6

SQL*Loader Concepts

This chapter explains the basic concepts of loading data into an Oracle database with SQL*Loader. This chapter covers the following topics:

SQL*Loader Features

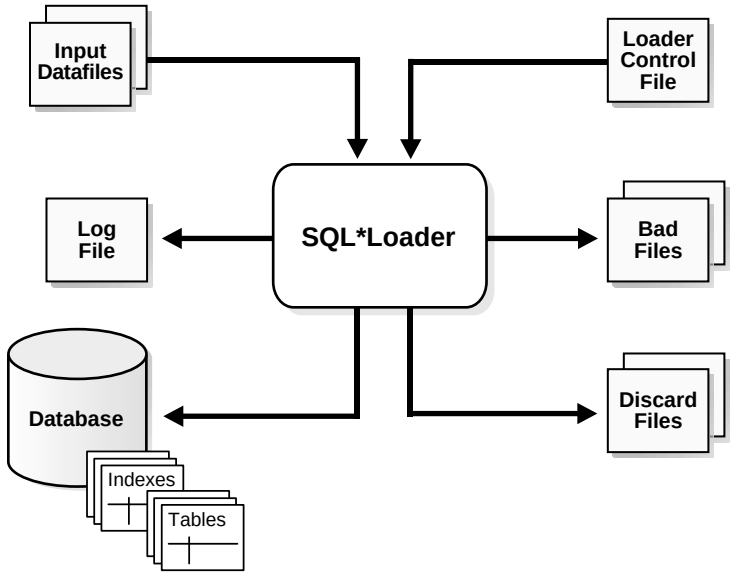
SQL*Loader loads data from external files into tables of an Oracle database. It has a powerful data parsing engine that puts little limitation on the format of the data in the datafile. You can use SQL*Loader to do the following:

- Load data across a network. This means that you can run the SQL*Loader client on a different system from the one that is running the SQL*Loader server.

- Load data from multiple datafiles during the same load session.
- Load data into multiple tables during the same load session.
- Specify the character set of the data.
- Selectively load data (you can load records based on the records' values).
- Manipulate the data before loading it, using SQL functions.
- Generate unique sequential key values in specified columns.
- Use the operating system's file system to access the datafiles.
- Load data from disk, tape, or named pipe.
- Generate sophisticated error reports, which greatly aid troubleshooting.
- Load arbitrarily complex object-relational data.
- Use secondary datafiles for loading LOBs and collections.
- Use either conventional or direct path loading. While conventional path loading is very flexible, direct path loading provides superior loading performance. See Chapter 11.

A typical SQL*Loader session takes as input a control file, which controls the behavior of SQL*Loader, and one or more datafiles. The output of SQL*Loader is an Oracle database (where the data is loaded), a log file, a bad file, and potentially, a discard file. An example of the flow of a SQL*Loader session is shown in Figure 6–1.

*Figure 6–1 SQL*Loader Overview*



SQL*Loader Parameters

SQL*Loader is invoked when you specify the `sqlldr` command and, optionally, parameters that establish session characteristics.

In situations where you always use the same parameters for which the values seldom change, it can be more efficient to specify parameters using the following methods, rather than on the command line:

- Parameters can be grouped together in a parameter file. You could then specify the name of the parameter file on the command line using the `PARFILE` parameter.
- Certain parameters can also be specified within the SQL*Loader control file by using the `OPTIONS` clause.

Parameters specified on the command line override any parameter values specified in a parameter file or `OPTIONS` clause.

See Also:

- Chapter 7 for descriptions of the SQL*Loader parameters
- PARFILE (parameter file) on page 7-10
- OPTIONS Clause on page 8-4

SQL*Loader Control File

The control file is a text file written in a language that SQL*Loader understands. The control file tells SQL*Loader where to find the data, how to parse and interpret the data, where to insert the data, and more.

Although not precisely defined, a control file can be said to have three sections.

The first section contains sessionwide information, for example:

- Global options such as bindsize, rows, records to skip, and so on
- `INFILE` clauses to specify where the input data is located
- Data to be loaded

The second section consists of one or more `INTO TABLE` blocks. Each of these blocks contains information about the table into which the data is to be loaded, such as the table name and the columns of the table.

The third section is optional and, if present, contains input data.

