Oracle Database Packages, Procedures, Functions, and Triggers; When and How to Use them.

Frank Ress
Argonne National Laboratory


INTRODUCTION

With Oracle7, application developers and database administrators have been given the means to store and reference procedural logic within the database itself. Used appropriately, these tools allow database developers to significantly enhance the capabilities of Oracle applications and operations.

This paper will examine the structure and use of these tools, and offer some guidelines to assist users in applying the tools to typical development problems. First, the advantages and disadvantages of implementing application logic within the database will be discussed, as opposed to the traditional approach that codes application logic within end-user tools. Next, the specific syntax and construction of these database objects will be examined. Some suggestions will be offered how to determine which tool, or combination of tools, to use in a given situation, and some tips on error handling and debugging will be presented.

WHY STORE LOGIC IN THE DATABASE?

There are a number of advantages to implementing application logic within the database itself, as well as some limitations to this approach. Understanding the advantages and disadvantages will allow the developer to decide whether this approach is appropriate in a given situation.

Implementing application logic within the database offers at least the following advantages:

- **Access to Data:** Since the logic is stored within the database, it can access data without the need for external systems.
- **Centralization:** Logic is centralized, making it easier to manage and maintain.
- **Security:** Security can be more tightly controlled, as the logic is contained within the database.
- **Performance:** Logic executed within the database can often be more efficient than logic executed outside the database.
- **Flexibility:** Logic can be easily modified and updated within the database.

Additionally, there are some limitations to consider:

- **Complexity:** Logic can become complex and difficult to maintain.
- **Performance overhead:** There can be a performance overhead associated with executing logic within the database.
- **Security concerns:** There are security concerns regarding access to and modification of database logic.

Understanding these advantages and limitations is crucial for making informed decisions about when and how to use database logic within the Oracle7 environment.
Guarantee procedural logic is applied to all transactions

Perhaps the most obvious advantage to the implementation of application logic in the database kernel is consistency. All access to the data will be subject to the business rules defined by the database trigger/function/procedure/package. It is unnecessary to duplicate validation or other logic in each application that accesses the data (or deal with the situation when some piece of logic should have been added to an application and wasn't). A complex business rule, integrity constraint, or other algorithm can be coded once, thoroughly tested, and applied universally to all transactions.

Centralized logic maintenance and administration

A logical consequence of this centralization is that ongoing maintenance and administration of these algorithms can be accomplished much more easily than when that logic is implemented in many places at the user (or client) end. Version control is also simplified if there are fewer instances of a particular script to update. There may still be cases where application logic will have to be distributed to several client-side tools, but these new database constructs offer an opportunity, through centralization of the code, to limit the number of copies to those actually required.

Enhanced database integrity and security features

These tools also provide new ways to implement database security and record activity. Using triggers, it is possible to allow end users the ability to perform DML operations on tables to which they have no explicit access privileges. A trigger operates with the privileges of the owner, and any user with execute rights on the trigger receives the same access privileges by proxy. These tools may also be used to enhance and extend the built-in Oracle7 data integrity features. For example, foreign key constraints cannot reference objects across database links, but triggers can be used to implement similar functionality.
Greater flexibility in creation of activity logs

These database tools also provide the means to create customized transaction logs. Since the global variable USER correctly identifies the session owner, rather than the trigger owner, it's possible to create detailed activity logs, to complement built-in auditing features. In many cases an application would benefit from some type of activity logging, but for performance or other reasons database auditing features would not be an acceptable solution.

There is one serious drawback to this technique, however. Database triggers must either run to completion with the initiating transaction or they must rollback with the transaction. Assuming such user-defined logs are written to a database table, this makes error logging nearly impossible. If the initiating transaction fails and rolls back, then the log entry also fails.

Improved database and network performance

Performance can be significantly enhanced in many cases by the creation of integrated routines that can be executed entirely within the database, rather than as a series of operations driven from an application. In addition, applications are becoming more distributed, experiencing a far wider variation in communication speeds between the end user and database engine than in traditional, centralized environments. Application developers must be aware of this issue and design applications to minimize traffic across the slowest-speed links.

In situations where a relatively high-speed LAN connects the client and the database, multiple calls to a database to do such things as field validation may be appropriate, to provide the operator with immediate feedback. When the cost (i.e. latency) of a database call is relatively low, it may be reasonable to include additional calls rather than defer application logic until the transaction is committed. On the other hand, communication with remote users over links like cellular phone services may be so slow that the goal is to bundle as much work as possible in
each database call. Database triggers and stored program logic are a powerful tool that can be used to manage this traffic to provide optimal performance without sacrificing functionality.

SYNTAX, STRUCTURE, AND FEATURES OF DATABASE STORED OBJECTS.

Database triggers, functions, procedures, and packages are new features of the procedural option of the Oracle7 database. In Oracle7, the procedural option is included as a standard part of the base RDBMS license. The CATPROC.SQL script must be executed by the SYS user to create the database structures needed to use the procedural option, and before any packages, procedures, functions, or database triggers can be created.

