CHAPTER 19
Triggers

Database triggers are named program units that are executed in response to events that occur in the database. Triggers are critical elements of a well-designed application built on the Oracle database and are used to do the following:

Perform validation on changes being made to tables
: Because the validation logic is attached directly to the database object, database triggers offer a strong guarantee that the required logic will always be executed and enforced.

Automate maintenance of the database
: Starting with Oracle8*i* Database, you can use database startup and shutdown triggers to automatically perform necessary initialization and cleanup steps. This is a distinct advantage over creating and running such steps as scripts external to the database.

Apply rules concerning acceptable database administration activity in a granular fashion
: You can use triggers to tightly control what kinds of actions are allowed on database objects, such as dropping or altering tables. Again, by putting this logic in triggers, you make it very difficult, if not impossible, for anyone to bypass the rules you have established.

Five different types of events can have trigger code attached to them:

Data Manipulation Language (DML) statements
: DML triggers are available to fire whenever a record is inserted into, updated in, or deleted from a table. These triggers can be used to perform validation, set default values, audit changes, and even disallow certain DML operations.

Data Definition Language (DDL) statements
: DDL triggers fire whenever DDL is executed—for example, whenever a table is created. These triggers can perform auditing and prevent certain DDL statements from executing.

Database events
: Database event triggers fire whenever the database starts up or is shut down, whenever a user logs on or off, and whenever an Oracle error occurs. For Oracle8*i* Database and above, these triggers provide a means of tracking activity in the database.

INSTEAD OF
: INSTEAD OF triggers are essentially alternatives to DML triggers. They fire when inserts, updates, and deletes are about to occur; your code specifies what to do in place of these DML operations. INSTEAD OF triggers control operations on views, not tables. They can be used to make nonupdateable views updateable and to override the behavior of views that are updateable.

Suspended statements
: Oracle9*i* Database introduced the concept of suspended statements. Statements experiencing space problems (lack of tablespace or quotas exceeded) can enter a suspended mode until the space problem is fixed. Triggers can be added to the mix to automatically alert someone of the problem or even to fix it.

This chapter describes these types of triggers; for each, I'll provide syntax details, example code, and suggested uses. I'll also touch on trigger maintenance at the end of the chapter.

If you need to emulate triggers on SELECT statements (queries), you should investigate the use of fine-grained auditing (FGA), which is described in Chapter 23 and in greater detail in the book *Oracle PL/SQL for DBAs*.

DML Triggers

Data Manipulation Language triggers fire when records are inserted into, updated within, or deleted from a particular table, as shown in Figure 19-1. These are the most common types of triggers, especially for developers; the other trigger types are used primarily by DBAs. And starting with Oracle Database 11*g*, you can combine several DML triggers into one *compound* trigger.

Before you begin writing your DML trigger, you should answer the following questions:

- Should the trigger fire once for the entire DML statement or once for each row involved in the statement?
- Should the trigger fire before or after the statement/row?

- Should the trigger fire for inserts, updates, deletes, or a combination thereof?

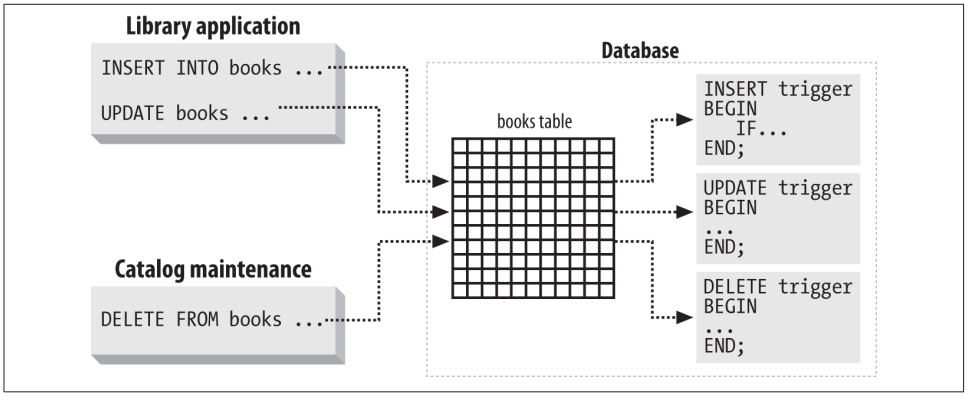

Figure 19-1. DML trigger flow of control

DML Trigger Concepts

Before diving into the syntax and examples, you may find it useful to review these DML trigger concepts and associated terminology:

BEFORE trigger
: A trigger that executes before a certain operation occurs, such as BEFORE INSERT.

AFTER trigger
: A trigger that executes after a certain operation occurs, such as AFTER UPDATE.

Statement-level trigger
: A trigger that executes for a SQL statement as a whole (which may, in turn, affect one or more individual rows in a database table).

Row-level trigger
: A trigger that executes for a single row that has been affected by the execution of a SQL statement. Suppose that the books table contains 1,000 rows. Then the following UPDATE statement will modify 1,000 rows:

```
UPDATE books SET title = UPPER (title);
```

And if I define a row-level update trigger on the books table, that trigger will fire 1,000 times.

NEW pseudo-record
: A data structure named NEW that looks like and (mostly) acts like a PL/SQL record. This pseudo-record is available only within update and insert DML triggers; it contains the values for the affected row after any changes were made.

OLD pseudo-record
: A data structure named OLD that looks like and (mostly) acts like a PL/SQL record. This pseudo-record is available only within update and delete DML triggers; it contains the values for the affected row before any changes were made.

WHEN clause
: The portion of the DML trigger that is run to determine whether or not the trigger code should be executed (allowing you to avoid unnecessary execution).

DML trigger scripts

To explore some of the concepts presented in the previous section, I have made the scripts in the following table available on the book's website.

Concept	Files	Description
Statement-level and row-level triggers	*copy_tables.sql*	Creates two identical tables, one with data and one empty.

	statement_vs_row.sql	Creates two simple triggers, one statement-level and one row-level. After running these scripts, execute this statement and view the results (with SERVEROUTPUT turned on to watch the activity):
		`INSERT INTO to_table`
		`  SELECT * FROM from_table;`

BEFORE and AFTER triggers	*before_vs_after.sql*	Creates BEFORE and AFTER triggers. After running the script, execute this statement and view the results:
		`INSERT INTO to_table`
		`  SELECT * FROM from_table;`

Triggers for various DML operations	*one_trigger_per_type.sql*	Creates AFTER INSERT, UPDATE, and DELETE triggers on to_table. After running the script, execute these commands and view the results:
		`INSERT INTO to_table`
		`  VALUES (1);`
		`UPDATE to_table`
		`  SET col1 = 10;`
		`DELETE to_table;`

Transaction participation

By default, DML triggers participate in the transaction from which they were fired. This means that:

- If a trigger terminates with an unhandled exception, then the statement that caused the trigger to fire is rolled back.
- If the trigger performs any DML itself (such as inserting a row into a log table), then that DML becomes a part of the main transaction.
- You cannot issue a COMMIT or ROLLBACK from within a DML trigger.

If you define your DML trigger to be an autonomous transaction (discussed in Chapter 14), however, then any DML performed inside the trigger will be saved or rolled back—with your explicit COMMIT or ROLLBACK statement—without affecting the main transaction.

The following sections present the syntax for creating a DML trigger, provide reference information on various elements of the trigger definition, and explore an example that uses the many components and options for these triggers.

Creating a DML Trigger

To create (or replace) a DML trigger, use the syntax shown here:

```
1    CREATE [OR REPLACE] TRIGGER trigger_name
2    {BEFORE | AFTER}
3    {INSERT | DELETE | UPDATE | UPDATE OF column_list } ON table_name
4    [FOR EACH ROW]
5    [WHEN (...)]
6    [DECLARE ... ]
7    BEGIN
8      ...executable statements...
9    [EXCEPTION ... ]
10   END [trigger_name];
```

The following table provides an explanation of these different elements.

Line(s)	Description
1	States that a trigger is to be created with the name supplied. Specifying OR REPLACE is optional. If the trigger exists and REPLACE is not specified, then your attempt to create the trigger anew will result in an ORA-4081 error. It is possible, by the way, for a table and a trigger (or a procedure and a trigger, for that matter) to have the same name. I recommend, however, that you adopt naming conventions to avoid the confusion that will result from this sharing of names.
2	Specifies if the trigger is to fire BEFORE or AFTER the statement or row is processed.
3	Specifies the combination of DML types to which the trigger applies: insert, update, and/or delete. Note that UPDATE can be specified for the whole record or just for a column list, separated by commas. The columns can be combined (separated with an OR) and may be specified in any order. Line 3 also specifies the table to which the trigger is to apply. Remember that each DML trigger can apply to only one table.
4	If FOR EACH ROW is specified, then the trigger will activate for each row processed by a statement. If this clause is missing, the default behavior is to fire only once for the statement (a statement-level trigger).
5	An optional WHEN clause that allows you to specify logic to avoid unnecessary execution of the trigger.
6	Optional declaration section for the anonymous block that constitutes the trigger code. If you do not need to declare local variables, you do not need this keyword. Note that you should never try to declare the NEW and OLD pseudo-records. This is done automatically.
7–8	The execution section of the trigger. This is required and must contain at least one statement.
9	Optional exception section. This section will trap and handle (or attempt to handle) any exceptions raised in the execution section only.

Line(s)	Description
10	Required END statement for the trigger. You can include the name of the trigger after the END keyword to explicitly document which trigger you are ending.

Here are a few examples of DML trigger usage:

- I want to make sure that whenever an employee is added or changed, all necessary validation is run. Notice that I pass the necessary fields of the NEW pseudo-record to individual check routines in this row-level trigger:

```
TRIGGER validate_employee_changes
   AFTER INSERT OR UPDATE
   ON employees
   FOR EACH ROW
BEGIN
   check_date (:NEW.hire_date);
   check_email (:NEW.email);
END;
```

- The following BEFORE INSERT trigger captures audit information for the CEO compensation table. It also relies on the autonomous transaction feature to commit this new row without affecting the "outer" or main transaction:

```
TRIGGER bef_ins_ceo_comp
   BEFORE INSERT
   ON ceo_compensation
   FOR EACH ROW
DECLARE
   PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
   INSERT INTO ceo_comp_history
        VALUES (:NEW.name,
                :OLD.compensation, :NEW.compensation,
                'AFTER INSERT', SYSDATE);
   COMMIT;
END;
```

The WHEN clause

Use the WHEN clause to fine-tune the situations under which the body of the trigger code will actually execute. In the following example, I use the WHEN clause to make sure that the trigger code does not execute unless the new salary is changing to a *different* value:

```
TRIGGER check_raise
   AFTER UPDATE OF salary
   ON employees
   FOR EACH ROW
WHEN ((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))
BEGIN
   ...
```

In other words, if a user issues an UPDATE to a row and for some reason sets the salary to its current value, the trigger will and must fire, but the reality is that you really don't need any of the PL/SQL code in the body of the trigger to execute. By checking this condition in the WHEN clause, you avoid some of the overhead of starting up the PL/SQL block associated with the trigger.

The *genwhen.sp* file on the book's website offers a procedure that will generate a WHEN clause to ensure that the new value is actually different from the old.

In most cases, you will reference fields in the OLD and NEW pseudo-records in the WHEN clause, as in the example just shown. You may also, however, write code that invokes built-in functions, as in the following WHEN clause that uses SYSDATE to restrict the INSERT trigger to fire only between 9 a.m. and 5 p.m.:

```
TRIGGER valid_when_clause
BEFORE INSERT ON frame
FOR EACH ROW
WHEN ( TO_CHAR(SYSDATE,'HH24') BETWEEN 9 AND 17 )
   ...
```

Here are some things to keep in mind when using the WHEN clause:

- Enclose the entire logical expression inside parentheses. These parentheses are optional in an IF statement, but required in the trigger WHEN clause.
- Do *not* include the ":" in front of the OLD and NEW names. This colon (indicating a host variable) is required in the body of the trigger PL/SQL code, but cannot be used in the WHEN clause.
- You can invoke SQL built-in functions only from within the WHEN clause; you will not be able to call user-defined functions or functions defined in built-in packages (such as DBMS_UTILITY). Attempts to do so will generate an *ORA-04076: invalid NEW or OLD specification* error. If you need to invoke such functions, move that logic to the beginning of the trigger execution section.

The WHEN clause can be used only with row-level triggers. You will get a compilation error (ORA-04077) if you try to use it with statement-level triggers.

Working with NEW and OLD pseudo-records

Whenever a row-level trigger fires, the PL/SQL runtime engine creates and populates two data structures that function much like records. They are the NEW and OLD pseudo-records ("pseudo" because they don't share all the properties of real PL/SQL records). OLD stores the original values of the record being processed by the trigger; NEW contains the new values. These records have the same structure as a record declared using %ROWTYPE on the table to which the trigger is attached.

