DATABASE MANGEMENT SYSTEMS

B.E IV SEM ‘A’ SEC

P.Mirunalini

AP ,CSE

SSNCE

overview

- What are Triggers?
- Procedures Vs Triggers
- Trigger Issues
- When to use Triggers?
- ECA Model
- Syntax
- Row vs Statement Triggers
- Before Vs After Triggers

What are Triggers?

- Triggers are action that executes (*fire*) automatically whenever some specified event occurs.
- Trigger is a **PL/SQL block or PL/SQL procedure** associated with a table, view, schema, or the database.
- Can be either:
  - **Application trigger**: Fires when an event occurs with a particular application
  - **Database trigger**: Fires when a data event (DML) or system event (logon or shutdown) occurs on schema or database.

Procedure vs Triggers

- Procedures and triggers differ in the way that they are invoked:
  - A procedure is explicitly executed by a user, application, or trigger.
  - Triggers are implicitly fired when a triggering event occurs, no matter which user is connected or which application is being used.
- Note that the database stores triggers separately from their associated tables.

Trigger Issues

- Triggers are not recommended way to implement integrity constraints.
  - Declarative constraints are checked on all relevant updates, whereas triggers are invoked only when the specified event occurs.
- Trigger T1 could cause trigger T2 to fire, which could cause trigger T3 and so on (a *trigger chain*).
- Trigger T might even cause itself to fire again (*recursive*).

When to use Triggers?

- Some useful purpose of triggers:
  - Alerting the user if some exception occurs
  - Debugging (e.g., monitoring references to, and/or state changes in, designated variables)
  - Auditing (e.g., tracking who performed what updates to which relations when)
  - Performance measurement (e.g., timing or tracking specified database events)
  - Carrying out compensating actions.

ECA Model

- Trigger specifies, an *event*, a *condition*, and an *action*:
  - The **event** is an operation on the database (any update operations).
  - The **condition** is a Boolean expression that has to evaluate to TRUE in order for the action to be executed. (if no condition, then the default is just TRUE)
  - The **action** is the triggered procedure.
- The event and condition together are called the *triggering event*
- Triggers are also known as *event-condition-action rules* [ECA]

ECA Model

- Events include **INSERT, DELETE, UPDATE**, end-of-transaction, reaching specified time-of-day, violating a specified constraint **,** and so on.
- The action can be performed **BEFORE, AFTER, or INSTEAD** OF the specified event.
- The action can be performed **FOR EACH ROW or FOR EACH STATEMENT.**
- A database that has associated triggers is sometimes called an *active database.*

Syntax

- CREATE [OR REPLACE] TRIGGER trigg_name
- [AFTER | BEFORE] [INSERT | UPDATE | DELETE]
- ON table_name [FOR EACH ROW]
- [WHEN (CONDITION)]
  - BEGIN

... .

PL/SQL Block

... ...

The event

The condition

The action

Types of Triggers

- Row Triggers and Statement Triggers
- BEFORE and AFTER Triggers
- INSTEAD OF Triggers
- Triggers on System Events and User Events

Row Vs Statement Trigger

- A **row trigger** is fired each time the table is affected by the triggering statement.
- For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired **once for each row** affected by the UPDATE statement.
- The **FOR EACH ROW** option determines whether the trigger is a *row* trigger or a *statement* trigger.

Row Triggers-Example

```
CREATE OR REPLACE TRIGGER Print_salary_changes
  BEFORE INSERT OR UPDATE ON Emp_tab
  FOR EACH ROW
        WHEN (new.Employee_id > 0)
        DECLARE
              sal_diff number;
         BEGIN
              sal_diff  := :new.salary  - :old.salary;
              dbms_output.put('Old salary: ' || :old.salary);
             dbms_output.put('  New salary: ' || :new.salary);
             dbms_output.put_line('  Difference ' || sal_diff);
        END;
```

Column values in Row triggers

- The PL/SQL code and SQL statements have access to the old and new column values of the current row affected by the triggering statement.
- The new column values are referenced using the *new* qualifier while the old column values are referenced using the *old* qualifier before the column name.
- Example:

```
sal_diff  := :new.salary  - :old.salary;
```

Row Vs Statement Trigger

- A **statement trigger** is fired once on behalf of the triggering statement, **regardless of the number of rows** in the table that the triggering statement affects, even if no rows are affected.
- The **absence of the FOR EACH ROW** option indicates that the trigger fires only once for each applicable statement, but not separately for each row affected by the statement.