Database Functions, Procedures, and Packages

Functions, procedures, and packages have two parts, a specification and a body. In the case of stored functions and procedures both parts are defined using a single SQL command. For packages, each part is defined with a separate SQL statement. Examples of these commands are shown below.

CREATE FUNCTION function-name
    (var1 IN VARCHAR2, var2 IN NUMBER) RETURN VARCHAR2
    AS pl/sql body
END;

CREATE PROCEDURE procedure-name
    (var1 IN VARCHAR2, var2 IN OUT NUMBER, var3 OUT NUMBER)
    AS pl/sql body
END;

CREATE PACKAGE package-name AS
    public-var-1 NUMBER; -- A persistent variable; can be referenced outside package
    FUNCTION public-function-1
        (var1 IN VARCHAR2)
        RETURN VARCHAR2;
    PROCEDURE public-proc-1
        (var1 IN OUT NUMBER);
END package-name;

CREATE PACKAGE BODY package_name AS
    public_var_1 VARCHAR2(5); -- A persistent variable; can be referenced inside package
    FUNCTION public_function_1
        (var1 IN VARCHAR2)
RETURN VARCHAR2 IS
  function_var_1 DATE -- A persistent variable; can be referenced inside function
  pl/sql body
PROCEDURE public_proc_1
  (var1 IN OUT NUMBER) AS
  pl/sql body
PROCEDURE private_proc_1
  (var1 IN OUT NUMBER) AS
  pl/sql body
BEGIN
  package_var_1 := USER;
END package_name;
END package_name;

Pl/sql body refers to standard PL/SQL syntax; variables and cursors may be defined, exceptions may be raised, etc. Note that any variables defined in the standalone procedures and functions are active only for the duration of statement execution; after execution the values cannot be referenced from other constructs or in subsequent executions of the function/procedure.

Package variables, on the other hand, are persistent for the duration of the session. Public_var_1, since it is defined in the package header, is available to other database objects referenced in the same session. Package_var_1 and function_var_1 are also persistent, but cannot be referenced outside the package or function, respectively. Also notice that package_var_1 is initialized at the end of the package body. This is a 'one-time-only' section of the statement, executed the first time the package is invoked in a user session. These rules also apply to constants and cursors.

Finally, note that only functions and procedures that are referenced in the package header may be executed by privileged users of the package. Functions and procedures that are defined only in the package body are private to the package.

Database Triggers
The structure of database triggers is much different from that of functions, procedures, and packages, reflecting the difference in the way they operate. Stored functions, procedures, and
packages must be invoked procedurally. In other words, they have to be explicitly called.

Database triggers are event-driven, and are run whenever the triggering DML operation is performed. This behavior accounts for the difference in statement structure and syntax.

There are at least two, and possibly three, parts to each trigger: a triggering statement, an action, and possibly a trigger restriction. A maximum of 12 triggers may be created for each table, and these may be grouped into 4 basic trigger types - BEFORE and AFTER STATEMENT, and BEFORE and AFTER ROW triggers. Therefore, the 12 possible triggers for a table are:

- BEFORE INSERT statement
- BEFORE UPDATE statement
- BEFORE DELETE statement
- AFTER INSERT statement
- AFTER UPDATE statement
- AFTER DELETE statement
- BEFORE INSERT row
- BEFORE UPDATE row
- BEFORE DELETE row
- AFTER INSERT row
- AFTER UPDATE row
- AFTER DELETE row

The basic SQL command used to create a database trigger looks like the following:

```
CREATE TRIGGER trigger_name
  BEFORE INSERT OR DELETE OR UPDATE OF
    col_1, col_2 ON table_name
  FOR EACH ROW
  WHEN condition
  pl/sql body
END;
```

A number of variations to this statement are possible. First, all three types of DML operations (INSERT, UPDATE, and DELETE) can be included in a single CREATE TRIGGER, as long as the trigger conforms to one of the 4 basic trigger types described above. Therefore a table with all 12 types of triggers defined may have as few as 4 associated CREATE TRIGGER statements.

If a single trigger action is desired regardless of the type of DML that caused the trigger to fire, the body of the trigger can have a standard PL/SQL structure. If different actions are needed depending on the type of DML, then the conditional predicates INSERTING, UPDATING, DELETING, can be referenced (as in 'IF UPDATING THEN...') to select different trigger actions.
In addition, the UPDATE triggering event can be limited to a subset of the columns in the table. If changes to the table include one or more of the columns referenced in the trigger, the trigger fires. Otherwise, it does not. Some problems with this feature will be discussed in detail later.

Notice that the FOR EACH ROW and WHEN clauses are optional. Omission of the FOR EACH ROW clause defines a statement trigger that fires once per DML event. Inclusion of the WHEN clause will prevent execution of the trigger body whenever the condition fails to be met.

Since a trigger is only active when a COMMIT is in progress, one very useful feature is to be able to use old and new column values in conditional logic. This is possible through the use of :old and :new correlation names that can prefix the column name to reference the desired values (as in 'IF :new.col_1 = :old.col_2...'). These correlation names are only available in ROW triggers. The :old prefix is undefined for INSERT and the :new is undefined for DELETE.

