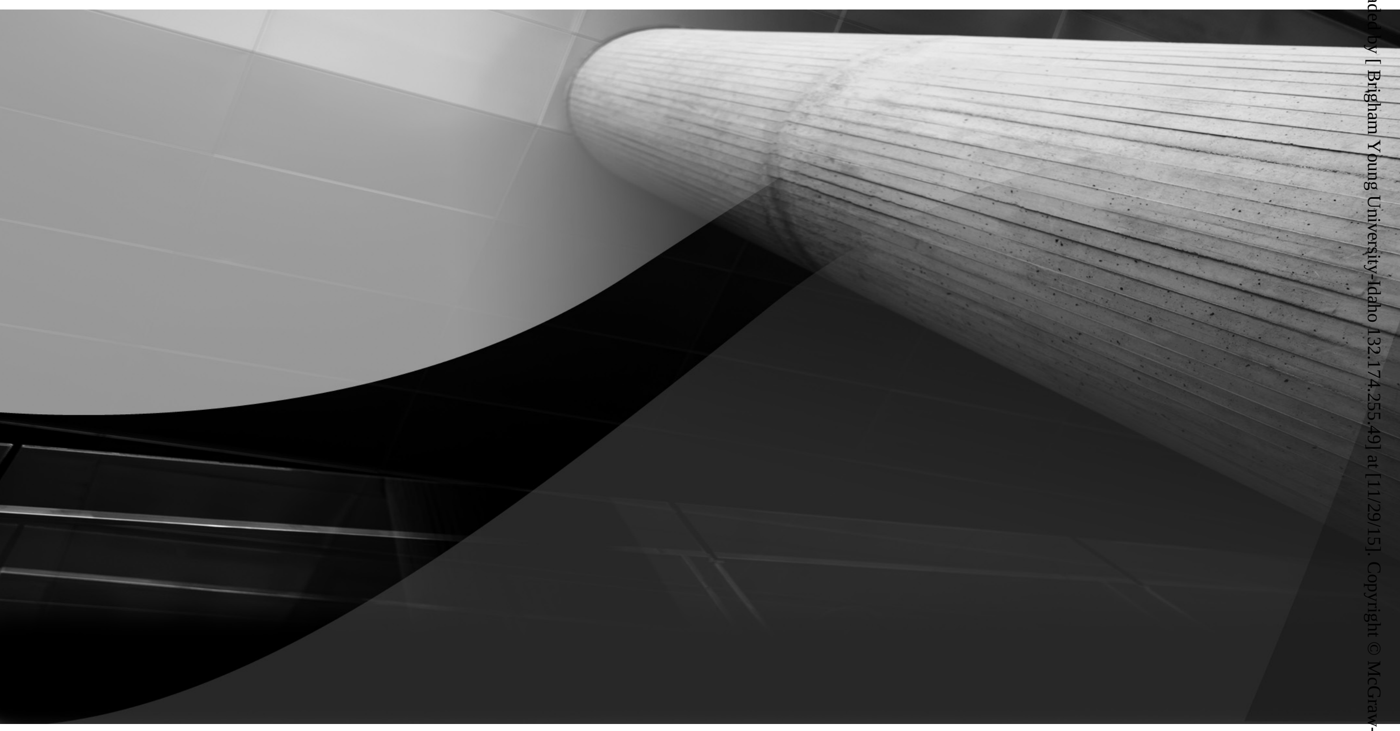

CHAPTER 12

Merging Data

Sometimes you need to insert new data and update existing data to relational tables during a bulk import of a physical file. You can accomplish this type of insert or update activity using a `MERGE` statement in Oracle and a `REPLACE INTO` statement in MySQL. MySQL also supports a merge operation with the `ON DUPLICATE KEY` clause of the `INSERT` statement (introduced in Chapter 7). This chapter covers these statements as methods for merging data in Oracle and MySQL.

Data import files often present interesting challenges, because they frequently contain rows of data that belongs in multiple tables (*denormalized data sets*). Import processes have to deal with this reality and discover the rules in order to break them up into normalized data sets. Only normalized data sets fit into the merge processes, because they work with single tables.

You need to understand why these files contain data from multiple tables. Although data modelers normalize information into single subject tables to avoid insertion, deletion, and update anomalies, analysts seldom use information in isolation. That's because the data is useful to them only when it's been assembled into information. Analysts typically apply business rules against data from a set of related tables. This means that they get these denormalized record sets from a query. As qualified in Chapter 11, queries let you assemble data into meaningful and actionable information, which makes the information as a whole greater than the sum of its data parts.

As developers of database-centric applications, we seldom have control over the origin of these files. Although it's not critical that we know where the files come from and why they're important, it can be helpful. Typically called *flat*, *loader*, or *batch import* files, these files are *import sources* that feed corrections and additions into our data repositories.

Import sources come from many places, such as business staff that sanitize data (analyzing data against business rules and fixing it) or other business partners. Business partners can be organizations within the corporation or company, or other companies with whom your company does business. The business staff sanitizing or exchanging data within the company are intra-company import sources. Intra-company imports can come from other IT organizations or from finance departments without professional IT staff. Those coming from other IT organizations are considered *business-to-business (B2B)* exchanges, and they package their outgoing files as export files. These export files support order management systems or financial systems. Professionally packaged export files are typically organized in an agreed upon XML (eXtensible Markup Language) or EDI (Electronic Data Interchange) formats.

Files coming from your company's internal finance, accounting, or operations departments are considered *consumer-to-business (C2B)* exchanges. That label works because the import files are typically comma-separated value (CSV) files from Microsoft Excel, which is what you would expect from a consumer (internal information consumer). These CSV files are typically managed by procedural programming interfaces.

This chapter demonstrates importing and merging data based on CSV and XML files on an *ad hoc* basis. The first section shows you how to use the `MERGE` statement and Oracle external tables as source files for the import source. You can flip back to Chapter 7 if you'd like to refresh your knowledge about external tables. The second section on MySQL shows you how to use the `LOAD DATA` and `LOAD XML` statements to get the values into a table, and then explains how to merge values with the `REPLACE INTO` statement and `INSERT` statement with the `ON DUPLICATE KEY` clause. You can flip back to Chapter 8 to read more about the `INSERT` statement than just its application demonstrated here.

Merging Data in Oracle

The MERGE statement in Oracle works similarly to the INSERT ALL statement presented in Chapter 8. It merges data from a query into a *target table* based on criteria evaluated in the ON subclause. The query in the subclause can be described as the *source result set*. The result set is like a virtual table, and virtual tables can be composed of data from multiple tables or a subset of data from one table. The results from a query can be thought of as the *source table*. The MERGE differs from the INSERT ALL statements because the WHEN clause only supports two logical conditions, MATCHED and NOT MATCHED, and the MERGE statement works with only one target table.

The MERGE statement has the following prototype:

```
MERGE INTO table_name
USING (select_statement) query_alias
ON ( condition_match [ {AND | OR } condition_match [...]] )
WHEN MATCHED THEN
update_statement
WHEN NOT MATCHED THEN
insert_statement;
```

Merging data from a source table to a target table requires that you know the columns that define the natural key of the target table, and in some cases the natural keys of all tables collapsed into the import source file. It also requires an outer join based on the natural key to ensure you always return the relative complement of the target table. The relative complement would be all the rows found in the source table that aren't found in the target table. The set of rows in the relative complement exists only when you're adding new rows from an import source file.

