CHAPTER 15

Data Loading and Unloading

In this chapter, we will discuss data loading and unloading—in other words, how to get data *into* and *out of* an Oracle database. The main focus of the chapter is on the following bulk data loading tools:

- *SQL*Loader (pronounced "sequel loader")*: This is still a predominant method for loading data.
- *External tables*: This is a new feature with Oracle9*i* and above that permits access to operating system files as if they were database tables and, in Oracle 10*g* and above, even allows for the creation of operating system files as extracts of tables.

In the area of data unloading, we'll look at two techniques:

- *Flat file unload*: The flat file unloads will be custom developed implementations, but will provide you with a result that is portable to other types of systems (even a spreadsheet).
- *Data Pump unload*: Data Pump is a binary format proprietary to Oracle and is accessible via the Data Pump tool and external tables.

SQL*Loader

SQL*Loader (SQLLDR) is Oracle's high-speed, bulk data loader. It is an extremely useful tool used to get data into an Oracle database from a variety of flat file formats. SQLLDR can be used to load enormous amounts of data in an amazingly short period of time. It has two modes of operation:

- *Conventional path*: SQLLDR will employ SQL inserts on our behalf to load data.
- *Direct path*: SQLLDR does not use SQL in this mode; it formats database blocks directly.

The direct path load allows you to read data from a flat file and write it directly to formatted database blocks, bypassing the entire SQL engine, undo generation and, optionally, redo generation at the same time. Parallel direct path load is among the fastest ways to go from having no data to a fully loaded database.

We will not cover every single aspect of SQLLDR. For all of the details, refer to the *Oracle Utilities* manual, which dedicates seven chapters to SQLLDR in Oracle 10*g*. The fact that it is covered in seven chapters is notable, since every other utility, such as DBVERIFY, DBNEWID, and LogMiner get one chapter or less. For complete syntax and all of the options, I will refer you to the *Oracle Utilities* manual, as this chapter is intended to answer the "How do I . . .?" questions that a reference manual does not address.

It should be noted that the Oracle Call Interface (OCI) allows you to write your own direct path loader using C, with Oracle 8.1.6 Release 1 and onward. This is useful when the operation you want to perform is not feasible in SQLLDR, or when seamless integration with your application is desired. SQLLDR is a command-line tool (i.e., it's a separate program). It is not an API or anything that can be "called from PL/SQL," for example.

If you execute SQLLDR from the command line with no inputs, it gives you the following help:

```
[tkyte@desktop tkyte]$ sqlldr
SQL*Loader: Release 10.1.0.4.0 - Production on Sat Jul 16 10:32:28 2005
Copyright (c) 1982, 2004, Oracle.  All rights reserved.

Usage: SQLLDR keyword=value [,keyword=value,...]

Valid Keywords:
    userid -- ORACLE username/password
   control -- control file name
       log -- log file name
       bad -- bad file name
      data -- data file name
   discard -- discard file name
discardmax -- number of discards to allow          (Default all)
      skip -- number of logical records to skip    (Default 0)
      load -- number of logical records to load    (Default all)
    errors -- number of errors to allow            (Default 50)
      rows -- number of rows in conventional path bind array or
              between direct path data saves
              (Default: Conventional path 64, Direct path all)
  bindsize -- size of conventional path bind array in bytes  (Default 256000)
    silent -- suppress messages during run
              (header,feedback,errors,discards,partitions)
    direct -- use direct path                      (Default FALSE)
   parfile -- parameter file: name of file that contains parameter specifications
  parallel -- do parallel load                     (Default FALSE)
      file -- file to allocate extents from
skip_unusable_indexes -- disallow/allow unusable indexes or index partitions
                        (Default FALSE)
skip_index_maintenance -- do not maintain indexes, mark affected indexes as unusable
                        (Default FALSE)
commit_discontinued -- commit loaded rows when load is discontinued  (Default FALSE)
readsize -- size of read buffer (Default 1048576)
external_table -- use external table for load; NOT_USED, GENERATE_ONLY, EXECUTE
```

```
                  (Default NOT_USED)
columnarrayrows -- number of rows for direct path column array  (Default 5000)
streamsize -- size of direct path stream buffer in bytes  (Default 256000)
multithreading -- use multithreading in direct path
resumable -- enable or disable resumable for current session  (Default FALSE)
resumable_name -- text string to help identify resumable statement
resumable_timeout -- wait time (in seconds) for RESUMABLE  (Default 7200)
date_cache -- size (in entries) of date conversion cache  (Default 1000)
...
```

Rather than explain what each individual parameter technically means, I will point you to the *Oracle Utilities* manual, specifically Chapter 7 in the Oracle 10*g Utilities Guide* and Chapter 4 in the Oracle9*i Utilities Guide*. I will demonstrate the usage of a few of these parameters in this chapter.

To use SQLLDR, you will need a *control file*. A control file simply contains information describing the input data—its layout, datatypes, and so on—as well as information about the target table(s). The control file can even contain the data to load. In the following example, we'll build a simple control file in a step-by-step fashion, and I'll provide an explanation of the commands. (Note that the parenthetical numbers to the left in the code are *not* part of this control file; they are just there for reference.)

```
(1)  LOAD DATA
(2)  INFILE *
(3)  INTO TABLE DEPT
(4)  FIELDS TERMINATED BY ','
(5)  (DEPTNO, DNAME, LOC )
(6)  BEGINDATA
(7)  10,Sales,Virginia
(8)  20,Accounting,Virginia
(9)  30,Consulting,Virginia
(10) 40,Finance,Virginia
```

- `LOAD DATA` (1): This tells SQLLDR what to do (in this case, load data). The other thing SQLLDR can do is `CONTINUE_LOAD`, to resume a load. You would use this latter option only when continuing a multitable direct path load.
- `INFILE *` (2): This tells SQLLDR the data to be loaded is actually contained within the control file itself, as shown on lines 6 through 10. Alternatively, you could specify the name of another file that contains the data. You can override this `INFILE` statement using a command-line parameter if you wish. Be aware that *command-line options override control file settings*.
- `INTO TABLE DEPT` (3): This tells SQLLDR to which table you are loading data (in this case, the `DEPT` table).
- `FIELDS TERMINATED BY ','` (4): This tells SQLLDR that the data will be in the form of comma-separated values. There are dozens of ways to describe the input data to SQLLDR; this is just one of the more common methods.

- (DEPTNO, DNAME, LOC) (5): This tells SQLLDR what columns you are loading, their order in the input data, and their datatypes. The datatypes are for the data in the *input* stream, not the datatypes in the database. In this case, they are defaulting to CHAR(255), which is sufficient.
- BEGINDATA (6): This tells SQLLDR you have finished describing the input data and that the very next lines, lines 7 to 10, are the actual data to be loaded into the DEPT table.

This is a control file in one of its most simple and common formats: to load delimited data into a table. We will take a look at some complex examples in this chapter, but this is a good one to get our feet wet with. To use this control file, which we will name demo1.ctl, all we need to do is create an empty DEPT table:

```
ops$tkyte@ORA10G> create table dept
  2  ( deptno  number(2) constraint dept_pk primary key,
  3    dname   varchar2(14),
  4    loc     varchar2(13)
  5  )
  6  /
Table created.
```

and run the following command:

```
[tkyte@desktop tkyte]$ sqlldr userid=/ control=demo1.ctl
SQL*Loader: Release 10.1.0.4.0 - Production on Sat Jul 16 10:59:06 2005
Copyright (c) 1982, 2004, Oracle.  All rights reserved.
Commit point reached - logical record count 4
```

If the table is not empty, we will receive an error message to the following effect:

```
SQLLDR-601: For INSERT option, table must be empty.  Error on table DEPT
```

This is because we allowed almost everything in the control file to default, and the default load option is INSERT (as opposed to APPEND, TRUNCATE, or REPLACE). To INSERT, SQLLDR assumes the table is empty. If we wanted to *add* records to the DEPT table, we could have specified APPEND, or to replace the data in the DEPT table, we could have used REPLACE or TRUNCATE. REPLACE uses a conventional DELETE statement; hence, if the table to be loaded into already contains many records, it could be quite slow to perform. TRUNCATE uses the TRUNCATE SQL command and is typically faster, as it does not have to physically remove each row.

Every load will generate a log file. The log file from our simple load looks like this:

```
SQL*Loader: Release 10.1.0.4.0 - Production on Sat Jul 16 10:59:06 2005
Copyright (c) 1982, 2004, Oracle.  All rights reserved.

Control File:   demo1.ctl
Data File:      demo1.ctl
  Bad File:     demo1.bad
  Discard File:  none specified

 (Allow all discards)
```

```
Number to load: ALL
Number to skip: 0
Errors allowed: 50
Bind array:     64 rows, maximum of 256000 bytes
Continuation:    none specified
Path used:      Conventional

Table DEPT, loaded from every logical record.
Insert option in effect for this table: INSERT

   Column Name                  Position   Len  Term Encl Datatype
------------------------------ ---------- ----- ---- ---- ---------------------
DEPTNO                              FIRST     *   ,       CHARACTER
DNAME                                NEXT     *   ,       CHARACTER
LOC                                  NEXT     *   ,       CHARACTER

Table DEPT:
  4 Rows successfully loaded.
  0 Rows not loaded due to data errors.
  0 Rows not loaded because all WHEN clauses were failed.
  0 Rows not loaded because all fields were null.

Space allocated for bind array:                 49536 bytes(64 rows)
Read   buffer bytes: 1048576

Total logical records skipped:          0
Total logical records read:             4
Total logical records rejected:         0
Total logical records discarded:        0

Run began on Sat Jul 16 10:59:06 2005
Run ended on Sat Jul 16 10:59:06 2005

Elapsed time was:     00:00:00.15
CPU time was:         00:00:00.03
```

The log file tells us about many of the aspects of our load. We can see the options we used (defaulted or otherwise). We can see how many records were read, how many loaded, and so on. The log file specifies the locations of all BAD and DISCARD files. It even tells us how long it took. The log file is crucial for verifying that the load was successful, as well as for diagnosing errors. If the loaded data resulted in SQL errors (i.e., the input data was "bad" and created records in the BAD file), these errors would be recorded here. The information in the log file is largely self-explanatory, so we will not spend any more time on it.

Loading Data with SQLLDR FAQs

We will now cover what I have found to be the most frequently asked questions with regard to loading data in an Oracle database using SQLLDR.

How Do I Load Delimited Data?

Delimited data, or data that is separated by some special character and perhaps enclosed in quotes, is the most popular data format for flat files today. On a mainframe, a fixed-length, fixed-format file would probably be the most recognized file format, but on UNIX and NT, delimited files are the norm. In this section, we will investigate the popular options used to load delimited data.

The most popular format for delimited data is the *comma-separated values* (*CSV*) format. In this file format, each field of data is separated from the next by a comma. Text strings can be enclosed within quotes, thus allowing for the string itself to contain commas. If the string must contain a quotation mark as well, the convention is to double up the quotation mark (in the following code we use "" in place of just "). A typical control file to load delimited data will look much like our first example earlier, but the `FIELDS TERMINATED BY` clause would generally be specified like this:

```
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
```

It specifies that a comma separates the data fields, and that each field *might* be enclosed in double quotes. If we were to modify the bottom of this control file to be

```
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(DEPTNO, DNAME, LOC )
BEGINDATA
10,Sales,"Virginia,USA"
20,Accounting,"Va, ""USA"""
30,Consulting,Virginia
40,Finance,Virginia
```

when we run SQLLDR using this control file, the results will be as follows:

```
ops$tkyte@ORA10G> select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 Sales          Virginia,USA
        20 Accounting     Va, "USA"
        30 Consulting     Virginia
        40 Finance        Virginia
```

Notice the following in particular:

- `Virginia,USA` in department `10`: This results from input data that was "`Virginia,USA`". This input data field had to be enclosed in quotes to retain the comma as part of the data. Otherwise, the comma would have been treated as the end-of-field marker, and `Virginia` would have been loaded without the `USA` text.
- `Va, "USA"`: This resulted from input data that was `"Va, ""USA"""`. SQLLDR counted the double occurrence of " as a single occurrence within the enclosed string. To load a string that contains the optional enclosure character, you must ensure the enclosure character is doubled up.

