CHAPTER 24

PL/SQL Architecture

In my experience, relatively few PL/SQL developers exhibit a burning curiosity about the underlying architecture of the PL/SQL language. As a community, we seem to mostly be content with learning the basic syntax of PL/SQL, writing our programs, and going home to spend quality time with family and friends. That's a very healthy perspective!

I suggest, however, that all PL/SQL developers would benefit from a basic understanding of this architecture. Not only will you be able to write programs that utilize memory more efficiently, but you will be able to configure your programs and overall applications to behave in ways that you might have thought were impossible.

You will find in this chapter answers to the following questions, and many more:

- How does the PL/SQL runtime engine use memory, and what can I do to manage how much memory is used?
- Should I be running natively compiled code or stick with the default interpreted code? What does *native compilation* mean, anyway?
- Why do my programs become INVALID, and how can I recompile them back to health?
- I have duplicated tables in 20 different schemas. Do I really have to maintain 20 copies of my code, in each of those schemas?
- And... who or what is *DIANA*?

DIANA

In earlier editions of this book, the title of this chapter was "Inside PL/SQL." I decided to change the name to "PL/SQL Architecture" for two reasons:

- Most of what was in the chapter was not truly in any sense "internals." In fact, it is very difficult for PL/SQL developers (or anyone outside of Oracle headquarters) to get information about "internal" aspects of PL/SQL.
- I don't want to encourage you to try to uncover otherwise hidden aspects of PL/SQL. Developers, I believe, benefit most from learning the syntax of the language, not from trying to "game" or trick the PL/SQL compiler into doing something it wouldn't do of its own volition.

Having said that, a very common question that touches on the internal structures of the PL/SQL compiler is: "Who or what is DIANA?"

Asking a PL/SQL programmer "Who is Diana?" is like asking a San Francisco resident "Who's Bart?" The answer to both questions is not so much a *who* as a *what*. For the San Francisco Bay Area resident, BART is the Bay Area Rapid Transit system—the subway. For the PL/SQL programmer, DIANA is the Distributed Intermediate Annotated Notation for Ada and is part of PL/SQL's heritage as an Ada-derived language. In some Ada compilers, the output of the first part of the compilation is a DIANA. Likewise, PL/SQL was originally designed to output a DIANA in the first part of the compilation process.

As a PL/SQL programmer, however, you never really see or interact with your program's DIANA. Oracle Corporation may decide, like some Ada compiler publishers, that PL/SQL has outgrown DIANA, and another mechanism may be used. Changes in the internals of the database happen. For example, segment space management used to use free lists, but now uses bitmaps. So, while you might get an error if your DIANA grows too large (how embarrassing—your DIANA is showing!), you don't really do anything with DIANA, and if she (it) goes away, you probably won't even know.

Knowing about DIANA might win you a T-shirt in a trivia contest at Oracle Open World, but it probably won't improve your programming skills— unless you are programming the PL/SQL compiler.

So... enough with internals. Let's get on with our discussion of critical aspects of the PL/SQL architecture.

How Oracle Executes PL/SQL Code

Before I explore how an Oracle database executes PL/SQL programs, I first need to define a couple of terms of art:[1]

1. Wiktionary defines a *term of art* as "a term whose use or meaning is specific to a particular field of endeavor."

PL/SQL runtime engine (a.k.a. PL/SQL virtual machine)
: The PL/SQL virtual machine (PVM) is the database component that executes a PL/SQL program's bytecode. In this virtual machine, the bytecode of a PL/SQL program is translated to machine code that makes calls to the database server and returns results to the calling environment. The PVM itself is written in C. Historically, Oracle has included a PVM in some client-side tools such as Oracle Forms, where the runtime engine opens a session to a remote database, communicating with the SQL engine over a networking protocol.

Database session
: For most (server-side) PL/SQL, the database session is the process and memory space associated with an authenticated user connection (note: with the multithreaded server, or MTS, sessions can share the same process and the same session can use different processes). Each session is a logical entity in the database instance memory that represents the state of a current user logged into that database. The sessions connected to a database are visible through the view V$SESSION. Oracle maintains different kinds of memory for a session, including the process global area (PGA), user global area (UGA), and call global area (CGA). These are discussed later in the chapter.

To put these terms into context, let's take a look at several variations on running a trivial program from a very common frontend, SQL*Plus. This is a good representative of a session-oriented tool that gives you direct access to the PL/SQL environment inside the database server. (I introduced SQL*Plus and showed how to use it with PL/SQL back in Chapter 2.) Of course, you may be calling the server from other tools, such as Oracle's other client-side tools or even a procedural language such as Perl, C, or Java. But don't worry: processing on the server side is relatively independent of the client environment.

PL/SQL execution launched directly from SQL*Plus always involves a top-level anonymous block. While you may know that the SQL*Plus EXECUTE command converts the call to an anonymous block, did you know that SQL's CALL statement uses a (simplified) kind of anonymous block? Actually, until Oracle9*i* Database's direct invocation of PL/SQL from SQL, *all* PL/SQL invocations from SQL used anonymous blocks.

An Example

Let's begin with a look at the simplest possible anonymous block:

```
BEGIN
   NULL;
END;
```

and find out just what happens when you send this block to the database server (Figure 24-1).

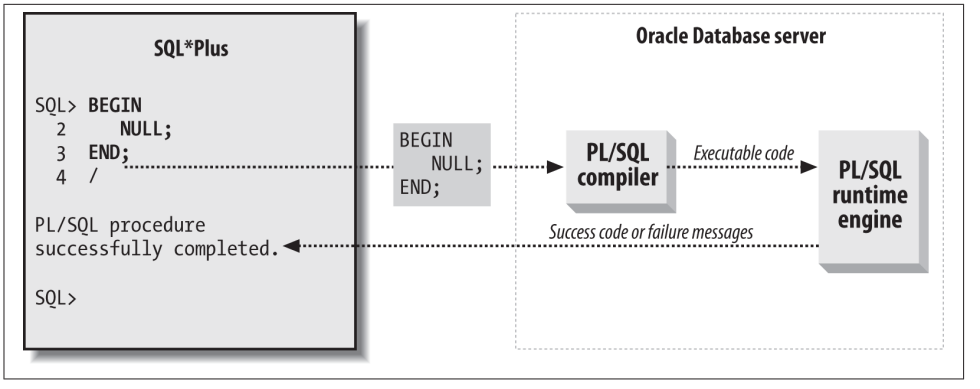

Figure 24-1. Execution of a trivial anonymous block

Let's step through the operations shown in this figure:

1. Starting on the left side of Figure 24-1, the user composes the source code for the block and then gives SQL*Plus the go-ahead command (a slash). As the figure shows, SQL*Plus sends the entire code block, exclusive of the slash, to the server. This transmission occurs over whatever connection the session has established (for example, Oracle Net or interprocess communication).
2. Next, the PL/SQL compiler tries to convert this anonymous block to bytecode.[2] A first phase is to check the syntax to ensure that the program adheres to the grammar of the language. In this simple case, there are no identifiers to figure out, only language keywords. If compilation succeeds, the database puts the block's compiled form (the bytecode) into a shared memory area; if it fails, the compiler will return error messages to the SQL*Plus session.
3. Finally, the PVM interprets the bytecode and ultimately returns a success or failure code to the SQL*Plus session.

Let's add an embedded SQL query statement into the anonymous block and see how that changes the picture. Figure 24-2 introduces some of the SQL-related elements of the database server.

2. Actually, if some session previously needed the database to compile the block, there is a good chance that the compile phase won't need to be repeated. That is because the server caches the outputs of relatively expensive operations like compilation in memory and tries to share them.

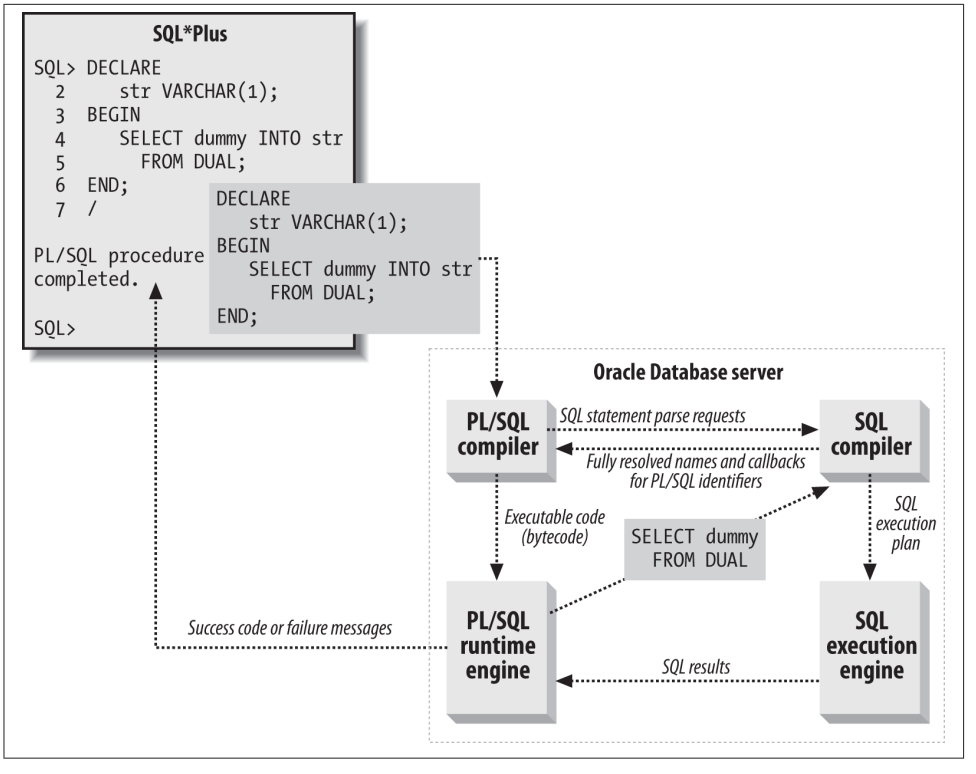

Figure 24-2. Execution of an anonymous block that contains SQL

This example fetches a column value from the well-known table DUAL.[3]

After checking that the PL/SQL portions of the code adhere to the language's syntax, the PL/SQL compiler communicates with the SQL compiler to hand off any embedded SQL for execution. Likewise, PL/SQL programs called from SQL statements cause the SQL compiler to hand off the PL/SQL calls to the PL/SQL compiler. The SQL parser will resolve any expressions, look for opportunities to use the function result cache (starting with Oracle Database 11*g*), execute semantic and syntax checks, perform name resolution, and determine an optimal execution plan. These steps are all part of the parse phase of the SQL execution and precede any substitution of bind variables and the execution and fetch of the SQL statement.

While PL/SQL shares a SQL compiler with the database, this does not mean that every SQL function is available in PL/SQL. For example, SQL supports the NVL2 function:

```
SELECT NVL2(NULL, 1, 2) FROM DUAL;
```

3. According to a reputable source, the name DUAL is from its dual singularity: one row, one column.

But attempting to use NVL2 directly in PL/SQL results in a *PLS-00201: identifier 'NVL2' must be declared* error:

```
SQL> EXEC DBMS_OUTPUT.PUT_LINE(NVL2(NULL, 1, 2));
   2 BEGIN DBMS_OUTPUT.PUT_LINE (NVL2(NULL, 1, 2)); END;

*
ERROR at line 1:
ORA-06550: line 1, column 28:
PLS-00201: identifier 'NVL2' must be declared
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored
```

When a PL/SQL program is compiled, embedded SQL statements are modified slightly such that INTO clauses are removed, local program variables are replaced with bind variables, comments are removed, and many of the identifiers in the statement are uppercased (including keywords, column names, table names, etc., but *not* hints and literals). For example, if myvar is a local program variable, PL/SQL will change this:

```
select dummy into str from dual where dummy = myvar
```

into something like this:

```
SELECT DUMMY FROM DUAL WHERE DUMMY = :B1
```

There are two other kinds of callouts you can make from PL/SQL:

Java stored procedures
: The default database server installation includes not just a PL/SQL virtual machine but also a Java virtual machine. You can write a PL/SQL call spec whose logic is implemented as a static Java class. For details and examples, see Chapter 27 on the book's website.

External procedures
: You can also implement the executable portion of a PL/SQL subprogram in custom C code, and at runtime the database will run your code in a separate process and memory space from the main database server. You are responsible for backing up these binaries and making sure each RAC node has a copy. Chapter 28 discusses external procedures and is also available on the book's website.

You can learn more about the runtime architecture of these two approaches by consulting their respective chapters.

Compiler Limits

Large PL/SQL programs may encounter the server error *PLS-00123: Program too large*. This means that the compiler bumped into the maximum allowed number of "nodes" in the parse tree. The normal workaround for this error is to split the program into several smaller programs or reengineer it (for example, to use a temporary table instead of 10,000 parameters). It's difficult to predict how many nodes a program will

need because nodes don't directly correspond to anything easily measurable, such as tokens or lines of code.

Oracle advises that a "typical" stored program generate four parse tree nodes per line of code; this equates to the approximate upper limits specified in the following table.

PL/SQL program type	Upper limit (estimated)
Package and type bodies; standalone functions and procedures	256 MB
Signatures (headers) of standalone functions and procedures	128 KB
Package and type specifications; anonymous blocks	128 KB

These are only estimates, though, and there can be a fair amount of variance in either direction.

Other documented hard limits in the PL/SQL compiler include those in the following table.

PL/SQL elements	Upper limit (estimated)
Levels of block nesting	255
Parameters you can pass to a procedure or function	65,536
Levels of record nesting	64
Objects referenced in a program unit	65,536
Number of exception handlers in one program	65,536
Precision of a NUMBER (digits)	38
Size of a VARCHAR2 (bytes)	32,767

Few of these are likely to cause a problem, but you can find a complete list of them in Oracle's official PL/SQL documentation, the *PL/SQL Language Reference*.

