MySQL\_for\_Python\_Albert\_c06

Inserting Multiple Entries

When we deal with large amounts of data that are all going into the same database, running single instances of individual INSERT commands can take a ridiculous amount of time and waste a considerable amount of I/O. What we need is a way to insert multiple items in one go.

In this chapter, we will look at the following:

* How iteration can help us execute several individual INSERT statements rapidly
* Using executemany() to submit several INSERT statements in one go
* When not to use executemany()
* Throttling how much data is inserted at a time

At the end of the chapter, we will put these lessons to use in writing a conversion program that inserts the contents of a comma-separated value (CSV) file into MySQL.

**The problem**

You need to collate and rearrange the contents of several databases into one table whenever a given indicator is achieved (the indicator may be, among other things, a stock price, a value in one of the databases, or the flight pattern of African swallows). The format and length of the tables are predictable. There are 5,000 records in each table so manually running a separate INSERT statement for each record is not a viable option even with the programs used in earlier chapters. The problem calls for a means of iterating through a series and changing the INSERT statement automatically. Using what we have covered in previous chapters, we could pursue one of the following two ways to do this:

* Write a MySQL script to insert the data in batch mode
* Iterate over the data to form and execute a MySQL INSERT statement accordinglyInserting Multiple Entries

None of these are a very good solution to the present problem.

**Why not a MySQL script?**

As we have seen when we created the world and Sakila databases, a MySQL script can contain the schema of a database, the values of the database tables, or both. To create data quickly, there is nothing better. However, following are the several drawbacks to using a script in this scenario:

* Lack of automation
* Debugging the process
* Inefficient I/O

**Lack of automation**

Barring the use of an automation daemon (for example, cron) to run a cron job or a similar scheduled task, a DBA or their designate would have to run the script. This unnecessarily consumes time. It is comparable to swapping tape backups when automated backup services are available and proven.

Most modern computing systems support automatic task scheduling. On Unix-based systems like Linux and Mac OS X, one can schedule processes to be run at set times and on a regular rotation. One of the most widely used programs for such scheduling is cron. A single scheduled task in cron has thus come to be known as a cron job.

**Debugging the process**

Creating a MySQL script can be a very tedious task, and the slightest error or oversight can ruin the entire process. Using the --force flag causes MySQL to ignore errors in the source file. It is therefore not something that should be used regularly if one values the integrity of data. If the script is malformed for any reason, a two minute data insertion job can quickly become a two hour (at least!), unscheduled debugging process.

**Inefficient I/O**

Dumping large amounts of data on MySQL can create latency across the network. If the script were run by a DBA or similar, that person should rightly evaluate the state of the network before running it. Regardless of experience and training, judgment calls naturally require estimation and can result in ambiguous decision-making. While this is unavoidable, it should be minimized where possible.

If the server is experiencing high traffic, the DBA would need to find something else to do and reschedule the running of the script. This randomly postpones the time of execution and the availability of the results. Also, it runs the risk of the DBA forgetting to execute the script.

If the script is automated with a cron job or similar, we risk dumping a lot of data onto MySQL at a time when others are doing more time sensitive tasks. On most servers, we can background the process, so it does not adversely impact the client processes. This, however, only ensures that the process will be run. It does not guarantee that the process will be finished by a particular time.

**Why not iterate?**

Every time a program iterates to read or insert data, it creates a certain amount of I/O processing. Depending on how a program is written will determine how much I/O is included in each loop.

**A test sample: Generating primes**

To illustrate this, consider a program that accepts a series of numbers, generates the prime numbers that are equal to or less than each of those numbers and inserts those numbers into a database called primes with a table of the same name. The table has the following description:

mysql> describe primes;

+--------+--------------+------+-----+---------+----------------+

| Field

| Type

| Null | Key | Default | Extra

|

+--------+--------------+------+-----+---------+----------------+

| ID

| int(11)| NO| PRI | NULL| auto\_increment |

| NUMBER | int(11)| NO|| 0|

|

| PRIMES | varchar(300) | NO|| 0|

|

+--------+--------------+------+-----+---------+----------------+

and can be created with the following statement:

CREATE TABLE `primes` (`ID` int(11) NOT NULL auto\_increment, `NUMBER`

int(11) NOT NULL default '0', `PRIMES` varchar(300) NOT NULL default

'0', PRIMARY KEY (`ID`)) ENGINE=MyISAM DEFAULT CHARSET=latin1;

