CHAPTER 13
Miscellaneous Datatypes

In this chapter, I'll explore all the native PL/SQL datatypes that have not yet been covered. These include the BOOLEAN, RAW, and UROWID/ROWID types, as well as the large object (LOB) family of types. I'll also discuss some useful predefined object types, including XMLType, which allows you to store XML data in a database column; the URI types, which allow you store Uniform Resource Identifier (URI) information; and the Any types, which allow you to store, well, just about anything.

The terminology for the LOB implementation changed in Oracle Database 11*g*. Oracle reengineered the implementation of LOBs using a technology called *SecureFiles*; the older, pre-Oracle Database 11*g* LOB technology is known as *BasicFiles*. In this chapter I'll also discuss SecureFiles and the performance benefits you can reap by using this updated technology.

The BOOLEAN Datatype

Boolean values and variables are very useful in PL/SQL. Because a Boolean variable can only be TRUE, FALSE, or NULL, you can use that variable to explain what is happening in your code. With Booleans you can write code that is easily readable because it is more English-like. You can replace a complicated Boolean expression involving many different variables and tests with a single Boolean variable that directly expresses the intention and meaning of the test.

Here is an example of an IF statement with a single Boolean variable (or function—you really can't tell the difference just by looking at this short bit of code):

```
IF report_requested
THEN
   print_report (report_id);
END IF;
```

The beauty of this technique is that it not only makes your code a bit more self-documenting, but also has the potential to insulate your code from future change. For example, consider the human interface that needs to precede the previous code fragment. How do you know that a report was requested? Perhaps you ask the user to answer a question with a Y or an N, or perhaps the user must select a checkbox or choose an option from a drop-down list. The point is that it doesn't matter. You can freely change the human interface of your code, and as long as that interface properly sets the report_requested Boolean variable, the actual reporting functionality will continue to work correctly.

While PL/SQL supports a Boolean datatype, the Oracle database does not. You can create and work with Boolean variables from PL/SQL, but you cannot create tables having Boolean columns.

The fact that Boolean variables can be NULL has implications for IF...THEN...ELSE statements. For example, look at the difference in behavior between the following two statements:

```
IF report_requested
THEN
   NULL; -- Executes if report_requested = TRUE
ELSE
   NULL; -- Executes if report_requested = FALSE or IS NULL
END IF;

IF NOT report_requested
THEN
   NULL; -- Executes if report_requested = FALSE
ELSE
   NULL; -- Executes if report_requested = TRUE or IS NULL
END IF;
```

If you need separate logic for each of the three possible cases, you can write a three-pronged IF statement as follows:

```
IF report_requested
THEN
   NULL;  -- Executes if report_requested = TRUE
ELSIF NOT report_requested
THEN
   NULL;  -- Executes if report_requested = FALSE
ELSE
   NULL;  -- Executes if report_requested IS NULL
END IF;
```

For more details on the effects of NULLs in IF statements, refer back to Chapter 4.

The RAW Datatype

The RAW datatype allows you to store and manipulate relatively small amounts of binary data. Unlike the case with VARCHAR2 and other character types, RAW data never undergoes any kind of character set conversion when traveling back and forth between your PL/SQL programs and the database. RAW variables are declared as follows:

```
variable_name RAW(maximum_size)
```

The value for *maximum_size* may range from 1 through 32767. Be aware, however, that while a RAW PL/SQL variable can hold up to 32,767 bytes of data, a RAW database column can hold only 2,000 bytes.

RAW is not a type that you will use or encounter very often. It's useful mainly when you need to deal with small amounts of binary data. When dealing with the large amounts of binary data found in images, sound files, and the like, you should look into using the BLOB (binary large object) type. BLOB is described later in this chapter (see "Working with LOBs" on page 422).

The UROWID and ROWID Datatypes

The UROWID and ROWID types allow you to work with database ROWIDs in your PL/SQL programs. A ROWID is a *row identifier*—a binary value that identifies the physical address for a row of data in a database table. A ROWID can be used to uniquely identify a row in a table, even if that table does not have a unique key. Two rows with identical column values will have different ROWIDs or UROWIDs.

Beware! ROWIDs in a table can change. In early Oracle releases (Oracle8 Database 8.0 and earlier) ROWIDs could not change during the life of a row. But starting with Oracle8*i* Database, new features were added that violated this old rule. If row movement is enabled on a regular (heap-organized) table or for any index-organized table, updates can cause a row's ROWID or UROWID to change. In addition, if someone alters the table to shrink, move, or perform some other operation on a row that will cause it to change from one physical data block to another, the ROWID will change.

With the caveat just noted, there can still sometimes be value in using ROWIDs. Referencing ROWIDs in SELECT, UPDATE, MERGE, and DELETE statements can lead to desirable improvements in processing speed, as access by ROWID is the fastest way to locate or retrieve a specific row in a table—faster than a search by primary key. Figure 13-1 contrasts the use of a ROWID in an UPDATE statement with the use of column values such as those for a primary key.

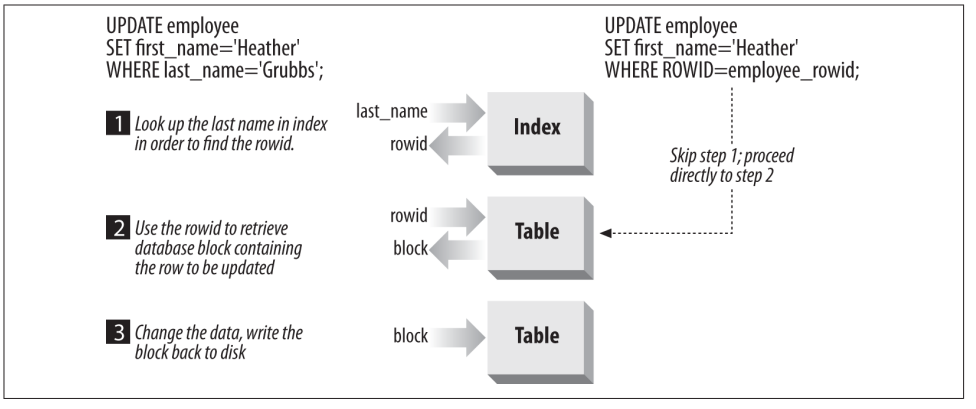

Figure 13-1. ROWIDs take you directly to rows in a table

Historically, the ROWID type came before UROWID. As Oracle added functionality such as index-organized tables (IOTs) and gateways to other types of databases, it developed new types of ROWIDs and hence had to develop a new datatype capable of holding them. Enter the UROWID datatype. The *U* in UROWID stands for *universal*, and a UROWID variable can contain any type of ROWID from any type of table.

I recommend the use of UROWID for all new development involving ROWIDs. The ROWID type provides backward compatibility but can't accommodate all types of ROWIDs now encountered in an Oracle database. UROWID is safer because it accommodates any type of ROWID while still providing the desired *access by rowid* execution plan.

Getting ROWIDs

You can get the ROWID for a given table row by including the keyword ROWID in your select list. For example:

```
DECLARE
   employee_rowid UROWID;
   employee_salary NUMBER;
BEGIN
   -- Retrieve employee information that we might want to modify
   SELECT rowid, salary INTO employee_rowid, employee_salary
     FROM employees
    WHERE last_name='Grubbs' AND first_name='John';
END;
```

Oracle calls the ROWID a *pseudocolumn* because the ROWID value is not stored in the same sense that other column values are, yet you can refer to the ROWID as if it were

a column. A ROWID is more akin to a pointer—it holds the physical address of a row in a table.

Using ROWIDs

The main use of ROWIDs is in repeating access to a given database row. This use is particularly beneficial when accessing the row is costly or frequent. Recall the example from the previous section in which I retrieved the salary for a specific employee. What if I later want to modify that salary? One solution would be to issue an UPDATE statement with the same WHERE clause as the one I used in my original SELECT:

```
DECLARE
   employee_rowid UROWID;
   employee_salary NUMBER;
BEGIN
   -- Retrieve employee information that we might want to modify
   SELECT rowid, salary INTO employee_rowid, employee_salary
     FROM employees
    WHERE last_name='Grubbs' AND first_name='John';

   /* Do a bunch of processing to compute a new salary */

   UPDATE employees
      SET salary = employee_salary
    WHERE last_name='Grubbs' AND first_name='John';
END;
```

While this code will certainly work, it has the disadvantage of having to repeat the same access path for the UPDATE as was used for the SELECT. Most likely, one or more indexes were accessed in order to find the employee row in question. But those indexes were just accessed for the SELECT statement, so why go through all the work of looking up the same ROWID twice? Internally, the purpose of accessing the index was to obtain the ROWID so that the row could be accessed directly. By including ROWID in my SELECT statement, I can simply supply that ROWID to the UPDATE statement, bypassing the index lookup:

```
DECLARE
   employee_rowid UROWID;
   employee_salary NUMBER;
BEGIN
   -- Retrieve employee information that we might want to modify
   SELECT rowid, salary INTO employee_rowid, employee_salary
     FROM employees
    WHERE last_name='Grubbs' AND first_name='John';

   /* Do a bunch of processing to compute a new salary */

   UPDATE employees
      SET salary = employee_salary
```

```
    WHERE rowid = employee_rowid;
END;
```

Recall my caveat about ROWIDs changing. If in my multiuser system the ROWID for the John Grubbs row in the employees table changes between the SELECT and the UPDATE, my code will not execute as intended. Why is that? Well, enabling row movement on a regular heap-organized table can allow a row's ROWID in that table to change. Row movement may be enabled because the DBA wants to do online table reorganizations, or the table may be partitioned, and row movement will allow a row to migrate from one partition to another during an update.

Often, a better way to achieve the same effect as using ROWID in an UPDATE or DELETE statement is to use an explicit cursor to retrieve data, and then use the WHERE CURRENT OF CURSOR clause to modify or delete it. See Chapter 15 for detailed information on this technique.

