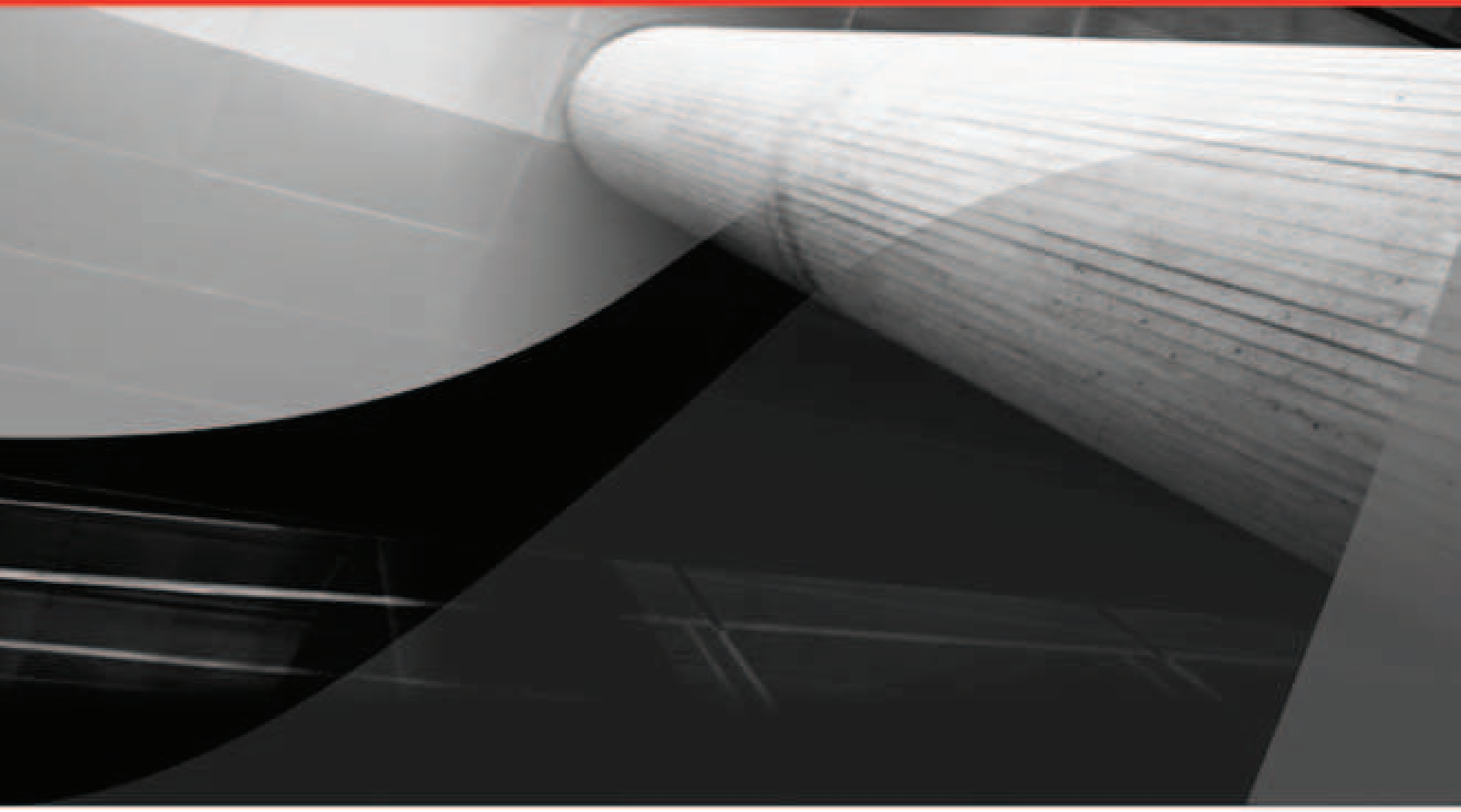
ORACLE

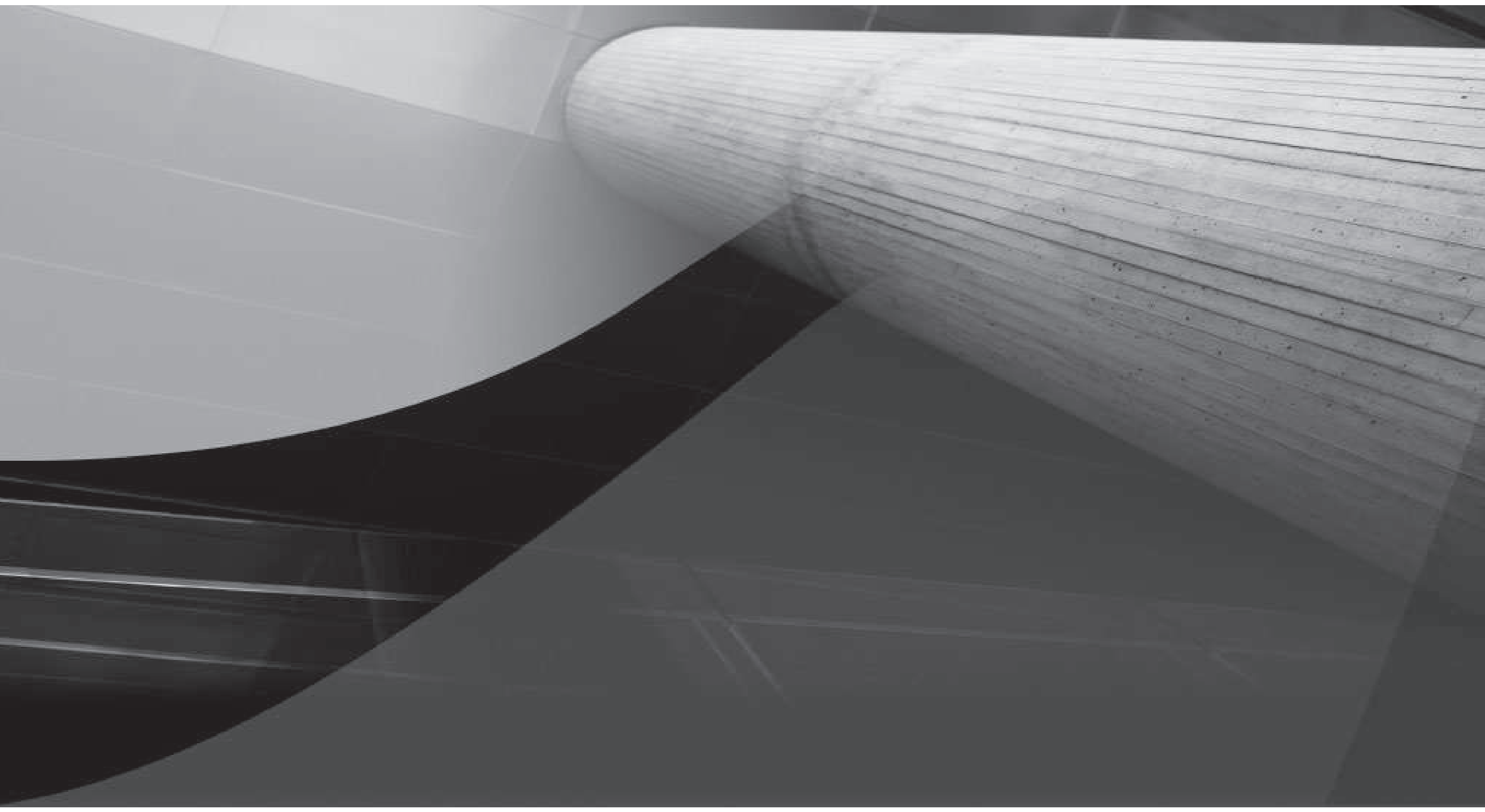


Oracle Database 11*g* PL/SQL Programming

Develop Robust, Database-Driven PL/SQL Applications

Oracle Press

Michael McLaughlin
Professor of Computer Information Technology, BYU-Idaho



CHAPTER 10

Triggers

Database *triggers* are specialized stored programs. They are not called directly but are triggered by events in the database. They run between the time you issue a command and the time you perform the database management system action. You can write triggers in PL/SQL or Java. Triggers can capture events that create, modify, or drop objects, and they can capture inserts to, updates of, and deletes from tables or views. They can also monitor changes in the state of the database or schema, and the actions of users.

This chapter covers the following:

- Introduction to triggers
- Trigger architecture
- Data Definition Language triggers
- Data Manipulation Language triggers
- Compound triggers
- Instead-of triggers
- System or database event triggers
- Trigger restrictions

The sections lay a foundation and develop ideas sequentially. They should also serve as a quick reference if you want to focus on writing a specific trigger type quickly. For example, you can go to the section "Data Manipulation Language Triggers" to learn how to write triggers for inserts, updates, and deletes.

Introduction to Triggers

Database *triggers* are specialized stored programs. As such, they are defined by very similar DDL rules. Likewise, triggers can call SQL statements and PL/SQL functions and procedures. You can choose to implement triggers in PL/SQL or Java. You can check Chapter 14 and Appendix D for clarification of syntax on writing Java libraries to support your triggers.

Database triggers differ from stored functions and procedures because you can't call them directly. Database triggers are fired when a triggering event occurs in the database. This makes them very powerful tools in your efforts to manage the database. You are able to limit or redirect actions through triggers.

You can do the following with triggers:

- Control the behavior of DDL statements, as by altering, creating, or renaming objects
- Control the behavior of DML statements, like inserts, updates, and deletes
- Enforce referential integrity, complex business rules, and security policies
- Control and redirect DML statements when they change data in a view
- Audit information of system access and behavior by creating transparent logs

On the other hand, you can't control the sequence of or synchronize calls to triggers, and this can present problems if you rely too heavily on triggers. The only control you have is to designate them as before or after certain events. Oracle 11*g* delivers compound triggers to help you manage larger events, like those triggering events that you would sequence.

There are risks with triggers. The risks are complex because while SQL statements fire triggers, triggers call SQL statements. A trigger can call a SQL statement that in turn fires another trigger. The subsequent trigger could repeat the behavior and fire another trigger. This creates *cascading triggers.* Oracle 11*g* and earlier releases limit the number of cascading trigger to 32, after which an exception is thrown.

The following summarizes the five types of triggers and their uses:

- **Data Definition Language triggers** These triggers fire when you create, change, or remove objects in a database schema. They are useful to control or monitor DDL statements. An *instead-of create* table trigger provides you with a tool to ensure table creation meets your development standards, like including storage or partitioning clauses. You can also use them to monitor poor programming practices, such as when programs *create* and *drop* temporary tables rather than use Oracle collections. Temporary tables can fragment disk space and degrade database performance over time.