The Default Packages of PL/SQL

A true object-oriented language like Java has a root class (in Java it is called Object, not surprisingly) from which all other classes are derived. PL/SQL is, officially, an object-relational language, but at its core it is a relational, procedural programming language and it has at *its* core a "root" package named STANDARD.

The packages you build are not derived from STANDARD, but almost every program you write will *depend on* and use this package. It is, in fact, one of the two *default* packages of PL/SQL, the other being DBMS_STANDARD.

To best understand the role that these packages play in your programming environment, it is worth traveling back in time to the late 1980s, before the days of Oracle7 and SQL*Forms 3, before Oracle PL/SQL even existed. Oracle had discovered that while SQL was a wonderful language, it couldn't do everything. Its customers found them-

selves writing C programs that executed the SQL statements, but those C programs had to be modified to run on each different operating system.

Oracle decided that it would create a programming language that could execute SQL statements natively and be portable across all operating systems on which the Oracle database was installed. The company also decided that rather than come up with a brand-new language on its own, it would evaluate existing languages and see if any of them could serve as the model for what became PL/SQL.

In the end, Oracle chose Ada as that model. Ada was originally designed for use by the US Department of Defense, and was named after Ada Lovelace, an early and widely respected software programming pioneer. Packages are a construct adopted from Ada. In the Ada language, you can specify a "default package" in any given program unit. When a package is the default, you do not have to qualify references to elements in the package with the *package_name dot* syntax, as in *my_package.call_procedure*.

When Oracle designed PL/SQL, it kept the idea of a default package but changed the way it is applied. We (users of PL/SQL) are not allowed to specify a default package in a program unit. Instead, there are just two default packages in PL/SQL, STANDARD and DBMS_STANDARD. They are defaults for the entire language, not for any specific program unit.

You can (and almost always will) reference elements in either of these packages without using the package name as a dot-qualified prefix. Let's now explore how Oracle uses the STANDARD package (and to a lesser extent, DBMS_STANDARD) to, as stated in the *Oracle PL/SQL User Guide*, "define the PL/SQL environment."

STANDARD declares a set of types, exceptions, and subprograms that are automatically available to all PL/SQL programs and would be considered (mistakenly) by many PL/SQL developers to be "reserved words" in the language. When compiling your code, Oracle must resolve all unqualified identifiers. It first checks to see if an element with that name is declared in the current scope. If not, it then checks to see if there is an element with that name defined in STANDARD or DBMS_STANDARD. If a match is found for each identifier in your program, the code can be compiled (assuming there are no syntax errors).

To understand the role of STANDARD, consider the following (very strange-looking) block of PL/SQL code. What do you think will happen when I execute this block?

```
   /* File on web: standard_demo.sql */
1  DECLARE
2     SUBTYPE DATE IS NUMBER;
3     VARCHAR2 DATE := 11111;
4     TO_CHAR       PLS_INTEGER;
5     NO_DATA_FOUND EXCEPTION;
6  BEGIN
7     SELECT 1 INTO TO_CHAR
```

```
 8       FROM SYS.DUAL WHERE 1 = 2;
 9  EXCEPTION
10    WHEN NO_DATA_FOUND
11    THEN
12       DBMS_OUTPUT.put_line ('Trapped!');
13  END;
```

Most PL/SQL developers will say one of two things: "This block won't even compile," or "It will display the word 'Trapped!' since 1 is never equal to 2."

In fact, the block will compile, but when you run it you will see an unhandled NO_DATA_FOUND exception:

```
ORA-01403: no data found
ORA-06512: at line 7
```

Now isn't that odd? NO_DATA_FOUND is the *only* exception I am actually handling, so how can it escape unhandled? Ah, but the question is: *which* NO_DATA_FOUND am I handling? You see, in this block, I have declared my *own* exception named NO_DATA_FOUND. This name is not a reserved word in the PL/SQL language (in contrast, BEGIN is a reserved word—you cannot name a variable "BEGIN"). Instead, it is an exception that is defined in the specification of the STANDARD package, as follows:

```
NO_DATA_FOUND exception;
PRAGMA EXCEPTION_INIT(NO_DATA_FOUND, 100);
```

Since I have a locally declared exception with the name NO_DATA_FOUND, any *unqualified* reference to this identifier in my block will be resolved as *my* exception and not STANDARD's exception.

If, on the other hand, line 12 in my exception section looked like this:

```
WHEN STANDARD.NO_DATA_FOUND
```

then the exception would be handled and the word "Trapped!" displayed.

In addition to the oddness of NO_DATA_FOUND, the lines pointed out in the following table also appear to be rather strange.

Line(s)	Description
2	Define a new type of data named DATE, which is actually of type NUMBER.
3	Declare a variable named VARCHAR2 of type DATA and assign it a value of 11111.
4	Declare a variable named TO_CHAR of type PLS_INTEGER.

I can "repurpose" these names of "built-in" elements of the PL/SQL language, because they are all defined in the STANDARD package. These names are not reserved by PL/SQL; they are simply and conveniently referenceable without their package name.

The STANDARD package contains the definitions of the supported datatypes in PL/SQL, the predefined exceptions, and the built-in functions, such as TO_CHAR, SYSDATE, and USER. The DBMS_STANDARD package contains transaction-related elements, such as COMMIT, ROLLBACK, and the trigger event functions INSERTING, DELETING, and UPDATING.

Here are a few things to note about STANDARD:

- You should never change the contents of this package. If you do, I suggest that you not contact Oracle Support and ask for help. You have likely just violated your maintenance agreement! Your DBA *should* give you read-only authority on the *RDBMS/Admin* directory, so that you can examine this package along with any of the other supplied packages, like DBMS_OUTPUT (check out *dbmsotpt.sql*) and DBMS_UTILITY (check out *dbmsutil.sql*).
- Oracle even lets you read the package body of STANDARD; unlike most package bodies, such as that for DBMS_SQL, it is not wrapped or pseudoencrypted. Look in the *stdbody.sql* script and you will see, for instance, that the USER function *always* executes a SELECT from SYS.dual, while SYSDATE will only execute a query if a C program to retrieve the system timestamp fails.
- Just because you see a statement in STANDARD doesn't mean you can write that same code in your own PL/SQL blocks. You cannot, for example, declare a subtype with a range of values, as is done for BINARY_INTEGER.
- Just because you see something defined in STANDARD doesn't mean you can use it in PL/SQL. For example, the DECODE function is declared in STANDARD, but it can be called only from within a SQL statement.

The STANDARD package is defined by the *stdspec.sql* and *stdbody.sql* files in *$ORACLE_HOME/RDBMS/Admin* (in some earlier versions of the database, this package may be found in the *standard.sql* file). You will find DBMS_STANDARD in *dbmsstdx.sql*.

If you are curious about which of the many predefined identifiers are actually reserved words in the PL/SQL language, check out the *reserved_words.sql* script on the book's website. This script is explained in Chapter 3.

Execution Authority Models

The Oracle database offers two different models for object permissions in your PL/SQL programs. The default (and the only model way back in the days before Oracle8*i* Database) is *definer rights*. With this model, a stored program executes under the authority

of its owner, or *definer*.[4] The other permission model uses the privileges of the user invoking the program and is referred to as *invoker rights*.

You need to understand the nuances of both the definer rights model and the invoker rights model, because many PL/SQL applications rely on a combination of the two. Let's explore these in a little more detail, so you know when you want to use each model.

The Definer Rights Model

Before a PL/SQL program can be executed from within a database instance, it must be compiled and stored in the database itself. Thus, a program unit is always stored within a specific schema or database account, even though the program might refer to objects in other schemas.

With the *definer rights model*, keep the following rules in mind:

- Any external reference in a program unit is resolved at compile time, using the directly granted privileges of the schema in which the program unit is compiled.
- Database roles are ignored completely when stored programs are compiling. All privileges needed for the program must be granted directly to the definer (owner) of the program.
- Whenever you run a program compiled with the definer rights model (the default), its SQL executes under the authority of the schema that owns the program.
- Although direct grants are needed to compile a program, you can grant EXECUTE authority to give other schemas and roles the ability to run your program.

Figure 24-3 shows how you can use the definer rights model to control access to underlying data objects. All the order entry data is stored in the OEData schema. All the order entry code is defined in the OECode schema. OECode has been granted the direct privileges necessary to compile the Order_Mgt package, which allows you to both place and cancel orders.

4. It was possible to get invoker rights by using a loopback database link.

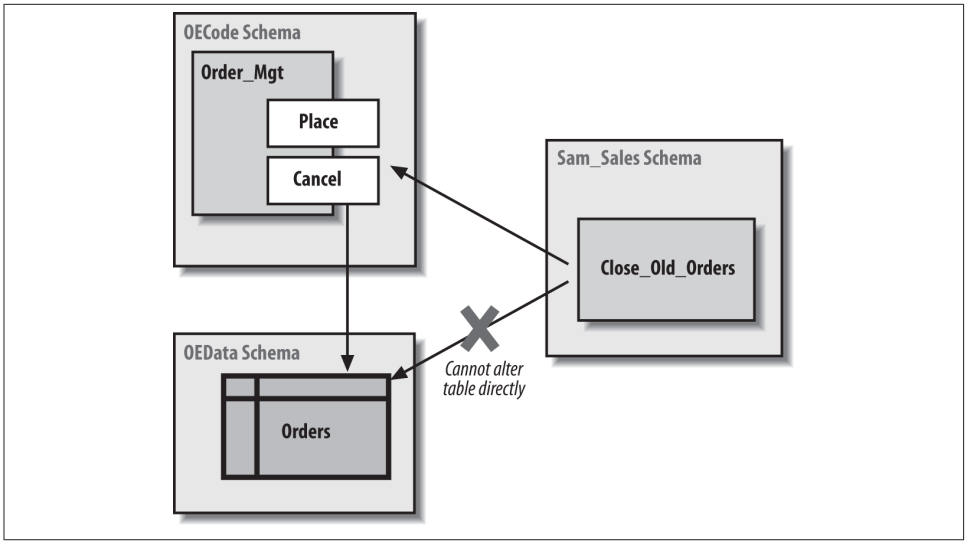

Figure 24-3. Controlling access to data with the definer rights model

To make sure that the orders table is updated properly, no direct access (either via roles or via privileges) is allowed to that table through any schema other than OECode. Suppose, for example, that the Sam_Sales schema needs to run through all the outstanding orders and close out old ones. Sam will not be able to issue a DELETE statement from the Close_Old_Orders procedure; instead, he will have to call Order_Mgt.cancel to get the job done.

Advantages of definer rights

Certain situations cry out for definer rights. This model offers the following advantages:

- You are better able to control access to underlying data structures. You can guarantee that the only way the contents of a table can be changed is by going through a specific programmatic interface (usually a package).
- Application performance improves dramatically because the PL/SQL engine does not have to perform checks at runtime to determine if you have the appropriate privileges or—just as important—which object you should actually be manipulating (my accounts table may be quite different from yours!).
- You don't have to worry about manipulating the wrong table. With definer rights, your code will work with the same data structure you would be accessing directly in SQL in your SQL*Plus (or other execution) environment. It is simply more intuitive.

Disadvantages of definer rights

But there are problems with the definer rights model as well. These are explored in the remainder of this section.

Where'd my table go?. Let's see what all those definer rights rules can mean to a PL/SQL developer on a day-to-day basis. In many databases, developers write code against tables and views that are owned by other schemas, with public synonyms created for them to hide the schema. Privileges are then granted via database roles.

This very common setup can result in some frustrating experiences. Suppose that my organization relies on roles to grant access to objects. I am working with a table called accounts, and I can execute this query without any problem in SQL*Plus:

```
SQL> SELECT account#, name FROM accounts;
```

Yet when I try to use that same table (and the same query, even) inside a procedure, I get an error:

```
SQL> CREATE OR REPLACE PROCEDURE show_accounts
  2  IS
  3  BEGIN
  4     FOR rec IN (SELECT account#, name FROM accounts)
  5     LOOP
  6        DBMS_OUTPUT.PUT_LINE (rec.name);
  7     END LOOP;
  8  END;
  9  /

Warning: Procedure created with compilation errors.

SQL> sho err
Errors for PROCEDURE SHOW_ACCOUNTS:

LINE/COL ERROR
-------- -----------------------------------------------------
4/16     PL/SQL: SQL Statement ignored
4/43     PLS-00201: identifier 'ACCOUNTS' must be declared
```

This doesn't make any sense... or does it? The problem is that ACCOUNTS is actually owned by another schema; I was unknowingly relying on a synonym and roles to get at the data. So, if you are ever faced with this seemingly contradictory situation, don't bang your head against the wall in frustration. Instead, ask the owner of the object or the DBA to grant you the privileges you require to get the job done.

How do I maintain all that code?. Suppose that my database instance is set up with a separate schema for each of the regional offices in my company. I build a large body of code that each office uses to analyze and maintain its data. Each schema has its own tables

with the same structure but different data, a design selected for both data security and ease of movement via transportable tablespaces.

Now, I would like to install this code so that I spend the absolute minimum amount of time and effort setting up and maintaining the application. The way to do that is to install the code in one schema and share that code among all the regional office schemas.

With the definer rights model, unfortunately, this goal and architecture are impossible to achieve. If I install the code in a central schema and grant the EXECUTE privilege authority to all regional schemas, then all those offices will be working with whatever set of tables is accessible to the central schema (perhaps one particular regional office or, more likely, a dummy set of tables). That's no good. I must instead install this body of code in each separate regional schema, as shown in Figure 24-4.

