MySQL\_for\_Python\_Albert\_c05

Results Record-by-Record

As seen earlier, MySQL's SELECT statement can be very greedy. Using fetchall() processes all hits in one go. However, as we will see in this chapter, there are times when you should avoid this.

In this chapter, we will look at the following:

* Certain circumstances under which record-by-record processing is desirable
* How iteration helps us get each record in turn
* Using custom-created Python iterators for record-by-record processing
* Using fetchone() to process one record at a time
* Using fetchmany() to retrieve data in small chunks
* How to use loops to walk through a series of records

At the end of the chapter, we will put these lessons to use for writing a program that works with returned data record-by-record.

**The problem**

You have set up your database program as a daemon, so it runs persistently and can be called by users throughout the network. Based on what we have covered in earlier chapters, it retrieves all the records into the application process itself before processing each one as a list. You have debugged it thoroughly and so you know it works. Still, within a week of deployment, the service is jamming up and colleagues are drawing comparisons between your program's processing time and the speed of molasses in January. What happened?

To understand the problem, it is worth revisiting the process by which a query is processed in MySQLdb. After making the connection, one passes an execute() call according to the cursor object—cursor.execute() for an instance of MySQLdb. Cursor that is named cursor. When that execute() call is passed, MySQLdb does not return the results, but returns the number of rows affected by the query (the results are retrieved, but not immediately returned). Consider the code for the execute() method, part of the docstring and the exception-handling clauses have been removed for this snippet and replaced with an ellipsis().

def execute(self, query, args=None):

"""Execute a query.

query -- string, query to execute on server

args -- optional sequence or mapping, parameters to use with

query.

...

Returns long integer rows affected, if any

"""

from types import ListType, TupleType

from sys import exc\_info

del self.messages[:]

db = self.\_get\_db()

charset = db.character\_set\_name()

if isinstance(query, unicode):

query = query.encode(charset)

if args is not None:

query = query % db.literal(args)

try:

r = self.\_query(query)

...

self.\_executed = query

...

return r

To get the results, we have previously used fetchall(). For most applications, this is tantamount to cracking a nut with a sledgehammer. Whenever you use fetchall(), MySQLdb returns all affected rows to your program in one go. This is typically unnecessary for user interaction. Further, being used simultaneously by multiple users, the multiple instances of the application then consume system resources like a sinkhole, causing the server to slow or, worse, to crash.

It is worth noting that MySQL for Python supports a rowcount attribute of cursor and this is the preferred way of accessing the total of affected rows. One simply references the attribute as follows:

rows = cursor.rowcount

The problem would be resolved by regulating the rate at which one draws down the information to the user. This results in greater user satisfaction.

**Why?**

There are a number of reasons why one might opt to process records individually or at least in smaller chunks. Three of the most compelling reasons are:

* Limits of computing resources
* Network latency
* Pareto's principle

From the perspective of efficiency, any one of these reasons is good enough to warrant retrieval of smaller amounts of data. We look at each in greater detail in the following section.

**Computing resources**

Despite Moore's law holding true for many years now, it is coming under increasing pressure as a trend and cannot be presumed to hold true indefinitely. While one may have a budget with numbers so big that it looks like a phone directory, one's resources are only faster or more powerful on a comparative scale.

Moore's law states that the number of transistors that one is able to fit on an integrated circuit will double every two years. This law is not much of a natural law as a business observation was made by Gordon Moore, co-founder of Intel. He did not state in such absolute terms, but wrote of the trend in the 19 April 1965 edition of Electronics magazine (Volume 38, Number 8):

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase.

Moore has since gone on record saying that the trend cannot continue forever, but can be expected to break within the next twenty years. His paper is available for download on the following Intel website: ftp://download.intel.com/museum/Moores\_Law/

Click on Articles-Press\_Releases and then on Gordon\_Moore\_ 1965\_Article.pdf

It therefore behooves one as the developer or project manager to continue prudent use of one's resources. Especially, there are two conditions where this applies, which we will look at now.

**Local resources**

In the largest economies of the world, the majority of businesses have a relatively small operating budget. In the United Kingdom, for example, 97 percent of businesses employ fewer than 20 employees. In the United States, this figure is 89 percent. Australian small businesses account for 96 percent of the corporate economic system. The dynamics are the same across Canada, New Zealand, and the rest of the English-speaking world and are not likely to change significantly in other countries. Most applications, therefore, will be used by small businesses that typically have restricted budgets for computing resources.