- **Data Manipulation Language triggers** These triggers fire when you *insert, update,* or *delete* data from a table. You can fire them once for all changes on a table, or for each row change, using *statement-* or *row-level* trigger types, respectively. DML triggers are useful to control DML statements. You can use these triggers to audit, check, save, and replace values before they are changed. Automatic numbering of numeric *primary keys* is frequently done by using a row-level DML trigger.

- **Compound triggers** These triggers acts as both statement- and row-level triggers when you *insert, update,* or *delete* data from a table. This trigger lets you capture information at four timing points: (a) before the firing statement; (b) before each row change from the firing statement; (c) after each row change from the firing statement; and (d) after the firing statement. You can use these types of triggers to audit, check, save, and replace values before they are changed when you need to take action at both the statement and row event levels.

- **Instead-of triggers** These triggers enable you to stop performance of a DML statement and redirect the DML statement. `INSTEAD OF` triggers are often used to manage how you write to non-updatable views. The `INSTEAD OF` triggers apply business rules and directly *insert, update,* or *delete* rows in tables that define updatable views. Alternatively, the `INSTEAD OF` triggers *insert, update,* or *delete* rows in designated tables unrelated to the view.

- **System or database event triggers** These triggers fire when a system activity occurs in the database, like the logon and logoff event triggers. They are useful for auditing information of system access. These triggers let you track system events and map them to users.

Triggers have some restrictions that are important to note. The largest one is that the trigger body can be no longer than 32,760 bytes. That's because trigger bodies are stored in `LONG` datatypes columns. This means you should consider keeping your trigger bodies small. You do that by placing the coding logic in other schema-level components, like functions, procedures, and packages. Another advantage of moving the coding logic out of the trigger body is that you can't wrap it when it's in trigger bodies, as explained in Appendix F.

Each of these triggers has a set of rules that govern its use. You will cover all five triggers in their respective sections. The next section describes the architecture of database triggers.

Privileges Required to Use Triggers

You must have the `CREATE TRIGGER` system privilege to create a trigger on an object that you own. If the object is owned by another user, you'll need that user to grant you the `ALTER` privilege on the object. Alternatively, the privileged user can grant you the `ALTER ANY TABLE` and `CREATE ANY TRIGGER` privileges.

You have definer permissions on your own schema-level components, but you must have `EXECUTE` permission when you call a schema-level component owned by another user. You should document any required privileges during development to streamline subsequent implementation.

Database Trigger Architecture

Database triggers are defined in the database like packages. They're composed of two pieces: the database trigger declaration and the body. The declaration states how and when a trigger is called. You can't call a trigger directly. They are triggered *(called)* by a firing event. Firing events are DDL or DML statements or database or system events. Database triggers implement an object-oriented observer pattern, which means they listen for an event and then take action.

Trigger declarations consist of four parts: a trigger name, a statement, a restriction, and an action. The first three define the trigger declaration, and the last defines the trigger body. A trigger name must be unique among triggers but can duplicate the name of any other object in a schema because triggers have their own namespace. A trigger statement identifies the event or statement type that fires the trigger. A trigger restriction, such as a `WHEN` clause or `INSTEAD OF` clause, limits when the trigger runs. A trigger action is a trigger body.

NOTE
A namespace is a unique list of identifiers maintained in the database catalog.

A database trigger declaration is valid unless you remove the object that it observes. A database trigger declaration also creates a run-time process when an event fires it. The trigger body is not as simple. A trigger body can depend on other tables, views, or stored programs. This means that you can invalidate a trigger body by removing a dependency. Dependencies are local schema objects, but those include synonyms that may resolve across the network. You invalidate a trigger when the trigger body becomes invalid. Trigger bodies are specialized anonymous-block programs. You can call and pass them parameters only through the trigger.

The linkage becomes acute when you define a DDL trigger on the create event. As discussed in the section "Data Definition Language Triggers," an invalid trigger body for a `CREATE` trigger disables your ability to recreate the missing dependency. Similar behaviors occur for other DDL events, like `ALTER` and `DROP`.

You can recompile triggers after you replace any missing dependencies. The syntax is:

```
ALTER TRIGGER trigger_name COMPILE;
```

Triggering events communicate directly with the trigger. You have no control over or visibility into how that communication occurs. You have no data other than that which is available through the system-defined event attributes (see the section "Event Attribute Functions" later in this chapter for more information on DDL, statement-level DML, and system and database event triggers). You do have access to the `new` and `old` pseudo-record types in row-level DML and `INSTEAD OF` triggers. The structure of these types is dynamic and defined at run time. The trigger declaration inherits the declaration of these values from the DML statement that fires it.

DML row-level and `INSTEAD OF` triggers call their trigger bodies differently than statement-level triggers. When an event fires this type of trigger, the trigger declaration spawns a run-time program unit. The run-time program unit is the real "trigger" in this process. The trigger makes available `new` and `old` pseudo-record structures by communicating with the DML statement that fired it. The trigger code block can access these pseudo-record structures by calling them as bind variables. The trigger code block is an anonymous PL/SQL block that is only accessible through a trigger declaration.

As discussed in Chapter 3, Table 3-1, bind variables allow you to reach outside of a program's scope. You can access variables defined in the calling program's scope. The `:in` and `:out` variables are bind variables inside trigger bodies. They let the trigger code block communicate with the trigger session. Only row-level triggers can reference these pseudo-record structure bind variables. Row-level trigger code blocks can read and write through these bind variables, as shown in Figure 10-1.

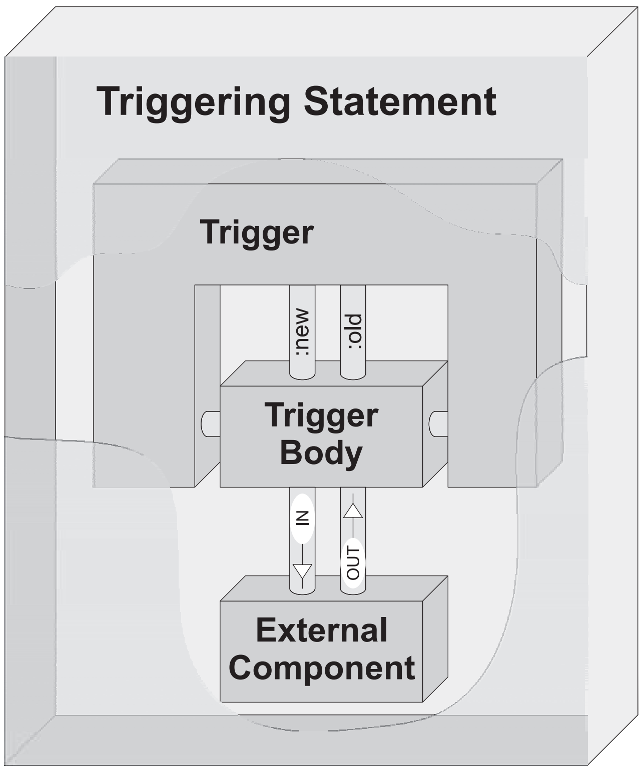


FIGURE 10-1 *Trigger architecture*

You can also call external standalone or package functions and procedures from trigger bodies. When you call programs from the trigger body, the called programs are *black boxes.* This means that external stored programs can't access the `:new` and `:old` bind variables. You do have the option to pass them by value or reference to other stored functions and procedures.

Oracle 11*g* introduces compound triggers. This new trigger changes the landscape of writing triggers. You can now fire a compound trigger, capture row-level statement information, accumulate it in a global trigger collection, and access that data in the `AFTER STATEMENT` timing block. You can read this in detail in the section "Compound Triggers."

You can define multiple triggers on any object or event. Oracle 11*g* provides you with no way to synchronize which trigger fires first, second, or last. This limit is because triggers are interleaved, that is, because program units work independently as discrete processes. Triggers can slow down your application interface, especially row-level statements. You should be careful when and where you deploy triggers to solve problems.

Data Definition Language Triggers

Data Definition Language triggers fire when you create, change, or remove objects in a database schema. They are useful to control or monitor DDL statements. Table 10-1 lists the data definition events that work with DDL triggers. These triggers support both `BEFORE` and `AFTER` event triggers and work at the database or schema level.

You often use DDL triggers to monitor significant events in the database. Sometimes you use them to monitor errant code. Errant code can perform activities that corrupt or destabilize your database. More often, you use these in development, test, and stage systems to understand and monitor the dynamics of database activities.

NOTE
A stage system is used for end-user testing and load balancing metrics before deployment to production.

DDL triggers are very useful when you patch your application code. They can let you find potential changes between releases. You can also use the instead-of create trigger during an upgrade to enforce table creation storage clauses or partitioning rules.

CAUTION
The overhead of these types of triggers should be monitored carefully in production systems.

These triggers can also track the creation and modification of tables by application programs that lead to database fragmentation. They are also effective security tools when you monitor `GRANT` and `REVOKE` privilege statements. The following sections list and describe in detail the event attribute functions you can use to supplement your DDL trigger.

DDL Event	Description
`ALTER`	You `ALTER` objects by changing something about them, like their constraints, names, storage clauses, or structure.
`ANALYZE`	You `ANALYZE` objects to compute statistics for the cost optimizer.
`ASSOCIATE STATISTICS`	You `ASSOCIATE STATISTICS` to link a statistic type to a column, function, package, type, domain index, or index type.
`AUDIT`	You `AUDIT` to enable auditing on an object or system.
`COMMENT`	You `COMMENT` to document column or table purposes.
`CREATE`	You `CREATE` objects in the database, like objects, privileges, roles, tables, users, and views.
`DDL`	You use the `DDL` event to represent any of the primary data definition events. It effectively says any DDL event acting on anything.
`DISASSOCIATE STATISTICS`	You `DISASSOCIATE STATISTICS` to unlink a statistic type from a column, function, package, type, domain index, or index type.
`DROP`	You `DROP` objects in the database, like objects, privileges, roles, tables, users, and views.
`GRANT`	You `GRANT` privileges or roles to users in the database. The privileges enable a user to act on objects, like objects, privileges, roles, tables, users, and views.
`NOAUDIT`	You `NOAUDIT` to disable auditing on an object or system.
`RENAME`	You `RENAME` objects in the database, like columns, constraints, objects, privileges, roles, synonyms, tables, users, and views.
`REVOKE`	You `REVOKE` privileges or roles from users in the database. The privileges enable a user to act on objects, like objects, privileges, roles, tables, users, and views.
`TRUNCATE`	You `TRUNCATE` tables, which drops all rows from a table and resets the high-water mark to the original storage clause initial extent value. Unlike the DML `DELETE` statement, the `TRUNCATE` command can't be reversed by a `ROLLBACK` command. You can use the new flashback to undo the change.