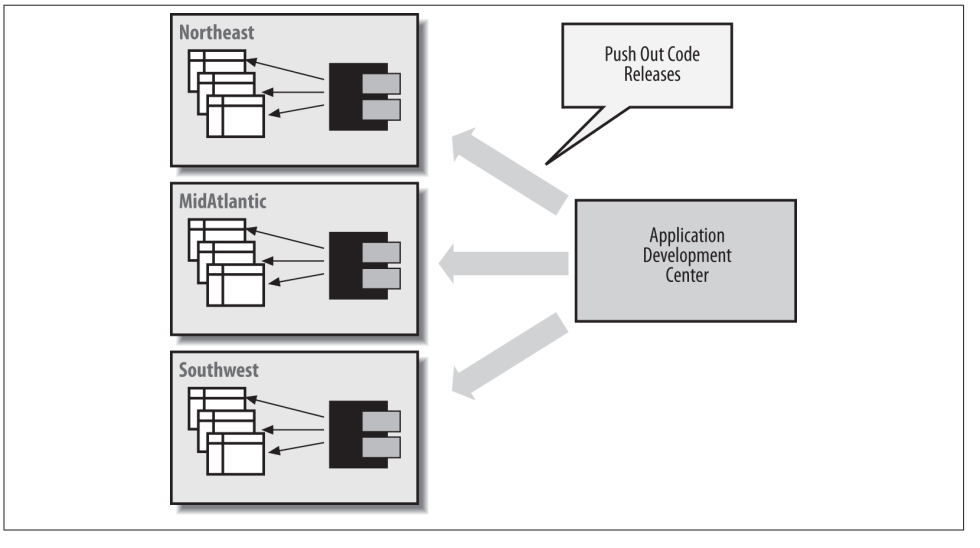

Figure 24-4. Repetitive installations of code needed with definer rights

The result is a maintenance and enhancement nightmare. Perhaps invoker rights will give us options for a better solution.

Dynamic SQL and definer rights. Another common source of confusion with definer rights occurs when using dynamic SQL (described in Chapter 16). Suppose that I create a generic "exec DDL" program (yes, this is a really bad idea security-wise, but it illustrates the unintended consequences of this learning exercise) as follows:

```
/* File on web: execddl.sp */
PROCEDURE execDDL (ddl_string IN VARCHAR2)
IS
BEGIN
```

```
    EXECUTE IMMEDIATE ddl_string;
END;
```

After testing it in my schema with outstanding results, I decide to share this neat utility with everyone else in my development organization. I compile it into the COMMON schema (where all reusable code is managed), grant EXECUTE to PUBLIC, and create a public synonym. Then I send out an email announcing its availability.

A few weeks later, I start getting calls from my coworkers. "Hey, I asked it to create a table, and it ran without any errors, but I don't have the table." "I asked it to drop my table, and the execddl procedure said that there is no such table. But I can do a DESCRIBE on it." You get the idea. I begin to have serious doubts about sharing my code with other people. Sheesh, if they can't use something as simple as the execddl procedure without screwing things up... but I decide to withhold judgment and do some research.

I log into the COMMON schema and find that, sure enough, all of the objects people were trying to create or drop or alter are sitting here in COMMON. And then it dawns on me: unless a user of execddl specifies her own schema when she asks to create a table, the results will be most unexpected.

In other words, this call to execddl:

```
EXEC execddl ('CREATE TABLE newone (rightnow DATE)')
```

would create the newone table in the COMMON schema. And this call to execddl:

```
EXEC execddl ('CREATE TABLE scott.newone (rightnow DATE)')
```

might solve the problem, but would fail with the following error:

```
ORA-01031: insufficient privileges
```

unless I grant the CREATE ANY TABLE privilege to the COMMON schema. Yikes... my attempt to share a useful piece of code got very complicated very fast! It sure would be nice to let people run the execddl procedure under their own authority and not that of COMMON, without having to install multiple copies of the code.

Privilege escalation and SQL injection. The time to pause and review your design with a colleague is when your program executes dynamic SQL that relies on the owner's directly granted system privileges—and you think definer rights is appropriate to use. You should be very reluctant to create procedures like the previous one in the SYS schema or any other schema having system privileges that could serve as a gateway to privilege escalation if unexpected, but legal, input is passed in.

Please review Chapter 16 for information on how to prevent code injection.

The Invoker Rights Model

Sometimes, your programs should execute using the privileges of the person running the program and not the owner of the program. In such cases, you should choose the *invoker rights model*. With this approach, all external references in the SQL statements in a PL/SQL program unit are resolved according to the privileges of the invoking schema, not those of the owning or defining schema.

Figure 24-5 demonstrates the fundamental difference between the definer and invoker rights models. Recall that in Figure 24-4, I had to push out copies of my application to each regional office so that the code would manipulate the correct tables. With invoker rights, this step is no longer necessary. Now I can compile the code into a single code repository. When a user from the Northeast region executes the centralized program (probably via a synonym), it will work automatically with tables in the Northeast schema.

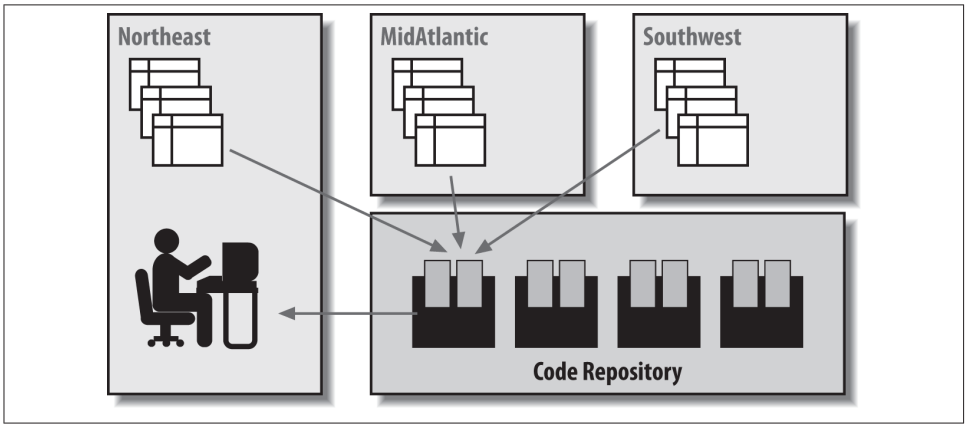

Figure 24-5. Use of invoker rights model

So that's the idea behind invoker rights. Let's see what is involved in terms of code, and then explore how best to exploit the feature.

Invoker rights syntax

The syntax to support this feature is simple enough. You add the following clause before your IS or AS keyword in the program header:

```
AUTHID CURRENT_USER
```

Here, for example, is that generic "exec DDL" engine again, this time defined as an invoker rights program:

```
/* File on web: execddl.sql */
PROCEDURE execddl (ddl_in in VARCHAR2)
```

```
   AUTHID CURRENT_USER
IS
BEGIN
   EXECUTE IMMEDIATE ddl_in;
END;
```

The AUTHID CURRENT_USER clause before the IS keyword indicates that when execddl executes, it should run under the authority of the invoker or "current user," not the authority of the definer. And that's all you have to do. If you do not include the AUTHID clause, or if you include it and explicitly request definer rights as shown here:

```
AUTHID DEFINER
```

then all references in your program will be resolved according to the directly granted privileges of the owning schema.

Invoker Rights for Dynamic SQL

I have written hundreds of programs using dynamic SQL, and with definer rights I would have to worry about schema issues. Where is the program running? Who is running it? What will happen when someone runs it? These are serious questions to ask about your code!

You may be tempted to use the AUTHID CURRENT_USER clause with every stored program unit that uses any kind of dynamic SQL. Once you take this step, you reason, you can rest assured that no matter where the program is compiled and which schema runs the program, it will always act on the currently connected schema.

The problem with this approach, though, is twofold. First, your users now require the same privileges that the programs have (and you may not want the HR clerk to be able to modify a salary outside the designated program). Second, there is extra runtime checking that the database must perform for invoker rights programs—and that can be a drag on performance. Use invoker rights where it is appropriate—neither model should be adopted blindly. I like to *think* about which privilege model my program needs and deliberately code to that model.

Rules and restrictions on invoker rights

There are a number of rules and restrictions to keep in mind when you are taking advantage of the invoker rights model:

- AUTHID DEFINER is the default option.
- The invoker rights model checks the privileges assigned to the invoker at the time of program execution to resolve any SQL-based references to database objects.
- With invoker rights, roles *are* in effect at runtime as long as the invoker rights program hasn't been called from a definer rights program.

- The AUTHID clause is allowed only in the header of a standalone subprogram (procedure or function), a package specification, or an object type specification. You cannot apply the AUTHID clause to individual programs or methods within a package or object type. So, the whole package will be invoker rights, or the whole package will be definer rights. If parts of your package should be invoker rights and parts should be definer rights, then you need two packages.
- Invoker rights resolution of external references will work for the following kinds of statements:
  - SELECT, INSERT, UPDATE, MERGE, and DELETE data manipulation statements
  - LOCK TABLE transaction control statements
  - OPEN and OPEN FOR cursor control statements
  - EXECUTE IMMEDIATE and OPEN FOR USING dynamic SQL statements
  - SQL statements parsed using DBMS_SQL.PARSE
- Definer rights will always be used to resolve all external references to PL/SQL programs and object type methods at compilation time.
- You can use invoker rights to change the resolution of static external data element references (tables and views).

You can also use invoker rights to resolve external references to PL/SQL programs. Here is one way to do it:

```
EXECUTE IMMEDIATE 'BEGIN someprogram; END;';
```

In this fragment, *someprogram* would get resolved at runtime according to the rights and namespace of the invoker. Alternatively, I could have used SQL's CALL statement instead of the anonymous block. (I can't just use a naked CALL statement because it is not directly supported within PL/SQL.)

Combining Rights Models

What do you think would happen if a definer rights program called an invoker rights program, or vice versa? The rules are simple:

- If a definer rights program calls an invoker rights program, the rights of the *calling* program's owner apply while the called program executes.
- If an invoker rights program calls a definer rights program, the rights of the *called* program's owner apply while the called program executes. When control returns to the caller, invoker rights resume.

To help keep all of this straight in your head, just remember that definer rights are "stronger" than (take precedence over) invoker rights.

Here are some files on the book's website that you can use to explore the nuances of the invoker rights model in more detail:

invdefinv.sql and invdefinv.tst
: Two scripts that demonstrate the impact of the precedence of definer rights over invoker rights

invdef_overhead.tst
: Examines the overhead of reliance on invoker rights (hint: runtime resolution is slower than compile-time resolution)

invrole.sql
: Demonstrates how a change in roles can affect how object references are resolved at runtime

irdynsql.sql
: Explores some of the complexities involved in using invoker and definer rights with dynamic SQL

Granting Roles to PL/SQL Program Units (Oracle Database 12c)

Before Oracle Database 12*c*, a definer rights program unit (defined with AUTHID DEFINER or no AUTHID clause) always executed with the privileges of the definer of that unit. An invoker rights program unit (defined with AUTHID CURRENT_USER clause) always executed with the privileges of the invoker of that unit.

A consequence of these two distinct AUTHID settings was that any program unit that needed to be executed by all users had to be created as a definer rights unit. It would then execute with *all* the privileges of the definer, which might not be optimal from a security standpoint.

As of Oracle Database 12*c*, you can grant *roles* to PL/SQL packages and schema-level procedures and functions. Role-based privileges for program units allow developers to fine-tune the privileges available to the invoker of a program unit.

We can now define a program unit as invoker rights and then *complement* the invoker's privileges with specific, limited privileges granted through the role.

Let's walk through an example showing how to grant roles to program units and the impact it has. Suppose that the HR schema contains the employees and departments tables, defined and populated with data as follows:

```
CREATE TABLE departments(
   department_id     INTEGER,
   department_name   VARCHAR2 (100),
   staff_freeze      CHAR (1)
)
/
BEGIN
```

```
   INSERT INTO departments
      VALUES (10, 'IT', 'Y');
   INSERT INTO departments
        VALUES (20, 'HR', 'N');
   COMMIT;
END;
/
CREATE TABLE employees
(
  employee_id     INTEGER,
  department_id   INTEGER,
   last_name       VARCHAR2 (100)
)
/
BEGIN
   DELETE FROM employees;
   INSERT INTO employees VALUES (100, 10, 'Price');
   INSERT INTO employees VALUES (101, 20, 'Sam');
   INSERT INTO employees VALUES (102, 20, 'Joseph');
   INSERT INTO employees VALUES (103, 20, 'Smith');
   COMMIT;
END;
/
```

The SCOTT schema contains *only* an employees table:

```
CREATE TABLE employees
(   employee_id     INTEGER,
  department_id   INTEGER,
   last_name       VARCHAR2 (100)
)
/
BEGIN
   DELETE FROM employees;
   INSERT INTO employees VALUES (100, 10, 'Price');
   INSERT INTO employees VALUES (101, 20, 'Sam');
   INSERT INTO employees VALUES (102, 20, 'Joseph');
   INSERT INTO employees VALUES (103, 20, 'Smith');
   COMMIT;
END;
/
```

HR also contains a procedure that removes all employees from the specified department as long as the department does not have its staff "frozen." I will first create this procedure as a definer rights unit:

```
CREATE OR REPLACE PROCEDURE remove_emps_in_dept (
   department_id_in IN employees.department_id%TYPE)
   AUTHID DEFINER
IS
   l_freeze   departments.staff_freeze%TYPE;
BEGIN
   SELECT staff_freeze
```

```
      INTO l_freeze
      FROM HR.departments
     WHERE department_id = department_id_in;
    IF l_freeze = 'N'
     THEN
        DELETE FROM employees
              WHERE department_id =
                       department_id_in;
     END IF;
  END;
  /
```

And allow SCOTT to execute this procedure:

```
GRANT EXECUTE
   ON remove_emps_in_dept
  TO SCOTT
/
```

When SCOTT executes the procedure as shown next, it will remove three rows—but from *HR's employees table*, since the procedure is a definer rights unit:

```
BEGIN
   HR.remove_emps_in_dept (20);
END;
/
```

I need to change this procedure so that it will remove rows from *SCOTT's* employees table, not HR's. That is precisely what invoker rights does for us. But if I change the AUTHID clause of this procedure to:

```
AUTHID CURRENT_USER
```

and run the procedure again, I see this:

```
BEGIN * ERROR at line 1:
ORA-00942: table or view does not exist
ORA-06512: at "HR.REMOVE_EMPS_IN_DEPT", line 7
ORA-06512: at line 2
```

The problem is that Oracle is now using the privileges of SCOTT to resolve the references to two tables: HR.departments and HR.employees. SCOTT has no privileges on HR's departments table, however, so Oracle raises the ORA-00942 error. Prior to Oracle Database 12*c*, the DBA would have had to grant the necessary privileges on HR.departments to SCOTT. Now, however, the following steps can be taken instead:

1. From a schema with the necessary privileges, create a role and grant that role to HR:

   ```
   CREATE ROLE hr_departments
   /
   GRANT hr_departments TO hr
   /
   ```

2. Connected to HR, grant the desired privilege to the role and *then* grant the role to the procedure:

```
GRANT SELECT
   ON departments
   TO hr_departments
/
GRANT hr_departments TO PROCEDURE remove_emps_in_dept
/
```

   Now when I execute the following statements from SCOTT, the rows are removed properly:

```
SELECT COUNT (*)
  FROM employees
 WHERE department_id = 20
/

  COUNT(*)
----------
         3

BEGIN
   hr.remove_emps_in_dept (20);
END;
/

SELECT COUNT (*)
  FROM employees
 WHERE department_id = 20
/

  COUNT(*)
----------
         0
```

Roles granted to a program unit do not affect compilation. Instead, they affect the privilege checking of SQL statements that the unit issues at runtime. Thus, the procedure or function executes with the privileges of *both* its own roles and any other currently enabled roles.