[ 163 ]Inserting Multiple Entries

Using the sys module to handle user input, the string module to split the series of entries, and MySQLdb to handle the database connection, we get the following preamble:

#!/usr/bin/env python

import MySQLdb, string, sys

Next, we need a function that will generate the prime numbers to a certain limit called n.

def primes(n):

"""Returns a list of prime numbers up to n using an algorithm

based on

the Sieve of Eratosthenes."""

if n < 2: return ['A number less than two is not prime by

definition.']

else:

s = range(3,n+1,2)

maxfactor = n \*\* 0.5

half = (n+1)/2-1

i = 0

m = 3

while m <= maxfactor:

if s[i]:

j = (m\*m-3)/2

s[j] = 0

while j < half:

s[j] = 0

j += m

i = i + 1

m = 2 \* i + 3

return str([2] + [x for x in s if x])

This algorithm is based on the Sieve of Eratosthenes, one of the simplest ways of generating prime numbers. It uses the following steps:

1. Generate a list of integers from 2 to n.

2. Discount the multiples of each number that remains and that has a square less than or equal to n; this leaves all the prime factors of n.

3. Stop when the square of the number in the series is greater than n.

Prime numbers by definition have no other factor, but themselves and one. The lowest prime number is therefore 2. For this reason, we check whether n is less than 2 and return a message accordingly.

For this program, we want a string returned so we convert the results before we return them.

For more on the Sieve of Eratosthenes and how it works, see the entry on Wikipedia:

http://en.wikipedia.org/wiki/Sieve\_of\_Eratosthenes

The previous algorithm can be found in many forms and in many languages on the Internet. Two of the best that informed this discussion are as follows:

http://www.maths.abdn.ac.uk/~igc/tch/mx3015/notes/

node79.html

http://code.activestate.com/recipes/366178/

Next, we create a function to form and execute the INSERT statement.

def insert(n, p, cur):

statement = """INSERT INTO primes(number, primes) VALUES("%s",

"%s")""" %(n\

, p)

cur.execute(statement)

return statement

This function takes the number, the primes, and the cursor object handle as arguments.

Finally, our main() function looks like this:

def main():

numbers = sys.argv[1]

iterations = numbers.split(',')

for n in iterations:

mydb = MySQLdb.connect(host = 'localhost',

user = 'skipper',

passwd = 'secret',

db = 'primes')

cur = mydb.cursor()

n = int(n)

try:

p = primes(n)

if p.isalpha():

raise ValueError

else:

statement = insert(n, p, cur)

print "Data entered with the following statement:\n",

statement

except:

raise

[ 165 ]Inserting Multiple Entries

We split the values passed by the user and iterate through them with a for loop. Note that we include the database connection and cursor object creation as part of the iteration.

We then include the usual if clause to call main():

if \_\_name\_\_ == '\_\_main\_\_':

main()

**Comparing execution speeds**

We can test the speed of this program by giving it a series of simple primes to generate—even of the same number. Here we call it with a series of sevens to process and with the Unix command time to measure its speed.

time ./primes.py "7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,

7, 7, 7, 7, 7, 7, 7"

Given this series of sevens, we get a real execution time of 0.175 seconds for the previously discussed implementation.

If we rearrange the program by moving the mydb and cur assignment lines to follow immediately after the preamble (remembering to adjust the indentation), we get an execution speed of 0.138 seconds. This difference in speed (0.037 seconds) is not particularly significant on a local system with a single user, but it would be magnified synergistically on a server with hundreds or thousands of users.

So, we see that, if the connection and cursor is created each time, the program will be less efficient than if the loop contained only the execution of the statement and the connection and cursor were persistent in the cycle of the program. However, even just passing a series of single statements to MySQL through Cursor.execute() will consume more in protocol than necessary. Excessive iteration consumes resources needlessly.

**Introducing the executemany() method**

As seen in previous chapters, MySQLdb provides multiple ways to retrieve data such as: fetchall(), fetchmany(), and fetchone(). These allow us to use different strategies for processing data according to how many records we have to process and the environment in which we are working. MySQLdb also provides more than one way to insert information.