In such a limited computing environment, performing and returning an exhaustive search can slow the system for other purposes. This applies to a single host or a local area network. One must keep in mind that most small businesses will run one MySQL server with several databases on it. Therefore, even if the application is not using the same database, it can overload the server, resulting in lag or latency, and loss of productivity.

It is worth noting that querying in large blocks to the point of locking up a system is a mistake that has been made even by professionals. One is reminded of earlier versions of Filemaker Pro—a database application for the Mac OS used widely in small business situations. When more than one large-ish query was passed to the server at one time, it became overloaded. It would then change the cursor to a teacup as if to tell the user, "Come back later. It might be a while."

The problem of resource scarcity is compounded by even good service administration. By their nature, the policies of system and network administrators tend to be at odds with the business cycle and therefore require constant revision and maintenance. Best practice says to design minimal resources than necessary for a task or for a project. However, the project requirement needs of a growing business will repeatedly push the resources toward breaking point. We thus need to make our software sufficiently robust to handle the tension, which ensues from that pushing. For Filemaker Pro, the system was pushed to its limit and therefore brought up a teacup.

This problem of anticipating how the user is going to push the system to its limits is as old as software engineering. Many modern methods of software development are designed to liaise with the end users throughout the development process; examples include evolutionary process systems (incremental and concurrent process models), component-based systems, and iterative models such as agile development. The point of this is not only to ensure a good fit for the software to the user's problems but, as part of that fit, to employ metrics in order to see how the system will be used most heavily in the target environment, and to ensure a reasonable amount of scalability.

**Web applications**

Perhaps the most common illustration of limited resources currently is the web application. Whether your site serves up a simple web page or a full service Web 2.0 application, you cannot escape the fundamental limits of the CPU to perform a given number of flops per second.

Combined with other system dynamics such as input and output (I/O), cache coherence, memory management, and interprocessor communication, the capabilities of the server are finite and measurable. This mandates a cap on the amount throughout from disk to port and subsequently limits the number of users that any service can support.

The common, non-technical view is that servers can process requests from multiple users at one time. This is erroneous. Servers process requests quickly using a first-in first-out (FIFO) queue system. Therefore, the greater the number of users requesting a page, the longer the time between the processing of the first request and the second for each user. If that delay becomes sufficiently long, the server and the client will lose contact with each other. Because HTTP persistent connections do not require the client to issue an end-of-request signal, the server will interpret this lack of communication as an end of request and close the connection. The user will then receive a message that the connection has timed out. Refreshing the page then opens a new connection and starts the process again.

Examples of such events abound. Perhaps the most common form is the so-called Slashdot effect, named after the Slashdot website, but the same dynamics are at play whenever a site gets more traffic than it is built to withstand. Barring a misconfiguration of either the server or the networks used along the way, a time out arises from the server becoming overwhelmed and unable to maintain communication with user clients. Of the variables located on the host machine, the primary reason for the server becoming overwhelmed is inefficiency in the way

dynamic content is generated.

**Network latency**

Beyond problems on the server itself, retrieving results by individual records is also advisable under certain network conditions. One needs to compare only the response times of a web application at times of peak and trough, to see how network latency impacts on the apparent responsiveness of an application. This latency is not always on any one network. Rather it is more often than not a negative synergy arising from small amounts of latency in networks on the route, from the browser to the server.

**Server-client communications**

The latency that occurs, impacts on the client-server communication and the timing of the TCP/IP dialogue that normally occurs. As this is disrupted, client requests can be dropped and it consequently appears to the server as if the client has closed the connection. The result is the same as the time out mentioned previously.

**Apparent responsiveness**

Network latency—whether on a LAN, WAN, or external network like the Internet, necessarily impinges upon the apparent responsiveness of the server and any application being served to the user. Even if it does not cause a timing out of the connection, latency slows down the entire communication process. The result is sluggishness in the application.

With the increasing ubiquity of AJAX calls, this sluggishness will become even more pronounced. A case in point; users of mainstream search engines or fully-featured web suites such as 2easy Office will be used to suggestions in completing forms. These suggestion lists are usually generated through an AJAX-like call to the server. Such calls necessarily increase the amount of traffic between server and client. When combined with other user traffic at a given time, the user's receipt of the system's recommendations can be delayed increasingly.