This feature will be of most use with invoker rights program units, as just shown. You will likely consider granting roles to a definer rights unit when that unit executes dynamic SQL, since the privileges for that dynamic statement are checked at runtime.

"Who Invoked Me?" Functions (Oracle Database 12c)

Oracle Database 12*c* offers two new functions (which can only be called from within SQL statements) that tell you the invoking user based on whether invoker rights or definer rights are used:

ORA_INVOKING_USER
: Returns the name of the user who is invoking the current statement or view. This function treats the intervening views as specified by their BEQUEATH clauses. If the invoking user is an Oracle Database Real Application Security–defined user (also new in 12.1, the next generation of the virtual private database feature), then this function returns XS$NULL.

ORA_INVOKING_USERID
: Returns the identifier (ID) of the user who is invoking the current statement or view. This function treats the intervening views as specified by their BEQUEATH clauses. If the invoking user is an Oracle Database Real Application Security–defined user, then this function returns an ID that is common to all Real Application Security sessions but is different from the ID of any database user.

These functions are demonstrated in the next section, when we explore BEQUEATH CURRENT_USER for views.

BEQUEATH CURRENT_USER for Views (Oracle Database 12c)

Prior to Oracle Database 12*c*, if your view executed a function, it would always be run under the privileges of the view's owner, not the function's owner. So if the function was defined as invoker rights, the behavior could be quite different from what you would have expected.

12*c* adds the BEQUEATH clause for views so that you can define a view that will accommodate invoker rights functions referenced in the view.

Let's take a look at how this feature works (all code in this section may be found in the *12c_bequeath.sql* file available on the book's website). Here, I will create a table and a function in the HR schema:

```
CREATE TABLE c12_emps
(
   employee_id     INTEGER,
   department_id   INTEGER,
   last_name       VARCHAR2 (100)
)
/

BEGIN
   INSERT INTO c12_emps VALUES (1, 100, 'abc');
   INSERT INTO c12_emps VALUES (2, 100, 'def');
   INSERT INTO c12_emps VALUES (3, 200, '123');
   COMMIT;
END;
/

CREATE OR REPLACE FUNCTION emps_count (
   department_id_in IN INTEGER)
```

```
   RETURN PLS_INTEGER
   AUTHID CURRENT_USER
IS
   l_count    PLS_INTEGER;
   l_user     VARCHAR2 (100);
   l_userid   VARCHAR2 (100);
BEGIN
   SELECT COUNT (*)
     INTO l_count
     FROM c12_emps
    WHERE department_id = department_id_in;

   /* Show who is invoking the function */
   SELECT ora_invoking_user, ora_invoking_userid
     INTO l_user, l_userid FROM DUAL;

   DBMS_OUTPUT.put_line ('Invoking user=' || l_user);
   DBMS_OUTPUT.put_line ('Invoking userID=' || l_userid);

   RETURN l_count;
END;
/
```

Notice that the function calls the two new invoking functions: ORA_INVOKING_USER and ORA_INVOKING_USER_ID.

Then I create a view, specifying invoker rights for the BEQUEATH setting, and I make sure that SCOTT can query that view:

```
CREATE OR REPLACE VIEW emp_counts_v
   BEQUEATH CURRENT_USER
AS
   SELECT department_id, emps_count (department_id) emps_in_dept
     FROM c12_emps
/

GRANT SELECT ON emp_counts_v TO scott
/
```

In the SCOTT schema, I create another c12_emps table, but I populate it with different data:

```
CREATE TABLE c12_emps
(
   employee_id     INTEGER,
   department_id   INTEGER,
   last_name       VARCHAR2 (100)
)
/

BEGIN
   INSERT INTO c12_emps VALUES (1, 200, 'SCOTT.ABC');
   INSERT INTO c12_emps VALUES (2, 200, 'SCOTT.DEF');
```

```
    INSERT INTO c12_emps VALUES (3, 400, 'SCOTT.123');
    COMMIT;
END;
/
```

Then I select all the rows from the view. Here's the output of the query:

```
SQL> SELECT * FROM hr.emp_counts_v
  2  /

DEPARTMENT_ID EMPS_IN_DEPT
------------- ------------
          100            0
          100            0
          200            2

SCOTT
107
SCOTT
107
SCOTT
107
```

As you can see, the data returned by the view is from HR's table (there is a department ID of 100), but the totals returned by the function call reflect data in SCOTT's table. And the ORA_INVOKING* functions return SCOTT's information.

Note that BEQUEATH CURRENT_USER does *not* transform the view itself into an invoker rights object. Name resolution within the view is still handled using the view owner's schema, and privilege checking for the view is done using the view owner's privileges.

The primary benefit of this feature is that it allows functions like SYS_CONTEXT and USERENV to return consistent results when these functions are referenced in a view.

Constraining Invoker Rights Privileges (Oracle Database 12c)

When a user runs a procedure or function defined in a PL/SQL program unit that has been created with the AUTHID CURRENT_USER clause, that subprogram temporarily inherits all of the privileges of the invoking user while the procedure runs. Conversely, during that time, the *owner* of the invoker rights unit has access to the invoking user's privileges.

If the invoker has a more powerful set of privileges than the owner, there is a risk that the subprogram's owner could misuse the higher privileges of the invoking user. These types of risks increase when application users are given access to a database that uses invoker rights procedures.

Oracle Database 12*c* adds two new privileges that you can use to control whether the owner's procedure can access the invoker's privileges: INHERIT PRIVILEGES and INHERIT ANY PRIVILEGES.

Any user can grant or revoke the INHERIT PRIVILEGES privilege on himself to the users whose invoker rights procedures he wants to run. The INHERIT ANY PRIVILEGES privilege is managed by SYS users.

When a user runs an invoker rights procedure, Oracle Database checks it to ensure that either the procedure owner has the INHERIT PRIVILEGES privilege on the invoking user, or the owner has been granted the INHERIT ANY PRIVILEGES privilege. If the privilege check fails, then Oracle Database returns the following error:

```
ORA-06598: insufficient INHERIT PRIVILEGES privilege error.
```

The benefit of these two privileges is that they give invoking users control over who can access their privileges when they run an invoker rights procedure or query a BEQUEATH CURRENT_USER view.

All users are granted INHERIT PRIVILEGES ON USER *newuser* TO PUBLIC by default, when their accounts are created or when accounts that were created earlier are upgraded to the current release. This means that if you do not have a risk of "weak owners" maliciously leveraging privileges of "strong invokers," everything will work the same as it did in earlier versions of Oracle Database.

An invoking user can revoke the INHERIT PRIVILEGES privilege from other users and grant it only to trusted users. The syntax for the INHERIT PRIVILEGES privilege grant is as follows

```
GRANT INHERIT PRIVILEGES ON USER invoking_user TO procedure_owner
```

Conditional Compilation

Introduced in Oracle Database 10*g* Release 2, *conditional compilation* allows the compiler to compile selected parts of a program based on conditions you provide with the $IF directive.

Conditional compilation will come in very handy when you need to:

- Write a program that will run under different versions of Oracle, taking advantage of features specific to those versions. More specifically, you may want to take advantage of new features of the Oracle database where available, but you also need your program to compile and run in older versions. Without conditional compilation, you would have to maintain multiple files or use complex SQL*Plus substitution variable logic.
- Run certain code during testing and debugging, but then omit that code from the production code. Prior to conditional compilation, you would need to either com-

ment out lines of code or add some overhead to the processing of your application —even in production.

- Install/compile different elements of your application based on user requirements, such as the components for which a user is licensed. Conditional compilation greatly simplifies the maintenance of a code base with this complexity.

You implement conditional compilation by placing compiler *directives* (commands) in your source code. When your program is compiled, the PL/SQL preprocessor evaluates the directives and selects those portions of your code that should be compiled. This pared-down source code is then passed to the compiler for compilation.

There are three types of directives:

Selection directives
: Use the $IF directive to evaluate expressions and determine which code should be included or avoided.

Inquiry directives
: Use the $$*identifier* syntax to refer to conditional compilation flags. These inquiry directives can be referenced within an $IF directive or used independently in your code.

Error directives
: Use the $ERROR directive to report compilation errors based on conditions evaluated when the preprocessor prepares your code for compilation.

I'll first show some simple examples, then delve more deeply into the capabilities of each directive. You'll also learn how to use two packages related to conditional compilation, DBMS_DB_VERSION and DBMS_PREPROCESSOR.

Examples of Conditional Compilation

Let's start with some examples of different types of conditional compilation.

Use application package constants in $IF directive

The $IF directive can reference constants defined in your own packages. In the following example, I vary the way that a bonus is applied depending on whether or not the location in which this third-party application is installed complies with the Sarbanes-Oxley guidelines. Such a setting is unlikely to change for a long period of time. If I rely on the traditional conditional statement in this case, I will leave in place a branch of logic that should never be applied. With conditional compilation, the code is removed before compilation:

```
/* File on web: cc_my_package.sql */
PROCEDURE apply_bonus (
   id_in IN employee.employee_id%TYPE
```

```
  ,bonus_in IN employee.bonus%TYPE)
IS
BEGIN
   UPDATE employee
      SET bonus =
       $IF employee_rp.apply_sarbanes_oxley
       $THEN
          LEAST (bonus_in, 10000)
       $ELSE
          bonus_in
       $END
     WHERE employee_id = id_in;
   NULL;
END apply_bonus;
```

Toggle tracing through conditional compilation flags

I can now set up my own debug/trace mechanisms and have them conditionally compiled into my code. This means that when my code rolls into production, I can have this code completely removed so that there will be no runtime overhead to this logic. Note that I can specify both Boolean and PLS_INTEGER values through the special PLSQL_CCFLAGS compile parameter:

```
/* File on web: cc_debug_trace.sql */
ALTER SESSION SET PLSQL_CCFLAGS = 'oe_debug:true, oe_trace_level:10';

PROCEDURE calculate_totals
IS
BEGIN
$IF $$oe_debug AND $$oe_trace_level >= 5
$THEN
   DBMS_OUTPUT.PUT_LINE ('Tracing at level 5 or higher');
$END
   NULL;
END calculate_totals;
```

The Inquiry Directive

An inquiry directive is a directive that makes an inquiry of the compilation environment. Of course, that doesn't really tell you much, so let's take a closer look at the syntax for inquiry directives and the different sources of information available through the inquiry directive.

The syntax for an inquiry directive is as follows:

```
$$identifier
```

where *identifier* is a valid PL/SQL identifier that can represent any of the following:

Compilation environment settings
: The values found in the USER_PLSQL_OBJECT_SETTINGS data dictionary view

Your own custom-named directive
: Defined with the ALTER...SET PLSQL_CCFLAGS command, described in "Using the PLSQL_CCFLAGS parameter" on page 1070

Implicitly defined directives
: $$PLSQL_LINE and $$PLSQL_UNIT, providing you with the line number and program name

Inquiry directives are designed for use within conditional compilation clauses, but they can also be used in other places in your PL/SQL code. For example, I can display the current line number in my program with this code:

```
DBMS_OUTPUT.PUT_LINE ($$PLSQL_LINE);
```

I can also use inquiry directives to define and apply application-wide constants in my code. Suppose, for example, that the maximum number of years of data supported in my application is 100. Rather than hardcoding this value in my code, I can do the following:

```
ALTER SESSION SET PLSQL_CCFLAGS = 'max_years:100';

PROCEDURE work_with_data (num_years_in IN PLS_INTEGER)
IS
BEGIN
   IF num_years_in > $$max_years THEN ...
END  work_with_data;
```

Even more valuable, I can use inquiry directives in places in my code where a variable is not allowed. Here are two examples:

```
DECLARE
   l_big_string VARCHAR2($$MAX_VARCHAR2_SIZE);

   l_default_app_err EXCEPTION;
   PRAGMA EXCEPTION_INIT (l_default_app_err, $$DEF_APP_ERR_CODE);
BEGIN
```

The DBMS_DB_VERSION package

The DBMS_DB_VERSION built-in package offers a set of constants that give you absolute and relative information about the version of your installed database. The constants defined in the 12.1 version of this package are shown in Table 24-1. With each new version of Oracle, two new relative constants are added, and the values returned by the VERSION and RELEASE constants are updated.

