

Chapter 7. Doing More with MySQL

MySQL is feature-rich. Over the past three chapters, you've seen the wide variety of techniques that can be used to query, modify, and manage data. However, there's still much more that MySQL can do, and some of those additional features are the subject of this chapter.

In this chapter, you'll learn how to:

- Insert data into a database from other sources, including with queries and from text files.
- Perform updates and deletes using multiple tables in a single statement.
- Replace data.
- Use MySQL functions in queries to meet more complex information needs.
- Analyze queries using the `EXPLAIN` statement and then improve their performance with simple optimization techniques.
- Use alternative storage engines to change table properties.

Inserting Data Using Queries

Much of the time, you'll create tables using data from another source. The examples you saw in [Chapter 3](#) therefore illustrate only part of the problem: they show you how to insert data that's already in the form you want (that is, formatted as a SQL `INSERT` statement). The other ways to insert data include using SQL `SELECT` statements on other tables or databases and reading in files from other sources. This section shows you how to tackle the former method of inserting data; you'll learn how to insert data from a file of comma-separated values in the next section, ["Loading Data from Comma-Delimited Files"](#).

Suppose we've decided to create a new table in the `sakila` database. It's going to store a random list of movies that we want to advertise more heavily. In the real world, you'd probably want to use some data science to find out what movies to highlight, but we're going to stick to the basics. This list of

films will be a way for customers to check out different parts of the catalog, rediscover some old favorites, and learn about hidden treasures they haven't yet explored. We've decided to structure the table as follows:

```
mysql> CREATE TABLE recommend
->     film_id SMALLINT UNSIGNED,
->     language_id TINYINT UNSIGNED,
->     release_year YEAR,
->     title VARCHAR(128),
->     length SMALLINT UNSIGNED,
->     sequence_id SMALLINT AUTO_INCREMENT,
->     PRIMARY KEY (sequence_id)
-> );
```

```
Query OK, 0 rows affected (0.05 sec)
```

This table stores a few details about each film, allowing you to find the actor, category, and other information using simple queries on the other tables. It also stores a `sequence_id`, which is a unique number that enumerates where the film is in our short list. When you start using the recommendation feature, you'll first see the movie with a `sequence_id` of 1, then 2, and so on. You can see that we're using the MySQL `AUTO_INCREMENT` feature to allocate the `sequence_id` values.

Now we need to fill up our new `recommend` table with a random selection of films. Importantly, we're going to do the `SELECT` and `INSERT` together in one statement. Here we go:

```
mysql> INSERT INTO recommend (film_id, language_id, re
->     SELECT film_id, language_id, release_year, title
->     FROM film ORDER BY RAND() LIMIT 10;
```

```
Query OK, 10 rows affected (0.02 sec)
Records: 10  Duplicates: 0  Warnings: 0
```

Now, let's investigate what happened before we explain how this command works:

```
mysql> SELECT * FROM recommend;
```

film_id	...	title	length	sequence_id
542	...	LUST LOCK	52	1
661	...	PAST SUICIDES	157	2
613	...	MYSTIC TRUMAN	92	3
757	...	SAGEBRUSH CLUELESS	106	4
940	...	VICTORY ACADEMY	64	5
917	...	TUXEDO MILE	152	6
709	...	RACER EGG	147	7
524	...	LION UNCUT	50	8
30	...	ANYTHING SAVANNAH	82	9
602	...	MOURNING PURPLE	146	10

10 rows in set (0.00 sec)

You can see that we have 10 films in our recommendation list, numbered with `sequence_id` values from 1 to 10. We're ready to start recommending the random movie selection. Don't worry if your results differ; it's a consequence of how the `RAND()` function works.

There are two parts to the SQL statement we used to populate the table: an `INSERT INTO` and a `SELECT`. The `INSERT INTO` statement lists the destination table into which the data will be stored, followed by an optional list of column names in parentheses; if you omit the column names, all columns in the destination table are assumed in the order they appear in the output of a `DESCRIBE TABLE` or `SHOW CREATE TABLE` statement. The `SELECT` statement outputs columns that must match the type and order of the list provided for the `INSERT INTO` statement (or the implicit complete list if one isn't provided). The overall effect is that the rows output from the `SELECT` statement is inserted into the destination table by the `INSERT INTO` statement. In our example, `film_id`, `language_id`, `release_year`, `title`, and `length` values from the `film` table are inserted into the five columns with the same names and types in the `recommend` table; the `sequence_id` is automatically created using MySQL's `AUTO_INCREMENT` feature, so it isn't specified in the statements.

Our example includes the clause `ORDER BY RAND()`; this orders the results according to the MySQL function `RAND()`. The `RAND()` function returns a pseudorandom number in the range 0 to 1:

```
mysql> SELECT RAND();  
  
+-----+  
| RAND() |  
+-----+  
| 0.4593397513584604 |  
+-----+  
1 row in set (0.00 sec)
```

A pseudorandom number generator doesn't generate truly random numbers, but rather generates numbers based on some property of the system, such as the time of day. This is sufficiently random for most applications; a notable exception is cryptography applications that depend on the true randomness of numbers for security.

If you ask for the `RAND()` value in a `SELECT` operation, you'll get a random value for each returned row:

```
mysql> SELECT title, RAND() FROM film LIMIT 5;  
  
+-----+-----+  
| title | RAND() |  
+-----+-----+  
| ACADEMY DINOSAUR | 0.5514843506286706 |  
| ACE GOLDFINGER | 0.37940252980161693 |  
| ADAPTATION HOLES | 0.2425596278557178 |  
| AFFAIR PREJUDICE | 0.07459058060738312 |  
| AFRICAN EGG | 0.6452740502034072 |  
+-----+-----+  
5 rows in set (0.00 sec)
```

Since the values are effectively random, you'll almost certainly see different results than we've shown here. Moreover, if you repeat the statement, you'll also see different values returned. It is possible to pass `RAND()` an integer argument called *seed*. That will result in the `RAND()` function generating the

same values for the same inputs each time that seed is used—it's not really useful for what we're trying to achieve here, but a possibility nonetheless. You can try running the following statement as many times as you want, and the results won't change:

```
SELECT title, RAND(1) FROM film LIMIT 5;
```

Let's return to the `INSERT` operation. When we ask that the results be ordered by `RAND()`, the results of the `SELECT` statement are sorted in a pseudorandom order. The `LIMIT 10` is there to limit the number of rows returned by the `SELECT`; we've limited in this example simply for readability.

The `SELECT` statement in an `INSERT INTO` statement can use all of the usual features of `SELECT` statements. You can use joins, aggregation, functions, and any other features you choose. You can also query data from one database in another, by prefacing the table names with the database name followed by a period (`.`) character. For example, if you wanted to insert the `actor` table from the `film` database into a new `art` database, you could do the following:

```
mysql> CREATE DATABASE art;
```

```
Query OK, 1 row affected (0.01 sec)
```

```
mysql> USE art;
```

```
Database changed
```

```
mysql> CREATE TABLE people
      ->     person_id SMALLINT UNSIGNED,
      ->     first_name VARCHAR(45),
      ->     last_name VARCHAR(45),
      ->     PRIMARY KEY (person_id)
      -> );
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
mysql> INSERT INTO art.people (person_id, first_name, ]  
-> SELECT actor_id, first_name, last_name FROM sakil
```

```
Query OK, 200 rows affected (0.01 sec)
```

```
Records: 200  Duplicates: 0  Warnings: 0
```

You can see that the new `people` table is referred to as `art.people` (though it doesn't need to be, since `art` is the database that's currently in use), and the `actor` table is referred to as `sakila.actor` (which it needs to be, since that isn't the database being used). Note also that the column names don't need to be the same for the `SELECT` and the `INSERT`.

Sometimes, you'll encounter duplication issues when inserting with a `SELECT` statement. If you try to insert the same primary key value twice, MySQL will abort. This won't happen in the `recommend` table, as long as you automatically allocate a new `sequence_id` using the `AUTO_INCREMENT` feature. However, we can force a duplicate into the table to show the behavior:

```
mysql> USE sakila;
```

```
Database changed
```

```
mysql> INSERT INTO recommend (film_id, language_id, re]   
-> title, length, sequence_id )  
-> SELECT film_id, language_id, release_year, title  
-> FROM film LIMIT 1;
```

```
ERROR 1062 (23000): Duplicate entry '1' for key 'recomm
```

If you want MySQL to ignore this and keep going, add the `IGNORE` keyword after `INSERT`:

```
mysql> INSERT IGNORE INTO recommend (film_id, language_
-> title, length, sequence_id )
-> SELECT film_id, language_id, release_year, title
-> FROM film LIMIT 1;
```

```
Query OK, 0 rows affected, 1 warning (0.00 sec)
Records: 1  Duplicates: 1  Warnings: 1
```

MySQL doesn't complain, but it does report that it encountered a duplicate. Note that the data is not changed; all we did was ignore the error. This is useful in bulk load operations where you don't want to fail halfway through running a script that inserts a million rows. We can inspect the warning to see the *Duplicate entry* error as a warning now:

```
mysql> SHOW WARNINGS;
```

```
+-----+-----+
| Level | Code | Message
+-----+-----+
| Warning | 1062 | Duplicate entry '1' for key 'recommen
+-----+-----+
1 row in set (0.00 sec)
```

Finally, note that it is possible to insert into a table that's listed in the `SELECT` statement, but you still need to avoid duplicate primary keys:

```
mysql> INSERT INTO actor SELECT
-> actor_id, first_name, last_name, NOW() FROM acto
```

```
ERROR 1062 (23000): Duplicate entry '1' for key 'actor.'
```

There are two ways to avoid getting the error. First, the `actor` table has `AUTO_INCREMENT` enabled for `actor_id`, so if you omit this column in the `INSERT` completely, you won't get an error, as the new values will be generated automatically. (`INSERT` statement syntax is explained in

[“Alternative Syntaxes”](#).) Here’s an example that would only one record (due to the `LIMIT` clause):

```
INSERT INTO actor(first_name, last_name, last_update)
SELECT first_name, last_name, NOW() FROM actor LIMIT 1;
```

The second way is to modify `actor_id` in the `SELECT` query in a way that prevents collisions. Let’s try that:

```
mysql> INSERT INTO actor SELECT
-> actor_id+200, first_name, last_name, NOW() FROM
```

```
Query OK, 200 rows affected (0.01 sec)
Records: 200  Duplicates: 0  Warnings: 0
```

Here, we’re copying the rows but increasing their `actor_id` values by 200 before we insert them, because we remember that there are 200 rows initially.

This is the result:

```
mysql> SELECT * FROM actor;
```

actor_id	first_name	last_name	last_update
1	PENELOPE	GUINESS	2006-02-15 04
2	NICK	WAHLBERG	2006-02-15 04
...			
198	MARY	KEITEL	2006-02-15 04
199	JULIA	FAWCETT	2006-02-15 04
200	THORA	TEMPLE	2006-02-15 04
201	PENELOPE	GUINESS	2021-02-28 10
202	NICK	WAHLBERG	2021-02-28 10
...			
398	MARY	KEITEL	2021-02-28 10
399	JULIA	FAWCETT	2021-02-28 10
400	THORA	TEMPLE	2021-02-28 10

400 rows in set (0.00 sec)

You can see how first names, last names, and `last_update` values start repeating from the `actor_id` 201.