Some control file syntax considerations to keep in mind are:

- The syntax is free-format (statements can extend over multiple lines).
- It is case insensitive; however, strings enclosed in single or double quotation marks are taken literally, including case.
- In control file syntax, comments extend from the two hyphens (--) that mark the beginning of the comment to the end of the line. The optional third section of the control file is interpreted as data rather than as control file syntax; consequently, comments in this section are not supported.
- The keywords `CONSTANT` and `ZONE` have special meaning to SQL*Loader and are therefore reserved. To avoid potential conflicts, Oracle recommends that you do not use either `CONSTANT` or `ZONE` as a name for any tables or columns.

See Also: Chapter 8 for details about control file syntax and semantics

Input Data and Datafiles

SQL*Loader reads data from one or more files (or operating system equivalents of files) specified in the control file. From SQL*Loader's perspective, the data in the datafile is organized as *records*. A particular datafile can be in fixed record format, variable record format, or stream record format. The record format can be specified in the control file with the `INFILE` parameter. If no record format is specified, the default is stream record format.

Note: If data is specified inside the control file (that is, `INFILE *` was specified in the control file), then the data is interpreted in the stream record format with the default record terminator.

Fixed Record Format

A file is in fixed record format when all records in a datafile are the same byte length. Although this format is the least flexible, it results in better performance than variable or stream format. Fixed format is also simple to specify. For example:

```
INFILE datafile_name "fix n"
```

This example specifies that SQL*Loader should interpret the particular datafile as being in fixed record format where every record is *n* bytes long.

Example 6–1 shows a control file that specifies a datafile that should be interpreted in the fixed record format. The datafile in the example contains five physical records. Assuming that a period (.) indicates a space, the first physical record is [001,...cd,.] which is exactly eleven bytes (assuming a single-byte character set). The second record is [0002,fghi,\n] followed by the newline character (which is the eleventh byte), and so on. Note that newline characters are not required with the fixed record format.

Note that the length is always interpreted in bytes, even if character-length semantics are in effect for the file. This is necessary because the file could contain a mix of fields, some of which are processed with character-length semantics and others which are processed with byte-length semantics. See Character-Length Semantics on page 8-23.

Example 6–1 Loading Data in Fixed Record Format

```
load data
infile 'example.dat'  "fix 11"
into table example
```

```
fields terminated by ',' optionally enclosed by '"'
(col1, col2)

example.dat:
001,   cd, 0002,fghi,
00003,lmn,
1, "pqrs",
0005,uvwx,
```

Variable Record Format

A file is in variable record format when the length of each record in a character field is included at the beginning of each record in the datafile. This format provides some added flexibility over the fixed record format and a performance advantage over the stream record format. For example, you can specify a datafile that is to be interpreted as being in variable record format as follows:

```
INFILE "datafile_name" "var n"
```

In this example, *n* specifies the number of bytes in the record length field. If *n* is not specified, SQL*Loader assumes a length of 5 bytes. Specifying *n* larger than 40 will result in an error.

Example 6–2 shows a control file specification that tells SQL*Loader to look for data in the datafile `example.dat` and to expect variable record format where the record length fields are 3 bytes long. The `example.dat` datafile consists of three physical records. The first is specified to be 009 (that is, 9) bytes long, the second is 010 bytes long (that is, 10, including a 1-byte newline), and the third is 012 bytes long (also including a 1-byte newline). Note that newline characters are not required with the variable record format. This example also assumes a single-byte character set for the datafile.

The lengths are always interpreted in bytes, even if character-length semantics are in effect for the file. This is necessary because the file could contain a mix of fields, some processed with character-length semantics and others processed with byte-length semantics. See Character-Length Semantics on page 8-23.

Example 6–2 Loading Data in Variable Record Format

```
load data
infile 'example.dat'  "var 3"
into table example
fields terminated by ',' optionally enclosed by '"'
(col1 char(5),
```

```
 col2 char(7))

example.dat:
009hello,cd,010world,im,
012my,name is,
```

Stream Record Format

A file is in stream record format when the records are not specified by size; instead SQL*Loader forms records by scanning for the *record terminator*. Stream record format is the most flexible format, but there can be a negative effect on performance. The specification of a datafile to be interpreted as being in stream record format looks similar to the following:

```
INFILE datafile_name ["str terminator_string"]
```

The *terminator_string* is specified as either '*char_string*' or *X'hex_string*' where:

- '*char_string*' is a string of characters enclosed in single or double quotation marks
- *X'hex_string*' is a byte string in hexadecimal format

When the *terminator_string* contains special (nonprintable) characters, it should be specified as a *X'hex_string*'. However, some nonprintable characters can be specified as ('*char_string*') by using a backslash. For example:

- \n indicates a line feed
- \t indicates a horizontal tab
- \f indicates a form feed
- \v indicates a vertical tab
- \r indicates a carriage return

If the character set specified with the NLS_LANG parameter for your session is different from the character set of the datafile, character strings are converted to the character set of the datafile. This is done before SQL*Loader checks for the default record terminator.