Table 24-1. DBMS_DB _VERSION constants

Name of packaged constant	Description	Value in Oracle Database 12*c* Release 1
DBMS_DB_VERSION.VERSION	The database version number	12
DBMS_DB_VERSION.RELEASE	The database release number	1
DBMS_DB_VERSION.VER_LE_9	TRUE if the current version is less than or equal to Oracle9*i* Database	FALSE
DBMS_DB_VERSION.VER_LE_9_1	TRUE if the current version is less than or equal to 9.1	FALSE
DBMS_DB_VERSION.VER_LE_9_2	TRUE if the current version is less than or equal to 9.2	FALSE
DBMS_DB_VERSION.VER_LE_10	TRUE if the current version is less than or equal to Oracle Database 10*g*	FALSE
DBMS_DB_VERSION.VER_LE_10_1	TRUE if the current version is less than or equal to10.1	FALSE
DBMS_DB_VERSION.VER_LE_10_2	TRUE if the current version is less than or equal to 10.2	FALSE
DBMS_DB_VERSION.VER_LE_11_1	TRUE if the current version is less than or equal to 11.1	FALSE
DBMS_DB_VERSION.VER_LE_11_2	TRUE if the current version is less than or equal to 11.2	FALSE
DBMS_DB_VERSION.VER_LE_12	TRUE if the current version is less than or equal to 12.1	TRUE
DBMS_DB_VERSION.VER_LE_12_1	TRUE if the current version is less than or equal to 12.1	TRUE

While this package was designed for use with conditional compilation, you can, of course, use it for your own purposes.

Interestingly, you can write expressions that include references to as-yet-undefined constants in the DBMS_DB_VERSION package. As long as they are not evaluated, as in the following case, they will not cause any errors. Here is an example:

```
$IF DBMS_DB_VERSION.VER_LE_12_1
$THEN
   Use this code.
$ELSIF DBMS_DB_VERSION.VER_LE_13
   This is a placeholder for the future.
$END
```

Setting compilation environment parameters

The following information (corresponding to the values in the USER_PLSQL_OBJECT_SETTINGS data dictionary view) is available via inquiry directives:

$$PLSQL_DEBUG
: Debug setting for this compilation unit

$$PLSQL_OPTIMIZE_LEVEL
: Optimization level for this compilation unit

$$PLSQL_CODE_TYPE
: Compilation mode for this compilation unit

$$PLSQL_WARNINGS
: Compilation warnings setting for this compilation unit

$$NLS_LENGTH_SEMANTICS
: Value set for the NLS length semantics

See the *cc_plsql_parameters.sql* file on the book's website for a demonstration that uses each of these parameters.

Referencing unit name and line number

Oracle implicitly defines four very useful inquiry directives for use in $IF and $ERROR directives:

$$PLSQL_UNIT
: The name of the compilation unit in which the reference appears. If that unit is an anonymous block, then $$PLSQL_UNIT contains a NULL value.

$$PLSQL_LINE
: The line number of the compilation unit where the reference appears.

$$PLSQL_UNIT_OWNER (Oracle Database 12c)
: The name of the owner of the current PL/SQL program unit. If that unit is an anonymous block, then $$PLSQL_UNIT_OWNER contains a NULL value.

$$PLSQL_UNIT_TYPE (Oracle Database 12c)
: The type of the current PL/SQL program unit—ANONYMOUS BLOCK, FUNCTION, PACKAGE, PACKAGE BODY, PROCEDURE, TRIGGER, TYPE, or TYPE BODY. Inside an anonymous block or non-DML trigger, $$PLSQL_UNIT_TYPE has the value ANONYMOUS BLOCK.

You can call two built-in functions, DBMS_UTILITY.FORMAT_CALL_STACK and DBMS_UTILITY.FORMAT_ERROR_BACKTRACE, to obtain current line numbers, but then you must also parse those strings to find the line number and program unit name. These inquiry directives provide the information more directly. Here is an example:

```
BEGIN
   IF l_balance < 10000
   THEN
      raise_error (
         err_name => 'BALANCE TOO LOW'
        ,failed_in => $$plsql_unit
        ,failed_on => $$plsql_line
      );
   END IF;
   ...
END;
```

Run the *cc_line_unit.sql* file on the book's website to see a demonstration of using these last two directives.

Note that when $$PLSQL_UNIT is referenced inside a package, it will return the name of the package, not the name of the individual procedure or function within the package.

Using the PLSQL_CCFLAGS parameter

Oracle offers a parameter, PLSQL_CCFLAGS, that you can use with conditional compilation. Essentially, it allows you to define name/value pairs, and the name can then be referenced as an inquiry directive in your conditional compilation logic. Here is an example:

```
ALTER SESSION SET PLSQL_CCFLAGS = 'use_debug:TRUE, trace_level:10';
```

The flag name can be set to any valid PL/SQL identifier, including reserved words and keywords (the identifier will be prefixed with $$, so there will be no confusion with normal PL/SQL code). The value assigned to the name must be one of the following: TRUE, FALSE, NULL, or a PLS_INTEGER literal.

The PLSQL_CCFLAGS value will be associated with each program that is then compiled in that session. If you want to keep those settings with the program, then future compilations with the ALTER...COMPILE command should include the REUSE SETTINGS clause.

Because you can change the value of this parameter and then compile selected program units, you can easily define different sets of inquiry directives for different programs.

Note that you can refer to a flag that is *not* defined in PLSQL_CCFLAGS; this flag will evaluate to NULL. If you enable compile-time warnings, this reference to an undefined flag will cause the database to report a *PLW-06003: unknown inquiry directive* warning (unless the source code is wrapped).

The $IF Directive

Use the selection directive, implemented through the $IF statement, to direct the conditional compilation step in the preprocessor. Here is the general syntax of this directive:

```
$IF Boolean-expression
$THEN
   code_fragment
[$ELSIF Boolean-expression
$THEN
   code_fragment]
[$ELSE
   code_fragment]
$END
```

where *Boolean-expression* is a static expression (it can be evaluated at the time of compilation) that evaluates to TRUE, FALSE, or NULL. The *code_fragment* can be any set of PL/SQL statements, which will then be passed to the compiler for compilation, as directed by the expression evaluations.

Static expressions can be constructed from any of the following elements:

- Boolean, PLS_INTEGER, and NULL literals, plus combinations of these literals.
- Boolean, PLS_INTEGER, and VARCHAR2 static expressions.
- Inquiry directives (i.e., identifiers prefixed with $$). These directives can be provided by Oracle (e.g., $$PLSQL_OPTIMIZE_LEVEL; the full list is provided in the section "The Optimizing Compiler" on page 838) or set via the PLSQL_CCFLAGS compilation parameter (explained earlier in this chapter).
- Static constants defined in a PL/SQL package.
- Most comparison operations (>, <, =, and <> are fine, but you cannot use an IN expression), logical Boolean operations such as AND and OR, concatenations of static character expressions, and tests for NULL.

A static expression may not contain calls to procedures or functions that require execution; they cannot be evaluated during compilation and therefore will render invalid the expression within the $IF directive. You will get a compile error as follows:

```
PLS-00174: a static boolean expression must be used
```

Here are a few examples of static expressions in $IF directives:

- If the user-defined inquiry directive controlling debugging is not NULL, then initialize the debug subsystem:

```
$IF $$app_debug_level IS NOT NULL $THEN
   debug_pkg.initialize;
$END
```

- Check the value of a user-defined package constant along with the optimization level:

```
$IF $$PLSQL_OPTIMIZE_LEVEL = 2 AND appdef_pkg.long_compilation
$THEN
   $ERROR 'Do not use optimization level 2 for this program!' $END
$END
```

String literals and concatenations of strings are allowed only in the $ERROR directive; they may not appear in the $IF directive.

The $ERROR Directive

Use the $ERROR directive to cause the current compilation to fail and return the error message provided. The syntax of this directive is:

```
$ERROR VARCHAR2_expression $END
```

Suppose that I need to set the optimization level for a particular program unit to 1, so that compilation time will be improved. In the following example, I use the $$ inquiry directive to check the value of the optimization level from the compilation environment. I then raise an error with the $ERROR directive as necessary:

```
/* File on web: cc_opt_level_check.sql */
SQL> CREATE OR REPLACE PROCEDURE long_compilation
  2  IS
  3  BEGIN
  4  $IF $$plsql_optimize_level != 1
  5  $THEN
  6     $ERROR 'This program must be compiled with optimization level = 1' $END
  7  $END
  8     NULL;
  9  END long_compilation;
 10  /

Warning: Procedure created with compilation errors.

SQL> SHOW ERRORS
Errors for PROCEDURE LONG_COMPILATION:

LINE/COL ERROR
-------- -----------------------------------------------------------------
6/4      PLS-00179: $ERROR: This program must be compiled with
         optimization level = 1
```

Synchronizing Code with Packaged Constants

Use of packaged constants within a selection directive allows you to easily synchronize multiple program units around a specific conditional compilation setting. This is possible because Oracle's automatic dependency management is applied to selection directives. In other words, if program unit PROG contains a selection directive that references package PKG, then PROG is marked as dependent on PKG. When the specification of PKG is recompiled, all program units using the packaged constant are marked INVALID and must be recompiled.

Suppose I want to use conditional compilation to automatically include or exclude debugging and tracing logic in my code base. I define a package specification to hold the required constants:

```
/* File on web: cc_debug.pks */
PACKAGE cc_debug
```

```
IS
   debug_active CONSTANT BOOLEAN := TRUE;
   trace_level CONSTANT PLS_INTEGER := 10;
END cc_debug;
```

I then use these constants in the procedure calc_totals:

```
PROCEDURE calc_totals
IS
BEGIN
$IF cc_debug.debug_active AND cc_debug.trace_level > 5 $THEN
   log_info (...);
$END
   ...
END calc_totals;
```

During development, the debug_active constant is initialized to TRUE. When it is time to move the code to production, I change the flag to FALSE and recompile the package. The calc_totals program and all other programs with similar selection directives are marked INVALID and must then be recompiled.

Program-Specific Settings with Inquiry Directives

Packaged constants are useful for coordinating settings across multiple program units. Inquiry directives, drawn from the compilation settings of individual programs, are a better fit when you need different settings applied to different programs.

Once you have compiled a program with a particular set of values, it will retain those values until the next compilation (from either a file or a simple recompilation using the ALTER...COMPILE command). Furthermore, a program is guaranteed to be recompiled with the same postprocessed source as was selected at the time of the *previous* compilation if all of the following conditions are TRUE:

- None of the conditional compilation directives refer to package constants. Instead, they rely only on inquiry directives.
- When the program is recompiled, the REUSE SETTINGS clause is used *and* the PLSQL_CCFLAGS parameter is not included in the ALTER...COMPILE command.

This capability is demonstrated by the *cc_reuse_settings.sql* script (available on the book's website), whose output is shown next. I first set the value of app_debug to TRUE and then compile a program with that setting. A query against USER_PLSQL_OBJECT_SETTINGS shows that this value is now associated with the program unit:

```
/* File on web: cc_reuse_settings.sql */

SQL> ALTER SESSION SET plsql_ccflags = 'app_debug:TRUE';
```

```
SQL> CREATE OR REPLACE PROCEDURE test_ccflags
  2  IS
  3  BEGIN
  4     NULL;
  5  END test_ccflags;
  6  /

SQL> SELECT name, plsql_ccflags
  2    FROM user_plsql_object_settings
  3   WHERE NAME LIKE '%CCFLAGS%';

NAME                           PLSQL_CCFLAGS
------------------------------ ----------------------------
TEST_CCFLAGS                   app_debug:TRUE
```

I now alter the session, setting $$app_debug to evaluate to FALSE. I compile a new program with this setting:

```
SQL> ALTER SESSION SET plsql_ccflags = 'app_debug:FALSE';

SQL> CREATE OR REPLACE PROCEDURE test_ccflags_new
  2  IS
  3  BEGIN
  4     NULL;
  5  END test_ccflags_new;
  6/
```

Then I recompile my existing program with REUSE SETTINGS:

```
SQL> ALTER  PROCEDURE test_ccflags COMPILE REUSE SETTINGS;
```

A query against the data dictionary view now reveals that my settings are different for each program:

```
SQL> SELECT name, plsql_ccflags
  2    FROM user_plsql_object_settings
  3   WHERE NAME LIKE '%CCFLAGS%';

NAME                           PLSQL_CCFLAGS
------------------------------ ----------------------------
TEST_CCFLAGS                   app_debug:TRUE
TEST_CCFLAGS_NEW               app_debug:FALSE
```

Working with Postprocessed Code

You can use the DBMS_PREPROCESSOR package to display or retrieve the source text of your program in its postprocessed form. DBMS_PREPROCESSOR offers two programs, overloaded to allow you to specify the object of interest in various ways, as well as to work with individual strings and collections:

DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE
: Retrieves the postprocessed source and then displays it with the function DBMS_OUTPUT.PUT_LINE

DBMS_PREPROCESSOR.GET_POST_PROCESSED_SOURCE
: Returns the postprocessed source as either a single string or a collection of strings

When working with the collection version of either of these programs, you will need to declare that collection based on the following package-defined collection:

```
TYPE DBMS_PREPROCESSOR.source_lines_t IS TABLE OF VARCHAR2(32767)
   INDEX BY BINARY_INTEGER;
```

The following sequence demonstrates the capability of these programs. First I compile a very small program with a selection directive based on the optimization level. I then display the postprocessed code, and it shows the correct branch of the $IF statement:

```
/* File on web: cc_postprocessor.sql */
PROCEDURE post_processed
IS
BEGIN
$IF $$PLSQL_OPTIMIZE_LEVEL = 1
$THEN
   -- Slow and easy
  NULL;
$ELSE
   -- Fast and modern and easy
   NULL;
$END
END post_processed;


SQL> BEGIN
  2     DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE (
  3        'PROCEDURE', USER, 'POST_PROCESSED');
  4  END;
 5  /

PROCEDURE post_processed
IS
BEGIN
-- Fast and modern and easy
NULL;
END post_processed;
```

In the following block, I use the "get" function to retrieve the postprocessed code, and then display it using DBMS_OUTPUT.PUT_LINE:

```
DECLARE
   l_postproc_code   DBMS_PREPROCESSOR.SOURCE_LINES_T;
   l_row             PLS_INTEGER;
BEGIN
```

```
   l_postproc_code :=
      DBMS_PREPROCESSOR.GET_POST_PROCESSED_SOURCE (
         'PROCEDURE', USER, 'POST_PROCESSED');
   l_row := l_postproc_code.FIRST;

   WHILE (l_row IS NOT NULL)
   LOOP
      DBMS_OUTPUT.put_line (  LPAD (l_row, 3)
                            || ' - '
                            || rtrim ( l_postproc_code (l_row),chr(10))
                           );
      l_row := l_postproc_code.NEXT (l_row);
   END LOOP;
END;
```

Conditional compilation opens up all sorts of possibilities for PL/SQL developers and application administrators. And its usefulness only increases as new versions of the Oracle database are released and the DBMS_DB_VERSION constants can be put to full use, allowing us to take additional advantage of each version's unique PL/SQL features.

