7

Understanding How to Use SQL*Loader

Learn about the basic concepts you should understand before loading data into an Oracle Database using SQL*Loader.

- SQL*Loader Features
  SQL*Loader loads data from external files into Oracle Database tables.
- SQL*Loader Parameters
  SQL*Loader is started either when you specify the `sqlldr` command, or when you specify parameters that establish various characteristics of the load operation.
- SQL*Loader Control File
  The control file is a text file written in a language that SQL*Loader understands.
- Input Data and Data Fields in SQL*Loader
  Learn how SQL*Loader loads data and identifies record fields.
- LOBFILEs and Secondary Data Files (SDFs)
  Large Object (LOB) data can be lengthy enough that it makes sense to load it from a LOBFILE.
- Data Conversion and Data Type Specification
  During a conventional path load, *data fields* in the data file are converted into *columns* in the database (direct path loads are conceptually similar, but the implementation is different).
- SQL*Loader Discarded and Rejected Records
  SQL*Loader can reject or discard some records read from the input file, either because of issues with the files, or because you have selected to filter the records out of the load.
- Log File and Logging Information
  When SQL*Loader begins processing, it creates a **log file.**
- Conventional Path Loads, Direct Path Loads, and External Table Loads
  SQL*Loader provides several methods to load data.
- Loading Objects, Collections, and LOBs with SQL*Loader
  You can bulk-load the column, row, LOB, and JSON database objects that you need to model real-world entities, such as customers and purchase orders.
- Partitioned Object Support in SQL*Loader
  Partitioned database objects enable you to manage sections of data, either collectively or individually. SQL*Loader supports loading partitioned objects.
- Application Development: Direct Path Load API
  Direct path loads enable you to load data from external files into tables and partitions.Oracle provides a direct path load API for application developers.
- SQL*Loader Case Studies
  To learn how you can use SQL*Loader features, you can run a variety of case studies that Oracle provides.

7.1 SQL*Loader Features

SQL*Loader loads data from external files into Oracle Database tables.

SQL*Loader has a powerful data parsing engine that puts few limitations on the format of the data in the data file. You can use SQL*Loader to do the following:

- Load data across a network, if your data files are on a different system than the database.
- Load data from multiple data files during the same load session.
- Load data into multiple tables during the same load session.
- Load data from large tables using automatic parallel loading, for both direct path and conventional path loading, and for both single tables and sharded tables.
- 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 data files.
- Load data from disk, tape, or named pipe.
- Generate sophisticated error reports, which greatly aid troubleshooting.
- Load arbitrarily complex object-relational data.
- Use secondary data files for loading Large Objects (LOBs) and collections.
- Use conventional, direct path, or external table loads.

LOBs are used to hold large amounts of data inside Oracle Database. SQL*Loader and external tables use LOBFILEs. Data for a LOB can be very large, and not fit in line in a SQL*Loader data file. Also, if the file contains binary data, then it can’t be in line. Instead, the data file has the name of a file containing the data for the LOB field. In that case, SQL*Loader and the external table code open the LOBFILE, and load the contents into the LOB column for the current row. The data is then passed to the server, just as with data for any other column type.

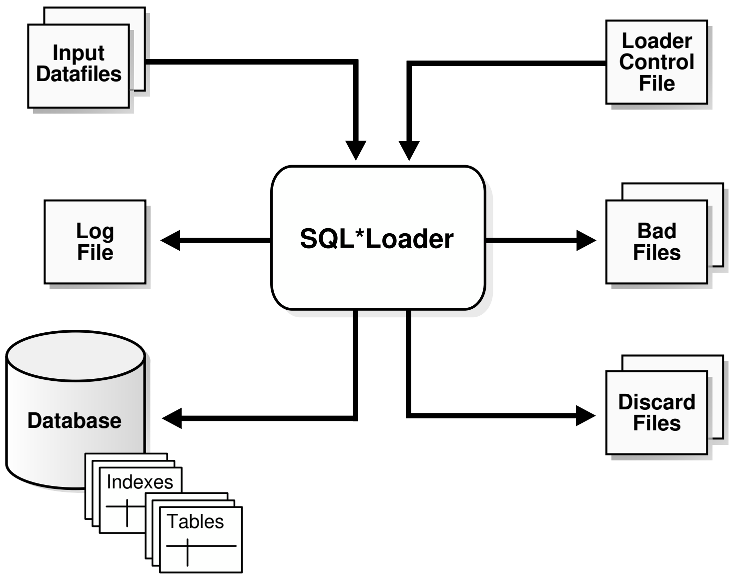
JSON columns can be loaded using the same methods used to load scalars and LOBs

You can use SQL*Loader in two ways: with or without a control file. A control file controls the behavior of SQL*Loader and one or more data files used in the load. Using a control file gives you more control over the load operation, which might be desirable for more complicated load situations. But for simple loads, you can use SQL*Loader without specifying a control file; this is referred to as SQL*Loader express mode.