TABLE 10-1 *Available Data Definition Events*

Event Attribute Functions

The following is a list of system-defined event attribute functions:

- `ORA_CLIENT_IP_ADDRESS`
- `ORA_DATABASE_NAME`

- ORA_DES_ENCRYPTED_PASSWORD
- ORA_DICT_OBJ_NAME
- ORA_DICT_OBJ_NAME_LIST
- ORA_DICT_OBJ_OWNER
- ORA_DICT_OBJ_OWNER_LIST
- ORA_DICT_OBJ_TYPE
- ORA_GRANTEE
- ORA_INSTANCE_NUM
- ORA_IS_ALTER_COLUMN
- ORA_IS_CREATING_NESTED_TABLE
- ORA_IS_DROP_COLUMN
- ORA_IS_SERVERERROR
- ORA_LOGIN_USER
- ORA_PARTITION_POS
- ORA_PRIVILEGE_LIST
- ORA_REVOKEE
- ORA_SERVER_ERROR
- ORA_SERVER_ERROR_DEPTH
- ORA_SERVER_ERROR_MSG
- ORA_SERVER_ERROR_NUM_PARAMS
- ORA_SERVER_ERROR_PARAM
- ORA_SQL_TXT
- ORA_SYSEVENT
- ORA_WITH_GRANT_OPTION
- SPACE_ERROR_INFO

ORA_CLIENT_IP_ADDRESS

The ORA_CLIENT_IP_ADDRESS function takes no formal parameters. It returns the client IP address as a VARCHAR2 datatype.

You can use it like this:

```
DECLARE
  ip_address VARCHAR2(11);
BEGIN
  IF ora_sysevent = 'LOGON' THEN
    ip_address := ora_client_ip_address;
  END IF;
END;
```

ORA_DATABASE_NAME

The ORA_DATABASE_NAME function takes no formal parameters. It returns the database name as a VARCHAR2 datatype.

You can use it like this:

```
DECLARE
  database VARCHAR2(50);
BEGIN
  database := ora_database_name;
END;
```

ORA_DES_ENCRYPTED_PASSWORD

The ORA_DES_ENCRYPTED_PASSWORD function takes no formal parameters. It returns the DES-encrypted password as a VARCHAR2 datatype. This is equivalent to the value in the SYS.USER$ table PASSWORD column in Oracle 11*g*. Passwords are no longer accessible in the DBA_USERS or ALL_USERS views.

You can use it like this:

```
DECLARE
  password VARCHAR2(60);
BEGIN
  IF ora_dict_obj_type = 'USER' THEN
    password := ora_des_encrypted_password;
  END IF;
END;
```

ORA_DICT_OBJ_NAME

The ORA_DICT_OBJ_NAME function takes no formal parameters. It returns an object name as a VARCHAR2 datatype. The object name represents the target of the DDL statement.

You can use it like this:

```
DECLARE
  database VARCHAR2(50);
BEGIN
  database := ora_obj_name;
END;
```

ORA_DICT_OBJ_NAME_LIST

The ORA_DICT_OBJ_NAME_LIST function takes one formal parameter. The formal parameter is also returned because it is passed by reference as an OUT mode list of VARCHAR2 variables. The formal parameter datatype is defined in the DBMS_STANDARD package as ORA_NAME_LIST_T. The ORA_NAME_LIST_T is a table of VARCHAR2(64) datatypes. The *function returns the number of elements in the list* as a PLS_INTEGER datatype. The name_list contains the list of object names touched by the triggering event.

You can use it like this:

```
DECLARE
  name_list DBMS_STANDARD.ORA_NAME_LIST_T;
  counter    PLS_INTEGER;
```

```
BEGIN
  IF ora_sysevent = 'ASSOCIATE_STATISTICS' THEN
    counter := ora_dict_obj_name_list(name_list);
  END IF;
END;
```

ORA_DICT_OBJ_OWNER

The `ORA_DICT_OBJ_OWNER` function takes no formal parameters. It returns an owner of the object acted upon by the event as a `VARCHAR2` datatype.

You can use it like this:

```
DECLARE
  owner VARCHAR2(30);
BEGIN
  database := ora_dict_obj_owner;
END;
```

ORA_DICT_OBJ_OWNER_LIST

The `ORA_DICT_OBJ_OWNER_LIST` function takes one formal parameter. The formal parameter is also returned because it is passed by reference as an `OUT` mode list of `VARCHAR2` variables. The formal parameter datatype is defined in the `DBMS_STANDARD` package as `ORA_NAME_LIST_T`. The `ORA_NAME_LIST_T` is a table of `VARCHAR2(64)` datatypes. The *function returns the number of elements in the list* indexed by a `PLS_INTEGER` datatype.

In the example, the `owner_list` contains the list of object owners, where their statistics were analyzed by a triggering event. You can use it like this:

```
DECLARE
  owner_list DBMS_STANDARD.ORA_NAME_LIST_T;
  counter    PLS_INTEGER;
BEGIN
  IF ora_sysevent = 'ASSOCIATE_STATISTICS' THEN
    counter := ora_dict_obj_owner_list(owner_list);
  END IF;
END;
```

ORA_DICT_OBJ_TYPE

The `ORA_DICT_OBJ_TYPE` function takes no formal parameters. It returns the datatype of the dictionary object changed by the event as a `VARCHAR2` datatype.

You can use it like this:

```
DECLARE
  type VARCHAR2(19);
BEGIN
  database := ora_dict_obj_type;
END;
```

ORA_GRANTEE

The `ORA_GRANTEE` function takes one formal parameter. The formal parameter is also returned because it is passed by reference as an `OUT` mode list of `VARCHAR2` variables. The formal parameter datatype is defined in the `DBMS_STANDARD` package as `ORA_NAME_LIST_T`. The `ORA_NAME_`

LIST_T is a table of VARCHAR2(64) datatypes. The *function returns the number of elements in the list* indexed by a PLS_INTEGER datatype. The user_list contains the list of users granted privileges or roles by the triggering event.
You can use it like this:

```
DECLARE
  user_list DBMS_STANDARD.ORA_NAME_LIST_T;
  counter   PLS_INTEGER;
BEGIN
  IF ora_sysevent = 'GRANT' THEN
    counter := ora_grantee(user_list);
  END IF;
END;
```

ORA_INSTANCE_NUM

The ORA_INSTANCE_NUM function takes no formal parameters. It returns the current database instance number as a NUMBER datatype.
You can use it like this:

```
DECLARE
  instance NUMBER;
BEGIN
  instance := ora_instance_num;
END;
```

ORA_IS_ALTER_COLUMN

The ORA_IS_ALTER_COLUMN function takes one formal parameter, which is a column name. The function returns a *true* or *false* value as a BOOLEAN datatype. It is true when the column has been altered, and it is false when it hasn't been changed. This function worked with the traditional uppercase catalog information, but in Oracle 11*g* you need to match the catalog case if you opted to save any tables in a case-sensitive format. The example uses a case-insensitive string as an actual parameter.
You can use it like this:

```
DECLARE
  TYPE column_list IS TABLE OF VARCHAR2(32);
  columns COLUMN_LIST := column_list('CREATED_BY','LAST_UPDATED_BY');
BEGIN
  IF ora_sysevent = 'ALTER' AND
     ora_dict_obj_type = 'TABLE' THEN
     FOR i IN 1..columns.COUNT THEN
       IF ora_is_alter_column(columns(i)) THEN
         INSERT INTO logging_table
         VALUES (ora_dict_obj_name||'.'||columns(i)||' changed.');
       END IF;
    END LOOP;
  END IF;
END;
```

This is very useful if you want to guard against changing standard *who-audit* columns, like CREATED_BY, CREATION_DATE, LAST_UPDATED_BY, or LAST_UPDATE_DATE. These are security columns generally used to identify who last touched the data through the standard application programming interface (API). Any change to columns like these can destabilize an API.

ORA_IS_CREATING_NESTED_TABLE

The ORA_IS_CREATING_NESTED_TABLE function takes no formal parameters. It returns a *true* or *false* value as a BOOLEAN datatype when you create a table with a nested table.

You can use it like this:

```
BEGIN
  IF ora_sysevent = 'CREATE' AND
     ora_dict_obj_type = 'TABLE' AND
     ora_is_creating_nested_table THEN
       INSERT INTO logging_table
       VALUES (ora_dict_obj_name||'.'||' created with nested table.');
  END IF;
END;
```

ORA_IS_DROP_COLUMN

The ORA_IS_DROP_COLUMN function takes one formal parameter, which is a column name. The function returns a *true* or *false* value as a BOOLEAN datatype. It is true when the column has been dropped, and it returns false when it hasn't been dropped. This function worked with the traditional uppercase catalog information, but in Oracle 11*g* you need to match the catalog case if you opted to save any tables in a case-sensitive format. The example uses a case-insensitive string as an actual parameter.

You can use it like this:

```
DECLARE
  TYPE column_list IS TABLE OF VARCHAR2(32);
  columns COLUMN_LIST := column_list('CREATED_BY','LAST_UPDATED_BY');
BEGIN
  IF ora_sysevent = 'DROP' AND
     ora_dict_obj_type = 'TABLE' THEN
     FOR i IN 1..columns.COUNT THEN
       IF ora_is_drop_column(columns(i)) THEN
         INSERT INTO logging_table
         VALUES (ora_dict_obj_name||'.'||columns(i)||' changed.');
       END IF;
    END LOOP;
  END IF;
END;
```

This function is very useful if you want to guard against changing standard *who-audit* columns, like those discussed for the ORA_IS_DROP_COLUMN function earlier in this table.

ORA_IS_SERVERERROR

The ORA_IS_SERVERERROR function takes one formal parameter, which is an error number. It returns a *true* or *false* value as a BOOLEAN datatype when the error is on the error stack.

You can use it like this:

```
BEGIN
  IF ora_is_servererror(4082) THEN
       INSERT INTO logging_table
```

```
      VALUES ('ORA-04082 error thrown.');
  END IF;
END;
```

ORA_LOGIN_USER

The `ORA_LOGIN_USER` function takes no formal parameters. The function returns the current schema name as a `VARCHAR2` datatype.