[ 166 ]Chapter 6

Up to now, we have used Cursor.execute() whenever we needed to insert data. As shown previously, however, executing single statements in iteration greatly increases the processing requirements of the program. For this reason, MySQL for Python provides another method for Cursor objects—executemany().

To be sure, executemany() is effectively the same as simple iteration. However, it is typically faster. It provides an optimized means of affecting INSERT and REPLACE across multiple rows.

**executemany(): Basic syntax**

The executemany() call has the following basic syntax:

<cursor object handle>.executemany(<statement>, <arguments>)

Note that neither the statement to be processed nor its arguments is optional. Both are required; otherwise, a TypeError will be thrown by Python.

The data type of each argument to executemany() should be noted as:

* statement: A string containing the query to execute
* arguments: A sequence containing parameters to use within the statement

Using any other type for statements or arguments will result in Python throwing an error.

The statement may be constructed using Python's string manipulation methods. The sequence reflected in the arguments must be either of a sequence or mapping type. Thus, the following data types are allowed as arguments:

* strings
* lists
* tuples
* dictionaries

If a comma-delimited list is inadvertently passed as a string, it will be evaluated as a string and will not be separated. Consequently, if one is passing values for multiple columns, one needs to use one of the other three types listed. As seen in the following project, one can construct a string of one's values and then convert them into a tuple for processing.

[ 167 ]Inserting Multiple Entries

**executemany(): Multiple INSERT statements**

In Chapter 2, Simple Querying we inserted records individually using execute(). Using that same method with iteration would look like this:

#!/usr/bin/env python

import MySQLdb, string, sys

mydb = MySQLdb.connect(host = 'localhost',

user = 'skipper',

passwd = 'secret',

db = 'fish')

cursor = mydb.cursor()

data = [

("bass", 6.75),

("catfish", 5),

("haddock", 6.50),

("salmon", 9.50)

]

for item in data:

cursor.execute(

"""INSERT INTO menu(name, price) VALUES("%s", %s)"""

%(item[0], item[1]))

print "Finished!"

This would take care of inserting four records with a single run of the program. But it needs to pass the statements individually through Python to do so. Using executemany(), we could easily have inserted the same data with a single call:

#!/usr/bin/env python

import MySQLdb, string, sys

mydb = MySQLdb.connect(host = 'localhost',

user = 'skipper',

passwd = 'secret',

db = 'fish')

cursor = mydb.cursor()

[ 168 ]Chapter 6

cursor.executemany(

"""INSERT INTO menu(name, price) VALUES (%s, %s)""",

[

("bass", 6.75),

("catfish", 5),

("haddock", 6.50),

("salmon", 9.50)

] )

print "Finished!"

You may notice previously that we used strings and integers in the first iterative example, but immediately a list of tuples in the second one. The executemany() function is more tolerant of mixed data types than the execute() method. As a consequence, we can get away with less precise type handling in the second where we had to explicitly handle the separate parts of item in the first. The executemany() method can handle multiple data types in one call where execute() does not.

Just as with Cursor.execute(), executemany() supports MySQL's UPDATE statements, as well.

In either event, the results are the same:

mysql> select \* from menu;

+----+---------+-------+

| ID | NAME

| PRICE |

+----+---------+-------+

|1 | bass|6.75 |

|2 | catfish |5.00 |

|3 | haddock |6.50 |

|4 | salmon9.50 |

|

+----+---------+-------+

But the more efficient way is to use executemany().

**executemany(): multiple SELECT statements**

The executemany() method may be used not only to insert or update data, but may also be used to run multiple queries. As with the INSERT statement, the value here is in having an optimized iteration that affects the same basic query. If the statements that need to be processed are disparate in structure, then one needs to default to Cursor.execute().

To process several SELECT queries in succession, simply format the argument to executemany() accordingly. Consider the following code snippet that uses the Sakila database that we used in the last chapter:

mydb = MySQLdb.connect('localhost', 'skipper', 'secret', 'sakila')

cur = mydb.cursor()

results = cur.executemany("SELECT title FROM film WHERE rating = %s",

('R', 'NC-17'))

If we then print the value of results, we get an output of 405. Like execute(), Cursor.executemany() does not return the actual results of the query in a Python program. It returns the number of affected records as a value that can be assigned to a variable. Even though the search is executed and the results returned from MySQL, MySQL for Python does not assume you want the affected records automatically. The results are returned in aggregate. That is, the output of 405 is the sum of all affected records. The value in this knows the total number of records to be returned before one processes them. Given the restricted context, executemany() tends to be used more frequently for INSERT statements than for SELECT.