**Pareto's Principle**

Pareto's Principle is a commonly applied rule of thumb: 80 percent of anything is trivial and that 20 percent is critical. It has been further extended to say that 20 percent of any process produces 80 percent of the output. The observation has been demonstrated in order to apply the rule from its origins in Italian land ownership to the solving of business problems to combat trends in trench warfare in the Second World War. It also applies, with some qualification, to user queries in an application.

Vilfredo Pareto was an Italian microeconomist who did most of his work in the 19th century. One of his most famous contributions is the observation that 80% of the land in his native Italy was owned by 20% of the population. This observation was later noted by Joseph Juran, an American management consultant, who observed that 80% of a business problem results from 20% of the apparent causes.

The Pareto Principle is alternatively known as the 80/20 rule or the law of the vital few. The point is that the major part of any situation, process, or thing results from and is largely controlled by a very significant minority. The percentages involved sometimes change depending on the milieu and can be 70/30 (search engine indexing), 90/10 (worldwide health

expenditure), or even 95/5 (fundraising for charities).

Assuming the query is formed well by the user, the majority of user queries that are not exhaustive in nature will be resolved by a minority of the available data. Therefore, unless the query mandates an exhaustive query, one would do well to avoid being greedy in results to be returned. As the main overhead in most computing processes is I/O, it is best to let MySQL and MySQLdb handle filtering. Otherwise, you are left to return all results to your process and mill through them. This has its place, as we shall see, but it is not usually the best route to the

desired results.

**How?**

As we have seen, record-by-record retrieval can save a lot of overhead. To retrieve a data piecemeal using MySQL for Python, one can call one of two methods of the Cursor object: fetchone() or fetchmany().

**The fetchone() method**

The fetchone() method of a cursor object returns exactly one row of results. If the query affects no rows, None is returned. Consider the following code:

import MySQLdb

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

user = 'skipper',

passwd = 'secret',

db = 'fish')

cur = mydb.cursor()

statement = "SELECT \* FROM menu WHERE name='shark'"

cur.execute(statement)

result = cur.fetchone()

print result

[ 135 ]Results Record-by-Record

The outcome will be a raw form of the first record that matches the query.

(11L, 'shark', Decimal('13.00')

Note that only the first result will be printed using the preceding code. As the query leaves us vulnerable if there is more than one result, this is undesirable. If we are only going to process one record, we should nuance the SELECT statement, so that as few matches as possible will be returned. The more criteria one applies to the query, the fewer the records that will be returned.

For example, let's say we had a database of students at a university. Retrieving all records from a database of students will necessarily return a very large set of records. Retrieving those of a particular discipline will return fewer. Retrieving those who studied that discipline and another, denominated one will return still fewer. Limiting the query by year of graduation or year of birth will return even fewer. Even with limiting the query, we are likely to get more than one result. In such circumstances, one will find the second method of the Cursor class to be helpful.

**The fetchmany() method**

The fetchmany() method returns blocks of results according to a set limit. Where fetchone() was simply a method of the cursor object, fetchmany() requires the desired number of records to be passed as an argument. The basic syntax of the call is as follows:

<variable name> = <cursor name>.fetchmany(<number of records to

retrieve>)

An example of its use in reducing how much data is returned is as follows. Our SQL statement is very greedy, but the fetchmany() method keeps the results manageable.

#!/usr/bin/env python

import MySQLdb

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

user = 'skipper',

passwd = 'secret',

db = 'world')

cur = mydb.cursor()

statement = "SELECT \* FROM City"

cur.execute(statement)

results = cur.fetchmany(10)

for result in results:

print result

[ 136 ]Chapter 5

The output of this program is as follows. Recall that we are leaving the data raw for the moment.