External data sets seldom have a copy of surrogate keys, because those columns aren't useful to analysts working with the data. The absence of surrogate keys means that you need to use the natural keys to determine how to get parts of the import source into their respective normalized tables. An outer join based on the natural key in the source query always returns a null value as the surrogate key. This is helpful for two reasons: the lack of a surrogate key identifies new rows, and surrogate keys can be auto-generated by available sequences.

You attempt a surrogate key match in the ON clause of a MERGE statement, and the ON DUPLICATE KEY clause of the INSERT statement assumes the same match when the surrogate key column is the table's primary key value. MySQL's REPLACE INTO statement performs a unique row match to find duplicates and inserts only new rows. New rows have a natural key that doesn't already exist in the table, and those are the only rows inserted by the REPLACE INTO statement.

Although not all import source files contain new rows, most do. The following sections contain the steps necessary for importing or modifying data through bulk uploads. Bulk imports are frequently accomplished through the use of externally organized tables, which were introduced in Chapter 6. An externally organized table is a table that points to a flat file (often a CSV file) deployed on the operating system. External tables in Oracle require that a DBA set up virtual directories and grants.

MySQL doesn't support externally organized tables, but it does support LOAD DATA and LOAD XML statements to reach out to the operating system and read a flat file into a table. Both databases require the system administrator to prepare and secure a directory in the operating system to support these flat files.

Step 1: Create a Virtual Directory

You must create at least one virtual directory to use external tables. Creating virtual directories and granting privileges to read and write to them are tasks reserved to the `SYS` or `SYSTEM` super user. The following lets a super user create an `upload` virtual directory and grant privileges to the `student` user to read and write to the directory:

```
CREATE DIRECTORY upload AS 'C:\import\upload';
GRANT READ, WRITE ON DIRECTORY upload TO student;
```

For this example, both read and write privileges are granted to the `student` user because the file needs to read the import source file and write log files from and to the same directory. Reading and writing files from the same virtual directory isn't recommended as a best practice, however. You should create `badfile`, `discard`, and `log` virtual directories to hold their respective output files, and then grant only read privilege to the `upload` directory. You should grant read and write privileges to the `badfile`, `discard`, and `log` virtual directories, especially if you're writing modules to let your business users confirm the upload of data.

Step 2: Position Your Physical CSV File

After you create the `upload` virtual directory in the `import` database, you need to create the physical directory in the file system of the server's operating system. Creating the physical directory after the virtual directory should help show you that a virtual directory's definition is independent of its physical directory. Virtual directories only store data that maps the virtual directory name to a physical directory, and they do not validate the existence of that location until you try to use it.

This example uses a `kingdom_import.csv` file that holds data for the `kingdom` and `knight` tables. The data in the file describe two epochs of the mythical kingdom of Narnia:

```
'Narnia',77600,'Peter the Magnificent','20-MAR-1272','19-JUN-1292',
'Narnia',77600,'Edmund the Just','20-MAR-1272','19-JUN-1292',
'Narnia',77600,'Susan the Gentle','20-MAR-1272','19-JUN-1292',
'Narnia',77600,'Lucy the Valiant','20-MAR-1272','19-JUN-1292',
'Narnia',42100,'Peter the Magnificent','12-APR-1531','31-MAY-1531',
'Narnia',42100,'Edmund the Just','12-APR-1531','31-MAY-1531',
'Narnia',42100,'Susan the Gentle','12-APR-1531','31-MAY-1531',
'Narnia',42100,'Lucy the Valiant','12-APR-1531','31-MAY-1531',
```

Notice that there are no surrogate key values in the data set. This means that the `MERGE` statement needs to provide them.

Step 3: Create Example Tables

You should now connect to the `student` schema in the database. As the `student` user, create two internally defined tables, the `kingdom` and `knight` tables; then create one externally defined table, the `kingdom_knight_import` table. The `kingdom` table is the parent and the `knight` table is the child in this relationship. This means a column in the `knight` table holds a foreign key that references (points back to) a column in the `kingdom` table. The `kingdom_id` column is the primary key in the `kingdom` table, and the `kingdom_allegiance_id` column holds the copy of the primary key value as a foreign key in the `knight` table.

Here's the CREATE TABLE statement for the kingdom table:

```
SQL> CREATE TABLE kingdom
  ( kingdom_id    NUMBER
  , kingdom_name  VARCHAR2(20)
  , population    NUMBER);
```

And here is its sequence:

```
CREATE SEQUENCE kingdom_s1;
```

Here's the CREATE TABLE statement for the knight table:

```
SQL> CREATE TABLE knight
  ( knight_id              NUMBER
  , knight_name            VARCHAR2(24)
  , kingdom_allegiance_id NUMBER
  , allegiance_start_date DATE
  , allegiance_end_date   DATE);
```

Here is its sequence:

```
CREATE SEQUENCE knight_s1;
```

Here is the CREATE TABLE statement for the kingdom_knight_import table:

```
SQL> CREATE TABLE kingdom_knight_import
  ( kingdom_name          VARCHAR2(8)
  , population            NUMBER
  , knight_name           VARCHAR2(24)
  , allegiance_start_date DATE
  , allegiance_end_date   DATE)
    ORGANIZATION EXTERNAL
    ( TYPE oracle_loader
      DEFAULT DIRECTORY upload
      ACCESS PARAMETERS
      ( RECORDS DELIMITED BY NEWLINE CHARACTERSET US7ASCII
        BADFILE     'UPLOAD':'kingdom_import.bad'
        DISCARDFILE 'UPLOAD':'kingdom_import.dis'
        LOGFILE     'UPLOAD':'kingdom_import.log'
        FIELDS TERMINATED BY ','
        OPTIONALLY ENCLOSED BY "'"
        MISSING FIELD VALUES ARE NULL )
      LOCATION ('kingdom_import.csv'))
  REJECT LIMIT UNLIMITED;
```

There is no sequence for an external table. Recall that there is also no surrogate key in the data set or the definition of this externally managed table. (You can find more about externally managed tables in Chapter 6.)