The last major difference between database triggers and other database stored objects is that triggers are compiled at execution time, rather than when the statement is defined. This can raise performance issues, since run time compilation may be prohibitive for more complex logic.

Finally, be aware that the CREATE statement for any trigger, function, procedure, package header, or package body may also include an OR REPLACE option (i.e. CREATE OR REPLACE TRIGGER ..., etc.) that will allow redefinition of the object in question. Use of this option is recommended in most circumstances, since the REPLACE option has the advantage of preserving any grants to the object. The alternative, to DROP the object and CREATE a new version, should only be used to deliberately revoke existing grants before recreating the object.

GUIDELINES FOR USING A PARTICULAR TYPE OF STORED OBJECT.
Use objects that are stored in compiled form (database function, procedure, package, or some combination) if the trigger is more than about 60 lines (according to Oracle).

Because triggers are not compiled until they are invoked, noticeable delays can be experienced by the user if the trigger is large. To avoid this problem, move most of the trigger logic to objects that are stored in compiled format to minimize the size of the trigger itself. It may be convenient to create large and complex triggers during development, but recode for performance before putting the code into production.

Use packages whenever persistent variables are needed to pass information between objects or to conserve the values of variables from one execution to the next.

Use of package variables is the only technique that provides variable persistence and scope to support passing information between stored objects and/or over time. Although other techniques may exist, using package variables to satisfy this type of requirement will be the best alternative in most cases.

Use packages to group related modules.

Grouping related modules in a package offers the added advantage that the modules will all be compiled together and stored as a unit. Therefore, all the related modules will be loaded into memory on the first call of the procedure. This will serve to reduce overall disk I/O and ensure that when subsequent modules are referenced, they will already have been loaded into memory.

Avoid duplication of standard database features and utilities.

Avoid the temptation to duplicate features, such as constraints, that are already provided in the database. It's very easy to find yourself creating a very specific routine that is actually a special case of a feature provided by the database in some way. Be alert to the possibility this may occur and attempt to use standard database features if possible. Don't reinvent the wheel.
Don't attempt to use database triggers to log failed transactions to a table.

As noted, logging activity in a transaction table is a useful feature made possible by database triggers. The only drawback is that when the transaction fails, all trigger actions are rolled back too, including any log table entries. Workarounds may exist, but there is no straightforward solution. If error logging is needed, database journalling features should be considered.

ERROR HANDLING AND DEBUGGING.

Implementing user-defined error codes

Oracle has reserved error numbers in the range of 20000 to 20999 for user-defined errors. Use PL/SQL exception handlers and the RAISE_APPLICATION_ERROR procedure to display error codes and associated messages. If a package detects more than a single type of error, consider coding a procedure to handle all error processing.

Phantom updates: the problem with using column qualifiers.

One obvious limitation of the 'ON UPDATE OF col_1...' option is that you may have more than one action for different sets of columns in the table. If that's the case, you have no option but to include the union of each set of columns in the list and use conditional logic in the trigger to determine what's being updated, and therefore what action to take.

However, even when there's only a single set of columns that fires one action for the trigger, compare the :old and :new values of the columns in question to determine whether an update has occurred. Why? Because the trigger DOESN'T. If the column is included in the UPDATE of the triggering statement, it doesn't matter whether the values change, the trigger just fires.

How likely it is that an UPDATE statement would perform the useless work of copying the same value back to some field of a record? If you use SQL*Forms, the answer is 'Almost always'.
Suppose a personnel form references a base table containing a person's height, weight, and address. The AFTER UPDATE ROW trigger on the address fields records changes in a log table. Unfortunately, the UPDATE generated by Forms includes all base table columns. Any time a record is updated the trigger will fire to record an address change.

Use the SQL*Plus SHOW ERROR command to display errors generated during compilation of packages, procedures, and functions.

Compilation errors for stored packages, procedures, and functions are not immediately displayed in SQL*Plus. Enter the SHOW ERROR command to display detailed error messages.

Use the DBMS_OUTPUT package to generate debugging output.

Use the DBMS_OUTPUT:PUT and :PUT_LINE procedures to generate debugging output in SQL*Plus or SQL*DBA. You will need to issue the SET SERVEROUTPUT ON command before executing the stored object to direct the output from these procedures to your display. Such debug statements can be left at strategic locations in production code to assist in error diagnosis when these routines are in production. Use the analogous routines :GET and :GET_LINE to provide debug information in SQL*Forms.

Avoid SELECTs on the trigger base table.

A trigger cannot query its own base table. This is known as the 'mutating table', because there is a transaction pending on the base table. The trigger's view of the table is not consistent, since it will depend on whether the transaction is ultimately committed or rolled back.

Avoid trigger loops.

A trigger cannot perform a DML operation on its own base table, but can perform DML on other tables. If the DML satisfies the triggering condition on the second table, that trigger will also fire.
It's possible that such a cascade could loop back to attempt DML on a table earlier in the chain.

Needless to say, this won't work. Plan your triggers carefully.