample message', TRUE);
END;
/

BEGIN
*
ERROR at line 1:
ORA-20001: Additional sample message
ORA-06512: at line 5
ORA-20000: Sample message
```

See It In Action

Row-Level Triggers

Row-Level Triggers fire once for each row within a trigger statement.

- Indicated by the FOR EACH ROW clause
- Must consider the performance implications.

```
CREATE OR REPLACE TRIGGER TriggerName
  BEFORE | AFTER
  INSERT OR DELETE OR UPDATE OF Column1, ColumnN
  ON TableName

  FOR EACH ROW
  REFERENCING OLD AS OldName
              NEW AS NewName
  WHEN (condition expression)

DECLARE
…
```

Row-Level Triggers (cont)

- Features available to the Statement-Level Trigger are also available to the Row-Level Trigger
  - RAISE_APPLICATION_ERROR()
  - IF INSERTING OR DELETING OR UPDATING
- Row-Level Triggers offer additional capabilities
  - Track BEFORE and AFTER values
  - “Last Line of Defense”

Old & New Column Values

Since Row-Level triggers execute within the realm of a single row, they can examine old and new column values.

- Column values may be referenced using what is known as correlation name and the column name.

```
…
BEGIN
  IF UPDATING
  AND :new.salary > :old.salary THEN
    INSERT INTO budget_request(acct_no, amount, desc)
      VALUES (101, :new.salary - :old.salary, 'raise');
  ELSE
    INSERT INTO budget_request(acct_no, amount, desc)
      VALUES (101, :new.salary, 'New employee');
  END IF;
```

Reading OLD & NEW Values

There are some limitations on when the old or new values may be read.

Triggering Statement	Availability	
INSERT	New:	Available
	Old:	NULL
UPDATE	New:	Available
	Old:	Available
DELETE	New:	NULL
	Old:	Available

Correlation Names

OLD and NEW are the default correlation names.

- They may be overridden using the REFERENCING clause.
- When referenced in the trigger body, they MUST use the preceding colon.

ⓘ OLD and NEW values are essentially external variable references.

Improve Performance

Row-Level triggers permit a conditional expression to be include[illegible]n.

- Include [illegible] the trigger specificatio[illegible]

ⓘ Either OLD or NEW values may be referenced in the WHEN() expression.

- Much better than including in the body

```
CREATE OR REPLACE TRIGGER employee_journal
  AFTER INSERT OR UPDATE OF salary ON EMPLOYEE
  FOR EACH ROW
  WHEN (new.salary > 70000)
…
```

ⓘ Where is the colon?

Employee Salary Check Example

This example will only allow an employee salary to exceed $70,000 if he/she is a manager.

- This condition cannot be implemented with a simple declarative constraint.
- Application logic could perform this test before the update is ever processed.

See It In Action

```
CREATE OR REPLACE TRIGGER EMPLOYEE_SALARY_CHECK
  BEFORE INSERT OR UPDATE OF salary ON employee
  FOR EACH ROW
  WHEN (new.salary > 70000)
DECLARE

 --Declare variables
  x_mgrssn      department.mgrssn%TYPE;
  message_text  VARCHAR2(100);

BEGIN

  --Execute query to determine whether the ssn refers to a manager
        SELECT MGRSSN
        INTO X_MGRSSN
        FROM department
        WHERE mgrssn = :new.ssn;

EXCEPTION
        WHEN NO_DATA_FOUND THEN
    --Set the message text
    message_text := 'Must be manager for salary of ' ||
TO_CHAR(:NEW.SALARY);

    --Raise application error to cancel operation
                 RAISE_APPLICATION_ERROR(-20001, message_text);

END;
```

Employee _Salary_Check Test SQL

```
--Attempt to update Smith
--This should fail because Smith is not a manager
UPDATE employee
SET salary = 71000
WHERE lname = 'Smith';
--Attempt to update Borg
--This should succeed because Borg is a manager
UPDATE employee
SET salary = 71000
WHERE lname = 'Borg';
```

Lab Exercise (1)

- Create a row level trigger after Insert or Update to record an entry in budget for any employee whose salary is greater than $70,00

Employee Journal Example

This AFTER INSERT OR UPDATE Row-Level Trigger will record an audit entry of any employee whose salary has been raised above $70,000.

- Requires a new table to hold the journal entries
- It will use the OLD and NEW Column values

Employee Journal Setup SQL

- First create new table to hold the journal entries

