Triggers

CHAPTER OBJECTIVES

In this chapter, you will learn about:

- What triggers are
- Types of triggers

What Triggers Are

- A database trigger is a named PL/SQL block stored in a database and executed implicitly when a *triggering event* occurs.
- The act of executing a trigger is called firing the trigger.
- A triggering event can be one of the following:
  - A DML statement (such as INSERT, UPDATE, or DELETE) executed against a database table. Such a trigger can fire before or after a triggering event. For example, if you have defined a trigger to fire before an INSERT statement on the STUDENT table, this trigger fires each time before you insert a row in the STUDENT table.
  - A DDL statement (such as CREATE or ALTER) executed either by a particular user against a schema or by any user. Such triggers are often used for auditing purposes and are specifically helpful to Oracle DBAs. They can record various schema changes, when they were made, and by which user.
  - A system event such as startup or shutdown of the database.
  - A user event such as logon and logoff. For example, you can define a trigger that fires after database logon that records the username and time of logon.

What Triggers Are

The general syntax for creating a trigger is as follows (the reserved words and phrases in brackets are optional):

```
CREATE [OR REPLACE] TRIGGER Ttrigger_name
{BEFORE|AFTER} Triggering_event ON table_name
[FOR EACH ROW]
[FOLLOWS another_trigger] [ENABLE/DISABLE]
[WHEN condition]
DECLARE
declaration statements
BEGIN
executable statements
EXCEPTION
exception-handling statements
END;
```

What Triggers Are

- The reserved word CREATE specifies that you are creating a new trigger.
- The reserved word REPLACE specifies that you are modifying an existing trigger.
- REPLACE is optional. However, note that both CREATE and REPLACE are present most of the time.
- Consider the following situation:

  You create a trigger as follows:

  ```
  CREATE TRIGGER Trigger_name
  ...
  ```

  In a few days you decide to modify this trigger. If you do not include the reserved word REPLACE in the CREATE clause of the trigger, an error message will be generated when you compile the trigger. The error message states that the name of your trigger is already being used by another object. When REPLACE is included in the CREATE clause of the trigger, there is less chance of an error because, if this is a new trigger, it is created, and if it is an old trigger, it is replaced.

What Triggers Are

- The trigger_name is the trigger's name.
- BEFORE or AFTER specifies when the trigger fires (before or after the triggering event).
- The triggering_event is a DML statement issued against the table.
- table_name is the name of the table associated with the trigger.
- The clause FOR EACH ROW specifies that a trigger is a row trigger and fires once for each row that is inserted, updated, or deleted.
- WHEN clause specifies a condition that must evaluate to TRUE for the trigger to fire.
- For example, this condition may specify a certain restriction on the column of a table. *This portion of the trigger is often called the trigger header*. Next, the trigger body is defined.

What Triggers Are

- Note the three clauses, FOLLOWS, ENABLE, and DISABLE.
- These were added to the CREATE OR REPLACE TRIGGER clause in Oracle 11g.
- Prior to Oracle 11g, you needed to issue the ALTER TRIGGER command to enable or disable a trigger after it was created.
- The ENABLE and DISABLE clauses specify whether the trigger is created in the enabled or disabled state.
- When the trigger is enabled, it fires when a triggering event occurs.
- Similarly, when a trigger is disabled, it does not fire when a triggering event occurs.
- Note that when trigger is first created without an ENABLE or DISABLE clause, it is enabled by default.

What Triggers Are

- To disable the trigger, you need to issue the ALTER TRIGGER command as follows:

  ALTER TRIGGER trigger_name DISABLE;

- Similarly, to enable a trigger that was disabled previously, you issue the ALTER TRIGGER command as follows:

  ALTER TRIGGER trigger_name ENABLE;

What Triggers Are

- The FOLLOWS option allows you to specify the order in which triggers should fire.
- This applies to triggers that are defined on the same table and that fire at the same timing point.
- For example, if you defined two triggers on the STUDENT table that fire before the insert occurs, Oracle does not guarantee the order in which these triggers will fire unless you specify it with the FOLLOWS clause.
- Note that the trigger referenced in the FOLLOWS clause must already exist and have been successfully compiled.
- It is important for you to realize that if you drop a table, the table's database triggers are dropped as well.