Here are some rules to keep in mind when working with NEW and OLD:

- With triggers on INSERT operations, the OLD structure does not contain any data; there *is* no "old" set of values.
- With triggers on UPDATE operations, both the OLD and NEW structures are populated. OLD contains the values prior to the update; NEW contains the values the row will contain after the update is performed.
- With triggers on DELETE operations, the NEW structure does not contain any data; the record is about to be erased.
- The NEW and OLD pseudo-records also contain the ROWID pseudocolumn; this value is populated in both OLD and NEW with the same value, in all circumstances. Go figure!
- You cannot change the field values of the OLD structure; attempting to do so will raise the ORA-04085 error. You *can* modify the field values of the NEW structure.
- You can't pass a NEW or OLD structure as a "record parameter" to a procedure or function called within the trigger. You can pass only individual fields of the pseudo-record. See the *gentrigrec.sp* script on the book's website for a program that will generate code transferring NEW and OLD values to records that *can* be passed as parameters.
- When referencing the NEW and OLD structures within the anonymous block for the trigger, you must preface those keywords with a colon, as in:

```
IF :NEW.salary > 10000 THEN...
```

- You cannot perform record-level operations with the NEW and OLD structures. For example, the following statement will cause the trigger compilation to fail:

```
BEGIN :new := NULL; END;
```

You can also use the REFERENCING clause to change the names of the pseudo-records within the database trigger; this allows you to write code that is more self-documenting and application-specific. Here is one example:

```
/* File on web: full_old_and_new.sql */
TRIGGER audit_update
   AFTER UPDATE
```

```
   ON frame
   REFERENCING OLD AS prior_to_cheat NEW AS after_cheat
   FOR EACH ROW
BEGIN
   INSERT INTO frame_audit
               (bowler_id,
                game_id,
                old_score,
                new_score,
                change_date,
                operation)

        VALUES (:after_cheat.bowler_id,
                :after_cheat.game_id,
                :prior_to_cheat.score,
                :after_cheat.score,
                SYSDATE,
                'UPDATE');
END;
```

Run the *full_old_and_new.sql* script available on the book's website to take a look at the behavior of the OLD and NEW pseudo-records.

Determining the DML action within a trigger

Oracle provides a set of functions (also known as *operational directives*) that allow you to determine which DML action caused the firing of the current trigger. Each of these functions returns TRUE or FALSE, as described next:

INSERTING
: Returns TRUE if the trigger was fired by an insert into the table to which the trigger is attached, and FALSE if not.

UPDATING
: Returns TRUE if the trigger was fired by an update of the table to which the trigger is attached, and FALSE if not.

DELETING
: Returns TRUE if the trigger was fired by a delete from the table to which the trigger is attached, and FALSE if not.

Using these directives, it's possible to create a single trigger that consolidates the actions required for each different type of operation. Here's one such trigger:

```
/* File on web: one_trigger_does_it_all.sql */
TRIGGER three_for_the_price_of_one
BEFORE DELETE OR INSERT OR UPDATE ON account_transaction
FOR EACH ROW
BEGIN
  -- track who created the new row
  IF INSERTING
```

```
  THEN
    :NEW.created_by := USER;
    :NEW.created_date := SYSDATE;

  -- track deletion with special audit program
  ELSIF DELETING
  THEN
     audit_deletion(USER,SYSDATE);

  -- track who last updated the row
  ELSIF UPDATING
  THEN
    :NEW.UPDATED_BY := USER;
    :NEW.UPDATED_DATE := SYSDATE;
  END IF;
END;
```

The UPDATING function is overloaded with a version that takes a specific column name as an argument. This is handy for isolating specific column updates:

```
/* File on web: overloaded_update.sql */
TRIGGER validate_update
BEFORE UPDATE ON account_transaction
FOR EACH ROW
BEGIN
  IF UPDATING ('ACCOUNT_NO')
  THEN
    errpkg.raise('Account number cannot be updated');
  END IF;
END;
```

Specification of the column name is not case sensitive. The name is not evaluated until the trigger executes, and if the column does not exist in the table to which the trigger is attached, it will evaluate to FALSE.

Operational directives can be called from within any PL/SQL block, not just triggers. They will, however, only evaluate to TRUE within a DML trigger or code called from within a DML trigger.

DML Trigger Example: No Cheating Allowed!

One application function for which triggers are perfect is change auditing. Consider the example of Paranoid Pam (or Ms. Trustful, as we call her), who runs a bowling alley and has been receiving complaints about people cheating on their scores. She recently implemented a complete Oracle application known as Pam's Bowlerama Scoring System, and now wants to augment it to catch the cheaters.

The focal point of Pam's application is the frame table that records the score of a particular frame of a particular game for a particular player:

```
/* File on web: bowlerama_tables.sql */
TABLE frame
(bowler_id    NUMBER,
 game_id      NUMBER,
 frame_number NUMBER,
 strike       VARCHAR2(1) DEFAULT 'N',
 spare        VARCHAR2(1) DEFAULT 'N',
 score        NUMBER,
 CONSTRAINT frame_pk
 PRIMARY KEY (bowler_id, game_id, frame_number))
```

Pam enhances the frame table with an audit version to catch all before and after values, so that she can compare them and identify fraudulent activity:

```
TABLE frame_audit
(bowler_id    NUMBER,
 game_id      NUMBER,
 frame_number NUMBER,
 old_strike   VARCHAR2(1),
 new_strike   VARCHAR2(1),
 old_spare    VARCHAR2(1),
 new_spare    VARCHAR2(1),
 old_score    NUMBER,
 new_score    NUMBER,
 change_date  DATE,
 operation    VARCHAR2(6))
```

For every change to the frame table, Pam would like to keep track of before and after images of the affected rows. So she creates the following simple audit trigger:

```
    /* File on web: bowlerama_full_audit.sql */
 1  TRIGGER audit_frames
 2  AFTER INSERT OR UPDATE OR DELETE ON frame
 3  FOR EACH ROW
 4  BEGIN
 5    IF INSERTING THEN
 6      INSERT INTO frame_audit(bowler_id,game_id,frame_number,
 7                              new_strike,new_spare,new_score,
 8                              change_date,operation)
 9      VALUES(:NEW.bowler_id,:NEW.game_id,:NEW.frame_number,
10             :NEW.strike,:NEW.spare,:NEW.score,
11             SYSDATE,'INSERT');
12
13     ELSIF UPDATING THEN
14       INSERT INTO frame_audit(bowler_id,game_id,frame_number,
15                               old_strike,new_strike,
16                               old_spare,new_spare,
17                               old_score,new_score,
18                               change_date,operation)
19       VALUES(:NEW.bowler_id,:NEW.game_id,:NEW.frame_number,
```

```
20              :OLD.strike,:NEW.strike,
21              :OLD.spare,:NEW.spare,
22              :OLD.score,:NEW.score,
23              SYSDATE,'UPDATE');
24
25      ELSIF DELETING THEN
26        INSERT INTO frame_audit(bowler_id,game_id,frame_number,
27                                old_strike,old_spare,old_score,
28                                change_date,operation)
29        VALUES(:OLD.bowler_id,:OLD.game_id,:OLD.frame_number,
30              :OLD.strike,:OLD.spare,:OLD.score,
31              SYSDATE,'DELETE');
32      END IF;
33    END audit_frames;
```

Notice that for the INSERTING clause (lines 6–11), she relies on the NEW pseudo-record to populate the audit row. For UPDATING (lines 14–23), a combination of NEW and OLD information is used. For DELETING (lines 26–31), Pam has only OLD information with which to work. With this trigger in place, Pam can sit back and wait for action.

Of course, Pam doesn't announce her new auditing system. In particular, Sally Johnson (a very ambitious but not terribly skilled bowler) has no idea she is being watched. Sally has decided that she really wants to be the champion this year, and will stop at nothing to make it happen. Her father owns the bowling alley, she has access to SQL*Plus, and she knows that her bowler ID is 1. All that constitutes enough privilege and information to allow her to bypass the application GUI altogether, connect directly into SQL*Plus, and work some very unprincipled "magic."

Sally starts out by giving herself a strike in the first frame:

```
SQL> INSERT INTO frame
  2  (BOWLER_ID,GAME_ID,FRAME_NUMBER,STRIKE)
  3  VALUES(1,1,1,'Y');
1 row created.
```

But then she decides to be clever. She immediately downgrades her first frame to a spare to be less conspicuous:

```
SQL> UPDATE frame
  2  SET strike = 'N',
  3      spare = 'Y'
  4  WHERE bowler_id = 1
  5    AND game_id = 1
  6    AND frame_number = 1;
1 row updated.
```

Uh oh! Sally hears a noise in the corridor. She loses her nerve and tries to cover her tracks:

```
SQL> DELETE frame
  2  WHERE bowler_id = 1
  3    AND game_id = 1
  4    AND frame_number = 1;
1 row deleted.

SQL> COMMIT;
Commit complete.
```

She even verifies that her entries were deleted:

```
SQL> SELECT * FROM frame;
no rows selected
```

Wiping the sweat from her brow, Sally signs out, but vows to come back later and follow through on her plans.

Ever suspicious, Pam signs in and quickly discovers what Sally was up to by querying the audit table (Pam might also consider setting up an hourly job via DBMS_JOB to automate this part of the auditing procedure):

```
SELECT bowler_id, game_id, frame_number
     , old_strike, new_strike
     , old_spare, new_spare
     , change_date, operation
  FROM frame_audit
```

Here is the output:

```
BOWLER_ID GAME_ID FRAME_NUMBER O N O N CHANGE_DA OPERAT
--------- ------- ------------ - - - - --------- ------
        1       1            1   Y   N 12-SEP-00 INSERT
        1       1            1 Y N N Y 12-SEP-00 UPDATE
        1       1            1 N   N   12-SEP-00 DELETE
```

Sally is so busted! The audit entries show what Sally was up to even though no changes remain behind in the frame table. All three statements were audited by Pam's DML trigger: the initial insert of a strike entry, the downgrade to a spare, and the subsequent removal of the record.

Applying the WHEN clause

After using her auditing system for many successful months, Pam undertakes an effort to further isolate potential problems. She reviews her application frontend and determines that the strike, spare, and score fields are the only ones that can be changed. Thus, her trigger can be more specific:

```
TRIGGER audit_update
   AFTER UPDATE OF strike, spare, score
   ON frame
   REFERENCING OLD AS prior_to_cheat NEW AS after_cheat
   FOR EACH ROW
```

```
BEGIN
   INSERT INTO frame_audit (...)
     VALUES (...);
END;
```

After a few weeks of this implementation, Pam is still not happy with the auditing situation because audit entries are being created even when values are set equal to themselves. Updates like this one are producing useless audit records that show nothing changing:

```
SQL> UPDATE FRAME
  2  SET strike = strike;
   1 row updated.

SQL> SELECT old_strike,
  2         new_strike,
  3          old_spare,
  4          new_spare,
  5          old_score,
  6          new_score
  7    FROM frame_audit;

O N O N  OLD_SCORE  NEW_SCORE
- - - - ---------- ----------
Y Y N N
```

Pam needs to further isolate the trigger so that it fires only when values actually change. She does this using the WHEN clause shown here:

```
/* File on web: final_audit.sql */
TRIGGER audit_update
AFTER UPDATE OF STRIKE, SPARE, SCORE ON FRAME
REFERENCING OLD AS prior_to_cheat NEW AS after_cheat
FOR EACH ROW
WHEN ( prior_to_cheat.strike != after_cheat.strike OR
       prior_to_cheat.spare != after_cheat.spare OR
       prior_to_cheat.score != after_cheat.score )
BEGIN
  INSERT INTO FRAME_AUDIT ( ... )
     VALUES ( ... );
END;
```

Now entries will appear in the audit table only if something did indeed change, allowing Pam to quickly identify possible cheaters. Pam performs a quick final test of her trigger:

```
SQL> UPDATE frame
  2  SET strike = strike;
1 row updated.

SQL> SELECT old_strike,
  2         new_strike,
  3         old_spare,
```

```
  4         new_spare,
  5         old_score,
  6         new_score
  7    FROM frame_audit;
no rows selected
```

Using pseudo-records to fine-tune trigger execution

Pam has implemented an acceptable level of auditing in her system; now she'd like to make it a little more user-friendly. Her most obvious idea is to have her system add 10 to the score for frames recording a strike or spare. This allows the scorekeeper to track only the score for subsequent bowls while the system adds the strike score:

```
/* File on web: set_score.sql */
TRIGGER set_score
BEFORE INSERT ON frame
FOR EACH ROW
WHEN ( NEW.score IS NOT NULL )
BEGIN
  IF :NEW.strike = 'Y' OR :NEW.spare = 'Y'
  THEN
    :NEW.score := :NEW.score + 10;
  END IF;
END;
```

Remember that field values in the NEW records can be changed only in BEFORE row triggers.