```
CREATE TABLE audit_entry (
  entry_date  DATE,
  entry_user  VARCHAR2(30),
  entry_text  VARCHAR2(2000),
  old_value   VARCHAR2(2000),
  new_value   VARCHAR2(2000));
```

Create Trigger SQL

```
CREATE OR REPLACE TRIGGER employee_journal
  AFTER INSERT OR UPDATE OF salary ON employee
  FOR EACH ROW
  WHEN (new.salary > 70000)

BEGIN

  --Insert the new row into the audit_entry table
  INSERT INTO audit_entry
    (entry_date, entry_user, entry_text, old_value, new_value)
  VALUES
    (SYSDATE, USER, 'Salary > 70000 for ' || :NEW.ssn,
     :OLD.salary,
     :NEW.salary);

END;
```

Test SQL

```
--Update the employee Borg
UPDATE employee
SET salary = 75000
WHERE lname = 'Borg';

--See if the record was added to the audit table
SELECT * FROM audit_entry;
```

Lab exercise (2)

- Create a row level trigger AFTER INSERT OR UPDATE to request additional money for a department budget when a raise is given or new employee is hired.

Budget Event Example

This AFTER INSERT OR UPDATE Row-Level Trigger will request additional money for a department's budget when:

- An employee is hired
- An employee is given a raise

Budget event setup SQL

```
CREATE TABLE budget_request (
  account_no    VARCHAR2(3),
  amount        NUMBER(6),
  description   VARCHAR2(2000),
  date_entered  DATE default
SYSDATE);
```

Budget event SQL

```
CREATE OR REPLACE TRIGGER budget_event
  AFTER INSERT OR UPDATE OF salary ON employee
  FOR EACH ROW

BEGIN

  --Test whether this is an Update event
  IF UPDATING
  AND :NEW.salary > :OLD.salary THEN

    --Insert the raise detail
    INSERT INTO budget_request (account_no, amount, description)
      VALUES(101, :NEW.salary - :OLD.salary, 'Employee raise');

  ELSE

    --Insert the new employee detail
    INSERT INTO budget_request (account_no, amount, description)
      VALUES(101, :NEW.salary, 'New employee');

  END IF;

END;
```

Budget event test sql

```
--Update the employee Borg
UPDATE employee
SET salary = 90000
WHERE lname = 'Borg';

--Add a new employee
INSERT INTO employee
  (fname, minit, lname, ssn, bdate, address, sex, salary,
superssn, dno)
VALUES
  ('John', 'R', 'McMillan', '011325555', '19-JUN-66', '55 Main,
Springfield, OH',
   'M', 69900, '888665555', 3);

--See if the record was added to the budget request table
SELECT * FROM budget_request;
```

INSTEAD OF Triggers

Update operations through views are limited to views which meet certain criteria.

- This is why users or applications may receive errors to DML operations.
- When a DML statement is issued against a view, the INSTEAD OF clause is used instead

INSTEAD OF Syntax

Use the following syntax to create the INSTEAD OF Trigger

```
CREATE OR REPLACE TRIGGER trigger_name
  INSTEAD OF INSERT | UPDATE | DELETE
  ON view_name

DECLARE
…
BEGIN
…
END;
```

INSTEAD OF Triggers Scenario

Consider the following View

```
CREATE OR REPLACE VIEW employee_department_info
      (employee_name, department_name, employee_ssn) AS

   SELECT lname, dname, ssn
   FROM employee
     INNER JOIN department ON (employee.dno =
                               department.dnumber)
   ORDER BY lname;
```

- It is illegal to delete from this view
- Updates are only allowed under certain circumstances

Instead of Trigger

```
--Create a new INSTEAD OF trigger to manage record deletions

CREATE OR REPLACE TRIGGER manage_delete
  INSTEAD OF DELETE
  ON employee_department_info
DECLARE
BEGIN

  --Create the DELETE statement to use
  DELETE FROM employee
  WHERE ssn = :old.employee_ssn;

END;
```

Test SQL

```
--Issue the delete DML statement using the View
DELETE FROM employee_department_info
WHERE employee_ssn = 123456789;

--Verify the department info
SELECT *
FROM employee_department_info;

--Verify the employee info
SELECT ssn, lname
FROM employee;
```

See It In Action