Hexadecimal strings are assumed to be in the character set of the datafile, so no conversion is performed.

On UNIX-based platforms, if no *terminator_string* is specified, SQL*Loader defaults to the line feed character, \n.

On Windows NT, if no *terminator_string* is specified, then SQL*Loader uses either `\n` or `\r\n` as the record terminator, depending on which one it finds first in the datafile. This means that if you know that one or more records in your datafile has `\n` embedded in a field, but you want `\r\n` to be used as the record terminator, you must specify it.

Example 6–3 illustrates loading data in stream record format where the terminator string is specified using a character string, `'|\n'`. The use of the backslash character allows the character string to specify the nonprintable line feed character.

Example 6–3 Loading Data in Stream Record Format

```
load data
infile 'example.dat'  "str '|\n'"
into table example
fields terminated by ',' optionally enclosed by '"'
(col1 char(5),
 col2 char(7))

example.dat:
hello,world,|
james,bond,|
```

Logical Records

SQL*Loader organizes the input data into physical records, according to the specified record format. By default a physical record is a logical record, but for added flexibility, SQL*Loader can be instructed to combine a number of physical records into a logical record.

SQL*Loader can be instructed to follow one of the following logical record-forming strategies:

- Combine a fixed number of physical records to form each logical record.
- Combine physical records into logical records while a certain condition is true.

See Also:

- Assembling Logical Records from Physical Records on page 8-27
- Case Study 4: Loading Combined Physical Records on page 12-14 for an example of how to use continuation fields to form one logical record from multiple physical records

Data Fields

Once a logical record is formed, field setting on the logical record is done. Field setting is a process in which SQL*Loader uses control-file field specifications to determine which parts of logical record data correspond to which control-file fields. It is possible for two or more field specifications to claim the same data. Also, it is possible for a logical record to contain data that is not claimed by any control-file field specification.

Most control-file field specifications claim a particular part of the logical record. This mapping takes the following forms:

- The byte position of the data field's beginning, end, or both, can be specified. This specification form is not the most flexible, but it provides high field-setting performance.
- The strings delimiting (enclosing and / or terminating) a particular data field can be specified. A delimited data field is assumed to start where the last data field ended, unless the byte position of the start of the data field is specified.
- The byte offset and / or the length of the data field can be specified. This way each field starts a specified number of bytes from where the last one ended and continues for a specified length.
- Length-value datatypes can be used. In this case, the first *n* number of bytes of the data field contain information about how long the rest of the data field is.

See Also:

- Specifying the Position of a Data Field on page 9-3
- Specifying Delimiters on page 9-25

LOBFILEs and Secondary Datafiles (SDFs)

LOB data can be lengthy enough that it makes sense to load it from a LOBFILE. In LOBFILEs, LOB data instances are still considered to be in fields (predetermined size, delimited, length-value), but these fields are not organized into records (the concept of a record does not exist within LOBFILEs). Therefore, the processing overhead of dealing with records is avoided. This type of organization of data is ideal for LOB loading.

For example, you might use LOBFILEs to load employee names, employee IDs, and employee resumes. You could read the employee names and IDs from the main datafiles and you could read the resumes, which can be quite lengthy, from LOBFILEs.

You might also use LOBFILEs to facilitate the loading of XML data. You can use `XML` columns to hold data that models structured and semistructured data. Such data can be quite lengthy.

Secondary datafiles (SDFs) are similar in concept to primary datafiles. Like primary datafiles, SDFs are a collection of records, and each record is made up of fields. The SDFs are specified on a per control-file-field basis. Only a `collection_fld_spec` can name an SDF as its data source.

SDFs are specified using the `SDF` parameter. The `SDF` parameter can be followed by either the file specification string, or a `FILLER` field that is mapped to a data field containing one or more file specification strings.