Being a stickler for rules, Pam decides to add score validation to her set of triggers:

```
/* File on web: validate_score.sql */
TRIGGER validate_score
   AFTER INSERT OR UPDATE
   ON frame
   FOR EACH ROW
BEGIN
   IF     :NEW.strike = 'Y' AND :NEW.score < 10
   THEN
      RAISE_APPLICATION_ERROR (
         -20001,
         'ERROR: Score For Strike Must Be >= 10'
      );
   ELSIF :NEW.spare = 'Y' AND :NEW.score < 10
   THEN
      RAISE_APPLICATION_ERROR (
         -20001,
         'ERROR: Score For Spare Must Be >= 10'
      );
```

```
   ELSIF :NEW.strike = 'Y' AND :NEW.spare = 'Y'
   THEN
      RAISE_APPLICATION_ERROR (
         -20001,
         'ERROR: Cannot Enter Spare And Strike'
      );
   END IF;
END;
```

Now when there is any attempt to insert a row that violates this condition, it will be rejected:

```
SQL> INSERT INTO frame VALUES(1,1,1,'Y',NULL,5);
  2 INSERT INTO frame
            *
ERROR at line 1:
ORA-20001: ERROR: Score For Strike Must >= 10
```

Multiple Triggers of the Same Type

Above and beyond all of the options presented for DML triggers, it is also possible to have multiple triggers of the same type attached to a single table. Switching from bowling to golf, consider the following example that provides a simple commentary on a golf score by determining its relationship to a par score of 72.

A single row-level BEFORE INSERT trigger would suffice:

```
/* File on web: golf_commentary.sql */
TRIGGER golf_commentary
   BEFORE INSERT
   ON golf_scores
   FOR EACH ROW
DECLARE
   c_par_score   CONSTANT PLS_INTEGER := 72;
BEGIN
   :new.commentary :=
      CASE
         WHEN :new.score < c_par_score THEN 'Under'
         WHEN :new.score = c_par_score THEN NULL
         ELSE 'Over' END || ' Par'
END;
```

However, the requirement could also be satisfied with three separate row-level BEFORE INSERT triggers with mutually exclusive WHEN clauses:

```
TRIGGER golf_commentary_under_par
BEFORE INSERT ON golf_scores
FOR EACH ROW
WHEN (NEW.score < 72)
BEGIN
  :NEW.commentary := 'Under Par';
END;
```

```
TRIGGER golf_commentary_par
BEFORE INSERT ON golf_scores
FOR EACH ROW
WHEN (NEW.score = 72)
BEGIN
  :NEW.commentary := 'Par';
END;

TRIGGER golf_commentary_over_par
BEFORE INSERT ON golf_scores
FOR EACH ROW
WHEN (NEW.score > 72)
BEGIN
  :NEW.commentary := 'Over Par';
END;
```

Both implementations are perfectly acceptable and have advantages and disadvantages. A single trigger is easier to maintain because all of the code is in one place, while separate triggers reduce parse and execution time when more complex processing is required.

Who Follows Whom

Prior to Oracle Database 11*g*, there was no way to guarantee the order in which multiple DML triggers would fire. While this is not a concern in the previous example, it could be a problem in others, as shown in the next example.

What values will be shown by the final query?

```
/* File on web: multiple_trigger_seq.sql */
TABLE incremented_values
(value_inserted    NUMBER,
 value_incremented NUMBER);

TRIGGER increment_by_one
BEFORE INSERT ON incremented_values
FOR EACH ROW
BEGIN
  :NEW.value_incremented := :NEW.value_incremented + 1;
END;

TRIGGER increment_by_two
BEFORE INSERT ON incremented_values
FOR EACH ROW
BEGIN
  IF :NEW.value_incremented > 1 THEN
    :NEW.value_incremented := :NEW.value_incremented + 2;
  END IF;
END;

INSERT INTO incremented_values
 VALUES(1,1);
```

```
SELECT *
   FROM incremented_values;
```

Any guesses? On my database I got this result:

```
SQL> SELECT *
  2    FROM incremented_values;

VALUE_INSERTED VALUE_INCREMENTED
-------------- -----------------
             1                 2
```

So, the increment_by_two trigger fired first and did nothing because the value_incremented column was not greater than 1; then the increment_by_one trigger fired to increase the value_incremented column by 1. Is this the result you will receive? The preceding example offers no guarantee. Will this always be the result on my database? Again, there is no guarantee. Prior to Oracle Database 11*g*, Oracle explicitly stated that there was no way to control or assure the order in which multiple triggers of the same type on a single table would fire. There are many theories, the most prevalent being that triggers fire in reverse order of creation or by order of object ID—but even those theories cannot be relied upon.

Starting with Oracle Database 11*g*, however, you can guarantee the firing order using the FOLLOWS clause, as shown in the following example:

```
TRIGGER increment_by_two
BEFORE INSERT ON incremented_values
FOR EACH ROW
FOLLOWS increment_by_one
BEGIN
  IF :new.value_incremented > 1 THEN
    :new.value_incremented := :new.value_incremented + 2;
  END IF;
END;
```

Now this trigger is guaranteed to fire after the increment_by_one trigger does, thus guaranteeing the final result of the insert as well:

```
SQL> INSERT INTO incremented_values
  2  VALUES(1,1);
1 row created.
SQL> SELECT *
  2    FROM incremented_values;
VALUE_INSERTED VALUE_INCREMENTED
-------------- -----------------
             1                 4
```

The increment_by_one trigger made the inserted value 2, and then the increment_by_two trigger bumped it up to 4. This will always be the behavior because it is specified within the trigger itself—no need to rely on theories.

The link between triggers and their followers is viewable as a reference dependency in the dependencies view of the Oracle data dictionary:

```
SQL> SELECT referenced_name,
  2         referenced_type,
  3         dependency_type
  4    FROM user_dependencies
  5   WHERE name = 'INCREMENT_BY_TWO'
  6     AND referenced_type = 'TRIGGER';
REFERENCED_NAME    REFERENCED_TYPE    DEPE
------------------ ------------------ ----
INCREMENT_BY_ONE   TRIGGER            REF
```

Despite the behavior I've described here for Oracle Database 11*g*, triggers will not follow blindly—attempts to compile a trigger to follow one that is undefined are met with this error message:

```
Trigger "SCOTT"."BLIND_FOLLOWER" referenced in FOLLOWS or PRECEDES clause may
not exist
```

Mutating Table Errors

When something mutates, it is changing. Something that is changing is hard to analyze and to quantify. A mutating table error (ORA-4091) occurs when a row-level trigger tries to examine or change a table that is already undergoing change (via an INSERT, UPDATE, or DELETE statement).

In particular, this error occurs when a row-level trigger attempts to read or write the table from which the trigger was fired. Suppose, for example, that I want to put a special check on my employees table to make sure that when a person is given a raise, that person's new salary is not more than 20% above the next-highest salary in his department.

I would therefore like to write a trigger like this:

```
TRIGGER brake_on_raises
   BEFORE UPDATE OF salary ON employees
   FOR EACH ROW
DECLARE
   l_curr_max NUMBER;
BEGIN
   SELECT MAX (salary) INTO l_curr_max
     FROM employees
     WHERE department_id = :NEW.department_id;
   IF l_curr_max * 1.20 < :NEW.salary
   THEN
      errpkg.RAISE (
         employee_rules.en_salary_increase_too_large,
         :NEW.employee_id,
         :NEW.salary
      );
```

```
    END IF;
END;
```

But when I try to perform an update that, say, doubles the salary of the PL/SQL programmer (yours truly), I get this error:

```
ORA-04091: table SCOTT.EMPLOYEE is mutating, trigger/function may not see it
```

Here are some guidelines to keep in mind regarding mutating table errors:

- In general, a row-level trigger may not read or write the table from which it has been fired. The restriction applies only to row-level triggers, however. Statement-level triggers are free to both read and modify the triggering table; this fact gives us a way to avoid the mutating table error.
- If you make your trigger an autonomous transaction (by adding the PRAGMA AUTONOMOUS TRANSACTION statement and committing inside the body of the trigger), then you will be able to *query* the contents of the firing table. However, you will still not be allowed to modify the contents of the table.

Because each release of the Oracle database renders mutating tables less and less of a problem, it's not really necessary to perform a full demonstration here. However, a demonstration script named *mutation_zone.sql* is available on the book's website. In addition, the file *mutating_template.sql* offers a package that can serve as a template for creating your own package to defer processing of row-level logic to the statement level.

Compound Triggers: Putting It All in One Place

The age-old saying, "I finally got it all together, but I forgot where I put it" often applies to triggers. As you create more and more triggers containing more and more business logic, it becomes difficult to recall which triggers handle which rules and how all of the triggers interact. In the previous section I demonstrated how the three types of DML (insert, update, delete) can be put together in a single trigger, but wouldn't it be nice to be able to put row and statement triggers together in the same code object as well? Starting with Oracle Database 11*g*, you can use the compound trigger to do just that.

Here's a very simple example to show the syntax:

```
    /* File on web: compound_trigger.sql */
 1  TRIGGER compounder
 2  FOR UPDATE OR INSERT OR DELETE ON incremented_values
 3  COMPOUND TRIGGER
 4
 5    v_global_var NUMBER := 1;
 6
 7    BEFORE STATEMENT IS
 8    BEGIN
 9      DBMS_OUTPUT.PUT_LINE('Compound:BEFORE S:' || v_global_var);
10      v_global_var := v_global_var + 1;
```

```
11    END BEFORE STATEMENT;
12
13    BEFORE EACH ROW IS
14    BEGIN
15      DBMS_OUTPUT.PUT_LINE('Compound:BEFORE R:' || v_global_var);
16      v_global_var := v_global_var + 1;
17    END BEFORE EACH ROW;
18
19    AFTER EACH ROW IS
20    BEGIN
21      DBMS_OUTPUT.PUT_LINE('Compound:AFTER  R:' || v_global_var);
22      v_global_var := v_global_var + 1;
23    END AFTER EACH ROW;
24
25    AFTER STATEMENT IS
26    BEGIN
27      DBMS_OUTPUT.PUT_LINE('Compound:AFTER  S:' || v_global_var);
28      v_global_var := v_global_var + 1;
29    END AFTER STATEMENT;
30
31  END;
```

Just like a package

Compound triggers look a lot like PL/SQL packages, don't they? All of the related code and logic is in one place, making it easy to debug and modify. Let's look at the syntax in detail.

The most obvious change is the COMPOUND TRIGGER statement, which advises Oracle that this trigger contains many triggers that it will need to make fire together.

The next (and most eagerly awaited) change appears somewhat innocently on line 5: a global variable! Finally, global variables can be defined together with the code that manages them—no more special packages to manage them like this:

```
PACKAGE BODY yet_another_global_package AS
  v_global_var NUMBER := 1;
  PROCEDURE reset_global_var IS
  ...
END;
```

The remaining compound trigger syntax is very similar to that of standalone triggers, but a bit more rigid:

BEFORE STATEMENT
: The code in this section will fire before a DML statement executes, just like a standalone BEFORE statement trigger does.

BEFORE EACH ROW
: The code in this section gets executed before each and every row is processed by the DML statement.

AFTER EACH ROW
: The code in this section gets executed after each and every row is processed by the DML statement.

AFTER STATEMENT
: The code in this section will fire after a DML statement executes, just like a standalone AFTER statement trigger does.

The rules for standalone triggers apply to compound triggers as well—for example, record values (OLD and NEW) cannot be modified in statement-level triggers.

Not just like a package

So compound triggers look like packages, but do they behave in the same way? The short answer is no—they behave better! Consider this example:

```
SQL> BEGIN
  2    insert into incremented_values values(1,1);
  3    insert into incremented_values values(2,2);
  4  END;
  5  /
Compound:BEFORE S:1
Compound:BEFORE R:2
Compound:AFTER  R:3
Compound:AFTER  S:4
Compound:BEFORE S:1
Compound:BEFORE R:2
Compound:AFTER  R:3
Compound:AFTER  S:4

PL/SQL procedure successfully completed.
```

Notice that the output of the global variable was set back to 1 when the second statement executed. That's because the scope of the compound trigger is the DML statement that fires it. Once that statement completes, the compound trigger and its in-memory values start anew. That simplifies the logic.

A further benefit of the tight scoping is simplified error handling. I'll demonstrate by putting a primary key on the table just so I can try to violate it later:

```
SQL> ALTER TABLE incremented_values
  2   add constraint a_pk
  3   primary key ( value_inserted );
```

Now to insert one record:

```
SQL> INSERT INTO incremented_values values(1,1);
Compound:BEFORE S:1
Compound:BEFORE R:2
Compound:AFTER  R:3
Compound:AFTER  S:4
```

```
1 row created.
```

No surprises so far. But the next INSERT should throw an error because it violates the new primary key:

```
SQL> INSERT INTO incremented_values values(1,1);
Compound:BEFORE S:1
Compound:BEFORE R:2
insert into incremented_values values(1,1)
*
ERROR at line 1:
ORA-00001: unique constraint (SCOTT.A_PK) violated
```

The next INSERT also throws the primary key error again, as expected. But that is not what's exceptional about this situation—what's exceptional is that the global variable was reinitialized back to 1 without any extra code having to be written. The firing DML completed, so the compound trigger fell out of scope and everything started anew for the next statement:

```
SQL> INSERT INTO incremented_values values(1,1);
Compound:BEFORE S:1
Compound:BEFORE R:2
insert into incremented_values values(1,1)
*
ERROR at line 1:
ORA-00001: unique constraint (DRH.A_PK) violated
```

I don't need to include extra exception handling or packages just to reset the values when exceptions occur.