(1L, 'Kabul', 'AFG', 'Kabol', 1780000L)

(2L, 'Qandahar', 'AFG', 'Qandahar', 237500L)

(3L, 'Herat', 'AFG', 'Herat', 186800L)

(4L, 'Mazar-e-Sharif', 'AFG', 'Balkh', 127800L)

(5L, 'Amsterdam', 'NLD', 'Noord-Holland', 731200L)

(6L, 'Rotterdam', 'NLD', 'Zuid-Holland', 593321L)

(7L, 'Haag', 'NLD', 'Zuid-Holland', 440900L)

(8L, 'Utrecht', 'NLD', 'Utrecht', 234323L)

(9L, 'Eindhoven', 'NLD', 'Noord-Brabant', 201843L)

(10L, 'Tilburg', 'NLD', 'Noord-Brabant', 193238L)

This code returns only the first ten rows and then exits. We can return more with another fetchmany() call. Assuming a cursor cur for an established database connection and the importing of the time module, we can run:

cur.execute("SHOW TABLES")

time.sleep(20)

print cur.fetchmany(10)

time.sleep(20)

print cur.fetchmany(10)

Using time.sleep(), we suspend execution of the program for a few seconds between retrievals. This does not shut the MySQL server down, but plays nicely with other processes that might issue a query, thus avoiding the problem in FileMaker Pro that was mentioned earlier.

Naturally, in the real world, you would normally want to work through all of the data. To do that, we need some iteration.

**Iteration: What is it?**

Unless we know that our fetchone() call by design will return a single record or that our fetchmany() call will return all results, it is necessary to retrieve the next record or set of records through an iterative cycle. How we implement that, however, depends on the conditions of our programming.

In its simplest form, iteration is the repetition of a process in order to progress through a series. The series may be data to be processed (the returned results of a database query) or a series of events to be performed (the calls necessary to retrieve records individually).

As we shall see, Python allows for the creation of iterative loops as well as iterator objects. Which one you use and when, naturally depends on the other dynamics of your application.

**Generating loops**

Iterating through results is done with one of two control flow tools: while or for.

**while...if loops**

The use of while...if for generating and controlling iteration is a combination of the two control-flow tools used elsewhere in generic Python. One initiates a recurring retrieval cycle that continues while there is valid data to process, but is broken if there is no data to process. If there is no data returned, fetchone() returns a value of None. This is therefore the value for which the if statement checks.

In using while-based controls, one does not always use if to check the data. Rather, as we have seen in previous chapters, one can simply allow an error to arise and thus break out of the loop.

Using the previous example of fetchone(), we can use the following nested loop to walk through the records:

import MySQLdb

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

user = 'skipper',

passwd = 'secret',

db = 'fish')

cur = mydb.cursor()

statement = "SELECT \* FROM menu WHERE name='shark'"

cur.execute(statement)

while 1:

result = cur.fetchone()

if result == None:

break

print result

Running the preceding code produced the following raw results:

(11L, 'shark', Decimal('13.00'))

As seen in earlier chapters, you can make this more useful to the user with a bit more code.

It is important to note that while creates the recurring retrieval of data that is checked by the if statement. Without while, only attempt at retrieval is made and the results only printed if not equal to None. Without the if statement, an infinite loop is created.

**The for loop**

Recall from the beginning of the chapter that the execute() method of the Cursor class returns the number of records affected by the query. That number is also stored in the rowcount attribute of the cursor object. Using that attribute, we can create a for loop to walk through the results individually:

import MySQLdb

import time

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

user = 'skipper',

passwd = 'secret',

db = 'fish')

cur = mydb.cursor()

statement = "SELECT \* FROM menu WHERE name='tuna'"

cur.execute(statement)

numrows = int(cursor.rowcount)

for x in xrange(0,numrows):

row = cursor.fetchone()

print row

time.sleep(5)

If one is seeking to optimize one's memory usage, use xrange() instead of range() for large series or for series accessed infrequently. Using range() causes Python to store the entire series at once, where xrange() creates series items on demand. You can learn more about xrange() in the Python documentation at <http://docs>. python.org/library/functions.html.

The output of this program will be the same as the previously discussed while loop.

As indicated previously, one uses a for loop to walk through the return of fetchmany(). One could rewrite the last code from just after the execute() call with the following:

numtakes = int(cur.rowcount)/5

for x in range(0,numtakes+1):

result = cur.fetchmany(5)

for row in result:

print row

[ 139 ]Results Record-by-Record

Unless you know that the number of rows being returned is divisible by the size you give fetchmany(), you need to account for the remainder. To accomplish this, we add one to the iteration count.

**Iterators**

Since Version 2.2, Python has supported the use of iterators. Iteration in its simplest form can be seen whenever one works through a list one-by-one, usually with a for loop. For example, assume a list of objects:

alist = ['chihuahua', 'boxer', 'greyhound']

We iterate through the list with a for loop:

for i in alist:

print i

The results are the list printed in order. Wherever and whenever you use a for loop to work through a series, you employ iteration (for example, lists by item, files by line, and so on).