The output of SQL*Loader is an Oracle Database database (where the data is loaded), a log file, a bad file if there are rejected records, and potentially, a discard file.

The following figure shows an example of the flow of a typical SQL*Loader session that uses a control file.

Figure 7-1   SQL*Loader Overview



Related Topics

- Conventional Path Loads, Direct Path Loads, and External Table Loads
  SQL*Loader provides several methods to load data.
- SQL*Loader Express
  SQL*Loader express mode allows you to quickly and easily use SQL*Loader to load simple data types.

7.2 SQL*Loader Parameters

SQL*Loader is started either when you specify the `sqlldr` command, or when you specify parameters that establish various characteristics of the load operation.

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

- You can group parameters together in a parameter file. You can then specify the name of the parameter file on the command line by using the `PARFILE` parameter.
- You can specify some parameters 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.

Related Topics

- SQL*Loader Command-Line Reference
  To start regular SQL*Loader, use the command-line parameters.
- PARFILE
  The `PARFILE` SQL*Loader command-line parameter specifies the name of a file that contains commonly used command-line parameters.

- OPTIONS Clause for Schema Data
  The following SQL*Loader command-line parameters can be specified using the `OPTIONS` clause.

7.3 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.

In general, the control file has three main sections, in the following order:

- Session-wide information
- Table and field-list information
- Input data (optional section)

Some control file syntax considerations to keep in mind are:

- The syntax is free-format (statements can extend over multiple lines).
- The syntax 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.

Related Topics

- SQL*Loader Control File Reference
  The SQL*Loader control file is a text file that contains data definition language (DDL) instructions for a SQL*Loader job.

7.4 Input Data and Data Fields in SQL*Loader

Learn how SQL*Loader loads data and identifies record fields.

- How SQL*Loader Reads Input Data and Data Files
  SQL*Loader reads data from one or more data files (or operating system equivalents of files) specified in the control file.
- Fixed Record Format
  A file is in fixed record format when all records in a data file are the same byte length.
- Variable Record Format and SQL*Loader
  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 data file.
- Stream Record Format and SQL*Loader
  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**.

- Logical Records and SQL*Loader
  SQL*Loader organizes input data into physical records, according to the specified record format. By default, a physical record is a logical record.
- Data Field Setting and SQL*Loader
  Learn how SQL*Loader determines the field setting on the logical record after a logical record is formed.

7.4.1 How SQL*Loader Reads Input Data and Data Files

SQL*Loader reads data from one or more data files (or operating system equivalents of files) specified in the control file.

From SQL*Loader's perspective, the data in the data file is organized as *records*. A particular data file 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, then 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.

7.4.2 Fixed Record Format

A file is in fixed record format when all records in a data file are the same byte length.

Although the fixed record format is the least flexible format, using 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 data file as being in fixed record format where every record is *n* bytes long.

The following example shows a control file that specifies a data file (`example1.dat`) to be interpreted in the fixed record format. The data file in the example contains five physical records; each record has fields that contain the number and name of an employee. Each of the five records is 11 bytes long, including spaces. For the purposes of explaining this example, periods are used to represent spaces in the records, but in the actual records there would be no periods. With that in mind, the first physical record is `396,...ty,.` which is exactly eleven bytes (assuming a single-byte character set). The second record is `4922,beth,` followed by the newline character (`\n`) which is the eleventh byte, and so on. (Newline characters are not required with the fixed record format; it is simply used here to illustrate that if used, it counts as a byte in the record length.)

Example 7-1 Loading Data in Fixed Record Format

Loading data:

```
load data
infile 'example1.dat'  "fix 11"
into table example
fields terminated by ',' optionally enclosed by '"'
(col1, col2)
```

Contents of `example1.dat`:

```
396,...ty,.4922,beth,\n
68773,ben,.
1,.."dave",
5455,mike,.
```

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 can contain a mix of fields. Some are processed with character-length semantics, and others are processed with byte-length semantics.

Related Topics

- Character-Length Semantics
  Byte-length semantics are the default for all data files except those that use the UTF16 character set (which uses character-length semantics by default).

7.4.3 Variable Record Format and SQL*Loader

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 data file.

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 data file 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, then SQL*Loader assumes a length of 5 bytes. Specifying *n* larger than 40 results in an error.