What Triggers Are

- You should be careful when using the reserved word REPLACE for a number of reasons.
- First, if you happen to use REPLACE and the name of an existing stored function, procedure, or package, you will end up with different database objects that have the same name. This occurs because triggers have separate naming space in the database. Although a trigger and a procedure, function, or package sharing the same name does not cause errors, potentially it might become confusing. As a result, it is not considered a good programming practice.
- Second, when you use the reserved word REPLACE and decide to associate a different table with your trigger, an error message is generated. For example, assume that you created a trigger STUDENT_BI on the STUDENT table. Next, you decide to modify this trigger and associate it with the ENROLLMENT table. As a result, the following error message is generated:

```
ERROR at line 1:
ORA-04095: trigger 'STUDENT_BI' already exists on another table, cannot replace it
```

What Triggers Are

Triggers are used for different purposes:

- Enforcing complex business rules that cannot be defined by using integrity constraints
- Maintaining complex security rules
- Automatically generating values for derived columns
- Collecting statistical information on table accesses
- Preventing invalid transactions
- Providing value auditing

What Triggers Are

The body of a trigger is a PL/SQL block. However, you need to know about several restrictions before creating a trigger:

- A trigger may not issue a transactional control statement such as COMMIT, SAVEPOINT, or ROLLBACK. When the trigger fires, all operations performed become part of a transaction. When this transaction is committed or rolled back, the operations performed by the trigger are committed or rolled back as well. An exception to this rule is a trigger that contains an autonomous transaction.
- Any function or procedure called by a trigger may not issue a transactional control statement unless it contains an autonomous transaction.
- It is not permissible to declare LONG or LONG RAW variables in the body of a trigger.

Before Triggers

Consider the following example of a trigger on the STUDENT table mentioned earlier in this chapter. This trigger fires before the INSERT statement on the STUDENT table and populates the STUDENT_ID, CREATED_DATE, MODIFIED_DATE, CREATED_BY, and MODIFIED_BY columns. Column STUDENT_ID is populated with the number generated by the STUDENT_ID_SEQ sequence, and columns CREATED_DATE, MODIFIED_DATE, CREATED_USER, and MODIFIED_USER are populated with the current date and the current username information.

Before Triggers

```
CREATE OR REPLACE TRIGGER student_bi
BEFORE INSERT ON student
FOR EACH ROW
BEGIN
:NEW.student_id := STUDENT_ID_SEQ.NEXTVAL;
:NEW.created_by := USER;
:NEW.created_date := SYSDATE;
:NEW.modified_by := USER;
:NEW.modified_date := SYSDATE;
END;
```

Before Triggers

- This trigger fires for each row before the INSERT statement on the STUDENT table.
- Notice that the name of the trigger is STUDENT_BI, where STUDENT is the name of the table on which the trigger is defined, and BI means BEFORE INSERT.
- There is no specific requirement for naming triggers; however, this approach to naming a trigger is descriptive.
- The name of the trigger contains the name of the table affected by the triggering event, the time of the triggering event (before or after), and the triggering event itself.

Before Triggers

- In the body of the trigger is a pseudorecord, :NEW, allowing you to access a row currently being processed. In other words, a row is being inserted into the STUDENT table. The :NEW pseudorecord is of type TRIGGERING_TABLE%TYPE, so, in this case, it is of the STUDENT%TYPE type.
- To access individual members of the pseudorecord :NEW, dot notation is used. In other words, :NEW.CREATED_BY refers to the member CREATED_BY of the :NEW pseudorecord, and the name of the record is separated from the name of its member by a dot.

Before Triggers

- Take a closer look at the statement that assigns a sequence value to the STUDENT_ID column.
- The ability to access a sequence via PL/SQL expression is a new feature in Oracle 11g. Prior to Oracle 11g, sequences could be accessed only via queries.

Before Triggers

```
CREATE OR REPLACE TRIGGER student_bi
BEFORE INSERT ON student
FOR EACH ROW
DECLARE
v_student_id STUDENT.STUDENT_ID%TYPE;
BEGIN
SELECT STUDENT_ID_SEQ.NEXTVAL
INTO v_student_id
FROM dual;
:NEW.student_id := v_student_id;
:NEW.created_by := USER;
:NEW.created_date := SYSDATE;
:NEW.modified_by := USER;
:NEW.modified_date := SYSDATE;
END;
```

Before Triggers

Before you create this trigger, consider the following INSERT statement on the STUDENT table:

INSERT INTO student (student_id, first_name, last_name, zip, registration_date, created_by, created_date, modified_by, modified_date)

VALUES (STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '00914', SYSDATE, USER, SYSDATE, USER, SYSDATE);

Before Triggers

This INSERT statement contains values for the columns STUDENT_ID, CREATED_BY, CREATED_DATE, MODIFIED_BY, and MODIFIED_DATE.