Step 4: Test Configuration

You should be able to query from the externally managed table after the first three steps have completed successfully. This query should return eight rows:

```
SQL> SELECT   kingdom_name AS kingdom
  2  ,        population
  3  ,        knight_name
  4  ,        TO_CHAR(allegiance_start_date,'DD-MON-YYYY') AS start_date
  5  ,        TO_CHAR(allegiance_end_date,'DD-MON-YYYY') AS end_date
  6  FROM     kingdom_knight_import;
```

You should get the following data set if it works:

```
KINGDOM  POPULATION KNIGHT_NAME            START_DATE  END_DATE
-------- ---------- ---------------------- ----------- -----------
Narnia        77600 Peter the Magnificent  20-MAR-1272 19-JUN-1292
Narnia        77600 Edmund the Just        20-MAR-1272 19-JUN-1292
Narnia        77600 Susan the Gentle       20-MAR-1272 19-JUN-1292
Narnia        77600 Lucy the Valiant       20-MAR-1272 19-JUN-1292
Narnia        42100 Peter the Magnificent  12-APR-1531 31-MAY-1531
Narnia        42100 Edmund the Just        12-APR-1531 31-MAY-1531
Narnia        42100 Susan the Gentle       12-APR-1531 31-MAY-1531
Narnia        42100 Lucy the Valiant       12-APR-1531 31-MAY-1531
```

An error like the following occurs when you failed in setting up the physical directory or file, the virtual directory, or the grant of permissions:

```
SELECT   kingdom_name
*
ERROR at line 1:
ORA-29913: error in executing ODCIEXTTABLEOPEN callout
ORA-29400: data cartridge error
KUP-04040: file kingdom_import.csv in UPLOAD not found
```

You need to fix whatever piece is broken before continuing.

CAUTION
Oracle assumes that any physical directory is on the local system disks. It is possible that you could run into the error shown in the example if the physical directory is a virtual directory itself.

Step 5: Merge the Import Source

Merges work with one table at a time, and they must start with the least dependent table. The `kingdom` table is the one without dependencies and should be the first table where you merge data. In a real situation, the `MERGE` statements would be bundled into a stored procedure and wrapped in Transaction Control Language (TCL) commands to make sure both statements worked or failed. TCL would require a `SAVEPOINT` before the first `MERGE` statement and a `COMMIT` after the last `MERGE` statement, and the following should appear in an exception handler in the event one `MERGE` statement failed:

```
ROLLBACK TO savepoint_name;
```

The following MERGE statement reads data from the externally managed kingdom_knight_import table and performs a LEFT JOIN operation between a copy of the target and source tables:

```
SQL>   MERGE INTO kingdom target
  2    USING
  3     (SELECT DISTINCT
  4             k.kingdom_id
  5      ,      kki.kingdom_name
  6      ,      kki.population
  7      FROM   kingdom_knight_import kki LEFT JOIN kingdom k
  8      ON     kki.kingdom_name = k.kingdom_name
  9      AND    kki.population = k.population) SOURCE
 10    ON (target.kingdom_id = SOURCE.kingdom_id)
 11    WHEN MATCHED THEN
 12    UPDATE SET kingdom_name = SOURCE.kingdom_name
 13    WHEN NOT MATCHED THEN
 14    INSERT VALUES
 15    ( kingdom_s1.NEXTVAL
 16    , SOURCE.kingdom_name
 17    , SOURCE.population);
```

The only time line 4 returns a kingdom_id value from the kingdom table is when the row already exists. The row can exist only when the natural key values match, which means the surrogate keys on line 10 also match. The UPDATE statement on line 12 doesn't change anything, because when line 10 matches, the kingdom_name column values also match on line 8. The INSERT statement on lines 14–17 runs when there is no match between the natural keys. Notice that the INSERT statement excludes a target table name, because it works with the target table of the MERGE statement. The source values in the INSERT statement come from the relative complement of the kingdom table (non-matching natural keys or new data). These are new rows from the import source file. Check Chapters 6 and 11 if you have any questions on how outer joins work.

The merge should report the number of rows merged, which might exceed the number of new rows inserted into the table. You confirm the number of rows inserted with the following query:

```
SQL> SELECT * FROM kingdom;
```

It returns the following:

```
KINGDOM_ID KINGDOM_NAME         POPULATION
---------- -------------------- ----------
         1 Narnia                    42100
         2 Narnia                    77600
```

Note that the ON subclause should use the natural key, and the nested UPDATE statement must SET a column not found in the ON clause. Changing line 12 from kingdom_name to kingdom_id raises this error message:

```
  ON (target.kingdom_id = SOURCE.kingdom_id)
      *
```

```
ERROR at line 9:
ORA-38104: Columns referenced in the ON Clause cannot be updated:
"TARGET"."KINGDOM_ID"
```

The second `MERGE` statement can work only when there are matching rows in the kingdom table for new rows in the import source file. That's why it performs an `INNER JOIN` operation between the `kingdom` and `kingdom_knight_import` tables before it performs an outer join against the `knight` table.

Here is the second `MERGE` statement:

```
SQL> MERGE INTO knight target
  2  USING
  3   (SELECT kn.knight_id
  4    ,      k.kingdom_id
  5    ,      kki.knight_name
  6    ,      kki.allegiance_start_date
  7    ,      kki.allegiance_end_date
  8    FROM   kingdom_knight_import kki INNER JOIN kingdom k
  9    ON     kki.kingdom_name = k.kingdom_name
 10    AND    kki.population = k.population LEFT JOIN knight kn
 11    ON     k.kingdom_id = kn.kingdom_allegiance_id
 12    AND    kki.knight_name = kn.knight_name
 13    AND    kki.allegiance_start_date = kn.allegiance_start_date
 14    AND    kki.allegiance_end_date = kn.allegiance_end_date) source
 15  ON (target.knight_id = source.knight_id)
 16  WHEN MATCHED THEN
 17  UPDATE SET target.knight_name = source.knight_name
 18  WHEN NOT MATCHED THEN
 19  INSERT
 20  ( knight_id
 21  , knight_name
 22  , kingdom_allegiance_id
 23  , allegiance_start_date
 24  , allegiance_end_date)
 25  VALUES
 26  ( knight_s1.NEXTVAL
 27  , source.knight_name
 28  , source.kingdom_id
 29  , source.allegiance_start_date
 30  , source.allegiance_end_date);
```

Although it works like the last `MERGE` statement, the query that provides the source uses an `INNER JOIN` operator to confirm that a matching `kingdom` exists. It checks whether a new `knight` exists in the import source only when a valid `kingdom` for that `knight` exists. The matching criterion on line 15 is the surrogate key value of the `knight` table. This is the same rule as for the prior `MERGE` statement.

Lines 20–24 show an overriding signature for the `INSERT` statement. Other than the absence of a target table name, the `INSERT` statement works as it does on its own. (For more on overriding signatures for INSERT statements, see Chapter 8.) A query for this data set requires a couple of SQL*Plus formatting commands to make it fit nicely here in the book, like so:

```
COLUMN knight_id   FORMAT 999 HEADING "Knight|ID #"
COLUMN knight_name FORMAT A22 HEADING "Knight Name"
COLUMN kingdom_allegiance_id FORMAT 999 HEADING "Allegiance|ID #"
COLUMN allegiance_start_date FORMAT A9 HEADING "Start|Date"
COLUMN allegiance_end_date FORMAT A9 HEADING "End|Date"
```

The query would be as follows:

```
SQL> SELECT * FROM knight;
```

and the knight table should yield the following rows:

```
Knight                        Allegiance Start     End
  ID # Knight Name                  ID # Date      Date
------ ---------------------- ---------- --------- ---------
     1 Peter the Magnificent           2 20-MAR-72 19-JUN-92
     2 Edmund the Just                 2 20-MAR-72 19-JUN-92
     3 Susan the Gentle                2 20-MAR-72 19-JUN-92
     4 Lucy the Valiant                2 20-MAR-72 19-JUN-92
     5 Peter the Magnificent           1 12-APR-31 31-MAY-31
     6 Edmund the Just                 1 12-APR-31 31-MAY-31
     7 Susan the Gentle                1 12-APR-31 31-MAY-31
     8 Lucy the Valiant                1 12-APR-31 31-MAY-31
```

At the end of this step, you see the results from the kingdom and knight tables. There should be two rows in the kingdom table for two epochs of Narnia and eight rows in the knight table for the two visits by the four Pevensie children, who become kings and queens in this mythical land (at least in the first two books).

