10 CHAPTER

Using SQL*Loader

✦ ✦ ✦ ✦

In This Chapter

Introducing SQL*Loader

Understanding the SQL*Loader control file

Understanding the SQL*Loader command

Studying SQL*Loader examples

✦ ✦ ✦ ✦

SQL*Loader is an Oracle utility that enables you to efficiently load large amounts of data into a database. If you have data in a flat file, such as a comma-delimited text file, and you need to get that data into an Oracle database, SQL*Loader is the tool to use. This chapter introduces you to the SQL*Loader utility, discusses its control file, provides the syntax for using the SQL*Loader command, and provides examples of using SQL*Loader to load data into databases.

Introducing SQL*Loader

SQL*Loader's sole purpose in life is to read data from a flat file and to place that data into an Oracle database. In spite of having such a singular purpose, SQL*Loader is one of Oracle's most versatile utilities. Using SQL*Loader, you can do the following:

- Load data from a delimited text file, such as a comma-delimited file
- Load data from a fixed-width text file
- Load data from a binary file
- Combine multiple input records into one logical record
- Store data from one logical record into one table or into several tables
- Write SQL expressions to validate and transform data as it is being read from a file
- Combine data from multiple data files into one
- Filter the data in the input file, loading only selected records

- Collect *bad* records — that is, those records that won't load — into a separate file where you can fix them
- And more!

The alternative to using SQL*Loader would be to write a custom program each time you needed to load data into your database. SQL*Loader frees you from that, because it is a generic utility that can be used to load almost any type of data. Not only is SQL*Loader versatile, it is also fast. Over the years, Oracle has added support for direct-path loads, and for parallel loads, all in an effort to maximize the amount of data that you can load in a given time period.

Understanding the SQL*Loader Control File

To use SQL*Loader, you need to have a database, a flat file to load, and a control file to describe the contents of the flat file. Figure 10-1 illustrates the relationship between these.

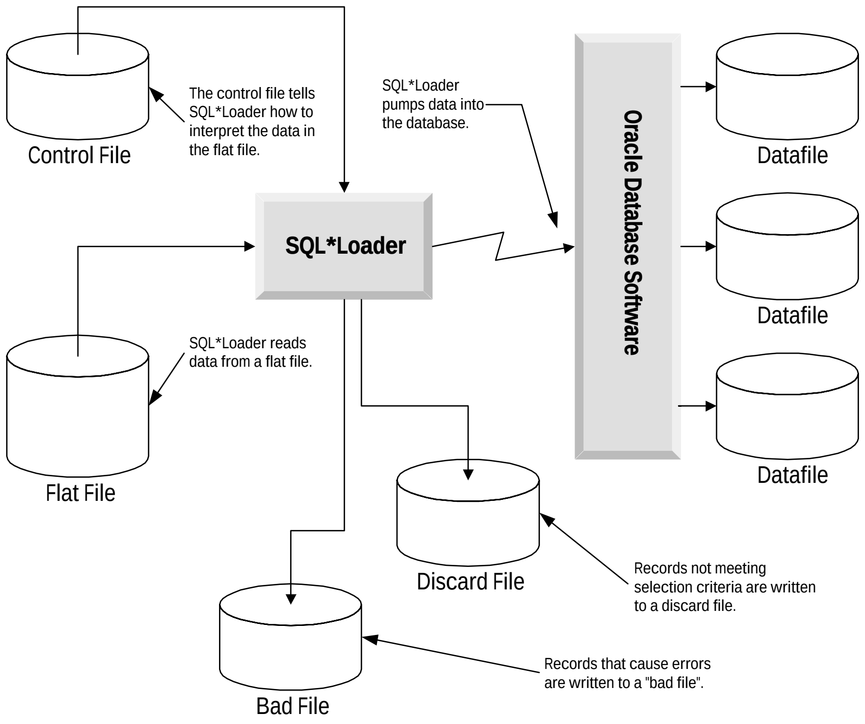
Control File
The control file tells SQL*Loader how to interpret the data in the flat file.
SQL*Loader pumps data into the database.
Oracle Database Software
Datafile
Datafile
Datafile
SQL*Loader
SQL*Loader reads data from a flat file.
Flat File
Discard File
Records not meeting selection criteria are written to a discard file.
Bad File
Records that cause errors are written to a "bad file".

Figure 10-1: The SQL*Loader environment

Control files, such as the one illustrated in Figure 10-1, contain a number of commands and clauses describing the data that SQL*Loader is reading. Control files also tell SQL*Loader where to store that data, and they can define validation expressions for the data. Understanding control file syntax is crucial to using SQL*Loader effectively.

The control file is aptly named, because it controls almost every aspect of how SQL*Loader operates. The control file describes the format of the data in the input file and tells SQL*Loader which tables and columns to populate with that data. When you write a control file, you need to be concerned with these questions:

- ✦ What file, or files, contain the data that you want to load?
- ✦ What table, or tables, are you loading?
- ✦ What is the format of the data that you are loading?
- ✦ What do you want to do with records that won't load?

All of these items represent things that you specify when you write a SQL*Loader control file. Generally, control files consist of one long command that starts out like this:

```
LOAD DATA
```

The keyword `DATA` is optional. Everything else in the control file is a clause of some sort that is added onto this command.

SQL*Loader is a broad subject that's difficult to condense into one chapter. The control file clauses shown in this chapter are the ones most commonly used when loading data from text files. The corresponding examples will help you understand SQL*Loader and how it's used, and should provide enough background for you to easily use the other features of SQL*Loader as explained in the *Oracle8i Server Utilities* manual.

Many of the control file clauses you'll encounter in this chapter are explained by example. The task of loading data into the following table forms the basis for those examples:

```
CREATE TABLE animal_feeding (
animal_id        NUMBER,
    feeding_date    DATE,
    pounds_eaten    NUMBER (5,2),
    note            VARCHAR2(80)    );
```

Some examples are based on loading data from a fixed-width text file into the `animal_feeding` table, while others are based on loading the same data from a comma-delimited file.