It's also possible to use subqueries in the `INSERT SELECT` statements. For example, the next statement is valid:

```
*INSERT INTO actor SELECT * FROM*
*(SELECT actor_id+400, first_name, last_name, NOW() FRC
```

Loading Data from Comma-Delimited Files

These days, databases are usually not an afterthought. They are ubiquitous and easier than ever to use, and most IT professionals know about them.

Nevertheless, end users find them difficult, and unless specialized UIs are created, a lot of data entry and analysis is instead done in various spreadsheet programs. These programs typically have unique file formats, open or closed, but most of them will allow you to export data as rows of comma-separated values (CSVs), also called a *comma-delimited format*. You can then import the data with a little effort into MySQL.

Another common task that can be accomplished by working with CSVs is transferring data in a heterogeneous environment. If you have various database software running in your setup, and especially if you're using a DBaaS in the cloud, moving data between these systems can be daunting. However, the basic CSV data can be a lowest common denominator for them. Note that in the case of any data transfer you should always remember that CSV does not have the notions of schemas, data types, or constraints. But as a flat data file format, it works well.

If you're not using a spreadsheet program, you can still often use command-line tools such as `sed` and `awk`—very old and powerful Unix utilities—to convert text data into a CSV format suitable for import by MySQL. Some cloud databases allow export of their data directly into CSV. In some other cases, small programs have to be written that read data and produce a CSV file. This section shows you the basics of how to import CSV data into MySQL.

Let's work through an example. We have a list of NASA facilities with their addresses and contact information that we want to store in a database. At present, it's stored in a CSV file named *NASA_Facilities.csv* and has the format shown in [Figure 7-1](#).

	A	B	C	D	E	F
1	Center	Center Search Status	Facility	FacilityURL	Occupied	Status
2	Kennedy Space Center	Public	Control Room 211/269/HQB/S	03/01/1967 12:00:00 AM		
3	Langley Research Center	Public	Microgravity/LDE Analysis Laboratory	03/01/1965 12:00:00 AM	Active	
4	Kennedy Space Center	Public	SRM Rotation and Processing Facility/KS-0494	03/01/1964 12:00:00 AM		
5	Marshall Space Flight Center	Public	ET WTC - 14-inch Torsion Wind Tunnel 4732	03/01/1966 12:00:00 AM	Active	
6	Kennedy Space Center	Public	EE 18 - Three Moment Gyro Test & Exp Fac 4487	03/01/1964 12:00:00 AM	Active	
7	Kennedy Space Center	Public	MMI - Sound Processing Bay/West High Bay/MF-1212	03/01/1964 12:00:00 AM		
8	Kennedy Space Center	Public	Central Supply Warehouse (MS-744)	03/01/1964 12:00:00 AM		
9	Jet Propulsion Lab	Public	DSS 46 Antenna	03/01/1964 12:00:00 AM		
10	Jet Propulsion Lab	Public	DSS 65 Antenna	03/01/1963 12:00:00 AM	Inactive	
11	Kennedy Space Center	Public	Utility Annex/X6-947/VAB AREA A	03/01/1966 12:00:00 AM		
12	Jet Propulsion Lab	Public	Microbiology Laboratory (MDL)	03/01/1990 12:00:00 AM	Active	
13	Glenn Research Center	Public	Thermal Stress A-2 #468	03/01/1964 12:00:00 AM	Active	
14	Marshall Space Flight Center	Public	LP - 100 Research Facility 4563	03/01/1967 12:00:00 AM	Active	
15	Wallops Flight Facility/GSEC	Public	WFF Research Airport	03/01/1968 12:00:00 AM	Active	
16	Marshall Space Flight Center	Public	IHF Vacuum Plasma Spray Facility 4707	03/01/1966 12:00:00 AM	Active	
17	Marshall Space Flight Center	Public	ET Test Facility (TS300) 4530	03/01/1966 12:00:00 AM	Active	
18	Jet Propulsion Lab	Public	DSS 16 Antenna	03/01/1966 12:00:00 AM	Active	
19	Jet Propulsion Lab	Public	Thermal Vacuum Chamber 306-TV-11	03/01/1968 12:00:00 AM	Inactive	

Figure 7-1. List of NASA facilities stored in a spreadsheet file

You can see that each facility is associated with a center, and may list the date it was occupied and optionally its status. The full column list is as follows:

- Center
- Center Search Status
- Facility
- FacilityURL
- Occupied
- Status
- URL Link
- Record Date
- Last Update
- Country
- Contact
- Phone
- Location
- City
- State
- Zipcode

This example comes directly from NASA's publicly available [Open Data Portal](#), and the file is available in the book's [GitHub repository](#). Since this is already a CSV file, we don't need to convert it from another file format (like XLS). However, if you do need to do that in your own project, it's usually as easy as using the Save As command in the spreadsheet program; just don't forget to pick CSV as the output format.

If you open the *NASA_facilities.csv* file using a text editor, you'll see that it has one line per spreadsheet row, with the values for each column separated by commas. If you're on a non-Windows platform, you may find that in some CSV files each line is terminated with a ^M, but don't worry about this; it's an artifact of the origins of Windows. Data in this format is often referred to as *DOS format*, and most software applications can handle it without problem. In our case, the data is in *Unix format*, and thus on Windows you may see that all the lines are concatenated. You can try to use another text editor if that's the case. Here are a few width-truncated lines selected from *NASA_Facilities.csv*:

```
Center,Center Search Status,Facility,FacilityURL,Occupi
Kennedy Space Center,Public,Control Room 2/1726/HGR-S ,
Langley Research Center,Public,Micrometeoroid/LDEF Analys
Kennedy Space Center,Public,SRM Rotation and Processing
Marshall Space Flight Center,..."35812(34.729538, -86.5
```

If there are commas or other special symbols within values, the whole value is enclosed in quotes, as in the last line shown here.

Let's import this data into MySQL. First, create the new `nasa` database:

```
mysql> CREATE DATABASE nasa;
```

```
Query OK, 1 row affected (0.01 sec)
```

Choose this as the active database:

```
mysql> USE nasa;
```

```
Database changed
```

Now, create the `facilities` table to store the data. This needs to handle all of the fields that we see in the CSV file, which conveniently has a header:

```
mysql> CREATE TABLE facilities (
    ->   center TEXT,
    ->   center_search_status TEXT,
```

```
->     facility TEXT,  
->     facility_url TEXT,  
->     occupied TEXT,  
->     status TEXT,  
->     url_link TEXT,  
->     record_date DATETIME,  
->     last_update TIMESTAMP NULL,  
->     country TEXT,  
->     contact TEXT,  
->     phone TEXT,  
->     location TEXT,  
->     city TEXT,  
->     state TEXT,  
->     zipcode TEXT  
-> );
```

```
Query OK, 0 rows affected (0.03 sec)
```

We're cheating here somewhat with the data types. NASA provides the schema of the dataset, but for most of the fields the type is given as the "Plain Text," and we can't really store a "Website URL" as anything but text, either. We don't, however, know how much data each column will hold. Thus, we default to using the `TEXT` type, which is similar to defining a column as `VARCHAR(65535)`. There are some differences between the two types, as you can probably remember from ["String types"](#), but they are not important in this example. We don't define any indexes and don't put any constraints on our table. If you're loading a completely new dataset that's quite small, it can be beneficial to load it first and then analyze it. For larger datasets, make sure the table is structured as well as possible, or you'll spend a considerable amount of time changing it later.

Now that we've set up the database table, we can import the data from the file using the `LOAD DATA INFILE` command:

```
mysql> LOAD DATA INFILE 'NASA_Facilities.csv' INTO TABLE  
-> FIELDS TERMINATED BY ',';
```

ERROR 1290 (HY000): The MySQL server is running with the `--secure-file-priv` option so it cannot execute this

Oh, no! We got an error. By default, MySQL doesn't let you load *any* data using the `LOAD DATA INFILE` command. The behavior is controlled by the `secure_file_priv` system variable. If the variable is set to a path, the file to be loaded should reside in that particular path and be readable by the MySQL server. If the variable isn't set, which is considered insecure, then the file to be loaded should be readable only by the MySQL server. By default, MySQL 8.0 on Linux sets this variable as follows:

```
mysql> SELECT @@secure_file_priv;
```

```
+-----+  
| @@secure_file_priv |  
+-----+  
| /var/lib/mysql-files/ |  
+-----+  
1 row in set (0.00 sec)
```

And on Windows:

```
mysql> SELECT @@secure_file_priv;
```

```
+-----+  
| @@secure_file_priv |  
+-----+  
| C:\ProgramData\MySQL\MySQL Server 8.0\Uploads\ |  
+-----+  
1 row in set (0.00 sec)
```

NOTE

The value of the `secure_file_priv` system variable may be different in your installation of MySQL, or it may even be empty. A `NULL` value for `secure_file_priv` means that MySQL will allow loading a file in any location as long as that file is accessible to the MySQL server. On Linux, that means that the file has to be readable by the `mysqld` process, which usually runs under the `mysql` user. You can change `secure_file_priv` variable's value by updating your MySQL configuration and restarting the server. You can find information on how to configure MySQL in [Chapter 9](#).

On Linux or other Unix-like systems, you need to copy the file into that directory, possibly using `sudo` to allow the operation, and then change its permissions so that the `mysqld` program can access the file. On Windows, you only need to copy the file to the correct destination.

Let's do this. On Linux or similar systems, you can run commands like this:

```
$ ls -lh $HOME/Downloads/NASA_Facilities.csv
-rw-r--r--. 1 skuzmichev skuzmichev 114K
              Feb 28 14:19 /home/skuzmichev/Downloads/NASA_Facili
$ sudo cp -vip ~/Downloads/NASA_Facilities.csv /var/lib/mysql-files/
[sudo] password for skuzmichev:
'/home/skuzmichev/Downloads/NASA_Facilities.csv'
-> '/var/lib/mysql-files/NASA_Facilities.csv'
$ sudo chown mysql:mysql /var/lib/mysql-files/NASA_Faci
$ sudo ls -lh /var/lib/mysql-files/NASA_Facilities.csv
-rw-r--r--. 1 mysql mysql 114K
              Feb 28 14:19 /var/lib/mysql-files/NASA_Facilities.c
```

On Windows, you can use the File Manager to copy or move the file.