You can use it like this:

```
BEGIN
  INSERT INTO logging_table
  VALUES (ora_login_user||' is the current user.');
END;
```

ORA_PARTITION_POS

The `ORA_PARTITION_POS` function takes no formal parameters. The function returns the numeric position with the SQL text where you can insert a partition clause. This is only available in an `INSTEAD OF CREATE` trigger.

You can use the following, provided you add your own partitioning clause:

```
DECLARE
  sql_text  ORA_NAME_LIST_T;
  sql_stmt  VARCHAR2(32767);
  partition VARCHAR2(32767) := 'partitioning_clause';
BEGIN
  FOR i IN 1..ora_sql_txt(sql_text) LOOP
    sql_stmt := sql_stmt || sql_text(i);
  END LOOP;
  sql_stmt := SUBSTR(sql_text,1,ora_partition_pos - 1)||' '
           || partition||' '||SUBSTR(sql_test,ora_partition_pos);
  -- Add logic to prepend schema because this runs under SYSTEM.
  sql_stmt := REPLACE(UPPER(sql_stmt),'CREATE TABLE '
                     ,'CREATE TABLE '||ora_login_user||'.');
  EXECUTE IMMEDIATE sql_stmt;
END;
```

The coding sample requires that you grant the owner of the trigger the `CREATE ANY TRIGGER` privilege. You should consider a master privileged user for your application, and avoid using the `SYSTEM` schema.

ORA_PRIVILEGE_LIST

The `ORA_PRIVILEGE_LIST` function takes one formal parameter. The formal parameter is also returned because it is passed by reference as an `OUT` mode list of `VARCHAR2` variables. The formal parameter datatype is defined in the `DBMS_STANDARD` package as `ORA_NAME_LIST_T`. The `ORA_NAME_LIST_T` is a table of `VARCHAR2(64)` datatypes. The *function returns the number of elements in the list* indexed by a `PLS_INTEGER` datatype. The `priv_list` contains the list of privileges or roles granted by the triggering event.

You can use it like this:

```
DECLARE
  priv_list DBMS_STANDARD.ORA_NAME_LIST_T;
  counter   PLS_INTEGER;
```

```
BEGIN
  IF ora_sysevent = 'GRANT' OR
     ora_sysevent = 'REVOKE' THEN
    counter := ora_privilege_list(priv_list);
  END IF;
END;
```

ORA_REVOKEE

The `ORA_REVOKEE` function takes one formal parameter. The formal parameter is also returned because it is passed by reference as an `OUT` mode list of `VARCHAR2` variables. The formal parameter datatype is defined in the `DBMS_STANDARD` package as `ORA_NAME_LIST_T`. The `ORA_NAME_LIST_T` is a table of `VARCHAR2(64)` datatypes. The *function returns the number of elements in the list* indexed by a `PLS_INTEGER` datatype. The `priv_list` contains the list of users that had privileges or roles revoked by the triggering event.

You can use it like this:

```
DECLARE
  revokee_list DBMS_STANDARD.ORA_NAME_LIST_T;
  counter      PLS_INTEGER;
BEGIN
  IF ora_sysevent = 'REVOKE' THEN
    counter := ora_revokee(priv_list);
  END IF;
END;
```

ORA_SERVER_ERROR

The `ORA_SERVER_ERROR` function takes one formal parameter, which is the position on the error stack, where 1 is the top of the error stack. It returns an error number as a `NUMBER` datatype.

You can use it like this:

```
DECLARE
  error NUMBER;
BEGIN
  FOR i IN 1..ora_server_error_depth LOOP
    error := ora_server_error(i);
  END LOOP;
END;
```

ORA_SERVER_ERROR_DEPTH

The `ORA_SERVER_ERROR_DEPTH` function takes no formal parameters. The function returns the number of errors on the error stack as a `PLS_INTEGER` datatype. The code samples for the `ORA_SERVER_ERROR` and `ORA_SERVER_ERROR_MSG` functions demonstrate how you can use it.

ORA_SERVER_ERROR_MSG

The `ORA_SERVER_ERROR_MSG` function takes one formal parameter, which is the position on the error stack, where 1 is the top of the error stack. It returns an error message text as a `VARCHAR2` datatype.

You can use it like this:

```
DECLARE
  error VARCHAR2(64);
BEGIN
  FOR i IN 1..ora_server_error_depth LOOP
    error := ora_server_error_msg(i);
  END LOOP;
END;
```

ORA_SERVER_ERROR_NUM_PARAMS

The ORA_SERVER_ERROR_NUM_PARAMS function takes no formal parameters. The function returns the count of any substituted strings from error messages as a PLS_INTEGER datatype. For example, an error format could be "Expected %s, found %s." The code sample for ORA_SERVER_ERROR_PARAM function shows how you can use it.

ORA_SERVER_ERROR_PARAM

The ORA_SERVER_ERROR_PARAM function takes one formal parameter, which is the position in an error message, where 1 is the first occurrence of a string in the error message. It returns an error message text as a VARCHAR2 datatype.

You can use it like this:

```
DECLARE
  param VARCHAR2(32);
BEGIN
  FOR i IN 1..ora_server_error_depth LOOP
    FOR j IN 1..ora_server_error_num_params(i) LOOP
      param := ora_server_error_param(j);
    END LOOP;
  END LOOP;
END;
```

ORA_SQL_TXT

The ORA_SQL_TXT function takes one formal parameter. The formal parameter is also returned because it is passed by reference as an OUT mode list of VARCHAR2 variables. The formal parameter datatype is defined in the DBMS_STANDARD package as ORA_NAME_LIST_T. The ORA_NAME_LIST_T is a table of VARCHAR2(64) datatypes. The function returns the number of elements in the list indexed by a PLS_INTEGER datatype. The list contains the substrings of the processed SQL statement that triggered the event. The coding example is shown with the ORA_PARTITION_POS function.

ORA_SYSEVENT

The ORA_SYSEVENT function takes no formal parameters. The function returns the system event that was responsible for firing the trigger as a VARCHAR2 datatype.

You can use it like this:

```
BEGIN
  INSERT INTO logging_table
  VALUES (ora_sysevent||' fired the trigger.');
END;
```

ORA_WITH_GRANT_OPTION

The `ORA_WITH_GRANT_OPTION` function has no formal parameters. The function returns a *true* or *false* value as a `BOOLEAN` datatype. It returns *true* when privileges are granted with *grant option.*

You can use it like this:

```
BEGIN
  IF ora_with_grant_option THEN
       INSERT INTO logging_table
       VALUES ('ORA-04082 error thrown.');
  END IF;
END;
```

SPACE_ERROR_INFO

The SPACE_ERROR_INFO function uses six formal pass-by-reference parameters. They are all `OUT` mode parameters. The prototype is

```
space_error_info( error_number OUT NUMBER
                , error_type OUT VARCHAR2
                , object_owner OUT VARCHAR2
                , table_space_name OUT VARCHAR2
                , object_name OUT VARCHAR2
                , sub_object_name OUT VARCHAR2)
```

This function returns *true* when the triggering event is related to an *out-of-space* condition, and it fills in all the outbound parameters. You implement this with a logging table that supports at least the six `OUT` parameters. When the function returns *false,* the `OUT` mode variables are null.

You can use it like this:

```
DECLARE
  error_number    NUMBER;
  error_type      VARCHAR2(12);
  object_owner    VARCHAR2(30);
  tablespace_name VARCHAR2(30);
  object_name     VARCHAR2(128);
  subobject_name  VARCHAR2(30);
BEGIN
  IF space_error_info( error_number, error_type
                     , object_owner, tablespace_name
                     , object_name, subobject_name) THEN
       INSERT INTO logging_table
       VALUES ( … implementation_dependent … );
  END IF;
END;
```

Building DDL Triggers

The prototype for building DDL triggers is

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} ddl_event ON {DATABASE | SCHEMA}
[WHEN (logical_expression)]
[DECLARE]
  declaration_statements;
```

```
BEGIN
  execution_statements;
END [trigger_name];
/
```

You can use the `INSTEAD OF` clause only when auditing a creation event. Before triggers make sure the contents of the trigger body occur before the triggering DDL command, while after triggers run last. See the section "ORA_PARTITION_POS" earlier in this chapter for an implementation of an `INSTEAD OF CREATE` trigger that appends a partitioning table.

The DDL example trigger requires that you create the `audit_creations` table and `audit_creations_s1` sequence before the trigger. If you forget to create one or both, you can't create either after you attempt to compile the database trigger. This limitation exists because you have a valid trigger declaration but an invalid trigger body. You must drop or disable the trigger *(declaration)* before you can create anything in the schema.

You should note that the table and trigger share the same name. This is possible because there are two namespaces in Oracle databases, one for triggers and another for everything else.

You create the table and sequence as follows:

```
CREATE TABLE audit_creation
( audit_creation_id NUMBER
, audit_owner_name  VARCHAR2(30) CONSTRAINT audit_creation_nn1 NOT NULL
, audit_obj_name    VARCHAR2(30) CONSTRAINT audit_creation_nn2 NOT NULL
, audit_date        DATE         CONSTRAINT audit_creation_nn3 NOT NULL
, CONSTRAINT audit_creation_p1   PRIMARY KEY (audit_creation_id));

CREATE SEQUENCE audit_creation_s1;
```

Now you can create the `audit_creation` system trigger. This trigger shows you the behavior of a DDL trigger when dependencies become unavailable to the trigger:

```
CREATE OR REPLACE TRIGGER audit_creation
BEFORE CREATE ON SCHEMA
BEGIN
  INSERT INTO audit_creation VALUES
  (audit_creation_s1.nextval,ORA_DICT_OBJ_OWNER,ORA_DICT_OBJ_NAME,SYSDATE);
END audit_creation;
/
```

The following DDL statement triggers the system trigger, which inserts data from the trigger attribute functions. It creates a synonym called `mythology` that doesn't translate to anything real, but it does create an event that fires the trigger.

The DDL statement is

```
CREATE SYNONYM mythology FOR plsql.some_myth;
```

You can query the results of the trigger using the following SQL*Plus formatting and statement:

```
COL audit_creation_id FORMAT 99999999 HEADING "Audit|Creation|ID #"
COL audit_owner_name  FORMAT A6 HEADING "Audit|Owner|Name"
COL audit_obj_name    FORMAT A8 HEADING "Audit|Object|Name"
COL audit_obj_name    FORMAT A9 HEADING "Audit|Object|Name"
SELECT * FROM audit_creation;
```

The query returns

```
Audit Audit  Audit
   Creation Owner  Object    Audit
       ID # Name   Name      Date
  --------- ------ --------- ---------
         21 PLSQL  MYTHOLOGY 17-NOV-08
```

You have now seen how to implement a DDL trigger. The next section examines DML triggers.

Data Manipulation Language Triggers

DML triggers can fire *before* or *after* INSERT, UPDATE, and DELETE statements. DML triggers can be *statement-* or *row-level* activities. *Statement-level* triggers fire and perform a statement or set of statements once no matter how many rows are affected by the DML event. *Row-level* triggers fire and perform a statement or set of statements for *each* row changed by a DML statement.

A *principal caveat* of triggers that manage data changes is that you cannot use SQL Data Control Language (DCL) in them, unless you declare the trigger as autonomous. When triggers run inside the scope of a transaction, they disallow setting a SAVEPOINT or performing either a ROLLBACK or COMMIT statement. Likewise, they can't have a DCL (also known as TCL) statement in the execution path of any function or procedure that they call.

The prototype for building DML triggers is

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER}
{INSERT | UPDATE | UPDATE OF column1 [, column2 [, column(n+1)]] | DELETE}
ON table_name
[FOR EACH ROW]
[WHEN (logical_expression)]
[DECLARE]
  [PRAGMA AUTONOMOUS_TRANSACTION;]
  declaration_statements;
BEGIN
  execution_statements;
END [trigger_name];
/
```

The BEFORE or AFTER clause determines whether the trigger fires *before* or *after* the change is written to your local copy of the data. You can define a BEFORE or AFTER clause on tables but not views. While the prototype shows either an insert, update, update of *(a column),* or delete, you can also use an inclusion, OR, operator between the events. Using one OR between two events creates a single trigger that runs for two events. You can create a trigger that supports all four possible events with multiple inclusion operators.