It is worth noting that this example is a bit artificial. You wouldn't normally rely on the number of affected rows returned through MySQLdb. Rather, we would use MySQL's built-in COUNT() function. For more on this, see Chapter 10, Aggregate Clauses and Functions.

**executemany(): Behind the scenes**

As seen previously, it is not necessary to have an intimate knowledge of the entire process behind executemany() in order to use it. But to truly know what you are doing with it, you must follow the Python motto for learning:

Use the source, Luke!

In that vein, let's take a look at the underlying code of executemany(). The definition line indicates that executemany() requires the SQL query to be processed as well as the arguments for that statement.

def executemany(self, query, args):

As noted under executemany(): Basic syntax section, previously, both the statement and the arguments are mandatory. executemany() expects you to feed it a template for query and the values to be inserted for args.

del self.messages[:]

db = self.\_get\_db()

if not args: return

We will skip the docstring as you can access it through help(cursor.executemany) in a Python shell. The method starts by deleting the contents of the messages attribute.

More on the del statement can be found at:

http://docs.python.org/reference/simple\_stmts.

html#the-del-statement

It then checks whether the cursor is still open by assigning to db the connection of the object. As mentioned previously, args must be declared. Here, if there are no arguments passed to the method, the method returns effectively empty-handed.

charset = db.character\_set\_name()

if isinstance(query, unicode): query = query.encode(charset)

Note that all queries must be strings and must be encoded for the character set used by the connection.

MySQLdb here checks for the character set of the database and encodes the query accordingly. isinstance is a built-in type checker and is here used to verify that query is unicode.

m = insert\_values.search(query)

if not m:

r = 0

for a in args:

r = r + self.execute(query, a)

return rThe value of insert\_values is defined on line 9

of the module as a regular expression:

insert\_values = re.compile(r"\svalues\s\*(\(((?<!\\)'.\*?\).\*(?<!\\)?'|.

)+?\))", re.IGNORECASE)

[ 171 ]Inserting Multiple Entries

So, MySQLdb checks whether any values are available in query. If there are none, it calls execute() for every argument that is passed to it, adding the number of affected lines to the number of would-be hits and returns the total. Otherwise, it goes on.

p = m.start(1)

e = m.end(1)

qv = m.group(1)

m is a pattern object and therefore provides for regular expression methods. Here p is the index for the start of the elements matched. e is the end index and typically represents the length of the match. qv is the entire group of results, if such exist.

Next, MySQL for Python appends to the value of qv, the values passed by MySQL.

try:

q = [ qv % db.literal(a) for a in args ]

qv is a string, so the percentage sign here serves as a string format operator to insert into the value of qv whatever is returned by the db.literal() method.

The literal() method of the Connections class simply nuances the arguments accordingly. If only one argument is passed to executemany(), Connections. literal() returns it as a single object. If more than one argument is passed, then they are converted into a sequence.

except TypeError, msg:

if msg.args[0] in ("not enough arguments for format

string",

"not all arguments converted"):

self.messages.append((ProgrammingError, msg.args[0]))

self.errorhandler(self, ProgrammingError, msg.args[0])

else:

self.messages.append((TypeError, msg))

self.errorhandler(self, TypeError, msg)

except:

from sys import exc\_info

exc, value, tb = exc\_info()

del tb

self.errorhandler(self, exc, value)

Then several types of errors are handled accordingly. Finally, executemany() prepares the number of affected records.

r = self.\_query('\n'.join([query[:p], ',\n'.join(q),

query[e:]]))

if not self.\_defer\_warnings: self.\_warning\_check()

return r

After checking for any warnings, it then returns a long integer of the number of rows affected.

**MySQL server has gone away**

Using executemany() can give a speed boost to your program. However, it has its limits. If you are inserting a large amount of data, you will need to throttle back how much you process in each call or be faced with an error message similar to this one:

MySQL server has gone away.

You will get this if you are lucky. Before this, you are likely to get another message:

Lost Connection to MySQL server during query.