Specifying the input file

You use the INFILE clause to identify the file containing the data that you want to load. The data can be in a file separate from the control file, which is usually the case, or you can place the data within the control file itself. Use multiple INFILE clauses if your data is spread across several files.

Control File Data

If you are loading data from a text file, you have the option of placing the LOAD command at the beginning of that file, which then becomes the control file. To specify that SQL*Loader looks in the control file for the data, supply an asterisk (*) for the file name in the INFILE clause. For example:

```
LOAD DATA
  INFILE *
...
...
...
BEGINDATA
data
data
data
```

If you do include your data in the control file, the last clause of your LOAD command must be the BEGINDATA clause. This tells SQL*Loader where the command ends and where your data begins. SQL*Loader will begin reading data from the line immediately following BEGINDATA.

Data in a Separate File

Although you can have data in the control file, it's more common to have it in a separate file. In that case, you place the file name after the keyword INFILE, as shown in this example:

```
LOAD DATA
  INFILE 'animal_feeding.csv'
...
...
...
```

Placing quotes around the file name often isn't necessary, but it's a good habit to get into. If the file name happens to match a SQL*Loader keyword, contains some strange punctuation, or is case sensitive (UNIX), you could run into problems unless it's quoted. You can use either single or double quotes. If necessary, you may include a path as part of the file name. The default extension is .dat.

Data in Multiple Files

You can use multiple INFILE clauses to load data from several files at once. The clauses must follow each other, as shown here:

```
LOAD DATA
  INFILE 'animal_feeding_fixed_1.dat'
  INFILE 'animal_feeding_fixed_2.dat'
...
...
...
```

When you specify multiple files like this, SQL*Loader will read them in the order in which they are listed.

Loading data into nonempty tables

After listing the input file, or files, in SQL*Loader, you need to specify whether you expect the table that you are loading to be empty. By default, SQL*Loader expects that you are loading data into a completely empty table. If, when the load starts, SQL*Loader finds even one row in the table, the load will be aborted. This is sometimes frustrating to people who haven't used SQL*Loader before, because it's not the behavior that you would intuitively expect.

Four keywords control SQL*Loader's behavior when it comes to dealing with empty vs. nonempty tables:

INSERT	Specifies that you are loading an empty table. SQL*Loader will abort the load if the table contains data to start with.
APPEND	Specifies that you are *adding* data to a table. SQL*Loader will proceed with the load even if preexisting data is in the table.
REPLACE	Specifies that you want to *replace* the data in a table. Before loading, SQL*Loader will *delete* any existing data.
TRUNCATE	Specifies the same as REPLACE, but SQL*Loader uses the TRUNCATE statement instead of a DELETE statement to delete existing data.

Place the keyword for whichever option you choose after the INFILE clause, as shown in this example:

```
LOAD DATA
  INFILE 'animal_feeding.csv'
  APPEND
...
...
...
```

If you don't specify an option, then INSERT is assumed by default.

Specifying the table to load

In SQL*Loader, you use the `INTO TABLE` clause to specify which table or tables you want to load. It also specifies the format of the data contained in the input file. The `INTO TABLE` clause is the most complex of all the clauses, and what you see here represents only a fraction of what it can include.

Loading One Table

To load one table, just place the `INTO TABLE` clause in your `LOAD` statement, as shown in the following example:

```
LOAD DATA
  INFILE 'animal_feeding.dat'
  APPEND
INTO TABLE animal_feeding
    (
      animal_id       POSITION (1:3) INTEGER EXTERNAL,
      feeding_date    POSITION (4:14) DATE "dd-mon-yyyy",
      pounds_eaten    POSITION (15:19) ZONED (5,2),
      note            POSITION (20:99) CHAR
    )
```

The table name shown in this example is the `animal_feeding` table. The same issues apply to table names as to file names. If the table name matches a reserved word or is case sensitive, enclose it within quotes.

A big part of the `INTO TABLE` clause is the list of field definitions. These describe the input file format for SQL*Loader and map the data in the input file onto the appropriate columns within the table being loaded. The sections “Describing delimited columns” and “Describing fixed-width columns,” later in this chapter, explain more about writing field definitions.

Loading More than One Table

You can use multiple `INTO TABLE` clauses to load more than one table. Each `INTO TABLE` clause gets its own set of field definitions. Listing 10-1 shows an example of how you can split data among two tables.

Listing 10-1: **Loading data into two tables**

```
LOAD DATA
  INFILE 'animal_feeding_fixed.dat'
  APPEND
INTO TABLE animal_feeding
    (
      animal_id       POSITION (1:3) INTEGER EXTERNAL,
      feeding_date    POSITION (4:14) DATE "dd-mon-yyyy",
      pounds_eaten    POSITION (15:19) ZONED (5,2)
    )
INTO TABLE animal_feeding_note
    (
      animal_id       POSITION (1:3) INTEGER EXTERNAL,
      feeding_date    POSITION (4:14) DATE "dd-mon-yyyy",
      note            POSITION (20:99) CHAR
    )
```

In this example, `animal_id` and `feeding_date` are loaded into both tables. After that, however, the `animal_feeding` table gets the `pounds_eaten` value, while the `animal_feeding_note` table gets the `note` value.

Loading Multiple Tables from Delimited Data

When you load data into multiple tables like this, you can run into complications if the data that you are loading is delimited. Consider the `LOAD` command shown in Listing 10-2.

Listing 10-2: **Problems loading delimited data into multiple tables**

```
LOAD DATA
  INFILE 'animal_feeding.csv'
  APPEND
INTO TABLE animal_feeding
    (
      animal_id       INTEGER EXTERNAL TERMINATED BY ',',
      feeding_date    DATE "dd-mon-yyyy" TERMINATED BY ',',
      pounds_eaten    DECIMAL EXTERNAL TERMINATED BY ','
    )
INTO TABLE animal_feeding_note
    (
      animal_id       INTEGER EXTERNAL TERMINATED BY ',',
      feeding_date    DATE "dd-mon-yyyy" TERMINATED BY ',',
      note            CHAR TERMINATED BY ','
                      OPTIONALLY ENCLOSED BY '"'
    )
```

The problem that you experience here is that SQL*Loader works through delimited fields in the order in which they are listed, and this order cuts across all the `INTO`

TABLE clauses. Thus, SQL*Loader would expect animal_id for the animal_feeding_note table to follow the pounds_eaten value in the input file. The second feeding_date would have to follow that, and so forth. To reset SQL*Loader to the beginning of the line where the second INTO TABLE clause is applied to a record, you need to add a POSITION clause to the first field listed for that table. The LOAD statement in Listing 10-3 would work here.