The following example shows a control file specification that tells SQL*Loader to look for data in the data file `example2.dat` and to expect variable record format where the record's first three bytes indicate the length of the field. The `example2.dat` data file consists of three physical records. The first is specified to be 009 (9) bytes long, the second is 010 (10) bytes long (plus a 1-byte newline), and the third is 012 (12) bytes long (plus 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 data file. For the purposes of this example, periods in `example2.dat` represent spaces; the fields do not contain actual periods.

Example 7-2 Loading Data in Variable Record Format

Loading data:

```
load data
infile 'example2.dat'  "var 3"
into table example
fields terminated by ',' optionally enclosed by '"'
(col1 char(5),
 col2 char(7))
```

Contents of `example2.dat`:

```
009.396,.ty,0104922,beth,01268773,benji,
```

Note that the lengths are always interpreted in bytes, even if character-length semantics are in effect for the file. This is necessary because the file can contain a mix of fields, some processed with character-length semantics and others processed with byte-length semantics.

Related Topics

- Character-Length Semantics
  Byte-length semantics are the default for all data files except those that use the UTF16 character set (which uses character-length semantics by default).

7.4.4 Stream Record Format and SQL*Loader

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 using it can result in a negative effect on performance. The specification of a data file to be interpreted as being in stream record format looks similar to the following:

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

In the preceding example, `str` indicates that the file is in stream record format. 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`*`'` byte string. However, you can specify some nonprintable characters 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` initialization parameter for your session is different from the character set of the data file, then character strings are converted to the character set of the data file. This is done before SQL*Loader checks for the default record terminator.

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

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

On Windows-based platforms, 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 data file. This means that if you know that one or more records in your data file has `\n` embedded in a field, but you want `\r\n` to be used as the record terminator, then you must specify it.

The following example 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.

See Also:

- *Oracle Database Globalization Support Guide* for information about using the Language and Character Set File Scanner (LCSSCAN) utility to determine the language and character set for unknown file text

Example 7-3 Loading Data in Stream Record Format

Loading data:

```
load data
infile 'example3.dat'  "str '|\n'"
into table example
fields terminated by ',' optionally enclosed by '"'
(col1 char(5),
 col2 char(7))
```

```
example3.dat
```

```
396,ty,|
4922,beth,|
```

7.4.5 Logical Records and SQL*Loader

SQL*Loader organizes input data into physical records, according to the specified record format. By default, a physical record is a logical record.

For added flexibility, SQL*Loader can be instructed to combine several 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.

Related Topics

- Assembling Logical Records from Physical Records
  This section describes assembling logical records from physical records.
- SQL*Loader Case Studies
  To learn how you can use SQL*Loader features, you can run a variety of case studies that Oracle provides.

7.4.6 Data Field Setting and SQL*Loader

Learn how SQL*Loader determines the field setting on the logical record after a logical record is formed.

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, terminating, or both) 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.
- You can specify the byte offset, the length of the data field, or both. This way each field starts a specified number of bytes from where the last one ended and continues for a specified length.
- Length-value data types 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.

Starting with Oracle Database 23ai, you can use SQL*Loader to load schemaless documents (documents that lack a fixed data structure, such as JSON or XML-based application data) into Oracle Database as SODA collections.

Related Topics

- SODA Collections and SQL*Loader
  SQL*Loader enables you to load external documents into SODA collections using the SQL*Loader utility in both control file and express modes.
- Specifying Delimiters
  The boundaries of `CHAR`, datetime, interval, or numeric `EXTERNAL` fields can also be marked by delimiter characters contained in the input data record.

7.5 LOBFILEs and Secondary Data Files (SDFs)

Large Object (LOB) data can be lengthy enough that it makes sense to load it from a LOBFILE.

With LOBFILEs, LOB data instances are still considered to be in fields (predetermined size, delimited, length-value). However, 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, suppose you have a table that stores employee names, IDs, and their resumes. When loading this table, you can read the employee names and IDs from the main data files and you can read the resumes, which can be quite lengthy, from LOBFILEs.

You can 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 data files (SDFs) are similar in concept to primary data files. As with primary data files, SDFs are a collection of records, and each record is made up of fields. The SDFs are specified as needed for a control file field. Only a `collection_fld_spec` can name an SDF as its data source.

You specify SDFs by using the `SDF` parameter. You can enter a value for the `SDF` parameter either by using the file specification string, or by using a `FILLER` field that is mapped to a data field containing one or more file specification strings.

Related Topics

- Loading LOB Data from LOBFILEs
  To load large LOB data files, consider using a LOBFILE with SQL*Loader.
- Secondary Data Files (SDFs)
  When you need to load large nested tables and `VARRAY`s, you can use secondary data files (SDFs). They are similar in concept to primary data files.