It is important to note that for every row you insert into the STUDENT table, the values for these columns must be provided, and they are always derived in the same fashion. Why do you think the values for these columns must be provided when you insert a record into the STUDENT table?

Before Triggers

- When the trigger shown earlier is created, there is no need to include these columns in the INSERT statement, because the trigger populates them with the required information.
- Therefore, the INSERT statement can be modified as follows:
- INSERT INTO student (first_name, last_name, zip, registration_date) VALUES ('John', 'Smith', '00914', SYSDATE);
- Notice that this version of the INSERT statement looks significantly shorter than the previous version.
- The columns STUDENT_ID, CREATED_BY, CREATED_DATE, MODIFIED_BY, and MODIFIED_DATE are not present. However, the trigger provides their values.
- As a result, there is no need to include them in the INSERT statement, and there is less chance of a transaction error.

Before Triggers

You should use BEFORE triggers in the following situations:

- When a trigger provides values for derived columns before an INSERT or UPDATE statement is completed. For example, the column FINAL_GRADE in the ENROLLMENT table holds the value of the student's final grade for a specific course. This value is calculated based on the student's performance for the duration of the course.
- When a trigger determines whether an INSERT, UPDATE, or DELETE statement should be allowed to complete. For example, when you insert a record into the INSTRUCTOR table, a trigger can verify whether the value provided for the column ZIP is valid, or, in other words, if a record in the ZIPCODE table corresponds to the value of zip that you provided.

After Triggers

- Assume that a table called STATISTICS has the following structure:

```
Name                           Null? Type
------------------------------ -------- ----
TABLE_NAME                     VARCHAR2(30)
TRANSACTION_NAME               VARCHAR2(10)
TRANSACTION_USER               VARCHAR2(30)
TRANSACTION_DATE               DATE
```

- This table is used to collect statistical information on different tables of the database. For example, you can record who deleted records from the INSTRUCTOR table and when they were deleted.
- Consider the following example of a trigger on the INSTRUCTOR table. This trigger fires after an UPDATE or DELETE statement is issued on the INSTRUCTOR table.

After Triggers

```
CREATE OR REPLACE TRIGGER instructor_aud
AFTER UPDATE OR DELETE ON INSTRUCTOR
DECLARE
v_type VARCHAR2(10);
BEGIN
IF UPDATING THEN
v_type := 'UPDATE';
ELSIF DELETING THEN
v_type := 'DELETE';
END IF;
UPDATE statistics
SET transaction_user = USER,
transaction_date = SYSDATE
WHERE table_name = 'INSTRUCTOR'
AND transaction_name = v_type;
IF SQL%NOTFOUND THEN
INSERT INTO statistics
VALUES ('INSTRUCTOR', v_type, USER, SYSDATE);
END IF;
END;
```

After Triggers

- This trigger fires after an UPDATE or DELETE statement on the INSTRUCTOR table.
- In the body of the trigger are two Boolean functions, UPDATING and DELETING.
- The function UPDATING evaluates to TRUE if an UPDATE statement is issued on the table, and the function DELETING evaluates to TRUE if a DELETE statement is issued on the table.
- There is another Boolean function called INSERTING. As you probably can guess, this function evaluates to TRUE when an INSERT statement is issued against the table.

After Triggers

- This trigger updates a record or inserts a new record into the STATISTICS table when an UPDATE or DELETE operation is issued against the INSTRUCTOR table.
- First, the trigger determines the type of the DML statement issued against the INSTRUCTOR table. This determination is made with the help of the UPDATING and DELETING functions.

After Triggers

- Next, the trigger tries to update a record in the STATISTICS table where TABLE_NAME is equal to INSTRUCTOR and TRANSACTION_NAME is equal to the current transaction (UPDATE or DELETE).
- Then the status of the UPDATE statement is checked with the help of the SQL%NOTFOUND constructor.
- The SQL%NOTFOUND constructor evaluates to TRUE if the UPDATE statement does not update any rows and evaluates to FALSE otherwise.
- So if SQL%NOTFOUND evaluates to TRUE, a new record is added to the STATISTICS table.

After Triggers

- After this trigger is created on the INSTRUCTOR table, any UPDATE or DELETE operation causes modification of old records or creation of new records in the STATISTICS table.
- Furthermore, you can enhance this trigger by calculating how many rows are updated or deleted from the INSTRUCTOR table.
- You should use AFTER triggers in the following situations:
    - When a trigger should fire after a DML statement is executed
    - When a trigger performs actions not specified in a BEFORE trigger

Autonomous Transaction

- An autonomous transaction is an independent transaction started by another transaction that is usually called the main transaction.
- In other words, the autonomous transaction may issue various DML statements and commit or roll them back, without committing or rolling back the DML statements issued by the main transaction.
- For example, consider the trigger created earlier that fires after the UPDATE or DELETE statement is issued on the INSTRUCTOR table where you record auditing data. Suppose you want to record auditing data even when the main transaction fails (in this case, the main transaction is the UPDATE or DELETE statement issued on the INSTRUCTOR table). You need to define an autonomous transaction that can be committed independently of the main transaction.