Listing 10-3: Repositioning SQL*Loader's pointer into the record

```
LOAD DATA
  INFILE 'animal_feeding.csv'
  APPEND
INTO TABLE animal_feeding
    (
      animal_id       INTEGER EXTERNAL TERMINATED BY ',',
      feeding_date    DATE "dd-mon-yyyy" TERMINATED BY ',',
      pounds_eaten    DECIMAL EXTERNAL TERMINATED BY ','
    )
INTO TABLE animal_feeding_note TRAILING NULLCOLS
    (
      animal_id       POSITION (1) INTEGER EXTERNAL
                      TERMINATED BY ',',
      feeding_date    DATE "dd-mon-yyyy" TERMINATED BY ',',
      pounds_eaten    FILLER DECIMAL EXTERNAL
                      TERMINATED BY ',',
      note            CHAR TERMINATED BY ','
                      OPTIONALLY ENCLOSED BY '"'
    )
```

Notice the following in the example shown in Listing 10-3:

✥ The second definition of animal_id contains the clause POSITION (1). This causes SQL*Loader to start scanning from the first character of the record. This is the behavior you want because you are loading the same field into two tables. Otherwise, SQL*Loader would look for another animal_id following the pounds_eaten column.

✥ The TRAILING NULLCOLS clause has been added to the second INTO TABLE clause because not all records in the input file contain notes.

✥ Even though you aren't storing it in the animal_feeding_note table, the pounds_eaten column doesn't go away. The FILLER keyword has been used to specify that SQL*Loader not load this field.

You can see that life does get a bit complex when loading a delimited file into multiple tables.

Describing fixed-width columns

The `INTO TABLE` clause contains a field list within parentheses. This list defines the fields being loaded from the flat file into the table. Each entry in the field list has this general format:

```
column_name    POSITION (start:end) datatype
```

- ✦ ***column_name.*** The name of a column in the table that you are loading.
- ✦ `POSITION` **(*start*:*end*).** The position of the column within the record. The values for *start* and *end* represent the character positions for the first and last characters of the column. The first character of a record is always position 1.
- ✦ ***datatype.*** A SQL*Loader datatype (not the same as an Oracle datatype) that identifies the type of data being loaded. Table 10-1 lists some of these.

You will need to write one field list entry for each column that you are loading. As an example, consider the following record:

```
10010-jan-200002350Flipper seemed unusually hungry today.
```

This record contains a three-digit ID number, followed by a date, followed by a five-digit number, followed by a text field. The ID number occupies character positions 1 through 3 and is an integer, so its definition would look like this:

```
animal_id       POSITION (1:3) INTEGER EXTERNAL,
```

The date field is next, occupying character positions 4 through 14, and its definition looks like this:

```
feeding_date    POSITION (4:14) DATE "dd-mon-yyyy",
```

Notice the `"dd-mon-yyyy"` string following the datatype. This tells SQL*Loader the specific format used for the date field. SQL*Loader uses this in a call to Oracle's built-in `TO_DATE` function, so any format that works for `TO_DATE` may be specified for SQL*Loader `DATE` fields.

You could continue to use the same method to define the rest of the fields that you want to load. The complete field list would look like this:

```
  (
  animal_id       POSITION (1:3) INTEGER EXTERNAL,
  feeding_date    POSITION (4:14) DATE "dd-mon-yyyy",
  pounds_eaten    POSITION (15:19) ZONED (5,2),
  note            POSITION (20:99) CHAR
)
```

Using SQL*Loader Datatypes

SQL*Loader supports a wide variety of datatypes. Table 10-1 lists those that are most useful when loading data from text files.

Table 10-1
SQL*Loader Text-based Datatypes

Datatype Name	***Description***
`CHAR`	Identifies character data. Don't confuse this with the `CHAR` datatype used within the database. No relationship exists between the two. If you are loading data into any type of text field, such as `VARCHAR2`, `CHAR`, or `CLOB`, use the SQL*Loader `CHAR` datatype.
`DATE` ["*format*"]	Identifies a date. Even though it's optional, specify a format. That way, you avoid problems if the default date format in the database is different from what you expect.
`INTEGER EXTERNAL`	Identifies an integer value that is stored in character form. For example, the character string `"123"` is a valid `INTEGER EXTERNAL` value.
`DECIMAL EXTERNAL`	Identifies a numeric value that is stored in character form and that may include a decimal point. The string `"-123.45"` is a good example of a `DECIMAL EXTERNAL` value.
`ZONED` (*precision*, *scale*)	Identifies a zoned decimal field, such as you might find in a file generated by a COBOL program. *Zoned decimal fields* are numeric values represented as character strings and that contain an assumed decimal point. For example, a definition of `ZONED (5,2)` would cause `"12345"` to be interpreted as 123.45. See the note following this table regarding zoned decimal and negative values.

Note

Be careful with the `ZONED` datatype. It can be handy for loading numeric values with assumed decimal places, but you have to be aware of how it expects the sign to be represented. This datatype harks back to the old card-punch days when data was stored on 80-character-wide punch cards. The sign for zoned decimal numbers was stored as an *overpunch* on one of the digits. The practical effect of that is that the `ZONED` data type will not recognize a hyphen (-) as a negative sign. It will, however, recognize some letters of the alphabet as valid digits. If you're not loading true zoned decimal data, such as a COBOL program might create, then use `ZONED` only for non-negative numbers.