Another popular format is *tab-delimited data*, which is data separated by tabs rather than commas. There are two ways to load this data using the TERMINATED BY clause:

- TERMINATED BY X'09' (the tab character using hexadecimal format; in ASCII, 9 is a tab character)
- TERMINATED BY WHITESPACE

The two are very different in implementation, as the following shows. Using the DEPT table from earlier, we'll load using this control file:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY WHITESPACE
(DEPTNO, DNAME, LOC)
BEGINDATA
10 Sales Virginia
```

It is not readily visible on the page, but there are *two* tabs between each piece of data here. The data line is really

```
10\t\tSales\t\tVirginia
```

where the \t is the universally recognized tab escape sequence. When you use this control file with the TERMINATED BY WHITESPACE clause as shown previously, the resulting data in the table DEPT is

```
ops$tkyte@ORA10G> select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 Sales          Virginia
```

TERMINATED BY WHITESPACE parses the string by looking for the first occurrence of white-space (tab, blank, or newline), and then it continues until it finds the next *non*-whitespace character. Hence, when it parsed the data, DEPTNO had 10 assigned to it, the two subsequent tabs were considered as whitespace, Sales was assigned to DNAME, and so on.

On the other hand, if you were to use FIELDS TERMINATED BY X'09', as the following modified control file does:

```
...
FIELDS TERMINATED BY X'09'
(DEPTNO, DNAME, LOC )
...
```

you would find DEPT loaded with the following data:

```
ops$tkyte@ORA10G> select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10                Sales
```

Here, once SQLLDR encountered a tab, it output a value. Hence, 10 is assigned to DEPTNO, and DNAME gets NULL since there is no data between the first tab and the next occurrence of a tab. Sales gets assigned to LOC.

This is the intended behavior of TERMINATED BY WHITESPACE and TERMINATED BY <character>. Which is more appropriate to use will be dictated by the input data and how you need it to be interpreted.

Lastly, when loading delimited data such as this, it is very common to want to skip over various columns in the input record. For example, you might want to load fields 1, 3, and 5, skipping columns 2 and 4. To do this, SQLLDR provides the FILLER keyword. This allows you to map a column in an input record, but not put it into the database. For example, given the DEPT table and the last control file from earlier, we can modify the control file to load the data correctly (skipping over the tabs) using the FILLER keyword:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY x'09'
(DEPTNO, dummy1 filler, DNAME, dummy2 filler, LOC)
BEGINDATA
10      Sales       Virginia
```

The resulting DEPT table is now as follows:

```
ops$tkyte@ORA10G> select * from dept;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 Sales          Virginia
```

How Do I Load Fixed Format Data?

Often, you have a flat file generated from some external system, and this file is a fixed-length file with positional data. For example, the NAME field is in bytes 1 to 10, the ADDRESS field is in bytes 11 to 35, and so on. We will look at how SQLLDR can import this kind of data for us.

This fixed-width, positional data is the optimal data format for SQLLDR to load. It will be the fastest way to process, as the input data stream is somewhat trivial to parse. SQLLDR will have stored fixed-byte offsets and lengths into data records, and extracting a given field is very simple. If you have an extremely large volume of data to load, converting it to a fixed position format is generally the best approach. The downside to a fixed-width file is, of course, that it can be much larger than a simple, delimited file format.

To load fixed-width positional data, you will use the POSITION keyword in the control file, for example:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
( DEPTNO position(1:2),
  DNAME  position(3:16),
  LOC    position(17:29)
)
BEGINDATA
10Accounting    Virginia,USA
```

This control file does not employ the FIELDS TERMINATED BY clause; rather, it uses POSITION to tell SQLLDR where fields begin and end. Of interest with the POSITION clause is that we could use overlapping positions, and go back and forth in the record. For example, if we were to alter the DEPT table as follows:

```
ops$tkyte@ORA10G> alter table dept add entire_line varchar(29);
Table altered.
```

and then use the following control file:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
( DEPTNO      position(1:2),
  DNAME       position(3:16),
  LOC         position(17:29),
  ENTIRE_LINE position(1:29)
)
BEGINDATA
10Accounting    Virginia,USA
```

the field ENTIRE_LINE is defined as POSITION(1:29). It extracts its data from all 29 bytes of input data, whereas the other fields are substrings of the input data. The outcome of the this control file will be as follows:

```
ops$tkyte@ORA10G> select * from dept;

    DEPTNO DNAME          LOC           ENTIRE_LINE
---------- -------------- ------------- -----------------------------
        10 Accounting     Virginia,USA  10Accounting    Virginia,USA
```

When using POSITION, we can use relative or absolute offsets. In the preceding example, we used absolute offsets. We specifically denoted where fields begin and where they end. We could have written the preceding control file as follows:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
( DEPTNO      position(1:2),
  DNAME       position(*:16),
  LOC         position(*:29),
  ENTIRE_LINE position(1:29)
)
BEGINDATA
10Accounting    Virginia,USA
```

The * instructs the control file to pick up where the last field left off. Therefore (*:16) is just the same as (3:16) in this case. Notice that you can mix relative and absolute positions in the control file. Additionally, when using the * notation, you can add to the offset. For example, if DNAME started 2 bytes *after* the end of DEPTNO, we could have used (*+2:16). In this example, the effect would be identical to using (5:16).

The ending position in the POSITION clause must be the absolute column position where the data ends. At times, it can be easier to specify just the length of each field, especially if they are contiguous, as in the preceding example. In this fashion, we would just have to tell SQLLDR the record starts at byte 1, and then specify the length of each field. This will save us from having to compute start and stop byte offsets into the record, which can be hard at times. In order to do this, we'll leave off the ending position and instead specify the *length* of each field in the fixed-length record as follows:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
( DEPTNO      position(1) char(2),
  DNAME       position(*) char(14),
  LOC         position(*) char(13),
  ENTIRE_LINE position(1) char(29)
)
BEGINDATA
10Accounting    Virginia,USA
```

Here we had to tell SQLLDR only where the first field begins and its length. Each subsequent field starts where the last one left off and continues for a specified length. It is not until the last field that we have to specify a position again, since this field goes back to the beginning of the record.

How Do I Load Dates?

Loading dates using SQLLDR is fairly straightforward, but it seems to be a common point of confusion. You simply need to use the DATE data type in the control file and specify the date mask to be used. This date mask is the same mask you use with TO_CHAR and TO_DATE in the database. SQLLDR will apply this date mask to your data and load it for you.

For example, if we alter our DEPT table again:

```
ops$tkyte@ORA10G> alter table dept add last_updated date;
Table altered.
```

we can load it with the following control file:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
(DEPTNO,
  DNAME,
  LOC,
  LAST_UPDATED date 'dd/mm/yyyy'
)
BEGINDATA
10,Sales,Virginia,1/5/2000
20,Accounting,Virginia,21/6/1999
30,Consulting,Virginia,5/1/2000
40,Finance,Virginia,15/3/2001
```

The resulting DEPT table will look like this:

```
ops$tkyte@ORA10G> select * from dept;

    DEPTNO DNAME          LOC           LAST_UPDA
---------- -------------- ------------- ---------
        10 Sales          Virginia      01-MAY-00
        20 Accounting     Virginia      21-JUN-99
        30 Consulting     Virginia      05-JAN-00
        40 Finance        Virginia      15-MAR-01
```

It is that easy. Just supply the format in the control file and SQLLDR will convert the date for us. In some cases, it might be appropriate to use a more powerful SQL function. For example, if your input file contains dates in many different formats: sometimes with the time component, sometimes without; sometimes in DD-MON-YYYY format; sometimes in DD/MM/YYYY format; and so on. You'll learn in the next section how to use functions in SQLLDR to overcome these challenges.

How Do I Load Data Using Functions?

In this section, you'll see how to refer to functions while loading data.

Using functions in SQLLDR is very easy once you understand how SQLLDR builds its INSERT statement. To have a function applied to a field in a SQLLDR script, simply add it to the control file in double quotes. For example, say you have the DEPT table from earlier, and you would like to make sure the data being loaded is in uppercase. You could use the following control file to load it:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
(DEPTNO,
  DNAME        "upper(:dname)",
  LOC          "upper(:loc)",
  LAST_UPDATED date 'dd/mm/yyyy'
)
BEGINDATA
10,Sales,Virginia,1/5/2000
20,Accounting,Virginia,21/6/1999
30,Consulting,Virginia,5/1/2000
40,Finance,Virginia,15/3/2001
```

The resulting data in the database will be as follows:

```
ops$tkyte@ORA10G> select * from dept;

DEPTNO DNAME          LOC           ENTIRE_LINE                   LAST_UPDA
------ -------------- ------------- ----------------------------- ---------
    10 SALES          VIRGINIA                                    01-MAY-00
    20 ACCOUNTING     VIRGINIA                                    21-JUN-99
    30 CONSULTING     VIRGINIA                                    05-JAN-00
    40 FINANCE        VIRGINIA                                    15-MAR-01
```

Notice how you are able to easily uppercase the data just by applying the UPPER function to a bind variable. It should be noted that the SQL functions could refer to any of the columns, regardless of the column the function is actually applied to. This means that a column can be the result of a function on two or more of the other columns. For example, if you wanted to load the column ENTIRE_LINE, you could use the SQL concatenation operator. It is a little more involved than that, though, in this case. Right now, the input data set has four data elements in it. If you were to simply add ENTIRE_LINE to the control file like this:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
(DEPTNO,
  DNAME        "upper(:dname)",
  LOC          "upper(:loc)",
  LAST_UPDATED date 'dd/mm/yyyy',
  ENTIRE_LINE  ":deptno||:dname||:loc||:last_updated"
)
BEGINDATA
```

```
10,Sales,Virginia,1/5/2000
20,Accounting,Virginia,21/6/1999
30,Consulting,Virginia,5/1/2000
40,Finance,Virginia,15/3/2001
```

you would find this error in your log file, for each input record:

```
Record 1: Rejected - Error on table DEPT, column ENTIRE_LINE.
Column not found before end of logical record (use TRAILING NULLCOLS)
```

Here, SQLLDR is telling you that it ran out of data in the record before it ran out of columns. The solution is easy in this case, and, in fact, SQLLDR even tells us what to do: use `TRAILING NULLCOLS`. This will have SQLLDR bind a `NULL` value in for that column if no data exists in the input record. In this case, adding `TRAILING NULLCOLS` will cause the bind variable `:ENTIRE_LINE` to be `NULL`. So, you retry with this control file:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
TRAILING NULLCOLS
(DEPTNO,
  DNAME        "upper(:dname)",
  LOC          "upper(:loc)",
  LAST_UPDATED date 'dd/mm/yyyy',
  ENTIRE_LINE  ":deptno||:dname||:loc||:last_updated"
)
BEGINDATA
10,Sales,Virginia,1/5/2000
20,Accounting,Virginia,21/6/1999
30,Consulting,Virginia,5/1/2000
40,Finance,Virginia,15/3/2001
```

Now the data in the table is as follows:

```
ops$tkyte@ORA10G> select * from dept;

DEPTNO DNAME          LOC           ENTIRE_LINE                   LAST_UPDA
------ -------------- ------------- ----------------------------- ---------
    10 SALES          VIRGINIA      10SalesVirginia1/5/2000       01-MAY-00
    20 ACCOUNTING     VIRGINIA      20AccountingVirginia21/6/1999 21-JUN-99
    30 CONSULTING     VIRGINIA      30ConsultingVirginia5/1/2000  05-JAN-00
    40 FINANCE        VIRGINIA      40FinanceVirginia15/3/2001    15-MAR-01
```

What makes this feat possible is the way SQLLDR builds its `INSERT` statement. SQLLDR will look at the preceding and see the `DEPTNO`, `DNAME`, `LOC`, `LAST_UPDATED`, and `ENTIRE_LINE` columns in the control file. It will set up five bind variables named after these columns. Normally, in the absence of any functions, the `INSERT` statement it builds is simply

```
INSERT INTO DEPT ( DEPTNO, DNAME, LOC, LAST_UPDATED, ENTIRE_LINE )
VALUES ( :DEPTNO, :DNAME, :LOC, :LAST_UPDATED, :ENTIRE_LINE );
```

It would then parse the input stream, assigning the values to its bind variables, and then execute the statement. When you begin to use functions, SQLLDR incorporates them into its `INSERT` statement. In the preceding example, the `INSERT` statement SQLLDR builds will look like this:

```
INSERT INTO T (DEPTNO, DNAME, LOC, LAST_UPDATED, ENTIRE_LINE)
VALUES ( :DEPTNO, upper(:dname), upper(:loc), :last_updated,
         :deptno||:dname||:loc||:last_updated );
```

It then prepares and binds the inputs to this statement, and executes it. So, pretty much anything you can think of doing in SQL, you can incorporate into your SQLLDR scripts. With the addition of the `CASE` statement in SQL, doing this can be extremely powerful and easy. For example, say your input file could have dates in the following formats:

- `HH24:MI:SS`: Just a time; the date should default to `SYSDATE`.
- `DD/MM/YYYY`: Just a date; the time should default to midnight.
- `HH24:MI:SS DD/MM/YYYY`: The date and time are both explicitly supplied.