Compound following

Compound triggers also can be used with the FOLLOWS syntax:

```
TRIGGER follows_compounder
BEFORE INSERT ON incremented_values
FOR EACH ROW
FOLLOWS compounder
BEGIN
   DBMS_OUTPUT.PUT_LINE('Following Trigger');
END;
```

Here's the output:

```
SQL> INSERT INTO incremented_values
  2  values(8,8);
Compound:BEFORE S:1
Compound:BEFORE R:2
Following Trigger
Compound:AFTER  R:3
Compound:AFTER  S:4
```

```
1 row created.
```

The specific triggers within the compound trigger cannot be defined to fire after any standalone or compound triggers.

If a standalone trigger is defined to follow a compound trigger that does not contain a trigger to fire on the same statement or row, then the FOLLOWS clause is simply ignored.

DDL Triggers

Oracle allows you to define triggers that will fire when DDL statements are executed. Simply put, a DDL statement is any SQL statement used to create or modify a database object such as a table or an index. Here are some examples of DDL statements:

- CREATE TABLE
- ALTER INDEX
- DROP TRIGGER

Each of these statements results in the creation, alteration, or removal of a database object.

The syntax for creating these triggers is remarkably similar to that of DML triggers, except that the firing events differ, and they are not applied to individual tables.

The INSTEAD OF CREATE TABLE trigger, described at the end of this section, allows the default behavior of a CREATE TABLE event to be manipulated and is a somewhat idiosyncratic DDL trigger. Not all of the aspects of syntax and usage described in the following subsections apply to this trigger type.

Creating a DDL Trigger

To create (or replace) a DDL trigger, use the syntax shown here:

```
1    CREATE [OR REPLACE] TRIGGER trigger name
2    {BEFORE | AFTER} {DDL event} ON {DATABASE | SCHEMA}
3    [WHEN (...)]
4    DECLARE
5    Variable declarations
6    BEGIN
7    ...some code...
8    END;
```

The following table summarizes what is happening in this code.

Line(s)	Description
1	This line specifies that a trigger is to be created with the name supplied. Specifying OR REPLACE is optional. If the trigger exists, and REPLACE is not specified, then good old Oracle error ORA-4081 will appear stating just that.
2	This line has a lot to say. It defines whether the trigger will fire before, after, or instead of the particular DDL event, as well as whether it will fire for all operations within the database or just within the current schema. Note that the INSTEAD OF option is available only in Oracle9*i* Release 1 and higher.
3	An optional WHEN clause that allows you to specify logic to avoid unnecessary execution of the trigger.
4–7	These lines simply demonstrate the PL/SQL contents of the trigger.

Here's an example of a somewhat uninformed town crier trigger that announces the creation of all objects:

```
/* File on web: uninformed_town_crier.sql */
SQL> CREATE OR REPLACE TRIGGER town_crier
  2  AFTER CREATE ON SCHEMA
  3  BEGIN
  4    DBMS_OUTPUT.PUT_LINE('I believe you have created something!');
  5  END;
  6  /
Trigger created.

SQL> SET SERVEROUTPUT ON
SQL> CREATE TABLE a_table
  2  (col1 NUMBER);
Table created.

SQL> CREATE INDEX an_index ON a_table(col1);
Index created.

SQL> CREATE FUNCTION a_function RETURN BOOLEAN AS
  2  BEGIN
  3    RETURN(TRUE);
  4  END;
  5  /
Function created.

SQL> /* flush the DBMS_OUTPUT buffer */
SQL> BEGIN NULL; END;
  2  /
I believe you have created something!
I believe you have created something!
I believe you have created something!

PL/SQL procedure successfully completed.
```

Text displayed using the DBMS_OUTPUT built-in package within DDL triggers will not display until you successfully execute a PL/SQL block, even if that block does nothing.

Over time, this town crier would be ignored due to a lack of information, always proudly announcing that something had been created but never providing any details. Thankfully, there is a lot more information available to DDL triggers, allowing for a much more nuanced treatment, as shown in this version:

```
/* File on web: informed_town_crier.sql */
SQL> CREATE OR REPLACE TRIGGER town_crier
  2  AFTER CREATE ON SCHEMA
  3  BEGIN
  4    -- use event attributes to provide more info
  5    DBMS_OUTPUT.PUT_LINE('I believe you have created a ' ||
  6                         ORA_DICT_OBJ_TYPE || ' called ' ||
  7                         ORA_DICT_OBJ_NAME);
  8  END;
  9  /
Trigger created.

SQL> SET SERVEROUTPUT ON
SQL> CREATE TABLE a_table
  2  col1 NUMBER);
Table created.

SQL> CREATE INDEX an_index ON a_table(col1);
Index created.

SQL> CREATE FUNCTION a_function RETURN BOOLEAN AS
  2  BEGIN
  3    RETURN(TRUE);
  4  END;
  5  /
Function created.

SQL> /* flush the DBMS_OUTPUT buffer */

SQL> BEGIN NULL; END;
  2 /
I believe you have created a TABLE called A_TABLE
I believe you have created a INDEX called AN_INDEX
I believe you have created a FUNCTION called A_FUNCTION

PL/SQL procedure successfully completed.
```

Much more attention will be paid now that the town crier is more forthcoming. The preceding examples touch upon two important aspects of DDL triggers: the specific events to which they can be applied and the event attributes available within the triggers.

Available Events

Table 19-1 lists the DDL events for which triggers can be coded. Each event can have a BEFORE and an AFTER trigger.

Table 19-1. Available DDL events

DDL event	Fires when...
ALTER	Any database object is altered via the SQL ALTER command.
ANALYZE	Any database object is analyzed via the SQL ANALYZE command.
ASSOCIATE STATISTICS	Statistics are associated with a database object.
AUDIT	Auditing is turned on via the SQL AUDIT command.
COMMENT	Comments are applied to a database object.
CREATE	Any database object is created via the SQL CREATE command.
DDL	Any of the events listed here occur.
DISASSOCIATE STATISTICS	Statistics are disassociated from a database object.
DROP	Any database object is dropped via the SQL DROP command.
GRANT	Privileges are granted via the SQL GRANT command.
NOAUDIT	Auditing is turned off via the SQL NOAUDIT command.
RENAME	A database object is renamed via the SQL RENAME command.
REVOKE	Privileges are revoked via the SQL REVOKE command.
TRUNCATE	A table is truncated via the SQL TRUNCATE command.

As with DML triggers, these DDL triggers fire when the events to which they are attached occur within the specified database or schema. There is no limit to the number of trigger types that can exist in the database or schema.

Available Attributes

Oracle provides a set of functions (defined in the DBMS_STANDARD package) that provide information about what fired the DDL trigger and other information about the trigger state (e.g., the name of the table being dropped). Table 19-2 displays these trigger attribute functions. The following sections offer some examples of usage.

Table 19-2. DDL trigger event and attribute functions

Name	Returns...
ORA_CLIENT_IP_ADDRESS	IP address of the client.
ORA_DATABASE_NAME	Name of the database.

Name	Returns...
ORA_DES_ENCRYPTED_PASSWORD	DES-encrypted password of the current user.
ORA_DICT_OBJ_NAME	Name of the database object affected by the firing DDL.
ORA_DICT_OBJ_NAME_LIST	Count of objects affected. It also returns a complete list of objects affected in the NAME_LIST parameter, which is a collection of type DBMS_STANDARD.ORA_NAME_LIST_T.
ORA_DICT_OBJ_OWNER	Owner of the database object affected by the firing DDL statement.
ORA_DICT_OBJ_OWNER_LIST	Count of objects affected. It also returns a complete list of object owners affected in the NAME_LIST parameter, which is a collection of type DBMS_STANDARD.ORA_NAME_LIST_T.
ORA_DICT_OBJ_TYPE	Type of database object affected by the firing DDL statement (e.g., TABLE or INDEX).
ORA_GRANTEE	Count of grantees. The USER_LIST argument contains the full list of grantees, which is a collection of type DBMS_STANDARD.ORA_NAME_LIST_T.
ORA_INSTANCE_NUM	Number of the database instance.
ORA_IS_ALTER_COLUMN	TRUE if the specified COLUMN_NAME argument is being altered, or FALSE if not.
ORA_IS_CREATING_NESTED_TABLE	TRUE if a nested table is being created, or FALSE if not.
ORA_IS_DROP_COLUMN	TRUE if the specified COLUMN_NAME argument is indeed being dropped, or FALSE if not.
ORA_LOGIN_USER	Name of the Oracle user for which the trigger fired.
ORA_PARTITION_POS	Position in the SQL command where a partitioning clause could be correctly added.
ORA_PRIVILEGE_LIST	Number of privileges being granted or revoked. The PRIVILEGE_LIST argument contains the full list of privileges affected, which is a collection of type DBMS_STANDARD.ORA_NAME_LIST_T.
ORA_REVOKEE	Count of revokees. The USER_LIST argument contains the full list of revokees, which is a collection of type DBMS_STANDARD.ORA_NAME_LIST_T.
ORA_SQL_TXT	Number of lines in the SQL statement firing the trigger. The SQL_TXT argument returns each line of the statement, which is an argument of type DBMS_STANDARD.ORA_NAME_LIST_T.
ORA_SYSEVENT	Type of event that caused the DDL trigger to fire (e.g., CREATE, DROP, or ALTER).
ORA_WITH_GRANT_OPTION	TRUE if privileges were granted with the GRANT option, or FALSE if not.

Note the following about the event and attribute functions:

- The datatype ORA_NAME_LIST_T is defined in the DBMS_STANDARD package as:

  ```
  TYPE ora_name_list_t IS TABLE OF VARCHAR2(64);
  ```

  In other words, this is a nested table of strings, each of which can contain up to 64 characters.
- The DDL trigger event and attribute functions are also defined in the DBMS_STANDARD package. Oracle creates a standalone function (which adds the "ORA_" prefix to the function name) for each of the packaged functions by executing the *$ORACLE_HOME/rdbms/dbmstrig.sql* script during database cre-

ation. In some releases of the Oracle database, there are errors in this script that cause the standalone functions to not be visible or executable. If you feel that these elements have not been properly defined, you should ask your DBA to check the script for problems and make the necessary corrections.

- The USER_SOURCE data dictionary view does not get updated until after both BEFORE and AFTER DDL triggers are fired. In other words, you cannot use these functions to provide a "before and after" version control system built entirely within the database and based on database triggers.

Working with Events and Attributes

The best way to demonstrate the possibilities offered by DDL trigger events and attributes is with a series of examples.

Here is a trigger that prevents any and all database objects from being created:

```
TRIGGER no_create
    AFTER CREATE ON SCHEMA
BEGIN
   RAISE_APPLICATION_ERROR (
      -20000,
      'ERROR : Objects cannot be created in the production database.'
   );
END;
```

After I install this trigger, attempts at creating anything meet with failure:

```
SQL> CREATE TABLE demo (col1 NUMBER);
*
ERROR at line 1:
ORA-20000: Objects cannot be created in the production database.
```

That is a rather terse and uninformative error message. There was a failure, but what failed? Wouldn't it be nice to have a little more information in the error message, such as the object I was attempting to create? Consider this version:

```
/* File on web: no_create.sql */
TRIGGER no_create
AFTER CREATE ON SCHEMA
BEGIN
  RAISE_APPLICATION_ERROR (-20000,
       'Cannot create the ' || ORA_DICT_OBJ_TYPE ||
       ' named '            || ORA_DICT_OBJ_NAME ||
       ' as requested by '  || ORA_DICT_OBJ_OWNER ||
       ' in production.');
END;
```

With this trigger installed, an attempt to create my table now offers much more diagnostic information:

```
SQL> CREATE TABLE demo (col1 NUMBER);
*
ERROR at line 1:
ORA-20000: Cannot create the TABLE named DEMO as requested by SCOTT in production
```

I could even place this logic within a BEFORE DDL trigger and take advantage of the ORA_SYSEVENT attribute to respond to specific events:

```
TRIGGER no_create
BEFORE DDL ON SCHEMA
BEGIN
  IF ORA_SYSEVENT = 'CREATE'
  THEN
    RAISE_APPLICATION_ERROR (-20000,
       'Cannot create the ' || ORA_DICT_OBJ_TYPE ||
       ' named '            || ORA_DICT_OBJ_NAME ||
       ' as requested by '  || ORA_DICT_OBJ_OWNER);
  ELSIF ORA_SYSEVENT = 'DROP'
  THEN
     -- Logic for DROP operations
     ...
  END IF;
END;
```

What column did I touch?