SQL*Loader supports a number of other datatypes, most of which are well beyond the scope of this chapter. One other that you will read about later is the `LOBFILE` type. An example near the end of this chapter shows you how to load files into `CLOB` columns.

Converting Blanks to Nulls

When you're dealing with data in fixed-width columns, you'll find that missing values appear as blanks in the data file. Take a look at the following two lines of data:

```
   10-jan-200002350Flipper seemed unusually hungry today.
10510-jan-200009945Spread over three meals.
```

The first record is missing the three-digit animal ID number. Should that be interpreted as a null value? Or should it be left alone, causing the record to be rejected because spaces do not constitute a valid number? The latter behavior is the default.

If you prefer to treat a blank field as a null, you can use the `NULLIF` clause to tell SQL*Loader to do that. The `NULLIF` clause comes after the datatype and takes the following form:

```
NULLIF field_name=BLANKS
```

To define `animal_id` so that blank values are stored as nulls, you would use this definition:

```
animal_id       POSITION (1:3) INTEGER EXTERNAL
                NULLIF animal_id=BLANKS,
```

You can actually have any valid SQL*Loader expression following the `NULLIF` clause, but comparing the column to `BLANKS` is the most common approach taken.

Describing delimited columns

The format for describing delimited data, such as comma-delimited data, is similar to that used for fixed-width data. The difference is that you need to specify the delimiter being used. The general format of a delimited column definition looks like this:

```
column_name    datatype TERMINATED BY 'delim'
            [OPTIONALLY ENCLOSED BY 'delim']
```

The elements of this column definition are described as follows:

column_name	The name of a column in the table that you are loading.
datatype	A SQL*Loader datatype. (See Table 10-1.)
TERMINATED BY *'delim*	Identifies the delimiter that marks the end of the column.
OPTIONALLY ENCLOSED BY *'delim'*	Specifies an optional enclosing character. Many text values, for example, are enclosed by quotation marks.

When describing delimited fields, you must be careful to describe them in the order in which they occur. Take a look at this record, which contains some delimited data:

```
100,1-jan-2000,23.5,"Flipper seemed unusually hungry today."
```

The first field in the record is a three-digit number, an ID number in this case, and can be defined as follows:

```
animal_id       INTEGER EXTERNAL TERMINATED BY ',',
```

The remaining fields can be defined similarly to the first. However, the note field represents a special case because it is enclosed within quotation marks. To account for that, you must add an ENCLOSED BY clause to that field's definition. For example:

```
note            CHAR TERMINATED BY ','
                OPTIONALLY ENCLOSED BY '"'
```

The keyword OPTIONALLY tells SQL*Loader that the quotes are optional. If they are there, SQL*Loader will remove them. Otherwise, SQL*Loader will load whatever text it finds.

Working with short records

When dealing with delimited data, you occasionally run into cases where not all fields are present in each record in a data file. Take, for example, these two records:

```
100,1-jan-2000,23.5,"Flipper seemed unusually hungry today."
151,1-jan-2000,55
```

The first record contains a note, while the second does not. SQL*Loader's default behavior is to consider the second record as an error because not all fields are present. You can change this behavior, and cause SQL*Loader to treat missing values at the end of a record as nulls, by using the TRAILING NULLCOLS clause. This clause is part of the INTO TABLE clause, and appears as follows:

```
...
INTO TABLE animal_feeding
    TRAILING NULLCOLS
    (
      animal_id       INTEGER EXTERNAL TERMINATED BY ',',
      feeding_date    DATE "dd-mon-yyyy" TERMINATED BY ',',
      pounds_eaten    DECIMAL EXTERNAL TERMINATED BY ',',
      note            CHAR TERMINATED BY ','
                      OPTIONALLY ENCLOSED BY '"'
    )
```

When you use TRAILING NULLCOLS, any missing fields in the record will be saved in the database as nulls.

Error-causing records

When SQL*Loader reads a record from the input file, and for one reason or another is unable to load that record into the database, two things happen:

- ✦ An error message is written to the log file.
- ✦ The record that caused the error is written to another file called the *bad file*.

Bad files have the same format as the input file from which they were created. The reason that SQL*Loader writes bad records to a bad file is to make it easy for you to find and correct the errors. Once the load is done, you can edit the bad file (assuming that it is text), correct the errors, and resubmit the load using the same control file as was originally used.

The default name for the bad file is the input file name, but with the extension .bad. You can specify an alternate bad file name as part of the INFILE clause. For example:

```
INFILE 'animal_feeding.csv'
    BADFILE 'animal_feeding_bad.bad'
```

Each input file gets its own bad file. If you are using multiple INFILE clauses, each of those can specify a different bad file name.

Concatenating records

SQL*Loader has the ability to combine multiple physical records into one logical record. You can do this in one of two ways. You can choose to combine a fixed number of logical records into one physical record, or you can base that determination on the value of some field in the record.

The CONCATENATE Clause

If you have a case where a logical record is always made up of a fixed number of physical records, you can use the `CONCATENATE` clause to tell SQL*Loader to combine the records. The `CONCATENATE` clause appears in the `LOAD` statement, as shown in this example:

```
LOAD DATA
  INFILE 'animal_feeding_concat.csv'
    BADFILE 'animal_feeding_concat'
  APPEND
  CONCATENATE 2
...
```

In this example, every two physical records in the input file will be combined into one longer, logical record. The effect will be as if you took the second record and added it to the end of the first record. The following two records, for example:

```
100,1-jan-2000,23.5,
"Flipper seemed unusually hungry today."
```

will be combined into this one:

```
100,1-jan-2000,23.5,"Flipper seemed unusually hungry today."
```

The `CONCATENATE` clause is the appropriate choice if the number of records to be combined is always the same. Sometimes, however, you have to deal with cases where a particular field in a record determines whether the record is continued. For those cases, you must use `CONTINUEIF`.

The CONTINUEIF Clause

The `CONTINUEIF` clause allows you to identify continuation characters in the input record that identify whether a record should be continued. There are three possible variations on the `CONTINUEIF` clause. Which one you use depends on how the continuation characters are specified in your input file:

`CONTINUEIF THIS`	Use this option if each record in your input file contains a flag indicating whether the next record should be considered a continuation of the current record.