You could use a control file like this:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
TRAILING NULLCOLS
(DEPTNO,
  DNAME        "upper(:dname)",
  LOC          "upper(:loc)",
  LAST_UPDATED
"case
 when length(:last_updated) > 9
 then to_date(:last_updated,'hh24:mi:ss dd/mm/yyyy')
 when instr(:last_updated,':') > 0
 then to_date(:last_updated,'hh24:mi:ss')
 else to_date(:last_updated,'dd/mm/yyyy')
 end"
)
BEGINDATA
10,Sales,Virginia,12:03:03 17/10/2005
20,Accounting,Virginia,02:23:54
30,Consulting,Virginia,01:24:00 21/10/2005
40,Finance,Virginia,17/8/2005
```

which results in the following:

```
ops$tkyte@ORA10G> alter session set nls_date_format = 'dd-mon-yyyy hh24:mi:ss';
Session altered.

ops$tkyte@ORA10G> select deptno, dname, loc, last_updated
  2  from dept;

    DEPTNO DNAME          LOC          LAST_UPDATED
---------- -------------- ------------ --------------------
        10 SALES          VIRGINIA     17-oct-2005 12:03:03
        20 ACCOUNTING     VIRGINIA     01-jul-2005 02:23:54
        30 CONSULTING     VIRGINIA     21-oct-2005 01:24:00
        40 FINANCE        VIRGINIA     17-aug-2005 00:00:00
```

Now, one of three date formats will be applied to the input character string (notice that you are *not* loading a DATE anymore; you are just loading a string). The CASE function will look at the length and the contents of the string to determine which of the masks it should use.

It is interesting to note that you can write your *own* functions to be called from SQLLDR. This is a straightforward application of the fact that PL/SQL can be called from SQL.

How Do I Load Data with Embedded Newlines?

This is something that has been problematic for SQLLDR historically: how to load free-form data that may include a newline in it. The newline character is the default end-of-line character to SQLLDR, and the ways around this did not offer much flexibility in the past. Fortunately, in Oracle 8.1.6 and later versions, we have some new options. The options for loading data with embedded newlines are now as follows:

- Load the data with some other character in the data that represents a newline (e.g., put the string \n in the text where a newline should appear) and use a SQL function to replace that text with a CHR(10) during load time.
- Use the FIX attribute on the INFILE directive, and load a fixed-length flat file.
- Use the VAR attribute on the INFILE directive, and load a variable-width file that uses a format such that the first few bytes of each line specify the length in bytes of the line to follow.
- Use the STR attribute on the INFILE directive to load a variable-width file with some sequence of characters that represents the end of line, as opposed to just the newline character representing this.

The following sections demonstrate each in turn.

Use a Character Other Than a Newline

This is an easy method if you have control over how the input data is produced. If it is easy enough to convert the data when creating the data file, this will work fine. The idea is to apply a SQL function to the data on the way into the database, replacing some string of characters with a newline. Let's add another column to our DEPT table:

```
ops$tkyte@ORA10G> alter table dept add comments varchar2(4000);
Table altered.
```

We'll use this column to load text into. An example control file with inline data could be as follows:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
TRAILING NULLCOLS
(DEPTNO,
  DNAME        "upper(:dname)",
  LOC          "upper(:loc)",
  COMMENTS     "replace(:comments,'\\n',chr(10))"
)
BEGINDATA
10,Sales,Virginia,This is the Sales\nOffice in Virginia
20,Accounting,Virginia,This is the Accounting\nOffice in Virginia
30,Consulting,Virginia,This is the Consulting\nOffice in Virginia
40,Finance,Virginia,This is the Finance\nOffice in Virginia
```

Notice how in the call to replace we had to use `\\n`, not just `\n`. This is because `\n` is recognized by SQLLDR as a newline, and SQLLDR would have converted it into a newline, not a two-character string. When we execute SQLLDR with the preceding control file, the table `DEPT` is loaded with the following:

```
ops$tkyte@ORA10G> select deptno, dname, comments from dept;

    DEPTNO DNAME          COMMENTS
---------- -------------- -------------------------
        10 SALES          This is the Sales
                          Office in Virginia

        20 ACCOUNTING     This is the Accounting
                          Office in Virginia

        30 CONSULTING     This is the Consulting
                          Office in Virginia

        40 FINANCE        This is the Finance
                          Office in Virginia
```

Use the FIX Attribute

The `FIX` attribute is another method available to us. If we use this, the input data must appear in fixed-length records. Each record will be exactly the same number of bytes as any other record in the input data set. When using *positional* data, use of the `FIX` attribute is especially valid. These files are typically fixed-length input files to begin with. When using free-form delimited data, it is less likely that we will have a fixed-length file, as these files are generally of varying length (this is the entire point of delimited files: to make each line only as big as it needs to be).

When using the `FIX` attribute, we must use an `INFILE` clause, as this is an option to `INFILE`. Additionally, the data must be stored externally, not in the control file itself, using this option. So, assuming we have fixed-length input records, we can use a control file such as this:

```
LOAD DATA
INFILE demo.dat "fix 80"
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
TRAILING NULLCOLS
(DEPTNO,
  DNAME        "upper(:dname)",
  LOC          "upper(:loc)",
  COMMENTS
)
```

This file specifies an input data file that will have records that are 80 bytes each. *This includes the trailing newline* that may or may not be there. In this case, the newline is nothing special in the input data file. It is just another character to be loaded or not. This is the thing to understand: the newline at the end of the record (if present) will become part of the record. To fully understand this, we need a utility to dump the contents of a file on the screen so we can see what is really in there. Using UNIX (or any Linux variant), this is pretty easy to do with `od`, a program to dump files to the screen in octal and other formats. We'll use the following `demo.dat` file. Note that the first column in the following output is actually in octal (base 8), so the number 0000012 on the second line is in octal and represents the decimal number 10. That tells us what byte in the file we are looking at. I've formatted the output to show ten characters per line (using `-w10`), so 0, 12, 24, and 36 are really 0, 10, 20, and 30.

```
[tkyte@desktop tkyte]$ od -c -w10 -v demo.dat
0000000   1   0   ,   S   a   l   e   s   ,   V
0000012   i   r   g   i   n   i   a   ,   T   h
0000024   i   s       i   s       t   h   e
0000036   S   a   l   e   s  \n   O   f   f   i
0000050   c   e       i   n       V   i   r   g
0000062   i   n   i   a
0000074
0000106
0000120   2   0   ,   A   c   c   o   u   n   t
0000132   i   n   g   ,   V   i   r   g   i   n
0000144   i   a   ,   T   h   i   s       i   s
0000156       t   h   e       A   c   c   o   u
0000170   n   t   i   n   g  \n   O   f   f   i
0000202   c   e       i   n       V   i   r   g
0000214   i   n   i   a
```

```
0000226
0000240   3   0   ,   C   o   n   s   u   l   t
0000252   i   n   g   ,   V   i   r   g   i   n
0000264   i   a   ,   T   h   i   s       i   s
0000276       t   h   e       C   o   n   s   u
0000310   l   t   i   n   g  \n   O   f   f   i
0000322   c   e       i   n       V   i   r   g
0000334   i   n   i   a
0000346
0000360   4   0   ,   F   i   n   a   n   c   e
0000372   ,   V   i   r   g   i   n   i   a   ,
0000404   T   h   i   s       i   s       t   h
0000416   e       F   i   n   a   n   c   e  \n
0000430   O   f   f   i   c   e       i   n
0000442   V   i   r   g   i   n   i   a
0000454
0000466
0000500
[tkyte@desktop tkyte]$
```

Notice that in this input file, the newlines (`\n`) are not there to indicate where the end of the record for SQLLDR is; rather, they are just data to be loaded in this case. SQLLDR is using the `FIX` width of 80 bytes to figure out how much data to read. In fact, if we look at the input data, the records for SQLLDR are not even terminated by `\n` in this input file. The character right before department 20's record is a space, not a newline.

Now that we know each and every record is 80 bytes long, we are ready to load it using the control file listed earlier with the `FIX 80` clause. When we do so, we can see the following:

```
ops$tkyte@ORA10G> select '"' || comments || '"' comments from dept;

COMMENTS
-------------------------------------------------------------------------------
"This is the Sales
Office in Virginia                          "

"This is the Accounting
Office in Virginia                "

"This is the Consulting
Office in Virginia                "

"This is the Finance
Office in Virginia                      "
```

You might need to "trim" this data, since the trailing whitespace is preserved. You can do that in the control file, using the `TRIM` built-in SQL function.

A word of caution to those of you lucky enough to work on both Windows and UNIX: the end-of-line marker is different on these platforms. On UNIX, it is simply \n (CHR(10) in SQL). On Windows NT, is it \r\n (CHR(13)||CHR(10) in SQL). In general, if you use the FIX approach, make sure to *create and load* the file on a homogenous platform (UNIX and UNIX, or Windows and Windows).

Use the VAR Attribute

Another method of loading data with embedded newline characters is to use the VAR attribute. When using this format, each record will begin with some fixed number of bytes that represent the total length of the incoming record. Using this format, we can load variable-length records that contain embedded newlines, but only if we have a *record length field* at the beginning of *each and every* record. So, if we use a control file such as this:

```
LOAD DATA
INFILE demo.dat "var 3"
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
TRAILING NULLCOLS
(DEPTNO,
  DNAME        "upper(:dname)",
  LOC          "upper(:loc)",
  COMMENTS
)
```

then the VAR 3 says that the first 3 bytes of each input record will be the length of that input record. If we take a data file such as the following:

```
[tkyte@desktop tkyte]$ cat demo.dat
05510,Sales,Virginia,This is the Sales
Office in Virginia
06520,Accounting,Virginia,This is the Accounting
Office in Virginia
06530,Consulting,Virginia,This is the Consulting
Office in Virginia
05940,Finance,Virginia,This is the Finance
Office in Virginia
[tkyte@desktop tkyte]$
```

we can load it using that control file. In our input data file, we have four rows of data. The first row starts with 055, meaning that the next 55 bytes represent the first input record. These 55 bytes include the terminating newline after the word Virginia. The next row starts with 065. It has 65 bytes of text, and so on. Using this format data file, we can easily load our data with embedded newlines.

Again, if you are using UNIX and Windows (the preceding example was with UNIX, where a newline is one character long), you would have to adjust the length field for each record. On Windows, the preceding example's .dat file would have to have 56, 66, 66, and 60 as the length fields.

Use the STR Attribute

This is perhaps the most flexible method of loading data with embedded newlines. Using the `STR` attribute, we can specify a new end-of-line character (or sequence of characters). This allows us to create an input data file that has some special character at the end of each line—the newline is no longer "special."

I prefer to use a sequence of characters, typically some special marker, and then a newline. This makes it easy to see the end-of-line character when viewing the input data in a text editor or some utility, as each record still has a newline at the end of it. The `STR` attribute is specified in hexadecimal, and perhaps the easiest way to get the exact hexadecimal string we need is to use SQL and `UTL_RAW` to produce the hexadecimal string for us. For example, assuming we are on UNIX where the end-of-line marker is `CHR(10)` (linefeed) and our special marker character is a pipe symbol (`|`), we can write this:

```
ops$tkyte@ORA10G> select utl_raw.cast_to_raw( '|'||chr(10) ) from dual;

UTL_RAW.CAST_TO_RAW('|'||CHR(10))
-------------------------------------------------------------------------------
7C0A
```

which shows us that the `STR` we need to use on UNIX is `X'7C0A'`.

■ **Note** On Windows, you would use `UTL_RAW.CAST_TO_RAW( '|'||chr(13)||chr(10) )`.

To use this, we might have a control file like this:

```
LOAD DATA
INFILE demo.dat "str X'7C0A'"
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ','
TRAILING NULLCOLS
(DEPTNO,
  DNAME        "upper(:dname)",
  LOC          "upper(:loc)",
  COMMENTS
)
```

So, if our input data looks like this:

```
[tkyte@desktop tkyte]$ cat demo.dat
10,Sales,Virginia,This is the Sales
Office in Virginia|
20,Accounting,Virginia,This is the Accounting
Office in Virginia|
30,Consulting,Virginia,This is the Consulting
```

```
Office in Virginia|
40,Finance,Virginia,This is the Finance
Office in Virginia|
[tkyte@desktop tkyte]$
```

where each record in the data file ends with a `|\n`, the previous control file will load it correctly.

Embedded Newlines Summary

We explored at least four ways to load data with embedded newlines in this section. In the upcoming section titled "Flat File Unload," we will use one of these techniques, the `STR` attribute, in a generic unload utility to avoid issues with regard to newlines in text.

Additionally, one thing to be very aware of—and I've mentioned it previously a couple of times—is that on Windows (all flavors), text files may end in `\r\n` (ASCII `13` + ASCII `10`, carriage return/linefeed). Your control file will have to accommodate this: that `\r` is part of the record. The byte counts in the `FIX` and `VAR`, and the string used with `STR` must accommodate this. For example, if you took any of the previous `.dat` files that currently contain just `\n` in them and FTP-ed them to Windows using an ASCII transfer (the default), every `\n` would turn into `\r\n`. The same control file that just worked in UNIX would not be able to load the data anymore. This is something you must be aware of and take into consideration when setting up the control file.

How Do I Load LOBs?

We will now consider some methods for loading into LOBs. This is not a `LONG` or `LONG RAW` field, but rather the preferred datatypes of `BLOB` and `CLOB`. These datatypes were introduced in Oracle 8.0 and later, and they support a much richer interface/set of functionality than the legacy `LONG` and `LONG RAW` types, as discussed in Chapter 12.