Using ROWIDs is a powerful technique to improve the performance of your PL/SQL programs because they cut through to the physical management layer of the database. Good application programs don't usually get involved in how the data is physically managed. Instead, they let the database and administrative programs work with the physical management and are themselves restricted to logical management of data. Therefore, I don't generally recommend using ROWIDs in your application programs.

The LOB Datatypes

Oracle and PL/SQL support several variations of large object datatypes. LOBs can store large amounts—from 8 to 128 terabytes—of binary data (such as images) or character text data.

Through Oracle9*i* Database Release 2, LOBs could store only up to 4 gigabytes. Starting with Oracle Database 10*g*, the limit was increased to a value between 8 and 128 terabytes that is dependent upon your database block size.

Within PL/SQL you can declare LOB variables of the following datatypes:

BFILE
: Binary file. Declares a variable that holds a file locator pointing to an operating system file outside the database. The database treats the data in the file as binary data.

BLOB
: Binary large object. Declares a variable that holds a LOB locator pointing to a large binary object stored inside the database.

CLOB
: Character large object. Declares a variable that holds a LOB locator pointing to a large block of character data in the database character set, stored inside the database.

NCLOB
: National Language Support (NLS) character large object. Declares a variable that holds a LOB locator pointing to a large block of character data in the national character set, stored inside the database.

LOBs can be categorized as *internal* or *external*. Internal LOBs (BLOBs, CLOBs, and NCLOBs) are stored in the database and can participate in transactions in the database server. External LOBs (BFILEs) represent binary data stored in operating system files outside the database tablespaces. External LOBs cannot participate in transactions. In other words, you cannot commit or roll back changes to a BFILE. Instead, you must rely on the underlying filesystem for data integrity. Likewise, the database's read consistency model does not extend to BFILEs. Repeated reads of a BFILE may not give the same results, unlike with internal LOBs, which do follow the database read consistency model.

LONG and LONG RAW

If you've been around Oracle for a few years, you've probably noticed that so far I've omitted any discussion of two datatypes: LONG and LONG RAW. This is intentional. In the database, LONG and LONG RAW allow you to store large amounts (up to 2 gigabytes) of character and binary data, respectively. The maximum lengths of the PL/SQL types, however, are much shorter: only 32,760 bytes, which is less than the 32,767 bytes supported by VARCHAR2 and RAW. Given this rather odd length limitation, I recommend using VARCHAR2 and RAW instead of LONG and LONG RAW in your PL/SQL programs.

If you're retrieving LONG and LONG RAW columns that may contain more than 32,767 bytes of data, you won't be able to store the returned values in VARCHAR2 or RAW variables. This is an unfortunate restriction and a good reason to avoid LONG and LONG RAW to begin with.

LONG and LONG RAW are obsolete types, maintained only for backward compatibility. Oracle doesn't recommend their use, and neither do I. For new applications where you have a choice, use CLOB and BLOB instead. For existing applications, Oracle's *SecureFiles and Large Objects Developer's Guide* provides guidance for migrating existing data from LONG to LOB columns.

Working with LOBs

The topic of working with large objects is, well, large, and I can't begin to cover every aspect of LOB programming in this chapter. What I can and will do, however, is provide you with a good introduction to the topic of LOB programming aimed especially at PL/SQL developers. I'll discuss some of the issues to be aware of and show examples of fundamental LOB operations. All of this, I hope, will provide you with a good foundation for your future LOB programming endeavors.

Before we get into the meat of this section, please note that all LOB examples are based on the following table definition (which can be found in the *ch13_code.sql* file on the book's website):

```
TABLE waterfalls (
   falls_name VARCHAR2(80),
   falls_photo BLOB,
   falls_directions CLOB,
   falls_description NCLOB,
   falls_web_page BFILE)
```

This table contains rows about waterfalls located in Michigan's Upper Peninsula. Figure 13-2 shows the Dryer Hose, a falls near Munising frequented by ice climbers in its frozen state.

The table implements one column for each of the four LOB types. Photos consist of large amounts of binary data, so the falls_photo column is defined as a BLOB. Directions and descriptions are text, so those columns are CLOB and NCLOB, respectively. Normally, you'd use either CLOB or NCLOB for both, but I wanted to provide an example that used each LOB type. Finally, the master copy of the web page for each waterfall is stored in an HTML file outside the database. I use a BFILE column to point to that HTML file. I'll use these columns in our examples to demonstrate various facets of working with LOB data in PL/SQL programs.

Figure 13-2. The Dryer Hose in Munising, Michigan

In my discussion of large objects, I'll frequently use the acronym LOB to refer to CLOBs, BLOBs, NCLOBs, and BFILEs in general. I'll use specific type names only when discussing something specific to a type.

Understanding LOB Locators

Fundamental to working with LOBs is the concept of a *LOB locator*. A LOB locator is a pointer to large object data in a database. Let's look at what happens when you select a BLOB column into a BLOB PL/SQL variable:

```
DECLARE
   photo BLOB;
BEGIN
```

```
SELECT falls_photo
  INTO photo
  FROM waterfalls
 WHERE falls_name='Dryer Hose';
```

What, exactly, is in the photo variable after the SELECT statement executes? Is the photo itself retrieved? No. Only a pointer to the photo is retrieved. You end up with the situation shown in Figure 13-3.

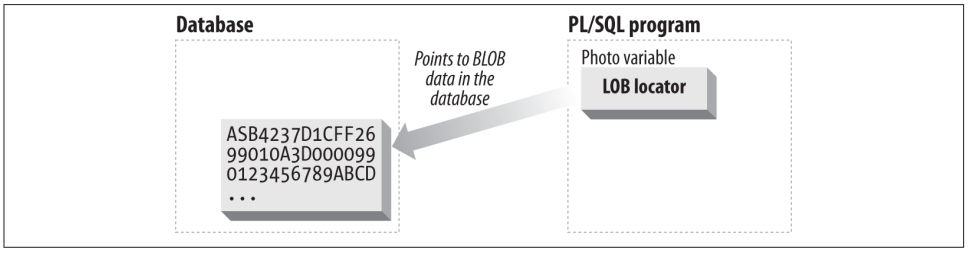

Figure 13-3. A LOB locator points to its associated large object data within the database

This is different from the way in which other datatypes work. Database LOB columns store LOB locators, and those locators point to the real data stored in a LOB segment elsewhere in the database. Likewise, PL/SQL LOB variables hold those same LOB locators, which point to LOB data within the database. To work with LOB data, you first retrieve a LOB locator, and you then use a built-in package named DBMS_LOB to retrieve and/or modify the actual LOB data. For example, to retrieve the binary photo data from the falls_photo BLOB column used in the previous example, you would go through the following steps:

1. Issue a SELECT statement to retrieve the LOB locator for the photo you wish to display.
2. Open the LOB via a call to DBMS_LOB.OPEN.
3. Make a call to DBMS_LOB.GETCHUNKSIZE to get the optimal chunk size to use when reading (and writing) the LOB's value.
4. Make a call to DBMS_LOB.GETLENGTH to get the number of bytes or characters in the LOB value.
5. Make multiple calls to DBMS_LOB.READ in order to retrieve the LOB data.
6. Close the LOB.

Not all of these steps are necessary, and don't worry if you don't understand them fully right now. I'll explain all the steps and operations shortly.

The use of locators might initially appear clumsy. It's a good approach, though, because it obviates the need to return all the data for a given LOB each time that you fetch a row from a table. Imagine how long a fetch would take if up to 128 terabytes of LOB data needed to be transferred. Imagine the waste if you had to access only a small fraction of that data. With the Oracle database's approach, you fetch locators (a quick operation), and then you retrieve only the LOB data that you need. In addition, LOBs are not cached in the buffer cache by default, and LOBs do not generate undo like normal data. (LOBs do generate redo like normal data, unless you specify the NOLOGGING option.) So, loading 50 gigabytes of LOB data will not flush your buffer cache or flood your undo tablespace and degrade overall performance. This separate cache and undo management of LOBs got even better with SecureFiles in Oracle Database 11*g*... but more on that later.

Oracle's LOB Documentation

If you are working with LOBs, I strongly recommend that you familiarize yourself with the following portions of Oracle's documentation set:

SecureFiles and Large Objects Developer's Guide
: Oracle Database 11*g* and later guide to LOB programming.

Application Developer's Guide—Large Objects
: Oracle Database 10*g* and earlier guide to LOB programming.

PL/SQL Packages and Types Reference
: See the chapter on the DBMS_LOB package.

SQL Reference
: The "Datatypes" section in Chapter 2, "Basic Elements of Oracle SQL," contains important information about LOBs.

This is not an exhaustive list of LOB documentation, but you'll find all the essential information in these sources.

Empty Versus NULL LOBs

Now that you understand the distinction between a LOB locator and the value to which it points, you need to wrap your mind around another key concept: the *empty LOB*. An empty LOB is what you have when a LOB locator doesn't point to any LOB data. This is not the same as a NULL LOB, which is a LOB column (or variable) that doesn't hold a LOB locator. Clear as mud, right? Let's look at some example code:

```
DECLARE
  directions CLOB;BEGIN
  IF directions IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('directions is NULL');
```

```
  ELSE
    DBMS_OUTPUT.PUT_LINE('directions is not NULL');
  END IF;END;

directions is NULL
```

Here I have declared a CLOB variable, which is atomically NULL because I haven't yet assigned it a value. You're used to this behavior, right? It's the same with any other datatype: declare a variable without assigning a value, and comparisons to NULL—such as "*variable* IS NULL"—evaluate to TRUE. In this regard, a LOB is similar to an object in that it must be initialized before data can be added to it. See Chapter 26 for more information on objects.