I can use the ORA_IS_ALTER_COLUMN function to decipher which column was altered by an ALTER TABLE statement. Here is one example:

```
/* File on web: preserve_app_cols.sql  */
TRIGGER preserve_app_cols
   AFTER ALTER ON SCHEMA
DECLARE
   -- Cursor to get columns in a table
   CURSOR curs_get_columns (cp_owner VARCHAR2, cp_table VARCHAR2)
   IS
      SELECT column_name
        FROM all_tab_columns
       WHERE owner = cp_owner AND table_name = cp_table;
BEGIN
   -- If it was a table that was altered...
   IF ora_dict_obj_type = 'TABLE'
   THEN
      -- for every column in the table...
      FOR v_column_rec IN curs_get_columns (
                             ora_dict_obj_owner,
                             ora_dict_obj_name
                          )
      LOOP
         -- Is the current column one that was altered?
         IF ORA_IS_ALTER_COLUMN (v_column_rec.column_name)
         THEN
            -- Reject change to "core application" column
```

```
            IF mycheck.is_application_column (
                  ora_dict_obj_owner,
                  ora_dict_obj_name,
                  v_column_rec.column_name
               )
            THEN
               CENTRAL_ERROR_HANDLER (
                  'FAIL',
                  'Cannot alter core application attributes'
               );
            END IF; -- table/column is core
         END IF; -- current column was altered
      END LOOP; -- every column in the table
   END IF; -- table was altered
END;
```

Attempts to change core application attributes will now be stopped.

Remember that this logic will not work when the trigger is fired for the addition of new columns. That column information is not yet visible in the data dictionary when the DDL trigger fires.

I can check for attempts to drop specific columns as follows:

```
IF ORA_IS_DROP_COLUMN ('COL2')
THEN
  do something!
ELSE
  do something else!
END IF;
```

The ORA_IS_DROP_COLUMN and ORA_IS_ALTER_COLUMN functions are blissfully unaware of the table to which the column is attached; they work on column name alone.

Lists returned by attribute functions

Some of the attribute functions return two pieces of data: a list of items and a count of items. For example, the ORA_GRANTEE function returns a list and a count of users that were granted a privilege, and the ORA_PRIVILEGE_LIST function returns a list and a count of privileges granted. These two functions are perfect for use in AFTER GRANT triggers. The *what_privs.sql* file available on the book's website offers an extended example of how to use both of these functions. Here is just a portion of the total code:

```
/* File on web: what_privs.sql */
TRIGGER what_privs
   AFTER GRANT ON SCHEMA
DECLARE
```

```
   v_grant_type     VARCHAR2 (30);
   v_num_grantees   BINARY_INTEGER;
   v_grantee_list   ora_name_list_t;
   v_num_privs      BINARY_INTEGER;
   v_priv_list      ora_name_list_t;
BEGIN
   -- Retrieve information about grant type and then the lists.
   v_grant_type := ORA_DICT_OBJ_TYPE;
   v_num_grantees := ORA_GRANTEE (v_grantee_list);
   v_num_privs := ORA_PRIVILEGE_LIST (v_priv_list);

   IF v_grant_type = 'ROLE PRIVILEGE'
   THEN
      DBMS_OUTPUT.put_line (
         'The following roles/privileges were granted');

      -- For each element in the list, display the privilege.
      FOR counter IN 1 .. v_num_privs
      LOOP
         DBMS_OUTPUT.put_line ('Privilege ' || v_priv_list (counter));
      END LOOP;
```

This trigger is great for detailing what privileges and objects are affected by grant operations, as shown next. In a more sophisticated implementation, you might consider storing this information in database tables so that you have a detailed history of changes that have occurred:

```
SQL> GRANT DBA TO book WITH ADMIN OPTION;
Grant succeeded.

SQL> EXEC DBMS_OUTPUT.PUT_LINE('Flush buffer');
        The following roles/privileges were granted
                Privilege UNLIMITED TABLESPACE
                Privilege DBA
        Grant Recipient BOOK
Flush buffer

SQL> GRANT SELECT ON x TO system WITH GRANT OPTION;
Grant succeeded.

SQL> EXEC DBMS_OUTPUT.PUT_LINE('Flush buffer');
        The following object privileges were granted
                Privilege SELECT
        On X with grant option
        Grant Recipient SYSTEM
Flush buffer
```

Dropping the Undroppable

I have shown that one use for DDL triggers is preventing a particular type of DDL operation on a particular object or type of object. But what if I create a trigger that

prevents DDL DROP operations and then attempt to drop the trigger itself? Will I be left with a trigger that is essentially undroppable? Fortunately, Oracle has thought of this scenario, as you can see here:

```
SQL> CREATE OR REPLACE TRIGGER undroppable
  2  BEFORE DROP ON SCHEMA
  3  BEGIN
  4    RAISE_APPLICATION_ERROR(-20000,'You cannot drop me! I am invincible!');
  5  END;

SQL> DROP TABLE employee;
*
ERROR at line 1:
ORA-20000: You cannot drop me! I am invincible!

SQL> DROP TRIGGER undroppable;
Trigger dropped.
```

When working with pluggable databases (Oracle Database 12*c* and higher), you can insert the word PLUGGABLE before DATABASE in your trigger definition. DATABASE (without PLUGGABLE) defines the trigger on the root. In a multitenant container database (CDB), only a common user who is connected to the root can create a trigger on the entire database. PLUGGABLE DATABASE defines the trigger on the PDB to which you are connected. The trigger fires whenever any user of the specified database or PDB initiates the triggering event.

The INSTEAD OF CREATE Trigger

Oracle provides the INSTEAD OF CREATE trigger to allow you to automatically partition a table. To do so, the trigger must trap the SQL statement being executed, insert the partition clause into it, and then execute it using the ORA_SQL_TXT function. The following trigger demonstrates these steps:

```
/* File on web: io_create.sql */
TRIGGER io_create
    INSTEAD OF CREATE ON DATABASE
WHEN (ORA_DICT_OBJ_TYPE = 'TABLE')
DECLARE
   v_sql     VARCHAR2 (32767);  -- sql to be built
   v_sql_t   ora_name_list_t;   -- table of sql
BEGIN
   -- get the SQL statement being executed
   FOR counter IN 1 .. ora_sql_txt (v_sql_t)
   LOOP
      v_sql := v_sql || v_sql_t (counter);
   END LOOP;
```

```
   -- Determine the partition clause and add it.
   -- We will call the my_partition function.
   v_sql :=
         SUBSTR (v_sql, 1, ora_partition_pos)
      || magic_partition_function
      || SUBSTR (v_sql, ora_partition_pos + 1);

   /* Prepend table name with login username.
   |  Replace CRLFs with spaces.
   |  Requires an explicit CREATE ANY TABLE privilege,
   |  unless you switch to AUTHID CURRENT_USER.
   */
   v_sql :=
      REPLACE (UPPER (REPLACE (v_sql, CHR (10), ' '))
             , 'CREATE TABLE '
             , 'CREATE TABLE ' || ora_login_user || '.'
              );

   -- now execute the SQL
   EXECUTE IMMEDIATE v_sql;
END;
```

Now tables will be partitioned automatically, as determined by the logic in the my_partition function.

Oracle offers several partitioning options (e.g., range, hash) and logical partitioning choices (e.g., by primary key, by unique key). You must decide which of these you want to utilize in your partitioning function.

If you do not include the WHEN clause just shown, you will find that attempts to create objects that are *not* tables will fail with this error:

```
ORA-00604: error occurred at recursive SQL level 1
ORA-30511: invalid DDL operation in system triggers
```

Further, if you try to create an INSTEAD OF trigger for any other DDL operation besides CREATE, you will receive this compilation error:

```
ORA-30513: cannot create system triggers of INSTEAD OF type
```

INSTEAD OF triggers for DML operations (insert, update, and delete) are addressed later in this chapter. These triggers share some syntax with the INSTEAD OF CREATE trigger for tables, but that is the extent of their similarity.

Database Event Triggers

Database event triggers fire whenever database-wide events occur. There are eight database event triggers, two of which are new to Oracle Database 12*c*:

STARTUP
: Fires when the database is opened.

SHUTDOWN
: Fires when the database is shut down normally.

SERVERERROR
: Fires when an Oracle error is raised.

LOGON
: Fires when an Oracle database session begins.

LOGOFF
: Fires when an Oracle database session terminates normally.

DB_ROLE_CHANGE
: Fires when a standby database is changed to be the primary database, or vice versa.

AFTER_CLONE (Oracle Database 12c)
: Can be specified only if PLUGGABLE DATABASE is specified. After the pluggable database (PDB) is copied (cloned), the database fires the trigger in the new PDB and then deletes the trigger. If the trigger fails, then the copy operation fails.

BEFORE UNPLUG (Oracle Database 12c)
: Can be specified only if PLUGGABLE DATABASE is specified. Before the PDB is unplugged, the database fires the trigger and then deletes it. If the trigger fails, then the unplug operation fails.

As any DBA will immediately see, these triggers offer stunning possibilities for automated administration and very granular control.

Creating a Database Event Trigger

The syntax used to create these triggers is quite similar to that used for DDL triggers:

```
1    CREATE [OR REPLACE] TRIGGER trigger_name
2    {BEFORE | AFTER} {database_event} ON {DATABASE | SCHEMA}
3    DECLARE
4    Variable declarations
5    BEGIN
6    ...some code...
7    END;
```

There are restrictions regarding what events can be combined with what BEFORE and AFTER attributes. Some situations just don't make sense:

No BEFORE STARTUP triggers
: Even if such triggers could be created, when would they fire? Attempts to create triggers of this type will be met by this straightforward error message:

```
ORA-30500: database open triggers and server error triggers cannot have
BEFORE type
```

No AFTER SHUTDOWN triggers
: Again, when would they fire? Attempts to create such triggers are deflected with this message:

```
ORA-30501: instance shutdown triggers cannot have AFTER type
```

No BEFORE LOGON triggers
: It would require some amazingly perceptive code to implement these triggers: "Wait, I think someone is going to log on—do something!" Being strictly reality-based, Oracles stops these triggers with this message:

```
ORA-30508: client logon triggers cannot have BEFORE type
```

No AFTER LOGOFF triggers
: "No wait, please come back! Don't sign off!" Attempts to create such triggers are stopped with this message:

```
ORA-30509: client logoff triggers cannot have AFTER type
```

No BEFORE SERVERERROR
: These triggers would be every programmer's dream! Think of the possibilities...

```
CREATE OR REPLACE TRIGGER BEFORE_SERVERERROR
BEFORE SERVERERROR ON DATABASE
BEGIN
  diagnose_impending_error;
  fix_error_condition;
  continue_as_if_nothing_happened;
END;
```

  Unfortunately, our dreams are shattered by this error message:

```
ORA-30500: database open triggers and server error triggers cannot have
BEFORE type
```

The STARTUP Trigger

Startup triggers execute during database startup processing. This is a perfect place to perform housekeeping steps, such as pinning objects in the shared pool so that they do not "age out" with the least recently used algorithm.

In order to create startup event triggers, users must have been granted the ADMINISTER DATABASE TRIGGER privilege.

Here is an example of creating a STARTUP event trigger:

```
CREATE OR REPLACE TRIGGER startup_pinner
AFTER STARTUP ON DATABASE
BEGIN
  pin_plsql_packages;
  pin_application_packages;
END;
```

The SHUTDOWN Trigger

BEFORE SHUTDOWN triggers execute before database shutdown processing is performed. This is a great place to gather system statistics. Here is an example of creating a SHUTDOWN event trigger:

```
CREATE OR REPLACE TRIGGER before_shutdown
BEFORE SHUTDOWN ON DATABASE
BEGIN
  gather_system_stats;
END;
```

SHUTDOWN triggers execute only when the database is shut down using NORMAL or IMMEDIATE mode. They do not execute when the database is shut down using ABORT mode or when the database crashes.

The LOGON Trigger

AFTER LOGON triggers fire when an Oracle database session is begun. They are the perfect place to establish session context and perform other session setup tasks. Here is an example of creating a LOGON event trigger:

```
TRIGGER after_logon
AFTER LOGON ON SCHEMA
DECLARE
  v_sql VARCHAR2(100) := 'ALTER SESSION ENABLE RESUMABLE ' ||
                         'TIMEOUT 10 NAME ' || '''' ||
                         'OLAP Session'     || '''';
BEGIN
  EXECUTE IMMEDIATE v_sql;
  DBMS_SESSION.SET_CONTEXT('OLAP Namespace',
                           'Customer ID',
                           load_user_customer_id);
END;
```

The LOGOFF Trigger

BEFORE LOGOFF triggers execute when sessions disconnect normally from the database. This is a good place to gather statistics regarding session activity. Here is an example of creating a LOGOFF event trigger:

```
TRIGGER before_logoff
BEFORE LOGOFF ON DATABASE
BEGIN
  gather_session_stats;
END;
```

The SERVERERROR Trigger

AFTER SERVERERROR triggers fire after an Oracle error is raised, unless the error is one of the following:

ORA-00600
: Oracle internal error

ORA-01034
: Oracle not available

ORA-01403
: No data found

ORA-01422
: Exact fetch returns more than requested number of rows

ORA-01423
: Error encountered while checking for extra rows in an exact fetch

ORA-04030
: Out-of-process memory when trying to allocate *N* bytes

In addition, the AFTER SERVERERROR trigger will *not* fire when an exception is raised *inside* this trigger (to avoid an infinite recursive execution of the trigger).