We will investigate two methods for loading these fields: SQLLDR and PL/SQL. Others exist, such as Java streams, Pro*C, and OCI. We will begin working with the PL/SQL method of loading LOBs, and then we'll look at using SQLLDR to load them as well.

Loading a LOB via PL/SQL

The `DBMS_LOB` package has entry points called `LoadFromFile`, `LoadBLOBFromFile`, and `LoadCLOBFromFile`. These procedures allow us to use a `BFILE` (which can be used to read operating system files) to populate a `BLOB` or `CLOB` in the database. There is not a significant difference between the `LoadFromFile` and `LoadBLOBFromFile` routines, other than the latter returns `OUT` parameters that indicate how far into the `BLOB` column we have loaded data. The `LoadCLOBFromFile` routine, however, provides a significant feature: character set conversion. If you recall, in Chapter 12 we discussed some of the National Language Support (NLS) features of the Oracle database and the importance of character sets. `LoadCLOBFromFile` allows us to tell the database that the file it is about to load is in a character set different from the one the database is using, and that it should perform the required character set conversion. For example, you may have a `UTF8`-compatible database, but the files received to be loaded are encoded in the `WE8ISO8859P1` character set, or vice versa. This function allows you to successfully load these files.

■ **Note** For complete details on the procedures available in the DBMS_LOB package and their full set of inputs and outputs, please refer to the Oracle9*i Oracle Supplied Packages Guide* and the Oracle 10*g Oracle PL/SQL Packages and Types Reference*.

To use these procedures, we will need to create a DIRECTORY object in the database. This object will allow us to create BFILES (and open them) that point to a file existing on the file system that the database server has access to. This last phrase, "that the database server has access to," is a key point when using PL/SQL to load LOBs. The DBMS_LOB package executes entirely in the server. It can see only the file systems the server can see. It cannot, in particular, see your local file system if you are accessing Oracle over the network.

So we need to begin by creating a DIRECTORY object in the database. This is a straightforward process. We will create two directories for this example (note that these examples are executed in a UNIX environment; you will use the syntax for referring to directories that is appropriate for your operating system):

```
ops$tkyte@ORA10G> create or replace directory dir1   as '/tmp/';
Directory created.

ops$tkyte@ORA10G> create or replace directory "dir2" as '/tmp/';
Directory created.
```

■ **Note** Oracle DIRECTORY objects are logical directories, meaning they are pointers to existing, physical directories in your operating system. The CREATE DIRECTORY command *does not* actually create a directory in the file system—you must perform that operation separately.

The user who performs this operation needs to have the CREATE ANY DIRECTORY privilege. The reason we create two directories is to demonstrate a common case-related ("case" as in uppercase versus lowercase characters) issue with regard to DIRECTORY objects. When Oracle created the first directory DIR1, it stored the object name in *uppercase* as it is the default. In the second example with dir2, it will have created the DIRECTORY object preserving the case we used in the name. The importance of this will be demonstrated shortly when we use the BFILE object.

Now, we want to load some data into either a BLOB or a CLOB. The method for doing so is rather easy, for example:

```
ops$tkyte@ORA10G> create table demo
  2  ( id        int primary key,
  3    theClob   clob
  4  )
  5  /
Table created.
```

```
ops$tkyte@ORA10G> host echo 'Hello World!' > /tmp/test.txt

ops$tkyte@ORA10G> declare
  2      l_clob    clob;
  3      l_bfile   bfile;
  4  begin
  5      insert into demo values ( 1, empty_clob() )
  6       returning theclob into l_clob;
  7
  8      l_bfile := bfilename( 'DIR1', 'test.txt' );
  9      dbms_lob.fileopen( l_bfile );
 10
 11      dbms_lob.loadfromfile( l_clob, l_bfile,
 12                             dbms_lob.getlength( l_bfile ) );
 13
 14      dbms_lob.fileclose( l_bfile );
 15  end;
 16   /
PL/SQL procedure successfully completed.

ops$tkyte@ORA10G> select dbms_lob.getlength(theClob), theClob from demo
  2  /
DBMS_LOB.GETLENGTH(THECLOB) THECLOB
--------------------------- ---------------
                         13 Hello World!
```

Walking through the preceding code we see

- On lines 5 and 6, we create a row in our table, set the `CLOB` to an `EMPTY_CLOB()`, and retrieve its value in one call. With the exception of temporary LOBs, LOBs "live" in the database—we cannot write to a LOB variable without having a pointer to either a temporary LOB or a LOB that is already in the database. An `EMPTY_CLOB()` is not a `NULL CLOB`; it is a valid non-`NULL` pointer to an empty structure. The other thing this did for us was to get a LOB locator, which points to data in a row that is locked. If we were to have selected this value out without locking the underlying row, our attempts to write to it would fail because LOBs must be locked prior to writing (unlike other structured data). By inserting the row, we have, of course, locked the row. If we were modifying an existing row instead of inserting, we would have used `SELECT FOR UPDATE` to retrieve and lock the row.

- On line 8, we create a `BFILE` object. Note how we use `DIR1` in uppercase—this is key, as we will see in a moment. This is because we are passing to `BFILENAME()` the *name* of an object, not the object itself. Therefore, we must ensure the name matches the case Oracle has stored for this object.

- On line 9, we open the LOB. This will allow us to read it.

- On lines 11 and 12, we load the entire contents of the operating system file /tmp/test.txt into the LOB locator we just inserted. We use DBMS_LOB.GETLENGTH() to tell the LOADFROMFILE() routine how many bytes of the BFILE to load (all of them).
- Lastly, on line 14, we close the BFILE we opened, and the CLOB is loaded.

If we had attempted to use dir1 instead of DIR1 in the preceding example, we would have encountered the following error:

```
ops$tkyte@ORA10G> declare
...
  6      returning theclob into l_clob;
  7
  8     l_bfile := bfilename( 'dir1', 'test.txt' );
  9     dbms_lob.fileopen( l_bfile );
...
 15  end;
 16  /
declare
*
ERROR at line 1:
ORA-22285: non-existent directory or file for FILEOPEN operation
ORA-06512: at "SYS.DBMS_LOB", line 523
ORA-06512: at line 9
```

This is because the directory dir1 does not exist—DIR1 does. If you prefer to use directory names in mixed case, you should use quoted identifiers when creating them as we did for dir2. This will allow you to write code as shown here:

```
ops$tkyte@ORA10G> declare
  2     l_clob    clob;
  3     l_bfile   bfile;
  4  begin
  5     insert into demo values ( 1, empty_clob() )
  6      returning theclob into l_clob;
  7
  8     l_bfile := bfilename( 'dir2', 'test.txt' );
  9     dbms_lob.fileopen( l_bfile );
 10
 11     dbms_lob.loadfromfile( l_clob, l_bfile,
 12                            dbms_lob.getlength( l_bfile ) );
 13
 14     dbms_lob.fileclose( l_bfile );
 15  end;
 16  /
PL/SQL procedure successfully completed.
```

There are methods other than the load from file routines by which you can populate a LOB using PL/SQL. Using DBMS_LOB and its supplied routines is by far the easiest if you are going to load the entire file. If you need to process the contents of the file while loading it, you may also use DBMS_LOB.READ on the BFILE to read the data. The use of UTL_RAW.CAST_TO_VARCHAR2 is handy here if the data you are reading is in fact text, not RAW. You may then use DBMS_LOB.WRITE or WRITEAPPEND to place the data into a CLOB or BLOB.

Loading LOB Data via SQLLDR

We will now investigate how to load data into a LOB via SQLLDR. There is more than one method for doing this, but we will investigate the two most common methods:

- When the data is "inline" with the rest of the data.
- When the data is stored out of line, and the input data contains a file name to be loaded with the row. These are also known as *secondary data files* (*SDFs*) in SQLLDR terminology.

We will start with data that is inline.

Loading LOB Data That Is Inline These LOBs will typically have newlines and other special characters embedded in them. Therefore, you will almost always use one of the four methods detailed in the "How Do I Load Data with Embedded Newlines?" section to load this data. Let's begin by modifying the DEPT table to have a CLOB instead of a big VARCHAR2 field for the COMMENTS column:

```
ops$tkyte@ORA10G> truncate table dept;
Table truncated.

ops$tkyte@ORA10G> alter table dept drop column comments;
Table altered.

ops$tkyte@ORA10G> alter table dept add comments clob;
Table altered.
```

For example, say we have a data file (demo.dat) that has the following contents:

```
10, Sales,Virginia,This is the Sales
Office in Virginia|
20,Accounting,Virginia,This is the Accounting
Office in Virginia|
30,Consulting,Virginia,This is the Consulting
Office in Virginia|
40,Finance,Virginia,"This is the Finance
Office in Virginia, it has embedded commas and is
much longer than the other comments field. If you
feel the need to add double quoted text in here like
this: ""You will need to double up those quotes!"" to
preserve them in the string. This field keeps going for up to
1000000 bytes (because of the control file definition I used)
or until we hit the magic end of record marker,
the | followed by an end of line - it is right here ->"|
```

Each record ends with a pipe symbol (|), followed by the end-of-line marker. The text for department 40 is much longer than the rest, with many newlines, embedded quotes, and commas. Given this data file, we can create a control file such as this:

```
LOAD DATA
INFILE demo.dat "str X'7C0A'"
INTO TABLE DEPT
REPLACE
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
TRAILING NULLCOLS
(DEPTNO,
  DNAME        "upper(:dname)",
  LOC          "upper(:loc)",
  COMMENTS     char(1000000)
)
```

■**Note** This example is from UNIX, where the end-of-line marker is 1 byte, hence the `STR` setting in the preceding control file. On Windows, it would have to be `'7C0D0A'`.

To load the data file, we specify `CHAR(1000000)` on the column `COMMENTS` since SQLLDR defaults to `CHAR(255)` for any input field. The `CHAR(1000000)` will allow SQLLDR to handle up to 1,000,000 bytes of input text. You must set this to a value that is larger than any expected chunk of text in the input file. Reviewing the loaded data, we see the following:

```
ops$tkyte@ORA10G> select comments from dept;

COMMENTS
-------------------------------------------------------------------------------
This is the Consulting
Office in Virginia

This is the Finance
Office in Virginia, it has embedded commas and is
much longer than the other comments field. If you
feel the need to add double quoted text in here like
this: "You will need to double up those quotes!" to
preserve them in the string. This field keeps going for up to
1000000 bytes or until we hit the magic end of record marker,
the | followed by an end of line - it is right here ->

This is the Sales
Office in Virginia

This is the Accounting
Office in Virginia
```

The one thing to observe here is that the doubled-up quotes are no longer doubled up. SQLLDR removed the extra quotes we placed there.

Loading LOB Data That Is Out of Line A common scenario is to have a data file that contains the names of files to load into the LOBs, instead of having the LOB data mixed in with the structured data. This offers a greater degree of flexibility, as the data file given to SQLLDR does not have to use one of the four methods to get around having embedded newlines in the input data, as would frequently happen with large amounts of text or binary data. SQLLDR calls this type of additional data file a `LOBFILE`.

SQLLDR can also support the loading of a structured data file that points to another, single data file. We can tell SQLLDR how to parse LOB data from this other file, so that each row in the structured data gets loaded with a piece of it. I find this mode to be of limited use (I've never found a use for it myself to date), and I will not discuss it here. SQLLDR refers to these externally referenced files as *complex secondary data files*.

`LOBFILE`s are relatively simple data files aimed at facilitating LOB loading. The attribute that distinguishes `LOBFILE`s from the main data files is that in `LOBFILE`s, there is no concept of a record, hence *newlines never get in the way*. In `LOBFILE`s, the data is in any of the following formats:

- Fixed-length fields (e.g., load bytes 100 through 1,000 from the `LOBFILE`)
- Delimited fields (terminated by something or enclosed by something)
- Length/value pairs, a variable-length field

The most common of these types is the delimited fields—ones that are terminated by an end-of-file (EOF), in fact. Typically, you have a directory full of files you would like to load into LOB columns, and each file in its entirety will go into a `BLOB`. The `LOBFILE` statement with `TERMINATED BY EOF` is what you will use.