PL/SQL and Database Instance Memory

By economizing on its use of machine resources such as memory and CPU, an Oracle database can support tens of thousands of simultaneous users on a single database. Oracle's memory management techniques have become quite sophisticated over the years, and correspondingly difficult to understand. Fortunately, the picture as it applies to PL/SQL and session-related memory is relatively simple. In addition, while there are a few things developers can do to reduce memory consumption in their PL/SQL code, database memory configuration and allocation is for the most part handled by DBAs.

The SGA, PGA, and UGA

When a client program such as SQL*Plus or SQL Developer interacts with the database, three memory structures may be used.

System global area (SGA)

The SGA is a group of shared memory structures, or *SGA components*, that contain data and control information for a single Oracle Database instance. All server and background processes share the SGA. Examples of data stored in the SGA include cached data blocks and shared SQL areas.

Process global area (PGA)

A PGA is a memory region that is not shared with multiple sessions and instead contains data and control information exclusively for use by a single Oracle process. Oracle Database creates the PGA when an Oracle process starts. There is one PGA for each

server process and background process. The collection of individual PGAs is referred to as the *total instance PGA*, or *instance PGA*. Database initialization parameters set the size of the instance PGA, not individual PGAs.

User global area (UGA)

The UGA is memory allocated for storing information associated with a user session, such as logon information, and other information required by a database session. Essentially, the UGA stores the session state.

The location of the UGA in memory depends on whether the session has connected to the database using a dedicated or shared server mode.

Dedicated server mode
: The database spawns a dedicated server process for each session. This is appropriate for workloads that either are intensive or involve long-running database calls. The UGA is placed in the PGA because no other server process will need to access it.

Shared server mode
: Database calls are queued to a group of shared server processes that can service calls on behalf of any session. This is appropriate if there are many hundreds of concurrent sessions making short calls with a lot of intervening idle time. The UGA is placed in the SGA so that it can be accessed by any of the shared server processes.

The total size of the PGA can vary quite a bit based on what kinds of operations your application requires the server to perform. A PL/SQL package that populates a large PL/SQL collection in a package-level variable may, for example, consume large amounts of UGA memory.

If your application uses shared server mode, user processes may have to wait in a queue to be serviced. If any of your user processes invoke long-running PL/SQL blocks or SQL statements, the DBA may need to either configure the server with a greater number of shared processes or assign to those sessions a dedicated server.

Let's next consider what memory looks like to an individual running program.

Cursors, Memory, and More

You may have written hundreds of programs that declare, open, fetch from, and close cursors. It's impossible to execute SQL or PL/SQL without cursors, and statements often automatically make recursive calls that open more cursors. Because every cursor, whether implicit or explicit, requires an allocation of memory on the database server, tuning the database sometimes involves reducing the number of cursors required by an application.

Although this section is devoted to memory, keep in mind that memory is only one aspect of tuning the database; you may actually improve overall performance by increasing the number of cursors to avoid soft parses.

The database assigns cursors to anonymous PL/SQL blocks in much the same way that it assigns cursors to SQL statements. For example, on the first parse call from a session, the database opens an area in UGA memory (the "private SQL area") where it will put information specific to the run.

When executing a SQL statement or a PL/SQL block, the server first looks in the library cache to see if it can find a reusable parsed representation of it. If it does find such a shared PL/SQL area, the runtime engine establishes an association between the private SQL area and the shared SQL area. The shared cursor has to be found or built before the private SQL area can be allocated. The private SQL area memory requirements are part of what is determined during compilation and cached in the shared cursor. If no reusable shared area exists, the database will "hard parse" the statement or block. (As an aside, note that the database also prepares and caches a simple execution plan for anonymous PL/SQL blocks, which consists of calling the PL/SQL engine to interpret the bytecode.)

The database interprets the simplest of PL/SQL blocks—those that call no subprograms and include no embedded SQL statements—using only the memory allocated for its primary cursor. If your PL/SQL program includes SQL or PL/SQL calls, though, the database requires additional private SQL areas in the UGA. PL/SQL manages these on behalf of your application.

This brings us to another important fact about cursors: there are two ways a cursor can be closed. A *soft-closed* cursor is one that you can no longer use in your application without reopening it. This is what you get when you close a cursor using a statement such as this one:

```
CLOSE cursor_name;
```

or even when an implicit cursor closes automatically. However, PL/SQL does not immediately free the session memory associated with this cursor. Instead, it caches the cursor to avoid a soft parse should the cursor be opened again, as often happens. You will see, if you look in the V$OPEN_CURSOR view, that the CLOSE alone does not reduce the count of this session's open cursors; as of 11.1, you can also fetch the new CURSOR_TYPE column value to get additional information about a cursor.

It turns out that PL/SQL maintains its own "session cursor cache"; that is, it decides when to close a cursor for good. This cache can hold a maximum number of cursors, as specified by the OPEN_CURSORS database initialization parameter. A least recently

used (LRU) algorithm determines which of the soft-closed cursors need to be *hard closed* and hence deallocated.

However, PL/SQL's internal algorithm works optimally only if your programs close their cursors immediately after they are through fetching with them. So remember:

> If you explicitly open a cursor, you should explicitly close it as soon as you are through using it (but not sooner).

There are a few ways that the database allows PL/SQL programmers to intervene in the default behavior. One way you can close all of your session cursors, of course, is to terminate the session! Less drastic ways include:

- Reset the package state, as discussed at the end of the section "Large collections in PL/SQL" on page 1085.
- Use DBMS_SQL to gain explicit control over low-level cursor behavior. On the whole, though, memory gains provided by this approach are unlikely to offset the corresponding performance costs and programming challenges.

Tips on Reducing Memory Use

Armed with a bit of theory, let's review some practical tips you can use in your day-to-day programming. Also check out the more general program tuning hints in Chapter 21. In addition, it helps to be able to *measure* the amount of memory your session is using at any given point in time in your application code. You can do this by querying the contents of various V$ views. The plsql_memory package (defined in the *plsql_memory.pkg* file on the book's website) will help you do this.

Statement sharing

The database can share the source and compiled versions of SQL statements and anonymous blocks, even if they are submitted from different sessions by different users. The optimizer determines the execution plan at parse time, so factors that affect parsing (including optimizer settings) will affect SQL statement sharing. For sharing to happen, certain conditions must be true:

- Variable data values must be supplied via bind variables rather than literals so that the text of the statements will not differ. Bind variables themselves must match in name and datatype.
- The letter case and formatting conventions of the source code must match exactly. If you are executing the same programs, this will happen automatically. Ad hoc statements may not match those in programs exactly.
- References to database objects must resolve to the same underlying object.

- For SQL, database parameters influencing the SQL optimizer must match. For example, the invoking sessions must be using the same "optimizer goal" (ALL_ROWS versus FIRST_ROWS).
- The invoking sessions must be using the same language (Globalization Support; formerly National Language Support, or NLS) environment.

I'm not going to talk much about the SQL-specific rules; specific reasons for nonsharing of SQL statements that otherwise pass these rules can be found in the view V$SQL_SHARED_CURSOR. For now, let's explore the impact of the first three rules on your PL/SQL programs.

Rule #1, regarding bind variables, is so critical that it has a later section devoted to it.

Rule #2, matching letter case and formatting, is a well-known condition for sharing statements. The text has to match because the database computes a hash value from its source text with which to locate and lock an object in the library cache.

Despite the fact that PL/SQL is normally a case-independent language, the block:

```
BEGIN NULL; END;
```

does not match:

```
begin null; end;
```

nor does it match:

```
BEGIN NULL;   END;
```

These statements will each hash to a different value and are inherently different SQL statements—they are logically the same but physically different. However, if all your anonymous blocks are short, and all your "real programs" are in stored code such as packages, there is much less chance of inadvertently disabling code sharing. The tip here is:

> Centralize your SQL and PL/SQL code in stored programs. Anonymous blocks should be as short as possible, generally consisting of a single call to a stored program.

In addition, an extension of this tip applies to SQL:

> To maximize the sharing of SQL statements, put SQL into programs. Then call these programs rather than writing the SQL you need in each block.

I've always felt that trying to force statement sharing by adopting strict formatting conventions for SQL statements was just too impractical; it's much easier to put the SQL into a callable program.

Moving on, Rule #3 says that database object references (to tables, procedures, etc.) must resolve to the same underlying object. Say that Scott and I are connected to the same database, and we both run a block that goes like this:

```
BEGIN
   XYZ;
END;
```

The database's decision about whether to share the cached form of this anonymous block boils down to whether the name xyz refers to the same underlying stored procedure. If Scott has a synonym xyz that points to my copy of the procedure, then the database will share this anonymous block; if Scott and I own independent copies of xyz, the database will not share this anonymous block. Even if the two copies of the xyz procedure are line-by-line identical, the database will cache these as different objects. The database also caches identical triggers on different tables as different objects. That leads to the following tip:

> Avoid proliferating copies of tables and programs in different accounts unless you have a good reason.

How far should you go with sharing of code at the application level? The conventional wisdom holds that you should identify and extract code that is common to multiple programs (especially triggers) and incorporate it by call rather than by duplicating it. In other words, set up one database account to own the PL/SQL programs, and grant the EXECUTE privilege to any other users who need it. While sharing code in this manner is an excellent practice for code maintainability, it is not likely to save any memory. In fact, every caller has to instantiate the additional object at a cost of typically several kilobytes per session. Of course, this won't be a huge burden unless you have enormous numbers of users.

There is another caveat on the conventional wisdom, and it applies if you are running in a high-concurrency environment—that is, many users simultaneously executing the same PL/SQL program. Whenever common code is called, a "library cache latch" is needed to establish and then release a pin on the object. In high-concurrency environments, this can lead to latch contention. In such cases, duplicating the code wherever it is needed may be preferred because doing so will avoid extra latching for the additional object and reduce the risk of poor performance due to latch contention.

But now, let's go back to Rule #1: use bind variables.

Bind variables

In an Oracle database, a *bind variable* is an input variable to a statement whose value is passed by the caller's execution environment. Bind variables are especially significant in the sharing of SQL statements, regardless of whether the statements are submitted by PL/SQL, Java, SQL*Plus, or OCI. Application developers using virtually any environment should understand and use bind variables. Bind variables allow an application to scale, help prevent code injection, and allow SQL statement sharing.

A requirement for two different statements to be considered identical is that any bind variables must themselves match in name, datatype, and maximum length. So, for example, the SQL statements given here do *not* match:

```
SELECT col FROM tab1 WHERE col = :bind1;
SELECT col FROM tab1 WHERE col = :bind_1;
```

But this requirement applies to the text of the statement as seen by the SQL engine. As mentioned much earlier in this chapter, PL/SQL rewrites your static SQL statements before SQL ever sees them! Here's an example:

```
FUNCTION plsql_bookcount (author IN VARCHAR2)
   RETURN NUMBER
IS
   title_pattern VARCHAR2(10) := '%PL/SQL%';
   l_count NUMBER;
BEGIN
   SELECT COUNT(*) INTO l_count
     FROM books
    WHERE title LIKE title_pattern
      AND author = plsql_bookcount.author;
   RETURN l_count;
END;
```

After executing plsql_bookcount, the V$SQLAREA view in Oracle Database 11*g* reveals that PL/SQL has rewritten the query as:

```
SELECT COUNT(*) FROM BOOKS WHERE TITLE LIKE :B2 AND AUTHOR = :B1
```

The parameter author and the local variable title_pattern have been replaced by the bind variables :B1 and :B2. This implies that, in static SQL, you don't need to worry about matching bind variable names; PL/SQL replaces your variable name with a generated bind variable name.

This automatic introduction of bind variables in PL/SQL applies to program variables that you use in the WHERE and VALUES clauses of static INSERT, UPDATE, MERGE, DELETE, and of course SELECT statements.