Let's press ahead with the example and initialize the LOB. The following code uses a call to EMPTY_CLOB to initialize (but not populate) the LOB variable:

```
DECLARE
   directions CLOB;
BEGIN
   IF directions IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('at first directions is NULL');
   ELSE
      DBMS_OUTPUT.PUT_LINE('at first directions is not NULL');
   END IF;
   DBMS_OUTPUT.PUT_LINE('Length = '
                        || DBMS_LOB.GETLENGTH(directions));

   -- initialize the LOB variable
   directions := EMPTY_CLOB();

   IF directions IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('after initializing, directions is NULL');
   ELSE
      DBMS_OUTPUT.PUT_LINE('after initializing, directions is not NULL');
   END IF;
   DBMS_OUTPUT.PUT_LINE('Length = '
                        || DBMS_LOB.GETLENGTH(directions));
END;
```

The output is:

```
at first directions is NULL
Length =
after initializing, directions is not NULL
Length = 0
```

You can see that at first the CLOB variable is atomically NULL. It comes as no surprise, then, that the length of the NULL LOB is also NULL. After I initialize the CLOB variable with the built-in function EMPTY_CLOB, my variable is no longer NULL because it contains a value: the locator. DBMS_LOB.GETLENGTH shows that while initialized (NOT NULL), the CLOB is empty. This difference is important to understand because

the way in which you test for the presence or absence of data is more complicated for a LOB than it is for scalar datatypes.

A simple IS NULL test suffices for traditional scalar datatypes:

```
IF some_number IS NULL THEN
   -- You know there is no data
```

If an IS NULL test on a NUMBER or a VARCHAR2 (or any other scalar type) returns TRUE, you know that the variable holds no data. With LOBs, however, you not only need to check for nullity (no locator), but you also need to check the length:

```
IF some_clob IS NULL THEN
   -- There is no data
ELSIF DBMS_LOB.GETLENGTH(some_clob) = 0 THEN
   -- There is no data
ELSE
   -- Only now is there data
END IF;
```

As illustrated in this example, you can't check the length of a LOB without first having a locator. Thus, to determine whether a LOB holds data, you must first check for the presence of a locator using an IS NULL test and then check for a nonzero length, or perform both checks together like this:

```
IF NVL(DBMS_LOB.GETLENGTH(some_clob),0) = 0 THEN
   -- There is no data
ELSE
   -- There is data
END IF;
```

The bottom line is that you need to check for two conditions, not just one.

When working with BLOBs, use EMPTY_BLOB() to create an empty BLOB. Use EMPTY_CLOB() for CLOBs and NCLOBs.

Writing into a LOB

Once you have a valid LOB locator, you can write data into that LOB using one of these procedures from the built-in DBMS_LOB package:

DBMS_LOB.WRITE
: Allows you to write data randomly into a LOB

DBMS_LOB.WRITEAPPEND
: Allows you to append data to the end of a LOB

Following is an extension of the previous examples in this chapter. It begins by creating a LOB locator for the directions column in the waterfalls table. After creating the locator, I use DBMS_LOB.WRITE to begin writing directions to Munising Falls into the CLOB column. I then use DBMS_LOB.WRITEAPPEND to finish the job:

```
/* File on web: munising_falls_01.sql */
DECLARE
   directions CLOB;
   amount BINARY_INTEGER;
   offset INTEGER;
   first_direction VARCHAR2(100);
   more_directions VARCHAR2(500);
BEGIN
   -- Delete any existing rows for 'Munising Falls' so that this
   -- example can be executed multiple times
   DELETE
     FROM waterfalls
    WHERE falls_name='Munising Falls';

   -- Insert a new row using EMPTY_CLOB() to create a LOB locator
   INSERT INTO waterfalls
             (falls_name,falls_directions)
      VALUES ('Munising  Falls',EMPTY_CLOB());

   -- Retrieve the LOB locator created by the previous INSERT statement
   SELECT falls_directions
     INTO directions
     FROM waterfalls
    WHERE falls_name='Munising Falls';

   -- Open the LOB; not strictly necessary, but best to open/close LOBs.
   DBMS_LOB.OPEN(directions, DBMS_LOB.LOB_READWRITE);

   -- Use DBMS_LOB.WRITE to begin
   first_direction := 'Follow I-75 across the Mackinac Bridge.';
   amount := LENGTH(first_direction);  -- number of characters to write
   offset := 1; -- begin writing to the first character of the CLOB
   DBMS_LOB.WRITE(directions, amount, offset, first_direction);

   -- Add some more directions using DBMS_LOB.WRITEAPPEND
   more_directions := ' Take US-2 west from St. Ignace to Blaney Park.'
                   || ' Turn north on M-77 and drive to Seney.'
                   || ' From Seney, take M-28 west to Munising.';
   DBMS_LOB.WRITEAPPEND(directions,
                        LENGTH(more_directions), more_directions);

   -- Add yet more directions
   more_directions := ' In front of the paper mill, turn right on H-58.'
                   || ' Follow H-58 to Washington Street. Veer left onto'
                   || ' Washington Street. You''ll find the Munising'
                   || ' Falls visitor center across from the hospital at'
                   || ' the point where Washington Street becomes'
```

```
                    || ' Sand Point Road.';
    DBMS_LOB.WRITEAPPEND(directions,
                         LENGTH(more_directions), more_directions);

    -- Close the LOB, and we are done
    DBMS_LOB.CLOSE(directions);
END;
```

In this example, I used both WRITE and WRITEAPPEND solely to demonstrate the use of both procedures. Because my LOB had no data to begin with, I could have done all the work using only WRITE. Notice that I opened and closed the LOB; while this is not strictly necessary, it is a good idea, especially if you are using Oracle Text. Otherwise, any Oracle Text domain- and function-based indexes will be updated with each WRITE or WRITEAPPEND call, rather than being updated once when you call CLOSE.

In the section on BFILEs, I show how to read LOB data directly from an external operating system file.

When you're writing to a LOB, as I have done here, there is no need to update the LOB column in the table. That's because the LOB locator does not change. I did not change the contents of falls_directions (the LOB locator). Rather, I added data to the LOB to which the locator pointed.

LOB updates take place within the context of a transaction. I did not COMMIT in my example code. You should issue a COMMIT after executing the PL/SQL block if you want the Munising Falls directions to remain permanently in your database. If you issue a ROLLBACK after executing the PL/SQL block, all the work done by this block will be undone.

My example writes to a CLOB column. You write BLOB data in the same manner, except that your inputs to WRITE and WRITEAPPEND should be of the RAW type instead of the VARCHAR2 type.

The following SQL*Plus example shows one way you can see the data just inserted by my example. The next section will show you how to retrieve the data using the various DBMS_LOB procedures.

```
SQL> SET LONG 2000
SQL> COLUMN falls_directions WORD_WRAPPED FORMAT A70
SQL> SELECT falls_directions
  2  FROM waterfalls
  3  WHERE falls_name='Munising Falls';
  4  /

FALLS_DIRECTIONS
```

```
--------------------------------------------------------------------
Follow I-75 across the Mackinac Bridge. Take US-2 west from St. Ignace
to Blaney Park. Turn north on M-77 and drive to Seney. From Seney,
take M-28 west to Munising. In front of the paper mill, turn right on
H-58. Follow H-58 to   Washington Street. Veer left onto Washington
Street. You'll find the Munising Falls visitor center across from the
hospital at the point where Washington Street becomes Sand Point Road.
```

Reading from a LOB

To retrieve data from a LOB, you use the DBMS_LOB.READ procedure. First, of course, you must retrieve the LOB locator. When reading from a CLOB, you specify an offset in terms of characters. Reading begins at the offset that you specify, and the first character of a CLOB is always number 1. When you are working with BLOBs, offsets are in terms of bytes. Note that when you are calling DBMS_LOB.READ, you must specify the number of characters (or bytes) that you wish to read. Given that LOBs are large, it's reasonable to plan on doing more than one read to get at all the data.

The following example retrieves and displays the directions to Munising Falls. I have carefully chosen the number of characters to read both to accommodate DBMS_OUTPUT's line-length restriction and to ensure a nice-looking line break in the final output:

```
/* File on web: munising_falls_02.sql */
DECLARE
   directions CLOB;
   directions_1 VARCHAR2(300);
   directions_2 VARCHAR2(300);
   chars_read_1 BINARY_INTEGER;
   chars_read_2 BINARY_INTEGER;
   offset INTEGER;
BEGIN
   -- Retrieve the LOB locator inserted previously
   SELECT falls_directions
     INTO directions
     FROM waterfalls
    WHERE falls_name='Munising Falls';

   -- Begin reading with the first character
   offset := 1;

   -- Attempt to read 229 characters of directions; chars_read_1 will
   -- be updated with the actual number of characters read
   chars_read_1 := 229;
   DBMS_LOB.READ(directions, chars_read_1, offset, directions_1);

   -- If we read 229 characters, update the offset and try to
   -- read 255 more
   IF chars_read_1 = 229 THEN
      offset := offset + chars_read_1;
      chars_read_2 := 255;
      DBMS_LOB.READ(directions, chars_read_2, offset, directions_2);
```

```
    ELSE
       chars_read_2 := 0;
       directions_2 := '';
    END IF;

    -- Display the total number of characters read
    DBMS_OUTPUT.PUT_LINE('Characters read = ' ||
                         TO_CHAR(chars_read_1+chars_read_2));

    -- Display the directions
    DBMS_OUTPUT.PUT_LINE(directions_1);
    DBMS_OUTPUT.PUT_LINE(directions_2);
END;
```

The output from this code is as follows:

```
Characters read = 414
Follow I-75 across the Mackinac Bridge. Take US-2 west from St. Ignace to Blaney
Park. Turn north on M-77 and drive to Seney. From Seney, take M-28 west to
Munising. In front of the paper mill, turn right on H-58. Follow H-58 to
Washington Street. Veer left onto Washington Street. You'll find the Munising
Falls visitor center across from the hospital at the point where Washington
Street becomes Sand Point Road.
```

The chars_read_1 (amount to read) parameter, which is the second parameter you pass to DBMS_LOB.READ, is an IN OUT parameter, and DBMS_LOB.READ will update it to reflect the number of characters (or bytes) actually read. You'll know you've reached the end of a LOB when the number of characters or bytes read is less than the number you requested. It seems to me a bit inconvenient that the offset is not updated in the same manner. When reading several sequential portions of a LOB, you must update the offset each time based on the number of characters or bytes just read.