7.6 Data Conversion and Data Type Specification

During a conventional path load, *data fields* in the data file 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 data file, parse the input data, and populate the bind arrays that correspond to a SQL `INSERT` statement using that data. A bind array is an area in memory where SQL*Loader stores data that is to be loaded. When the bind array is full, the data is transmitted to the database. The bind array size is controlled by the SQL*Loader `BINDSIZE` and `READSIZE` parameters.
2. The database accepts the data and executes the `INSERT` statement to store the data in the database.

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

See Also:

- BINDSIZE
- READSIZE

7.7 SQL*Loader Discarded and Rejected Records

SQL*Loader can reject or discard some records read from the input file, either because of issues with the files, or because you have selected to filter the records out of the load.

Rejected records are placed in a bad file, and discarded records are placed in a discard file.

- The SQL*Loader Bad File
  The bad file contains records that were rejected, either by SQL*Loader or by Oracle Database.
- The SQL*Loader Discard File
  As SQL*Loader runs, it can filter some records out of the load, and create a file called the discard file.

7.7.1 The SQL*Loader Bad File

The bad file contains records that were rejected, either by SQL*Loader or by Oracle Database.

If you do not specify a bad file, and there are rejected records, then SQL*Loader automatically creates one. A rejected record has the same name as the data file, with a `.bad` extension. There can be several causes for rejections.

- Records Rejected by SQL*Loader
  Data file records are rejected by SQL*Loader when the input format is invalid.
- Records Rejected by Oracle Database During a SQL*Loader Operation
  After a data file record is accepted for processing by SQL*Loader, it is sent to the database for insertion into a table as a row.

7.7.1.1 Records Rejected by SQL*Loader

Data file 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, then SQL*Loader rejects the record. Rejected records are placed in the bad file.

7.7.1.2 Records Rejected by Oracle Database During a SQL*Loader Operation

After a data file record is accepted for processing by SQL*Loader, it is sent to the database for insertion into a table as a row.

If the 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 data type.

7.7.2 The SQL*Loader Discard File

As SQL*Loader runs, it can filter some records out of the load, and create a file called the discard file.

A discard 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.

Because the discard file contains record filtered out of the load, the contents of the discard file are 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.

7.8 Log File and Logging Information

When SQL*Loader begins processing, it creates a **log file.**

If SQL*Loader cannot create a log file, then processing terminates. The log file contains a detailed summary of the load, including a description of any errors that occurred during the load.

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

SQL*Loader provides several methods to load data.

- 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 (an area in memory where SQL*Loader stores data to be loaded).
- Direct Path Loads
  A direct path load parses the input records according to the field specifications, converts the input field data to the column data type, and builds a column array.
- Parallel Direct Path
  A parallel direct path load allows multiple direct path load sessions to concurrently load the same data segments (allows intrasegment parallelism).
- External Table Loads
  External tables are defined as tables that do not reside in the database, and can be in any format for which an access driver is provided.
- Choosing External Tables Versus SQL*Loader
  Learn which method can provide the best load performance for your data load situations.
- Behavior Differences Between SQL*Loader and External Tables
  Oracle recommends that you review the differences between loading data with external tables, using the `ORACLE_LOADER` access driver, and loading data with SQL*Loader conventional and direct path loads.
- Loading Tables Using Data Stored into Object Storage
  Learn how to load your data from Object Storage into standard Oracle Database tables using SQL*Loader.

7.9.1 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 (an area in memory where SQL*Loader stores data to be loaded).

When the bind array is full (or no more data is left to read), an array insert operation is performed.

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), then 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 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.

See Also:

- Data Loading Methods
- Bind Arrays and Conventional Path Loads

7.9.2 Direct Path Loads

A direct path load parses the input records according to the field specifications, converts the input field data to the column data type, 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.

7.9.3 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

Direct Path Load

7.9.4 External Table Loads

External tables are defined as tables that do not reside in the database, and can be in any format for which an access driver is provided.

Oracle Database provides two access drivers: `ORACLE_LOADER`, and `ORACLE_DATAPUMP`. By providing the database with metadata describing an external table, the database is able to expose the data in the external table as if it were data residing in a regular database table.

An external table load creates an external table for data that is contained in an external data file. The load runs `INSERT` statements to insert the data from the data file into the target table.

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

- If a data file is big enough, then an external table load attempts 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.

> **Note:**
>
> An external table load is not supported using a named pipe on Windows operating systems.