As the name implies, iterators are objects used to iterate over a set of values. By definition, an iterator is an object that has a next method to return the next item in a series.

More information on iterators is available in the Python documentation by following the links at: http://docs.python.org/glossary.html#term-iterator

One creates an iterator using the iter() function. One passes the set to iter() as an argument.

iterator = iter(<name of set>)

This creates an iterator object that is normally named with an assignment statement. One then simply requests items in the series one-by-one with the next() method of the object just created.

idem = iterator.next()

This assigns the next item in the series to the variable idem.

**Illustrative iteration**

The function of iterators can be seen in an example using the fetchall() method:

#!/usr/bin/env python

import MySQLdb

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

user = 'skipper',

passwd = 'secret',

db = 'fish')

cur = mydb.cursor()

statement = "SELECT \* FROM menu WHERE name='shark'"

cur.execute(statement)

result = cur.fetchall()

series = iter(result)

for i in xrange(0, cur.rowcount):

print series.next()

In this instance, we create an iterator out of the series of records returned by fetchall(). We then iterate through the series according to the number of rows returned, calling the next method of the iterator object series whenever we needed another record.

One can use while with an iterator object. In doing so, however, one must make allowances for either the point at which the returned data ends or the point where the iterator object raises a StopIteration exception.

A StopIteration exception is always raised by an iterator at the end of a series. Using a for loop usually avoids having to handle it.

**Iteration and MySQL for Python**

MySQL for Python's Cursor class supports two methods that are primarily used in iteration: fetchone() and fetchmany(). As shown in the previous sections, it is possible to use within the main flow of a program. While this gets the job done, the better way is to abstract the iteration into a function or method.

**Generators**

Functions and methods that contain loops for controlling iteration are usually generators. By definition, generators are functions or methods that contain the keyword yield.

The term yield was introduced in Python 2.2 as a special keyword. Its sole purpose was to indicate that a function is a generator. Whenever a function contains the keyword yield, Python's byte code compiler would compile it as a result. (see PEP 255 for more on this).

The net effect is that yield causes two main results:

* The function does not return a single value, but a generator object for the iterator defined within it
* The dynamics of the function are suspended after each iteration until the next method is called

What this means for us as programmers is that we never get the full series from a generator, but the ability to access the series in a resource-sensitive way. The data is generated only when we ask for it. In the context of a generator, yield functions like return. The function that includes yield returns a generator object, not a particular value.

**Using fetchone() in a generator**

Creating a generator using fetchone() is simply a matter of incorporating the method into an iterative function. MySQLdb cursors are iterators themselves, but the following code illustrates how to include them in other iterative structures:

#!/usr/bin/env python

import MySQLdb

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

user = 'skipper',

passwd = 'secret',

db = 'world')

cur = mydb.cursor()

statement = "SELECT \* FROM City"

cur.execute(statement)

def iter\_results(cursor, recordnum):

for x in xrange(0,recordnum):

[ 142 ]Chapter 5

result = cursor.fetchone()

if not result:

break

else:

yield result

myresults = iter\_results(cur, 5)

for item in myresults:

print item

Using the world database, we can pass a statement to select every city, but then take only the first three. When dealing with the potential return of large amounts of data, generators allow us to deal with the data in chunks. In this instance, the iterator that is returned will return only the first five rows of those affected by the SQL statement.

**Using fetchmany() in a generator**

Similarly to fetchone(), the fetchmany() method can be incorporated into a generator to return an iterator. As with fetchone(), MySQLdb cursors are iterators themselves. The following code illustrates how to include fetchmany() into another iterative structure. The following code returns all results, but in blocks set by the program call:

#!/usr/bin/env python

import MySQLdb

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

user = 'skipper',

passwd = 'secret',

db = 'world')

cur = mydb.cursor()

statement = "SELECT \* FROM City"

cur.execute(statement)

def iter\_results(cursor, recordnum):

while 1:

result = cursor.fetchmany(recordnum)

if result == None:

break

for item in result:

yield item

myresults = iter\_results(cur, 5)

for idem in myresults:

print idem

[ 143 ]Results Record-by-Record

Currently, the preceding code retrieves results in batches of five rows. But it does so with such speed that it might as well be all at once. The savings here are in memory usage—more or less the same system resources are allotted for each iteration and then released back to the system to be allotted again. So memory usage does not ramp up with each iteration.