Autonomous Transaction

- To define an autonomous transaction, you employ the AUTONOMOUS_TRANSACTION pragma.
- Recall that a pragma is a special instruction to the PL/SQL compiler that is processedat the time of compilation. The AUTONOMOUS_TRANSACTION pragma appears in the declaration section of a block:

```
DECLARE
PRAGMA AUTONOMOUS_TRANSACTION;
```

- Consider a modified version of INSTRUCTOR_AUD with the autonomous transaction. Changes are shown in bold:

Autonomous Transaction

```
CREATE OR REPLACE TRIGGER instructor_aud
AFTER UPDATE OR DELETE ON INSTRUCTOR
DECLARE
v_type VARCHAR2(10);
PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
IF UPDATING THEN v_type := 'UPDATE';
ELSIF DELETING THEN v_type := 'DELETE';
END IF;
UPDATE statistics
SET transaction_user = USER, transaction_date = SYSDATE WHERE table_name = 'INSTRUCTOR‘ AND
transaction_name = v_type;
IF SQL%NOTFOUND THEN INSERT INTO statistics
VALUES ('INSTRUCTOR', v_type, USER, SYSDATE);
END IF;
COMMIT;
END;
```

Autonomous Transaction

- In this version of the trigger, you add the AUTONOMOUS_TRANSACTION pragma to the declaration portion of the trigger and the COMMIT statement to the executable portion.
- Next, consider the UPDATE statement on the INSTRUCTOR table that is rolled back, and the SELECT against the STATISTICS table:

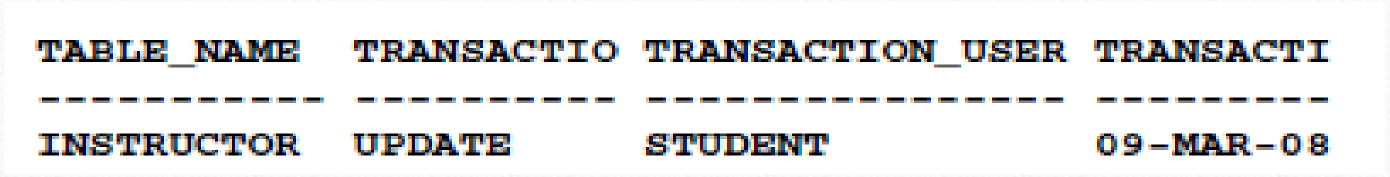
```
UPDATE instructor
SET phone = '7181234567'
WHERE instructor_id = 101;
```

1 row updated.

Autonomous Transaction

ROLLBACK;

SELECT * FROM statistics;



```
TABLE_NAME   TRANSACTIO TRANSACTION_USER TRANSACTI
------------ ---------- ---------------- ---------
INSTRUCTOR   UPDATE     STUDENT          09-MAR-08
```

Notice that even though you roll the UPDATE statement against the INSTRUCTOR table, the record is inserted in the STATISTICS table due to the autonomous transaction specified in the trigger body.

Exercise

Create the following trigger:

```
CREATE OR REPLACE TRIGGER instructor_bi
BEFORE INSERT ON INSTRUCTOR
FOR EACH ROW
DECLARE
v_work_zip CHAR(1);
BEGIN
:NEW.CREATED_BY := USER;
:NEW.CREATED_DATE := SYSDATE;
:NEW.MODIFIED_BY := USER;
:NEW.MODIFIED_DATE := SYSDATE;
SELECT 'Y‘ INTO v_work_zip FROM zipcode WHERE zip = :NEW.ZIP;
EXCEPTION
WHEN NO_DATA_FOUND THEN
RAISE_APPLICATION_ERROR (-20001, 'Zip code is not valid!');
END;
```

Exercise

- If an INSERT statement issued against the INSTRUCTOR table is missing a value for the column ZIP, does the trigger raise an exception? Explain your answer.
- **ANSWER:** Yes, the trigger raises an exception. When an INSERT statement does not provide a value for the column ZIP, the value of :NEW.ZIP is NULL. This value is used in the WHERE clause of the SELECT INTO statement. As a result, the SELECT INTO statement is unable to return data. Therefore, the trigger raises a NO_DATA_FOUND exception.