Related Topics

- The ORACLE_LOADER Access Driver
  Learn how to control the way external tables are accessed by using the ORACLE_LOADER access driver parameters to modify the default behavior of the access driver.
- The ORACLE_DATAPUMP Access Driver
  The `ORACLE_DATAPUMP` access driver provides a set of access parameters that are unique to external tables of the type `ORACLE_DATAPUMP`.
- Managing External Tables in *Oracle Database Administrator's Guide*

7.9.5 Choosing External Tables Versus SQL*Loader

Learn which method can provide the best load performance for your data load situations.

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 may be more appropriate than the other.

Use external tables for the best load performance in the following situations:

- 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

Use SQL*Loader for the best load performance in the following situations:

- You want to load data remotely
- Transformations are not required on the data, and the data does not need to be loaded in parallel
- You want to load data, and additional indexing of the staging table is required

7.9.6 Behavior Differences Between SQL*Loader and External Tables

Oracle recommends that you review the differences between loading data with external tables, using the `ORACLE_LOADER` access driver, and loading data with SQL*Loader conventional and direct path loads.

The information in this section does not apply to the `ORACLE_DATAPUMP` access driver.

- Multiple Primary Input Data Files
  If there are multiple primary input data files with SQL*Loader loads, then a bad file and a discard file are created for each input data file.
- Syntax and Data Types
  With external table loads, you cannot use SQL*Loader to load unsupported syntax and data types.
- Byte-Order Marks
  With SQL*Loader, whether the byte-order mark is written depends on the character set or on the table load.
- Default Character Sets, Date Masks, and Decimal Separator
  The display of NLS character sets are controlled by different settings for SQL*Loader and external tables.
- Use of the Backslash Escape Character
  SQL*Loader and external tables use different conventions to identify single quotation marks as an enclosure character.

7.9.6.1 Multiple Primary Input Data Files

If there are multiple primary input data files with SQL*Loader loads, then a bad file and a discard file are created for each input data file.

With external table loads, there is only one bad file and one discard file for all input data files. If parallel access drivers are used for the external table load, then each access driver has its own bad file and discard file.

7.9.6.2 Syntax and Data Types

With external table loads, you cannot use SQL*Loader to load unsupported syntax and data types.

As part of your data migration plan, do not attempt to use SQL*Loader with unsupported syntax or data types. Resolve issues before your migration. You cannot use the following syntax or data types:

- Use of `CONTINUEIF` or `CONCATENATE` to combine multiple physical records into a single logical record.
- Loading of the following SQL*Loader data types: `GRAPHIC`, `GRAPHIC EXTERNAL`, and `VARGRAPHIC`
- Use of the following database column types: `LONG`, nested table, `VARRAY`, `REF`, primary key `REF`, and `SID`

Note:

All forms of `LONG` data types (`LONG`, `LONG RAW`, `LONG VARCHAR`, `LONG VARRAW`) were deprecated in Oracle8i Release 8.1.6. For succeeding releases, the `LONG` data type was provided for backward compatibility with existing applications. In new applications developed with later releases, Oracle strongly recommends that you use `CLOB` and `NCLOB` data types for large amounts of character data.

7.9.6.3 Byte-Order Marks

With SQL*Loader, whether the byte-order mark is written depends on the character set or on the table load.

If a primary data file uses a Unicode character set (`UTF8` or `UTF16`), and it also contains a byte-order mark (BOM), then the byte-order mark is written at the beginning of the corresponding bad and discard files.

With external table loads, the byte-order mark is not written at the beginning of the bad and discard files.

7.9.6.4 Default Character Sets, Date Masks, and Decimal Separator

The display of NLS character sets are controlled by different settings for SQL*Loader and external tables.

With SQL*Loader, the default character set, date mask, and decimal separator are determined by the settings of NLS environment variables on the client.

For fields in external tables, the database settings of the NLS parameters determine the default character set, date masks, and decimal separator.

7.9.6.5 Use of the Backslash Escape Character

SQL*Loader and external tables use different conventions to identify single quotation marks as an enclosure character.

With SQL*Loader, to identify a single quotation mark as the enclosure character, you can use the backslash (\) escape character. For example

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

In external tables, the use of the backslash escape character within a string raises an error. The workaround is to use double quotation marks to identify a single quotation mark as the enclosure character. For example:

```
TERMINATED BY ',' ENCLOSED BY "'"
```

7.9.7 Loading Tables Using Data Stored into Object Storage

Learn how to load your data from Object Storage into standard Oracle Database tables using SQL*Loader.

Starting with Oracle Database 21c, you can use the SQL*Loader parameter CREDENTIAL to provide credentials to enable read access to object stores. Parallel loading from the object store is supported.