There are two options for DML triggers. You can declare them as *statement*-level triggers, which are also known as *table*-level triggers, or you can declare them as *row*-level triggers.

You have a FOR EACH ROW clause, a WHEN clause, and new and old pseudo-records in row-level triggers. The FOR EACH ROW clause specifies that the trigger should fire for each row as opposed to once per statement. The WHEN clause acts as a filter specifying when the trigger fires. Unlike when working with other stored program units, you must qualify a DECLARE block when you declare local variables, types, or cursors in a trigger.

Triggers require the DECLARE block in trigger bodies because the declaration of a trigger is separate from the trigger body. Trigger bodies are like anonymous-block PL/SQL programs. They are called by the trigger, and the trigger implicitly manages parameter passing. Trigger bodies don't support substitution variables, like anonymous blocks. They support *bind* variables, but only in the context of row-level triggers. There is no parameter passing to statement-level triggers.

Statement- and *row*-level triggers have different purposes and approaches. The trigger types are covered in the next two subsections.

Statement-Level Triggers

Statement-level triggers are also known as table-level triggers because they're triggered by a change to a table. Statement-level triggers capture and process information when a user inserts, updates, or deletes one or more rows in a table. You can also restrict *(filter)* UPDATE statement triggers by constraining them to fire only when a specific column value changes. You can restrict the trigger by using a UPDATE OF clause. The clause can apply to a *column name* or a *comma-delimited list of column names.*

You can't use a WHEN clause in a statement-level trigger. You also can't reference the new or old pseudo-records without raising an ORA-04082 exception. The exception is a compile-time error, and it tells you that new or old references aren't allowed in table-level triggers.

You can implement statement-level triggers on inserting, updating, or deleting events. Statement-level triggers don't let you collect transaction details. You have access to only the type of event and values returned by event attribute functions. The UPDATE OF clause lets you filter the triggering event to a specific column change.

The *statement*-level example uses an UPDATE OF *column name* event. The trigger depends on your running the create_store.sql script from the publisher's web site. You can find a reference to it in the introduction.

The trigger logs events in the price_type_log table. It must be created before you compile the trigger. The following statement creates the table:

```
-- This is found in create_price_type_trigger.sql on the publisher's web site.
CREATE TABLE price_type_log
( price_id     NUMBER             CONSTRAINT price_type_log_nn1 NOT NULL
, user_id      VARCHAR2(32)       CONSTRAINT price_type_log_nn2 NOT NULL
, action_date  DATE               CONSTRAINT price_type_log_nn3 NOT NULL
, CONSTRAINT   price_type_log_p1 PRIMARY KEY (price_id))
/
```

After creating the table, you can create the trigger. It is possible that the trigger can fail if you've already declared another price_t1 trigger on another table. The REPLACE command only works when the CREATE OR REPLACE TRIGGER command works against the same table. You raise an ORA-04095 exception when a trigger name already exists for another table.

The following trigger works in Oracle 10*g* or 11*g*. Oracle 10*g* doesn't support references to sequence .nextval or .currval pseudo-columns in SQL statements when they're inside a PL/SQL block.

This backward-compatible trigger is not found in the script on the publisher's web site.

```
CREATE OR REPLACE TRIGGER price_t1
AFTER UPDATE OF price_type ON price
DECLARE
  price_id NUMBER;
```

```
BEGIN
  SELECT price_log_s1.nextval INTO price_id FROM dual;
  INSERT INTO price_type_log
  VALUES (price_log_s1.nextval,USER,SYSDATE);
END price_t1;
/
```

Oracle 11*g* *does* support references to sequence `.nextval` or `.currval` pseudo-columns in SQL statements when they're inside PL/SQL blocks. The following is included:

```
-- This is found in create_price_type_trigger.sql on the publisher's web site.
CREATE OR REPLACE TRIGGER price_t1
AFTER UPDATE OF price_type ON price
BEGIN
  -- This statement only works in Oracle 11g forward.
  INSERT INTO price_type_log VALUES (price_log_s1.nextval,USER,SYSDATE);
END price_t1;
/
```

You can trigger this by running the following `UPDATE` statement that changes nothing because it simply reassigns the current value of `price_type` column to itself:

```
UPDATE price p
SET    p.price_type = p.price_type
WHERE  EXISTS (SELECT NULL
               FROM   price q
               WHERE  q.price_id = p.price_id);
```

The following query shows that the trigger fired and wrote audit information to the `price_type_log` table:

```
SELECT * FROM price_type_log;
```

This subsection has shown you how to use statement-level DML triggers. The next section shows you how to write row-level triggers.

Row-Level Triggers

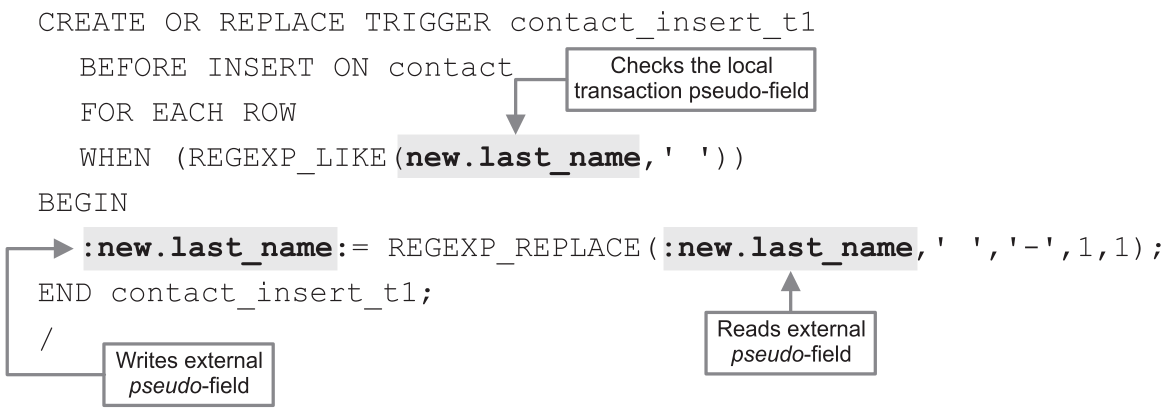
Row-level triggers let you capture new and prior values from each row. This information can let you audit changes, analyze behavior, and recover prior data without performing a database recovery operation.

There are two *pseudo*-records when you use the `FOR EACH ROW` clause in a *row-level* trigger. They both refer to the columns referenced in the DML statement. The pseudo-records are composite variables; `new` or `old` are the pseudo-record variable names in the `WHEN` clause, and `:new` and `:old` are the *bind* variables in the trigger body. They differ because the trigger declaration and body are separate PL/SQL blocks. The `new` and `old` pseudo-records are declared in scope by the row-level trigger declaration. The trigger declaration is the calling block, and the trigger body is the called block. *Bind* variables are passed *by reference* between PL/SQL blocks when an event fires a trigger in a database session. The elements of the pseudo-record are pseudo-fields.

The `new` or `old` pseudo-records are session-level composite variables. They're implicitly declared in the scope of the triggering event, which is the DML statement. Triggers do not have formal signatures like standalone functions or procedures, but they have access to column values changed by DML statements. These column values are the elements of the pseudo-records, or pseudo-fields. Pseudo-field values are those columns inserted by an `INSERT` statement, set by an `UPDATE` statement, or destroyed by a `DELETE` statement.

You access pseudo-fields by referencing the `new` or `old` pseudo-records, a component selector, and a *column name* in the `WHEN` clause. Inside a trigger body, you preface the *pseudo*-records with a colon (`:`). The colon let you reference the externally scoped pseudo-records in the trigger body. The DML statement declares the list of column names *(pseudo-fields).*

The following example demonstrates a trigger that replaces a whitespace in a last name with a dash for hyphenated names.



```
CREATE OR REPLACE TRIGGER contact_insert_t1
  BEFORE INSERT ON contact
  FOR EACH ROW
  WHEN (REGEXP_LIKE(new.last_name,' '))
BEGIN
  :new.last_name:= REGEXP_REPLACE(:new.last_name,' ','-',1,1);
END contact_insert_t1;
/
```

Checks the local transaction pseudo-field
Reads external *pseudo*-field
Writes external *pseudo*-field

The `WHEN` clause checks whether the value of the pseudo-field for the `last_name` column in the `contact` table contains a whitespace. If the condition is met, the trigger passes control to the trigger body. The trigger body has one statement; the `REGEXP_REPLACE` function takes a copy of the pseudo-field as an actual parameter. `REGEXP_REPLACE` changes any whitespace in the string to a dash, and it returns the modified value as a result. The result is assigned to the pseudo-field, and becomes the value in the `INSERT` statement. This is an example of using a DML trigger to enforce a business policy of entering all last names as hyphenated.