`CONTINUEIF NEXT`	Use this option if the continuation flag is not in the first record to be continued, but rather in each subsequent record.
`CONTINUEIF LAST`	Use this option if the continuation flag is always the last nonblank character or string of characters in the record.

One you've made this choice, your next two tasks are to specify the string that marks a continued record and to tell SQL*Loader the character positions where that string can be found. Let's say you have an input file that uses a dash as a continuation character and that looks like this:

```
-17510-jan-200003550
 Paintuin skipped his first meal.
-19910-jan-200000050
 Nosey wasn't very hungry today.
 20210-jan-200002200
```

The hyphen (`-`) character in the first column of a line indicates that the record is continued to the next line in the file. Records need to be concatenated until one is encountered that doesn't have a hyphen. Because the hyphen is in the record being continued, and because it is not the last nonblank character in the record, the `CONTINUEIF THIS` option is the appropriate one to use. The proper `CONTINUEIF` clause then becomes:

```
CONTINUEIF THIS (1:1) = '-'
```

The `(1:1)` tells SQL*Loader that the continuation string starts in column 1 and ends in column 1. The equal sign (=) tells SQL*Loader to keep combining records as long as the continuation field contains the specified string.

When concatenating records, be aware that SQL*Loader removes the continuation string when it does the concatenation. Thus, the following two records:

```
-17510-jan-200003550
 Paintuin skipped his first meal.
```

will be combined into one like this:

```
17510-jan-200003550Paintuin skipped his first meal.
```

Notice that the leading character from each record, the one indicating whether the record is continued, has been removed. With one exception, SQL*Loader always does this. The exception is when you use `CONTINUEIF LAST`. When you use `CONTINUEIF LAST`, SQL*Loader leaves the continuation character or characters in the record.

The `CONTINUEIF NEXT` parameter works similarly to `CONTINUEIF THIS`, except that SQL*Loader looks for the continuation flag in the record subsequent to the one being processed. The `CONTINUEIF LAST` parameter always looks for the continuation string at the end of the record, so you don't need to specify an exact position. The `CONTINUEIF LAST` parameter is ideal for delimited records, and there's an example later in this chapter showing how it's used.

Understanding the SQL*Loader Command

You must invoke the SQL*Loader utility from the command line. The command is usually `sqlldr`, but it can vary depending on the operating system you're using.

Note

Older releases of Oracle on Windows NT embedded part of the release number into the file name. So this command would be `sqlldr80`, `sqlldr73`, and so forth.

As with other Oracle command-line utilities, SQL*Loader can accept a number of command-line arguments. SQL*Loader can also read command-line arguments from a separate parameter file (not to be confused with the control file). The syntax for the SQL*Loader command looks like this:

```
sqlldr [param=value[, param=value...]]
```

If you invoke SQL*Loader without any parameters, a short help screen will appear. This is similar to the behavior of the Export and Import utilities. Table 10-2 documents the SQL*Loader parameters.

Table 10-2
SQL*Loader Parameters

Parameter	***Description***
`userid`	Passes in your username, password, and Net8 service name. The syntax to use is the same as for any other command-line utility, and looks like this: `userid=username[/password][@service]`
`control`	Passes in the control file name. Here's an example: `control=[path]filename[.ext]` The default extension for control files is .ctl.
`log`	Passes in the log file name. Here's an example: `log=[path]filename[.ext]` The default extension used for log files is .log. If you don't supply a file name, the log file will be named to match the control file.

Parameter	***Description***
`bad`	Passes in the bad file name. Here's an example: `bad=[path]filename[.ext]` The default extension for bad files is .bad. If you don't supply a file name, the bad file will be named to match the control file. Using this parameter overrides any file name that may be specified in the control file.
`data`	Passes in the data file name. Here's an example: `data=[path]filename[.ext]` The default extension used for data files is .dat. Specifying a data file name on the command line overrides the name specified in the control file. If no data file name is specified anywhere, it defaults to the same name as the control file, but with the .dat extension.
`discard`	Passes in the discard file name. Here's an example: `discard=[path]filename[.ext]` The default extension used for discard files is .dis. If you don't supply a file name, the discard file will be named to match the control file. Using this parameter overrides any discard file name that may be specified in the control file.
`discardmax`	Optionally places a limit on the number of discarded records that will be allowed. The syntax looks like this: `discardmax=`*`number_of_records`* If the number of discarded records exceeds this limit, the load is aborted.
`skip`	Allows you to skip a specified number of logical records. The syntax looks like this: `skip=`*`number_of_records`* Use the `skip` parameter when you want to continue a load that has been aborted and when you know how far into the file you want to go before you restart.
`load`	Optionally places a limit on the number of logical records to load into the database. The syntax looks like this: `load=`*`number_of_records`* Once the specified limit has been reached, SQL*Loader will stop.

Continued

Table 10-2 *(continued)*

Parameter	*Description*
`errors`	Specifies the number of errors to allow before SQL*Loader aborts the load. The syntax looks like this: `errors=`*`number_of_records`* SQL*Loader will stop the load if more than the specified number of errors has been received. The default limit is 50. There is no way to allow an unlimited number. The best you can do is to specify a very high value, such as 999999999.
`rows`	Indirectly controls how often commits occur during the load process. The rows parameter specifies the size of the bind array used for conventional-path loads in terms of rows. SQL*Loader will round that value off to be some multiple of the I/O block size. The syntax for the rows parameter looks like this: `rows=`*`number_of_rows`* The default value is 64 for conventional-path loads. Direct-path loads, by default, are saved only when the entire load is done. However, when a direct-path load is done, this parameter can be used to control the commit frequency directly.
`bindsize`	Specifies the maximum size of the bind array. The syntax looks like this: `bindsize=`*`number_of_bytes`* The default is 65,536 bytes (64KB). If you use `bindsize`, any value that you specify overrides the size specified by the `rows` parameter.
`silent`	Allows you to suppress messages displayed by SQL*Loader. You can pass one or more arguments to the silent parameter, as shown in this syntax: `silent=(`*`keyword`*`[, `*`keyword`*`...]])` Valid keywords are the following: `header` — Suppresses introductory messages `feedback` — Suppresses the "commit point reached" messages `errors` — Suppresses data-related error messages `discards` — Suppresses messages related to discarded records
`partitions`	Disables writing of partition statistics when loading a partitioned table