For a data file, you can specify the URI for the data file that you want to read on the object store. The CREDENTIAL values specify credentials granted to the user running SQL*Loader. These permissions enable SQL*Loader to access the object.

> **Note:**
>
> Mixing local files with object store files is not supported.

In the following example, you have a table (`T`) into which you are loading data:

```
SQL> create table t (x int, y int);
```

You have a data file that you want to load to this table, named `file1.txt`. The contents are as follows:

```
X,Y
1,2
4,5
```

To load this table into an object store, complete the following procedure:

1. Install the libraries required to enable object store input/output (I/O):

   ```
   % cd $ORACLE_HOME/rdbms/lib
   % make -f ins_rdbms.mk opc_on
   ```

2. Upload the file `file1.txt` to the bucket in Object Storage.

   The easiest way to upload file to object storage is to upload the file from the Oracle Cloud console:

   a. Open the Oracle Cloud console.

   b. Select the Object Storage tile.

   c. If not already created, create a bucket.

   d. Click **Upload**, and select the file `file1.txt` to upload it into the bucket.

3. In Oracle Database, create the wallet and the credentials.

   For example:

   ```
   $ orapki wallet create -wallet /home/oracle/wallets  -pwd mypassword-
   auto_login
   $ mkstore -wrl /home/oracle/wallets -createEntry
   oracle.sqlldr.credential.myfedcredential.username
   oracleidentitycloudservice/myuseracct@example.com
   $ mkstore -wrl /home/oracle/wallets -createEntry
   oracle.sqlldr.credential.myfedcredential.password "MhAVCDfW+-ReskK4:Ho-
   zH"
   ```

   This example shows the use of a federated user account (*`myfedcredential`*). The password is automatically generated, as described in Oracle Cloud Infrastructure Documentation. "Managing Credentials," in the section "To create an auth token."

Note:

The `mkstore` wallet management command line tool is deprecated with Oracle Database 23ai, and can be removed in a future release.

To manage wallets, Oracle recommends that you use the `orapki` command line tool.

4. After creating the wallet, add the location in the `sqlnet.ora` file in the directory `$ORACLE_HOME/network/admin` directory.
   For example:

```
vi test.ctl
LOAD DATA
INFILE  'https://objectstorage.eu-frankfurt-1.oraclecloud.com/n/
dbcloudoci/b/myobjectstore/o/file1.txt'
truncate
INTO TABLE T
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(X,Y)
```

5. Run SQL*Loader to load the data into the object store.

   For example:

```
sqlldr test/mypassword@pdb1 /home/oracle/test.ctl
credential=myfedcredentiallog=test.log  external_table=not_used
```

Related Topics

- "Managing Credentials: To create an auth token," Oracle Cloud Infrastructure Documentation
- Using the Console, Oracle Cloud Infrastructure Documentation

7.10 Loading Objects, Collections, and LOBs with SQL*Loader

You can bulk-load the column, row, LOB, and JSON database objects that you need to model real-world entities, such as customers and purchase orders.

- Supported Object Types
  SQL*Loader supports loading of the column and row object types.
- Supported Collection Types
  SQL*Loader supports loading of nested tables and `VARRAY` collection types.
- SODA Collections and SQL*Loader
  SQL*Loader enables you to load external documents into SODA collections using the SQL*Loader utility in both control file and express modes.
- Supported LOB Data Types
  SQL*Loader supports multiple large object types (LOBs).

7.10.1 Supported Object Types

SQL*Loader supports loading of the column and row 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.
- row objects
  These objects are stored in tables, known as object tables, that have columns corresponding to the attributes of the object.

7.10.1.1 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.

7.10.1.2 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
- Loading Object Tables

7.10.2 Supported Collection Types

SQL*Loader supports loading of nested tables and `VARRAY` collection types.

- Nested Tables
  A nested table is a table that appears as a column in another table.
- VARRAYs
  A `VARRAY` is a variable sized arrays.

7.10.2.1 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.

7.10.2.2 VARRAYs

A `VARRAY` is a 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 you create a `VARRAY` type, you must specify the maximum size. Once you have declared a `VARRAY` type, it can be used as the data type 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) for details on using SQL*Loader control file data definition language to load these collection types

7.10.3 SODA Collections and SQL*Loader

SQL*Loader enables you to load external documents into SODA collections using the SQL*Loader utility in both control file and express modes.

Starting with Oracle Database 23ai, you can use SQL*Loader to load schemaless documents (documents that lack a fixed data structure, such as JSON or XML-based application data) into Oracle Database as SODA collections. A SODA (Simple Oracle Document Access) collection is a set of documents that is backed by an Oracle Database table or view. A document is stored in Oracle Database as a row in a table or view, with each component in its own column.