AFTER SERVERERROR triggers do not provide facilities to fix the error, only to log information about the error. It is therefore possible to build some powerful logging mechanisms around these triggers.

Oracle also provides built-in functions (again, defined in DBMS_STANDARD) that retrieve information about the error stack generated when an exception is raised:

ORA_SERVER_ERROR
: Returns the Oracle error number at the specified position in the error stack. It returns 0 if no error is found at that position.

ORA_IS_SERVERERROR
: Returns TRUE if the specified error number appears in the current exception stack.

ORA_SERVER_ERROR_DEPTH
: Returns the number of errors on the stack.

ORA_SERVER_ERROR_MSG
: Returns the full text of the error message at the specified position. It returns NULL if no error is found at that position.

ORA_SERVER_ERROR_NUM_PARAMS
: Returns the number of parameters associated with the error message at the given position. It returns 0 if no error is found at that position.

ORA_SERVER_ERROR_PARAM
: Returns the value for the specified parameter position in the specified error. It returns NULL if none is found.

SERVERERROR examples

Let's look at some examples of using SERVERERROR triggers. I'll start with a very simple example of a SERVERERROR trigger that echoes the fact that an error occurred:

```
TRIGGER error_echo
AFTER SERVERERROR
ON SCHEMA
BEGIN
  DBMS_OUTPUT.PUT_LINE ('You experienced an error');
END;
```

Whenever an Oracle error occurs (assuming that SERVEROUTPUT is ON), the preceding coded message will display:

```
SQL> SET SERVEROUTPUT ON
SQL> EXEC DBMS_OUTPUT.PUT_LINE(TO_NUMBER('A'));
You experienced an error
BEGIN DBMS_OUTPUT.PUT_LINE(TO_NUMBER('A')); END;

*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character to number conversion error
ORA-06512: at line 1
```

Note that the Oracle error message was delivered after the trigger message. This allows the error to be accessed and logged prior to the actual failure, as shown in the next example.

SERVERERROR triggers are automatically isolated in their own autonomous transaction (autonomous transactions were covered in Chapter 14). This means that you can, for example, write error information out to a log table and save those changes with a COMMIT, while not affecting the session transaction in which the error occurred.

The error_logger trigger guarantees that information about all but a handful of errors (listed earlier) will be automatically logged, regardless of the application, user, or program in which the error was raised:

```
/* File on web: error_log.sql */
TRIGGER error_logger
AFTER SERVERERROR
ON SCHEMA
DECLARE

  v_errnum    NUMBER;          -- the Oracle error #
  v_now       DATE := SYSDATE; -- current time

BEGIN

  -- For every error in the error stack...
  FOR e_counter IN 1..ORA_SERVER_ERROR_DEPTH LOOP

    -- Write the error out to the log table; no
    -- commit is required because we are in an
    -- autonomous transaction.
    INSERT INTO error_log(error_id,
                          username,
                          error_number,
                          sequence,
                          timestamp)
    VALUES(error_seq.nextval,
           USER,
           ORA_SERVER_ERROR(e_counter),
           e_counter,
           v_now);

  END LOOP;  -- every error on the stack

END;
```

Remember that all these new rows in the error_log have been committed by the time the END statement is reached, because the trigger is executed within an autonomous transaction. The following lines demonstrate this trigger in action:

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE(TO_NUMBER('A'));
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character to number conversion error

SQL> SELECT * FROM error_log;

USERNAME                       ERROR_NUMBER   SEQUENCE TIMESTAMP
------------------------------ ------------ ---------- ---------
BOOK                                   6502          1 04-JAN-02
BOOK                                   6512          2 04-JAN-02
```

Why do two errors appear in the table when only one error was raised? The actual error stack generated by the database contains both ORA-06502 and ORA-06512, so they are both logged and denoted by their sequence of occurrence.

If you want to determine quickly if a certain error number is located in the stack without parsing it manually, use the companion function ORA_IS_SERVERERROR. This function is very useful for monitoring specific errors that may require extra handling, such as user-defined exceptions. This is the kind of code you might write:

```
-- Special handling of user-defined errors
-- 20000 through 20010 raised by calls to
-- RAISE_APPLICATION_ERROR

FOR errnum IN 20000 .. 20010
LOOP
  IF ORA_IS_SERVERERROR (errnum)
  THEN
     log_user_defined_error (errnum);
  END IF;
END LOOP;
```

All Oracle error numbers are negative, except for 1 (user-defined exception) and 100 (synonymous with 1403, NO_DATA_FOUND). When you specify an error number in the call to ORA_IS_SERVERERROR, however, you must supply a positive number, as shown in the previous example.

Central error handler

While it is possible to implement separate SERVERERROR triggers in every schema in a database, I recommend creating a single central trigger with an accompanying PL/SQL package to provide the following features:

Centralized error logging
: There is only one trigger and package to maintain and keep in Oracle's memory.

Session-long searchable error log
: The error log can be accumulated over the course of a session rather than error by error. It can be searched to return details like the number of occurrences, the timestamps of the first and last occurrence, etc. The log can also be purged on demand.

Option to save error log
: The error log can be saved to a permanent table in the database if desired.

Viewable current log
: The current log of errors is viewable by specific error number and/or date range.

You can find the implementation of one such centralized error-handling package in the *error_log.sql* file on the book's website. Once this package is in place, I can create the SERVERERROR trigger as follows:

```
CREATE OR REPLACE TRIGGER error_log
AFTER SERVERERROR
ON DATABASE
BEGIN
  central_error_log.log_error;
END;
```

Here are some example usages. First, I will generate an error:

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE(TO_NUMBER('A'));
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character to number conversion error
```

Now I can search for a specific error number and retrieve that information in a record:

```
DECLARE
  v_find_record central_error_log.v_find_record;
BEGIN
  central_error_log.find_error(6502,v_find_record);
  DBMS_OUTPUT.PUT_LINE('Total Found   = ' || v_find_record.total_found);
  DBMS_OUTPUT.PUT_LINE('Min Timestamp = ' || v_find_record.min_timestamp);
  DBMS_OUTPUT.PUT_LINE('Max Timestamp = ' || v_find_record.max_timestamp);
END;
```

The output is:

```
Total Found   = 1
Min Timestamp = 04-JAN-02
Max Timestamp = 04-JAN-02
```

INSTEAD OF Triggers

INSTEAD OF triggers control insert, update, merge, and delete operations on *views*, not tables. They can be used to make nonupdateable views updateable and to override the default behavior of views that are updateable.

Creating an INSTEAD OF Trigger

To create (or replace) an INSTEAD OF trigger, use the syntax shown here:

```
1    CREATE [OR REPLACE] TRIGGER trigger_name
2    INSTEAD OF operation
3    ON view_name
4    FOR EACH ROW
5    BEGIN
6      ...code goes here...
7    END;
```

The following table contains an explanation of this code.

Line(s)	Description
1	States that a trigger is to be created with the unique name supplied. Specifying OR REPLACE is optional. If the trigger exists, and REPLACE is not specified, then my attempt to create the trigger anew will result in an ORA-4081 error.
2	This is where we see differences between INSTEAD OF triggers and other types of triggers. Because INSTEAD OF triggers aren't really triggered by an event, I don't need to specify AFTER or BEFORE or provide an event name. What I do specify is the operation that the trigger is to fire in place of. Stating INSTEAD OF followed by one of INSERT, UPDATE, MERGE, or DELETE accomplishes this.
3	This line is somewhat like the corresponding line for DDL and database event triggers, in that the keyword ON is specified. The similarities end there: instead of specifying DATABASE or SCHEMA, I provide the name of the view to which the trigger is to apply.
4–7	Contains standard PL/SQL code.

INSTEAD OF triggers are best explained with an example. Let's use one of my favorite topics: pizza delivery! Before I can start pounding the dough, I have to put a system in place to monitor my deliveries. I will need three tables—one to track actual deliveries, one to track delivery areas, and one to track my massive fleet of drivers (remember the first rule of business—always think big!):

```
/* File on web: pizza_tables.sql */
CREATE TABLE delivery
(delivery_id NUMBER,
 delivery_start DATE,
 delivery_end DATE,
 area_id NUMBER,
 driver_id NUMBER);

CREATE TABLE area
   (area_id NUMBER, area_desc   VARCHAR2(30));

CREATE TABLE driver
   (driver_id NUMBER, driver_name   VARCHAR2(30));
```

For the sake of brevity, I will not create any primary or foreign keys.

I will also need three sequences to provide unique identifiers for my tables:

```
CREATE SEQUENCE delivery_id_seq;
CREATE SEQUENCE area_id_seq;
CREATE SEQUENCE driver_id_seq;
```

To avoid having to explain relational database design and normalization to my employees, I will simplify deliveries into a single view displaying delivery, area, and driver information:

```
VIEW delivery_info AS
SELECT d.delivery_id,
        d.delivery_start,
        d.delivery_end,
```

```
       a.area_desc,
       dr.driver_name
  FROM delivery      d,
       area          a,
       driver        dr
 WHERE a.area_id = d.area_id
   AND dr.driver_id = d.driver_id
```

Because my system relies heavily on this view for query functionality, why not make it available for insert, update, and delete operations as well? I cannot directly issue DML statements against the view; it is a join of multiple tables. How would the database know what to do with an INSERT? In fact, I need to tell the database very explicitly what to do when an insert, update, or delete is issued against the delivery_info view; in other words, I need to tell it what to do *instead of* trying to insert, update, or delete. Thus, I will use INSTEAD OF triggers. Let's start with the INSERT trigger.

The INSTEAD OF INSERT Trigger

My INSERT trigger will perform four basic operations:

- Ensure that the delivery_end value is NULL. All delivery completions must be done via an update.
- Try to find the driver ID based on the name provided. If the name cannot be found, then assign a new ID and create a driver entry using the name and the new ID.
- Try to find the area ID based on the name provided. If the name cannot be found, then assign a new ID and create an area entry using the name and the new ID.
- Create an entry in the delivery table.

Bear in mind that this example is intended to demonstrate triggers—not how to effectively build a business system! After a while I will probably wind up with a multitude of duplicate driver and area entries. However, using this view speeds things up by not requiring drivers and areas to be predefined, and in the fast-paced world of pizza delivery, time is money!

```
/* File on web: pizza_triggers.sql */
TRIGGER delivery_info_insert
   INSTEAD OF INSERT
   ON delivery_info
DECLARE
   -- cursor to get the driver ID by name
   CURSOR curs_get_driver_id (cp_driver_name VARCHAR2)
   IS
      SELECT driver_id
        FROM driver
       WHERE driver_name = cp_driver_name;

   v_driver_id   NUMBER;
```

```
   -- cursor to get the area ID by name
   CURSOR curs_get_area_id (cp_area_desc VARCHAR2)
   IS
      SELECT area_id
        FROM area
       WHERE area_desc = cp_area_desc;

   v_area_id     NUMBER;
BEGIN
 /*
   || Make sure the delivery_end value is NULL.
 */
   IF :NEW.delivery_end IS NOT NULL
   THEN
      raise_application_error
                (-20000
               , 'Delivery end date value must be NULL when delivery created'
                );
   END IF;

 /*
   || Try to get the driver ID using the name. If not found
   || then create a brand new driver ID from the sequence.
 */
 OPEN curs_get_driver_id (UPPER (:NEW.driver_name));

   FETCH curs_get_driver_id
    INTO v_driver_id;

   IF curs_get_driver_id%NOTFOUND
   THEN
      SELECT driver_id_seq.NEXTVAL
        INTO v_driver_id
        FROM DUAL;

      INSERT INTO driver
                  (driver_id, driver_name
                  )
           VALUES (v_driver_id, UPPER (:NEW.driver_name)
                  );
   END IF;

   CLOSE curs_get_driver_id;

  /*
    || Try to get the area ID using the name. If not found
    || then create a brand new area ID from the sequence.
   */
   OPEN curs_get_area_id (UPPER (:NEW.area_desc));

   FETCH curs_get_area_id
```

```
     INTO v_area_id;

   IF curs_get_area_id%NOTFOUND
   THEN
      SELECT area_id_seq.NEXTVAL
        INTO v_area_id
        FROM DUAL;

      INSERT INTO area
                  (area_id, area_desc
                  )
           VALUES (v_area_id, UPPER (:NEW.area_desc)
                  );
   END IF;

   CLOSE curs_get_area_id;

   /*
     || Create the delivery entry.
   */
   INSERT INTO delivery
               (delivery_id, delivery_start
               , delivery_end, area_id, driver_id
               )
        VALUES (delivery_id_seq.NEXTVAL, NVL (:NEW.delivery_start, SYSDATE)
               , NULL, v_area_id, v_driver_id
               );
END;
```