Both of these messages indicate that the MySQL server is overwhelmed with your data. This is because MySQL's maximum default packet size is 1 MB and the maximum data that the server will receive per packet is 1 MB.

If you encounter these error messages, you have two choices: Decrease the amount of data you are passing to MySQL in each packet or increase MySQL's tolerance for data. If you have system administrator access, the latter option can be affected by changing the max\_allowed\_packet parameter either in the command-line arguments with which the server is started or in a MySQL configuration file.

**Command-line option configuration**

To change the value of max\_allowed\_packet on the command-line, adapt the following example:

$> mysqld --max\_allowed\_packet=16M

This will change the value to 16 megabytes until the MySQL server is halted and restarted without the flagged value.

It is important to note that the 16 megabytes are not held in reserve. MySQL does not use more memory than is necessary for any transaction. The present value only serves to cap how much memory MySQL may try to use.

If you want to make it permanent and do not want to change the configuration file for some reason, you will need to change the initialization script of your system. On Unix variants (Unix, Linux, and Mac OS X), this will be in the directory /etc/init. d. On Windows systems, this file could be located in different places depending on the configuration of the system, but the Windows System directory is common.

**Using a configuration file**

Adapting MySQL's configuration file is the simplest way to affect a permanent change in the value of max\_allowed\_packet. The location of the configuration file depends on your platform:

Unix/Linux: /etc/my.cnf

Windows: C:\WINDOWS\my.cnf or C:\my.cnf or C:\my.ini

Also, Windows installations may use the my.cnf or my.ini files in the installation directory of MySQL.

After locating the configuration file for your system, copy it to an archive. For example, on a Linux system, one would use the following:

#\etc> cp my.cnf my.cnf.original

The exact naming conventions available for the new filename will depend on your platform. Once that is done, you are ready to open the configuration file in your favorite text editor.

With the file opened and having system permission to write to it, locate the mysqld section in the file:

[mysqld]

One of the first values, if not the first value, in this section should be max\_allowed\_ packet. If it is there, change the value appropriately and if it's not, amend the beginning of the mysqld section to look like this:

[mysqld]

max\_allowed\_packet=16M

After saving the file with the affected changes, restart the MySQL server.

**More than 16 MB is often unnecessary**

It is worth noting that in the preceding examples for the command-line configuration and configuration file options, we changed the value to 16 megabytes. More than this is typically unnecessary without making further configuration changes to the MySQL client.

The default packet size limit of the MySQL client is 16 megabytes. That is therefore the largest packet that the MySQL client will send by default. If you try to pass a larger packet through the client, an error message of Packet too large will be passed. Therefore, if you want to handle particularly large files (for example, binary files), you need to change the value of max\_packet\_size for both the server and the client.

**Project: Converting a CSV file to a MySQL table**

The CSV file is a very common way to share tabular information across several platforms and programs. Every mature spreadsheet application supports the format. Therefore, various sectors of enterprise rely on it daily.

As the name implies, CSV files contain comma-separated values. These values are often enclosed in quotation marks. This tends to make CSV a predictable format for data and makes for easy development.

But for all its uses, the CSV format is very slow for searching and for collating data. Relational databases (RDBMS) are much more efficient. Therefore, it is useful to be able to import data from a CSV file into MySQL. Unfortunately, MySQL does not do this natively and not without considerable trouble.

For examples of what some MySQL users do to import CSV file content into MySQL, see the comments on the mysqlimport client: <http://dev>. mysql.com/doc/refman/5.4/en/mysqlimport.html

Further, if one tries to use mysqlimport for files larger than 1 MB in size, one encounters errors.

Consequently, we need a program to import CSV files with the following functionality:

* The file is comma-separated (that is, not tab, space, or semi-colon)
* The file is written using the Excel dialect of CSV
* The user will indicate at runtime the filename as well as the database, table and column headings for the INSERT command
* The program will evaluate the file's size on disk and adapt its insertion strategy to fit within MySQL's defaults

**The preamble**

Aside from the obvious requirement of MySQLdb, we will need two other modules to be imported into the program immediately. To parse user options, we will again use optparse. Then, to evaluate the file's size, we will use the os module.