If we wanted to affect the same results as we did for the function written for fetchone(), previously, we need to limit the iteration of the generator. We do not need to change the generator. Instead, we need to change the for loop by which we walk through the results. Change the last loop to read as follows:

for span in xrange(0,5):

print myresults.next()

This treats IterResults() as the generator that it is and simply calls the next function of the generator object named myresults.

The output will be the return of the first five records of affected rows.

(1L, 'Kabul', 'AFG', 'Kabol', 1780000L)

(2L, 'Qandahar', 'AFG', 'Qandahar', 237500L)

(3L, 'Herat', 'AFG', 'Herat', 186800L)

(4L, 'Mazar-e-Sharif', 'AFG', 'Balkh', 127800L)

(5L, 'Amsterdam', 'NLD', 'Noord-Holland', 731200L)

**Project: A movie database**

The project for this chapter involves querying the Sakila database from MySQL.

Sakila is a database of fictitious movies and films.

For this project, we will write a program with the following features:

* Accepts user input for the name of either a film or an actor
* Returns the first record for confirmation by the user
* If the first set of data is confirmed as being what the user wants, the entire set is returned
* All errors are fatal and result in error messages being passed to the user
* All warnings are explicitly silenced

[ 144 ]Chapter 5

**Getting Sakila**

The Sakila sample database represents the possible tables for a DVD rental store. It is available for download from the MySQL website:

http://downloads.mysql.com/docs/sakila-db.zip

Once you have downloaded the file, unpack it into a temporary directory. Then you are ready to create the database.

**Creating the Sakila database**

To create the Sakila database, one follows similar steps to what we did for the world database. Where the world database came in a single \*.sql file dump and so required a single source command, Sakila comes in two files and requires them to be sourced in a specific order.

In a terminal session, enter the sakila-db directory (that is, the directory created when you unzip the sakila-db.zip file). Then log into MySQL.

Once logged into MySQL, we may need to create the database itself.

mysql> CREATE DATABASE sakila;

This is not necessary with Sakila as the file will include the CREATE statement. We do not need to tell MySQL the names and structures for the tables explicitly. Instead, we source the schema file:

mysql> source sakila-schema.sql

This creates the structure of the database tables. Next, fill the tables with data by source-ing the data file:

mysql> source sakila-data.sql

Then, having created the database, its tables, and its data, we can look around in the database by use-ing it.

mysql> use sakila;

Further information on installing the Sakila database can be found at

http://dev.mysql.com/doc/sakila/en/sakila.html#sakila-installation.

**The structure of Sakila**

The Sakila database is intended to represent the tables that would drive a DVD rental database application. As such, it has several more tables than world:

mysql> show tables;

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

| Tables\_in\_sakila

|

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

| actor|

| actor\_info|

| address|

| category|

| city|

| country|

| customer|

| customer\_list|

| film|

| film\_actor|

| film\_category|

| film\_list|

| film\_text|

| inventory|

| language|

| nicer\_but\_slower\_film\_list |

| payment|

| rental|

| sales\_by\_film\_category|

| sales\_by\_store|

| staff|

| staff\_list|

| store|

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

[ 146 ]Chapter 5

Of all of these tables now available, we shall use two for this project: actor\_info and film\_list. The first contains the names and films of the actors in the database and has the following structure:

mysql> describe actor\_info;

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

| Field

| Type

| Null | Key | Default | Extra |

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

| actor\_id

| smallint(5) unsigned | NO

|| 0||

| first\_name | varchar(45)| NO|| NULL||

| last\_name| varchar(45)| NO|| NULL||

| film\_info| varchar(341)| YES|| NULL||

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

The film\_list table contains all of the films and their respective actors:

mysql> describe film\_list;

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

-+-------+

| Field

| Extra |

| Type

| Null | Key | Default

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

-+-------+

| FID

|

| smallint(5) unsigned| YES|| 0

| varchar(255)| YES|| NULL

| description | text

|

|| YES|| NULL

| category

|

|| varchar(25)| NO|| NULL

| price

|

|| decimal(4,2)| YES|| 4.99

| length

|

|| smallint(5) unsigned| YES|| NULL

| rating

|

|| enum('G','PG','PG-13','R','NC-17') | YES|| G

| actors

|

|| varchar(341)|| NULL

|

| title

|

|

| YES

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

-+-------+

[ 147 ]Results Record-by-Record

**Planning it out**

Knowing a bit more about the Sakila database, we can take another look at our planned functionality and consider what we will need to do.