In additional experiments, I have found that changing the PL/SQL variable to have a different maximum length did not *always* result in an additional statement in the SQL area, but that changing the variable's datatype can add a statement. Don't take my word for it, though: privileges permitting, you can run your own experiments to determine whether SQL statements are being shared in the way you think they are. Look in V$SQLAREA. For the code just listed (assuming that I am the only person running this particular code):

```
SQL> SELECT executions, sql_text
   2   FROM v$sqlarea
   3  WHERE sql_text like 'SELECT COUNT(*) FROM BOOKS%'

EXECUTIONS SQL_TEXT
```

```
---------- ------------------------------------------------
         1 SELECT COUNT(*) FROM BOOKS WHERE TITLE LIKE :B2
           AND AUTHOR = :B1
```

You might say, "Well, if PL/SQL is that smart, I don't need to worry about bind variables then, do I?" Hang on there: even though PL/SQL automatically binds program variables into *static* SQL statements, this feature is not automatic when using *dynamic* SQL. Sloppy programming can easily result in statements getting built with literal values. For example:

```
FUNCTION count_recent_records (tablename_in IN VARCHAR2,
   since_in IN VARCHAR2)
RETURN PLS_INTEGER
AS
   l_count PLS_INTEGER;
BEGIN
   EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM '
      || DBMS_ASSERT.SIMPLE_SQL_NAME(tablename_in)
      || ' WHERE lastupdate > TO_DATE('
      || DBMS_ASSERT.ENQUOTE_LITERAL(since_in)
      || ', ''YYYYMMDD'')'
      INTO l_count;
   RETURN l_count;
END;
```

This causes the dynamic construction of statements such as:

```
SELECT COUNT(*) FROM tabname WHERE lastupdate > TO_DATE('20090315','YYYYMMDD')
```

Repeated invocation with different since_in arguments can result in a lot of statements that are unlikely to be shared. For example:

```
SELECT COUNT(*) FROM tabname WHERE lastupdate > TO_DATE('20090105','YYYYMMDD')
SELECT COUNT(*) FROM tabname WHERE lastupdate > TO_DATE('20080704','YYYYMMDD')
SELECT COUNT(*) FROM tabname WHERE lastupdate > TO_DATE('20090101','YYYYMMDD')
```

This is wasteful of memory and other server resources.

Use DBMS_ASSERT to Avoid Code Injection

What are those calls to DBMS_ASSERT in the bind variable example? Well, dynamic SQL that uses raw user input should be validated before you blindly execute it. DBMS_ASSERT helps to ensure that the code being passed in is what you expect. If you tried to call the count_recent_records function with a weird-looking table name like "books where 1=1;--", then DBMS_ASSERT would raise an exception stopping the program before damage was done. DBMS_ASSERT.SIMPLE_SQL_NAME ensures that the input passes muster as a legitimate SQL name. The DBMS_ASSERT.ENQUOTE_LITERAL encloses the input in quotes and makes sure that there are no unmatched em-

bedded quote characters. See the Oracle's *PL/SQL Packages and Types Reference* for full details on DBMS_ASSERT.

A bind variable version of this program would be:

```
FUNCTION count_recent_records (tablename_in IN VARCHAR2,
   since_in IN VARCHAR2)
RETURN PLS_INTEGER
AS
   count_l PLS_INTEGER;
BEGIN
   EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM '
      || DBMS_ASSERT.SIMPLE_SQL_NAME(tablename_in)
      || ' WHERE lastupdate > :thedate'
      INTO count_l
      USING TO_DATE(since_in,'YYYYMMDD');
   RETURN count_l;
END;
```

which results in statements that look like this to the SQL compiler:

```
SELECT COUNT(*) FROM tabname WHERE lastupdate > :thedate
```

Not only is the second version prettier and easier to follow, but it will also perform much better over repeated invocations with the same tablename_in but with different since_in arguments.

The database offers the initialization parameter CURSOR_SHARING, which *may* provide some benefits to applications with a lot of non-bind-variable SQL. By setting this parameter to FORCE or SIMILAR (deprecated in 12.1), you can ask the database to replace some or all SQL literals with bind variables at runtime, thus avoiding some of the hard-parse overhead. Unfortunately, this is one of those "sounds better in theory than in practice" features.

Even if you can derive some performance benefits from using CURSOR_SHARING, you should view it *only as a stopgap measure*. It's not nearly as efficient as using true bind variables and can have a number of unexpected and undesirable side effects. If you must use this feature with certain pathological (often third-party) software, do so only until the code can be modified to use true bind variables. Note that you can also set this up on a session level with a LOGON trigger.

On the other hand, if you consistently exercise the small amount of discipline required to use true bind variables in your dynamic SQL, you will be rewarded, perhaps richly so, at runtime. Just remember to keep your CURSOR_SHARING parameter set to its default value of EXACT.

Packaging to improve memory use and performance

When retrieving the bytecode of a stored PL/SQL program, the database reads the entire program. This rule applies not only to procedures and functions, but also to packages. In other words, you can't get the database to retrieve only a part of a package; the first time any session uses some element of a package, even just a single package variable, the database loads the compiled code for the entire package into the library cache. Having fewer large package instantiations requires less memory (and disk) overhead than more smaller instantiations. It also minimizes the number of pins taken and released, which is very important in high-concurrency applications. So, a logical grouping of package elements is not just a good design idea; it will also help your system's performance.

Because the database reads an entire package into memory at once, design each of your packages with functionally related components that are likely to be invoked together.

Large collections in PL/SQL

Sharing is a wonderful thing, but of course not everything can be shared at runtime. Even when two or more users are executing the same program owned by the same schema, each session has its own private memory area, which holds run-specific data such as the values of local or package variables, constants, and cursors. It wouldn't make much sense to try to share values that are specific to a given session.

Large collections are a case in point (Chapter 12 describes collections in detail). Imagine that I declare a PL/SQL associative array as follows:

```
DECLARE
   TYPE number_tab_t IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
   number_tab number_tab_t;
   empty_tab number_tab_t;
```

Now I create a bunch of elements in this array:

```
   FOR i IN 1..100000
   LOOP
      number_tab(i) := i;
   END LOOP;
```

The database has to put all of those elements somewhere. Following the rules discussed earlier, memory for this array will come from the UGA in the case of package-level data, or the CGA (call global area) in the case of data in anonymous blocks or top-level procedures or functions. Either way, it's easy to see that manipulating large collections can require very large amounts of memory.

You may wonder how you can get that memory back once a program is through with it. This is a case where the natural and easy thing to do will help quite a bit. You can use one of these two forms:

```
number_tab.DELETE;
```

or:

```
number_tab := empty_tab;
```

Using either of these statements will cause the database to free the memory into its originating free list; that is, package-level memory frees into the session state heap, call-level memory frees into the CGA, and the CGA is freed into the PGA at the end of the call. The same thing happens when the collection passes out of scope; if you declare and use the collection only in a standalone procedure, the database realizes that you don't need it anymore after the procedure finishes executing. Either way, though, this memory is not available to other sessions, nor is it available to the current session for CGA memory requirements. So, if a subsequent DML operation requires a large sort, you could wind up with some huge memory requirements. Not until the session ends will the database release this memory to its parent heap.

I should point out that it is no great hardship for a virtual memory operating system with plenty of paging/swap space if processes retain large amounts of inactive virtual memory in their address spaces. This inactive memory consumes only paging space, not real memory. There may be times, however, when you don't want to fill up paging space, and you would prefer that the database release the memory. For those times, the database supplies an on-demand "garbage collection" procedure. The syntax is simply:

```
DBMS_SESSION.FREE_UNUSED_USER_MEMORY;
```

This built-in procedure will find most of the UGA memory that is no longer in use by any program variables and release it back to the parent memory heap—the PGA in the case of a dedicated server process, or the SGA in the case of shared server processes.

I have run quite a few test cases to determine the effect of running garbage collection in different scenarios: for example, associative arrays versus nested tables, shared server versus dedicated server mode, and anonymous blocks versus package data. The conclusions and tips that are described next apply to using large collections:

- Merely assigning a NULL to a nested table or VARRAY will fail to mark its memory as unused. Instead, you can do one of three things: use the method *collection*.DELETE, assign a null but initialized collection to it, or wait for it to go out of scope.
- If you need to release memory to the parent heap, use DBMS_SESSION.FREE_UNUSED_USER_MEMORY when your program has populated one or more large PL/SQL tables, marked them as unused, and is unlikely to need further large memory allocations for similar operations.

- Shared server mode can be more prone than dedicated server mode to memory-shortage errors. This is because the UGA is drawn from the SGA, which is limited in size. As discussed in the section "What to Do If You Run Out of Memory" on page 1090, you may get an ORA-04031 error.
- If you must use shared server connections, you cannot release the memory occupied by PL/SQL tables unless the table is declared at the package level.

As a practical matter, for a collection of NUMBER elements, there seems to be no difference in the amount of storage required to store NULL elements versus, say, 38-digit number elements. However, the database does seem to allocate memory for VARCHAR2 elements dynamically if the elements are declared larger than VARCHAR2(30).

When populating an associative array in dedicated server mode, a million-element associative array of NUMBERs occupies about 38 MB; even if the million elements are just Booleans, almost 15 MB of memory is required. Multiply that by even 100 users, and you're talking some big numbers, especially if you don't want the operating system to start paging this memory out to disk.

If you'd like to discover for yourself how much UGA and PGA your current session uses, you can run a query like the following:

```
SELECT n.name, ROUND(m.value/1024) kbytes
  FROM V$STATNAME n, V$MYSTAT m
 WHERE n.statistic# = m.statistic#
   AND n.name LIKE 'session%memory%'
```

(You'll need nondefault privileges to read the two V$ views in this query.) This will show you the "current" and the "max" memory usage thus far in your session.

Incidentally, if you want to clear out the memory used by packaged collections but don't want to terminate the session (for example, if you are running scripts that test memory usage), you can use one of these built-ins:

DBMS_SESSION.RESET_PACKAGE
: Frees all memory allocated to package state. This has the effect of resetting *all* package variables to their default values. For packages, this built-in goes beyond what FREE_UNUSED_USER_MEMORY does because RESET_PACKAGE doesn't care whether the memory is in use or not.

DBMS_SESSION.MODIFY_PACKAGE_STATE (action_flag IN PLS_INTEGER)
: You can supply one of two constants as the *action flag*: DBMS_SESSION.free_all_resources or DBMS_SESSION.reinitialize. The first has the same effect as using the RESET_PACKAGE procedure. Supplying the latter constant resets state variables to their defaults but doesn't actually free and recreate the package instantiation from scratch; also, it only soft closes open cursors and does not flush the cursor cache. If these behaviors are acceptable in your application, use the second constant because it will perform better than a complete reset.

BULK COLLECT...LIMIT operations

Bulk binds are a great way to process data efficiently, but you should take care to limit your memory consumption and not let your collections grow too large. When you BULK COLLECT into a collection, the default is to fetch all the rows into the collection. When you have a lot of data, this results in a very large collection. That's where the LIMIT clause comes into play. It allows you to limit your memory consumption and make your programs faster.

When I benchmarked the LIMIT clause, I expected it to use less memory but was surprised to find that it ran faster too. Here is the example benchmark, using a test table containing a million rows. To get good numbers for comparison, I ran it once to warm up the cache, reconnected to zero out my memory consumption, and then ran it a second time to compare. I've included calls to the plsql_memory package (see the *plsql_memory.pkg* file on the book's website) to report on memory use:

```
/* File on web: LimitBulkCollect.sql */
DECLARE
  -- set up the collections
  TYPE numtab  IS TABLE OF NUMBER         INDEX BY PLS_INTEGER;
  TYPE nametab IS TABLE OF VARCHAR2(4000) INDEX BY PLS_INTEGER;
  TYPE tstab   IS TABLE OF TIMESTAMP      INDEX BY PLS_INTEGER;
  CURSOR test_c IS
    SELECT hi_card_nbr,hi_card_str ,hi_card_ts
    FROM data_test
  ;
  nbrs    numtab;
  txt     nametab;
  tstamps tstab;
  counter number;
  strt    number;
  fnsh    number;BEGIN
  plsql_memory.start_analysis;   -- initialize memory reporting
  strt := dbms_utility.get_time;  -- save starting time
  OPEN test_c;
  LOOP
    FETCH test_c BULK COLLECT INTO nbrs,txt,tstamps LIMIT 10000;
    EXIT WHEN nbrs.COUNT = 0;
    FOR i IN 1..nbrs.COUNT LOOP
      counter := counter + i;   -- do somthing with the data
    END LOOP;
  END LOOP;
  plsql_memory.show_memory_usage;
  CLOSE test_c;
  fnsh := dbms_utility.get_time;
  -- convert the centi-seconds from get_time to milliseconds
  DBMS_OUTPUT.PUT_LINE('Run time = '||(fnsh-strt)*10||' ms');
END;
/
```

Here are the results I got:

```
Change in UGA memory: 394272 (Current = 2459840)
Change in PGA memory: 1638400 (Current = 5807624)
Run time = 1530 ms
```

You can see that with a limit of 10,000 rows, I grew the PGA by 1,638,400 bytes. When I show the memory again after the PL/SQL block completes:

```
-- report on memory again, after the program completes
EXEC  plsql_memory.show_memory_usage;
```

You see that much (but not all) of this memory has been released:

```
Change in UGA memory: 0 (Current =2394352)
Change in PGA memory: -458752 (Current = 3907080)
```

I then removed the LIMIT clause from the preceding block and ran it again, so I read all the rows into the collection in one fetch, consuming more memory. How much?

```
Change in UGA memory: 0 (Current = 1366000)
Change in PGA memory: 18153472 (Current = 22519304)
```

As you can see, I used substantially more memory by leaving off LIMIT. So when it comes to production code, I strongly urge you to include a LIMIT clause.

Preservation of state

The database normally maintains the state of package-level constants, cursors, and variables in your UGA for as long as your session is running. Contrast this behavior with that of the variables instantiated in the declaration section of a standalone module. The scope of those variables is restricted to the module. When the module terminates, the memory and values associated with those variables are released. They are no more.