Now we're ready to try the loading again. When our target file is not in the current directory, we need to pass the full path to the command:

```
mysql> LOAD DATA INFILE '/var/lib/mysql-files/NASA_Faci
      -> INTO TABLE facilities FIELDS TERMINATED BY ',';
```

```
ERROR 1292 (22007): Incorrect datetime value:
'Record Date' for column 'record_date' at row 1
```

Well, that doesn't look correct: `Record Date` is indeed not a date, but a column name. We've made a silly but common mistake, loading the CSV file with the header. We need to tell MySQL to omit it:

```
mysql> LOAD DATA INFILE '/var/lib/mysql-files/NASA_Faci
      -> INTO TABLE facilities FIELDS TERMINATED BY ',''
      -> IGNORE 1 LINES;
```

```
ERROR 1292 (22007): Incorrect datetime value:  
'03/01/1996 12:00:00 AM' for column 'record_date' at row 1
```

Turns out, that date format we have is not something MySQL expects. That's an extremely common issue. There are a couple of ways out. First, we can just change our `record_date` column to the `TEXT` type. We'll lose the niceties of a proper date-time data type, but we'll be able to get the data into our database. Second, we can convert the data ingested from the file on the fly. To demonstrate the difference in the results, we specified the `occupied` column (which is a date field) to be `TEXT`. Before we jump into the conversion complexities, though, let's try running the same command on Windows:

```
mysql> LOAD DATA INFILE  
-> 'C:\ProgramData\MySQL\MySQL Server 8.0\Uploads\facilities.csv'  
-> INTO TABLE facilities FIELDS TERMINATED BY ',';
```

```
ERROR 1290 (HY000): The MySQL server is running with  
the --secure-file-priv option so it cannot execute this
```

Even though the file is present in that directory, `LOAD DATA INFILE` errors out. The reason for that is how MySQL works with paths on Windows. We can't just use regular Windows-style paths with this or other MySQL commands. We need to escape each backslash (\) with another backslash, or change our path to use forward slashes (/). Both will work...or rather, in this case, both will error out due to the expected `record_date` conversion issue:

```
mysql> LOAD DATA INFILE  
-> 'C:\\ProgramData\\MySQL\\MySQL Server 8.0\\Uploads\\facilities.csv'  
-> INTO TABLE facilities FIELDS TERMINATED BY ',';
```

```
ERROR 1292 (22007): Incorrect datetime value:  
'Record Date' for column 'record_date' at row 1
```

```
mysql> LOAD DATA INFILE  
-> 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/facilities.csv'
```

```
-> INTO TABLE facilities FIELDS TERMINATED BY ',';
```

```
ERROR 1292 (22007): Incorrect datetime value:  
'Record Date' for column 'record_date' at row 1
```

With that covered, let's get back to our date conversion issue. As we mentioned, this is an extremely common issue. You will inevitably face type conversion problems, because CSV is typeless, and different databases have different expectations about various types. In this case, the open dataset that we obtained has dates in the following format: `03/01/1996 12:00:00 AM`. While this will make our operation more complex, we believe converting the date values from our CSV file is a good exercise. To convert an arbitrary string to a date, or at least to attempt such a conversion, we can use the `STR_TO_DATE()` function. After reviewing the documentation, we came up with the following cast:

```
mysql> SELECT STR_TO_DATE('03/01/1996 12:00:00 AM',  
-> '%m/%d/%Y %h:%i:%s %p') converted;
```

```
+-----+  
| converted |  
+-----+  
| 1996-03-01 00:00:00 |  
+-----+  
1 row in set (0.01 sec)
```

Since the function returns `NULL` when the cast is unsuccessful, we know we have managed to find a correct invocation. Now we need to find out how to use the function in the `LOAD DATA INFILE` command. The much longer version using the function looks like this:

```
mysql> LOAD DATA INFILE '/var/lib/mysql-files/NASA_Faci  
-> INTO TABLE facilities FIELDS TERMINATED BY ','  
-> OPTIONALLY ENCLOSED BY '\"'  
-> IGNORE 1 LINES  
-> (center, center_search_status, facility, facilit  
-> occupied, status, url_link, @var_record_date, @\br/>-> country, contact, phone, location, city, state,  
-> SET record_date = IF(
```

```
->      CHAR_LENGTH(@var_record_date)=0, NULL,
->      STR_TO_DATE(@var_record_date, '%m/%d/%Y %h:%
-> ),
-> last_update = IF(
->      CHAR_LENGTH(@var_last_update)=0, NULL,
->      STR_TO_DATE(@var_last_update, '%m/%d/%Y %h:%
-> );
```

```
Query OK, 485 rows affected (0.05 sec)
Records: 485 Deleted: 0 Skipped: 0 Warnings: 0
```

That's a lot of command! Let's break it down. The first line specifies our `LOAD DATA INFILE` command and the path to a file. The second line specifies the target table and begins the `FIELDS` specification, starting with `TERMINATED BY ','`, which means our fields are delimited by commas, as expected for CSV. The third line adds another parameter to the `FIELDS` specification and tells MySQL that some fields (but not all) are enclosed by the `"` symbol. That's important, because our dataset has some entries with commas within `"..."` fields. On line four we specify that we skip the first line of the file, where we know the header resides.

Lines 5 through 7 have the column list specification. We need to convert two date-time columns, and for that we need to read their values into variables, which are then set to the `nasa.facilities` table's column values. However, we can't tell that to MySQL without also specifying all the other columns. If we were to omit some columns from the list or specify them in the wrong order, MySQL would not assign the values correctly. CSV is inherently a position-based format. By default, when the `FIELDS` specification is not given, MySQL will read each CSV line and will expect each field in all lines to map to a column in the target table (in the order of columns that the `DESCRIBE` or `SHOW CREATE TABLE` command gives). By changing the order of columns in this specification, we can populate a table from a CSV file that has fields misplaced. By specifying fewer columns, we can populate a table from a file that is missing some of the fields.

Lines 8 through 15 are our function calls to convert the date-time values. In the preceding column spec, we defined that field 8 is read into the `@var_record_date` variable, and field 9 into `@var_last_update`. We know that fields 8 and 9 are our problematic date-time fields. With the

variables populated, we can define the `SET` parameter, which allows modification of the target table column values based on fields read from the CSV file. In this very basic example, you could multiply a specific value by two. In our case, we cast two functions: first we check that a variable is not empty (`,` in CSV) by assessing the number of characters read from the file, and second we call the actual conversion if the previous check doesn't return zero. If we found the length to be zero, we set the value to `NULL`.

Finally, when the command has been executed, it's possible to check the results:

```
mysql> SELECT facility, occupied, last_update
-> FROM facilities
-> ORDER BY last_update DESC LIMIT 5;
```

facility	occupied	last_update
Turn Basin/K7-1005	01/01/1963 12:00:00 AM	201
RPSF Surge Building	01/01/1984 12:00:00 AM	201
Thermal Protection S...	01/01/1988 12:00:00 AM	201
Intermediate Bay/M7-...	01/01/1995 12:00:00 AM	201
Orbiter Processing F...	01/01/1987 12:00:00 AM	201

5 rows in set (0.00 sec)

Remember we mentioned that `occupied` will remain `TEXT`. You can see that here. While it can be used for sorting, no date functions will work on values in this column unless they are explicitly cast to `DATETIME`.

This was a complex example, but it shows the unexpected complexity of loading data and the power of the `LOAD DATA INFILE` command.

Writing Data into Comma-Delimited Files

You can use the `SELECT INTO OUTFILE` statement to write out the result of a query into a CSV file that can be opened by a spreadsheet or other program.

Let's export the list of current managers from our `employees` database into a CSV file. The query used to list all the current managers is shown here:

```
mysql> USE employees;
```

Database changed

```
mysql> SELECT emp_no, first_name, last_name, title, fro
-> FROM employees JOIN titles USING (emp_no)
-> WHERE title = 'Manager' AND to_date = '9999-01-01'
```

emp_no	first_name	last_name	title	from_date
110039	Vishwani	Minakawa	Manager	1991-10-01
110114	Isamu	Legleitner	Manager	1989-12-31
110228	Karsten	Sigstam	Manager	1992-03-15
110420	Oscar	Ghazalie	Manager	1996-08-16
110567	Leon	DasSarma	Manager	1992-04-23
110854	Dung	Pesch	Manager	1994-06-09
111133	Hauke	Zhang	Manager	1991-03-15
111534	Hilary	Kambil	Manager	1991-04-01
111939	Yuchang	Weedman	Manager	1996-01-15

9 rows in set (0.13 sec)

We can change this `SELECT` query slightly to write this data into an output file as comma-separated values. `INTO OUTFILE` is subject to the same --secure-file-priv option rules as `LOAD DATA INFILE`. The file path by default is limited, and we listed the default options in "[Loading Data from Comma-Delimited Files](#)".

```
mysql> SELECT emp_no, first_name, last_name, title, fro
-> FROM employees JOIN titles USING (emp_no)
-> WHERE title = 'Manager' AND to_date = '9999-01-01'
-> INTO OUTFILE '/var/lib/mysql-files/managers.csv'
-> FIELDS TERMINATED BY ',';
```

```
Query OK, 9 rows affected (0.14 sec)
```

Here, we've saved the results into the file *managers.csv* in the */var/lib/mysql-files* directory; the MySQL server must be able to write to the directory that you specify, and it should be one listed in the `secure_file_priv` system variable (if set). On a Windows system, specify a path such as *C:\ProgramData\MySQL\MySQL Server 8.0\Uploads\managers.csv* instead. If you omit the `FIELDS TERMINATED BY` clause, the server will use tabs as the default separator between the data values.

You can view the contents of the file *managers.csv* in a text editor, or import it into a spreadsheet program:

```
110039,Vishwani,Minakawa,Manager,1991-10-01
110114,Isamu,Legleitner,Manager,1989-12-17
110228,Karsten,Sigstam,Manager,1992-03-21
110420,Oscar,Ghazalie,Manager,1996-08-30
110567,Leon,DasSarma,Manager,1992-04-25
110854,Dung,Pesch,Manager,1994-06-28
111133,Hauke,Zhang,Manager,1991-03-07
111534,Hilary,Kambil,Manager,1991-04-08
111939,Yuchang,Weedman,Manager,1996-01-03
```

When our data fields contain commas or another delimiter of our choice, MySQL by default will escape the delimiters within fields. Let's switch to the `sakila` database and test this:

```
mysql> USE sakila;
```

```
Database changed
```

```
mysql> SELECT title, special_features FROM film LIMIT 1
      -> INTO OUTFILE '/var/lib/mysql-files/film.csv'
      -> FIELDS TERMINATED BY ',';
```

```
Query OK, 10 rows affected (0.00 sec)
```

If you take a look at the data in the *film.csv* file now (again, feel free to use a text editor, a spreadsheet program, or a command-line utility like `head` on Linux), here's what you'll see:

```
ACADEMY DINOSAUR,Deleted Scenes\,Behind the Scenes
ACE GOLDFINGER,Trailers\,Deleted Scenes
ADAPTATION HOLES,Trailers\,Deleted Scenes
AFFAIR PREJUDICE,Commentaries\,Behind the Scenes
AFRICAN EGG,Deleted Scenes
AGENT TRUMAN,Deleted Scenes
AIRPLANE SIERRA,Trailers\,Deleted Scenes
AIRPORT POLLOCK,Trailers
ALABAMA DEVIL,Trailers\,Deleted Scenes
ALADDIN CALENDAR,Trailers\,Deleted Scenes
```

Notice how in rows where the second field contains a comma, it has been automatically escaped with a backslash to distinguish it from the separator. Some spreadsheet programs may understand this and remove the backslashes when importing the file, and some may not. MySQL will respect the escaping and not treat such commas as separators. Note that if we specified `FIELDS TERMINATED BY '^'`, all ^ symbols within fields would get escaped; this is not specific to commas.