Parameter	***Description***
`all`	Disables all the messages described above
`direct`	Controls whether SQL*Loader performs a direct-path load. The syntax looks like this: `direct={true|false}` The default is `false`, causing a conventional-path load to be performed.
`parfile`	Specifies the name of a parameter file containing command-line parameters. The syntax looks like this: `parfile=[path]filename[.ext]` When the `parfile` parameter is encountered, SQL*Loader opens the file and reads command-line parameters from that file.
`parallel`	Controls whether direct loads are performed using parallel processing. The syntax looks like this: `parallel={true|false}` The default value is `false`.
`readsize`	Controls the size of the buffer used to hold data read from the input file. The syntax looks like this: readsize=size_in_bytes The default value is 65,536 bytes. SQL*Loader will ensure that the `readsize` and `bindsize` values match. If you specify different values for each, SQL*Loader will use the larger value for both settings.
`file`	Specifies the database datafile in which the data is to be stored and may be used when doing a parallel load. The syntax looks like this: `file=`*`datafile_name`* The file must be one of the files in the tablespace for the table or partition being loaded.

Using keywords by position

SQL*Loader allows you to pass command-line parameters using two different methods. You can name each parameter (recommended), or you can pass them positionally. The naming method is the easiest to understand, and looks like this:

```
sqlldr userid=system/manager control=animal_feeding.ctl
```

The positional method allows you to pass parameters without explicitly naming them. You must pass the parameters in the exact order in which Table 10-2 lists them, and you must not skip any. Converting the previous command to use positional notation yields the following:

```
sqlldr system/manager animal_feeding.ctl
```

You can even mix the two methods, passing one or more parameters by position and the remaining parameters by name. For example:

```
sqlldr system/manager control=animal_feeding.ctl
```

Since it's conventional for Oracle utilities to accept a username and password as the first parameter to a command, this last example represents a good compromise between the two methods. With the one exception of the username and password, it is recommended that you name all your parameters. You're much less likely to make a mistake that way.

Using parameter files

As with the Export and Import utilities, SQL*Loader also allows you to place command-line parameters in a text file. You can then use the `parfile` parameter to point to that file. For example, suppose you have a text file named `animal_feeding.par` that contains these lines:

```
userid=system/manager
control=animal_feeding.ctl
```

You could invoke SQL*Loader, and use the parameters from the text file, by issuing this command:

```
sqlldr parfile=animal_feeding.par
```

Parameter files provide a stable place in which to record the parameters used for a load and can serve as a means of documenting loads that you perform regularly.

Studying SQL*Loader Examples

SQL*Loader is best explained through the use of examples, and that is what you are going to see in the remainder of this chapter. The five examples in this section illustrate how to do to the following:

- Loading comma-delimited data
- Concatenating multiple physical records into one logical record

✦ Loading fixed-width, columnar data, and loading from multiple files

✦ Using expressions to modify data before loading it

✦ Loading large amounts of text into a large object column

With one exception, all the examples in this section will load data into the following table:

```
CREATE TABLE animal_feeding (
animal_id        NUMBER,
    feeding_date    DATE,
    pounds_eaten    NUMBER (5,2),
    note            VARCHAR2(80)
    );
```

The one exception involves the last example, which shows you how to load large objects. For that example, the note column is assumed to be a `CLOB` rather than a `VARCHAR2` column. If you want to try these examples yourself, you can find the scripts on the CD in the directory sql_loader_examples.

Loading comma-delimited data

Let's assume that you have the following data in a file named animal_feedings.csv:

```
100,1-jan-2000,23.5,"Flipper seemed unusually hungry today."
105,1-jan-2000,99.45,"Spread over three meals."
112,1-jan-2000,10,"No comment."
151,1-jan-2000,55
166,1-jan-2000,17.5,"Shorty ate Squacky."
145,1-jan-2000,0,"Squacky is no more."
175,1-jan-2000,35.5,"Paintuin skipped his first meal."
199,1-jan-2000,0.5,"Nosey wasn't very hungry today."
202,1-jan-2000,22.0
240,1-jan-2000,28,"Snoops was lethargic and feverish."
...
```

This format is the typical comma-separated values (CSV) format that you might get if you had entered the data in Excel and saved it as a comma-delimited file. The fields are all delimited by commas, and the text fields are also enclosed within quotes. The following control file, named animal_feedings.ctl, would load this data:

```
LOAD DATA
  INFILE 'animal_feeding.csv'
    BADFILE 'animal_feeding'
  APPEND
  INTO TABLE animal_feeding
    TRAILING NULLCOLS
    (
      animal_id       INTEGER EXTERNAL TERMINATED BY ",",
      feeding_date    DATE "dd-mon-yyyy" TERMINATED BY ",",
      pounds_eaten    DECIMAL EXTERNAL TERMINATED BY ",",
      note            CHAR TERMINATED BY ","
                      OPTIONALLY ENCLOSED BY '"'
   )
```

Here are some points worth noting about this control file:

- ✦ Any records that won't load because of an error will be written to a file named animal_feeding.bad. The `BADFILE` clause specifies the file name, and the extension .bad is used by default.
- ✦ The `APPEND` keyword causes SQL*Loader to insert the new data regardless of whether the table has any existing data. While not the default, the `APPEND` option is one you'll often want to use.
- ✦ The `TRAILING NULLCOLS` option is used because not all records in the input file contain a value for all fields. The `note` field is frequently omitted. Without `TRAILING NULLCOLS`, omitting a field would result in an error, and those records would be written to the bad file.
- ✦ The definition of the date field includes a format mask. This is the same format mask that you would use with Oracle's built-in `TO_DATE` function.