Statement Triggers- Example

-------Creating statement triggers

CREATE OR REPLACE TRIGGER emp_alert_trig

BEFORE INSERT ON emp_tab

BEGIN

DBMS_OUTPUT.PUT_LINE('New employees are about to be added');

END;

-------Inserting values

- INSERT INTO emp_tab (employee_id,last_name,salary) SELECT employee_id + 1000, last_name, salary FROM emp_tab WHERE employee_id BETWEEN 200 AND 206

Using Predicates

- More than one type of DML operation can fire a trigger:
  - Ex: ON INSERT OR DELETE OR UPDATE OF salary
- The trigger body can use the conditional predicates **INSERTING, DELETING, and UPDATING** to check which type of statement fire the trigger.

Ex:

```
IF INSERTING THEN .... .... END IF;
IF DELETING THEN .... .... END IF;
IF UPDATING ('SAL') THEN ... ... END IF;
```

Inserting, Deleting, Updating-Predicates Example

```
CREATE OR REPLACE TRIGGER LogRSChanges
      BEFORE INSERT OR DELETE OR UPDATE ON employee
      FOR EACH ROW
       DECLARE
        v_ChangeType CHAR(1);
       BEGIN
        /* Use 'I' for an INSERT, 'D' for DELETE, and 'U' for UPDATE. */
          IF INSERTING THEN
                v_ChangeType := 'I';
          ELSIF UPDATING THEN
                v_ChangeType := 'U';
          ELSE

               v_ChangeType := 'D';
          END IF;

        DBMS_OUTPUT.put_line(v_ChangeType ||' '|| USER ||' ' ||SYSDATE);
      END LogRSChanges;
```

Before Vs After Triggers

- When defining a trigger, you can specify the trigger timing – whether the trigger action is to be executed before or after the triggering statement.
- **BEFORE and AFTER** apply to both statement and row triggers.
- BEFORE – For row triggers, the trigger is fired before each affected row is changed.
- AFTER – For row triggers, the trigger is fired after each affected row is changed.

Before Vs After Triggers

Restrictions on AFTER Triggers:

- You cannot specify an AFTER trigger on a view or an object view.
- You cannot write either the :OLD or the :NEW value.

Restrictions on BEFORE Triggers:

- You cannot specify a BEFORE trigger on a view or an object view.
- You can write to the :NEW value but not to the :OLD value.

Before-Example

```
CREATE OR REPLACE TRIGGER emp_trig
        BEFORE UPDATE OF last_name
         ON emp_tab
       FOR EACH ROW
      BEGIN
       DBMS_OUTPUT.PUT_LINE('First Name '||:OLD.fname||'
      has change to '||:NEW.fname);
      END;
```

After-Example

```
-----Creating emp-log table
CREATE TABLE Emp_log ( Emp_id NUMBER,
Log_date DATE,
 New_salary NUMBER,
Action VARCHAR2(20));
```

After-Example

-----Creating trigger

```
CREATE OR REPLACE TRIGGER Log_salary_increase
AFTER UPDATE ON Emp_tab
 FOR EACH ROW WHEN (new.Sal > 1000)
 BEGIN
      INSERT INTO Emp_log (Emp_id, Log_date, New_salary, Action)
       VALUES (:new.Empno, SYSDATE, :new.SAL, 'NEW SAL');
END;
```

After-Example

UPDATE Emp_tab SET Sal = Sal + 1000.0 WHERE last_name=‘pan’;

INSTEAD OF Triggers

- INSTEAD OF triggers provide a transparent way of modifying views that cannot be modified directly through DML statements
- You can write normal INSERT, UPDATE, and DELETE statements against the view
- The INSTEAD OF trigger is fired to update the underlying tables appropriately.
- INSTEAD OF triggers are activated for each row of the view that gets modified.
- INSTEAD OF triggers are valid only for views. You cannot specify an INSTEAD OF trigger on a table.
- Can read both the :OLD and the :NEW value, but cannot write either the :OLD or the :NEW value.

INSTEAD OF Triggers-Example

```
CREATE VIEW order_info AS
    SELECT c.customer_id, c.cust_last_name, c.cust_first_name,
    o.order_id, o.order_date, o.order_status
  FROM customers c, orders o
  WHERE c.customer_id = o.customer_id;
```

```
CREATE OR REPLACE TRIGGER order_info_insert
 INSTEAD OF INSERT ON order_info
DECLARE
duplicate_info  EXCEPTION;
PRAGMA EXCEPTION_INIT (duplicate_info, -00001);
BEGIN
INSERT INTO customers (customer_id, cust_last_name, cust_first_name)
    VALUES ( :new.customer_id, :new.cust_last_name,
    :new.cust_first_name);
 INSERT INTO orders (order_id, order_date, customer_id)
 VALUES ( :new.order_id, :new.order_date, :new.customer_id);
 EXCEPTION WHEN duplicate_info THEN
    RAISE_APPLICATION_ERROR ( num=> -20107, msg=> 'Duplicate
    customer or order ID');
END order_info_insert;
```

Thank you