The INSTEAD OF UPDATE Trigger

Now let's move on to the UPDATE trigger. For the sake of simplicity, I will allow updating only of the delivery_end field, and only if it is NULL to start with. I can't have drivers resetting delivery times!

```
/* File on web: pizza_triggers.sql */
TRIGGER delivery_info_update
   INSTEAD OF UPDATE
   ON delivery_info
DECLARE
   -- cursor to get the delivery entry
   CURSOR curs_get_delivery (cp_delivery_id NUMBER)
   IS
      SELECT delivery_end
        FROM delivery
       WHERE delivery_id = cp_delivery_id
      FOR UPDATE OF delivery_end;

   v_delivery_end   DATE;
BEGIN
   OPEN curs_get_delivery (:NEW.delivery_id);
   FETCH curs_get_delivery INTO v_delivery_end;
```

```
   IF v_delivery_end IS NOT NULL
   THEN
      RAISE_APPLICATION_ERROR (
         -20000, 'The delivery end date has already been set');
   ELSE
      UPDATE delivery
         SET delivery_end = :NEW.delivery_end
       WHERE CURRENT OF curs_get_delivery;
   END IF;

   CLOSE curs_get_delivery;
END;
```

The INSTEAD OF DELETE Trigger

The DELETE trigger is the simplest of all. It merely ensures that I am not deleting a completed entry and then removes the delivery record. The driver and area records remain intact:

```
/* File on web: pizza_triggers.sql */
TRIGGER delivery_info_delete
   INSTEAD OF DELETE
   ON delivery_info
BEGIN
  IF :OLD.delivery_end IS NOT NULL
  THEN
     RAISE_APPLICATION_ERROR (
        -20000,'Completed deliveries cannot be deleted');
  END IF;

  DELETE delivery
   WHERE delivery_id = :OLD.delivery_id;
END;
```

Populating the Tables

Now, with a single INSERT focused on the delivery information I know (the driver and the area), all of the required tables are populated:

```
SQL> INSERT INTO delivery_info(delivery_id,
  2                            delivery_start,
  3                            delivery_end,
  4                            area_desc,
  5                            driver_name)
  6  VALUES
  7    NULL, NULL, NULL, 'LOCAL COLLEGE', 'BIG TED');

1 row created.

SQL> SELECT * FROM delivery;
```

```
DELIVERY_ID DELIVERY_ DELIVERY_    AREA_ID  DRIVER_ID
----------- --------- --------- ---------- ----------
          1 13-JAN-02                   1          1

SQL> SELECT * FROM area;

   AREA_ID AREA_DESC
---------- ------------------------------
         1 LOCAL COLLEGE

SQL> SELECT * FROM driver;

 DRIVER_ID DRIVER_NAME
---------- ------------------------------
         1 BIG TED
```

INSTEAD OF Triggers on Nested Tables

Oracle has introduced many ways to store complex data structures as columns in tables or views. This is logically effective because the linkage between a table or view and its columns is obvious. Technically, it can require some not-so-obvious trickery to allow even the simplest of operations, like inserting records into these complex structures. One of these complex situations can be resolved with a special type of INSTEAD OF trigger, as shown in this section.

Consider the following view joining each of the chapters of a book with the lines in that chapter:

```
VIEW book_chapter_view AS
SELECT chapter_number,
       chapter_title,
       CAST(MULTISET(SELECT *
                       FROM book_line
                      WHERE chapter_number = book_chapter.chapter_number)
            AS book_line_t) lines
  FROM book_chapter;
```

I agree that the view is far too obtuse for its purpose (why not just join the tables directly?), but it easily demonstrates the use of INSTEAD OF triggers on nested table columns—or on any object or collection column in a view.

After creating a record in the book_chapter table and querying the view, I'll see the following, which explains that there are no lines in the chapter yet:

```
CHAPTER_NUMBER CHAPTER_TITLE
-------------- ------------------------------
LINES(CHAPTER_NUMBER, LINE_NUMBER, LINE_TEXT)
---------------------------------------------
            18 Triggers
BOOK_LINE_T()
```

So I then try to create the first line to get past my writer's block:

```
SQL> INSERT INTO TABLE(SELECT lines
  2                      FROM book_chapter_view
  3                     WHERE chapter_number = 18)
  4  VALUES(18,1,'Triggers are...');
INSERT INTO TABLE(SELECT lines
*
ERROR at line 1:
ORA-25015: cannot perform DML on this nested table view column
```

Apparently, the database has determined that there is not enough information available to just insert values into the book_line table masquerading as the LINES column in the view. Thus, an INSTEAD OF trigger is required to make the intent crystal clear:

```
TRIGGER lines_ins
INSTEAD OF INSERT ON NESTED TABLE lines OF book_chapter_view
BEGIN
  INSERT INTO book_line
             (chapter_number,
              line_number,
              line_text)
  VALUES(:PARENT.chapter_number,
         :NEW.line_number,
         :NEW.line_text);
END;
```

Now I can add the first line:

```
SQL> INSERT INTO TABLE ( SELECT lines
  2                       FROM book_chapter_view
  3                      WHERE chapter_number = 18 )
  4  VALUES(18,1,'Triggers Are...');

1 row created.

SQL> SELECT *
  2    FROM book_chapter_view;

CHAPTER_NUMBER CHAPTER_TITLE
-------------- ------------------------------
LINES(CHAPTER_NUMBER, LINE_NUMBER, LINE_TEXT)
-----------------------------------------------------
            18 Triggers
BOOK_LINE_T(BOOK_LINE_O(18, 1, 'Triggers Are...'))
```

Note that the SQL used to create the trigger is just like what is used for other INSTEAD OF triggers, except for two things:

- The ON NESTED TABLE COLUMN OF clause used to denote the involved column
- The new PARENT pseudo-record containing values from the view's parent record

AFTER SUSPEND Triggers

Oracle9*i* Database Release 1 introduced a new type of trigger that fires whenever a statement is suspended. This might occur as the result of a space issue, such as exceeding an allocated tablespace quota. This functionality can be used to address the problem and allow the stalled operation to continue. AFTER SUSPEND triggers are a boon to busy developers tired of being held up by space errors, and to even busier DBAs who constantly have to resolve these errors.

The syntax used to create an AFTER SUSPEND trigger follows the same format as DDL and database event triggers. It declares the firing event (SUSPEND), the timing (AFTER), and the scope (DATABASE or SCHEMA):

```
1 CREATE [OR REPLACE] TRIGGER trigger_name
2 AFTER SUSPEND
3 ON {DATABASE | SCHEMA}
4 BEGIN
5 ...code...
6 END;
```

Let's take a closer look at AFTER SUSPEND, starting with an example of a scenario that would call for creation of this type of trigger.

Consider the situation faced by Batch Only, the star Oracle developer at Totally Controlled Systems. He is responsible for maintaining hundreds of programs that run overnight, performing lengthy transactions to summarize information and move it between disparate applications. At least twice a week, his pager goes off during the wee hours of the morning because one of his programs has encountered this Oracle error:

```
ERROR at line 1:
ORA-01536: space quota exceeded for tablespace 'USERS'
```

Batch then has the unenviable task of phoning Totally's Senior DBA, Don T. Planahead, and begging for a space quota increase. Don's usual question is, "How much do you need?" to which Batch can only feebly reply, "I don't know because the data load fluctuates so much." This leaves them both very frustrated, because Don wants control over the space allocation for planning reasons, and Batch doesn't want his night's sleep interrupted so often.

Setting Up for the AFTER SUSPEND Trigger

Thankfully, an AFTER SUSPEND trigger can eliminate the dark circles under both Don's and Batch's eyes. Here is how they work through the situation.

Batch discovers the particular point in his code that encounters the error most frequently. It is an otherwise innocuous INSERT statement at the end of a program that takes hours to run:

```
INSERT INTO monthly_summary (
   acct_no, trx_count, total_in, total_out)
VALUES (
   v_acct, v_trx_count, v_total_in, v_total_out);
```

What makes this most maddening is that the values take hours to calculate, only to be immediately lost when the final INSERT statement fails. At the very least, Batch wants the program to suspend itself while he contacts Don to get more space allocated. He discovers that this can be done with a simple ALTER SESSION statement:

```
ALTER SESSION ENABLE RESUMABLE TIMEOUT 3600 NAME 'Monthly Summary';
```

This means that whenever this Oracle database session encounters an out-of-space error, it will go into a suspended (and potentially resumable) state for 3,600 seconds (1 hour). This provides enough time for Totally's monitoring system to page Batch, for Batch to phone Don, and for Don to allocate more space. It's not a perfect system, but at least the hours spent calculating the data are no longer wasted.

Another problem faced by Batch and Don is that when they try to diagnose the situation in the middle of the night, they are both so tired and grumpy that time is wasted on misunderstandings. Thankfully, the need for explanations can be alleviated by another feature of suspended/resumable statements: the DBA_RESUMABLE view. This shows all sessions that have registered for resumable statements with the ALTER SESSION command just shown.

The RESUMABLE system privilege must be granted to users before they can enable the resumable option.

Now, whenever Batch's programs go into the suspended state, he only has to phone Don and mumble, "Check the resumable view." Don then queries it from his DBA account to see what is going on:

```
SQL> SELECT session_id,
  2         name,
  3         status,
  4         error_number
  5   FROM dba_resumable

SESSION_ID NAME                 STATUS    ERROR_NUMBER
---------- -------------------- --------- ------------
         8 Monthly Summary      SUSPENDED         1536

1 row selected.
```

This shows that session 8 is suspended because of *ORA-01536: space quota exceeded for tablespace <tablespace_name>*. From past experience, Don knows which schema and

tablespace are involved, so he corrects the problem and mumbles into the phone, "It's fixed." The suspended statement in Batch's code immediately resumes, and both Don and Batch can go back to sleep in their own beds.

Invalid DDL Operation in System Triggers

AFTER SUSPEND triggers are not allowed to actually perform certain DDL operations (ALTER USER and ALTER TABLESPACE) to fix the problems they diagnose. They simply raise the error *ORA-30511: invalid DDL operation in system triggers*. One way to work around this situation is as follows:

1. Have the AFTER SUSPEND trigger write the SQL statement necessary to fix a problem in a table.
2. Create a PL/SQL package that reads SQL statements from the table and executes them.
3. Submit the PL/SQL package to DBMS_JOB every minute or so.

Looking at the Actual Trigger

After a few weeks, both Don and Batch are tired of their repetitive (albeit abbreviated) late-night conversations, so Don sets out to automate things with an AFTER SUSPEND trigger. Here's a snippet of what he cooks up and installs in the DBA account:

```
/* File on web: smart_space_quota.sql */
TRIGGER after_suspend
AFTER SUSPEND
ON DATABASE
DECLARE
...
BEGIN

  -- if this is a space-related error...
  IF ORA_SPACE_ERROR_INFO ( error_type => v_error_type,
                            object_type => v_object_type,
                            object_owner => v_object_owner,
                            table_space_name => v_tbspc_name,
                            object_name => v_object_name,
                            sub_object_name => v_subobject_name ) THEN

    -- if the error is a tablespace quota being exceeded...
    IF v_error_type = 'SPACE QUOTA EXCEEDED' AND
       v_object_type = 'TABLE SPACE' THEN
      -- get the username
      OPEN curs_get_username;
      FETCH curs_get_username INTO v_username;
      CLOSE curs_get_username;
```

```
        -- get the current quota for the username and tablespace
        OPEN curs_get_ts_quota(v_object_name,v_username);
        FETCH curs_get_ts_quota INTO v_old_quota;
        CLOSE curs_get_ts_quota;

        -- create an ALTER USER statement and send it off to
        -- the fixer job because if we try it here we will raise
        -- ORA-30511: invalid DDL operation in system triggers

        v_new_quota := v_old_quota + 40960;
        v_sql := 'ALTER USER ' || v_username  || ' ' ||
                 'QUOTA '      || v_new_quota || ' ' ||
                 'ON '         || v_object_name;
        fixer.fix_this(v_sql);

      END IF;  -- tablespace quota exceeded

    END IF;  -- space-related error

  END;
```

This creates a trigger that fires whenever a statement enters a suspended state and attempts to fix the problem. (Note that this particular example handles only tablespace quotas being exceeded.)

Now when Batch's programs encounter the tablespace quota problem, the database-wide AFTER SUSPEND trigger fires and puts a SQL entry in the "stuff to fix" table via the fixer package. In the background, a fixer job is running; it picks the SQL statement out of the table and executes it, thus alleviating the quota problem without requiring anyone to pick up the phone.