Exercise

- Modify this trigger so that another error message is displayed when an INSERT statement is missing a value for the column ZIP.

Exercise

```
CREATE OR REPLACE TRIGGER instructor_bi
BEFORE INSERT ON INSTRUCTOR
FOR EACH ROW
DECLARE
v_work_zip CHAR(1);
BEGIN
:NEW.CREATED_BY := USER;
:NEW.CREATED_DATE := SYSDATE;
:NEW.MODIFIED_BY := USER;
:NEW.MODIFIED_DATE := SYSDATE;
IF :NEW.ZIP IS NULL THEN RAISE_APPLICATION_ERROR (-20002, 'Zip code is missing!');
ELSE
SELECT 'Y‘ INTO v_work_zip FROM zipcode WHERE zip = :NEW.ZIP;
END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN
RAISE_APPLICATION_ERROR (-20001, 'Zip code is not valid!');
END;
```

Exercise

Notice that an IF-ELSE statement is added to the body of the trigger.This IF-ELSE statement evaluates the value of :NEW.ZIP.

If the value of :NEW.ZIP is NULL, the IF-ELSE statement evaluates to TRUE, and another error message is displayed, stating that the value of ZIP is missing.

If the IF-ELSE statement evaluates to FALSE, control is passed to the ELSE part of the statement, and the SELECT INTO statement is executed.

Row and Statement Triggers

- A row trigger is fired as many times as there are rows affected by the triggering statement.
- When the statement FOR EACH ROW is present in the CREATE TRIGGER clause, the trigger is a row trigger. Consider the following code:

FOR EXAMPLE

```
CREATE OR REPLACE TRIGGER course_au
AFTER UPDATE ON COURSE
FOR EACH ROW
...
```

Row and Statement Triggers

- In this code fragment, the statement FOR EACH ROW is present in the CREATE TRIGGER clause.
- Therefore, this trigger is a row trigger. If an UPDATE statement causes 20 records in the COURSE table to be modified, this trigger fires 20 times.
- A statement trigger is fired once for the triggering statement. In other words, a statement trigger fires once, regardless of the number of rows affected by the triggering statement.
- To create a statement trigger, you omit the FOR EACH ROW in the CREATE TRIGGER clause. Consider the following code fragment:

```
CREATE OR REPLACE TRIGGER enrollment_ad
AFTER DELETE ON ENROLLMENT
...
```

Row and Statement Triggers

- This trigger fires once after a DELETE statement is issued against the ENROLLMENT table.
- Whether the DELETE statement removes one row or five rows from the ENROLLMENT table, this trigger fires only once.
- Statement triggers should be used when the operations performed by the trigger do not depend on the data in the individual records.
- For example, if you want to limit access to a table to business hours only, a statement trigger is used. Consider the following example:

Row and Statement Triggers

```
CREATE OR REPLACE TRIGGER instructor_biud
BEFORE INSERT OR UPDATE OR DELETE ON INSTRUCTOR
DECLARE
v_day VARCHAR2(10);
BEGIN
v_day := RTRIM(TO_CHAR(SYSDATE, 'DAY'));
IF v_day LIKE ('S%') THEN
RAISE_APPLICATION_ERROR (-20000, 'A table cannot be modified during off
hours');
END IF;
END;
```

Row and Statement Triggers

- This is a statement trigger on the INSTRUCTOR table, and it fires before an INSERT, UPDATE, or DELETE statement is issued.
- First, the trigger determines the day of the week. If the day is Saturday or Sunday, an error message is generated. When the following UPDATE statement on the INSTRUCTOR table is issued on Saturday or Sunday:

```
UPDATE instructor
SET zip = 10025
WHERE zip = 10015;
```

the trigger generates this error message:

```
update INSTRUCTOR
*
ERROR at line 1:
ORA-20000: A table cannot be modified during off hours
ORA-06512: at "STUDENT.INSTRUCTOR_BIUD", line 6
ORA-04088: error during execution of trigger
'STUDENT.INSTRUCTOR_BIUD'
```

Row and Statement Triggers

- Notice that this trigger checks for a specific day of the week.
- However, it does not check the time of day.
- You can create a more sophisticated trigger that checks what day of the week it is and if the current time is between 9 a.m. and 5 p.m.
- If the day is during the business week but the time of day is not between 9 a.m. and 5 p.m., the error is generated.