You can use DBMS_LOB.GET_LENGTH (*lob_locator*) to retrieve the length of a LOB. The length is returned as a number of bytes for BLOBs and BFILEs, and as a number of characters for CLOBs.

BFILEs Are Different

As mentioned earlier, the BLOB, CLOB, and NCLOB types represent *internal LOBs*, meaning that they are stored within the database. A BFILE, on the other hand, is an *external LOB* type. BFILEs are very different from internal LOBs in three important ways:

- The value of a BFILE is stored in an operating system file, not within the database.

- BFILEs do not participate in transactions (i.e., changes to a BFILE cannot be rolled back or committed). However, changes to a BFILE locator can be rolled back and committed.
- From within PL/SQL and the Oracle database in general, you can only read BFILEs. The database does not allow you to write BFILE data. You must generate the external files to which BFILE locators point completely outside of the database.

When you work with BFILEs in PL/SQL, you still work with LOB locators. In the case of a BFILE, however, the locator simply points to a file stored on the server. For this reason, two different rows in a database table can have a BFILE column that points to the same file.

A BFILE locator is composed of a directory alias and a filename. You use the BFILENAME function, which I will describe shortly, to return a locator based on those two pieces of information. A *directory alias* is simply a database-specific name for an operating system directory. Directory aliases allow your PL/SQL programs to work with directories in an operating system–independent manner. If you have the CREATE ANY DIRECTORY privilege, you can create a directory alias (the directory must already exist in the filesystem) and grant access to it as follows:

```
CREATE DIRECTORY bfile_data AS 'c:\PLSQL Book\Ch13_Misc_Datatypes\'

GRANT READ ON DIRECTORY bfile_data TO gennick;
```

Creating directory aliases and dealing with access to those aliases are more database administration functions than PL/SQL issues, so I won't go too deeply into those topics. The examples here should be enough to get you started. To learn more about directory aliases, talk to your DBA or read the section in Oracle's *SQL Reference* on the CREATE DIRECTORY command. To see directories that you have access to, query the ALL_DIRECTORIES view.

Creating a BFILE locator

BFILE locators are trivial to create; you simply invoke the BFILENAME function and pass it a directory alias and a filename. In the following example, I create a BFILE locator for the HTML file containing the Tannery Falls web page. I then store that locator into the waterfalls table:

```
DECLARE
   web_page BFILE;
BEGIN
   -- Delete row for Tannery Falls so this example can
   -- be executed multiple times
   DELETE FROM waterfalls WHERE falls_name='Tannery Falls';

   -- Invoke BFILENAME to create a BFILE locator
   web_page := BFILENMAE('BFILE_DATA','TanneryFalls.htm');
```

```
    -- Save our new locator in the waterfalls table
    INSERT INTO waterfalls (falls_name, falls_web_page)
       VALUES ('Tannery  Falls',web_page);
END;
```

A BFILE locator is simply a combination of directory alias and filename. The actual file and directory don't even need to exist. That is, the database allows you to create directory aliases for directories that do not yet exist, and BFILENAME allows you to create BFILE locators for files that do not yet exist. There are times when it's convenient to do these things.

The directory name you specify in calls to BFILENAME is case-sensitive, and its case must match that shown by the ALL_DIRECTORIES data dictionary view. I first used lowercase bfile_data in my example, only to be greatly frustrated by errors when I tried to access my external BFILE data (as shown in the next section). In most cases, you'll want to use all uppercase for the directory name in a call to BFILENAME.

Accessing BFILEs

Once you have a BFILE locator, you can access the data from an external file in much the same manner as you would access a BLOB. The following example retrieves the first 60 bytes of HTML from the Tannery Falls web page. The results, which are of the RAW type, are cast to a character string using the built-in UTL_RAW.CAST_TO_VARCHAR2 function:

```
DECLARE
   web_page BFILE;
   html RAW(60);
   amount BINARY_INTEGER := 60;
   offset INTEGER := 1;
BEGIN
   -- Retrieve the LOB locator for the web page
   SELECT falls_web_page
     INTO web_page
     FROM waterfalls
    WHERE falls_name='Tannery Falls';

   -- Open the locator, read 60 bytes, and close the locator
   DBMS_LOB.OPEN(web_page);
   DBMS_LOB.READ(web_page, amount, offset, html);
   DBMS_LOB.CLOSE(web_page);

   -- Uncomment following line to display results in hex
   -- DBMS_OUTPUT.PUT_LINE(RAWTOHEX(html));

   -- Cast RAW results to a character string we can read
```

```
   DBMS_OUTPUT.PUT_LINE(UTL_RAW.CAST_TO_VARCHAR2(html));
END;
```

The output from this code will appear as follows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
```

The maximum number of BFILEs that can be opened within a session is established by the database initialization parameter SESSION_MAX_OPEN_FILES. This parameter defines an upper limit on the number of files that can be opened simultaneously in a session (not just BFILEs, but all kinds of files, including those opened using the UTL_FILE package).

Remember that from within the Oracle database, you can only read BFILEs. The BFILE type is ideal when you want to access binary data, such as a collection of images that is generated outside the database environment. For example, you might upload a collection of images from a digital camera to your server and create a BFILE locator to point to each of those images. You could then access the images from your PL/SQL programs.

Using BFILEs to load LOB columns

In addition to allowing you to access binary file data created outside the Oracle database environment, BFILEs provide a convenient means to load data from external files into internal LOB columns. Up through Oracle9*i* Database Release 1, you could use the DBMS_LOB.LOADFROMFILE function to read binary data from a BFILE and store it into a BLOB column. Oracle9*i* Database Release 2 introduced the following, much improved, functions:

DBMS_LOB.LOADCLOBFROMFILE
: Loads CLOBs from BFILEs. Takes care of any needed character set translation.

DBMS_LOB.LOADBLOBFROMFILE
: Loads BLOBs from BFILEs. Does the same thing as DBMS_LOB.LOADFROMFILE, but with an interface that is consistent with that of LOADCLOBFROMFILE.

Imagine that I had directions to Tannery Falls in an external text file named *Tannery Falls.directions* in a directory pointed to by the BFILE_DATA directory alias. The following example shows how I could use DBMS_LOB.LOADCLOBFROMFILE to load the directions into the falls_directions CLOB column in the waterfalls table:

```
/* File on web: munising_falls_03.sql */
DECLARE
   Tannery_Falls_Directions BFILE
      := BFILENAME('BFILE_DATA','TanneryFalls.directions');
   directions CLOB;
   destination_offset INTEGER := 1;
   source_offset INTEGER := 1;
   language_context INTEGER := DBMS_LOB.default_lang_ctx;
   warning_message INTEGER;
BEGIN
```

```
    -- Delete row for Tannery Falls, so this example
    -- can run multiple times
    DELETE FROM waterfalls WHERE falls_name='Tannery Falls';

    -- Insert a new row using EMPTY_CLOB() to create a LOB locator
    INSERT INTO waterfalls
              (falls_name,falls_directions)
       VALUES ('Tannery Falls',EMPTY_CLOB());

    -- Retrieve the LOB locator created by the previous INSERT statement
    SELECT falls_directions
      INTO directions
      FROM waterfalls
     WHERE falls_name='Tannery Falls';

    -- Open the target CLOB and the source BFILE
    DBMS_LOB.OPEN(directions, DBMS_LOB.LOB_READWRITE);
    DBMS_LOB.OPEN(Tannery_Falls_Directions);

    -- Load the contents of the BFILE into the CLOB column
    DBMS_LOB.LOADCLOBFROMFILE
                (directions, Tannery_Falls_Directions,
                             DBMS_LOB.LOBMAXSIZE,
                             destination_offset, source_offset,
                             NLS_CHARSET_ID('US7ASCII'),
                             language_context, warning_message);

    -- Check for the only possible warning message
    IF warning_message = DBMS_LOB.WARN_INCONVERTIBLE_CHAR THEN
         DBMS_OUTPUT.PUT_LINE (
            'Warning! Some characters couldn''t be converted.');
    END IF;

    -- Close both LOBs
    DBMS_LOB.CLOSE(directions);
    DBMS_LOB.CLOSE(Tannery_Falls_Directions);
END;
```

The real work in this snippet of code is done by the call to DBMS_LOB.LOADCLOB-FROMFILE. That procedure reads data from the external file, performs any character set translation that's necessary, and writes the data to the CLOB column. I use the DBMS_LOB.LOBMAXSIZE constant to specify the amount of data to load. I really want *all* the data from the external file, and DBMS_LOB.LOBMAXSIZE is as much as a CLOB will hold.

The destination and source offsets both begin at 1. I want to begin reading with the first character in the BFILE, and I want to begin writing to the first character of the CLOB. To facilitate multiple sequential calls to LOADCLOBFROMFILE, the procedure will update both these offsets to point one character past the most recently read character.

Because they are IN OUT parameters, I must use variables and not constants in my procedure call.

The call to NLS_CHARSET_ID returns the character set ID number for the character set used by the external file. The LOADCLOBFROMFILE procedure will then convert the data being loaded from that character set to the database character set. The only possible warning message LOADCLOBFROMFILE can return is that some characters were not convertible from the source to the target character set. I check for this warning in the IF statement following the load.

A warning is not the same as a PL/SQL error; the load will still have occurred, just as I requested.

The following SQL*Plus example shows the data loaded from my external file using LOADCLOBFROMFILE:

```
SQL> SET LONG 2000
SQL> COLUMN falls_directions WORD_WRAPPED FORMAT A70
SQL> SELECT falls_directions
  2  FROM waterfalls
  3  WHERE falls_name='Tannery Falls';
  4 /

FALLS_DIRECTIONS
----------------------------------------------------------------------
From downtown Munising, take Munising Avenue east. It will
shortly turn into H-58. Watch for Washington Street veering
off to your left. At that intersection you'll see a wooden
stairway going into the woods on your right. Go up that
stairway and follow the trail to the falls. Do not park
on H-58! You'll get a ticket. You can park on Nestor Street,
which is just uphill from the stairway.
```

SecureFiles Versus BasicFiles

SecureFiles, introduced in Oracle Database 11*g*, offer many improvements over the older implementation of LOBs, which are now known as BasicFiles. These improvements are internal and largely transparent to us as programmers—the same keywords, syntax, and programming steps are used. The internal implementation of SecureFiles involves improvements to many aspects of managing LOBs, including disk format, caching, locking, redo, and space management algorithms. This updated technology significantly improves performance and allows LOBs to be deduplicated, compressed, and encrypted using simple parameter settings. In addition, a new logging level, FILESYSTEM_LIKE_LOGGING, has been introduced to augment the existing LOGGING

and NOLOGGING options. This new logging level logs only metadata changes, much as a journaled filesystem would do.

The SecureFiles features improve the performance of LOBs substantially. Oracle testing reports 200% to 900% improvements. In a simple test loading PDF files on a Microsoft Windows server, I experienced a decrease in load times of 80% to 90%—from 169 seconds down to 20 to 30 seconds (depending on the options used and how many times I ran the load). I noted more moderate improvements on x86 Linux. Your experience may differ, but expect improvements!

To use SecureFiles with your LOBs, your database initialization parameter DB_SECUREFILE must be set to PERMITTED (the default setting). In addition, the tablespace that will store the LOBs must use Automatic Segment Space Management (ASSM). If you are not sure about your database's settings, ask your DBA. If you are the DBA, check in V$PARAMETER for initialization parameters and in DBA_TABLESPACES for Segment Space Management settings.

In Oracle Database 12*c*, SecureFiles became the default storage mechanism; if you are using Oracle Database 11*g* (Release 1 or Release 2), the default storage is determined by the DB_SECUREFILE parameter. If it is set to ALWAYS, the default LOB is SecureFiles; otherwise, it is BasicFiles.

Deduplication

With the SecureFiles deduplication option, the database will store only one copy of each LOB. The database will generate a hash key for a LOB and compare it to existing LOBs in that table or partition of a table, storing only one copy of each identical LOB. Note that deduplication does not work across partitions or subpartitions.

Compression

The SecureFiles compression option causes the database to compress the LOBs both on disk and in memory. Compression can be specified as MEDIUM (the default) or HIGH. HIGH compression will consume more CPU during the compression step, but will result in smaller LOBs. My simple test with PDF files showed that HIGH required about 25% longer to load than MEDIUM compression.

You can specify both deduplication and compression by including both options in the LOB clause, like this:

```
TABLE waterfalls
(
   falls_name          VARCHAR2 (80)
 , falls_photo         BLOB
 , falls_directions    CLOB
 , falls_description   NCLOB
```

```
 , falls_web_page      BFILE
)
LOB (falls_photo) STORE AS SECUREFILE (COMPRESS DEDUPLICATE)
LOB (falls_directions) STORE AS SECUREFILE (COMPRESS DEDUPLICATE)
LOB (falls_description) STORE AS SECUREFILE (COMPRESS DEDUPLICATE)
```

When you specify both options, deduplication occurs first, and then compression. Both deduplication and compression are part of the Advanced Compression Option of the database.

Encryption

As with deduplication and compression, you specify the SecureFiles encryption option by telling the database to encrypt your LOB in the LOB clause of your CREATE TABLE statement. You can optionally specify the encryption algorithm you want to use. The valid algorithms are 3DES168, AES128, AES192 (the default), and AES256. You can use any combination of deduplication, compression, and encryption, as shown in this example:

```
TABLE waterfalls
(
   falls_name          VARCHAR2 (80)
 , falls_photo         BLOB
 , falls_directions    CLOB
 , falls_description   NCLOB
 , falls_web_page      BFILE
)
LOB (falls_photo) STORE AS SECUREFILE (COMPRESS DEDUPLICATE)
LOB (falls_directions) STORE AS SECUREFILE (ENCRYPT USING 'AES256')
LOB (falls_description) STORE AS SECUREFILE
   (ENCRYPT DEDUPLICATE COMPRESS HIGH
   )
```

If your database has not been configured for transparent data encryption (TDE), described in Chapter 23), you will have a couple of prerequisite steps to follow before you can start encrypting your LOBs. First, you need to create a *wallet*. This is where the master key will be stored. If you choose to use the default location for the wallet (*$ORACLE_BASE/admin/$ORACLE_SID/wallet*), you can create and open the wallet in one step like this:

```
ALTER SYSTEM SET ENCRYPTION KEY AUTHENTICATED BY "My-secret!passc0de";
```

If you want to store your wallet in a nondefault location, you will need to specify this location via the *SQLNET.ORA* file. If you want to store your wallet in the directory */oracle/wallet*, include these lines in your *SQLNET.ORA* file:

```
ENCRYPTION_WALLET_LOCATION=(SOURCE=(METHOD=file)
  (METHOD_DATA=(DIRECTORY=/oracle/wallet)))
```

Once the wallet has been created, it will need to be opened again after each instance restart. You open and close the wallet like this:

```
ALTER SYSTEM SET ENCRYPTION WALLET OPEN AUTHENTICATED BY "My-secret!passc0de";
-- now close the wallet
ALTER SYSTEM SET ENCRYPTION WALLET CLOSE;
```

Temporary LOBs

So far, we've been talking about permanently storing large amounts of unstructured data by means of the various LOB datatypes. Such LOBs are known as *persistent LOBs*. Many applications have a need for *temporary LOBs* that act like local variables but do not exist permanently in the database. This section discusses temporary LOBs and the use of the DBMS_LOB built-in package to manipulate them.

Starting with Oracle8*i* Database, the database supports the creation, freeing, access, and update of temporary LOBs through the Oracle Call Interface (OCI) and DBMS_LOB calls. The default lifetime of a temporary LOB is the lifetime of the session that created it, but such LOBs may be explicitly freed sooner by the application. Temporary LOBs are ideal as transient workspaces for data manipulation, and because no logging is done and no redo records are generated, they offer better performance than persistent LOBs do. In addition, whenever you rewrite or update a LOB, the Oracle database copies the entire LOB to a new segment. By avoiding all the associated redo logging, applications that perform lots of piecewise operations on LOBs should see significant performance improvements with temporary LOBs.

A temporary LOB is empty when it is created: you don't need to (and, in fact, you can't) use the EMPTY_CLOB and EMPTY_BLOB functions to initialize LOB locators for a temporary LOB. By default, all temporary LOBs are deleted at the end of the session in which they were created. If a process dies unexpectedly or if the database crashes, any temporary LOBs are deleted and the space for temporary LOBs is freed.

Temporary LOBs are just like persistent LOBs in that they exist on disk inside your database. Don't let the word *temporary* fool you into thinking that they are memory structures. Temporary LOBs are written to disk, but instead of being associated with a specific LOB column in a specific table, they are written to disk in your session's temporary tablespace. Thus, if you use temporary LOBs, you need to make sure that your temporary tablespace is large enough to accommodate them.

Let's examine the processes for creating and freeing temporary LOBs. Then I'll explain how you can test to see whether a LOB locator points to a temporary or a permanent LOB. I'll finish up by covering some of the administrative details to consider when you're working with temporary LOBs.

Creating a temporary LOB

Before you can work with a temporary LOB, you need to create it. One way to do this is with a call to the DBMS_LOB.CREATETEMPORARY procedure. This procedure creates a temporary BLOB or CLOB and its corresponding index in your default temporary tablespace. The header is:

```
DBMS_LOB.CREATETEMPORARY (
   lob_loc IN OUT NOCOPY [ BLOB | CLOB CHARACTER SET ANY_CS ],
   cache   IN BOOLEAN,
   dur     IN PLS_INTEGER := DBMS_LOB.SESSION);
```

The parameters to DBMS_LOB.CREATETEMPORARY are listed in Table 13-1.

Table 13-1. CREATETEMPORARY parameters

Parameter	Description
lob_loc	Receives the locator for the LOB.
cache	Specifies whether or not the LOB should be read into the buffer cache.
dur	Controls the duration of the LOB. The *dur* argument can be one of these two named constants: *DBMS_LOB.SESSION* Specifies that the temporary LOB created should be cleaned up (memory freed) at the end of the session. This is the default. *DBMS_LOB.CALL* Specifies that the temporary LOB created should be cleaned up (memory freed) at the end of the current program call in which the LOB was created.

Another way to create a temporary LOB is to declare a LOB variable in your PL/SQL code and assign a value to it. For example, the following code creates both a temporary BLOB and a temporary CLOB:

```
DECLARE
   temp_clob CLOB;
   temp_blob BLOB;
BEGIN
   -- Assigning a value to a null CLOB or BLOB variable causes
   -- PL/SQL to implicitly create a session-duration temporary
   -- LOB for you.
   temp_clob :=' http://www.nps.gov/piro/';
   temp_blob := HEXTORAW('7A');
END;
```

I don't really have a strong preference as to which method you should use to create a temporary LOB, but I do believe the use of DBMS_LOB.CREATETEMPORARY makes the intent of your code a bit more explicit.

Freeing a temporary LOB

The DBMS_LOB.FREETEMPORARY procedure explicitly frees a temporary BLOB or CLOB, releasing the space from your default temporary tablespace. The header for this procedure is:

```
PROCEDURE DBMS_LOB.FREETEMPORARY (
   lob_loc IN OUT NOCOPY
      [ BLOB | CLOB CHARACTER SET ANY_CS ]);
```

In the following example, I again create two temporary LOBs. Then I explicitly free them:

```
DECLARE
   temp_clob CLOB;
   temp_blob BLOB;
BEGIN
   -- Assigning a value to a null CLOB or BLOB variable causes
   -- PL/SQL to implicitly create a session-duration temporary
   -- LOB for you.
   temp_clob := 'http://www.exploringthenorth.com/alger/alger.html';
   temp_blob := HEXTORAW('7A');

   DBMS_LOB.FREETEMPORARY(temp_clob);
   DBMS_LOB.FREETEMPORARY(temp_blob);
END;
```

After a call to FREETEMPORARY, the LOB locator that was freed (*lob_loc* in the previous specification) is marked as invalid. If an invalid LOB locator is assigned to another LOB locator through an assignment operation in PL/SQL, then the target of the assignment is also freed and marked as invalid.

PL/SQL will implicitly free temporary LOBs when they go out of scope at the end of a block.

Checking to see whether a LOB is temporary

The ISTEMPORARY function tells you if the LOB locator (*lob_loc* in the following specification) points to a temporary or a persistent LOB. The function returns an integer value: 1 means that it is a temporary LOB, and 0 means that it is not (it's a persistent LOB instead).