Say we have a directory full of files we would like to load into the database. We would like to load the `OWNER` of the file, the `TIME_STAMP` of the file, the `NAME` of the file, and the file itself. The table we would load into would be as follows:

```
ops$tkyte@ORA10G> create table lob_demo
  2  ( owner      varchar2(255),
  3    time_stamp date,
  4    filename   varchar2(255),
  5    data       blob
  6  )
  7  /
Table created.
```

Using a simple `ls -l` on UNIX, and `dir /q /n` on Windows, and capturing that output, we can generate our input file and load it using a control file such as this on UNIX:

```
LOAD DATA
INFILE *
REPLACE
INTO TABLE LOB_DEMO
( owner      position(17:25),
  time_stamp  position(44:55) date "Mon DD HH24:MI",
  filename    position(57:100),
  data        LOBFILE(filename) TERMINATED BY EOF
)
BEGINDATA
-rw-r--r--    1 tkyte    tkyte     1220342 Jun 17 15:26 classes12.zip
-rw-rw-r--    1 tkyte    tkyte          10 Jul 16 16:38 foo.sql
-rw-rw-r--    1 tkyte    tkyte         751 Jul 16 16:36 t.ctl
-rw-rw-r--    1 tkyte    tkyte         491 Jul 16 16:38 testa.sql
-rw-rw-r--    1 tkyte    tkyte         283 Jul 16 16:38 testb.sql
-rw-rw-r--    1 tkyte    tkyte         231 Jul 16 16:38 test.sh
-rw-rw-r--    1 tkyte    tkyte         235 Apr 28 18:03 test.sql
-rw-rw-r--    1 tkyte    tkyte        1649 Jul 16 16:36 t.log
-rw-rw-r--    1 tkyte    tkyte        1292 Jul 16 16:38 uselast.sql
-rw-rw-r--    1 tkyte    tkyte         909 Jul 16 16:38 userbs.sql
```

Now, if we inspect the contents of the LOB_DEMO table after running SQLLDR, we will discover the following:

```
ops$tkyte@ORA10G> select owner, time_stamp, filename, dbms_lob.getlength(data)
  2  from lob_demo
  3  /

OWNER    TIME_STAM FILENAME       DBMS_LOB.GETLENGTH(DATA)
-------- --------- -------------- ------------------------
tkyte    17-JUN-05 classes12.zip                   1220342
tkyte    16-JUL-05 foo.sql                              10
tkyte    16-JUL-05 t.ctl                               875
tkyte    16-JUL-05 testa.sql                           491
tkyte    16-JUL-05 testb.sql                           283
tkyte    16-JUL-05 test.sh                             231
tkyte    28-APR-05 test.sql                            235
tkyte    16-JUL-05 t.log                                 0
tkyte    16-JUL-05 uselast.sql                        1292
tkyte    16-JUL-05 userbs.sql                          909

10 rows selected.
```

This works with CLOBs as well as BLOBs. Loading a directory of text files using SQLLDR in this fashion is easy.

Loading LOB Data into Object Columns Now that we know how to load into a simple table we have created ourselves, we might also find the need to load into a table that has a complex object type with a LOB in it. This happens most frequently when using the image capabilities. The image capabilities are implemented using a complex object type, ORDSYS.ORDIMAGE. We need to be able to tell SQLLDR how to load into this.

To load a LOB into an ORDIMAGE type column, we must understand a little more about the structure of the ORDIMAGE type. Using a table we want to load into, and a DESCRIBE on that table in SQL*Plus, we can discover that we have a column called IMAGE of type ORDSYS.ORDIMAGE, which we want to ultimately load into IMAGE.SOURCE.LOCALDATA. The following examples will work only if you have interMedia installed and configured; otherwise, the datatype ORDSYS.ORDIMAGE will be an unknown type:

```
ops$tkyte@ORA10G> create table image_load(
  2    id number,
  3    name varchar2(255),
  4    image ordsys.ordimage
  5  )
  6  /
Table created.

ops$tkyte@ORA10G> desc image_load
 Name                                    Null?    Type
 --------------------------------------- -------- ----------------------------
 ID                                               NUMBER
 NAME                                             VARCHAR2(255)
 IMAGE                                            ORDSYS.ORDIMAGE

ops$tkyte@ORA10G> desc ordsys.ordimage
 Name                                    Null?    Type
 --------------------------------------- -------- ----------------------------
 SOURCE                                           ORDSYS.ORDSOURCE
 HEIGHT                                           NUMBER(38)
 WIDTH                                            NUMBER(38)
 CONTENTLENGTH                                    NUMBER(38)
...

ops$tkyte@ORA10G> desc ordsys.ordsource
 Name                                    Null?    Type
 --------------------------------------- -------- ----------------------------
 LOCALDATA                                        BLOB
 SRCTYPE                                          VARCHAR2(4000)
 SRCLOCATION                                      VARCHAR2(4000)
...
```

■ **Note** You could issue SET DESC DEPTH ALL or SET DESC DEPTH <n> in SQL*Plus to have the entire hierarchy displayed at once. Given that the output for the ORDSYS.ORDIMAGE would have been many pages long, I chose to do it piece by piece.

So a control file to load this might look like this:

```
LOAD DATA
INFILE *
INTO TABLE image_load
REPLACE
FIELDS TERMINATED BY ','
( ID,
  NAME,
  file_name FILLER,
  IMAGE column object
  (
    SOURCE column object
    (
      LOCALDATA LOBFILE (file_name) TERMINATED BY EOF
                NULLIF file_name = 'NONE'
    )
  )
)
BEGINDATA
1,icons,icons.gif
```

I have introduced two new constructs here:

- `COLUMN OBJECT`: This tells SQLLDR that this is not a column name; rather, it is part of a column name. It is not mapped to a field in the input file, but is used to build the correct object column reference to be used during the load. In the preceding file, we have two column object tags, one nested in the other. Therefore, the column name that will be used is `IMAGE.SOURCE.LOCALDATA`, as we need it to be. Note that we are not loading any of the other attributes of these two object types (e.g., `IMAGE.HEIGHT`, `IMAGE.CONTENTLENGTH`, and `IMAGE.SOURCE.SRCTYPE`). Shortly, we'll see how to get those populated.

- `NULLIF FILE_NAME = 'NONE'`: This tells SQLLDR to load a `NULL` into the object column in the event that the field `FILE_NAME` contains the word `NONE` in it.

Once you have loaded an interMedia type, you will typically need to postprocess the loaded data using PL/SQL to have interMedia operate on it. For example, with the preceding data, you would probably want to run the following to have the properties for the image set up correctly:

```
begin
  for c in ( select * from image_load ) loop
    c.image.setproperties;
  end loop;
end;
/
```

`SETPROPERTIES` is an object method provided by the `ORDSYS.ORDIMAGE` type, which processes the image itself and updates the remaining attributes of the object with appropriate values.

How Do I Call SQLLDR from a Stored Procedure?

The short answer is that you can't do this. SQLLDR is not an API; it isn't something that is callable. SQLLDR is a command-line program. You can definitely write an external procedure in Java or C that runs SQLLDR, but that isn't the same as "calling" SQLLDR. The load will happen in another session, and it won't be subject to your transaction control. Additionally, you'll have to parse the resulting log file to determine if the load was successful or not, and how successful (i.e., how many rows got loaded before an error terminated the load) it may have been. Invoking SQLLDR from a stored procedure isn't something I recommend doing.

In the past, before Oracle9*i*, you might have implemented your own SQLLDR-like process. For example, the options could have been as follows:

- Write a mini-SQLLDR in PL/SQL. It can use either `BFILES` to read binary data or `UTL_FILE` to read text data to parse and load.
- Write a mini-SQLLDR in Java. This can be a little more sophisticated than a PL/SQL-based loader and can make use of the many available Java routines.
- Write a SQLLDR in C, and call it as an external procedure.

Fortunately, in Oracle9*i* and later, we have external tables that provide almost all of the functionality of SQLLDR and, additionally, can do many things SQLLDR cannot. We saw a quick example of external tables in the last chapter, where we used them to automate a parallel direct path load. We'll take a longer look at them in a moment. But first, I'd like to finish up our discussion of SQLLDR with some caveats.

SQLLDR Caveats

In this section, we will discuss some things to watch out for when using SQLLDR.

TRUNCATE Appears to Work Differently

The `TRUNCATE` option of SQLLDR might appear to work differently than `TRUNCATE` does in SQL*Plus, or any other tool. SQLLDR, working on the assumption you will be reloading the table with a similar amount of data, uses the extended form of `TRUNCATE`. Specifically, it issues the following:

```
truncate table t reuse storage
```

The `REUSE STORAGE` option does not release allocated extents—it just marks them as "free space." If this were not the desired outcome, you would truncate the table prior to executing SQLLDR.

SQLLDR Defaults to CHAR(255)

The default length of input fields is 255 characters. If your field is longer than this, you will receive an error message:

```
Record N: Rejected - Error on table T, column C.
Field in data file exceeds maximum length
```

This does not mean the data will not fit into the database column; rather, it indicates that SQLLDR was expecting 255 bytes or less of input data, and it received somewhat more than that. The solution is to simply use `CHAR(N)` in the control file, where `N` is big enough to accommodate the largest field length in the input file.

Command Line Overrides Control File

Many of the SQLLDR options may be either placed in the control file or used on the command line. For example, I can use `INFILE FILENAME` as well as `SQLLDR ... DATA=FILENAME`. The command line overrides any options in the control file. You cannot count on the options in a control file actually being used, as the person executing SQLLDR can override them.

SQLLDR Summary

In this section, we explored many areas of loading data. We covered the typical, everyday issues we will encounter: loading delimited files, loading fixed-length files, loading a directory full of image files, using functions on input data to transform the input, and so on. We did not cover massive data loads using the direct path loader in any detail; rather, we touched lightly on that subject. Our goal was to answer the questions that arise frequently with the use of SQLLDR and that affect the broadest audience.

External Tables

External tables were first introduced in Oracle9*i* Release 1. Put simply, they allow us to treat an operating system file as if it is a read-only database table. They are not intended to be a replacement for a "real" table, or to be used in place of a real table; rather, they are intended to be used as a tool to ease the loading and, in Oracle 10*g*, unloading of data.

When the external tables feature was first unveiled, I often referred to it as "the replacement for SQLLDR." This idea still holds true—*most* of the time. Having said this, you might wonder why we just spent so much time looking at SQLLDR. The reason is that SQLLDR has been around for a long time, and there are many, many legacy control files lying around. SQLLDR is still a commonly used tool; it is what many people know and have used. We are still in a period of transition from the use of SQLLDR to external tables, so SQLLDR is still very relevant.

What many DBAs don't realize is that their knowledge of SQLLDR control files is very readily transferable to the use of external tables. You'll discover, as we work through the examples in this part of the chapter, that external tables incorporate much of the SQLLDR syntax and many of the SQLLDR techniques.

SQLLDR should be chosen over external tables in the following three situations:

- You have to load data over a network—in other words, when the input file is not on the database server itself. One of the restrictions of external tables is that the input file must be accessible on the database server.
- Multiple users must *concurrently* work with the *same* external table processing different input files.
- You have to work with LOB types. External tables do not support LOBs.

With those three situations in mind, in general I strongly recommend using external tables for their extended capabilities. SQLLDR is a fairly simple tool that generates an `INSERT` statement and loads data. Its ability to use SQL is limited to calling SQL functions on a row-by-row basis. External tables open up the entire SQL set of functionality to data loading. Some of the key functionality features that external tables have over SQLLDR in my experience are as follows:

- The ability to use complex `WHERE` conditions to selectively load data. SQLLDR has a `WHEN` clause to select rows to load, but you are limited to using only `AND` expressions and expressions using equality—no ranges (greater than, less than), no `OR` expressions, no `IS NULL`, and so on.
- The ability to `MERGE` data. You can take an operating system file full of data and update existing database records from it.
- The ability to perform efficient code lookups. You can join an external table to other database tables as part of your load process.
- Easier multitable inserts using `INSERT`. Starting in Oracle9*i*, an `INSERT` statement can insert into one or *more* tables using complex `WHEN` conditions. While SQLLDR can load into multiple tables, it can be quite complex to formulate the syntax.
- A shallower learning curve for new developers. SQLLDR is "yet another tool" to learn, in addition to the programming language, the development tools, the SQL language, and so on. As long as a developer knows SQL, he can immediately apply that knowledge to bulk data loading, without having to learn a new tool (SQLLDR).

So, with that in mind, let's look at how to use external tables.

Setting Up External Tables

As a first simple demonstration of external tables, we'll rerun the previous SQLLDR example, which bulk loaded data into the `DEPT` table. Just to refresh your memory, the simple control file we used was as follows:

```
LOAD DATA
INFILE *
INTO TABLE DEPT
FIELDS TERMINATED BY ','
(DEPTNO, DNAME, LOC )
BEGINDATA
10,Sales,Virginia
20,Accounting,Virginia
30,Consulting,Virginia
40,Finance,Virginia
```

By far the easiest way to get started is to use this existing legacy control file to provide the definition of our external table. The following SQLLDR command will generate the `CREATE TABLE` statement for our external table:

```
[tkyte@desktop tkyte]$ sqlldr / demo1.ctl external_table=generate_only
SQL*Loader: Release 10.1.0.4.0 - Production on Sat Jul 16 17:34:51 2005
Copyright (c) 1982, 2004, Oracle.  All rights reserved.
[tkyte@desktop tkyte]$
```

The EXTERNAL_TABLE parameter has one of three values:

- NOT_USED: This is self-evident in meaning and is the default value.
- EXECUTE: This value means that SQLLDR will not generate a SQL INSERT statement and execute it; rather, it will create an external table and use a single bulk SQL statement to load it.
- GENERATE_ONLY: This value causes SQLLDR to not actually load any data, but only to generate the SQL DDL and DML statements it would have executed into the log file it creates.