Since not all programs may deal with escapes gracefully, we can ask MySQL to explicitly define fields by using the `ENCLOSED` option:

```
mysql> SELECT title, special_features FROM film LIMIT 1
-> INTO OUTFILE '/var/lib/mysql-files/film_quoted.csv'
-> FIELDS TERMINATED BY ',' ENCLOSED BY '\"';
```

Query OK, 10 rows affected (0.00 sec)

We used this option before when loading data. Take a look at the results in the file *film_quoted.csv*:

```
"ACADEMY DINOSAUR","Deleted Scenes,Behind the Scenes"
"ACE GOLDFINGER","Trailers,Deleted Scenes"
"ADAPTATION HOLES","Trailers,Deleted Scenes"
"AFFAIR PREJUDICE","Commentaries,Behind the Scenes"
```

```
"AFRICAN EGG", "Deleted Scenes"  
"AGENT TRUMAN", "Deleted Scenes"  
"AIRPLANE SIERRA", "Trailers,Deleted Scenes"  
"AIRPORT POLLOCK", "Trailers"  
"ALABAMA DEVIL", "Trailers,Deleted Scenes"  
"ALADDIN CALENDAR", "Trailers,Deleted Scenes"
```

Our delimiters—commas—are now not escaped, which may work better with modern spreadsheet programs. You may wonder what will happen if there are double quotes within exported fields: MySQL will escape those instead of the commas, which again may cause problems. When doing data export, do not forget to make sure that the resulting output will work for your consumers. ➔

Creating Tables with Queries

You can create a table or easily create a copy of a table using a query. This is useful when you want to build a new database using existing data—for example, you might want to copy across a list of countries—or when you want to reorganize data for some reason. Data reorganization is common when producing reports, merging data from two or more tables, and redesigning on the fly. This short section shows you how it's done.

TIP

We base all the examples here on the unmodified `sakila` database. You should repeat the steps given in “[Entity Relationship Modeling Examples](#)” to get the database back to its clean state before proceeding.

In MySQL, you can easily duplicate the structure of a table using a variant of the `CREATE TABLE` syntax:

```
mysql> USE sakila;
```

```
Database changed
```

```
mysql> CREATE TABLE actor_2 LIKE actor;
```

```
Query OK, 0 rows affected (0.24 sec)
```

```
mysql> DESCRIBE actor_2;
```

Field	Type	Null	Key	...
actor_id	smallint unsigned	NO	PRI	...
first_name	varchar(45)	NO		...
last_name	varchar(45)	NO	MUL	...
last_update	timestamp	NO		...

Default	Extra
NULL	auto_increment
NULL	
NULL	
CURRENT_TIMESTAMP	DEFAULT_GENERATED on update CURRENT_TIMESTAMP

4 rows in set (0.01 sec)

```
mysql> SELECT * FROM actor_2;
```

```
Empty set (0.00 sec)
```

The `LIKE` syntax allows you to create a new table with exactly the same structure as another, including keys. You can see that it doesn't copy the data across. You can also use the `IF NOT EXISTS` and `TEMPORARY` features with this syntax.

If you want to create a table and copy some data, you can do that with a combination of the `CREATE TABLE` and `SELECT` statements. Let's remove the `actor_2` table and re-create it using this new approach:

```
mysql> DROP TABLE actor_2;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
mysql> CREATE TABLE actor_2 AS SELECT * from actor;
```

```
Query OK, 200 rows affected (0.03 sec)
```

```
Records: 200  Duplicates: 0  Warnings: 0
```

```
mysql> SELECT * FROM actor_2 LIMIT 5;
```

actor_id	first_name	last_name	last_update
1	PENELOPE	GUINNESS	2006-02-15 04:
2	NICK	WAHLBERG	2006-02-15 04:
3	ED	CHASE	2006-02-15 04:
4	JENNIFER	DAVIS	2006-02-15 04:
5	JOHNNY	LOLLOBRIGIDA	2006-02-15 04:

5 rows in set (0.01 sec)

An identical table, `actor_2`, is created, and all the data is copied across by the `SELECT` statement. `CREATE TABLE AS SELECT`, or `CTAS`, is a common name for this action, but it's actually not mandatory to specify the `AS` part, and we'll omit that later.

This technique is powerful. You can create new tables with new structures and use powerful queries to populate them with data. For example, here's a `report` table that's created to contain the names of the films in our database and their categories:

```
mysql> CREATE TABLE report (title VARCHAR(128), category
-> SELECT title, name AS category FROM
-> film JOIN film_category USING (film_id)
-> JOIN category USING (category_id);
```

```
Query OK, 1000 rows affected (0.06 sec)
Records: 1000  Duplicates: 0  Warnings: 0
```

You can see that the syntax is a little different from the previous example. In this example, the new table name, `report`, is followed by a list of column names and types in parentheses; this is necessary because we're not duplicating the structure of an existing table. Moreover, we actually change `name` to `category`. Then, the `SELECT` statement follows, with its output matching the new columns in the new table. You can check the contents of the new table to see the result:

```
mysql> SELECT * FROM report LIMIT 5;
```

```
+-----+-----+
| title          | category |
+-----+-----+
| AMADEUS HOLY    | Action   |
| AMERICAN CIRCUS | Action   |
| ANTITRUST TOMATOES | Action   |
| ARK RIDGEMONT    | Action   |
| BAREFOOT MANCHURIAN | Action   |
+-----+-----+
5 rows in set (0.00 sec)
```

So, in this example, the `title` and `name` values from the `SELECT` statement are used to populate the new `title` and `category` columns in the `report` table.

Creating tables with a query has a major caveat that you need to be careful about: it doesn't copy the indexes (or foreign keys, if you use them). This is a feature, since it gives you a lot of flexibility, but it can be a catch if you forget. Have a look at our `actor_2` example:

```
mysql> DESCRIBE actor_2;
```

```
+-----+-----+-----+-----+...
| Field      | Type       | Null | Key |...
+-----+-----+-----+-----+...
| actor_id    | smallint unsigned | NO   |      |...
```

```

| first_name | varchar(45)      | NO   |       | ...
| last_name  | varchar(45)      | NO   |       | ...
| last_update | timestamp       | NO   |       | ...
+-----+-----+-----+-----+
...+-----+-----+
...| Default          | Extra
...+-----+-----+
...| 0                |
...| NULL             |
...| NULL             |
...| CURRENT_TIMESTAMP | DEFAULT_GENERATED on update CL
...+-----+-----+
4 rows in set (0.00 sec)

```

`mysql> SHOW CREATE TABLE actor_2\G`

```

***** 1. row *****
Table: actor_2
Create Table: CREATE TABLE `actor_2` (
  `actor_id` smallint unsigned NOT NULL DEFAULT '0',
  `first_name` varchar(45) NOT NULL,
  `last_name` varchar(45) NOT NULL,
  `last_update` timestamp NOT NULL
    DEFAULT CURRENT_TIMESTAMP
    ON UPDATE CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4
  COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)

```

You can see that there's no primary key; the `idx_actor_last_name` key is missing as well, as is the `AUTO_INCREMENT` property of the `actor_id` column.

To copy indexes across to the new table, there are at least three things you can do. The first is to use the `LIKE` statement to create the empty table with the indexes, as described earlier, and then copy the data across using an `INSERT` with a `SELECT` statement, as described in [“Inserting Data Using Queries”](#).

The second thing you can do is to use `CREATE TABLE` with a `SELECT` statement and then add indexes using `ALTER TABLE` as described in

Chapter 4.

The third option is to use the `UNIQUE` (or `PRIMARY KEY` or `KEY`) keyword in combination with `CREATE TABLE` and `SELECT` to add a primary key index. Here's an example of this approach:

```
mysql> DROP TABLE actor_2;
```

```
Query OK, 0 rows affected (0.04 sec)
```

```
mysql> CREATE TABLE actor_2 (UNIQUE(actor_id))
-> AS SELECT * from actor;
```

```
Query OK, 200 rows affected (0.05 sec)
Records: 200  Duplicates: 0  Warnings: 0
```

```
mysql> DESCRIBE actor_2;
```

Field	Type	Null	Key	...
actor_id	smallint unsigned	NO	PRI	...
first_name	varchar(45)	NO		...
last_name	varchar(45)	NO		...
last_update	timestamp	NO		...

Default	Extra
0	
NULL	
NULL	
CURRENT_TIMESTAMP	DEFAULT_GENERATED on update CL

4 rows in set (0.01 sec)

The `UNIQUE` keyword is applied to the `actor_id` column, making it the primary key in the newly created table. The keywords `UNIQUE` and `PRIMARY KEY` can be interchanged.

You can use different modifiers when you're creating tables using these techniques. For example, here's a table created with defaults and other settings:

```
mysql> CREATE TABLE actor_3 (
    ->   actor_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
    ->   first_name VARCHAR(45) NOT NULL,
    ->   last_name VARCHAR(45) NOT NULL,
    ->   last_update TIMESTAMP NOT NULL
        DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_
    -> PRIMARY KEY (actor_id),
    -> KEY idx_actor_last_name (last_name)
    -> ) SELECT * FROM actor;
```

Query OK, 200 rows affected (0.05 sec)
Records: 200 Duplicates: 0 Warnings: 0

Here, we've set `NOT NULL` for the new columns, used the `AUTO_INCREMENT` feature on `actor_id`, and created two keys. Anything you can do in a regular `CREATE TABLE` statement can be done in this variant; just remember to add those indexes explicitly!

Performing Updates and Deletes with Multiple Tables

In [Chapter 3](#), we showed you how to update and delete data. In the examples there, each update and delete affected one table and used properties of that table to decide what to modify. This section shows you more complex updates and deletes. As you'll see, you can delete or update rows from more than one table in one statement, and you can use those or other tables to decide what rows to change.

Deletion

Imagine you're cleaning up a database, perhaps because you're running out of space. One way to solve this problem is to remove some data. For example, in the `sakila` database, it might make sense to remove films that are present in our inventory, but have never been rented. Unfortunately, this means you need to remove data from the `inventory` table using information from the `rental` table.

With the techniques we've described so far in the book, there's no way of doing this without creating a table that combines the two tables (perhaps using `INSERT` with `SELECT`), removing unwanted rows, and copying the data back to its source. This section shows you how you can perform this procedure and other more advanced types of deletion more elegantly.

Consider the query you need to write to find films in the `inventory` table that have never been rented. One way to do it is to use a nested query, employing the techniques we showed you in [Chapter 5](#), with the `NOT EXISTS` clause. Here's the query:

```
mysql> SELECT * FROM inventory WHERE NOT EXISTS
      -> (SELECT 1 FROM rental WHERE
      -> rental.inventory_id = inventory.inventory_id);
```

```
+-----+-----+-----+
| inventory_id | film_id | store_id | last_update
+-----+-----+-----+
|          5 |       1 |         2 | 2006-02-15 05:09:
+-----+-----+-----+
1 row in set (0.01 sec)
```

You can probably see how it works, but let's briefly discuss it anyway before we move on. You can see that this query uses a correlated subquery, where the current row being processed in the outer query is referenced by the subquery; you can tell this because the `inventory_id` column from `inventory` is referenced, but the `inventory` table isn't listed in the `FROM` clause of the subquery. The subquery produces output when there's a row in the `rental` table that matches the current row in the outer query (and so an `inventory`

entry was rented). However, since the query uses `NOT EXISTS`, the outer query doesn't produce output when this is the case, and so the overall result is that rows are output for inventory records of movies that haven't been rented.