The following example shows SQL*Loader being invoked to load this data:

```
E:\> sqlldr seapark/seapark@bible_db control=animal_feeding

SQL*Loader: Release 8.1.5.0.0 - Production on Wed Aug 18 11:02:24 1999

(c) Copyright 1999 Oracle Corporation.  All rights reserved.

Commit point reached - logical record count 28
```

The command-line parameter `control` is used to pass in the control file name. The extension defaults to .ctl. The same command, but with different control file names, can be used for all the examples in this section.

Concatenating physical records into one logical record

Some of the lines in the animal_feeding.csv file shown in the previous example are quite long. That's because the note field can be up to 80 characters long. If you took that same file, named it animal_feeding_concat.csv, and placed each note on a line by itself, you would have a file containing records like those shown in Listing 10-4.

Listing 10-4: **Placing each note on its own line**

```
100,4-jan-2000,23.5,
"Flipper seemed unusually hungry today."
105,4-jan-2000,99.45,
"Spread over three meals."
112,4-jan-2000,10,
"No comment."
151,4-jan-2000,55
166,4-jan-2000,17.5,
"Shorty ate Squacky."
145,4-jan-2000,0,
"Squacky is no more."
175,4-jan-2000,35.5,
"Paintuin skipped his first meal."
199,4-jan-2000,0.5,
"Nosey wasn't very hungry today."
202,4-jan-2000,22.0
240,4-jan-2000,28,
"Snoops was lethargic and feverish."
...
```

This presents an interesting problem, because sometimes you want to concatenate two records into one, and sometimes you don't. In this case, the key lies in the fact that for each logical record that contains a comment, a trailing comma (,) has been left at the end of the physical record containing the numeric and date data. You can use the control file shown in Listing 10-5, which is named animal_feeding_concat.ctl, to key off of that comma, combine records appropriately, and load the data.

Listing 10-5: **A control file that combines two records into one**

```
LOAD DATA
  INFILE 'animal_feeding_concat.csv'
    BADFILE 'animal_feeding_concat'
  APPEND
  CONTINUEIF LAST = ","
  INTO TABLE animal_feeding
    TRAILING NULLCOLS
    (
      animal_id       INTEGER EXTERNAL TERMINATED BY ",",
      feeding_date    DATE "dd-mon-yyyy" TERMINATED BY ",",
      pounds_eaten    DECIMAL EXTERNAL TERMINATED BY ",",
      note            CHAR TERMINATED BY ","
                      OPTIONALLY ENCLOSED BY '"'
    )
```

There are two keys to making this approach work:

- ✦ If a line is to be continued, the last nonblank character must be a comma. Since all fields are delimited by commas, this is pretty easy to arrange.
- ✦ The `CONTINUEIF LAST = ","` clause tells SQL*Loader to look for a comma at the end of each line read from the file. Whenever it finds a comma, the next line is read and appended onto the first.

You aren't limited to concatenating two lines together. You can actually concatenate as many lines as you like, as long as they each contain a trailing comma. For example, you could enter the 5-Jan-2000 feeding for animal #100 as follows:

```
100,
5-jan-2000,
19.5,
"Flipper's appetite has returned to normal."
```

All four lines can be concatenated because the first three end with a comma. The fourth line doesn't end with a comma, and that signals the end of the logical record.

Loading fixed-width data

The following example shows you how to load fixed-width data and how to combine data from two files into one load. When you load fixed-width data, you need to use the `POSITION` keyword to specify the starting and ending positions of each field. Say you had two files, named animal_feeding_fixed_1.dat and animal_feeding_fixed_2.dat, containing records that looked like these:

```
10001-jan-200002350Flipper seemed unusually hungry today.
10501-jan-200009945Spread over three meals.
11201-jan-200001000No comment.
15101-jan-200005500
16601-jan-200001750Shorty ate Squacky.
14501-jan-200000000Squacky is no more.
17501-jan-200003550Paintuin skipped his first meal.
19901-jan-200000050Nosey wasn't very hungry today.
20201-jan-200002200
24001-jan-200002800Snoops was lethargic and feverish.
...
```

You could load this data, reading from both files, using the control file shown in Listing 10-6.

Listing 10-6: **Loading fixed-width data from two files**

```
LOAD DATA
  INFILE 'animal_feeding_fixed_1.dat'
    BADFILE 'animal_feeding_fixed_1'
  INFILE 'animal_feeding_fixed_2.dat'
    BADFILE 'animal_feeding_fixed_2'
  APPEND
  INTO TABLE animal_feeding
    TRAILING NULLCOLS
    (
      animal_id       POSITION (1:3) INTEGER EXTERNAL,
      feeding_date    POSITION (4:14) DATE "dd-mon-yyyy",
      pounds_eaten    POSITION (15:19) ZONED (5,2),
      note            POSITION (20:99) CHAR
    )
```

Notice the following about this control file:

- ✦ Two `INFILE` clauses are used, one for each file. Each clause contains its own `BADFILE` name.
- ✦ The `POSITION` clause, instead of the `TERMINATED BY` clause, is used for each field to specify the starting and ending column for that field.
- ✦ The datatype for the `pounds_eaten` field has been changed from `DECIMAL EXTERNAL` to `ZONED` because the decimal point is assumed to be after the third digit and doesn't really appear in the number. For example, 123.45 is recorded in the input file as 12345. COBOL programs commonly create files containing zoned decimal data.
- ✦ The `POSITION` clause appears before the datatype, whereas the `TERMINATED BY` clause appears after the datatype. That's just the way the syntax is.

Other than the use of the `POSITION` clause, there really is no difference between loading fixed-width data and delimited data.