In addition to disconnecting, several other actions can cause a package to obliterate its state:

- Someone recompiles the program, or the database invalidates it, as discussed earlier.
- The DBMS_SESSION.RESET_PACKAGE built-in procedure executes in your session.
- You include the SERIALLY_REUSABLE pragma (see Chapter 18) in your program, which causes the database to put the private SQL area into the SGA for reuse by other sessions. State will be retained only for the duration of the call, rather than for the entire session.[5]
- You are using the web gateway in the default mode, which does not maintain persistent database sessions for each client.

5. By the way, instances with many concurrent sessions can use this technique to save a *lot* of memory.

- An error like ORA-04069 (cannot drop or replace a library with table dependent) is propagated to the client session.

Subject to these limitations, package data structures can act as "globals" within the PL/SQL environment; that is, they provide a way for different PL/SQL programs running in the same session to exchange data.

From an application design perspective, there are two types of global data, public and private:

Public
: A data structure declared in the specification of a package is a global public data structure. Any calling program or user with the EXECUTE privilege has access to the data. Programs can assign even meaningless values to package variables not marked CONSTANT. Public global data is the proverbial "loose cannon" of programming: convenient to declare but tempting to overuse, leading to a greater risk of unstructured code that is susceptible to ugly side effects.

  The specification of a module should give you all the information you need to call and use that module. If the program reads and/or writes global data structures, you cannot tell this from the module specification; you cannot be sure of what is happening in your application and which program changes what data. It is always preferable to pass data into and out of modules using parameters. That way, the reliance on those data structures is documented in the specification and can be accounted for by the developer. In my own code, I try to limit global public data to those values that can truly be made CONSTANT.

Private
: Not so problematic are global but private data structures (also called *package-level data*) that you might declare in the body of the package. Because it does not appear in the specification, this data cannot be referenced from outside the package—only from within the package, by other package elements.

Packaged data items are global only within a single database session or connection. Package data is not shared across sessions. If you need to share data between different database sessions, there are other tools at your disposal, including the DBMS_PIPE package, Oracle Advanced Queuing, and the UTL_TCP package... not to mention database tables!

What to Do If You Run Out of Memory

Let's say you're cruising along with your database running just fine, with lots of PL/SQL and SQL statements happily zipping by, and then it strikes: *ORA-04031: unable to allocate n bytes of shared memory*. This error is more common in shared server mode,

which caps a shared server's UGA memory. In dedicated server mode, the database can usually grab more virtual memory from the operating system, but you may still encounter the analogous error *ORA-04030: out of process memory when trying to allocate n bytes*. In fact, with a dedicated server, you can't grab more than roughly 4 gigabytes per session. With a shared server, however, you can set the large pool manually to whatever size you need.

There are several ways to correct this condition. If you're the application developer, you can attempt to reduce your use of shared memory. Steps you could take include (more or less in order):

1. Modify the code to ensure that the maximum number of SQL statements get shared.
2. Reduce the size or number of in-memory collections.
3. Reduce the amount of application code in memory.
4. Tune the database-wide settings and/or buy more server memory.

Steps 1 and 2 have already been covered; let's take a look at step 3. How can you assess the size of your source code once it's loaded into memory? And how can you reduce it?

Before running a PL/SQL program, the database must load all of its bytecode into memory. You can see how much space a program object actually occupies in the shared pool by having your DBA run the built-in procedure DBMS_SHARED_POOL.SIZES, which lists all objects over a given size.

Here is an example that looks at the memory required by objects in the shared pool immediately after database startup:[6]

```
SQL> SET SERVEROUTPUT ON SIZE 1000000
SQL> EXEC DBMS_SHARED_POOL.sizes(minsize => 125)

SIZE(K) KEPT   NAME
------- ------ ----------------------------------------------------------------
433        SYS.STANDARD                (PACKAGE)
364        SYS.DBMS_RCVMAN             (PACKAGE BODY)
249        SYSMAN.MGMT_JOB_ENGINE      (PACKAGE BODY)
224        SYS.DBMS_RCVMAN             (PACKAGE)
221        SYS.DBMS_STATS_INTERNAL     (PACKAGE)
220        SYS.DBMS_BACKUP_RESTORE     (PACKAGE)
125        MERGE INTO cache_stats_1$ D USING (select * from table(dbms_sta
ts_internal.format_cache_rows(CURSOR((select dataobj# o, st
atistic# stat, nvl(value, 0) val from gv$segstat where stat
istic# in (0, 3, 5) and obj# > 0 and inst_id = 1) union all
(select obj# o, 7 stat,nvl(sum(num_buf), 0) val from x$kcb
```

6. If you're wondering why the columns of data do not line up properly with their headings, it's probably because of the severe limitations of DBMS_OUTPUT. If you don't like it, write your own (grab the query from V$SQLAREA after running the package).

```
    oqh x where inst_id = 1 group by obj#) order by o))) wh
    (20B5C934,3478682418)     (CURSOR)
```

The minsize => 125 means "show only objects that are 125 KB or larger." The output shows that the package STANDARD occupies the most shared memory, 433 KB.[7]

Knowing the amount of memory your programs use is necessary, but not sufficient, information if you wish to reduce 4031 or 4030 errors; you must also know the size of the shared pool and how much of the shared pool is filled by "recreatable" objects—that is, objects that can be aged out of memory and loaded again later when needed. Some of this information is difficult to tease out of the database and may require knowledge of the mysterious X$ views. However, versions 9.2.0.5 and later will automatically generate a heap dump in your USER_DUMP_DEST directory every time you hit a 4031 error. See what you can discover from that, or just punt it over to Oracle Support. As a developer, you also need to figure out if your applications contain a large amount of unshared code that logically could be shared, because this can have a big impact on memory requirements.

Natively compiled PL/SQL programs are linked to shared library files, but the database still allocates some memory inside the database in order to run them. The same is true for external procedures. A privileged user can use operating system-level utilities such as *pmap* on Solaris to measure the amount of memory they require outside of the database.

Now, on to step 4: tune the database or buy more memory. A competent DBA (hey, don't look at me) will know how to tune the shared pool by adjusting parameters such as these:

SHARED_POOL_SIZE
: Bytes set aside for the shared pool.

DB_CACHE_SIZE
: Bytes of memory reserved to hold rows of data from the database (may need to be reduced in order to increase the size of the shared pool).

LARGE_POOL_SIZE
: Bytes of memory reserved for an optional region of memory that holds the UGA for shared server connections. This prevents the variable portion of the UGA from competing for use of the shared pool.

JAVA_POOL_SIZE
: Bytes used by the Java memory manager.

7. There is a bug in older versions of DBMS_SHARED_POOL.SIZES that results in the amount being overreported by about 2.3%. Oracle's package erroneously computed kilobytes by dividing bytes by 1,000 instead of by 1,024.

STREAMS_POOL_SIZE
: Bytes used by the Oracle Streams feature.

SGA_TARGET
: Set to a nonzero number of bytes, which indicates the size of the SGA from which the database will automatically allocate the cache and pools indicated previously.

PGA_AGGREGATE_TARGET
: Total amount of memory used by all of the server processes in the instance. Generally, it should be equal to the amount of server memory available to the database minus the SGA size.

PGA_AGGREGATE_LIMIT (new to Oracle Database 12c)
: Specifies a limit on the aggregate PGA memory consumed by the instance. When the limit is exceeded, the sessions using the most memory will have their calls aborted. Parallel queries will be treated as a unit. If the total PGA memory usage is still over the limit, then sessions using the most memory will be terminated. SYS processes and fatal background processes will not be subjected to any of the actions described here.

You can also ask the DBA to force the shared pool to hold a PL/SQL program unit, sequence, table, or cursor in memory with the DBMS_SHARED_POOL.KEEP procedure.[8] For example, the following block would require that the database keep the STANDARD package pinned in memory:

```
BEGIN
   DBMS_SHARED_POOL.KEEP('SYS.STANDARD');
END;
```

It can be especially beneficial to pin into memory large program units that are executed relatively *infrequently*. If you don't, the partially compiled code will likely be aged out of the shared pool. When called again, its loading could force many smaller objects out of the shared pool, and degrade performance.

It's probably obvious, but if you're encountering ORA-04031 errors resulting from a few users or applications, consider moving the offenders to dedicated server mode.

Native Compilation

In its default mode (interpreted), your code is partially compiled, but also interpreted at runtime. PL/SQL executes in a virtual machine, and it first translates (compiles) your code into virtual machine code, sometimes called *bytecode* or *mcode*. This is basically

8. In the fine print, Oracle says that it may obsolete this feature when it comes up with better memory management algorithms.

the same model that Java uses. When it is time to actually run your code, that bytecode is translated (interpreted) into system calls.

However, once you have the code running well, you can choose to improve the runtime efficiency of your PL/SQL programs by having the database perform the translation from bytecode to machine code early, at compile time (this is called *native mode*). This second half compilation results in machine code in a shared library. The database will dynamically load this compiled machine code at runtime.

When to Run in Interpreted Mode

So, if native mode is faster, why run in interpreted mode? Let's look at this question from the other end. The goal of native mode is fast execution speed. So, to get the fastest execution speed, you crank up the optimization level and try to do as much work ahead of execution time as possible (including early translation to machine code). When you are developing and unit testing your code, you need the capabilities of the debugger more than you need fast execution speed. If you need to step through your source code and step over subprogram calls in a debugger, you surely can't have the optimizing compiler rearranging your source code (optimization level 2) or inlining subprograms (optimization level 3). So, to debug, you have to revert to optimization level 0 or 1, at which point native mode is of questionable value. Therefore, I recommend running in interpreted mode in development environments.

When to Go Native

Native mode is built for speed. You run in native mode when you have your program debugged and want to make it go as fast as possible. Native compilation goes hand in hand with higher optimization levels. This configuration is usually for production and some test environments. With native mode, the compile times are slightly longer because you are doing more work in the compiler, but the execution times will be faster (or perhaps the same as) in interpreted mode.

Note that PL/SQL native compilation provides the smallest improvement in the performance of subprograms that spend most of their time running SQL. In addition, when you have natively compiled many subprograms and they are active simultaneously, the large amount of shared memory required might affect system performance. Oracle identifies 15,000 subprograms compiled natively as the boundary when you might start to see this effect.

Native Compilation and Database Release

How you set up native compilation and execution varies between major database releases. The details are spelled out in Chapter 20, but let's review a little native compilation history here:

Oracle9i Database
: Native compilation was introduced with Oracle9*i* Database. Everything worked quite well with native compilation in this release, as long as you weren't running Real Application Clusters (RAC) and didn't mind complicated backups. RAC databases were a problem (they weren't supported) and database backups needed to include the shared libraries, which Oracle Recovery Manager (RMAN) didn't capture.

Oracle Database 10g
: Native compilation was improved with Oracle Database 10*g*. RAC databases and shared servers were supported, but you needed the C compiler and copies of the shared libraries on each RAC node. Database backups were still an issue, though—they still needed to include the shared libraries, but RMAN didn't capture these shared libraries.

Oracle Database 11g
: Native compilation was again improved with Oracle Database 11*g*. You no longer need a C compiler, and the shared libraries are stored in the data dictionary, where every backup tool on the planet (well, those that work with Oracle databases, at least) locates them and backs them up. So, with no issues related to backups or managing shared library files, there is little to hold you back from going native on your production and test databases. Try it—you'll like it, and you won't go back.

What You Need to Know

So do you really *need* to remember everything in this chapter? I certainly hope not, because I can't even remember it in my day-to-day work. Your DBA, on the other hand, probably needs to know most of this stuff.

In addition to satisfying healthy curiosity, my goal in presenting this material was to help allay any misgivings programmers might have about the PL/SQL architecture. Whether or not you've ever had such concerns, there are a number of important points to remember about what goes on inside PL/SQL:

- To avoid compilation overhead, programs you plan to use more than a few times should be put in stored programs rather than stored in files as anonymous blocks.
- In addition to their unique ability to preserve state throughout a session, PL/SQL packages offer performance benefits. You should put most of your extensive application logic into package bodies.
- When upgrading Oracle versions, new features in the PL/SQL compiler warrant thorough application testing. In some (probably rare) cases when upgrading to Oracle Database 11*g* or later, slight changes in execution order resulting from freedoms exploited by the optimizing compiler could affect application results.

- While the Oracle database's automatic dependency management approach relieves a huge burden on developers, upgrading applications on a live production database should be undertaken with great care because of the need for object locking and package state reset.
- If you use signature-based remote dependency checking in remote procedure calls or with a loopback-link synonym as a way to avoid invalidations, you should institute (manual) procedures to eliminate the possibility of the signature check returning a false negative (which would cause a runtime error).
- Use definer rights to maximize performance and to help simplify the management and control of privileges on database tables. Use invoker rights only to address particular problems (for example, programs that use dynamic SQL and that create or destroy database objects).
- The database's sophisticated approaches aimed at minimizing the machine resources needed to run PL/SQL occasionally benefit from a little help from developers and DBAs—for example, by explicitly freeing unused user memory or pinning objects in memory.
- Where it makes sense to your application logic, use the cursor FOR loop idiom, rather than OPEN/FETCH/CLOSE, to take advantage of the automatic bulk-binding feature in Oracle Database 10*g* and later versions.
- When your program does need to open an explicit cursor in a PL/SQL program, be sure to close the cursor as soon as fetching is complete.
- Native compilation of PL/SQL may not offer any performance advantages for SQL-intensive applications, but it can significantly improve the performance of compute-intensive programs.
- Calling remote packages entails some special programming considerations if you want to take advantage of anything in the package other than procedures, functions, types, and subtypes.
- Use program variables in embedded static SQL statements in PL/SQL, and bind variables in dynamic SQL statements, to avoid subverting the database's cursor sharing features.