Now let's take our query and turn it into a `DELETE` statement. Here it is:

```
mysql> DELETE FROM inventory WHERE NOT EXISTS
-> (SELECT 1 FROM rental WHERE
-> rental.inventory_id = inventory.inventory_id);
```

Query OK, 1 row affected (0.04 sec)

You can see that the subquery remains the same, but the outer `SELECT` query is replaced by a `DELETE` statement. Here, we're following the standard `DELETE` syntax: the keyword `DELETE` is followed by `FROM` and a specification of the table or tables from which rows should be removed, then a `WHERE` clause (and any other query clauses, such as `GROUP BY` or `HAVING`). In this query, rows are deleted from the `inventory` table, but in the `WHERE` clause a subquery is specified within a `NOT EXISTS` statement.

While this statement does indeed delete rows from one table based on data from another table, it's basically a variation of a regular `DELETE`. To convert this particular statement into a multitable `DELETE`, we should switch from a nested subquery to a `LEFT JOIN`, like so:

```
DELETE inventory FROM inventory LEFT JOIN rental
USING (inventory_id) WHERE rental.inventory_id IS NULL;
```

Note how the syntax changes to include the specific table (or tables) where we want to delete the rows we find. These tables are specified after `DELETE` but before the `FROM` and query specification. There's another way to write this query, however, and it's the one we prefer:

```
DELETE FROM inventory USING inventory
LEFT JOIN rental USING (inventory_id)
WHERE rental.inventory_id IS NULL;
```

This query is a mix of the previous two. We do not specify the deletion targets between `DELETE` and `FROM`, and write them down as if this were a regular deletion. Instead, we use a special `USING` clause, which indicates that a filter query (a join or otherwise) is going to follow. This is slightly clearer in our opinion than the previous example of `DELETE table FROM table`. One downside of using the `USING` keyword is that it can be mixed up with the `USING` keyword of a `JOIN` statement. With some practice, you'll never make that mistake, though.

Now that we know both multitable syntax variants, we can construct a query that actually requires a multitable delete. One example of a situation that could require such a statement is deleting records from tables that are involved in foreign key relationships. In the `sakila` database, there are records for films in the `film` table that have no associated records in the `inventory` table. That is, there are films that there's information on, but that cannot be rented. Suppose that as part of the database cleanup operation, we're tasked with removing such dangling data. Initially this seems easy enough:

```
mysql> DELETE FROM film WHERE NOT EXISTS
-> (SELECT 1 FROM inventory WHERE
-> film.film_id = inventory.film_id);
```

```
ERROR 1451 (23000): Cannot delete or update a parent rc
a foreign key constraint fails (
`sakila`.`film_actor`, CONSTRAINT `fk_film_actor_film`
FOREIGN KEY (`film_id`) REFERENCES `film` (`film_id`)
ON DELETE RESTRICT ON UPDATE CASCADE)
```

Alas, the integrity constraint prevents this deletion. We will have to remove not only the films, but also relationships between those films and actors. That may generate orphan actors, which can be deleted next. We could try to delete films and references to actors in one go, like this:

```
DELETE FROM film_actor, film USING
film JOIN film_actor USING (film_id)
LEFT JOIN inventory USING (film_id)
WHERE inventory.film_id IS NULL;
```

Unfortunately, even though the `film_actor` table is listed before the `film` table, the deletion from `film` still fails. It's not possible to tell the optimizer to process tables in a particular order. Even if this example were to have executed successfully, it's not a good practice to rely on such behavior as the optimizer may later change the table order unpredictably, causing failures.

This example highlights a difference between MySQL and the SQL standard: the standard specifies that the foreign keys are checked at transaction commit, whereas MySQL checks them immediately, preventing this statement from succeeding. Even if we were able to resolve this problem, films are also related to categories, so that will have to be taken care of too.

MySQL allows a few ways out of this situation. The first one is to execute a series of `DELETE` statements within one transaction (we talked more about transactions in Chapter 6):

```
mysql> BEGIN;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> DELETE FROM film_actor USING
      -> film JOIN film_actor USING (film_id)
      -> LEFT JOIN inventory USING (film_id)
      -> WHERE inventory.film_id IS NULL;
```

```
Query OK, 216 rows affected (0.01 sec)
```

```
mysql> DELETE FROM film_category USING
      -> film JOIN film_category USING (film_id)
      -> LEFT JOIN inventory USING (film_id)
      -> WHERE inventory.film_id IS NULL;
```

```
Query OK, 42 rows affected (0.00 sec)
```

```
mysql> DELETE FROM film USING
      -> film LEFT JOIN inventory USING (film_id)
      -> WHERE inventory.film_id IS NULL;
```

```
Query OK, 42 rows affected (0.00 sec)
```

```
mysql> ROLLBACK;
```

```
Query OK, 0 rows affected (0.02 sec)
```

You can see that we executed `ROLLBACK` instead of `COMMIT` to preserve the rows. In reality, you would of course use `COMMIT` to “save” the results of your operation.

The second option is dangerous. It is possible to suspend foreign key constraints by temporarily setting the `foreign_key_checks` system variable to `0` on the session level. We recommend against this practice, but it's the only way to delete from all three tables at the same time:

```
mysql> SET foreign_key_checks=0;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> BEGIN;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> DELETE FROM film, film_actor, film_category
      -> USING film JOIN film_actor USING (film_id)
      -> JOIN film_category USING (film_id)
      -> LEFT JOIN inventory USING (film_id)
      -> WHERE inventory.film_id IS NULL;
```

```
Query OK, 300 rows affected (0.03 sec)
```

```
mysql> ROLLBACK;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SET foreign_key_checks=1;
```

```
Query OK, 0 rows affected (0.00 sec)
```

While we don't recommend disabling foreign key checks, doing so allows us to show the power of multitable deletes. Here, with one query it was possible to achieve what took three queries in the previous example.

Let's break down this query. The tables from which rows will be deleted (if matched) are `film`, `film_actor`, and `film_category`. We specified them between the `DELETE FROM` and `USING` terms for clarity. `USING` starts our query, the filtering part of the `DELETE` statement. In this example, we have constructed a four-table join. We have joined `film`, `film_actor`, and `film_category` using `INNER JOIN`, as we need only matching rows. To the result of those joins, we `LEFT JOIN` the `inventory` table. In this context, using a left join is extremely important, because we are actually interested only in rows where `inventory` will have no records. We express that with `WHERE inventory.film_id IS NULL`. The result of this query is that we get all films not in `inventory`, then all film-actor relationships for those films, along with all category relationships for the films.

Is it possible to make this query safe to use with foreign keys? Not without breaking it down, unfortunately, but we can do better than having to run three queries:

```
mysql> BEGIN;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> DELETE FROM film_actor, film_category USING
   -> film JOIN film_actor USING (film_id)
   -> JOIN film_category USING (film_id)
   -> LEFT JOIN inventory USING (film_id)
   -> WHERE inventory.film_id IS NULL;
```

```
Query OK, 258 rows affected (0.02 sec)
```

```
mysql> DELETE FROM film USING
-> film LEFT JOIN inventory USING (film_id)
-> WHERE inventory.film_id IS NULL;
```

Query OK, 42 rows affected (0.01 sec)

```
mysql> ROLLBACK;
```

Query OK, 0 rows affected (0.01 sec)

What we've done here is combined deletion from the `film_actor` and `film_category` tables into a single `DELETE` statement, thus allowing deletion from `film` without any error. The difference from the previous example is that we `DELETE FROM` two tables instead of three.

Let's talk about the number of rows affected. In the first example, we deleted 42 rows from `film`, 42 rows from `film_category`, and 216 rows from the `film_actor` table. In the second example, our single `DELETE` query removed 300 rows. In the final example, we removed 258 rows combined from the `film_category` and `film_actor` tables, and 42 rows from the `film` table. You can probably guess by now that for a multitable delete MySQL will output the total number of rows deleted, without a breakdown into individual tables. This makes it harder to keep track of exactly how many rows were touched in each table.

Also, in multitable deletes you can't use the `ORDER BY` or `LIMIT` clauses.

Updates

Now we'll contrive an example using the `sakila` database to illustrate multiple-table updates. We've decided to change the ratings of all horror films to R, regardless of the original rating. To begin, let's display the horror films and their ratings:

```
mysql> SELECT name category, title, rating
-> FROM film JOIN film_category USING (film_id)
```

```
-> JOIN category USING (category_id)
-> WHERE name = 'Horror';
```

```
+-----+-----+-----+
| category | title           | rating |
+-----+-----+-----+
| Horror   | ACE GOLDFINGER    | G      |
| Horror   | AFFAIR PREJUDICE   | G      |
| Horror   | AIRPORT POLLOCK    | R      |
| Horror   | ALABAMA DEVIL      | PG-13  |
| ...      | ...                | ...    |
| Horror   | ZHIVAGO CORE       | NC-17  |
+-----+-----+-----+
56 rows in set (0.00 sec)
```

```
mysql> SELECT COUNT(title)
-> FROM film JOIN film_category USING (film_id)
-> JOIN category USING (category_id)
-> WHERE name = 'Horror' AND rating <> 'R';
```

```
+-----+
| COUNT(title) |
+-----+
|        42 |
+-----+
1 row in set (0.00 sec)
```

We don't know about you, but we'd love to see a G-rated horror film! Now, let's put that query into an `UPDATE` statement:

```
mysql> UPDATE film JOIN film_category USING (film_id)
-> JOIN category USING (category_id)
-> SET rating = 'R' WHERE category.name = 'Horror';
```

```
< ----- >
Query OK, 42 rows affected (0.01 sec)
Rows matched: 56  Changed: 42  Warnings: 0
```

Let's look at the syntax. A multiple-table update looks similar to a `SELECT` query. The `UPDATE` statement is followed by a list of tables that incorporates

whatever join clauses you need or prefer; in this example, we've used `JOIN` (remember, that's `INNER JOIN`) to bring together the `film` and `film_category` tables. This is followed by the keyword `SET`, with assignments to individual columns. Here you can see that only one column is modified (to change the ratings to R), so columns in all other tables besides `film` aren't modified. The following `WHERE` is optional, but is necessary in this example to only touch rows with the category name `Horror`.

Note how MySQL reports that 56 rows were matched, but only 42 updated. If you look at the results of the previous `SELECT` queries, you'll see that they show the counts of films in the Horror category (56), and films in that category with a rating other than R (42). Only 42 rows were updated because the other films already had that rating.

As with multitable deletes, there are some limitations on multitable updates:

- You can't use `ORDER BY`.
- You can't use `LIMIT`.
- You can't update a table that's read from in a nested subquery.

Other than that, multitable updates are much the same as single-table ones.

Replacing Data

You'll sometimes want to overwrite data. You can do this in two ways using the techniques we've shown previously:

- Delete an existing row using its primary key and then insert a replacement with the same primary key.
- Update a row using its primary key, replacing some or all of the values (except the primary key).

The `REPLACE` statement gives you a third, convenient way to change data. This section explains how it works.

The `REPLACE` statement is just like `INSERT`, but with one difference. You can't `INSERT` a new row if there is an existing row in the table with the same primary key. You can get around this problem with a `REPLACE` query, which

first removes any existing row with the same primary key and then inserts the new one.