When you create a SODA document collection, the following is created in Oracle Database:

- Persistent default collection metadata.
- A table for storing the collection.

You can insert, append, and replace external documents into SODA collections in Oracle Database applications

To load a SODA collection, you supply one to three pieces of information to the SQL*Loader utility:

- `$CONTENT`: The content that you want to load (Required).

  This field can be an actual text document, or a secondary data file containing one or more documents. There are two types of content that you can specify:

  – `RAW(*)`: Use the `RAW(*)` data field either when text documents are stored directly in the control or data file, or when the documents are specified in the `INFILE` clause.

  – `CONTENTFILE(`*`soda_filename`*`)`: use the CONTENTFILE name to specify an secondary data file name `(`*`soda_filename`*`)` from which you want SQL*Loader to load the data. One or more documents can be contained in the secondary data file that you specify.

- `$KEY`: A key to identify the document (Optional)

  In a collection, each document must have a document key, which is unique for the collection. However, you do not need to provide a key if the SODA collection automatically generates keys. If `$KEY` is specified, then there is a one-to-one relationship between the key and the content.

- `$MEDIA`: A media type to describe the type of the content (Optional)
  `$MEDIA` is not required if the SODA collection is defined to hold documents of one media type. The default media type is JSON but this can be modified using the SODA_MEDIA keyword.

7.10.4 Supported LOB Data Types

SQL*Loader supports multiple large object types (LOBs).

This release of SQL*Loader supports loading of four LOB data 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 data types, and except for `NCLOB`, they can be an object's attribute data types. LOBs can have an actual value, they can be `null`, or they can be "empty."

JSON columns can be loaded using the same methods used to load scalars and LOBs

See Also:

Loading LOBs for details on using SQL*Loader control file data definition language to load these LOB types

7.11 Partitioned Object Support in SQL*Loader

Partitioned database objects enable you to manage sections of data, either collectively or individually. SQL*Loader supports loading partitioned objects.

A **partitioned object** in Oracle Database instances is a table or index consisting of partitions (pieces) that have been grouped, typically by common logical attributes. For example, sales data for a particular year 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

7.12 Application Development: Direct Path Load API

Direct path loads enable you to load data from external files into tables and partitions.Oracle provides a direct path load API for application developers.

Related Topics

- *Oracle Call Interface Developer's Guide*

7.13 SQL*Loader Case Studies

To learn how you can use SQL*Loader features, you can run a variety of case studies that Oracle provides.

- How to Access and Use the Oracle SQL*Loader Case Studies
  Oracle provides 11 case studies that illustrate features of SQL*Loader
- Case Study Files
  Each of the SQL*Loader case study files has a set of files required to use that case study
- Running the Case Studies
  The typical steps for running SQL*Loader case studies is similar for all of the cases.
- Case Study Log Files
  Log files for the case studies are not provided in the `$ORACLE_HOME/rdbms/demo` directory.
- Checking the Results of a Case Study
  To check the results of running a case study, start SQL*Plus and perform a select operation from the table that was loaded in the case study.

7.13.1 How to Access and Use the Oracle SQL*Loader Case Studies

Oracle provides 11 case studies that illustrate features of SQL*Loader

The case studies are based upon the Oracle demonstration database tables, `emp` and `dept`, owned by the user `scott`. (In some case studies, additional columns have been added.) The case studies are numbered 1 through 11, starting with the simplest scenario and progressing in complexity.

Note:

Files for use in the case studies are located in the `$ORACLE_HOME/rdbms/demo` directory. These files are installed when you install the Oracle Database Examples (formerly Companion) media.

The following is a summary of the case studies:

- Case Study 1: Loading Variable-Length Data - Loads stream format records in which the fields are terminated by commas and may be enclosed by quotation marks. The data is found at the end of the control file.
- Case Study 2: Loading Fixed-Format Fields - Loads data from a separate data file.
- Case Study 3: Loading a Delimited, Free-Format File - Loads data from stream format records with delimited fields and sequence numbers. The data is found at the end of the control file.
- Case Study 4: Loading Combined Physical Records - Combines multiple physical records into one logical record corresponding to one database row.
- Case Study 5: Loading Data into Multiple Tables - Loads data into multiple tables in one run.