* Accept user input for the name of either a film or a thespian

This will entail use of the optparse module again. Depending on which is flagged, we will need one of two possible MySQL statements.

* Return the first record for confirmation by the user

We will rely on fetchone() to do this. This gives the user a preview of the data so we don't waste resources fetching unwanted data.

* If the first set of data is confirmed as being what the user wants, the entire set is returned

This will require fetchmany() to cut down on system resources.

* All errors are fatal and result in error messages being passed to the user
* All warnings are explicitly silenced to the user (they are, however, logged to stderr)

This requires explicit exception-handling as detailed in the previous chapter. Errors and warnings will be explicitly handled in a try...except structure.

**The SQL statements to be used**

To create the functionality we detailed just now, we will need two MySQL statements. One returns the films of an actor or actress. The other returns the actors for a particular film.

**Returning the films of an actor**

The first query uses the actor\_info table to return the movies in which an actor has appeared. Because we ask the user only for the surname of the actor or actress, we need to fetch the first name from the database, along with the list of film titles in which the thespian has appeared. For simplicity's sake, we will return the last name of the actor, as well, in order to keep all of the data in a neat bundle for processing.

The template for this MySQL query is thus:

SELECT first\_name,last\_name,film\_info FROM actor\_info WHERE last\_name

= '<a value>';

Note that we require the user to specify the surname specifically. We will see in the following section how to allow the user to enter partial data.

**Returning the actors of a film**

The second query uses the film\_list table and returns the actors for a given film. To make troubleshooting easier by keeping all results in a single returned value, we will also ask for the title field of the record. A first go at a template for this MySQL query therefore looks like this:

SELECT title,actors FROM film\_list WHERE title = '<a value>';

But this is not satisfactory for our program in terms of usability. The main problem is that this statement requires the user to remember the name of the film in its entirety. But most people do not do that. Usually, the title of a film gets abbreviated to the first few words. Therefore, we need to use a special MySQL keyword: LIKE. To use LIKE in the previous statement, we substitute it for the equals sign =:

SELECT title,actors FROM film\_list WHERE title LIKE '<a value>';

LIKE allows us to use wildcard characters and regular expressions. If we simply gave MySQL a value in the preceding template, the results would tend to be the same as using a symbol of equality. For our purposes, we should anticipate a lack of specificity on the part of the user and allow him or her to input the first words, or even just the first letter of a title. We then rely on MySQL to sort out which titles match. To do this we use the percentage symbol %.

SELECT title,actors FROM film\_list WHERE title LIKE '<a value>%';

The % symbol in MySQL comparisons is similar to the use of an asterisk (\*) in Python's regular expressions.

Note that % is a universal quantifier and matches any number of any characters— even zero occurrences of characters. Therefore, ZERO% will match ZEROS, ZERO TOLERANCE, and ZERO alone. If you only want to match one character, use the underscore wildcard (\_) as an existential quantifier. The string ZERO\_ would then match ZEROS but not ZERO TOLERANCE or ZERO itself.

To match a value that incorporates one of the wildcards, escape from the wildcard with the backslash (\)like you do in Python. So to match ZERO\_ we would use ZERO\\_.

**Accepting user data**

To accept the user data, we have two options. We can either generate a dialogue that walks the user through a series of questions and assigns values within the program as appropriate. Or we can use the optparse module and rely on the user to indicate their preference at runtime. We shall leave the former as an exercise and implement the latter. The beginning of the preamble to this program thus reads:

#!/usr/bin/env python

import MySQLdb

import optparse

We then need to parse the options:

opt = optparse.OptionParser()

opt.add\_option("-a", "--actor", action="store", help="denotes the

lastname/surname of the actor for search - only ONE of actor or film

can be used at a time", dest="actor")

opt.add\_option("-f", "--film", action="store", help="denotes film for

search", dest="film")

opt, args = opt.parse\_args()

As our specification is only for one type of data at a time, we need to ensure that the user does not ask for both actor and film simultaneously. We do this with a simple while test.

badoptions = 0

while opt.film and opt.actor:

print "Please indicate either an actor or a film for which you

would like to search. This program does not support search for both

in tandem."

badoptions = 1

break

The value of status will indicate the overall status of the program—whether it should still be executed. As we shall see, by setting status to 1 we ensure the program does not execute if the user asks for both kinds of searches.