See Also:

- Loading LOB Data from LOBFILEs on page 10-22
- Secondary Datafiles (SDFs) on page 10-32

Data Conversion and Datatype Specification

During a conventional path load, *data fields* in the datafile are converted into *columns* in the database (direct path loads are conceptually similar, but the implementation is different). There are two conversion steps:

1. SQL*Loader uses the field specifications in the control file to interpret the format of the datafile, parse the input data, and populate the bind arrays that correspond to a SQL `INSERT` statement using that data.
2. The Oracle database accepts the data and executes the `INSERT` statement to store the data in the database.

The Oracle database uses the datatype of the column to convert the data into its final, stored form. Keep in mind the distinction between a *field* in a datafile and a *column* in the database. Remember also that the *field datatypes* defined in a SQL*Loader control file are *not* the same as the *column datatypes*.

Discarded and Rejected Records

Records read from the input file might not be inserted into the database. Such records are placed in either a bad file or a discard file.

The Bad File

The bad file contains records that were rejected, either by SQL*Loader or by the Oracle database. Some of the possible reasons for rejection are discussed in the next sections.

SQL*Loader Rejects

Datafile records are rejected by SQL*Loader when the input format is invalid. For example, if the second enclosure delimiter is missing, or if a delimited field exceeds its maximum length, SQL*Loader rejects the record. Rejected records are placed in the bad file.

Oracle Database Rejects

After a datafile record is accepted for processing by SQL*Loader, it is sent to the Oracle database for insertion into a table as a row. If the Oracle database determines that the row is valid, then the row is inserted into the table. If the row is determined to be invalid, then the record is rejected and SQL*Loader puts it in the bad file. The row may be invalid, for example, because a key is not unique, because a required field is null, or because the field contains invalid data for the Oracle datatype.

See Also:

- Specifying the Bad File on page 8-12
- Case Study 4: Loading Combined Physical Records on page 12-14 for an example use of a bad file

The Discard File

As SQL*Loader executes, it may create a file called the discard file. This file is created only when it is needed, and only if you have specified that a discard file should be enabled. The discard file contains records that were filtered out of the load because they did not match any record-selection criteria specified in the control file.

The discard file therefore contains records that were not inserted into any table in the database. You can specify the maximum number of such records that the discard file can accept. Data written to any database table is not written to the discard file.

See Also:

- Case Study 4: Loading Combined Physical Records on page 12-14
- Specifying the Discard File on page 8-14

Log File and Logging Information

When SQL*Loader begins execution, it creates a *log file*. If it cannot create a log file, execution terminates. The log file contains a detailed summary of the load, including a description of any errors that occurred during the load.

See Also: Chapter 12, "SQL*Loader Case Studies" for sample log files

Conventional Path Loads, Direct Path Loads, and External Table Loads

SQL*Loader provides the following methods to load data:

- Conventional Path Loads
- Direct Path Loads
- External Table Loads

Conventional Path Loads

During conventional path loads, the input records are parsed according to the field specifications, and each data field is copied to its corresponding bind array. When the bind array is full (or no more data is left to read), an array insert is executed.

See Also:

- Data Loading Methods on page 11-1
- Bind Arrays and Conventional Path Loads on page 8-45

SQL*Loader stores LOB fields after a bind array insert is done. Thus, if there are any errors in processing the LOB field (for example, the LOBFILE could not be found), the LOB field is left empty. Note also that because LOB data is loaded after the array insert has been performed, BEFORE and AFTER row triggers may not work as expected for LOB columns. This is because the triggers fire before SQL*Loader has a chance to load the LOB contents into the column. For instance, suppose you are

loading a LOB column, C1, with data and that you want a BEFORE row trigger to examine the contents of this LOB column and derive a value to be loaded for some other column, C2, based on its examination. This is not possible because the LOB contents will not have been loaded at the time the trigger fires.

Direct Path Loads

A direct path load parses the input records according to the field specifications, converts the input field data to the column datatype, and builds a column array. The column array is passed to a block formatter, which creates data blocks in Oracle database block format. The newly formatted database blocks are written directly to the database, bypassing much of the data processing that normally takes place. Direct path load is much faster than conventional path load, but entails several restrictions.

See Also: Direct Path Load on page 11-5

Parallel Direct Path

A parallel direct path load allows multiple direct path load sessions to concurrently load the same data segments (allows intrasegment parallelism). Parallel direct path is more restrictive than direct path.