Let's try an example, where we'll replace the row for the actress PENELOPE GUINNESS in the `sakila` database:

```
mysql> REPLACE INTO actor VALUES (1, 'Penelope', 'Guine
```

```
ERROR 1451 (23000): Cannot delete or update a parent rc  
a foreign key constraint fails (`sakila`.`film_actor`,  
CONSTRAINT `fk_film_actor_actor` FOREIGN KEY (`actor_ic  
REFERENCES `actor` (`actor_id`) ON DELETE RESTRICT ON L
```

Unfortunately, as you'll have guessed after reading the previous paragraph, `REPLACE` actually has to perform a `DELETE`. If your database is highly constrained referentially, like the `sakila` database is, the `REPLACE` will often not work. Let's not fight against the database here and instead use the `actor_2` table we created in "[Creating Tables with Queries](#)":

```
mysql> REPLACE actor_2 VALUES (1, 'Penelope', 'Guiness'
```

```
Query OK, 2 rows affected (0.00 sec)
```

You can see that MySQL reports that two rows were affected: first the old row was deleted, and then the new row was inserted. You can see that the change we made was minor—we just changed the case of the name—and therefore it could easily have been accomplished with an `UPDATE`. Because the tables in the `sakila` database are relatively small, it's difficult to construct an example in which `REPLACE` looks simpler than `UPDATE`.

You can use the different `INSERT` syntaxes with `REPLACE`, including using `SELECT` queries. Here are some examples:

```
mysql> REPLACE INTO actor_2 VALUES (1, 'Penelope', 'Guine
```

```
Query OK, 2 rows affected (0.00 sec)
```

```
mysql> REPLACE INTO actor_2 (actor_id, first_name, last_name)
-> VALUES (1, 'Penelope', 'Guiness');
```

```
Query OK, 2 rows affected (0.00 sec)
```

```
mysql> REPLACE actor_2 (actor_id, first_name, last_name)
-> VALUES (1, 'Penelope', 'Guiness');
```

```
Query OK, 2 rows affected (0.00 sec)
```

```
mysql> REPLACE actor_2 SET actor_id = 1,
-> first_name = 'Penelope', last_name = 'Guiness';
```

```
Query OK, 2 rows affected (0.00 sec)
```

The first variant is almost identical to our previous example, except it includes the optional `INTO` keyword (which, arguably, improves the readability of the statement). The second variant explicitly lists the column names that the matching values should be inserted into. The third variant is the same as the second, without the optional `INTO` keyword. The final variant uses the `SET` syntax; you can add the optional keyword `INTO` to this variant if you want. Note that if you don't specify a value for a column, it's set to its default value, just like for `INSERT`.

You can also bulk-replace into a table, removing and inserting more than one row. Here's an example:

```
mysql> REPLACE actor_2 (actor_id, first_name, last_name)
-> VALUES (2, 'Nick', 'Wahlberg'),
-> (3, 'Ed', 'Chase');
```

```
Query OK, 4 rows affected (0.00 sec)
Records: 2  Duplicates: 2  Warnings: 0
```

Note that four rows are affected: two deletions and two insertions. You can also see that two duplicates were found, meaning the replacement of existing rows succeeded. In contrast, if there isn't a matching row in a `REPLACE` statement, it acts just like an `INSERT`:

```
mysql> REPLACE actor_2 (actor_id, first_name, last_name
-> VALUES (1000, 'William', 'Dyer');
```

```
Query OK, 1 row affected (0.00 sec)
```

You can tell that only the insert occurred, since only one row was affected.

Replacing also works with a `SELECT` statement. Recall the `recommend` table from [“Inserting Data Using Queries”](#), at the beginning of this chapter. Suppose you've added 10 films to it, but you don't like the choice of the seventh film in the list. Here's how you can replace it with a random choice of another film:

```
mysql> REPLACE INTO recommend SELECT film_id, language_
-> release_year, title, length, 7 FROM film
-> ORDER BY RAND() LIMIT 1;
```

```
Query OK, 2 rows affected (0.00 sec)
Records: 1  Duplicates: 1  Warnings: 0
```

Again, the syntax is the same as with `INSERT`, but a deletion is attempted (and succeeds!) before the insertion. Note that we keep the value of the `sequence_id` as 7.

If a table doesn't have a primary key or another unique key, replacing doesn't make sense. This is because there's no way of uniquely identifying a matching row in order to delete it. When you use `REPLACE` on such a table, its behavior is identical to `INSERT`. Also, as with `INSERT`, you can't replace rows in a table that's used in a subquery. Finally, note the difference between

`INSERT IGNORE` and `REPLACE` : the first keeps the existing data with the duplicate key and does not insert the new row, while the second deletes the existing row and replaces it with the new one.

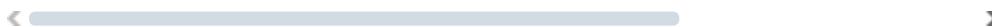
When specifying a list of columns for `REPLACE` , you have to list every column that does not have a default value. In our examples, we had to specify `actor_id` , `first_name` , and `last_name` , but we omitted the `last_update` column, which has a default value of `CURRENT_TIMESTAMP` .

WARNING

`REPLACE` is a powerful statement, but be careful when using it, as the results can be unexpected. Pay special attention when you have auto-increment columns and multiple unique keys defined.

MySQL provides another nonstandard extension of SQL: `INSERT ... ON DUPLICATE KEY UPDATE` . It is similar to `REPLACE` , but instead of `DELETE` followed by `INSERT` , it executes an `UPDATE` whenever a duplicate key is found. At the beginning of this section, we had an issue replacing a row in the `actor` table. MySQL refused to run a `REPLACE` , because deleting a row from the `actor` table would violate a foreign key constraint. It is, however, easily possible to achieve the desired result with the following statement:

```
mysql> INSERT INTO actor_3 (actor_id, first_name, last_
-> VALUES (1, 'Penelope', 'Guiness')
-> ON DUPLICATE KEY UPDATE first_name = 'Penelope',
```



Query OK, 2 rows affected (0.00 sec)

Note that we're using the `actor_3` table created in “[Creating Tables with Queries](#)”, as it has all the same constraints as the original `actor` table. The statement that we've just shown is very similar to `REPLACE` semantically, but has a few key differences. When you do not specify a value for a field in a `REPLACE` command, that field must have a `DEFAULT` value, and that default value will be set. That naturally follows from the fact that a completely new row is inserted. In the case of `INSERT ... ON DUPLICATE KEY`

UPDATE , we are updating an existing row, so it's not necessary to list every column. We can do that if we want, though:

```
mysql> INSERT INTO actor_3 VALUES (1, 'Penelope', 'Guir
      -> ON DUPLICATE KEY UPDATE
      -> actor_id = 1, first_name = 'Penelope',
      -> last_name = 'Guiness', last_update = NOW();
```

Query OK, 2 rows affected (0.01 sec)

To minimize the amount of typing necessary for this command and to allow inserting multiple rows, we can refer to the new field values in the UPDATE clause. Here's an example with multiple rows, one of which is new:

```
mysql> INSERT INTO actor_3 (actor_id, first_name, last_
      -> (1, 'Penelope', 'Guiness'), (2, 'Nick', 'Wahlber
      -> (3, 'Ed', 'Chase'), (1001, 'William', 'Dyer')
      -> ON DUPLICATE KEY UPDATE first_name = VALUES(first_
      -> last_name = VALUES(last_name);
```

Query OK, 5 rows affected (0.01 sec)

Records: 4 Duplicates: 2

Let's review this query in more detail. We're inserting four rows into the `actor_3` table, and by using `ON DUPLICATE KEY UPDATE` we're telling MySQL to run an update on any duplicate rows it finds. Unlike in our previous example, however, this time we don't set updated column values explicitly. Instead, we use the special `VALUES()` function to obtain the value of each column in the rows we passed to the `INSERT`. For example, for the second row, `2, Nick, Wahlberg`, `VALUES(first_name)` will return `Nick`. Notice that MySQL reports we've updated an odd number of rows: five. Whenever a new row is inserted, the number of affected rows is incremented by one. Whenever an old row is updated, the number of affected rows is incremented by two. Since we've already updated the record for `Penelope` by running the previous query, our new insert doesn't add anything new, and MySQL will skip the update as well. We are left with two

updates for duplicate rows, and insertion of a completely new row, or five rows affected in total.

TIP

In most situations, we recommend that you default to using `INSERT ... ON DUPLICATE KEY UPDATE` instead of `REPLACE`.

The EXPLAIN Statement

You'll sometimes find that MySQL doesn't run queries as quickly as you expect. For example, you'll often notice that a nested query runs slowly. You might also find—or, at least, suspect—that MySQL isn't doing what you hoped, because you know an index exists but the query still seems slow. You can diagnose and solve query optimization problems using the `EXPLAIN` statement.

Analyzing query plans, understanding optimizer decisions, and tuning query performance are advanced topics, and more art than science: there's no one way to do it. We are adding this section so that you know this capability exists, but we won't get too deep into this topic.

The `EXPLAIN` statement helps you learn about a `SELECT` or any other query. Specifically, it tells you how MySQL is going to do the job in terms of the indexes, keys, and steps it'll take if you ask it to resolve a query.

`EXPLAIN` does not actually execute a query (unless you ask it to) and in general doesn't take a lot of time to run.

Let's try a simple example that illustrates the idea:

```
mysql> EXPLAIN SELECT * FROM actor\G
```

```
***** 1. row *****  
    id: 1  
select_type: SIMPLE  
      table: actor  
     partitions: NULL  
       type: ALL  
possible_keys: NULL
```

```
    key: NULL
key_len: NULL
    ref: NULL
   rows: 200
filtered: 100.00
    Extra: NULL
1 row in set, 1 warning (0.00 sec)
```

The statement gives you lots of information. It tells you that:

- The `id` is 1, meaning this row in the output refers to the first (and only!) `SELECT` statement in this query. If we utilize a subquery, each `SELECT` statement will have a different `id` in the `EXPLAIN` output (although some subqueries will not result in multiple `id`s being reported, as MySQL might rewrite the query). We'll show an example with a subquery and different `id` values later.
- The `select_type` is `SIMPLE`, meaning it doesn't use a `UNION` or subqueries.
- The `table` that this row is referring to is `actor`.
- The `partitions` column is empty, because no tables are partitioned.
- The `type` of join is `ALL`, meaning all rows in the table are processed by this `SELECT` statement. This is often bad, but not in this case; we'll explain why later.
- The `possible_keys` that could be used are listed. In this case, no index will help find all rows in a table, so `NULL` is reported.
- The `key` that is actually used is listed, taken from the list of `possible_keys`. In this case, since no key is available, none is used.
- The `key_len` (key length) of the key MySQL plans to use is listed. Again, no key means a `NULL` `key_len` is reported.
- The `ref` (reference) columns or constants that are used with the key are listed. Again, there are none in this example.
- The `rows` that MySQL thinks it needs to process to get an answer are listed.
- The `filtered` column tells us the percentage of rows from the table that this stage will return: 100 means all rows will be returned. This is expected as we're asking for all rows.
- Any `Extra` information about the query resolution is listed. Here, there's none.

In summary, the output of `EXPLAIN SELECT * FROM actor` tells you that all rows from the `actor` table will be processed (there are 200 of them), and

no indexes will be used to resolve the query. This makes sense and is probably exactly what you expected would happen.