The trigger depends on your having run the `create_store.sql` script, as discussed in the introduction. After compiling the trigger in your test schema, you can test the trigger by running the following insert:

```
INSERT INTO contact
VALUES (contact_s1.nextval, 1001, 1003
,       'Zeta Jones','Catherine',NULL
,       3, SYSDATE, 3, SYSDATE);
```

It converts the last name to a hyphenated last name. You query `last_name` from the `contact` table to see the actual inserted value:

```
SELECT last_name FROM contact WHERE last_name LIKE 'Zeta%';
```

You should have the following results:

```
LAST_NAME
--------------------
Zeta-Jones
```

The only problem with the trigger is that a user can simply update the column to remove the dash from the `last_name` column. You can prevent that in a single trigger by using the inclusion `OR` operator, like

```
CREATE OR REPLACE TRIGGER contact_insert_t1
  BEFORE INSERT OR UPDATE OF last_name ON contact
  FOR EACH ROW
  WHEN (REGEXP_LIKE(new.last_name,' '))
BEGIN
  :new.last_name := REGEXP_REPLACE(:new.last_name,' ','-',1,1);
END contact_insert_t1;
/
```

The trigger is now fired on any `INSERT` statement and *only* for `UPDATE` statements that change the `last_name` column. It is always better to build triggers that work with multiple DML events when you take the same type of action.

Another common use for a DML trigger is automatic numbering for a primary key column. As you know, Oracle doesn't support automatic numbering, like Microsoft Access or SQL Server. You create a sequence and a trigger to manage automatic numbering.

While you can create this type of trigger with or without a `WHEN` clause, the `WHEN` clause filters when the trigger should or shouldn't run. A `WHEN` clause let you *insert* a manual primary key value, which can synchronize pseudo-columns `.nextval` and `.currval` for *primary* and *foreign keys* during a single transaction insert.

Rather than build a multiple table example, you will examine automatic numbering from the perspective of logging new connections to and disconnections from the database. The balance of the code for this example is in the section "Data Definition Language Triggers." The DDL triggers that monitor login and logout events call a `user_connection` package that logs to a `connection_log` table. The table definition is

```
CREATE TABLE connection_log
( event_id           NUMBER(10)
, event_user_name    VARCHAR2(30) CONSTRAINT log_event_nn1 NOT NULL
, event_type         VARCHAR2(30) CONSTRAINT log_event_nn2 NOT NULL
, event_date         DATE         CONSTRAINT log_event_nn3 NOT NULL
, CONSTRAINT connection_log_p1    PRIMARY KEY (event_id));
```

The *row-level* trigger `connection_log_t1` demonstrates the proper way to write a pseudo-automatic numbering trigger for Oracle 10*g*:

```
-- This is found in create_signon_trigger.sql on the publisher's web site.
CREATE OR REPLACE TRIGGER connection_log_t1
  BEFORE INSERT ON connection_log
  FOR EACH ROW
  WHEN (new.event_id IS NULL)
BEGIN
  SELECT   connection_log_s1.nextval
  INTO     :new.event_id
  FROM     dual;
END;
/
```

The `connection_log_t1` trigger demonstrates managing a sequence, but it also shows you how to `SELECT INTO` a pseudo-field variable. You should really modify the trigger when deploying it on an Oracle 11*g* database because you no longer have to select a sequence value into a variable from the pseudo-table dual. You can simply assign it directly.

The *row-level* trigger `connection_log_t2` demonstrates the proper way to write a pseudo-automatic numbering trigger for Oracle 11*g*:

```
CREATE OR REPLACE TRIGGER connection_log_t1
  BEFORE INSERT ON connection_log
  FOR EACH ROW
  WHEN (new.event_id IS NULL)
BEGIN
  :new.event_id := connection_log_s1.nextval;
END;
/
```

The `connection_log_t1` and `connection_log_t2` triggers fire only when you fail to provide a primary key value during an `INSERT` statement.

These row-level triggers illustrate two processing rules. One rule is that you can reference a *pseudo*-row column as an *ordinary* variable in the `WHEN` clause because the actual trigger fires in the same memory scope as the DML transaction. The other rule is that you must reference a *pseudo*-row column as a *bind* variable inside the actual trigger scope, where it is running in a different memory space. The *pseudo*-rows `NEW` and `OLD` are *pass-by-reference* structures, and they contain your active DML session variable values when arriving at the trigger body. The `new` and `old` *pseudo*-record variables also receive any changes made in the trigger body when they are returned to your active DML session.

All the `old` pseudo-record columns are null when you execute an `INSERT` statement, and the `new` pseudo-record columns are null when you run a `DELETE` statement. Both `new` and `old` pseudo-records are present during `UPDATE` statements, but only for those columns referenced by the `SET` clause.

This subsection has shown you how to write *row-level* triggers. It demonstrates how to use the new and old pseudo-record in your `WHEN` clause and trigger body.

This section has covered how to use DML triggers and examined both *statement-* and *row-level* trigger implementation. You should be able to use DML triggers by drawing on what you have learned in this section.

Compound Triggers

Compound triggers acts as both statement- and row-level triggers when you *insert, update,* or *delete* data from a table. You can use a compound trigger to capture information at four timing points: (a) before the firing statement; (b) before each row change from the firing statement; (c) after each row change from the firing statement; and (d) after the firing statement. You can use these types of triggers to audit, check, save, and replace values before they are changed when you need to take action at both the statement and row event levels.

Prior to compound triggers, you went to great lengths to mimic this behavior and ran the risk of a memory leak with the failure of an after statement trigger. A compound trigger functions like a multithreaded process. There is a declaration section for the trigger as a whole, and each *timing point section* has its own local declaration section. Timing point sections are subordinate trigger blocks of the compound trigger.

You can use a compound trigger when you want the behavior of both statement-level and row-level triggers. They can be defined on either a table or a view. Compound triggers don't support filtering actions with the `WHEN` clause or the use of the autonomous transaction `PRAGMA`. You can use the `UPDATE OF` *column name* filter as a governing event in updates. Also, the firing order of compound triggers is not guaranteed because they can be interleaved *(mixed between)* with the firing of standalone triggers.

TIP
You can always call out to a stored function or procedure that runs autonomously.

Compound triggers don't support an `EXCEPTION` block, but you can implement `EXCEPTION` blocks in any of the subordinate timing point blocks. The `GOTO` command is restricted to a single timing point block, which means you can't call between timing blocks. You can use the `:new` and `:old` pseudo-records in the row-level statement blocks but nowhere else.

The minimum implementation of a compound trigger requires that you implement at least one timing point block. Only DML statements trigger compound triggers. Also, compound triggers don't fire when (a) the DML statement doesn't change any rows and (b) the trigger hasn't implemented at least a `BEFORE STATEMENT` or `AFTER STATEMENT` block. Compound triggers have significant performance advantages when your DML statements use bulk operations.

The prototype for a compound trigger is

```
CREATE [OR REPLACE] TRIGGER trigger_name
FOR {INSERT | UPDATE | UPDATE OF column1 [, column2 [, column(n+1)]] | DELETE}
ON table_name
COMPOUND TRIGGER
[BEFORE STATEMENT IS
  [declaration_statement;]
 BEGIN
   execution_statement;
 END BEFORE STATEMENT;]
[BEFORE EACH ROW IS
  [declaration_statement;]
 BEGIN
   execution_statement;
 END BEFORE EACH ROW;]
[AFTER EACH ROW IS
  [declaration_statement;]
 BEGIN
   execution_statement;
 END AFTER EACH ROW;]
[AFTER STATEMENT IS
  [declaration_statement;]
 BEGIN
   execution_statement;
 END AFTER STATEMENT;]
END [trigger_name];
/
```

The example rewrites the insert event row-level trigger from the section "Row-Level Triggers" as a compound trigger. The code follows:

```
-- This is found in create_signon_trigger.sql on the publisher's web site.
CREATE OR REPLACE TRIGGER compound_connection_log_t1
  FOR INSERT ON connection_log
  COMPOUND TRIGGER
  BEFORE EACH ROW IS
  BEGIN
    IF :new.event_id IS NULL THEN
      :new.event_id := connection_log_s1.nextval;
    END IF;
  END BEFORE EACH ROW;
END;
/
```

You should note three key elements about compound triggers. You can't filter events in this type of trigger by using a `WHEN` clause. As mentioned, `:new` and `:old` pseudo-records are only available in the `BEFORE EACH ROW` and `AFTER EACH ROW` timing blocks. Variables declared in the global declaration block retain their value through the execution of all timing blocks that you've implemented.

You can collect row-level information in either the `BEFORE EACH ROW` or `AFTER EACH ROW` timing points and transfer that information to a global collection declared in the trigger body. Then, you can perform bulk operations with the collection contents in the `AFTER STATEMENT` timing point. If you don't write the data to another table, you could raise a maximum number of recursive calls error, `ORA-00036`.

The next example demonstrates collecting information in the row-level timing points, transferring it to a global collection, and processing it as a bulk transaction in the `AFTER STATEMENT` timing block. This example depends on your running the `create_store.sql` script, which is described in the introduction. The first step requires creating a log repository, which is done by creating the following table and sequence:

```
-- This is found in create_compound_trigger.sql on the publisher's web site.
CREATE TABLE price_event_log
( price_log_id     NUMBER
, price_id         NUMBER
, created_by       NUMBER
, creation_date    DATE
, last_updated_by  NUMBER
, last_update_date DATE );

CREATE SEQUENCE price_event_log_s1;
```

The trigger populates `created_by` and `last_updated_by` columns as part of the applications "*who-audit*" information. It assumes that you're striping the data, which means you need to set a `CLIENT_INFO` value for the session. The physical `CLIENT_INFO` section is found in the `V$SESSION` view. You can read more on these concepts in the sidebar "Reading and Writing Session Metadata" later in this chapter.

The following sets the `CLIENT_INFO` value to 3, which is a valid `system_user_id` in the `system_user` table:

```
EXEC dbms_application_info.set_client_info('3');
```

The trigger depends on the state of the `CLIENT_INFO` column, but as you might imagine, it can't control it. Therefore, the trigger assigns a –1 when the `CLIENT_INFO` value is missing during its execution.