Writing expressions to modify loaded data

SQL*Loader provides you with the ability to write expressions to modify data read from the input file. Look at this comma-delimited data, which is similar to what you loaded earlier:

```
100,13-jan-2000,23.5,"Flipper seemed unusually hungry today."
105,13-jan-2000,99.45,"Spread over three meals."
112,13-jan-2000,10,"No comment."
```

```
151,13-jan-2000,55
166,13-jan-2000,17.5,"Shorty ate Squacky."
145,13-jan-2000,0,"Squacky is no more."
175,13-jan-2000,35.5,"Paintuin skipped his first meal."
199,13-jan-2000,0.5,"Nosey wasn't very hungry today."
202,13-jan-2000,22.0
240,13-jan-2000,28,"Snoops was lethargic and feverish."
```

Imagine for a moment that you want to uppercase the contents of the note field. Imagine also that the weights in this file are in kilograms, and that you must convert those values to pounds as you load the data. You can do that by writing expressions to modify the `note` field and the `pounds_eaten` field. The following example shows you how to multiply the weight by 2.2 to convert from kilograms to pounds:

```
pounds_eaten    DECIMAL EXTERNAL TERMINATED BY ","
                ":pounds_eaten * 2.2",
```

As you can see, the expression has been placed within quotes, and it has been added to the end of the field definition. You can use a field name within an expression, but when you do, you must precede it with a colon (:). You can use any valid SQL expression that you like, but it must be one that will work within the `VALUES` clause of an `INSERT` statement. Indeed, SQL*Loader uses the expression that you supply as part of the actual `INSERT` statement that it builds to load your data. With respect to the `pounds_eaten` example, SQL*Loader will build an `INSERT` statement like this:

```
INSERT INTO animal_feeding
  (animal_id, feeding_date, pounds_eaten, note)
  VALUES (:animal_id, :feeding_date,
          :pounds_eaten * 2.2, :note)
```

Listing 10-7 shows a control file that will both convert from kilograms to pounds and uppercase the `note` field.

Listing 10-7: **Using expressions to transform data**

```
LOAD DATA
  INFILE 'animal_feeding_expr.csv'
    BADFILE 'animal_feeding_expr'
  APPEND
  INTO TABLE animal_feeding
    TRAILING NULLCOLS
    (
      animal_id       INTEGER EXTERNAL TERMINATED BY ",",
      feeding_date    DATE "dd-mon-yyyy" TERMINATED BY ",",
      pounds_eaten    DECIMAL EXTERNAL TERMINATED BY ","
```

```
                          ":pounds_eaten * 2.2",
         note             CHAR TERMINATED BY ","
                          OPTIONALLY ENCLOSED BY '"'
                          "UPPER(:note)"
      )
```

Oracle has a rich library of such functions that you can draw from. These are documented in Appendix B, "SQL Built-in Function Reference."

Loading large amounts of text

So far, all of the examples in this section have shown you how to load *scaler data*. This is the type of data that is normally associated with business applications, and it consists of character strings, dates, and numbers. In addition to loading simple, scaler data, you can also use SQL*Loader to load large object types. Consider this variation of the `animal_feeding` table:

```
CREATE TABLE animal_feeding (
        animal_id               NUMBER,
    feeding_date    DATE,
    pounds_eaten    NUMBER (5,2),
    note            CLOB
    );
```

Instead of an 80-character-wide note column, this version of the table defines the note column as a character-based large object, or `CLOB`. `CLOB`s may contain up to 2GB of data, effectively removing any practical limit on the length of a note. Can you load such a column using SQL*Loader? Yes.

For purposes of our example, let's assume that you have created note files for each animal, and that each file contains information similar to what you see here:

```
NAME: Shorty
DATE: 16-Jan-2000
TEMPERATURE: 115.2
ACTIVITY LEVEL: High
HUMOR: Predatory, Shorty caught Squacky and literally ate
       him for supper. Shorty should be isolated from the
       other animals until he can be checked out by
       Seapark's resident marine psychologist.
```

Let's also assume that you have modified the comma-delimited file so that instead of containing the note text, each line in that file contains the name of the note file to load. For example:

```
100,13-jan-2000,23.5,note_100.txt
105,13-jan-2000,99.45,note_105.txt
112,13-jan-2000,10,note_112.txt
151,13-jan-2000,55
166,13-jan-2000,17.5,note_166.txt
145,13-jan-2000,0,note_145.txt
175,13-jan-2000,35.5,note_175.txt
199,13-jan-2000,0.5,note_199.txt
202,13-jan-2000,22.0
240,13-jan-2000,28,note_240.txt
```

To load this data using SQL*Loader, do the following:

- Define a `FILLER` field to contain the file name. This field will not be loaded into the database.
- Define a `LOBFILE` field that loads the contents of the file identified by the `FILLER` field into the `CLOB` column.

The resulting control file is shown in Listing 10-8.

Listing 10-8: **Loading files into a large object column**

```
LOAD DATA
  INFILE 'animal_feeding_clob.csv'
    BADFILE 'animal_feeding_clob'
  APPEND
  INTO TABLE animal_feeding
    TRAILING NULLCOLS
    (
      animal_id       INTEGER EXTERNAL TERMINATED BY ",",
      feeding_date    DATE "dd-mon-yyyy" TERMINATED BY ",",
      pounds_eaten    DECIMAL EXTERNAL TERMINATED BY ",",
      note_file_name  FILLER CHAR TERMINATED BY ",",
      note            LOBFILE (note_file_name)
                      TERMINATED BY EOF
    )
```

Each time SQL*Loader inserts a record into the `animal_feeding` table, it will also store the entire contents of the associated note file in the note field.

Summary

In this chapter, you learned:

- SQL*Loader is a versatile utility for loading large amounts of data into an Oracle database.
- SQL*Loader control files are used to describe the data being loaded and to specify the table(s) into which that data is stored.
- You can use the `INFILE` clause to identify the file, or files, that you want SQL*Loader to read.
- You can use the `INTO TABLE` clause to identify the table, and the columns within that table, that you wish to populate using the data read from the input file.
- You can use the `APPEND` option after the `INFILE` clause to tell SQL*Loader to insert data into a table that already contains data to begin with.
- You can use SQL*Loader to load delimited data, such as comma-delimited data, or you can use it to load data stored in fixed-width columns.
- SQL*Loader fully supports all of Oracle8i's datatypes, even to the point of allowing you to populate `LOB` columns.

✦ ✦ ✦