Instead of Triggers

- So far you have seen triggers that are defined on database tables.
- PL/SQL provides another kind of trigger that is defined on database views. A view is a custom representation of data and can be called a *stored query*.
- Consider the following example of the view created against the COURSE table:

```
CREATE VIEW course_cost AS
SELECT course_no, description, cost
FROM course;
```

Instead of Triggers

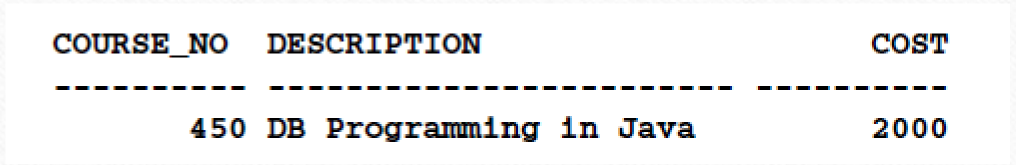
- Similar to tables, views can be manipulated via INSERT, UPDATE, or DELETE statements, with some restrictions.
- However, it is important to note that when any of these statements are issued against a view, the corresponding data is modified in the underlying tables.
- For example, consider an UPDATE statement against the COURSE_COST view:

```
UPDATE course_cost
SET cost = 2000
WHERE course_no = 450;
COMMIT;
```

Instead of Triggers

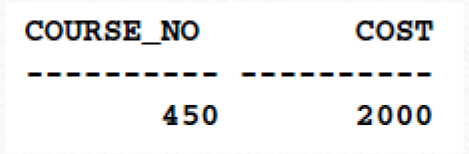
- After the UPDATE statement is executed, both SELECT statements against the COURSE_COST view and the COURSE table return the same value of the cost for course number 450:

SELECT * FROM course_cost WHERE course_no = 450;



```
COURSE_NO  DESCRIPTION                     COST
---------- ------------------------ ----------
       450 DB Programming in Java         2000
```

SELECT course_no, cost FROM course WHERE course_no = 450;



```
 COURSE_NO       COST
---------- ----------
       450       2000
```

Instead of Triggers

- As mentioned earlier, some views are restricted as to whether they can be modified by INSERT, UPDATE, or DELETE statements.
- Specifically, these restrictions apply to the underlying SELECT statement, which is also called a *view query*.
- Thus, if a view query performs any of the operations or contains any of the following constructs, a view cannot be modified by an UPDATE, INSERT, or DELETE statement:
    - Set operations such as UNION, UNION ALL, INTERSECT, and MINUS
    - Group functions such as AVG, COUNT, MAX, MIN, and SUM
    - GROUP BY or HAVING clauses
    - CONNECT BY or START WITH clauses
    - The DISTINCT operator
    - The ROWNUM pseudocolumn

Instead of Triggers

- Consider the following view created on the INSTRUCTOR and SECTION tables:

```
CREATE VIEW instructor_summary_view AS
SELECT i.instructor_id, COUNT(s.section_id) total_courses
FROM instructor i
LEFT OUTER JOIN section s
ON (i.instructor_id = s.instructor_id)
GROUP BY i.instructor_id;
```

- Note that the SELECT statement is written in the ANSI 1999 SQL standard. It uses the outer join between the INSTRUCTOR and SECTION tables. The LEFT OUTER JOIN indicates that an instructor record in the INSTRUCTOR table that does not have a corresponding record in the SECTION table is included in the result set with TOTAL_COURSES equal to 0.

Instead of Triggers

- This view is not updatable, because it contains the group function, COUNT().
- As a result, the following DELETE statement: DELETE FROM instructor_summary_view WHERE instructor_id = 109; causes the error shown:

```
DELETE FROM instructor_summary_view
*
ERROR at line 1:
ORA-01732: data manipulation operation not legal on this view
```

Instead of Triggers

- PL/SQL provides a special kind of trigger that can be defined on database views.
- This trigger is called an INSTEAD OF trigger and is created as a row trigger. An INSTEAD OF trigger fires instead of the triggering statement (INSERT, UPDATE, DELETE) that has been issued against a view and directly modifies the underlying tables.
- Consider an INSTEAD OF trigger defined on the INSTRUCTOR_SUMMARY_VIEW created earlier. This trigger deletes a record from the INSTRUCTOR table for the corresponding value of the instructor’s ID.