- Case Study 6: Loading Data Using the Direct Path Load Method - Loads data using the direct path load method.
- Case Study 7: Extracting Data from a Formatted Report - Extracts data from a formatted report.
- Case Study 8: Loading Partitioned Tables - Loads partitioned tables.
- Case Study 9: Loading LOBFILEs (CLOBs) - Adds a `CLOB` column called `resume` to the table `emp`, uses a `FILLER` field (`res_file`), and loads multiple LOBFILEs into the `emp` table.
- Case Study 10: REF Fields and VARRAYs - Loads a customer table that has a primary key as its OID and stores order items in a `VARRAY`. Loads an order table that has a reference to the customer table and the order items in a `VARRAY`.
- Case Study 11: Loading Data in the Unicode Character Set - Loads data in the Unicode character set, UTF16, in little-endian byte order. This case study uses character-length semantics.

7.13.2 Case Study Files

Each of the SQL*Loader case study files has a set of files required to use that case study

Usage Notes

Generally, each case study is comprised of the following types of files:

- Control files (for example, `ulcase5.ctl`)
- Data files (for example, `ulcase5.dat`)
- Setup files (for example, `ulcase5.sql`)

These files are installed when you install the Oracle Database Examples (formerly Companion) media. They are installed in the directory `$ORACLE_HOME/rdbms/demo`.

If the example data for the case study is contained within the control file, then there is no `.dat` file for that case.

Case study 2 does not require any special set up, so there is no `.sql` script for that case. Case study 7 requires that you run both a starting (setup) script and an ending (cleanup) script.

The following table lists the files associated with each case:

Table 7-1 Case Studies and Their Related Files

Case	`.ctl`	`.dat`	`.sql`
1	ulcase1.ctl	N/A	ulcase1.sql
2	ulcase2.ctl	ulcase2.dat	N/A
3	ulcase3.ctl	N/A	ulcase3.sql
4	ulcase4.ctl	ulcase4.dat	ulcase4.sql
5	ulcase5.ctl	ulcase5.dat	ulcase5.sql
6	ulcase6.ctl	ulcase6.dat	ulcase6.sql
7	ulcase7.ctl	ulcase7.dat	ulcase7s.sql ulcase7e.sql
8	ulcase8.ctl	ulcase8.dat	ulcase8.sql
9	ulcase9.ctl	ulcase9.dat	ulcase9.sql

Table 7-1 (Cont.) Case Studies and Their Related Files

Case	`.ctl`	`.dat`	`.sql`
10	ulcase10.ctl	N/A	ulcase10.sql
11	ulcase11.ctl	ulcase11.dat	ulcase11.sql

7.13.3 Running the Case Studies

The typical steps for running SQL*Loader case studies is similar for all of the cases.

Be sure you are in the `$ORACLE_HOME/rdbms/demo` directory, which is where the case study files are located.

Also, be sure to read the control file for each case study before you run it. The beginning of the control file contains information about what is being demonstrated in the case study, and any other special information you need to know. For example, case study 6 requires that you add `DIRECT=TRUE` to the SQL*Loader command line.

1. At the system prompt, type `sqlplus` and press Enter to start SQL*Plus. At the user-name prompt, enter `scott`. At the password prompt, enter `tiger`.

   The SQL prompt is displayed.
2. At the SQL prompt, execute the SQL script for the case study. :

   For example, to execute the SQL script for case study 1, enter the following command:

   ```
   SQL> @ulcase1
   ```

   This command prepares and populates tables for the case study and then returns you to the system prompt.
3. At the system prompt, start SQL*Loader and run the case study.

   For example, to run case 1, enter the following command:

   ```
   sqlldr USERID=scott CONTROL=ulcase1.ctl LOG=ulcase1.log
   ```

   Substitute the appropriate control file name and log file name for the `CONTROL` and `LOG` parameters, and press **Enter**. When you are prompted for a password, type `tiger` and then press **Enter**.

7.13.4 Case Study Log Files

Log files for the case studies are not provided in the `$ORACLE_HOME/rdbms/demo` directory.

This is because the log file for each case study is produced when you execute the case study, provided that you use the `LOG` parameter. If you do not want to produce a log file, then omit the `LOG` parameter from the command line.

7.13.5 Checking the Results of a Case Study

To check the results of running a case study, start SQL*Plus and perform a select operation from the table that was loaded in the case study.

1. At the system prompt, type `sqlplus` and press Enter to start SQL*Plus. At the user-name prompt, enter `scott`. At the password prompt, enter `tiger`.

   The SQL prompt is displayed.

2. At the SQL prompt, use the `SELECT` statement to select all rows from the table that the case study loaded.

   For example, if you load the table `emp`, then enter the following statement:

   ```
   SQL> SELECT * FROM emp;
   ```

   The contents of each row in the `emp` table are displayed.