■ **Caution** DIRECT=TRUE overrides EXTERNAL_TABLE=GENERATE_ONLY. If you specify DIRECT=TRUE, the data will be loaded and no external table will be generated.

When using GENERATE_ONLY, we can see the following in the demo1.log file:

```
CREATE DIRECTORY statements needed for files
------------------------------------------------------------------------
CREATE DIRECTORY SYS_SQLLDR_XT_TMPDIR_00000 AS '/home/tkyte'
```

We may or may not see a CREATE DIRECTORY statement in the log file. SQLLDR connects to the database during the external table script generation and queries the data dictionary to see if a suitable directory already exists. In this case, there was no suitable directory in place, so SQLLDR generated a CREATE DIRECTORY statement for us. Next, it generated the CREATE TABLE statement for our external table:

```
CREATE TABLE statement for external table:
------------------------------------------------------------------------
CREATE TABLE "SYS_SQLLDR_X_EXT_DEPT"
(
  "DEPTNO" NUMBER(2),
  "DNAME" VARCHAR2(14),
  "LOC" VARCHAR2(13)
)
```

SQLLDR had logged into the database; that is how it knows the exact datatypes to be used in this external table definition (e.g., that DEPTNO is a NUMBER(2)). It picked them up right from the data dictionary. Next, we see the beginning of the external table definition:

```
ORGANIZATION external
(
  TYPE oracle_loader
  DEFAULT DIRECTORY SYS_SQLLDR_XT_TMPDIR_00000
```

The ORGANIZATION EXTERNAL clause tells Oracle this is not a "normal" table. We saw this clause before in Chapter 10 when we looked at IOTs. Currently there are three organization types: HEAP for a "normal" table, INDEX for an IOT, and EXTERNAL for an external table. The rest of the text starts to tell Oracle more about the external table. The ORACLE_LOADER type is one of two supported types (in Oracle9*i* it is the *only* supported type). The other type is ORACLE_DATAPUMP, the proprietary Data Pump format used by Oracle in Oracle 10*g* and later. We will take a look at that type in a subsequent section on data unloading—it is a format that can be used to both load and unload data. An external table may be used both to create a Data Pump format file and to subsequently read it.

The very next section we encounter is the ACCESS PARAMETERS section of the external table. Here we describe to the database how to process the input file. As you look at this, you should notice the similarity to a SQLLDR control file; this is no accident. For the most part, SQLLDR and external tables use very similar syntax.

```
  ACCESS PARAMETERS
  (
    RECORDS DELIMITED BY NEWLINE CHARACTERSET WE8ISO8859P1
    BADFILE 'SYS_SQLLDR_XT_TMPDIR_00000':'demo1.bad'
    LOGFILE 'demo1.log_xt'
    READSIZE 1048576
    SKIP 7
    FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY '"' LDRTRIM
    REJECT ROWS WITH ALL NULL FIELDS
    (
      "DEPTNO" CHAR(255)
        TERMINATED BY "," OPTIONALLY ENCLOSED BY '"',
      "DNAME" CHAR(255)
        TERMINATED BY "," OPTIONALLY ENCLOSED BY '"',
      "LOC" CHAR(255)
        TERMINATED BY "," OPTIONALLY ENCLOSED BY '"'
    )
  )
```

These access parameters show how to set up an external table so that it processes files pretty much identically to the way SQLLDR would:

- RECORDS: Records are terminated by newlines by default, as they are for SQLLDR.
- BADFILE: There is a bad file (a file where records that fail processing are recorded to) set up in the directory we just created.
- LOGFILE: There is a log file that is equivalent to SQLLDR's log file set up in the current working directory.

- `READSIZE`: This is the default buffer used by Oracle to read the input data file. It is 1MB in this case. This memory comes from the PGA in dedicated server mode and the SGA in shared server mode, and it is used to buffer the information in the input data file for a session (refer to Chapter 4, where we discussed PGA and SGA memory). Keep that fact in mind if you're using shared servers: the memory is allocated from the SGA.
- `SKIP 7`: This determines how many records in the input file should be skipped. You might be asking, "Why 'skip 7'?" Well, we used `INFILE *` in this example; `SKIP 7` is used to skip over the control file itself to get to the embedded data. If we did not use `INFILE *`, there would be no `SKIP` clause at all.
- `FIELDS TERMINATED BY`: This is just as we used in the control file itself. However, the external table did add `LDRTRIM`, which stands for *LoaDeR TRIM*. This is a trim mode that emulates the way in which SQLLDR trims data by default. Other options include `LRTRIM`, `LTRIM`, and `RTRIM` for left/right trimming of whitespace; and `NOTRIM` to preserve all leading/trailing whitespace.
- `REJECT ROWS WITH ALL NULL FIELDS`: This causes the external table to log to the bad file any entirely blank lines and to not load that row.
- *The column definitions themselves*: This is the metadata about the expected input data values. They are all character strings in the data file to be loaded, and they can be up to 255 characters in length (SQLLDR's default size), and terminated by `,` and optionally enclosed in quotes.

■ **Note** For a comprehensive list of all options available to you when using external tables, review the *Oracle Utilities Guide* manual. This reference contains a section dedicated to external tables. The *Oracle SQL Reference Guide* manual provides the basic syntax, but not the details of the `ACCESS PARAMETERS` section.

Lastly, we get to the `LOCATION` section of the external table definition:

```
  location
  (
    'demo1.ctl'
  )
) REJECT LIMIT UNLIMITED
```

That tells Oracle the name of the file to load, which is `demo1.ctl` in this case since we used `INFILE *` in the original control file. The next statement in the control file is the default `INSERT` that can be used to load the table from the external table itself:

```
INSERT statements used to load internal tables:
------------------------------------------------------------------------
INSERT /*+ append */ INTO DEPT
(
  DEPTNO,
  DNAME,
  LOC
)
SELECT
  "DEPTNO",
  "DNAME",
  "LOC"
FROM "SYS_SQLLDR_X_EXT_DEPT"
```

That would perform the logical equivalent of a direct path load if possible (assuming the APPEND hint may be obeyed; the existence of triggers or foreign key constraints may prevent the direct path operation from taking place).

Lastly, in the log file, we'll see statements that may be used to remove the objects SQLLDR would have us create after the load was complete:

```
statements to cleanup objects created by previous statements:
------------------------------------------------------------------------
DROP TABLE "SYS_SQLLDR_X_EXT_DEPT"
DROP DIRECTORY SYS_SQLLDR_XT_TMPDIR_00000
```

And that is it. If we take that log file and insert / where appropriate to make it a valid SQL*Plus script, then we should be ready to go—or not, depending on the permissions in place. For example, assuming the schema I log into has the CREATE ANY DIRECTORY privilege or READ access to an existing directory, I might observe the following:

```
ops$tkyte@ORA10G> INSERT /*+ append */ INTO DEPT
  2  (
  3    DEPTNO,
  4    DNAME,
  5    LOC
  6  )
  7  SELECT
  8    "DEPTNO",
  9    "DNAME",
 10    "LOC"
 11  FROM "SYS_SQLLDR_X_EXT_DEPT"
 12  /
INSERT /*+ append */ INTO DEPT
*
ERROR at line 1:
ORA-29913: error in executing ODCIEXTTABLEOPEN callout
ORA-29400: data cartridge error
KUP-04063: unable to open log file demo1.log_xt
OS error Permission denied
ORA-06512: at "SYS.ORACLE_LOADER", line 19
ORA-06512: at line 1
```

Well, that doesn't seem right at first. I'm logged into the operating system as TKYTE, the directory I'm logging into is /home/tkyte, and I own that directory, so I can certainly write to it (I created the SQLLDR log file there, after all!). What happened? The fact is that the external table code is running in the Oracle server software, in my dedicated or shared server. The process trying to read the input data file is the Oracle software owner, not my account. The process trying to create the log file is the Oracle software owner, not my account. Apparently, Oracle did not have the privilege required to write into my directory, and hence the attempted access of the external table failed. This is an important point. To read a table, the account under which the database is running (the Oracle software owner) must be able to do the following:

- *Read* the file we are pointing to. In UNIX, that means the Oracle software owner must have read and execute permissions on all directory paths leading to the file. In Windows, the Oracle software owner must be able to read that file.
- *Write* to the directories where the log file will be written to (or bypass the generation of the log file altogether, but this is not recommended in general). In fact, if the log file already exists, the Oracle software owner must be able to write to the existing file.
- *Write* to any of the bad files we have specified, just like the log file.

Returning to the example, the following command gives Oracle the ability to write into my directory:

```
ops$tkyte@ORA10G> host chmod a+rw .
```

■ **Caution** This command actually gives everyone the ability to write into my directory! This is just a demonstration; normally I would use a special directory perhaps owned by the Oracle software owner itself to do this.

Next, I rerun my `INSERT` statement:

```
ops$tkyte@ORA10G> l
  1  INSERT /*+ append */ INTO DEPT
  2  (
  3    DEPTNO,
  4    DNAME,
  5    LOC
  6  )
  7  SELECT
  8    "DEPTNO",
  9    "DNAME",
 10    "LOC"
 11* FROM "SYS_SQLLDR_X_EXT_DEPT"
ops$tkyte@ORA10G> /
4 rows created.

ops$tkyte@ORA10G> host ls -l demo1.log_xt
-rw-r--r--    1 ora10g   ora10g        578 Jul 17 10:45 demo1.log_xt
```

You can see that this time I accessed the file, I successfully loaded four rows, and the log file was created and, in fact, is owned by “Oracle,” not by my operating system account.

Dealing with Errors

In a perfect world, there would be no errors. The data in the input file would be perfect, and it would all load correctly. That almost never happens. So, how can we track errors with this process?

The most common method is to use the `BADFILE` option. Here, Oracle will record all records that failed processing. For example, if our control file contained a record with `DEPTNO 'ABC'`, that record would fail and end up in the bad file because `'ABC'` cannot be converted into a number. We’ll demonstrate this in the following example.

First, we add the following as the last line of `demo1.ctl` (this will add a line of data that cannot be loaded to our input):

```
ABC,XYZ,Hello
```

Next, we run the following command, to prove that the `demo1.bad` file does not exist yet:

```
ops$tkyte@ORA10G> host ls -l demo1.bad
ls: demo1.bad: No such file or directory
```

Then we query the external table to display the contents:

```
ops$tkyte@ORA10G> select * from SYS_SQLLDR_X_EXT_DEPT;

    DEPTNO DNAME          LOC
---------- -------------- -------------
        10 Sales          Virginia
        20 Accounting     Virginia
        30 Consulting     Virginia
        40 Finance        Virginia
```

Now we will find that the file exists and we can review its contents:

```
ops$tkyte@ORA10G> host ls -l demo1.bad
-rw-r--r--    1 ora10g   ora10g         14 Jul 17 10:53 demo1.bad

ops$tkyte@ORA10G> host cat demo1.bad
ABC,XYZ,Hello
```

But how can we programmatically inspect these bad records and the log that is generated? Fortunately, that is easy to do by using yet another external table. Suppose we set up this external table:

```
ops$tkyte@ORA10G> create table et_bad
  2  ( text1 varchar2(4000) ,
  3    text2 varchar2(4000) ,
  4    text3 varchar2(4000)
  5  )
  6  organization external
```

```
(type oracle_loader
 default directory SYS_SQLLDR_XT_TMPDIR_00000
 access parameters
 (
   records delimited by newline
   fields
   missing field values are null
   ( text1 position(1:4000),
     text2 position(4001:8000),
     text3 position(8001:12000)
   )
 )
 location ('demo1.bad')
)
/
Table created.
```

This is just a table that can read any file without failing on a datatype error, as long as the lines in the file consist of fewer than 12,000 characters. If have more than 12,000 characters, then we can simply add more text columns to accommodate them.

We can clearly see the rejected records via a simple query:

```
ops$tkyte@ORA10G> select * from et_bad;

TEXT1           TEXT2           TEXT3
--------------- --------------- ---------------
ABC,XYZ,Hello
```

A COUNT(*) could tell us how many records were rejected. Another external table created on the log file associated with this external table could tell us why the record was rejected. We would need to go one step further to make this a repeatable process, however. The reason is that the bad file is not "blanked" out if there were no errors in our use of the external table. So, if there were some preexisting bad file with data in it and our external table generated no errors, we would be misled into thinking there were errors.

I've taken three approaches in the past to resolve this issue:

- Use UTL_FILE and reset the bad file—truncate it, in effect, by simply opening it for write and closing it.
- Use UTL_FILE to rename any preexisting bad files, preserving their contents, but allowing us to create a new one.
- Incorporate the PID into the bad (and log) file names. We'll demonstrate this later in the "Multiuser Issues" section.

In that fashion, we'll be able to tell if the bad records in the bad file were generated by us just recently or if they were left over from some older version of the file itself and are not meaningful.