The preamble therefore looks like this:

#!/usr/bin/env python

import MySQLdb

import optparse

import os

**The options**

The user options are then processed with the following code:

# Get options

opt = optparse.OptionParser()

opt.add\_option("-d", "--database", action="store", type="string",

help="name of the local database", dest="database")

opt.add\_option("-t", "--table", action="store", type="string",

help="table in the indicated database", dest="table")

opt.add\_option("-f", "--file", action="store", type="string",

help="file to be processed", dest="file")

opt.add\_option("-F", "--Fields", action="store", type="string",

help="Fields of file to be processed", dest="Fields")

opt, args = opt.parse\_args()

As in previous chapters, we will pass the attributes of opt to simpler variable names.

database = opt.database

table = opt.table

file = opt.file

We should not assume that the user will know to pass the field headings without extra spaces nor can we rely on the user to avoid that mistake. Therefore, we need to strip the extraneous spaces out of the field headings as they are passed to the list fields. As MySQL will not allow spaces in columnar headings without the use of backticks, we should surround the field names with backticks by default.

fields = opt.Fields.split(',')

for i in xrange(0, len(fields)):

fields[i] = "`" + fields[i].strip() + "`"

We now have stored all of the user input.

**Defining the connection**

To create the connection object and the cursor necessary for data insertion, we again use a function. As mentioned in previous projects concerning this function, one needs to be careful about the user being used in the connection.

def connection(database):

"""Creates a database connection and returns the cursor. Host is

hardwired to 'localhost'."""

try:

mydb = MySQLdb.connect(host = 'localhost',

user = 'skipper',

passwd = 'secret',

db = database)

cur = mydb.cursor()

return cur

except MySQLdb.Error:

print "There was a problem in connecting to the database.

Please ensure that the database exists on the local host system."

raise MySQLdb.Error

except MySQLdb.Warning:

pass

Of course, if the previous project programs are on the same system as this one, you can simply import the other program and call this function from there.

**Creating convert**

The second function that we need for this program will take care of the actual file conversion.

def convert(file):

"""Processes contents of file and returns a reader object, an

iterative object that returns a dictionary for each record."""

import csv

filehandle = open(file)

sheet = csv.DictReader(filehandle, fields)

return sheet

We will pass the function a filename file. It returns to us an iterative object sheet. The object is an iterator that returns a dictionary for each record in turn.

There is an alternative way of processing this file for conversion. Here we use csv. DictReader(), which returns a dictionary when its next() method is called. But we could use csv.reader(), which returns a string instead. The latter requires only the filename where the former requires both the file handle and the fields to be declared.

The basic syntax for each call is as follows:

csv.DictReader():

<variable> = csv.DictReader(<filehandle>, <field names>)

csv.reader():

<variable> = csv.reader(<filehandle>)

**The main() function**

Having defined functions for the database connection and for the file conversion, we next need to call them and then process the data they return. In addition to the two functions already mentioned, we need to get the file's size on disk using os.path.getsize().

def main():

"""The main function creates the MySQL statement in accordance

with the user's input and using the connection(), convert(), and

os.path.getsize()."""

cursor = connection(database)

data = convert(file)

filesize = os.path.getsize(file)

Next, we create two variables. The first, values, is to hold the data to be inserted into the MySQL statement. The second, r, is a counter that is incremented every time we process a line.

values = []

r = 0

We then need to walk through the data by record using a for loop.

for a in data:

if r == 0:

columns = ','.join(fields)

Within the for loop, we use the value of r to gauge our location in the file. If r is 0, we are at the first record, the column headers. Here we have opted to use the user's field names. However, you could easily adapt this to use the table's field names.

If r is not 0, we have a record with actual data and need to process it.

else:

value = ""

for column\_no in xrange(0, len(fields)):

if column\_no == 0:

value = "'" + a[fields[column\_no]]

else:

value = value + "', '" + a[fields[column\_no]]

value = value + "'"

Here we use the length of fields to determine how long the nested for loop will run. For each iteration, we use the ordinal value of the column to walk through the names of the fields as submitted by the user. We then use those values as the key by which the dictionary values of a are accessed. These are then appended to the value of value. If the field is the first one, column\_no is 0 and the field contents are added without a comma being included. At the end, value needs a quote to be added.