Instead of Triggers

```
CREATE OR REPLACE TRIGGER instructor_summary_del
INSTEAD OF DELETE ON instructor_summary_view
FOR EACH ROW
BEGIN
DELETE FROM instructor
WHERE instructor_id = :OLD.INSTRUCTOR_ID;
END;
```

Instead of Triggers

- After the trigger is created, the DELETE statement against the INSTRUCTOR_SUMMARY_VIEW does not generate any errors:

```
DELETE FROM instructor_summary_view
WHERE instructor_id = 109;
1 row deleted.
```

- When the DELETE statement is issued, the trigger deletes a record from the INSTRUCTOR table corresponding to the specified value of INSTRUCTOR_ID.

Instead of Triggers

- Consider the same DELETE statement with a different instructor ID:

```
DELETE FROM instructor_summary_view WHERE instructor_id = 101;
```

- When this DELETE statement is issued, it causes the error shown:

```
DELETE FROM instructor_summary_view
*
ERROR at line 1:
ORA-02292: integrity constraint (STUDENT.SECT_INST_FK) violated -
child record found
ORA-06512: at "STUDENT.INSTRUCTOR_SUMMARY_DEL", line 2
ORA-04088: error during execution of trigger -
'STUDENT.INSTRUCTOR_SUMMARY_DEL'
```

Instead of Triggers

- The INSTRUCTOR_SUMMARY_VIEW joins the INSTRUCTOR and SECTION tables based on the INSTRUCTOR_ID column that is present in both tables.
- The INSTRUCTOR_ID column in the INSTRUCTOR table has a primary key constraint defined on it.
- The INSTRUCTOR_ID column in the SECTION table has a foreign key constraint that references the INSTRUCTOR_ID column of the INSTRUCTOR table. Thus, the SECTION table is considered a child table of the INSTRUCTOR table.
- The original DELETE statement does not cause any errors because no record in the SECTION table corresponds to the instructor ID of 109. In other words, the instructor with the ID of 109 does not teach any courses.

Instead of Triggers

- The second DELETE statement causes an error because the INSTEAD OF trigger tries to delete a record from the INSTRUCTOR table, the parent table.
- However, a corresponding record in the SECTION table, the child table, has the instructor ID of 101. This causes an integrity constraint violation error. It may seem that one more DELETE statement should be added to the INSTEAD OF trigger, as shown here:

Instead of Triggers

```
CREATE OR REPLACE TRIGGER instructor_summary_del
INSTEAD OF DELETE ON instructor_summary_view
FOR EACH ROW
BEGIN
DELETE FROM section WHERE instructor_id = :OLD.INSTRUCTOR_ID;
DELETE FROM instructor WHERE instructor_id = :OLD.INSTRUCTOR_ID;
END;
```

Instead of Triggers

- Notice that the new DELETE statement removes records from the SECTION table before the INSTRUCTOR table because the SECTION table contains child records of the INSTRUCTOR table.
- However, the DELETE statement against the INSTRUCTOR_SUMMARY_VIEW causes another error:

```
DELETE FROM instructor_summary_view
WHERE instructor_id = 101;
DELETE FROM instructor_summary_view
*
ERROR at line 1:
ORA-02292: integrity constraint (STUDENT.GRTW_SECT_FK) violated -
child record found
ORA-06512: at "STUDENT.INSTRUCTOR_SUMMARY_DEL", line 2
ORA-04088: error during execution of trigger -
'STUDENT.INSTRUCTOR_SUMMARY_DEL'
```

Instead of Triggers

- This time, the error refers to a different foreign key constraint that specifies the relationship between the SECTION and the GRADE_TYPE_WEIGHT tables. In this case, the child records are found in the GRADE_TYPE_WEIGHT table.
- This means that before deleting records from the SECTION table, the trigger must delete all corresponding records from the GRADE_TYPE_WEIGHT table.
- However, the GRADE_TYPE_WEIGHT table has child records in the GRADE table, so the trigger must delete records from the GRADE table first.

Instead of Triggers

- This example illustrates the complexity of designing an INSTEAD OF trigger.
- To design such a trigger, you must be aware of two important factors: the relationship among tables in the database, and the ripple effect that a particular design may introduce.
- This example suggests deleting records from four underlying tables. However, it is important to realize that those tables contain information that relates not only to the instructors and the sections they teach, but also to the students and the sections they are enrolled in.

Exercise

In this exercise, you create a view STUDENT_ADDRESS and an INSTEAD OF trigger that fires instead of an INSERT statement issued against the view.

Create the following view:

```
CREATE VIEW student_address AS
SELECT s.student_id, s.first_name, s.last_name,
s.street_address, z.city, z.state, z.zip
FROM student s
JOIN zipcode z
ON (s.zip = z.zip);
```

Note that the SELECT statement is written in the ANSI 1999 SQL standard.