A verification of the ability to merge data can be achieved by adding a single row to the kingdom_import.csv file, which would give it this extra line for Caspian X:

```
'Narnia',40100,'Caspian X','31-MAY-1531','30-SEP-1601',
```

Rerunning the MERGE statements, you would see one row added to previous two rows in the kingdom table, and one row added to the previous eight rows in the knight table. The UPDATE clause of the statement assigns the existing knight_id surrogate key value back to the same column, which results in no net change.

Merging Data in MySQL

You merge data in MySQL with an INSERT statement that uses the ON DUPLICATE KEY clause or a REPLACE INTO statement. The MySQL INSERT statement with the ON DUPLICATE KEY clause is the closest to the MERGE statement in Oracle because it can validate on a natural key. While the REPLACE INTO statement also accomplishes a merge, it doesn't use a natural key comparison. The REPLACE INTO statement performs an elimination of duplicate rows based on a unique key.

Unlike Oracle, MySQL doesn't support externally managed tables. In lieu of external tables, MySQL lets you import sources through the LOAD DATA statement or export data with a SELECT statement and an INTO OUTFILE clause. Both import and export data processes are covered in the first of the following three subsections. The subsequent sections cover examples of using both merge approaches: the INSERT statement with the ON DUPLICATE KEY clause and the REPLACE INTO statement.

Import and Export Data Processes

MySQL lets you input data from plain text files with the LOAD DATA statement and export data to plain text files with a SELECT statement with the INTO OUTFILE clause. The SELECT statement also supports the output of XML tagged data, and when combined with the INTO OUTFILE clause, the SELECT statement can export table data into XML files. Import sources can now also be XML files because of the LOAD XML statement added in MySQL 5.5.

Importing Data in MySQL

Some of the LOAD DATA statement options can be confusing until you understand the rules governing how you should use this statement. These options work differently based on whether you create a constrained or an unconstrained import table. The import table takes the place of the externally managed table in the Oracle technology example—you grab the data from the import table to move it into the normalized tables.

An unconstrained table is the default for externally managed tables in Oracle. Although that approach might seem fine in MySQL because it works with Oracle's external tables and MERGE statement, it's not workable in MySQL—at least, it's not workable unless you always clean up the previously imported data by truncating the table or deleting all rows from the last merge. That's because the LOAD DATA statement adds everything in the import source to the target table unless you've specified a unique or primary key for the target table.

When you've specified a unique or primary key, the LOAD DATA statement replaces the row where the key columns match with a new row or ignores the new row in the import source. Ignoring the row is safe when non-key columns won't change between imports, but it is unsafe when non-key columns might change. Fortunately, the default is the REPLACE option in the LOAD DATA statement, which means changes in import source rows replace existing rows when the key matches.

Here is the prototype for the LOAD DATA statement:

```
LOAD DATA [{LOW_PRIORITY | CONCURRENT}] [LOCAL] INFILE 'file_name'
[{REPLACE | IGNORE}] INTO TABLE [database_name.]table_name
[PARTITION (partition_name [,...])]
[CHARACTER SET characterset_name]
[{FIELDS | COLUMNS}
 [TERMINATED BY 'character']
 [[OPTIONALLY] ENCLOSED BY 'character']
 [ESCAPED BY 'character']]
[LINES
 [STARTING BY 'character']
 [TERMINATED BY 'character']]
[IGNORE number LINES]
[({table_column_name | session_variable} [, {...}])]
[SET table_column_name = expression [,...]]
```

Only super users, or those users to whom a super user grants a global privilege such as FILE, can run the LOAD DATA statement. The minimum privileges required for a user to run the LOAD DATA statement are the global FILE privilege and the INSERT privilege on the target table.

You would grant the global FILE privilege with the following syntax:

```
GRANT FILE ON *.* TO user_name;
```

Users without the global FILE privilege can include the LOCAL keyword to run a file from the computer where they're running MySQL Monitor. MySQL Monitor is the client software for the database and can be run on a remote client or on the server where the database is running. Sometimes LOCAL behavior is disallowed by the DBA because of its security implications, and when it is disallowed you get the following error:

```
ERROR 1148 (42000): The used command is not allowed with this MySQL version
```

This error means that you'll find the following setting in your my.ini (Windows) or my.cnf (Linux) file in the [mysqld] section:

```
local-infile=0
```

You can reset the local-infile parameter to 1 or remove it from the configuration file. Then, you'll need to stop and start the MySQL server for the new setting to take effect. The LOCAL parameter doesn't present security restrictions, but does allow you to import data from a client machine where you're running MySQL Monitor. This provides tremendous control to developers over flat files that might contain company-sensitive information. As a rule, you don't want to craft a process in which you load bulk data from remote machines, because this is insecure and can place an inordinate load on your network bandwidth. The best practice is to run imports through specialized users that can access MySQL only from the (localhost) server.

Like many DML statements in MySQL, the LOAD DATA statement lets you specify *low priority* or *concurrent* threaded operations. A low priority operation means that the INSERT statement waits until no queries are pending against the table; concurrent operations work with MyISAM files by dividing the insert into threads, which can perform concurrent insert operations. Concurrency works provided there are no free disk blocks between the two concurrent insertion loci (points of inserts). You can also override the default character set by providing one of your own choosing.

The REPLACE and IGNORE options provide instructions to the LOAD DATA statement that govern behavior when the import target table has a unique or primary key. The default raises an exception when a duplicate key is encountered, stops processing, and raises the following error:

```
ERROR 1062 (23000): Duplicate entry 'duplicate_row' for key 'key_name'
```

The REPLACE option replaces the row when the key values match a row in the target table, and it inserts a new row when they don't match. The IGNORE disables the replacing behavior and simply ignores import source rows when the key values match a row in the target table.

As mentioned, you should ensure that all import target tables have a unique or primary key. Without such a key, the LOAD DATA command simply adds values to the target table without avoiding duplication. You should place that unique or primary key on the natural key columns of the table, because flat files don't typically store surrogate key values.

The PARTITION clause is new in MySQL 5.6.2. It lets you specify a list of partitions for the data load. If a failure in loading one partition occurs, it causes the entire statement to fail.

FIELDS represent literal values or variables, and COLUMNS represent columns in the SELECT list of a query. Note another syntax nuance when you use a query inside an INSERT statement with the ON DUPLICATE KEY clause: You can't assign an alias to the result set as you would with an inline view, and you must refer to the column name using a fully qualified reference to

SELECT list columns. A fully qualified reference includes a table name or alias from the query, a dot, and the column name.