The following defines the compound trigger with both `BEFORE EACH ROW` and `AFTER STATEMENT` timing blocks:

```
-- This is found in create_compound_trigger on the publisher's web site.
CREATE OR REPLACE TRIGGER compound_price_update_t1
  FOR UPDATE ON price
  COMPOUND TRIGGER
    -- Declare a global record type.
    TYPE price_record IS RECORD
    ( price_log_id     price_event_log.price_log_id%TYPE
    , price_id         price_event_log.price_id%TYPE
    , created_by       price_event_log.created_by%TYPE
    , creation_date    price_event_log.creation_date%TYPE
    , last_updated_by  price_event_log.last_updated_by%TYPE
    , last_update_date price_event_log.last_update_date%TYPE );
    -- Declare a global collection type.
    TYPE price_list IS TABLE OF PRICE_RECORD;
    -- Declare a global collection and initialize it.
    price_updates  PRICE_LIST := price_list();
  BEFORE EACH ROW IS
    -- Declare or define local timing point variables.
    c       NUMBER;
    user_id NUMBER := NVL(TO_NUMBER(SYS_CONTEXT('userenv','client_info')),-1);
  BEGIN
    -- Extend space and assign dynamic index value.
    price_updates.EXTEND;
    c := price_updates.LAST;
    price_updates(c).price_log_id := price_event_log_s1.nextval;
    price_updates(c).price_id := :old.price_id;
    price_updates(c).created_by := user_id;
    price_updates(c).creation_date := SYSDATE;
    price_updates(c).last_updated_by := user_id;
    price_updates(c).last_update_date := SYSDATE;
  END BEFORE EACH ROW;
  AFTER STATEMENT IS
  BEGIN
    -- Bulk insert statement.
    FORALL i IN price_updates.FIRST..price_updates.LAST
      INSERT INTO price_event_log
      VALUES
      ( price_updates(i).price_log_id
      , price_updates(i).price_id
      , price_updates(i).created_by
      , price_updates(i).creation_date
      , price_updates(i).last_updated_by
      , price_updates(i).last_update_date );
  END AFTER STATEMENT;
END;
/
```

The BEFORE EACH ROW timing block collects row-level data and stores it in a global collection, which can then be read from another timing block. The numeric index for the collection is dynamic and leverages the Collection API LAST method. If you'd like to check how that works, please refer to Chapter 7, where it is covered.

The AFTER STATEMENT timing block reads the global collection and performs a bulk insert of the data to the log table. The next time the trigger is fired, the global collection is empty because the compound trigger implementation is serialized.

You can test the trigger by running the following UPDATE statement:

```
UPDATE price
SET    last_updated_by = NVL(TO_NUMBER(SYS_CONTEXT('userenv','client_info')),-1);
```

Then, you can query the price_event_log table:

```
SELECT * FROM price_event_log;
```

This example has shown you how to capture row-level data, save it in a global collection, and reuse it in a statement-level statement.

Reading and Writing Session Metadata

The process of writing to and reading from the session CLIENT_INFO column requires you to use the DBMS_APPLICATION_INFO package. You use the SET_CLIENT_INFO procedure in the DBMS_APPLICATION_INFO package to write data into the 64-character CLIENT_INFO column found in the V$SESSION view. The following anonymous PL/SQL block assumes that the CREATED_BY and LAST_UPDATED_BY columns should be 3:

```
BEGIN
   -- Write value to V$SESSION.CLIENT_INFO column.
   DBMS_APPLICATION_INFO.SET_CLIENT_INFO('3');
END;
/
```

You can now read this value by calling the READ_CLIENT_INFO procedure. You should enable SERVEROUTPUT using SQL*Plus to see the rendered output when you run the following program:

```
DECLARE
  client_info      VARCHAR2(64);
BEGIN
   -- Read value from V$SESSION.CLIENT_INTO column.
   DBMS_APPLICATION_INFO.READ_CLIENT_INFO(client_info);
   DBMS_OUTPUT.PUT_LINE('[ '||client_info||']');
END;
/
```

User-defined session columns let you store unique information related to user credentials from your Access Control List (ACL). You assign a session column value during user authentication. Then, the session CLIENT_INFO column allows you to manage multiple user interactions in a single schema. Authenticated users can access rows from tables when their session CLIENT_INFO column value matches a striping column value in the table.

This section has explained the new Oracle 11*g* compound triggers and shown you how to implement them. They allow you to mix the benefits and operations of statement- and row-level triggers in a single trigger.

Instead-of Triggers

You can use the `INSTEAD OF` trigger to intercept `INSERT`, `UPDATE`, and `DELETE` statements and replace those instructions with alternative procedural code. Non-updatable views generally have `INSTEAD OF` triggers to accept the output and resolve the issues that make the view non-updatable.

The prototype for building an `INSTEAD OF` trigger is

```
CREATE [OR REPLACE] TRIGGER trigger_name
INSTEAD OF {dml_statement }
ON {object_name | database | schema}
FOR EACH ROW
[WHEN (logical_expression)]
[DECLARE]
  declaration_statements;
BEGIN
  execution_statements;
END [trigger_name];
/
```

`INSTEAD OF` triggers are powerful alternatives that resolve how you use complex and non-updatable views. When you know how the `SELECT` statement works, you can write procedural code to update the data not directly accessible through non-updatable views.

You can only deploy an `INSTEAD OF` DML trigger against a view. There is no restriction as to whether the view is updatable or non-updatable, but generally `INSTEAD OF` triggers are built for non-updatable views.

The following view is supported by the data model provided on the publisher's web site. It is also a non-updatable view because of the `DECODE` statement, as shown:

```
-- This is found in create_insteadof_trigger.sql on the publisher's web site.
CREATE OR REPLACE VIEW account_list AS
  SELECT c.member_id
  ,      c.contact_id
  ,      m.account_number
  ,      c.first_name
  ||     DECODE(c.middle_initial,NULL,' ',' '||c.middle_initial||' ')
  ||     c.last_name FULL_NAME
  FROM contact c JOIN member m ON c.member_id = m.member_id;
```

Without an `INSTEAD OF` trigger, a DML statement against this view can raise an `ORA-01776` exception that says you're disallowed from modifying more than one base table through a join. You could also raise an `ORA-01779` exception that tells you you're disallowed to modify a column because it fails to map to a non-key-preserved table.

You can create an `INSTEAD OF` trigger that would allow you to update or delete from this view. However, the view doesn't have enough information to support `INSERT` statements to either base table. Without redefining the view, there is also no programmatic way to fix these shortcomings.

The following is an `INSTEAD OF INSERT` trigger. It raises an exception for any insertion attempt to the non-updatable view.

```
CREATE OR REPLACE TRIGGER account_list_insert
  INSTEAD OF INSERT ON account_list
  FOR EACH ROW
BEGIN
  RAISE_APPLICATION_ERROR(-20000,'Not enough data for insert!');
END;
/
```

After compiling the trigger, an `INSERT` statement run against the view now raises the following exception stack:

```
INSERT INTO account_list
            *
ERROR at line 1:
ORA-20000: Not enough data for insert!
ORA-06512: at "PLSQL.ACCOUNT_LIST_INSERT", line 2
ORA-04088: error during execution of trigger 'PLSQL.ACCOUNT_LIST_INSERT'
```

The question here is, do you want to define three `INSTEAD OF` event triggers or one? Some developers opt for multiple `INSTEAD OF` triggers as opposed to one that does everything. You should consider defining one trigger for inserting, updating, and deleting events. Table 10-2 qualifies the `INSERTING`, `UPDATING`, and `DELETING` functions from the `DBMS_STANDARD` package. These functions let you determine the type of DML event and write one trigger that manages all three DML events.

Certain required fields for an insert to either the member or contact tables are missing from the view. There is also a programmatic way to fix these shortcomings.

Function Name	**Return Datatype**	**Description**
`DELETING`	`BOOLEAN`	The `DELETING` function returns a Boolean true when the DML event is deleting.
`INSERTING`	`BOOLEAN`	The `INSERTING` function returns a Boolean true when the DML is inserting.
`UPDATING`	`BOOLEAN`	The `UPDATING` function returns a Boolean true when the DML is updating.

TABLE 10-2 *Data Manipulation Language Event Functions*

You can build a complete trigger for all DML statements by using the event function from Table 10-2. The following provides an example `INSTEAD OF` trigger:

```
-- This is found in create_insteadof_trigger.sql on the publisher's web site.
CREATE OR REPLACE TRIGGER account_list_dml
  INSTEAD OF INSERT OR UPDATE OR DELETE ON account_list
  FOR EACH ROW
DECLARE
  -- Source variable.
  source account_list.full_name%TYPE := :new.full_name;
  -- Parsed variables.
  fname  VARCHAR2(43);
  mname  VARCHAR2(1);
  lname  VARCHAR2(43);
  -- Check whether all dependents are gone.
  FUNCTION get_dependents (member_id NUMBER) RETURN BOOLEAN IS
    rows NUMBER := 0;
    CURSOR c (member_id_in NUMBER) IS
      SELECT COUNT(*) FROM contact WHERE member_id = member_id_in;
  BEGIN
    OPEN c (member_id);
    FETCH c INTO rows;
    IF rows > 0 THEN
      RETURN FALSE;
    ELSE
      RETURN TRUE;
    END IF;
  END get_dependents;
BEGIN
  IF INSERTING THEN -- On insert event.
    RAISE_APPLICATION_ERROR(-20000,'Not enough data for insert!');
  ELSIF UPDATING THEN -- On update event.
    -- Assign source variable.
    source := :new.full_name;
    -- Parse full_name for elements.
    fname := LTRIM(REGEXP_SUBSTR(source,'(^|^ +)([[:alpha:]]+)',1));
    mname := REGEXP_SUBSTR(
               REGEXP_SUBSTR(
                 source,'( +)([[:alpha:]]+)(( +|. +))',1),'([[:alpha:]])',1);
    lname := REGEXP_SUBSTR(
               REGEXP_SUBSTR(
                 source,'( +)([[:alpha:]]+)( +$|$)',1),'([[:alpha:]]+)',1);
    -- Update name change in base table.
    UPDATE contact
    SET    first_name = fname
    ,      middle_initial = mname
    ,      last_name = lname
    WHERE  contact_id = :old.contact_id;
  ELSIF DELETING THEN -- On delete event.
```

```
    DELETE FROM contact WHERE member_id = :old.member_id;
    -- Only delete the parent when there aren't any more children.
    IF get_dependents(:old.member_id) THEN
      DELETE FROM member WHERE member_id = :old.member_id;
    END IF;
  END IF;
END;
/
```

Some tricks or risks are inherent in this type of trigger. Risks are bad in triggers because they should be foolproof. One potential flaw in *this* trigger is the assignment of the pseudo-field `:new.full_name` in the declaration section. The database doesn't check when you compile the trigger if the size of the `source` variable is large enough to handle possible assignments. This is a critical place to use type anchoring as discussed in Chapter 9.

The `account_list_dml` trigger anchors the source variable to the assigned column value, which ensures you won't raise `ORA-06502`, `ORA-06512`, and `ORA-04088` errors. An assignment in the `DECLARE` block of a trigger body does raise a run-time exception, like standalone anonymous-block programs.