A complete AFTER SUSPEND trigger and fixer package are available in the *fixer.sql* file on the book's website.

The ORA_SPACE_ERROR_INFO Function

You can garner information on the cause of the statement suspension using the ORA_SPACE_ERROR_INFO function shown in earlier examples. Now let's look at the syntax for specifying this function; the parameters are defined as shown in Table 19-3.

Table 19-3. ORA_SPACE_ERROR_INFO parameters

Parameter	Description
error_type	Type of space error; will be one of the following: • SPACE QUOTA EXCEEDED: if a user has exceeded her quota for a tablespace • MAX EXTENTS REACHED: if an object attempts to go beyond its maximum extents specification • NO MORE SPACE: if there is not enough space in a tablespace to store the new information
object_type	Type of object encountering the space error
object_owner	Owner of the object encountering the space error
table_space_name	Tablespace encountering the space error
object_name	Name of the object encountering the space error
sub_object_name	Name of the subobject encountering the space error

The function returns a Boolean value of TRUE if the suspension occurs because of one of the errors shown in the table, and FALSE if not.

The ORA_SPACE_ERROR_INFO function does not actually fix whatever space problems occur in your system; its role is simply to provide the information you need to take further action. In the earlier example, you saw how the quota error was addressed. Here are two additional examples of SQL you might supply to fix space problems diagnosed by the ORA_SPACE_ERROR_INFO function:

- Specify the following when your table or index has achieved its maximum extents and no more extents are available:

```
ALTER object_type object_owner.object_name STORAGE (MAXEXTENTS UNLIMITED);
```

- Specify the following when your tablespace is completely out of space:

```
/* Assume Oracle Managed Files (Oracle9i Database and later) being used so
   explicit datafile declaration not required */
ALTER TABLESPACE tablespace_name ADD DATAFILE;
```

The DBMS_RESUMABLE Package

If the ORA_SPACE_ERROR_INFO function returns FALSE, then the situation causing the suspended statement cannot be fixed. Thus, there is no rational reason for remaining suspended. You can abort unfixable statements from within the AFTER_SUSPEND trigger using the ABORT procedure in the DBMS_RESUMABLE package. The following provides an example of issuing this procedure:

```
/* File on web: local_abort.sql */
TRIGGER after_suspend
AFTER SUSPEND
ON SCHEMA
DECLARE
```

```
  CURSOR curs_get_sid IS
  SELECT sid
    FROM v$session
   WHERE audsid = SYS_CONTEXT('USERENV','SESSIONID');
  v_sid        NUMBER;
  v_error_type VARCHAR2(30);
  ...

BEGIN

  IF ORA_SPACE_ERROR_INFO(...
    ...try to fix things...
  ELSE  -- can't fix the situation
    OPEN curs_get_sid;
    FETCH curs_get_sid INTO v_sid;
    CLOSE curs_get_sid;
    DBMS_RESUMABLE.ABORT(v_sid);
  END IF;

END;
```

The ABORT procedure takes a single argument, the ID of the session to abort. This allows ABORT to be called from a DATABASE- or SCHEMA-level AFTER SUSPEND trigger. The aborted session receives this error:

```
ORA-01013: user requested cancel of current operation
```

After all, the cancellation was requested by a user—but exactly which user is unclear.

In addition to the ABORT procedure, the DBMS_RESUMABLE package contains functions and procedures to get and set timeout values:

GET_SESSION_TIMEOUT
: Returns the timeout value of the suspended session by session ID:

```
FUNCTION DBMS_RESUMABLE.GET_SESSION_TIMEOUT (sessionid IN NUMBER)
      RETURN NUMBER;
```

SET_SESSION_TIMEOUT
: Sets the timeout value of the suspended session by session ID:

```
PROCEDURE DBMS_RESUMABLE.SET_SESSION_TIMEOUT (sessionid IN NUMBER, TIMEOUT IN NUMBER);
```

GET_TIMEOUT
: Returns the timeout value of the current session:

```
FUNCTION DBMS_RESUMABLE.GET_TIMEOUT RETURN NUMBER;
```

SET_TIMEOUT
: Sets the timeout value of the current session:

```
PROCEDURE DBMS_REUSABLE.SET_TIMEOUT (TIMEOUT IN NUMBER);
```

New timeout values take effect immediately but do not reset the counter to zero.

Trapped Multiple Times

AFTER SUSPEND triggers fire whenever a statement is suspended. Therefore, they can fire many times during the same statement. For example, suppose that the following hardcoded trigger is implemented:

```
/* File on web: increment_extents.sql */
TRIGGER after_suspend
AFTER SUSPEND ON SCHEMA
DECLARE
  -- get the new max (current plus one)
  CURSOR curs_get_extents IS
  SELECT max_extents + 1
    FROM user_tables
   WHERE table_name = 'MONTHLY_SUMMARY';
  v_new_max NUMBER;

BEGIN
  - fetch the new maximum extent value
  OPEN curs_get_extents;
  FETCH curs_get_extents INTO v_new_max;
  CLOSE curs_get_extents;

  -- alter the table to take on the new value for maxextents
  EXECUTE IMMEDIATE 'ALTER TABLE MONTHLY_SUMMARY ' ||
                    'STORAGE ( MAXEXTENTS '        ||
                    v_new_max                      || ')';

  DBMS_OUTPUT.PUT_LINE('Incremented MAXEXTENTS to ' || v_new_max);
END;
```

If you start with an empty table with MAXEXTENTS (maximum number of extents) specified as 1, inserting four extents' worth of data produces this output:

```
SQL> @test

Incremented MAXEXTENTS to 2
Incremented MAXEXTENTS to 3
Incremented MAXEXTENTS to 4

PL/SQL procedure successfully completed.
```

To Fix or Not to Fix?

That is the question! The previous examples have shown how "lack of space" errors can be handled on the fly by suspending statements until intervention (human or automated) allows them to continue. Taken to an extreme, this approach allows applications to be installed with minimal tablespace, quota, and extent settings, and then to grow as required. While over-diligent DBAs may see this situation as nirvana, it does have its downsides:

Intermittent pauses
: Suspended statement pauses may wreak havoc with high-volume online transaction processing (OLTP) applications that require high throughput levels. This will be even more troublesome if the fix takes a long time.

Resource contention
: Suspended statements maintain their table locks, which may cause other statements to wait for long periods of time or fail needlessly.

Management overhead
: The resources required to continuously add extents or datafiles or increment quotas may wind up overwhelming those required to actually run the application.

For these reasons, I recommend that AFTER SUSPEND triggers be used judiciously. They are perfect for long-running processes that must be restarted after failure, as well as for incremental processes that require DML to undo their changes before they can be restarted. However, they are not well suited to OLTP applications.

Maintaining Triggers

Oracle offers a number of DDL statements that can help you manage your triggers. As explained in the following sections, you can enable, disable, and drop triggers; view information about triggers; and check the status of triggers.

Disabling, Enabling, and Dropping Triggers

Disabling a trigger causes it not to fire when its triggering event occurs. Dropping a trigger causes it to be removed from the database altogether. The SQL syntax for disabling triggers is relatively simple compared to that for creating them:

```
ALTER TRIGGER trigger_name DISABLE;
```

For example:

```
ALTER TRIGGER emp_after_insert DISABLE;
```

A disabled trigger can also be reenabled, as shown in the following example:

```
ALTER TRIGGER emp_after_insert ENABLE;
```

The ALTER TRIGGER command is concerned only with the trigger name; it does not require identifying the trigger type or anything else. You can also easily create stored procedures to handle these steps for you. The following procedure, for example, uses dynamic SQL to disable or enable all triggers on a table:

```
/* File on web: settrig.sp */
PROCEDURE settrig (
   tab      IN   VARCHAR2
 , sch      IN   VARCHAR DEFAULT NULL
 , action   IN   VARCHAR2
)
IS
   l_action         VARCHAR2 (10) := UPPER (action);
   l_other_action   VARCHAR2 (10) := 'DISABLED';
BEGIN
   IF l_action = 'DISABLE'
   THEN
      l_other_action := 'ENABLED';
   END IF;

   FOR rec IN (SELECT trigger_name FROM user_triggers
               WHERE table_owner = UPPER (NVL (sch, USER))
                 AND table_name = tab AND status = l_other_action)
   LOOP
      EXECUTE IMMEDIATE
        'ALTER TRIGGER ' || rec.trigger_name || ' ' || l_action;
   END LOOP;
END;
```

The DROP TRIGGER command is just as easy; simply specify the trigger name, as shown in this example:

```
DROP TRIGGER emp_after_insert;
```

Creating Disabled Triggers

Starting with Oracle Database 11*g*, it is possible to create triggers in a disabled state so they don't fire. This is very helpful in situations where you want to validate a trigger but don't want it to start firing just yet. Here's a very simple example:

```
TRIGGER just_testing
AFTER INSERT ON abc
DISABLE
BEGIN
  NULL;
END;
```

Because the DISABLE keyword is included in the header, this trigger gets validated, compiled, and created, but it will not fire until it is explicitly enabled later on. Note that the DISABLE keyword is not present in what gets saved into the database, though:

```
SQL> SELECT trigger_body
  2  FROM user_triggers
  3  WHERE trigger_name = 'JUST_TESTING';

TRIGGER_BODY
----------------------------------------
BEGIN
  NULL;
END;
```

When you are using a GUI tool, be careful to avoid accidentally enabling triggers when they are recompiled.

Viewing Triggers

You can find out lots of information about triggers by issuing queries against the following data dictionary views:

DBA_TRIGGERS
: All triggers in the database

ALL_TRIGGERS
: All triggers accessible to the current user

USER_TRIGGERS
: All triggers owned by the current user

Table 19-4 summarizes the most useful (and common) columns in these views.

Table 19-4. Useful columns in trigger views

Name	Description
TRIGGER_NAME	Name of the trigger
TRIGGER_TYPE	Type of the trigger; you can specify: • For DML triggers: BEFORE_STATEMENT, BEFORE EACH ROW, AFTER EACH ROW, or AFTER STATEMENT • For DDL triggers: BEFORE EVENT or AFTER EVENT • For INSTEAD OF triggers: INSTEAD OF • For AFTER_SUSPEND triggers: AFTER EVENT
TRIGGERING_EVENT	Event that causes the trigger to fire: • For DML triggers: UPDATE, INSERT, or DELETE • For DDL triggers: DDL operation (see full list in "DDL Triggers" on page 710) • For database event triggers: ERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN • For INSTEAD OF triggers: INSERT, UPDATE, or DELETE • For AFTER SUSPEND triggers: SUSPEND

Name	Description
TABLE_OWNER	Contains different information depending on the type of trigger: • For DML triggers: name of the owner of the table to which the trigger is attached • For DDL triggers: if database-wide, then SYS; otherwise, the owner of the trigger • For database event triggers: if database-wide, then SYS; otherwise, the owner of the trigger • For INSTEAD OF triggers: owner of the view to which the trigger is attached • For AFTER SUSPEND triggers: if database-wide, then SYS; otherwise, the owner of the trigger
BASE_OBJECT_TYPE	Type of object to which the trigger is attached: • For DML triggers: TABLE • For DDL triggers: SCHEMA or DATABASE • For database event triggers: SCHEMA or DATABASE • For INSTEAD OF triggers: VIEW • For AFTER SUSPEND triggers: SCHEMA or DATABASE
TABLE_NAME	For DML triggers: name of the table the trigger is attached to; for other types of triggers: NULL
REFERENCING_NAMES	For DML (row-level) triggers: clause used to define the aliases for the OLD and NEW records; for other types of triggers: text "REFERENCING NEW AS NEW OLD AS OLD"
WHEN_CLAUSE	For DML triggers: trigger's conditional firing clause
STATUS	Trigger's status (ENABLED or DISABLED)
ACTION_TYPE	Indicates whether the trigger executes a call (CALL) or contains PL/SQL (PL/SQL)
TRIGGER_BODY	Text of the trigger body (LONG column); this information is also available in the USER_SOURCE table starting with Oracle9*i* Database

Checking the Validity of Triggers

Oddly enough, the trigger views in the data dictionary do not display whether or not a trigger is in a valid state. If a trigger is created with invalid PL/SQL, it is saved in the database but marked as INVALID. You can query the USER_OBJECTS or ALL_OBJECTS views to determine this status, as shown here:

```
SQL> CREATE OR REPLACE TRIGGER invalid_trigger
  2  AFTER DDL ON SCHEMA
  3  BEGIN
  4    NULL
  5  END;
  6  /

Warning: Trigger created with compilation errors.

SQL> SELECT object_name,
  2         object_type,
  3         status
  4    FROM user_objects
  5   WHERE object_name = 'INVALID_TRIGGER';
```

```
OBJECT_NAME     OBJECT TYPE STATUS
-------------   ----------- -------
INVALID_TRIGGER TRIGGER     INVALID
```