The FIELDS TERMINATED BY clause refers to the character that separates field or column values. Typically, this is a comma (,) in CSV files, but sometimes it's a tab (represented by a backslash and lowercase letter t [\t]) when the source file is a tab-separated value (TSV) file. The tab is the default value for the TERMINATED BY clause when you don't provide an override value. Table 12-1 shows a list of special metacharacters and their escape sequence values.

The ENCLOSED BY clause is optional, and using the OPTIONAL keyword before the ENCLOSED BY phrase is rarely done. The default for ENCLOSED BY is an empty string, and excluding the clause is equivalent to entering this:

```
ENCLOSED BY ''
```

Double quotes (") is the most common overriding value for the ENCLOSED BY clause and would be entered like so:

```
ENCLOSED BY '"'
```

The backslash (\) character is the default value for the ESCAPED BY clause. You would assign a backslash like this:

```
ESCAPED BY '\\'
```

By default, the STARTING BY clause is an empty string, and the LINES TERMINATED BY clause is a newline (metacharacter \n) character. You should use these defaults in a Linux system. In Windows, using a combination of a newline and carriage return (metacharacters \n\r) is required, because that's how the platform's software writes line returns. The exception to that rule is the Microsoft WordPad application that writes only a newline character.

NOTE
The default LINES TERMINATED BY value is \n because it is designed to run on a Linux system.

Metacharacter	Escape Sequence
\0	An ASCII NUL (0x00) character
\b	A backspace character
\n	A newline (linefeed) character
\r	A carriage return character
\t	A tab character
\Z	An ASCII 26 (CTRL-Z) file close character
\N	A NULL

TABLE 12-1. *Escape Characters*

The second `IGNORE` option in the `LOAD DATA` statement has to do with skipping header lines at the top of an import source file. You would use this to skip two header lines in the import source file:

```
IGNORE 2 LINES
```

TIP
Use `\n\r` to terminate lines on the Windows platform because that's how Microsoft's operating system writes line returns.

The column list at the end of the statement allows you to override the definition of the table and import only mandatory columns. You can also override an import source field with a local session variable. The last choice is the `SET` option, which lets you assign a value to a column from an expression. A key restriction on the `SET` option is that you can't use values from the import source file.

It seems best to provide a quick example here (others are provided in the "Merging with the INSERT Statement" and "Merging with the REPLACE INTO Statement" sections). This example uses the `IGNORE n LINES` option for a header in the source file, an overriding column list, and the `SET` option.

The first step is to define the `employee` table, like this:

```
CREATE TABLE employee
( name          VARCHAR(20) PRIMARY KEY
, salary        DECIMAL(8,2)
, bonus         DECIMAL(8,2)
, upload_date   DATETIME);
```

Stage a file on the server with a two-row header. For this example, the file is placed in the `C:\Data\Upload` directory:

```
Employee Name,Salary
-------------,--------
"Harry James",52121.82,
"Sally Fields",54234.22,
```

To use the `SET` option with a calculation, you need to define a session variable:

```
SET @rate = .05;
```

The last step is to import the source file into the table with this statement:

```
LOAD DATA INFILE 'c:/Data/mysql/employee.csv'
REPLACE INTO TABLE employee
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
ESCAPED BY '\\'
LINES TERMINATED BY '\r\n'
IGNORE 2 LINES
( name, salary )
SET bonus = 1000 * @rate
,   upload_date = NOW();
```

A query against the employee table shows two rows:

```
+--------------+----------+-------+---------------------+
| name         | salary   | bonus | upload_date         |
+--------------+----------+-------+---------------------+
| Harry James  | 52121.82 | 50.00 | 2011-06-26 22:01:42 |
| Sally Fields | 54234.22 | 50.00 | 2011-06-26 22:01:42 |
+--------------+----------+-------+---------------------+
```

As you can tell, the import process offers you some nice options for text-based files. The `LOAD XML` statement extends those behaviors for XML files.

Here is the `LOAD XML` statement prototype, which is available starting in MySQL 5.5:

```
LOAD XML [{LOW_PRIORITY | CONCURRENT}] [LOCAL] INFILE 'file_name'
[{REPLACE | IGNORE}] INTO TABLE [database_name.]table_name
[PARTITION (partition_name [,...])]
[CHARACTER SET characterset_name]
[ROWS IDENTIFIED BY '<xml_tag_name>']
[IGNORE number [LINES | ROWS]]
[({table_column_name | session_variable} [, {...}])]
[SET table_column_name = expression [,...]]
```

The clauses of the `LOAD XML` statement are a subset of those found in the `LOAD DATA` statement. The single exception is the `ROWS IDENTIFIED BY` clause, which maps the root node of an XML structure. If no value is specified in this clause, `ROW` is the assumed XML tag name for root nodes.

Leveraging the `employee` table from the prior example, the following example shows you how to load an XML file. The import source file must be changed from a CSV to an XML file, like this:

```
<?xml version="1.0"?>
<list>
  <name>Harry James</name>
  <salary>52121.82</salary>
</list>
<list>
  <name>Sally Fields</name>
  <salary>54234.22</salary>
</list>
```

This example uses an overriding column list and the `SET` option, which means you need to set the session variable:

```
SET @rate = .05;
```

Here's the `LOAD XML` statement that works with these:

```
LOAD XML INFILE 'c:/Data/mysql/employee.xml'
INTO TABLE employee
ROWS IDENTIFIED BY '<list>'
( name, salary )
```

```
SET bonus = 1000 * @rate
,   upload_date = NOW();
```

A query of the table yields these rows:

```
+--------------+----------+-------+---------------------+
| name         | salary   | bonus | upload_date         |
+--------------+----------+-------+---------------------+
| Harry James  | 52121.82 | 50.00 | 2011-06-26 22:50:50 |
| Sally Fields | 54234.22 | 50.00 | 2011-06-26 22:50:50 |
+--------------+----------+-------+---------------------+
```

You've now seen how to load flat files (CSVs) and XML files. The next section shows you how to extract data from a MySQL database into an external file.

Exporting Data from MySQL

Unlike Oracle's spooling feature, MySQL lets you configure and format files in a specialized `SELECT` statement. It takes the column values from the `SELECT` list and puts them into the external file. The statement uses the same formatting clauses as the `LOAD DATA` statement, but instead of striping delimiters it puts them into the data set.

Here's the prototype for the `SELECT INTO OUTFILE` statement:

```
SELECT [column_name1 [,column_name2 [,...]]] INTO OUTFILE file_name
[CHARACTER SET characterset_name]
[TERMINATED BY 'character']
[[OPTIONALLY] ENCLOSED BY 'character']
[ESCAPED BY 'character']
[TERMINATED BY 'character']
FROM file_name
WHERE [condition1 {AND | OR} condition2 [{AND | OR} ...]];
```

Applying the syntax to the employee table that was populated by the `LOAD DATA` statement earlier, you can query the data and put it in an `employee.txt` file. The following statement returns all columns from the `employee` table (provided the user has the global `FILE` privilege and operating system write privileges to the directory):

```
SELECT * INTO OUTFILE 'c:/Data/mysql/employee.txt'
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
ESCAPED BY '\\'
LINES TERMINATED BY '\r\n'
FROM employee;
```

And here is the contents of the `employee.txt` file that was generated by the `SELECT INTO OUTFILE` statement:

```
"Harry James","52121.82","50.00","2011-06-26 23:18:02"
"Sally Fields","54234.22","50.00","2011-06-26 23:18:02"
```

Merging with the INSERT statement

The INSERT INTO statement in MySQL works very much like the MERGE statement in Oracle. You can override the default signature of the table by providing a list of at least all mandatory columns. This list is the override signature for the INSERT statement, as discussed in Chapter 8.