See Also: Parallel Data Loading Models on page 11-31

External Table Loads

An external table load creates an external table for data in a datafile and executes INSERT statements to insert the data from the datafile into the target table.

The advantages of using external table loads over conventional path and direct path loads are as follows:

- An external table load attempts to load datafiles in parallel. If a datafile is big enough, it will attempt to load that file in parallel.
- An external table load allows modification of the data being loaded by using SQL functions and PL/SQL functions as part of the INSERT statement that is used to create the external table.

See Also:

- Chapter 13, "External Tables Concepts"
- Chapter 14, "The ORACLE_LOADER Access Driver"

Choosing External Tables Versus SQL*Loader

The record parsing of external tables and SQL*Loader is very similar, so normally there is not a major performance difference for the same record format. However, due to the different architecture of external tables and SQL*Loader, there are situations in which one method is more appropriate than the other.

In the following situations, use external tables for the best load performance:

- You want to transform the data as it is being loaded into the database.
- You want to use transparent parallel processing without having to split the external data first.

However, in the following situations, use SQL*Loader for the best load performance:

- You want to load data remotely.
- Transformations are not required on the data, and the data does not need to be loaded in parallel.

Loading Objects, Collections, and LOBs

You can use SQL*Loader to bulk load objects, collections, and LOBs. It is assumed that you are familiar with the concept of objects and with Oracle's implementation of object support as described in *Oracle Database Concepts* and in the *Oracle Database Administrator's Guide*.

Supported Object Types

SQL*Loader supports loading of the following two object types:

column objects

When a column of a table is of some object type, the objects in that column are referred to as column objects. Conceptually such objects are stored in their entirety in a single column position in a row. These objects do not have object identifiers and cannot be referenced.

If the object type of the column object is declared to be nonfinal, then SQL*Loader allows a derived type (or subtype) to be loaded into the column object.

row objects

These objects are stored in tables, known as object tables, that have columns corresponding to the attributes of the object. The object tables have an additional system-generated column, called SYS_NC_OID$, that stores system-generated unique identifiers (OIDs) for each of the objects in the table. Columns in other tables can refer to these objects by using the OIDs.

If the object type of the object table is declared to be nonfinal, then SQL*Loader allows a derived type (or subtype) to be loaded into the row object.

See Also:

- Loading Column Objects on page 10-1
- Loading Object Tables on page 10-12

Supported Collection Types

SQL*Loader supports loading of the following two collection types:

Nested Tables

A nested table is a table that appears as a column in another table. All operations that can be performed on other tables can also be performed on nested tables.

VARRAYs

VARRAYs are variable sized arrays. An array is an ordered set of built-in types or objects, called elements. Each array element is of the same type and has an index, which is a number corresponding to the element's position in the VARRAY.

When creating a VARRAY type, you must specify the maximum size. Once you have declared a VARRAY type, it can be used as the datatype of a column of a relational table, as an object type attribute, or as a PL/SQL variable.

See Also: Loading Collections (Nested Tables and VARRAYs) on page 10-29 for details on using SQL*Loader control file data definition language to load these collection types

Supported LOB Types

A LOB is a large object type. This release of SQL*Loader supports loading of four LOB types:

- BLOB: a LOB containing unstructured binary data

- CLOB: a LOB containing character data
- NCLOB: a LOB containing characters in a database national character set
- BFILE: a BLOB stored outside of the database tablespaces in a server-side operating system file

LOBs can be column datatypes, and with the exception of the NCLOB, they can be an object's attribute datatypes. LOBs can have an actual value, they can be null, or they can be "empty."

> **See Also:** Loading LOBs on page 10-18 for details on using SQL*Loader control file data definition language to load these LOB types

Partitioned Object Support

SQL*Loader supports loading partitioned objects in the database. A partitioned object in an Oracle database is a table or index consisting of partitions (pieces) that have been grouped, typically by common logical attributes. For example, sales data for the year 2000 might be partitioned by month. The data for each month is stored in a separate partition of the sales table. Each partition is stored in a separate segment of the database and can have different physical attributes.

SQL*Loader partitioned object support enables SQL*Loader to load the following:

- A single partition of a partitioned table
- All partitions of a partitioned table
- A nonpartitioned table

Application Development: Direct Path Load API

Oracle provides a direct path load API for application developers. See the *Oracle Call Interface Programmer's Guide* for more information.