If r is greater than 0, we are processing actual data, not column headings. The way we process the data depends on the size of the file. If the file is over 1 Mb in size, we do not want to use executemany() and risk getting an error. Instead, for this project we will process each line with a single INSERT statement.

if r > 0:

if filesize <= 1000000:

value = eval(value)

values.append(value)

else:

query = """INSERT INTO %s (%s) VALUES""" %(table,

columns)

statement = query + "(" + value +")"

cursor.execute(statement)

While a few different definitions of 1 Mb abound, we here use the definition for 1 Mb on disk, which also happens to be the most conservative definition.

The actual size of 1 megabyte depends on context. Here we use the International System of Units definition which is also used by most manufacturers of hard drives and flash drives. For computer RAM, 1 Mb is 1048576 bytes (220 or 10242). More on the size of a megabyte may be found at http://en.wikipedia.org/wiki/Megabyte#Definition

If filesize is under 1 Mb, we process value as a string and convert it into a tuple using Python's built-in eval() function. We then append the tuple as a single item of the list values.

It is worth noting that eval() comes with several security drawbacks in the real world. Frequently, it is best not to use it because Python evaluates the value in the context in about the same terms as if you entered it into the Python shell. Therefore, if you used it, you would need to protect against someone passing commands to the operating system through

your Python program. The following is an example of bad data that could be passed to eval that would destroy a Linux system: """import\_('os').system('rm -rf /')""". This would result in a recursive deletion of the entire file tree. For the purposes of this program, we need to convert the strings to tuples. The eval() function is used here for the sake of simplicity.

If filesize is greater than 1 Mb, we form a simple INSERT statement from the data. We then use a common Cursor.execute() statement to process it.

r += 1

Finally, the last part of the for loop is to increment the counter r by 1. Without this, the evaluations in the rest of the loop will not give us the results we want.

Next, if the file size is less than 1 Mb, we need to form the INSERT statement template for executemany() and, with the data values, pass it to executemany() for execution. When we are done, we should give the user feedback about how many records have been affected.

if filesize <= 1000000:

query = "INSERT INTO " + table + "(" + columns + ") VALUES(%s"

for i in xrange(0, len(fields)-1):

query = query + ", %s"

query = query + ")"

query = str(query)

affected = cursor.executemany(query, values)

print affected, "rows affected."

else:

print r, "rows affected."

[ 180 ]Chapter 6

Within the if loop, we create a string query that contains the basic template of the INSERT statement. We append to this value a string formatting %s for every field beyond the first one that the data includes. Finally, we finish the formation of query with a closing parenthesis.

At this point, values is a list of tuples and query is a string. We then pass the two to cursor.executemany() and capture the returned value in affected.

Regardless of file size, the user should always get a statement of the number of rows affected. So, we close the if loop with a statement to that affect. As r is the number of times that the simple INSERT statement would have been executed in the last if loop of the preceding for loop, we can use it to indicate the rows affected as part of the else clause.

This last statement is more reflective of the number of iterations we have had of the for loop than of the number of times that cursor.execute() has been successfully called. If one wanted to be more precise, one could instead introduce a counter to which the value returned by cursor.execute() is added on each successful iteration.

**Calling main()**

Finally, we need to call main(). Once again, we use the usual if loop:

if \_\_name\_\_ == '\_\_main\_\_':

main()

**Room to grow**

The project is now finished and can take a basic CSV file and insert it into MySQL. As you use it, you will, without any doubt, see ways in which it could be better tailored to your needs. Some of these that are left as an exercise are:

* Handling tab-delimited files and files that don't speak the Excel dialect of CSV (for example, files that don't put cell contents in quotation marks)
* Dynamically evaluating the data format of the file and adjusting its processing accordingly
* Processing Excel (\*.xls) binary files (for this use the xlrd module from http://pypi.python.org/pypi/xlrd)
* Creating a table in the given database with columns that fit the data in the CSV file (for help with this, see Chapter 7, Creating and Dropping)

**Summary**

In this chapter, we have covered how to retrieve different sets of data using MySQL for Python. We have seen:

* How to use iteration to execute several individual INSERT statements rapidly
* Cursor.executemany() is optimized for such iteration
* When one should avoid using executemany()
* Why certain errors will arise when using executemany()
* How to throttle the amount of data we INSERT and why

In the next chapter, we will look at creating and dropping tables from within Python.