The prototype for the INSERT INTO statement with an ON DUPLICATE KEY clause is:

```
INSERT INTO table_name
[(column_name_list)]
{VALUES (value_list) [,(value_list) [,(...)]] | (select_statement)}
ON DUPLICATE KEY UPDATE update_list;
```

After the optional column list, you must choose whether you use a VALUES clause with a list of values or a subquery. Recall that you don't include the VALUES clause when you provide a subquery to an INSERT statement. The INSERT statement supports multiple value lists, as it does without the ON DUPLICATE KEY clause. The UPDATE should include those things that might change, such as non-key column values.

The following subsections show you how to merge values in the kingdom and knight tables using the INSERT INTO statement with the ON DUPLICATE KEY in a MySQL database. It takes one fewer step in MySQL than in the Oracle process, because MySQL doesn't support virtual directories. MySQL also doesn't support externally managed tables, and you'll see how to use the LOAD DATA statement to grab an external CSV file.

Step 1: Position Your Physical CSV File

As with the preceding examples, you should put the kingdom_mysql_import.csv file in the C:\Data\MySQL directory. You'll need to modify the coding examples if you use a different directory. The file differs from the Oracle example file in only one way: the dates use the MySQL default date format mask. This makes the upload work smoothly.

Here's the content of the file:

```
Narnia, 77600,'Peter the Magnificent',12720320,12920609
Narnia, 77600,'Edmund the Just',12720320,12920609
Narnia, 77600,'Susan the Gentle',12720320,12920609
Narnia, 77600,'Lucy the Valiant',12720320,12920609
Narnia, 42100,'Peter the Magnificent',15310412,15310531
Narnia, 42100,'Edmund the Just',15310412,15310531
Narnia, 42100,'Susan the Gentle',15310412,15310531
Narnia, 42100,'Lucy the Valiant',15310412,15310531
```

Step 2: Create Example Tables

Connect to the MySQL server and choose a database in which you will work. Then create the three tables for this example. The first table is a temporary table using the Memory database engine, because it should exist only during the import session. Here's the CREATE TEMPORARY TABLE statement:

```
CREATE TEMPORARY TABLE kingdom_knight_import
( kingdom_name          VARCHAR(20)
, population            INT UNSIGNED
, knight_name           VARCHAR(24)
, allegiance_start_date DATE
, allegiance_end_date   DATE
, CONSTRAINT import
  UNIQUE ( kingdom_name
         , population
         , knight_name
         , allegiance_start_date
         , allegiance_end_date )) ENGINE=MEMORY;
```

Notice that the table is created with a unique key, which ensures that duplicate records won't be loaded twice. It's clearly not a perfect natural key, but it works to support the data set for the problem. The unique key qualifies the natural key of the two subjects in the import table: the `kingdom` and `knight` subjects. The `kingdom_name` and population columns are the natural key of the `kingdom` table, while the other three columns in the unique key qualify the natural key of the `knight` table.

The target tables are normal tables, assuming this is a real model. You would use `CREATE TABLE` statements to build them (shown a bit later). They both use surrogate keys that are populated by auto-incrementing sequence values, and they specify being built using the InnoDB engine (a precaution for anybody testing in an MyISAM database).

As stated, MySQL doesn't support external tables. This means you need to load the import source into the import table. The following `LOAD DATA` statement loads the source file into the `kingdom_knight_import` table, which is a mirror to the external file's structure.

```
LOAD DATA INFILE 'c:/Data/mysql/kingdom_mysql_import.csv'
REPLACE INTO TABLE kingdom_knight_import
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
ESCAPED BY '\\'
LINES TERMINATED BY '\r\n';
```

The `REPLACE` clause ensures that duplicate rows are never inserted into the import table. Any duplicate row would raise an exception when the `REPLACE` or `IGNORE` option is excluded from the `LOAD DATA` statement.

After staging the import data into a database table, you need to create the two normalized tables where you're going to put the final data sets. This `CREATE TABLE` statement creates the `kingdom` table, which is the parent table in the relationship between the two tables. Here's the syntax:

```
CREATE TABLE kingdom
( kingdom_id    INT UNSIGNED PRIMARY KEY AUTO_INCREMENT
, kingdom_name  VARCHAR(20)
, population    INT UNSIGNED
, CONSTRAINT unique_kingdom
  UNIQUE ( kingdom_name, population )) ENGINE=INNODB;
```

The next statement creates the `knight` table. As the child table in the relationship, it also includes a foreign key constraint to the `kingdom` table. The foreign key maps the link between the `kingdom_id` and `kingdom_allegiance_id` columns. Here's the syntax:

```
CREATE TABLE knight
( knight_id             INT UNSIGNED PRIMARY KEY AUTO_INCREMENT
, knight_name           VARCHAR(24)
, kingdom_allegiance_id INT UNSIGNED
, allegiance_start_date DATE
, allegiance_end_date   DATE
, CONSTRAINT import
  UNIQUE ( knight_name
         , population
         , kingdom_allegiance_id
         , allegiance_start_date
         , allegiance_end_date )
, CONSTRAINT fk_kingdom FOREIGN KEY (kingdom_allegiance_id)
  REFERENCES kingdom (kingdom_id)) ENGINE=INNODB;
```

You don't need sequences, because they're properties of tables where one column holds the `AUTO_INCREMENT` keyword. The unique keys in these tables prevent the `INSERT INTO` statement with an `ON DUPLICATE KEY` clause from adding the same row more than once.

Step 3: Test Configuration

At this point, you can query the results from the `kingdom_knight_import` table. It should contain the eight rows from the external file. This query formats the output into a single page:

```
mysql> SELECT   kingdom_name AS "Kingdom"
    -> ,        population AS "Citizens"
    -> ,        knight_name AS "Knight's Name"
    -> ,        allegiance_start_date AS "Start Date"
    -> ,        allegiance_end_date AS "End Date"
    -> FROM     kingdom_knight_import;
```

It should return the following:

```
+---------+----------+-------------------------+------------+------------+
| Kingdom | Citizens | Knight's Name           | Start Date | End Date   |
+---------+----------+-------------------------+------------+------------+
| Narnia  |    77600 | 'Peter the Magnificent' | 1272-03-20 | 1292-06-09 |
| Narnia  |    77600 | 'Edmund the Just'       | 1272-03-20 | 1292-06-09 |
| Narnia  |    77600 | 'Susan the Gentle'      | 1272-03-20 | 1292-06-09 |
| Narnia  |    77600 | 'Lucy the Valiant'      | 1272-03-20 | 1292-06-09 |
| Narnia  |    42100 | 'Peter the Magnificent' | 1531-04-12 | 1531-05-31 |
| Narnia  |    42100 | 'Edmund the Just'       | 1531-04-12 | 1531-05-31 |
| Narnia  |    42100 | 'Susan the Gentle'      | 1531-04-12 | 1531-05-31 |
| Narnia  |    42100 | 'Lucy the Valiant'      | 1531-04-12 | 1531-05-31 |
+---------+----------+-------------------------+------------+------------+
```

If you didn't get the right results, revisit the steps and fix what's missing. Make sure that you didn't try to reuse the Oracle import source, because the default date formats differ between the two files.