ALTER TABLE T PROJECT COLUMN REFERENCED | ALL

The COUNT(*) earlier in this section made me think about a new feature in Oracle 10*g*: the ability to optimize external table access by only accessing the fields in the external file that are referenced in the query. That is, if the external table is defined to have 100 number fields, but you select out only one of them, you can direct Oracle to bypass converting the other 99 strings into numbers. It sounds great, but it can cause a different number of rows to be returned from each query. Suppose the external table has 100 lines of data in it. All of the data for column C1 is "valid" and converts into a number. None of the data for column C2 is "valid," and it does not convert into a number. If you select C1 from that external table, you'll get 100 rows back. If you select C2 from that external table, you'll get 0 rows back.

You have to explicitly enable this optimization, and you should think about whether it is "safe" for you to use or not (only you know enough about your application and its processing to answer the question "Is it safe?"). Using the earlier example with the bad line of data added, we would expect to see the following output upon querying our external table:

```
ops$tkyte@ORA10G> select dname
  2    from SYS_SQLLDR_X_EXT_DEPT
  3  /

DNAME
--------------
Sales
Accounting
Consulting
Finance

ops$tkyte@ORA10G> select deptno
  2    from SYS_SQLLDR_X_EXT_DEPT
  3  /

    DEPTNO
----------
        10
        20
        30
        40
```

We know the "bad" record has been logged into the BADFILE. But if we simply ALTER the external table and tell Oracle to only "project" (process) the referenced columns, as follows:

```
ops$tkyte@ORA10G> alter table SYS_SQLLDR_X_EXT_DEPT
  2  project column referenced
  3  /

Table altered.
```

```
ops$tkyte@ORA10G> select dname
  2    from SYS_SQLLDR_X_EXT_DEPT
  3  /

DNAME
--------------
Sales
Accounting
Consulting
Finance
XYZ

ops$tkyte@ORA10G> select deptno
  2    from SYS_SQLLDR_X_EXT_DEPT
  3  /

    DEPTNO
----------
        10
        20
        30
        40
```

we get different numbers of rows from each query. The `DNAME` field was valid for every single record in the input file, but the `DEPTNO` column was not. If we do not retrieve the `DEPTNO` column, it does not fail the record—the resultset is materially changed.

Using an External Table to Load Different Files

A common need is to use an external table to load data from differently named files over a period of time. That is, this week we must load `file1.dat`, and next week it will be `file2.dat`, and so on. So far, we've been loading from a fixed file name, `demo1.dat`. What if we need subsequently to load from a second file, `demo2.dat`?

Fortunately, that is pretty easy to accommodate. The `ALTER TABLE` command may be used to repoint the location setting of the external table:

```
ops$tkyte@ORA10G> alter table SYS_SQLLDR_X_EXT_DEPT
   2  location( 'demo2.dat' );
Table altered.
```

And that would pretty much be it—the very next query of that external table would have it accessing the file `demo2.dat`.

Multiuser Issues

In the introduction to this section, I described three situations where external tables might not be as useful as SQLLDR. One of them was a specific multiuser issue. We just saw how to change the location of an external table—how to make it read from file 2 instead of file 1, and so on. The problem arises when multiple users each try to concurrently use that external table and have it point to different files for each session.

This cannot be done. The external table will point to a single file (or set of files) at any given time. If I log in and alter the table to point to file 1 and you do the same at about the same time, and then we both query that table, we'll both be processing the same file.

Generally, this issue is not one that you should encounter. External tables are not a replacement for "database tables"; they are a means to load data, and as such you would not use them on a daily basis as part of your application. They are generally a DBA or developer tool used to load information, either as a one-time event or on a recurring basis, as in a data warehouse load. If the DBA has ten files to load into the database using the same external table, she would *not* do them sequentially—that is, pointing the external file to file 1 and processing it, then file 2 and processing it, and so on. Rather, she would simply point the external table to *both files* and let the database process both of them:

```
ops$tkyte@ORA10G> alter table SYS_SQLLDR_X_EXT_DEPT
  2  location( 'file1.dat', 'file2.dat')
  3  /
Table altered.
```

If "parallel processing" is required, then the database already has the built-in ability to do this, as demonstrated in the last chapter.

So the only multiuser issue would be if two sessions both tried to alter the location at about the same time—and this is just a possibility to be aware of, not something I believe you'll actually run into very often.

Another multiuser consideration is that of the bad and log file names. What if you have many sessions concurrently looking at the same external table, or using parallel processing (which in some respects is a multiuser situation)? It would be nice to be able to segregate these files by session, and fortunately you can do that. You may incorporate the following special strings:

- `%p`: PID.
- `%a`: Parallel execution servers agent ID. The parallel execution servers have the numbers 001, 002, 003, and so on assigned to them.

In this fashion, each session will tend to generate its own bad and log files. For example, if you used the following `BADFILE` syntax in the `CREATE TABLE` from earlier:

```
    RECORDS DELIMITED BY NEWLINE CHARACTERSET WE8ISO8859P1
    BADFILE 'SYS_SQLLDR_XT_TMPDIR_00000':'demo1_%p.bad'
    LOGFILE 'demo1.log_xt'
```

you would expect to find a file named similarly to the following:

```
$ ls *.bad
demo1_7108.bad
```

However, you still might have issues over lengthy periods of time. The PIDs will be reused on most operating systems. So the techniques outlined in dealing with errors may well still be relevant—you'll need to reset your bad file or rename it if it exists and if you determine this to be an issue.

External Tables Summary

In this section, we explored external tables. They are a new feature of Oracle9*i* and later that may, for the most part, replace SQLLDR. We investigated the quickest way to get going with external tables: the technique of using SQLLDR to convert the control files we have from past experiences. We demonstrated some techniques for detecting and handling errors via the bad files and, lastly, we explored some multiuser issues regarding external tables.

We are now ready to get into the last sections in this chapter, which deal with unloading data from the database.

Flat File Unload

One thing SQLLDR does not do, and that Oracle supplies no command-line tools for, is unload data in a format understandable by SQLLDR or other programs. This would be useful for moving data from system to system without using `EXP`/`IMP` or `EXPDP`/`IMPDP` (the new Data Pump replacements for `EXP` and `IMP`). Using `EXP(DP)`/`IMP(DP)` to move data from system to system works fine—as long as both systems are Oracle.

■ **Note** HTML DB provides a data export feature as part of its SQL Workshop. You may export the information in a CSV format easily. This works well for a few megabytes of information, but it is not appropriate for many tens of megabytes or more.

We will develop a small PL/SQL utility that may be used to unload data on a server in a SQLLDR-friendly format. Also, equivalent tools for doing so in Pro*C and SQL*Plus are provided on the Ask Tom web site at `http://asktom.oracle.com/~tkyte/flat/index.html`. The PL/SQL utility will work fine in most small cases, but better performance will be had using Pro*C. Pro*C and SQL*Plus are also useful if you need the files to be generated on the client and not on the server, which is where PL/SQL will create them.

The specification of the package we will create is as follows:

```
ops$tkyte@ORA10G> create or replace package unloader
AUTHID CURRENT_USER
as
/* Function run -- unloads data from any query into a file
                   and creates a control file to reload that
                   data into another table

    p_query      = SQL query to "unload".  May be virtually any query.
    p_tname      = Table to load into.  Will be put into control file.
    p_mode       = REPLACE|APPEND|TRUNCATE -- how to reload the data
    p_dir        = directory we will write the ctl and dat file to.
    p_filename   = name of file to write to.  I will add .ctl and .dat
                   to this name
    p_separator  = field delimiter.  I default this to a comma.
    p_enclosure  = what each field will be wrapped in
    p_terminator = end of line character.  We use this so we can unload
              and reload data with newlines in it.  I default to
             "|\n" (a pipe and a newline together) and "|\r\n" on NT.
              You need only to override this if you believe your
              data will have that sequence in it. I ALWAYS add the
              OS "end of line" marker to this sequence, you should not
    */
    function run( p_query     in varchar2,
                  p_tname     in varchar2,
                  p_mode      in varchar2 default 'REPLACE',
                  p_dir       in varchar2,
                  p_filename  in varchar2,
                  p_separator in varchar2 default ',',
                  p_enclosure in varchar2 default '"',
                  p_terminator in varchar2 default '|' )
    return number;
end;
/
Package created.
```

Note the use of AUTHID CURRENT_USER. This permits this package to be installed *once* in a database and used by anyone to unload data. All the person needs is SELECT privileges on the table(s) he wants to unload and EXECUTE privileges on this package. If we did not use AUTHID CURRENT_USER in this case, then the owner of this package would need direct SELECT privileges on all tables to be unloaded.

■ **Note** The SQL will execute with the privileges of the invoker of this routine. However, all PL/SQL calls will run with the privileges of the *definer* of the called routine; therefore, the ability to use UTL_FILE to write to a directory is implicitly given to anyone with execute permission on this package.

The package body follows. We use UTL_FILE to write a control file and a data file. DBMS_SQL is used to dynamically process any query. We use one datatype in our queries: a VARCHAR2(4000). This implies we cannot use this method to unload LOBs, and that is true if the LOB is greater than 4,000 bytes. We can, however, use this to unload up to 4,000 bytes of any LOB using DBMS_LOB.SUBSTR. Additionally, since we are using a VARCHAR2 as the only output data type, we can handle RAWs up to 2,000 bytes in length (4,000 hexadecimal characters), which is sufficient for everything except LONG RAWs and LOBs. Additionally, any query that references a nonscalar attribute (a complex object type, nested table, and so on) will not work with this simple implementation. The following is a 90 percent solution, meaning it solves the problem 90 percent of the time.

```
ops$tkyte@ORA10G> create or replace package body unloader
as


g_theCursor     integer default dbms_sql.open_cursor;
g_descTbl       dbms_sql.desc_tab;
g_nl            varchar2(2) default chr(10);

```

These are some global variables used in this package body. The global cursor is opened once, the first time we reference this package, and it will stay open until we log out. This avoids the overhead of getting a new cursor every time we call this package. The `G_DESCTBL` is a PL/SQL table that will hold the output of a `DBMS_SQL.DESCRIBE` call. `G_NL` is a newline character. We use this in strings that need to have newlines embedded in them. We do not need to adjust this for Windows—`UTL_FILE` will see the `CHR(10)` in the string of characters and automatically turn that into a carriage return/linefeed for us.

Next, we have a small convenience function used to convert a character to hexadecimal. It uses the built-in functions to do this:

```

function to_hex( p_str in varchar2 ) return varchar2
is
begin
    return to_char( ascii(p_str), 'fm0x' );
end;

```

Finally, we create one more convenience function, `IS_WINDOWS`, that returns `TRUE` or `FALSE` depending on if we are on the Windows platform, and therefore the end of line is a two-character string instead of the single character it is on most other platforms. We are using the built-in `DBMS_UTILITY` function, `GET_PARAMETER_VALUE`, which can be used to read almost any parameter. We retrieve the `CONTROL_FILES` parameter and look for the existence of a \ in it—if we find one, we are on Windows:

```
function is_windows return boolean
is
        l_cfiles varchar2(4000);
        l_dummy  number;
begin
 if (dbms_utility.get_parameter_value( 'control_files', l_dummy, l_cfiles )>0)
 then
       return instr( l_cfiles, '\' ) > 0;
 else
       return FALSE;
 end if;
end;
```

■**Note** The `IS_WINDOWS` function does rely on you using `\` in your `CONTROL_FILES` parameter. Be aware that you may use `/`, but it would be highly unusual.