```
DBMS_LOB.ISTEMPORARY (
   lob_loc IN [ BLOB | CLOB CHARACTER SET ANY_CS ])
  RETURN INTEGER;
```

Note that while this function returns true (1) or false (0), it does not return a BOOLEAN datatype.

Managing temporary LOBs

Temporary LOBs are handled quite differently from normal persistent, internal LOBs. With temporary LOBs, there is no support for transaction management, consistent read operations, rollbacks, and so forth. There are various consequences of this lack of support:

- If you encounter an error when processing with a temporary LOB, you must free that LOB and start your processing over again.
- You should not assign multiple LOB locators to the same temporary LOB. Lack of support for consistent read and undo operations can cause performance degradation with multiple locators.
- If a user modifies a temporary LOB while another locator is pointing to it, a copy (referred to by Oracle as a *deep copy*) of that LOB is made. The different locators will then no longer see the same data. To minimize these deep copies, use the NOCOPY compiler hint whenever you're passing LOB locators as arguments.
- To make a temporary LOB permanent, you must call the DBMS_LOB.COPY program and copy the temporary LOB into a permanent LOB.
- Temporary LOB locators are unique to a session. You cannot pass a locator from one session to another (through a database pipe, for example) in order to make the associated temporary LOB visible in that other session. If you need to pass a LOB between sessions, use a permanent LOB.

Oracle9*i* Database introduced a V$ view called V$TEMPORARY_LOBS that shows how many cached and uncached LOBs exist per session. Your DBA can combine information from V$TEMPORARY_LOBS and the DBA_SEGMENTS data dictionary view to see how much space a session is using for temporary LOBs.

Native LOB Operations

Almost since the day Oracle unleashed LOB functionality to the vast hordes of database users, programmers and query-writers have wanted to treat LOBs as very large versions of regular, scalar variables. In particular, users wanted to treat CLOBs as very large character strings, passing them to SQL functions, using them in SQL statement WHERE clauses, and so forth. To the dismay of many, CLOBs originally could not be used interchangeably with VARCHAR2s. For example, in Oracle8 Database and Oracle8*i* Database, you could not apply a character function to a CLOB column:

```
SELECT SUBSTR(falls_directions,1,60)
  FROM waterfalls
```

Starting in Oracle9*i* Database, you can use CLOBs interchangeably with VARCHAR2s in a wide variety of situations:

- You can pass CLOBs to most SQL and PL/SQL VARCHAR2 functions—they are overloaded with both VARCHAR2 and CLOB parameters.
- In PL/SQL, but not in SQL, you can use various relational operators such as less-than (<), greater-than (>), and equals (=) with LOB variables.
- You can assign CLOB values to VARCHAR2 variables, and vice versa. You can also select CLOB values into VARCHAR2 variables and vice versa. This is because PL/SQL now implicitly converts between the CLOB and VARCHAR2 types.

SQL semantics

Oracle refers to the capabilities introduced in the previous section as offering *SQL semantics* for LOBs. From a PL/SQL developer's standpoint, it means that you can manipulate LOBs using native operators rather than a supplied package.

Following is an example showing some of the things you can do with SQL semantics:

```
DECLARE
   name CLOB;
   name_upper CLOB;
   directions CLOB;
   blank_space VARCHAR2(1) := ' ';
BEGIN
   -- Retrieve a VARCHAR2 into a CLOB, apply a function to a CLOB
   SELECT falls_name, SUBSTR(falls_directions,1,500)
   INTO name, directions
   FROM waterfalls
   WHERE falls_name = 'Munising Falls';

   -- Uppercase a CLOB
   name_upper := UPPER(name);

   -- Compare two CLOBs
   IF name = name_upper THEN
      DBMS_OUTPUT.PUT_LINE('We did not need to uppercase the name.');
   END IF;

   -- Concatenate a CLOB with some VARCHAR2 strings
   IF INSTR(directions,'Mackinac Bridge') <> 0 THEN
      DBMS_OUTPUT.PUT_LINE('To get to ' || name_upper || blank_space
                           || 'you must cross the Mackinac Bridge.');
   END IF;
END;
```

The output is:

```
To get to MUNISING FALLS you must cross the Mackinac Bridge.
```

The small piece of code in this example does several interesting things:

- The falls_name column is a VARCHAR2 column, yet it is retrieved into a CLOB variable. This is a demonstration of implicit conversion between the VARCHAR2 and CLOB types.
- The SUBSTR function is used to limit retrieval to only the first 500 characters of the directions to Munising Falls. Further, the UPPER function is used to uppercase the falls name. This demonstrates the application of SQL and PL/SQL functions to LOBs.
- The IF statement that compares name to name_upper is a bit forced, but it demonstrates that relational operators may now be applied to LOBs.
- The uppercased falls name, a CLOB, is concatenated with some string constants and one VARCHAR2 string (blank_space). This shows that CLOBs may be concatenated.

There are many restrictions and caveats that you need to be aware of when using this functionality. For example, not every function that takes a VARCHAR2 input will accept a CLOB in its place; there are some exceptions. The regular expression functions notably work with SQL semantics, while aggregate functions do not. Likewise, not all relational operators are supported for use with LOBs. All of these restrictions and caveats are described in detail in the section called "SQL Semantics and LOBs" in Chapter 10 of the *SecureFiles and Large Objects Developer's Guide* manual for Oracle Database 11*g* and 12*c*. For Oracle Database 10*g* see Chapter 9, "SQL Semantics and LOBs," of the *Application Developers Guide – Large Objects* manual. If you're using SQL semantics, I strongly suggest that you take a look at this section of the manual for your database.

SQL semantics for LOBs apply only to internal LOBs: CLOBs, BLOBs, and NCLOBs. SQL semantics support does not apply to BFILEs.

SQL semantics may yield temporary LOBs

One issue you will need to understand when applying SQL semantics to LOBs is that the result is often the creation of a temporary LOB. Think about applying the UPPER function to a CLOB:

```
DECLARE
   directions CLOB;
BEGIN
   SELECT UPPER(falls_directions)
     INTO directions
     FROM waterfalls
    WHERE falls_name = 'Munising Falls';
END;
```

Because they are potentially very large objects, CLOBs are stored on disk. The database can't uppercase the CLOB being retrieved because that would mean changing its value on disk—in effect, changing a value that you simply want to retrieve. Nor can the database make the change to an in-memory copy of the CLOB, because the value may not fit in memory, and also because what is being retrieved is only a locator that points to a value that must be on disk. The only option is for the database software to create a temporary CLOB in your temporary tablespace. The UPPER function copies data from the original CLOB to the temporary CLOB, uppercasing the characters during the copy operation. The SELECT statement then returns a LOB locator pointing to the temporary CLOB, not to the original CLOB. There are two extremely important ramifications to all this:

- You cannot use the locator returned by a function or expression to update the original LOB. The directions variable in my example cannot be used to update the persistent LOB stored in the database because it really points to a temporary LOB returned by the UPPER function.
- Disk space and CPU resources are expended to create a temporary LOB, which can be of considerable size. I'll discuss this issue more in "Performance impact of using SQL semantics" on page 446.

If I want to retrieve an uppercase version of the directions to Munising Falls while maintaining the ability to update the directions, I'll need to retrieve two LOB locators:

```
DECLARE
   directions_upper CLOB;
   directions_persistent CLOB;
BEGIN
   SELECT UPPER(falls_directions), falls_directions
     INTO directions_upper, directions_persistent
     FROM waterfalls
    WHERE falls_name = 'Munising Falls';
END;
```

Now I can access the uppercase version of the directions via the locator in directions_upper, and I can modify the original directions via the locator in directions_persistent. There's no performance penalty in this case from retrieving the extra locator. The performance hit comes from uppercasing the directions and placing them into a temporary CLOB. The locator in directions_persistent is simply plucked as is from the database table.

In general, any character-string function to which you normally pass a VARCHAR2, and that normally returns a VARCHAR2 value, will return a temporary CLOB when you pass in a CLOB as input. Similarly, expressions that return CLOBs will most certainly return temporary CLOBs. Temporary CLOBs and BLOBs cannot be used to update the LOBs that you originally used in an expression or function.

Performance impact of using SQL semantics

You'll need to give some thought to performance when you are using the new SQL semantics for LOB functionality. Remember that the *L* in LOB stands for *large*, which can mean as much as 128 terabytes (4 gigabytes prior to Oracle Database 10*g*). Consequently, you may encounter some serious performance issues if you indiscriminately treat LOBs the same as any other type of variable or column. Have a look at the following query, which attempts to identify all waterfalls for which a visit might require a trip across the Mackinac Bridge:

```
SELECT falls_name
  FROM waterfalls
 WHERE INSTR(UPPER(falls_directions),'MACKINAC BRIDGE') <> 0;
```

Think about what the Oracle database must do to resolve this query. For every row in the waterfalls table, it must take the falls_directions column, uppercase it, and place those results into a temporary CLOB (residing in your temporary tablespace). Then it must apply the INSTR function to that temporary LOB to search for the string 'MACKINAC BRIDGE'. In my examples, the directions have been fairly short. Imagine, however, that falls_directions were truly a large LOB, and that the average column size were one gigabyte. Think of the drain on your temporary tablespace as the database allocates the necessary room for the temporary LOBs created when uppercasing the directions. Then think of all the time required to make a copy of each CLOB in order to uppercase it, the time required to allocate and deallocate space for temporary CLOBs in your temporary tablespace, and the time required for the INSTR function to search character-by-character through an average of one gigabyte per CLOB. Such a query would surely bring the wrath of your DBA down upon you.

Oracle Text and SQL Semantics

If you need to execute queries that look at uppercase versions of CLOB values, and you need to do so efficiently, Oracle Text may hold the solution. For example, you might reasonably expect to write a query such as the following some day:

```
SELECT falls_name
  FROM waterfalls
 WHERE INSTR(UPPER(falls_directions), 'MACKINAC BRIDGE') <> 0;
```

If falls_directions is a CLOB column, this query may not be all that efficient. However, if you are using Oracle Text, you can define a case-insensitive Oracle Text index on that CLOB column, and then use the CONTAINS predicate to efficiently evaluate the query:

```
SELECT falls_name
  FROM waterfalls
 WHERE
     CONTAINS(falls_directions,'mackinac bridge') > 0;
```

For more information on CONTAINS and case-insensitive indexes using Oracle Text, see Oracle Corporation's *Text Application Developer's Guide*.

Because of all the performance ramifications of applying SQL semantics to LOBs, Oracle's documentation suggests that you limit such applications to LOBs that are 100 KB or less in size. I myself don't have a specific size recommendation to pass on to you; you should consider each case in terms of your particular circumstances and how much you need to accomplish a given task. I encourage you always to give thought to the performance implications of using SQL semantics for LOBs, and possibly to run some tests to experience these implications, so that you can make a reasonable decision based on your circumstances.

LOB Conversion Functions

Oracle provides several conversion functions that are sometimes useful for working with large object data, described in Table 13-2.

Table 13-2. LOB conversion functions

Function	Description
TO_CLOB (*character_data*)	Converts character data into a CLOB. The input to TO_CLOB can be any of the following character types: VARCHAR2, NVARCHAR2, CHAR, NCHAR, CLOB, and NCLOB. If necessary (for example, if the input is NVARCHAR2), input data is converted from the national character set into the database character set.
TO_BLOB(*raw_data*)	Similar to TO_CLOB, but converts RAW or LONG RAW data into a BLOB.
TO_NCLOB (*character_data*)	Does the same as TO_CLOB, except that the result is an NCLOB using the national character set.
TO_LOB (*long_data*)	Accepts either LONG or LONG RAW data as input, and converts that data to a CLOB or a BLOB, respectively. TO_LOB may be invoked only from the select list of a subquery in an INSERT...SELECT...FROM statement.
TO_RAW(*blob_data*)	Takes a BLOB as input and returns the BLOB's data as a RAW value.

The TO_LOB function is designed specifically to enable one-time conversion of LONG and LONG RAW columns into CLOB and BLOB columns, because LONG and LONG RAW are now considered obsolete. The TO_CLOB and TO_NCLOB functions provide a convenient mechanism for converting character large object data between the database and national language character sets.

Predefined Object Types

Starting with Oracle9*i* Database Release 1, Oracle provides a collection of useful predefined object types:

XMLType
: Use this to store and manipulate XML data.

URI types
: Use these to store uniform resource identifiers (such as HTML addresses).

Any types
: Use these to define a PL/SQL variable that can hold any type of data.

The following subsections discuss these predefined object types in more detail.

The XMLType Type

Oracle9*i* Database introduced a native object type called XMLType. You can use XMLType to define database columns and PL/SQL variables containing XML documents. Methods defined on XMLType enable you to instantiate new XMLType values, to extract portions of an XML document, and to otherwise manipulate the contents of an XML document in various ways.

XML is a huge subject that I can't hope to cover in detail in this book. However, if you're working with XML from PL/SQL, there are at least two things you need to know about:

XMLType
: A built-in object type that enables you to store XML documents in a database column or in a PL/SQL variable. XMLType was introduced in Oracle9*i* Database Release 1.

XQuery
: A query language used for retrieving and constructing XML documents. XQuery was introduced in Oracle Database 10*g* Release 2.

Starting with these two technologies and exploring further, you'll encounter many other XML-related topics that will likely prove useful: XPath for referring to portions of an XML document, XML Schema for describing document structure, and so forth.

Using XMLType, you can easily create a table to hold XML data:

```
CREATE TABLE fallsXML (
   fall_id NUMBER,
   fall XMLType
);
```

The fall column in this table is of type XMLType and can hold XML data. To store XML data into this column, you must invoke the static CreateXML method, passing it your XML data. CreateXML accepts XML data as input and instantiates a new XMLType object to hold that data. The new object is then returned as the method's result, and it is that object that you must store in the column. CreateXML is overloaded to accept both VARCHAR2 strings and CLOBs as input.

Use the following INSERT statements to create three XML documents in the falls table:

```
INSERT INTO fallsXML VALUES (1, XMLType.CreateXML(
   '<?xml version="1.0"?>
```

```
    <fall>
       <name>Munising Falls</name>
       <county>Alger</county>
       <state>MI</state>
       <url>
          http://michiganwaterfalls.com/munising_falls/munising_falls.html
       </url>
    </fall>'));

INSERT INTO fallsXML VALUES (2, XMLType.CreateXML(
   '<?xml version="1.0"?>
    <fall>
       <name>Au Train Falls</name>
       <county>Alger</county>
       <state>MI</state>
       <url>
          http://michiganwaterfalls.com/autrain_falls/autrain_falls.html
       </url>
    </fall>'));

INSERT INTO fallsXML VALUES (3, XMLType.CreateXML(
   '<?xml version="1.0"?>
    <fall>
       <name>Laughing Whitefish Falls</name>
       <county>Alger</county>
       <state>MI</state>
       <url>
         http://michiganwaterfalls.com/whitefish_falls/whitefish_falls.html
       </url>
    </fall>'));
```

You can query XML data in the table using various XMLType methods. The existsNode method used in the following example allows you to test for the existence of a specific XML node in an XML document. The built-in SQL EXISTSNODE function, also in the example, performs the same test. Whether you use the method or the built-in function, you identify the node of interest using an XPath expression.[1]

Both of the following statements produce the same output:

```
SQL> SELECT f.fall_id
   2  FROM fallsxml f
   3  WHERE f.fall.existsNode('/fall/url') > 0;

SQL> SELECT f.fall_id
   2  FROM fallsxml f
   3  WHERE EXISTSNODE(f.fall,'/fall/url') > 0;
   4  /
```

1. XPath is a syntax that describes parts of an XML document. Among other things, you can use XPath to specify a particular node or attribute value in an XML document.

```
   FALL_ID
----------
         1
         2
```

You can, of course, also work with XML data from within PL/SQL. In the following example, I retrieve the fall column for Munising Falls into a PL/SQL variable that is also of type XMLType. Thus, I retrieve the entire XML document into my PL/SQL program, where I can work further with it. After retrieving the document, I extract and print the text from the */fall/url* node:

```
<<demo_block>>
DECLARE
   fall XMLType;
   url VARCHAR2(100);
BEGIN
   -- Retrieve XML for Munising Falls
   SELECT f.fall
     INTO demo_block.fall
     FROM fallsXML f
    WHERE f.fall_id = 1;

   -- Extract and display the URL for Munising Falls
   url := fall.extract('/fall/url/text()').getStringVal;
   DBMS_OUTPUT.PUT_LINE(url);
END;
```

The output is:

```
http://michiganwaterfalls.com/munising_falls/munising_falls.html
```

Pay special attention to the following two lines:

`SELECT f.fall INTO demo_block.fall`
: My variable name, fall, matches the name of the column in the database table. In my SQL query, therefore, I qualify my variable name with the name of my PL/SQL block.

`url := fall.extract('/fall/url/text()').getStringVal;`
: To get the text of the URL, I invoke two of XMLType's methods:

  `extract`
  : Returns an XML document, of type XMLType, containing only the specified fragment of the original XML document. Use XPath notation to specify the fragment you want returned.

  `getStringVal`
  : Returns the text of an XML document.

In my example, I apply the getStringVal method to the XML document returned by the extract method, thus retrieving the text for the Munising Falls URL. The extract method

returns the contents of the <url> node as an XMLType object, and getStringVal then returns that content as a text string that I can display.

You can even index XMLType columns to allow for efficient retrieval of XML documents based on their content. You do this by creating a function-based index, for which you need the QUERY REWRITE privilege. The following example creates a function-based index on the first 80 characters of each falls name:

```
CREATE INDEX falls_by_name
   ON fallsxml f (
      SUBSTR(
         XMLType.getStringVal(
            XMLType.extract(f.fall,'/fall/name/text()')
         ),1,80
      ))
```

Note that I used the SUBSTR function in the creation of this index. The getStringVal method returns a string that is too long to index, resulting in an *ORA-01450: maximum key length (3166) exceeded* error. Thus, when creating an index like this, I must use SUBSTR to restrict the results to some reasonable length.

If you decide to use XMLType in any of your applications, be sure to consult Oracle Corporation's documentation for more complete and current information. The *XML DB Developer's Guide* is an important, if not critical, reference for developers working with XML. The *SQL Reference* also has some useful information on XMLType and on the built-in SQL functions that support XML. Oracle's *PL/SQL Packages and Types Reference* documents the programs, methods, and exceptions for each of the predefined object types, as well as several packages that work with XML data, such as DBMS_XDB, DBMS_XMLSCHEMA, and DBMS_XMLDOM.

The URI Types

The URI types, introduced in Oracle9*i* Database, consist of a supertype and a collection of subtypes that provide support for storing URIs in PL/SQL variables and in database columns. UriType is the supertype, and a UriType variable can hold any instance of one of these subtypes:

HttpUriType
: A subtype of UriType that is specific to HTTP URLs, which usually point to web pages.

DBUriType
: A subtype of UriType that supports URLs that are XPath expressions.

XDBUriType
: A subtype of UriType that supports URLs that reference Oracle XML DB objects. XML DB is Oracle's name for a set of XML technologies built into the database.

To facilitate your work with URIs, the Oracle database also provides a UriFactory package that automatically generates the appropriate URI type for whatever URI you pass to it.

The URI types are created by the script named *$ORACLE_HOME/rdbms/admin/dbmsuri.sql*. All the types and subtypes are owned by the user SYS.

Starting with Oracle Database 11*g*, you need to create and configure access control lists (ACLs) to allow network access. This security enhancement requires a few prerequisites before you can access the Internet. You have to create a network ACL, add privileges to it, and then define the allowable destinations to which the ACL permits access:

```
BEGIN
  -- create the ACL
  DBMS_NETWORK_ACL_ADMIN.CREATE_ACL(
     acl          => 'oreillynet-permissions.xml'
    ,description  => 'Network permissions for www.oreillynet.com'
    ,principal    => 'WEBROLE'
    ,is_grant     => TRUE
    ,privilege    => 'connect'
    ,start_date   => SYSTIMESTAMP
    ,end_date     => NULL
  );
  -- assign privileges to the ACL
  DBMS_NETWORK_ACL_ADMIN.ADD_PRIVILEGE (
      acl        => 'oreillynet-permissions.xml'
     ,principal  => 'WEBROLE'
     ,is_grant   => TRUE
     ,privilege  => 'connect'
     ,start_date => SYSTIMESTAMP
     ,end_date   => null
  );
  -- define the allowable destinations
  DBMS_NETWORK_ACL_ADMIN.ASSIGN_ACL (
      acl        => 'oreillynet-permissions.xml'
     ,host       => 'www.orillynet.com'
     ,lower_port => 80
     ,upper_port => 80
  );
  COMMIT;  -- you must commit the changes
END;
```

Now I can retrieve my web pages using HttpUriType:

```
DECLARE
   WebPageURL HttpUriType;
   WebPage CLOB;
BEGIN
   -- Create an instance of the type pointing
   -- to Steven's Author Bio page at O'Reilly
   WebPageURL := HttpUriType.createUri('http://www.oreillynet.com/pub/au/344');
```

```
    -- Retrieve the page via HTTP
    WebPage := WebPageURL.getclob();

    -- Display the page title
    DBMS_OUTPUT.PUT_LINE(REGEXP_SUBSTR(WebPage,'<title>.*</title>'));
END;
```

The output from this code example is:

```
<title>Steven Feuerstein</title>
```

For more information on the use of the UriType family, see Chapter 20, "Accessing Data Through URIs," of the *XML DB Developer's Guide*.

The Any Types

Back in Chapter 7, I described PL/SQL as a statically typed language. For the most part this is true—datatypes must be declared and checked at compile time. However, there are occasions when you really need the capabilities of dynamic typing, and for those occasions, the *Any* types were introduced with Oracle9*i* Database Release 1. These dynamic datatypes enable you to write programs that manipulate data when you don't know the type of that data until runtime. Member functions support *introspection*, allowing you to determine the type of a value at runtime and to access that value.

An introspection function is one that you can use in a program to examine and learn about variables declared by your program. In essence, your program learns about itself—hence the term *introspection*.

The Any types are opaque, meaning that you cannot manipulate the internal structures directly, but instead must use programs.

The following predefined types belong to this family:

AnyData
: Can hold a single value of any type, whether it's a built-in scalar datatype, a user-defined object type, a nested table, a large object, a varying array (VARRAY), or any other type not listed here.

AnyDataSet
: Can hold a set of values of any type, as long as all values are of the same type.

AnyType
: Can hold a description of a type. Think of this as an AnyData without the data.

The Any types are included with a starter database or can be created with the script named *dbmsany.sql* found in *$ORACLE_HOME/rdbms/admin*, and they are owned by the user SYS.

In addition to creating the Any types, the *dbmsany.sql* script also creates a package named DBMS_TYPES that defines a set of named constants, such as TYPECODE_DATE. You can use these constants in conjunction with introspection functions such as GETTYPE in order to determine the type of data held by a given AnyData or AnyDataSet variable. The specific numeric values assigned to the constants are not important; you should always reference the named constants, not their underlying values.

The following example creates two user-defined types representing two kinds of geographic features. The subsequent PL/SQL block then uses SYS.AnyType to define a heterogeneous array of features (i.e., one where each array element can be of a different datatype).

First, I'll create the following two geographic feature types:

```
/* File on web: ch13_anydata.sql */
TYPE waterfall AS OBJECT (
   name VARCHAR2(30),
   height NUMBER
)

TYPE river AS OBJECT (
   name VARCHAR2(30),
   length NUMBER
)
```

Next, I'll execute the following PL/SQL code block:

```
DECLARE
   TYPE feature_array IS VARRAY(2) OF SYS.AnyData;
   features feature_array;
   wf waterfall;
   rv river;
   ret_val NUMBER;
BEGIN
   -- Create an array where each element is of
   -- a different object type
   features := feature_array(
                  AnyData.ConvertObject(
                     waterfall('Grand Sable Falls',30)),
                  AnyData.ConvertObject(
                     river('Manistique River', 85.40))
               );

   -- Display the feature data
   FOR x IN 1..features.COUNT LOOP
      -- Execute code pertaining to whatever object type
      -- we are currently looking at.
      -- NOTE! GetTypeName returns SchemaName.TypeName,
      -- so replace PLUSER with the schema you are using.
       CASE features(x).GetTypeName
```

```
        WHEN 'PLUSER.WATERFALL' THEN
           ret_val := features(x).GetObject(wf);
           DBMS_OUTPUT.PUT_LINE('Waterfall: '
              || wf.name || ', Height = ' || wf.height || ' feet.');
        WHEN 'PLUSER.RIVER' THEN
           ret_val := features(x).GetObject(rv);
           DBMS_OUTPUT.PUT_LINE('River: '
              || rv.name || ', Length = ' || rv.length || ' miles.');
        ELSE
           DBMS_OUTPUT.PUT_LINE('Unknown type '||features(x).GetTypeName);
        END CASE;
     END LOOP;
  END;
```

Finally, my output should appear as follows:

```
Waterfall: Grand Sable  Falls, Height = 30 feet.
River: Manistique River, Length = 85.4 miles.
```

Let's look at this code one piece at a time. The features are stored in a VARRAY, which is initialized as follows:

```
features := feature_array(
               AnyData.ConvertObject(
                  waterfall('Grand Sable Falls',30)),
               AnyData.ConvertObject(
                  river('Manistique River, 85.40))
            );
```

Working from the inside out and focusing on Grand Sable Falls, we can interpret this code as follows:

`waterfall('Grand Sable Falls',30)`
: Invokes the constructor for the waterfall type to create an object of type waterfall.

`AnyData.ConvertObject(`
: Converts (casts) the waterfall object into an instance of SYS.AnyData, allowing it to be stored in my array of SYS.AnyData objects.

`feature_array(`
: Invokes the constructor for the array. Each argument to feature_array is of type AnyData. The array is built from the two arguments I pass.

VARRAYs were discussed in Chapter 12, and you can read about object types in more detail in Chapter 26.

The next significant part of the code is the FOR loop in which each object in the features array is examined. A call to:

```
features(x).GetTypeName
```

returns the fully qualified type name of the current features object. For user-defined objects, the type name is prefixed with the schema name of the user who created the object. I had to include this schema name in my WHEN clauses—for example:

```
WHEN 'PLUSER.WATERFALL' THEN
```

If you're running this example on your own system, be sure to replace the schema I used (PLUSER) with the one that is valid for you. When creating TYPEs that will be used with introspection, consider the type's owner carefully, as that owner may need to be statically included in the code.

For built-in types such as NUMBER, DATE, and VARCHAR2, GetTypeName will return just the type name. Schema names apply only to user-defined types (i.e., those created using CREATE TYPE).

Once I determined which datatype I was dealing with, I retrieved the specific object using the following call:

```
ret_val := features(x).GetObject(wf);
```

In my example, I ignored the return code. There are two possible return code values:

DBMS_TYPES.SUCCESS
: The value (or object, in this case) was successfully returned.

DBMS_TYPES.NO_DATA
: No data was ever stored in the AnyData variable in question, so no data can be returned.

Once I had the object in a variable, it was an easy enough task to write a DBMS_OUTPUT statement specific to that object type. For example, to print information about waterfalls, I used:

```
DBMS_OUTPUT.PUT_LINE('Waterfall: '
   || wf.name || ', Height = ' || wf.height || ' feet.');
```

For more information on the Any family of types:

- See Chapter 26, which examines the Any datatypes from an object-oriented perspective.
- Check out Oracle's *PL/SQL Packages and Types Reference* and *Object-Relational Developer's Guide*.
- Try out the *anynums.pkg* and *anynums.tst* scripts on the book's website.

From an object-oriented design standpoint, there are better ways to deal with multiple feature types than the method I used in this section's example. In the real world, however, not everything is ideal, and my example does serve the purpose of demonstrating the utility of the SYS.AnyData predefined object type.