Step 4: Merge from the Import Source

The import source in MySQL is an internally managed table that mirrors the import source file's structure. You loaded the data from the external file into the table with the `LOAD DATA` statement in Step 2.

Before examining how a query can load all the relevant data from the kingdom_knight_import table, you will see how a VALUES clause works in this INSERT statement. This example uses string and numeric literal values inside the VALUES clause and loads one of the two Narnia rows into the kingdom table:

```
INSERT INTO kingdom
(kingdom_name, population)
VALUES
('Narnia',77600)
ON DUPLICATE KEY
UPDATE kingdom_id = kingdom_id;
```

The statement uses an overriding signature and inserts only the second and third columns of the table. The INSERT statement updates the kingdom_id column with the current row's column value when a duplicate row is found, and it inserts a row when no other row is found that shares the unique key values.

You could query the table after the INSERT statement, and you should see this:

```
+------------+--------------+------------+
| kingdom_id | kingdom_name | population |
+------------+--------------+------------+
|          1 | Narnia       |      77600 |
+------------+--------------+------------+
```

The next INSERT statement works with two rows of literal values in the VALUES clause. The statement inserts only one new row, because the first row is already inserted by the previous statement.

The syntax for a list of rows in the VALUES clause is shown here:

```
INSERT INTO kingdom
(kingdom_name, population)
VALUES
('Narnia',77600),('Narnia',42100)
ON DUPLICATE KEY
UPDATE kingdom_id = kingdom_id;
```

The results now contain these two rows:

```
+------------+--------------+------------+
| kingdom_id | kingdom_name | population |
+------------+--------------+------------+
|          1 | Narnia       |      77600 |
|          2 | Narnia       |      42100 |
+------------+--------------+------------+
```

The ON DUPLICATE KEY clause combined with unique key (index) on the kingdom table prevented a reload of the first row of data. Both rows could have been loaded with a query from the internal import source table, like this:

```
INSERT INTO kingdom
( kingdom_id, kingdom_name, population )
( SELECT   DISTINCT
           k.kingdom_id
  ,        kki.kingdom_name
  ,        kki.population
  FROM     kingdom_knight_import kki LEFT JOIN kingdom k
  ON       kki.kingdom_name = k.kingdom_name
  AND      kki.population = k.population )
  ON DUPLICATE KEY
  UPDATE kingdom_id = k.kingdom_id;
```

Although not necessary, because the query provides values for all columns defined in the `kingdom` table, the column list qualifies the column names of the target table of the `INSERT` statement. The `LEFT JOIN` operator returns the right relative complement or every row from the table on the left of the operator and matching rows from the table on the right or null values. That means you get all rows from `kingdom_knight_import` and all rows from the `kingdom` table, or null values. In this scenario, the null values map to the `kingdom_id` column for those rows returned as the right relative complement (or new rows not found in the `kingdom` table). Since the `kingdom_id` column is an auto-incrementing column, MySQL automatically assigns the next sequence value to the table when it receives a null value in an auto-incrementing column.

If you had deleted all rows from the table before running the preceding `INSERT` statement, it would insert two rows: one row for each unique combination of the `kingdom_name` and `population` columns not found in the `kingdom` table. It wouldn't insert any new rows if both columns matched the result sets previously returned by the query.

The `k.kingdom_id` provided in the `UPDATE` clause is required, because the subquery can't have an alias, and the column definition must match exactly with a column returned by the `SELECT` list of the query. The match requires that you use any alias provided inside the subquery and dot notation that precedes the column name (like `kn.knight_id`).

After you've inserted values in the `kingdom` table, you can insert values in the `knight` table. Any row returned from the query that fails to match the `kingdom_id` primary key value with the `kingdom_allegiance_id` foreign key values is excluded from the result set. This behavior enforces referential integrity between the two tables.

The query in the following statement secures the (surrogate) foreign key from the parent table through the aforementioned join and returns null values for the `knight_id` column when inserting new knights:

```
INSERT INTO knight
( SELECT   kn.knight_id
  ,        kki.knight_name
  ,        k.kingdom_id
  ,        kki.allegiance_start_date AS start_date
  ,        kki.allegiance_end_date AS end_date
  FROM     kingdom_knight_import kki INNER JOIN kingdom k
  ON       kki.kingdom_name = k.kingdom_name
  AND      kki.population = k.population LEFT JOIN knight kn
  ON       k.kingdom_id = kn.kingdom_allegiance_id
  AND      kki.knight_name = kn.knight_name
  AND      kki.allegiance_start_date = kn.allegiance_start_date
  AND      kki.allegiance_end_date = kn.allegiance_end_date )
  ON DUPLICATE KEY
  UPDATE knight_id = kn.knight_id;
```

You should see that the column list isn't necessary and isn't provided. That means the list of column values returned by the query must match against the data catalog definition of the `knight` table, and it does. As mentioned in the preceding `INSERT` statement, the `kn.knight_id` value in the `UPDATE` clause must match exactly with a column returned by the `SELECT` list of the subquery.

Merging with the REPLACE INTO Statement

The `REPLACE INTO` statement in MySQL works like the `MERGE` statement in Oracle with a key difference: there's no `ON` subclause. The `REPLACE INTO` statement sorts the target data set, determines whether a matching row exists, and inserts source rows when they're new. More or less, it performs similar to a `UNION ALL` operation on the two sets followed by a `MINUS` operation. The `MINUS` operation subtracts the existing rows in the target table from the new source result set. See Chapter 11 for more on how set operators work.

Here's the prototype of the `REPLACE INTO` statement:

```
REPLACE INTO table_name
( select_statement );
```

Like the Oracle `MERGE` statement, you need to know the columns that define the natural key of the target table for the `REPLACE INTO` statement. The natural key is the basis of the join between the target and source tables in the nested `SELECT` statement. Also, the final join to the target table in the nested query must be an outer join, because that's the only way to find the new rows and return a sequence generated surrogate key value. The new rows are the relative complement of the existing rows.

Step 1: Position Your Physical CSV File

Because you should have provisioned the `kingdom_mysql_import.csv` file in the `C:\Data\MySQL` directory earlier, there's no need to repeat the instructions here.

Step 2: Create Example Tables

These tables should already exist in the database (if you followed the steps earlier in the chapter). You should also have run the `LOAD DATA` statement to move the import source file contents into an internally managed table. If you skipped these steps, do them now.

You should delete the rows from the `knight` and `kingdom` tables, like this:

```
TRUNCATE TABLE knight;
TRUNCATE TABLE kingdom;
```

Removing the data from the tables should avoid confusion as to whether something worked using this or the prior section's syntax. Next, you'll test your configuration.