The following is a procedure to create a control file to reload the unloaded data, using the `DESCRIBE` table generated by `DBMS_SQL.DESCRIBE_COLUMNS`. It takes care of the operating system specifics for us, such as whether the operating system uses a carriage return/linefeed (this is used for the `STR` attribute):

```

procedure  dump_ctl( p_dir        in varchar2,
                     p_filename   in varchar2,
                     p_tname      in varchar2,
                     p_mode       in varchar2,
                     p_separator  in varchar2,
                     p_enclosure  in varchar2,
                     p_terminator in varchar2 )
is
    l_output        utl_file.file_type;
    l_sep           varchar2(5);
    l_str           varchar2(5) := chr(10);

begin
    if ( is_windows )
    then
        l_str := chr(13) || chr(10);
    end if;

    l_output := utl_file.fopen( p_dir, p_filename || '.ctl', 'w' );

    utl_file.put_line( l_output, 'load data' );
    utl_file.put_line( l_output, 'infile ''' ||
                                  p_filename || '.dat'' "str x''' ||
                                  utl_raw.cast_to_raw( p_terminator ||
                                  l_str ) || '''"' );
    utl_file.put_line( l_output, 'into table ' || p_tname );
    utl_file.put_line( l_output, p_mode );
    utl_file.put_line( l_output, 'fields terminated by X''' ||
                                  to_hex(p_separator) ||
                                 ''' enclosed by X''' ||
                                  to_hex(p_enclosure) || ''' ' );
    utl_file.put_line( l_output, '(' );

    for i in 1 .. g_descTbl.count
    loop
        if ( g_descTbl(i).col_type = 12 )
        then
            utl_file.put( l_output, l_sep || g_descTbl(i).col_name ||
                               ' date ''ddmmyyyyhh24miss'' ');
        else
            utl_file.put( l_output, l_sep || g_descTbl(i).col_name ||
                          ' char(' ||
                          to_char(g_descTbl(i).col_max_len*2) ||' )' );
        end if;
        l_sep := ','||g_nl ;
    end loop;
    utl_file.put_line( l_output, g_nl || ')' );
    utl_file.fclose( l_output );
end;

```

Here is a simple function to return a quoted string using the chosen enclosure character. Notice how it not only encloses the character, but also doubles up the enclosure character if it exists in the string as well, so that they are preserved:

```
function quote(p_str in varchar2, p_enclosure in varchar2)
         return varchar2
is
begin
    return p_enclosure ||
           replace( p_str, p_enclosure, p_enclosure||p_enclosure ) ||
           p_enclosure;
end;

```

Next we have the main function, RUN. As it is fairly large, I'll comment on it as we go along:

```
function run( p_query        in varchar2,
              p_tname     in varchar2,
              p_mode      in varchar2 default 'REPLACE',
              p_dir       in varchar2,
              p_filename  in varchar2,
              p_separator in varchar2 default ',',
              p_enclosure in varchar2 default '"',
              p_terminator in varchar2 default '|' ) return number
is
    l_output        utl_file.file_type;
    l_columnValue   varchar2(4000);
    l_colCnt        number default 0;
    l_separator     varchar2(10) default '';
    l_cnt           number default 0;
    l_line          long;
    l_datefmt       varchar2(255);
    l_descTbl       dbms_sql.desc_tab;
begin
```

We will save the NLS_DATE_FORMAT into a variable so we can change it to a format that preserves the date and time when dumping the data to disk. In this fashion, we will preserve the time component of a date. We then set up an exception block so that we can reset the NLS_DATE_FORMAT upon any error:

```
select value
  into l_datefmt
  from nls_session_parameters
 where parameter = 'NLS_DATE_FORMAT';

/*
   Set the date format to a big numeric string. Avoids
   all NLS issues and saves both the time and date.
*/
execute immediate
   'alter session set nls_date_format=''ddmmyyyyhh24miss'' ';

/*
   Set up an exception block so that in the event of any
   error, we can at least reset the date format.
*/
begin
```

Next we will parse and describe the query. The setting of G_DESCTBL to L_DESCTBL is done to “reset” the global table; otherwise, it might contain data from a previous DESCRIBE, in addition to data for the current query. Once we have done that, we call DUMP_CTL to actually create the control file:

```
    /*
       Parse and describe the query. We reset the
       descTbl to an empty table so .count on it
       will be reliable.
    */
    dbms_sql.parse( g_theCursor, p_query, dbms_sql.native );
    g_descTbl := l_descTbl;
    dbms_sql.describe_columns( g_theCursor, l_colCnt, g_descTbl );

    /*
       Create a control file to reload this data
       into the desired table.
    */
    dump_ctl( p_dir, p_filename, p_tname, p_mode, p_separator,
                     p_enclosure, p_terminator );

    /*
       Bind every single column to a varchar2(4000). We don't care
       if we are fetching a number or a date or whatever.
       Everything can be a string.
    */
```

We are ready to dump the actual data out to disk. We begin by defining every column to be a VARCHAR2(4000) for fetching into. All NUMBERs, DATEs, RAWs—every type will be converted into VARCHAR2. Immediately after this, we execute the query to prepare for the fetching phase:

```
for i in 1 .. l_colCnt loop
   dbms_sql.define_column( g_theCursor, i, l_columnValue, 4000);
end loop;

/*
   Run the query - ignore the output of execute. It is only
   valid when the DML is an insert/update or delete.
*/
```

Now we open the data file for writing, fetch all of the rows from the query, and print it out to the data file:

```
l_cnt := dbms_sql.execute(g_theCursor);

/*
   Open the file to write output to and then write the
   delimited data to it.
*/
l_output := utl_file.fopen( p_dir, p_filename || '.dat', 'w',
                                   32760 );
loop
    exit when ( dbms_sql.fetch_rows(g_theCursor) <= 0 );
    l_separator := '';
    l_line := null;
    for i in 1 .. l_colCnt loop
        dbms_sql.column_value( g_theCursor, i,
                               l_columnValue );
        l_line := l_line || l_separator ||
                   quote( l_columnValue, p_enclosure );
        l_separator := p_separator;
    end loop;
    l_line := l_line || p_terminator;
    utl_file.put_line( l_output, l_line );
    l_cnt := l_cnt+1;
end loop;
utl_file.fclose( l_output );

```

Lastly, we set the date format back (and the exception block will do the same if any of the preceding code fails for any reason) to what it was and return:

```
/*
   Now reset the date format and return the number of rows
   written to the output file.
*/
```

```
        execute immediate
           'alter session set nls_date_format=''' || l_datefmt || '''';
        return l_cnt;
exception
        /*
           In the event of ANY error, reset the data format and
           re-raise the error.
        */
        when others then
           execute immediate
           'alter session set nls_date_format=''' || l_datefmt || '''';
           RAISE;
    end;
end run;


end unloader;
/
Package body created.
```

To run this, we can simply use the following (note that the following does, of course, require that you have SELECT on SCOTT.EMP granted to one of your roles or to yourself directly):

```
ops$tkyte@ORA10G> set serveroutput on

ops$tkyte@ORA10G> create or replace directory my_dir as '/tmp';
Directory created.

ops$tkyte@ORA10G> declare
    l_rows    number;
begin
    l_rows := unloader.run
              ( p_query      => 'select * from scott.emp order by empno',
                p_tname      => 'emp',
                p_mode       => 'replace',
                p_dir        => 'MY_DIR',
                p_filename   => 'emp',
                p_separator  => ',',
                p_enclosure  => '"',
                p_terminator => '~' );

    dbms_output.put_line( to_char(l_rows) ||
                          ' rows extracted to ascii file' );
end;
/
14 rows extracted to ascii file
PL/SQL procedure successfully completed.
```

The control file that was generated by this shows the following (note that the numbers in parentheses in **bold** on the right are not actually in the file; they are solely for reference purposes):

```
load data                                        (1)
infile 'emp.dat' "str x'7E0A'"                   (2)
into table emp                                   (3)
replace                                          (4)
fields terminated by X'2c' enclosed by X'22'     (5)
(                                                (6)
EMPNO char(44 ),                                 (7)
ENAME char(20 ),                                 (8)
JOB char(18 ),                                   (9)
MGR char(44 ),                                   (10)
HIREDATE date 'ddmmyyyyhh24miss' ,               (11)
SAL char(44 ),                                   (12)
COMM char(44 ),                                  (13)
DEPTNO char(44 ),                                (14)
)                                                (15)
```

The things to note about this control file are as follows:

- Line (2): We use the STR feature of SQLLDR. We can specify what character or string is used to terminate a record. This allows us to load data with embedded newlines easily. The string x'7E0A' is simply a tilde followed by a newline.
- Line (5): We use our separator character and enclosure character. We do not use OPTIONALLY ENCLOSED BY, since we will be enclosing every single field after doubling any occurrence of the enclosure character in the raw data.
- Line (11): We use a large "numeric" date format. This does two things: it avoids any NLS issues with regard to the data, and it preserves the time component of the date field.

The raw data (.dat) file generated from the preceding code looks like this:

```
"7369","SMITH","CLERK","7902","17121980000000","800","","20"~
"7499","ALLEN","SALESMAN","7698","20021981000000","1600","300","30"~
"7521","WARD","SALESMAN","7698","22021981000000","1250","500","30"~
"7566","JONES","MANAGER","7839","02041981000000","2975","","20"~
"7654","MARTIN","SALESMAN","7698","28091981000000","1250","1400","30"~
"7698","BLAKE","MANAGER","7839","01051981000000","2850","","30"~
"7782","CLARK","MANAGER","7839","09061981000000","2450","","10"~
"7788","SCOTT","ANALYST","7566","19041987000000","3000","","20"~
"7839","KING","PRESIDENT","","17111981000000","5000","","10"~
"7844","TURNER","SALESMAN","7698","08091981000000","1500","0","30"~
"7876","ADAMS","CLERK","7788","23051987000000","1100","","20"~
"7900","JAMES","CLERK","7698","03121981000000","950","","30"~
"7902","FORD","ANALYST","7566","03121981000000","3000","","20"~
"7934","MILLER","CLERK","7782","23011982000000","1300","","10"~
```

Things to note in the `.dat` file are as follows:

- Each field is enclosed in our enclosure character.
- The `DATE`s are unloaded as large numbers.
- Each line of data in this file ends with a ~ as requested.

We can now reload this data easily using SQLLDR. You may add options to the SQLLDR command line as you see fit.

As stated previously, the logic of the unload package may be implemented in a variety of languages and tools. On the Ask Tom web site, you will find this example implemented not only in PL/SQL as here, but also in Pro*C and SQL*Plus scripts. Pro*C is the fastest implementation, and it always writes to the client workstation file system. PL/SQL is a good all-around implementation (there's no need to compile and install on client workstations), but it always writes to the server file system. SQL*Plus is a good middle ground, offering fair performance and the ability to write to the client file system.

Data Pump Unload

Oracle9*i* introduced external tables as a method to read external data into the database. Oracle 10*g* introduced the ability to go the other direction and use a `CREATE TABLE` statement to create external data, to unload data from the database. As of Oracle 10*g*, this data is extracted in a proprietary binary format known as *Data Pump format*, which is the same format the `EXPDB` and `IMPDP` tools provided by Oracle use to move data from database to database.

Using the external table unload is actually quite easy—as easy as a `CREATE TABLE AS SELECT` statement. To start, we need a `DIRECTORY` object:

```
ops$tkyte@ORA10GR1> create or replace directory tmp as '/tmp'
  2  /
 Directory created.
```

Now we are ready to unload data to this directory using a simple `SELECT` statement, for example:

```
ops$tkyte@ORA10GR1> create table all_objects_unload
  2  organization external
  3  ( type oracle_datapump
  4    default directory TMP
  5    location( 'allobjects.dat' )
  6  )
  7  as
  8  select
  9  *
 10  from all_objects
 11  /
Table created.
```

I purposely chose the ALL_OBJECTS view, because it is a quite complex view with lots of joins and predicates. This example shows we can use this Data Pump unload technique to extract arbitrary data from our database. We could add predicates, or whatever we wanted, to extract a slice of data.

■ **Note** This example shows we can use this Data Pump unload technique to extract arbitrary data from our database. Yes, that is repeated text. From a security perspective, this does make it rather easy for someone with access to the information to "take" the information elsewhere. You need to control access to the set of people who have the ability to create DIRECTORY objects and write to them, and who have the necessary access to the physical server to get the unloaded data.

The final step would be to copy allobjects.dat onto another server, perhaps a development machine for testing with, and extract the DDL to re-create this table over there:

```
ops$tkyte@ORA10GR1> select dbms_metadata.get_ddl( 'TABLE', 'ALL_OBJECTS_UNLOAD' )
  2    from dual;

DBMS_METADATA.GET_DDL('TABLE','ALL_OBJECTS_UNLOAD')
--------------------------------------------------------------------------------
  CREATE TABLE "OPS$TKYTE"."ALL_OBJECTS_UNLOAD"
   (    "OWNER" VARCHAR2(30),
        "OBJECT_NAME" VARCHAR2(30),
        "SUBOBJECT_NAME" VARCHAR2(30),
        "OBJECT_ID" NUMBER,
        "DATA_OBJECT_ID" NUMBER,
        "OBJECT_TYPE" VARCHAR2(19),
        "CREATED" DATE,
        "LAST_DDL_TIME" DATE,
        "TIMESTAMP" VARCHAR2(19),
        "STATUS" VARCHAR2(7),
        "TEMPORARY" VARCHAR2(1),
        "GENERATED" VARCHAR2(1),
        "SECONDARY" VARCHAR2(1)
   )
   ORGANIZATION EXTERNAL
    ( TYPE ORACLE_DATAPUMP
      DEFAULT DIRECTORY "TMP"

      LOCATION
       ( 'allobjects.dat'
       )
    )
```

This makes it rather easy to load this extract on another database, as it would simply be

```
SQL> insert /*+ append */ into some_table select * from all_objects_unload;
```

and you are done—the data is loaded.

Summary

In this chapter, we covered many of the ins and outs of data loading and unloading. We started with SQL*Loader (SQLLDR) and examined many of the basic techniques for loading delimited data, fixed-width data, LOBs, and the like. We discussed how this knowledge carries right over into external tables, a new feature in Oracle9*i* and later that is useful as a replacement for SQLLDR, but that still utilizes our SQLLDR skills.

Then we looked at the reverse process, data unloading, and how to get data out of the database in a format that other tools, such as spreadsheets or the like, may use. In the course of that discussion, we developed a PL/SQL utility to demonstrate the process—one that unloads data in a SQLLDR-friendly format, but could easily be modified to meet our needs.

Lastly, we looked at a new Oracle 10*g* feature, the external table unload, and the ability to easily create and move extracts of data from database to database.