This trigger fires on any DML event against the non-updatable view, and it handles the insert, update, or deletion to the base tables where appropriate. As mentioned, there wouldn't be enough information to perform `INSERT` statements to the base tables. The trigger raises a user-defined exception when someone attempts to insert a new record through the view. There is enough information to *update* the name, but as you can tell, it isn't a trivial bit of work. You should know that the regular expression for the middle name won't work if you have leading whitespace before the first name. The `DELETE` statement only touches one table unless all dependent rows in the contact table have been deleted first, because you never want to leave orphaned rows in a dependent table.

This section has shown you how to write individual-event and multiple-event `INSTEAD OF` triggers. You should try to write all DML events in a single `INSTEAD OF` trigger because they're much easier to maintain.

Non-Updatable Views

Views are non-updatable when they contain any of the following constructs:

- Set operators
- Aggregate functions
- `CASE` or `DECODE` statements
- `CONNECT BY`, `GROUP BY`, `HAVING`, or `START WITH` clauses
- The `DISTINCT` operator
- Joins (with exceptions when they contain the joining key)

You also cannot reference any pseudo-columns or expressions when you update a view.

System or Database Event Triggers

System triggers enable you to audit *server startup* and *shutdown, server errors,* and *user logon* and *logoff* activities. They are convenient for tracking the duration of connections per user and the uptime of the database server.

The prototype for building a database SYSTEM trigger is

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER} database_event ON {database | schema}
[DECLARE]
  declaration_statements;
BEGIN
  execution_statements;
END [trigger_name];
/
```

The *logon* and *logoff* triggers monitor the duration of connections. The DML statements for these triggers are in the user_connection package. Both the connecting_trigger and the disconnecting_trigger call procedures in the user_connection package to insert *logon* and *logoff* information per user.

The connecting_trigger provides an example of a system trigger that monitors users' logons to the database, as shown:

```
-- This is found in create_system_triggers.sql on the publisher's web site.
CREATE OR REPLACE TRIGGER connecting_trigger
  AFTER LOGON ON DATABASE
BEGIN
  user_connection.connecting(sys.login_user);
END;
/
```

The disconnecting_trigger provides an example of a system trigger that monitors users' logoffs from the database, as shown:

```
-- This is found in create_system_triggers.sql on the publisher's web site.
CREATE OR REPLACE TRIGGER disconnecting_trigger
  BEFORE LOGOFF ON DATABASE
BEGIN
  user_connection.disconnecting(sys.login_user);
END;
/
```

Both triggers are compact and call methods of the user_connection package. This package requires the connection_log table, which is

```
-- This is found in create_system_triggers.sql on the publisher's web site.
CREATE TABLE connection_log
( event_id           NUMBER
, event_user_name    VARCHAR2(30) CONSTRAINT log_event_nn1 NOT NULL
, event_type         VARCHAR2(14) CONSTRAINT log_event_nn2 NOT NULL
, event_date         DATE         CONSTRAINT log_event_nn3 NOT NULL
, CONSTRAINT connection_log_p1    PRIMARY KEY (event_id));
```

The package body declares two procedures. One supports the *logon* trigger, and the other supports the *logoff* trigger. The package specification is

```
-- This is found in create_system_triggers.sql on the publisher's web site.
CREATE OR REPLACE PACKAGE user_connection AS
  PROCEDURE connecting (user_name IN VARCHAR2);
  PROCEDURE disconnecting (user_name IN VARCHAR2);
END user_connection;
/
```

The implementation of the `user_connection` package body is

```
-- This is found in create_system_triggers.sql on the publisher's web site.
CREATE OR REPLACE PACKAGE BODY user_connection AS
  PROCEDURE connecting (user_name IN VARCHAR2) IS
  BEGIN
    INSERT INTO connection_log (event_user_name, event_type, event_date)
    VALUES (user_name,'CONNECT',SYSDATE);
  END connecting;
  PROCEDURE disconnecting (user_name IN VARCHAR2) IS
  BEGIN
    INSERT INTO connection_log (event_user_name, event_type, event_date)
    VALUES (user_name,'DISCONNECT',SYSDATE);
  END disconnecting;
END user_connection;
/
```

You may notice that the `connection_log` table has four columns but the `INSERT` statement only uses three. This is possible because the `connection_log_t1` trigger automatically assigns the next value from the `connection_log_s1` sequence. You can find the source of the `connection_log_t1` trigger in the section "Row-Level Triggers" in this chapter.

This section has demonstrated how you can build system triggers.

Trigger Restrictions

There are several restrictions on how you implement triggers in Oracle 11*g*. They are fairly consistent between releases, but Oracle 11*g* has relaxed some mutating table restrictions. Restrictions have been covered in earlier sections when they apply to only one type of trigger.

The following subsections cover the remaining restrictions.

Maximum Trigger Size

A trigger body can be no longer than 32,760 bytes, as noted in the section "Introduction to Triggers" at the beginning of this chapter. This size limitation means that you should consider keeping your trigger bodies small in size. You can accomplish this without losing any utility by moving coding logic into other schema-level components, such as functions, procedures, and packages. An advantage of moving the coding logic out of the trigger body is that you can reuse the code. You can also wrap schema-level objects, whereas you can't wrap trigger bodies. Appendix F discusses wrapping your PL/SQL code from prying eyes.

SQL Statements

Nonsystem trigger bodies can't contain DDL statements. They also can't contain Data Control Language (DCL) or Transaction Control Language (TCL) commands, like `ROLLBACK`, `SAVEPOINT`, or `COMMIT`. This rule holds true for the schema-level components that you call from nonsystem trigger bodies when the trigger runs within the scope of the triggering statement.

If you declare a trigger as autonomous, nonsystem trigger bodies can contain Data Control Language commands because they don't alter the transaction scope. They act outside of it. You enable a trigger to work outside the scope of a triggering statement by putting the following in its `DECLARE` block:

```
PRAGMA AUTONOMOUS_TRANSACTION;
```

A larger problem with SQL statements exists with remote transactions. If you call a remote schema-level function or procedure from a trigger body, it is possible that you may encounter a timestamp or signature mismatch. A mismatch invalidates the trigger and causes the triggering SQL statement to fail.

LONG and LONG RAW Datatypes

The `LONG` and `LONG RAW` datatypes are legacy components. No effort is spent on updating them, and you should migrate to LOBs at your earliest opportunity.

You can't declare a local variable in a trigger with the `LONG` or `LONG RAW` datatype. However, you can insert into a `LONG` or `LONG RAW` column when the value can be converted to a constrained datatype, like a `CHAR` or `VARCHAR2`. The maximum length is 32,000 bytes.

Row-level triggers cannot use a *:new* or *:old* pseudo-record, or row of data, when the column is declared as a `LONG` or `LONG RAW` datatype.

Mutating Tables

A *mutating* table is one undergoing change. Change can come from an `INSERT`, `UPDATE`, or `DELETE` statement, or from a `DELETE CASCADE` constraint.

This type of error can only happen on row-level triggers.

You can't query or modify tables when they're changing. This makes sense if you think about it. If a trigger fires because of a change on a table, it can't see the change until it is final. While you can access the `new` and `old` pseudo-records, you can't read the state of the table. Any attempt to do so raises an `ORA-04091` exception.

The following demonstrates how mutating errors can occur. You create a `mutant` table, as follows:

```
CREATE TABLE mutant
( mutant_id     NUMBER
, mutant_name   VARCHAR2(20));
```

You can then insert the four primary ninja turtles:

```
INSERT INTO mutant VALUES (mutant_s1.nextval,'Donatello');
INSERT INTO mutant VALUES (mutant_s1.nextval,'Leonardo');
INSERT INTO mutant VALUES (mutant_s1.nextval,'Michelangelo');
INSERT INTO mutant VALUES (mutant_s1.nextval,'Raphael');
```

After inserting the data, you can build the following trigger:

```
CREATE OR REPLACE TRIGGER mutator
AFTER DELETE ON mutant
FOR EACH ROW
```

```
DECLARE
  rows NUMBER;
BEGIN
  SELECT COUNT(*) INTO rows FROM mutant;
  dbms_output.put_line('[rows] has '||rows||']');
END;
/
```

The trigger body attempts to get the number of rows but it can't find the number of rows because the record set is not final. This restriction exists to prevent the trigger from seeing inconsistent data.

You can fire the trigger by running the following command to delete Michelangelo from the mutant table. The `DELETE` statement is

```
DELETE FROM MUTANT WHERE mutant_name = 'Michelangelo';
```

After running that statement, the `DELETE` statement raises the following error stack:

```
DELETE FROM mutant WHERE mutant_name = 'Michelangelo'

ERROR at line 1:
ORA-04091: table PLSQL.MUTANT is mutating, trigger/function may not see it
ORA-06512: at "PLSQL.MUTATOR", line 4
ORA-04088: error during execution of trigger 'PLSQL.MUTATOR'
```

A trigger rolls back the trigger body instructions and triggering statement when it encounters a mutating table. You should be careful to avoid mutating table errors now that you understand why they can occur.

System Triggers

System triggers can present interesting problems. Most problems relate to limitations or constraints imposed by event attribute functions. Some of the event attribute functions may be undefined for certain DDL events. You should refer back to the section "Event Attribute Functions" earlier in this chapter to understand exactly what to expect from event attribute functions.

Event attribute functions are declared and implemented in the Oracle `STANDARD` package. You can also encounter a problem creating objects after a system trigger fails to compile. This occur for a `CREATE` event trigger when a `CREATE` event fires the trigger and the trigger body is invalid due to a missing object dependency. The missing dependency invalidates the trigger and marks it as invalid. When you try to create the missing object, the `CREATE` event trigger raises an `ORA-04098` error and disallows the DDL statement. You must drop the invalid trigger, fix the object dependency, and recompile the trigger to proceed.

You can use the `audit_creation` trigger created in the section "Data Definition Language Triggers" to illustrate this restriction. If you drop the `audit_creation` table, the `audit_creation` trigger becomes invalid. Subsequently, you raise an `ORA-04098` while attempting to create this missing table. You can't proceed until you drop the trigger, or you disable it. You disable the trigger by running the following command:

```
ALTER TRIGGER audit_creations DISABLE;
```

You can now create the table, and the trigger should re-validate when it is called. If the trigger is still invalid, you can compile it with this syntax:

```
ALTER TRIGGER audit_new_stuff COMPILE;
```

This section has covered some trigger restrictions. You should check the individual sections for restrictions that are specific to certain trigger types.

Summary

This chapter has reviewed the five types of database triggers. It has explained triggers and their architecture.