Note that every `EXPLAIN` statement reports a warning. Each query we send to MySQL gets rewritten before execution, and the warning message will contain the rewritten query. For example, `*` may be expanded to an explicit list of columns, or a subquery may be optimized implicitly into a `JOIN`.

Here's an example:

```
mysql> EXPLAIN SELECT * FROM actor WHERE actor_id IN
-> (SELECT actor_id FROM film_actor
-> WHERE film_id = 11);
```

id	select_type	table	partitions	type
1	SIMPLE	film_actor	NULL	ref
1	SIMPLE	actor	NULL	eq_ref
...	possible_keys		key	key_len
...	PRIMARY, idx_fk_film_id	idx_fk_film_id	2	
...	PRIMARY	PRIMARY		2
...				
...	ref		rows	filtered Ext
...	const		4	100.00 Usi
...	sakila.film_actor.actor_id		1	100.00 NUL
...				

2 rows in set, 1 warning (0.00 sec)

```
mysql> SHOW WARNINGS\G
```

```
***** 1. row *****
```

Level: Note

Code: 1003

```
Message: /* select#1 */ select
`sakila`.`actor`.`actor_id` AS `actor_id`,
`sakila`.`actor`.`first_name` AS `first_name`,
`sakila`.`actor`.`last_name` AS `last_name`,
`sakila`.`actor`.`last_update` AS `last_update`
from `sakila`.`film_actor` join `sakila`.`actor` where
((`sakila`.`actor`.`actor_id` = `sakila`.`film_actor`.`
and (`sakila`.`film_actor`.`film_id` = 11))
1 row in set (0.00 sec)
```

We mentioned that we would show an example with different `id` values and a subquery. Here's the query:

```
mysql> EXPLAIN SELECT * FROM actor WHERE actor_id IN
-> (SELECT actor_id FROM film_actor JOIN
-> film USING (film_id)
-> WHERE title = 'ZHIVAGO CORE');
```

id	select_type	table	partitions	type
1	SIMPLE	<subquery2>	NULL	ALL
1	SIMPLE	actor	NULL	ALL
2	MATERIALIZED	film	NULL	ref
2	MATERIALIZED	film_actor	NULL	ref

possible_keys	key	key_len
NULL	NULL	NULL
PRIMARY	NULL	NULL
PRIMARY, idx_title	idx_title	514
PRIMARY, idx_fk_film_id	idx_fk_film_id	2

rows	filtered	Extra
NULL	100.00	NULL
200	0.50	Using where; Using join buffer (

```

...|    1 | 100.00 | Using index
...|    5 | 100.00 | Using index
...
4 rows in set, 1 warning (0.01 sec)

```

In this example, you can see that `id 1` is used for the `actor` and `<subquery2>` tables, and `id 2` is used for `film` and `film_actor`. But what is `<subquery2>`? That's a virtual table name used here because the optimizer materialized the results of the subquery, or in other words stored them in a temporary table in memory. You can see that the query with an `id > 2` has a `select_type` of `MATERIALIZED`. The outside query (`id 1`) will look up the results of the inner query (`id 2`) from this temporary table. This is just one of many optimizations that MySQL can perform while executing complex queries.

Next, we'll give the `EXPLAIN` statement some real work to do. Let's ask it to explain an `INNER JOIN` between `actor`, `film_actor`, `film`, `film_category`, and `category`:

```

mysql> EXPLAIN SELECT first_name, last_name FROM actor
   -> JOIN film_actor USING (actor_id)
   -> JOIN film USING (film_id)
   -> JOIN film_category USING (film_id)
   -> JOIN category USING (category_id)
   -> WHERE category.name = 'Horror';

```

id	select_type	table	partitions	type
1	SIMPLE	category	NULL	ALL
1	SIMPLE	film_category	NULL	ref
1	SIMPLE	film	NULL	eq_ref
1	SIMPLE	film_actor	NULL	ref
1	SIMPLE	actor	NULL	eq_ref

possible_keys	key
PRIMARY	NULL
PRIMARY,fk_film_category_category	fk_film_categc
PRIMARY	PRIMARY
PRIMARY,IDX_fk_film_id	idx_fk_film_ic

```

... | PRIMARY          | PRIMARY
...+-----+-----+
...+-----+-----+-----+
...| ref           | rows | filtered | E
...+-----+-----+-----+
...| NULL          | 16   | 10.00   | L
...| sakila.category.category_id | 62   | 100.00  | L
...| sakila.film_category.film_id | 1    | 100.00  | L
...| sakila.film_category.film_id | 5    | 100.00  | L
...| sakila.film_actor.actor_id  | 1    | 100.00  | N
...+-----+-----+-----+
5 rows in set, 1 warning (0.00 sec)

```

Before we discuss the output, think about how the query could be evaluated.

MySQL could go through each row in the `actor` table, then match that with `film_actor`, then with `film`, `film_category`, and finally `category`. We have a filter on the `category` table, so in this imaginary case MySQL would only be able to match fewer rows once it gets to that table. That is a poor execution strategy. Can you think of a better one?

Let's now look at what MySQL actually decided to do. This time, there are five rows because there are five tables in the join. Let's run through this, focusing on those things that are different from the previous examples:

- The first row is similar to what we saw before. MySQL will read all 16 rows from the `category` table. This time, the value in the `Extra` column is `Using where`. That means a filter based on a `WHERE` clause is going to be applied. In this example, the `filtered` column shows 10, meaning that roughly 10% of the table rows will be produced by this stage for further operations. The MySQL optimizer expects 16 rows in the table and expects that one to two rows will be returned here.
- Now let's look at row 2. The join `type` for the `film_category` table is `ref`, meaning that all rows in the `film_category` table that match rows in the `category` table will be read. In practice, this means one or more rows from the `film_category` table will be read for each `category_id` from the `category` table. The `possible_keys` column shows both `PRIMARY` and `fk_film_category_category`, and the latter is chosen as the index. The primary key of the

`film_category` table has two columns, and the first one of them is `film_id`, making that index less optimal for filtering on `category_id`. The key used to search `film_category` has a `key_len` of 1 and is searched using the `sakila.category.category_id` value from the `category` table.

- Moving to the next row, we can see that the join `type` for the `film` table is `eq_ref`. This means that for each row we got from the previous stage (scanning `film_category`), we'll read exactly one row in this stage. MySQL can guarantee that because the index used to access the `film` table is `PRIMARY`. In general, if a `UNIQUE NOT NULL` index is used, `eq_ref` is possible. This is one of the best join strategies.

The two nested rows in the output do not show us anything new. In the end, we see that MySQL selected an optimal execution plan. Usually, the fewer rows that are read during the first step of the execution, the faster the query will be.

MySQL 8.0 introduced a new format of `EXPLAIN PLAN` output, which is available through the `EXPLAIN ANALYZE` statement. While it may be somewhat easier to read, the caveat here is that the statement actually has to be executed, unlike with the regular `EXPLAIN`. We won't go into the details of this new format, but we'll show an example here:

```
mysql> EXPLAIN ANALYZE SELECT first_name, last_name
   -> FROM actor JOIN film_actor USING (actor_id)
   -> JOIN film USING (film_id)
   -> WHERE title = 'ZHIVAGO CORE'\G
```

```
***** 1. row *****
EXPLAIN:
-> Nested loop inner join
  (cost=3.07 rows=5)
  (actual time=0.036..0.055 rows=6 loops=1)
-> Nested loop inner join
  (cost=1.15 rows=5)
  (actual time=0.028..0.034 rows=6 loops=1)
-> Index lookup on film
  using idx_title (title='ZHIVAGO CORE')
  (cost=0.35 rows=1)
  (actual time=0.017..0.018 rows=1 loops=1)
-> Index lookup on film_actor
```

```
using idx_fk_film_id (film_id=film.film_id)
(cost=0.80 rows=5)
(actual time=0.010..0.015 rows=6 loops=1)
-> Single-row index lookup on actor
    using PRIMARY (actor_id=film_actor.actor_id)
(cost=0.27 rows=1)
(actual time=0.003..0.003 rows=1 loops=6)

1 row in set (0.00 sec)
```

This output is even more advanced than the regular `EXPLAIN` output, as it gives more data. We'll leave analyzing it as an exercise for the reader. You should be able to figure it out based on our explanation for the regular `EXPLAIN` output.

Alternative Storage Engines

One of the features of MySQL that distinguishes it from many other RDBMSs is its support for different storage engines. The mechanism of MySQL's support of multiple engines is complicated, and to explain it properly we'd need to go into more depth on its architecture and implementation than we have space for here. We can, however, try to give you a bird's-eye overview of what engines are available, why you might want to use a nondefault engine, and why having this choice is important.

Instead of saying *storage engine*, which sounds complicated, we could say *table type*. In very simplified terms, MySQL allows you to create tables of different types, with each type giving those tables distinct properties. There's no universally good table type, as each storage engine has pros and cons.

In the book so far, we've used only the default InnoDB table type. The reason is simple: almost everything you're likely to want from a modern database can be achieved using InnoDB. It's generally fast, reliable, and a proven and well-supported engine, and is widely considered (including by us) to provide the best balance of pros and cons. We've seen this engine used successfully by applications requiring very high throughput of short queries, and also by data warehouse applications that run few but "large" queries.

At the time of writing, the official MySQL documentation documents 8 additional storage engines, and 18 additional engines are documented for

MariaDB. In reality, there are even more storage engines available, but not all of them make it into a major MySQL flavor's documentation. Here we'll only describe those engines we find useful and that are at least somewhat commonly used. It may well be that the storage engine that best fits your use case is not one we describe. Take no offense; there are just too many of them to cover them all fairly.

Before we dive into our overview of different engines, let's briefly look at why this matters. The pluggable nature of storage engines in MySQL and the ability to create tables with different types is important because it allows you to unify your database access layer. Instead of using multiple database products, each with its own driver, query language, configuration, management, backups, and so on, you can just use MySQL and achieve different behaviors by changing table types. Your apps may not even need to know what types tables have. That said, it's not all that simple and rosy. You may not be able to use all of the backup solutions we'll explain in [Chapter 10](#). You will also need to understand the trade-offs each engine provides. However, we still think that it's better to have this ability to change table types than not.

We'll start our review by defining broad categories based on important properties of the different storage engines. One of the most important divisions is the ability of the engine to support transactions (you can read more about transactions, locking, and why all of this is important in [Chapter 6](#)).

Currently available transactional engines include the default InnoDB, the actively developed MyRocks, and the deprecated TokuDB. All of the different engines available across major MySQL flavors; only these three support transactions. Every other engine is nontransactional.

The next broad division we can perform is based on crash safety, or the ability of the engine to guarantee the durability property of the ACID set of properties. If a table uses a crash-safe engine, then we can expect every bit of data a committed transaction has written to be available after an unclean instance restart. Crash-safe engines include the already mentioned InnoDB, MyRocks, and TokuDB, as well as the Aria engine. None of the other available engines guarantees crash safety.