Step 3: Test Configuration

Rather than flip back at this point, you can query the results from the `kingdom_knight_import` table and determine whether they're correct. This is the same query used earlier, and it formats the output into a single page:

```
mysql> SELECT   kingdom_name AS "Kingdom"
    -> ,        population AS "Citizens"
    -> ,        knight_name AS "Knight's Name"
    -> ,        allegiance_start_date AS "Start Date"
    -> ,        allegiance_end_date AS "End Date"
    -> FROM     kingdom_knight_import;
```

It should return the following:

```
+---------+----------+-------------------------+------------+------------+
| Kingdom | Citizens | Knight's Name           | Start Date | End Date   |
+---------+----------+-------------------------+------------+------------+
| Narnia  |    77600 | 'Peter the Magnificent' | 1272-03-20 | 1292-06-09 |
| Narnia  |    77600 | 'Edmund the Just'       | 1272-03-20 | 1292-06-09 |
| Narnia  |    77600 | 'Susan the Gentle'      | 1272-03-20 | 1292-06-09 |
| Narnia  |    77600 | 'Lucy the Valiant'      | 1272-03-20 | 1292-06-09 |
| Narnia  |    42100 | 'Peter the Magnificent' | 1531-04-12 | 1531-05-31 |
| Narnia  |    42100 | 'Edmund the Just'       | 1531-04-12 | 1531-05-31 |
| Narnia  |    42100 | 'Susan the Gentle'      | 1531-04-12 | 1531-05-31 |
| Narnia  |    42100 | 'Lucy the Valiant'      | 1531-04-12 | 1531-05-31 |
+---------+----------+-------------------------+------------+------------+
```

As with the prior section's instructions, revisit the setup steps and fix what's missing if you don't see this data. Is it possible that you truncated this table when you removed the data from the target tables?

Step 4: Merge from the Import Source

Like the example of the `INSERT` statement with the `ON DUPLICATE KEY` clause, this example works with an internally managed table that mirrors the import source file's structure. The first example uses a `REPLACE INTO` statement with a distinct subquery (one returning a unique row set); the subquery performs an outer join between the import and target tables:

```
REPLACE INTO kingdom
( kingdom_id, kingdom_name, population )
(SELECT   DISTINCT
          k.kingdom_id
,         kki.kingdom_name
,         kki.population
FROM      kingdom_knight_import kki LEFT JOIN kingdom k
ON        kki.kingdom_name = k.kingdom_name
AND       kki.population = k.population);
```

Notice that this is the same `LEFT JOIN` that is used in the `INSERT` statement of the preceding example. It also uses a fully qualified target list, although one isn't required because the query's columns align themselves with the definition of the `kingdom` table.

The next `REPLACE INTO` statement performs a match between the `kingdom` and `knight` tables before an outer join against the `kingdom_knight_import` table. It is also the same query shown in the equivalent `INSERT ON DUPLICATE KEY` statement.

```
REPLACE INTO knight
(SELECT   kn.knight_id
,         kki.knight_name
,         k.kingdom_id
,         kki.allegiance_start_date AS start_date
,         kki.allegiance_end_date AS end_date
FROM      kingdom_knight_import kki INNER JOIN kingdom k
ON        kki.kingdom_name = k.kingdom_name
AND       kki.population = k.population LEFT JOIN knight kn
ON        k.kingdom_id = kn.kingdom_allegiance_id
AND       kki.knight_name = kn.knight_name
AND       kki.allegiance_start_date = kn.allegiance_start_date
AND       kki.allegiance_end_date = kn.allegiance_end_date);
```

Many developers prefer the `REPLACE INTO` statement over the `INSERT ON DUPLICATE KEY` statement for imports. They work more or less the same way, but the `INSERT ON DUPLICATE KEY` statement is more similar to the Oracle `MERGE` statement, and it makes the logic more portable between databases.

Summary

This chapter discussed how you import data in Oracle and MySQL databases. It showed you the process of leveraging external import sources through external files in Oracle and through the `LOAD DATA` and `LOAD XML` statements in MySQL.

Sample programs showed you how to take denormalized import source files and merge them into relational tables. You also learned how you can use the Oracle `MERGE` statement and the `INSERT ON DUPLICATE KEY` and `REPLACE INTO` statements in conjunction with outer joins to merge data with existing tables.

Mastery Check

The mastery check is a series of true or false and multiple choice questions that let you confirm how well you understand the material in the chapter. You may check the Appendix for answers to these questions.

1. **True** ☐ **False** ☐ In Oracle, you use external tables to import data from CSV files.
2. **True** ☐ **False** ☐ MySQL supports external files with the `LOAD LOCAL DATA INFILE` statement.
3. **True** ☐ **False** ☐ The definition of an external table relies on the ability to create virtual directories.
4. **True** ☐ **False** ☐ You need to grant only `READ ON DIRECTORY` privileges to read external data and write error log files.
5. **True** ☐ **False** ☐ The `ON` clause in an external table definition sets the criteria for whether or not you should insert or update records.
6. **True** ☐ **False** ☐ You can update columns used to evaluate joins in the `ON` clause of a subquery.
7. **True** ☐ **False** ☐ You can load XML files with the `LOAD DATA` statement.
8. **True** ☐ **False** ☐ You can load XML files with the `LOAD XML` statement.
9. **True** ☐ **False** ☐ The `INSERT ON DUPLICATE KEY` statement supports a multiple row `VALUES` clause.
10. **True** ☐ **False** ☐ The `REPLACE INTO` statement works only with a subquery.
11. Which of the following isn't a valid option of the `MERGE` statement in Oracle?
    A. The `OPTIONALLY ENCLOSED BY` option
    B. The `ACCESS PARAMETERS` option

- **C.** The `FIELDS TERMINATED BY` option
- **D.** The `ROWS IDENTIFIED BY` option
- **E.** The `DEFAULT DIRECTORY` option

12. Which of the following isn't a valid option of the `LOAD DATA` statement in MySQL?

- **A.** The `LINES TERMINATED BY` option
- **B.** The `ENCLOSED BY` option
- **C.** The `FIELDS TERMINATED BY` option
- **D.** The `ROWS IDENTIFIED BY` option
- **E.** The `ESCAPED BY` option

13. Which of the following isn't a valid option of the `LOAD XML` statement in MySQL?

- **A.** The `LINES TERMINATED BY` option
- **B.** The `ENCLOSED BY` option
- **C.** The `FIELDS TERMINATED BY` option
- **D.** The `ROWS IDENTIFIED BY` option
- **E.** The `ESCAPED BY` option

14. Which of the following options or clauses enables a `LOAD DATA` statement to replace the values of non-key columns?

- **A.** A `REPLACE` option
- **B.** An `IGNORE` option
- **C.** An `IGNORE` *`n`* `LINES` option
- **D.** An `IGNORE` *`n`* `ROWS` option
- **E.** An `ESCAPED BY` clause

15. The target table of an `INSERT ON DUPLICATE KEY` statement requires which of the following?

- **A.** A primary key constraint
- **B.** A foreign key constraint
- **C.** An index
- **D.** A non-unique key
- **E.** None of the above