Exercise

Create the following INSTEAD OF trigger:

```
CREATE OR REPLACE TRIGGER student_address_ins
INSTEAD OF INSERT ON student_address
FOR EACH ROW
BEGIN
INSERT INTO STUDENT
(student_id, first_name, last_name, street_address, zip,registration_date, created_by, created_date, modified_by,modified_date)
VALUES
(:NEW.student_id, :NEW.first_name, :NEW.last_name,:NEW.street_address, :NEW.zip, SYSDATE, USER, SYSDATE, USER,
SYSDATE);
END;
```

Exercise

Issue the following INSERT statements:

```
INSERT INTO student_address
VALUES (STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '123 Main Street',
'New York', 'NY', '10019');
INSERT INTO student_address
VALUES (STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '123 Main Street',
'New York', 'NY', '12345');
```

Exercise

Issue the following INSERT statements:

```
INSERT INTO student_address
VALUES (STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '123 Main Street',
'New York', 'NY', '10019');
INSERT INTO student_address
VALUES (STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '123 Main Street',
'New York', 'NY', '12345');
```

Exercise

What output is produced after each INSERT statement is issued?

Exercise

ANSWER: The output should look similar to the following:

```
INSERT INTO student_address
VALUES (STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '123 Main Street',
'New York', 'NY', '10019');
1 row created.
INSERT INTO student_address
VALUES (STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '123 Main Street',
'New York', 'NY', '12345');
VALUES (STUDENT_ID_SEQ.NEXTVAL, 'John', 'Smith', '123 Main Street',
'New York',
*
ERROR at line 2:
ORA-02291: integrity constraint (STUDENT.STU_ZIP_FK) violated -
parent key not found
ORA-06512: at "STUDENT.STUDENT_ADDRESS_INS", line 2
ORA-04088: error during execution of trigger
'STUDENT.STUDENT_ADDRESS_INS'
```

Exercise

Explain why the second INSERT statement causes an error.

Exercise

ANSWER: The second INSERT statement causes an error because it violates the foreign key constraint on the STUDENT table.The value of the zip code provided at the time of an insert does not have a corresponding record in the ZIPCODE table.

The ZIP column of the STUDENT table has a foreign key constraint STU_ZIP_FK defined on it. This means that each time a record is inserted into the STUDENT table, the system checks the incoming value of the zip code in the ZIPCODE table. If there is a corresponding record, the INSERT statement against the STUDENT table does not cause errors. For example, the first INSERT statement is successful because the ZIPCODE table contains a record corresponding to the value of zip code 10019.The second insert statement causes an error because no record in the ZIPCODE table corresponds to the value of zip code 12345.

Exercise

Modify the trigger so that it checks the value of the zip code provided by the INSERT statement against the ZIPCODE table and raises an error if there is no such value.

Exercise

ANSWER: The trigger should look similar to the following. All changes are shown in bold.

```
CREATE OR REPLACE TRIGGER student_address_ins
INSTEAD OF INSERT ON student_address
FOR EACH ROW
DECLARE
v_zip VARCHAR2(5);
BEGIN
SELECT zip INTO v_zip FROM zipcode WHERE zip = :NEW.ZIP;
INSERT INTO STUDENT (student_id, first_name, last_name, street_address, zip, registration_date,
created_by, created_date, modified_by, modified_date)
VALUES
(:NEW.student_id, :NEW.first_name, :NEW.last_name, :NEW.street_address, :NEW.zip, SYSDATE, USER,
SYSDATE, USER, SYSDATE);
EXCEPTION
WHEN NO_DATA_FOUND THEN
RAISE_APPLICATION_ERROR (-20002, 'Zip code is not valid!');
END;
```

Exercise

Modify the trigger so that it checks the value of the zip code provided by the INSERT statement against the ZIPCODE table. If the ZIPCODE table has no corresponding record, the trigger should create a new record for the given value of zip before adding a new record to the STUDENT table.

Exercise

ANSWER: The trigger should look similar to the following. All changes are shown in bold.

```
CREATE OR REPLACE TRIGGER student_address_ins
INSTEAD OF INSERT ON student_address
FOR EACH ROW
DECLARE
v_zip VARCHAR2(5);
BEGIN
SELECT zip INTO v_zip FROM zipcode WHERE zip = :NEW.zip;
EXCEPTION
WHEN NO_DATA_FOUND THEN
INSERT INTO ZIPCODE
(zip, city, state, created_by, created_date, modified_by, modified_date)
VALUES
(:NEW.zip, :NEW.city, :NEW.state, USER, SYSDATE, USER, SYSDATE);
END;
```