We could come up with more examples of how to group the table types, but let's get to actually describing some of the engines and their properties. First things first, let's see how to actually view the list of engines available. To achieve that, we use the special `SHOW ENGINES` command. Here's its output on a default MySQL 8.0.23 Linux installation:

```
mysql> SHOW ENGINES;
```

Engine	Support	...
ARCHIVE	YES	...
BLACKHOLE	YES	...
MRG_MYISAM	YES	...
FEDERATED	NO	...
MyISAM	YES	...
PERFORMANCE_SCHEMA	YES	...
InnoDB	DEFAULT	...
MEMORY	YES	...
CSV	YES	...
...
Comment		
Archive storage engine		
/dev/null storage engine (anything you write to it)		
Collection of identical MyISAM tables		
Federated MySQL storage engine		
MyISAM storage engine		
Performance Schema		
Supports transactions, row-level locking, and foreign keys		
Hash based, stored in memory, useful for temporary tables		
CSV storage engine		
...
Transactions	XA	Savepoints
...
NO	NO	NO
NO	NO	NO
NO	NO	NO
NULL	NULL	NULL
NO	NO	NO
NO	NO	NO
YES	YES	YES

```
... | NO      | NO      | NO      |
... | NO      | NO      | NO      |
...+-----+-----+-----+
9 rows in set (0.00 sec)
```

You can see that MySQL conveniently tells us whether an engine supports transactions. The `XA` column is for distributed transactions—we won’t be covering these in this book. Savepoints are basically the ability to create mini-transactions within transactions, another advanced topic. As an exercise, consider executing `SHOW ENGINES;` in MariaDB and Percona Server installations.

InnoDB

Before we move on to “alternative” storage engines, let’s discuss the default one: InnoDB. InnoDB is reliable, performant, and full-featured. Pretty much everything you’d expect from a modern RDBMS is achievable in some way with InnoDB. In this book, we never change the engine of a table, so every example uses InnoDB. While you are learning MySQL, we recommend that you stick with this engine. It’s important to understand its downsides, but unless they become problematic for you, there’s almost no reason not to use it all the time.

The InnoDB table type includes the following features:

Support for transactions

This is discussed in detail in [Chapter 6](#).

Advanced crash recovery features

The InnoDB table type uses logs, which are files that contain a record of the actions that MySQL has taken to change the database. Logs enable MySQL to recover effectively from power losses, crashes, and other basic database failures. Of course, nothing can help you recover from the loss of a machine, failure of a disk drive, or other catastrophic failures. For these, you need offsite backups and new hardware. Every backup tool we explore in [Chapter 10](#) works with InnoDB.

Row-level locking

Unlike the previous default engine, MyISAM (which we'll explore in the following section), InnoDB provides fine-grained locking infrastructure. The lowest level of locking is row-level, meaning that an individual row can be locked by a running query or transaction. This is important for most write-heavy online transaction processing (OLTP) applications; if you're locking at a higher level, like the table level, you can end up with too many concurrency issues.

Foreign key support

InnoDB is currently the only MySQL table type that supports foreign keys. If you are building a system that requires a high level of data safety enforced by referential constraints, InnoDB is your only choice.

Encryption support

InnoDB tables can be encrypted transparently by MySQL.

Partitioning support

InnoDB supports *partitioning*; that is, spreading of data physically between multiple data files based on some rules. This allows InnoDB to work with tables of tremendous size efficiently.

That's a lot of pros, but there are also a few cons:

Complexity

InnoDB is relatively complex. This means that there's a lot to configure and understand. Out of almost a thousand server options in MySQL, more than two hundred are specific to InnoDB. This downside is, however, far outweighed by the benefits this engine provides.

Data footprint

InnoDB is a relatively disk-hungry storage engine, making it less appealing for storing extremely large datasets.

Scaling with database size

InnoDB shines when the so-called “hot” dataset, or frequently accessed data, is present in its buffer pool. This limits its scalability.

MyISAM and Aria

MyISAM was the default storage engine in MySQL for a long time, and a staple of this database. It is simple in use and design, is quite performant, and has low overhead. So why did it stop being the default? There are actually several good reasons for this, as you'll see when we discuss its limitations.

Nowadays, we do not recommend using MyISAM unless it's required for legacy reasons. You may read on the internet that its performance is better than InnoDB's. Unfortunately, most of that information is very old and hasn't aged well—today, that's simply not the case in the vast majority of cases. One reason for this is the changes to the Linux kernel necessitated by the Spectre and Meltdown security vulnerabilities in January 2018, which resulted in a performance decrease of up to 90% for MyISAM.

Until MySQL 8.0, MyISAM was used in MySQL for all data dictionary objects. Starting with that version, the data dictionary is now fully InnoDB, to support advanced features like atomic DDL.

Aria is a reworked MyISAM provided in MariaDB. Apart from promising better performance and being improved and worked on continuously, the most important feature of Aria is its crash safety. MyISAM, unlike InnoDB, does not guarantee data safety when your write succeeds, which is a major drawback of this storage engine. Aria, on the other hand, allows the creation of durable tables, supported by a global transaction log. In the future Aria may also support full-fledged transactions, but this is not the case at the time of writing.

The MyISAM table type includes the following features:

Table-level locking

Unlike InnoDB, MyISAM only supports locks at the high level of whole tables. This is much simpler and less nuanced than row-level locking and has lower overhead and memory requirements. However, a major drawback becomes apparent with highly concurrent, write-heavy workloads: even if each session would update or insert a separate row, they will each execute in turn. Reads in MyISAM can coexist simultaneously, but they will block concurrent writes. Writes also block reads.

Partitioning support

Until MySQL 8.0, MyISAM supported partitioning. In MySQL 8.0 this is no longer the case, and to achieve this one must resort to using different storage engines (Merge or MRG_MyISAM).

Compression

It's possible to create read-only compressed tables with the `myisampack` utility, which are quite a bit smaller than the equivalent InnoDB tables without compression. Since InnoDB supports compression, however, we recommend you first check whether this option will give you better results.

The MyISAM type has the following limitations:

Crash safety and recovery

MyISAM tables are not crash-safe. MySQL does not guarantee that when a write succeeds, the data actually reaches files on the disk. If MySQL doesn't exit cleanly, MyISAM tables may get corrupted, require repairs, and lose data.

Transactions

MyISAM does not support transactions. Thus, MyISAM only provides atomicity for each individual statement, which may not be enough in your case.

Encryption

MyISAM tables do not support encryption.

MyRocks and TokuDB

One of the most significant problems with InnoDB is its relative difficulty in dealing with large datasets. We've mentioned that it is desirable to have your frequently accessed data in memory, but that is not always possible to achieve. Moreover, when data sizes go into multiterabyte territory, InnoDB's on-disk performance suffers, too. The objects in InnoDB also have quite a large overhead in terms of size. In recent years, a few different projects have appeared that attempt to fix issues inherent to InnoDB's basic data structure the B-tree by basing the storage engine on a different data structure. These

include MyRocks, based on the LSM-tree, and TokuDB, based on a proprietary fractal tree data structure.

NOTE

We wanted to mention TokuDB in this section for completeness, but its developer, Percona, has deprecated this storage engine, and its future is unclear. TokuDB has similar properties to MyRocks, and in fact MyRocks is the preferable migration path off of TokuDB.

How data structures affect the properties of storage engines is a complex topic, arguably falling outside the scope of database administration and operation. We try to keep things reasonably simple in this book, so we won't go into that particular topic. You should also remember what we wrote earlier about InnoDB: that default is not unreasonable, and more often than not, just using InnoDB is going to give you the best set of trade-offs. It continues to be our recommendation that you use InnoDB while learning MySQL, and beyond that, but we also feel that we should cover the alternatives.

The MyRocks table type includes the following features:

Support for transactions

MyRocks is a transactional storage engine, supporting regular transactions and distributed transactions. Savepoints are not fully supported.

Advanced crash recovery features

MyRocks relies on internal log files called WAL files (for “write-ahead log”) to provide crash recovery guarantees. You can expect everything that was committed to be present once the database is restarted after a crash.

Encryption support

MyRocks tables can be encrypted.

Partitioning support

MyRocks tables can be partitioned.

Data compression and compactness

The storage footprint of MyRocks tables is usually lower than that of InnoDB tables. There are two properties leading to that: it uses a more compact storage structure and data within that storage structure can be compressed. While compression is not unique to MyRocks, and InnoDB in fact provides compression options, MyRocks consistently shows better results.

Consistent write performance at scale

This one is difficult to properly explain without going deep into the weeds. However, the minimal version is that the write performance of MyRocks is almost unaffected by the volume of the data. In the real world, this means that MyRocks tables show worse performance than InnoDB tables until the size of the data becomes much larger than memory. What happens then is that InnoDB's performance decreases faster than MyRocks', eventually falling behind.

The MyRocks table type has the following limitations:

Transactions and locking

MyRocks doesn't support the `SERIALIZABLE` isolation level or gap locking, described in [Chapter 6](#).

Foreign keys

Only InnoDB supports foreign key constraints.

Performance tradeoffs

MyRocks does not cope well with read-heavy and analytical workloads. InnoDB provides better generalized performance.

Complexity

We mentioned that InnoDB is more complex than MyISAM. However, in some respects MyRocks is more complex than InnoDB. It is not well documented, is being actively developed (so is less stable), and can be difficult to operate.

General availability

MyRocks is not available in Community or Enterprise MySQL; to use it, you need to use another version of MySQL, like MariaDB or Percona Server. That may result in operational difficulties. Packaged versions lag behind development, and to use all of the current features, a dedicated MySQL server has to be built with MyRocks sources.

Other Table Types

We've covered all the major table types, but there are a few more that we will summarize briefly. Some of these storage engines are rarely used, and they may have documentation issues and bugs.

Memory

Tables of this type are stored entirely in memory and are never persisted on disk. The obvious advantage is performance: memory is many times faster than disk and will probably always be. The disadvantage is that the data is lost as soon as MySQL is restarted or crashes. Memory tables are usually used as temporary tables. Apart from that, memory tables can be used to hold small-sized, frequently accessed hot data, such as a dictionary of sorts.

Archive

This type provides a way to store data in a highly compressed and append-only manner. You cannot modify or delete data in tables using the Archive storage engine. As its name suggests, it's mostly useful for long-term storage of data. In reality, it's rarely used, and it has a few issues with primary key and auto-increment handling. InnoDB with compressed tables and MyRocks may provide better alternatives.

CSV

This storage engine stores tables on disk in CSV format. That allows you to view and manipulate such tables with spreadsheet applications or just text editors. It's not often used, but can be an alternative approach to what we explained in "["Loading Data from Comma-Delimited Files"](#)", and also can be used for data export.

Federated

This type provides a way to query data in remote MySQL systems. Federated tables do not contain any data, only some metadata related to connection details. This is an interesting way of getting or modifying remote data without setting up replication. Compared to just connecting to remote MySQL, it has the benefit of simultaneously providing access to local and remote tables.

Blackhole

This storage engine discards every bit of data that would be stored within its tables. In other words, whatever is written into a Blackhole table is immediately lost. That doesn't sound terribly useful, but there are use cases for this engine. Usually it's used to filter replication through an intermediate server, where unneeded tables are blackholed. Another potential use case is to get rid of a table in a closed-source application: you can't just drop the table, as that'll break the app, but by making it Blackhole you remove any processing and storage overhead.

These storage engines are pretty exotic and are rarely seen in the wild. However, you should know they exist, as you may never know when something might become useful.