**A MySQL query with class**

For the rest of the features, we will implement a class called MySQLQuery. This class will have the following methods and attributes:

* \_\_init\_\_: To create an instance of the class.
* Type: The type of statement required—whether actor or film.
* Connection: To create the database connection and to return the cursor for data retrieval.
* Query: To create the appropriate type of SQL query. Passes a statement to execute() and passes the results along.
* Execute: To execute the statement formed by query(), retrieve the data as appropriate, and then pass it along to format() for formatting. It then receives the results back from format() and passes it up the chain to query().
* Format: To receive output from execute(), parse it and repackage it appropriately, and pass it back to execute for further returning.

**The \_\_init\_\_ method: The consciousness of the class**

The first method of the class is, of course, \_\_init\_\_. As discussed in previous chapters, an \_\_init\_\_ method is customary for the proper functioning of a class. It is not necessary; the class can function without it. However, using one allows us to customize the nature of the object being initiated.

More on Python classes and the use of \_\_init\_\_ can be found in the Python documentation:

<http://docs.python.org/tutorial/classes.html#class-> objects

For this project, the \_\_init\_\_ method looks like this:

def \_\_init\_\_(self):

"""Creates an instance to form and execute a MySQL

statement."""

self.Statement = []

**Setting the query's type**

Once the object is instantiated, we need to set the type of the query. This is critical to the smooth running of the rest of the program as follows:

def type(self, kind):

"""Indicates the type of statement that the instance is.

Supported types are select, insert, and update. This must be set

before using any of the object methods."""

self.type = kind

As in the project from last chapter, we simply assign the value of kind to self.type.

**Creating the cursor**

Naturally, in order to query the database, we must have a cursor by which we execute a query and fetch results. Rather than code this into the main() function or into the preamble, we put it here in a function on its own:

def connection(self):

"""

Creates a database connection and returns the cursor.

information is hardwired.

HOST = localhost

USER = skipper

DATABASE = sakila

"""

try:

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

user = 'skipper',

passwd = 'secret',

db = "sakila")

cur = mydb.cursor()

return cur

All login

Obviously, this implementation can be called from any other Python program that can import it. You therefore want to be careful about permissions and other security issues. While this can be a trajectory by which a login is leaked, hardwiring the login ensures that a rogue user cannot exceed the permissions of the given user.

It is worth noting that any Python program that can import a module can also read that module's code by using the inspect module. To illustrate, name the program as moviesearch.py. Then open a Python shell in the directory that holds the file. In the shell, type the following:

>>> import moviesearch, inspect

>>> inspect.getsource(moviesearch)

You will then be treated to a full printout of the code for this program. Any user on a network can do the same thing if they have access to the module.

Naturally, we need to handle any fallout from a failed connection. Here we handle exceptions in a blanket manner, but one can (and should) implement appropriate rules for each possible exception.

except MySQLdb.Error:

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

Please ensure that the 'sakila' database exists on the local host

system."

raise

except MySQLdb.Warning:

pass

**Forming the query**

The next step requires us to form the query. To do this, we will use the templates discussed previously and insert them appropriately into the method. The user's input will be received as value.

This function will be used to handle all queries. So we must be able to toggle between the initial sample and a fuller query. To do this, we will use a sample switch. The opening line of the definition thus reads:

def query(self, value, sample):

We then start by testing the value of sample. If it is 1, we return a sample of the data.

if sample == 1:

if self.type == 'actor':

statement = """SELECT first\_name,last\_name,film\_info

FROM actor\_info WHERE last\_name = '%s'""" %(value)

else:

statement = """SELECT title,actors FROM film\_list

WHERE title LIKE '%s'""" %("%" + value + "%"

returnself.execute(statement, sample)

Otherwise, we retrieve all records that match.

else:

if self.type == 'actor':

statement = """SELECT first\_name,last\_name,film\_info

FROM actor\_info WHERE last\_name = '%s'""" %(value)

else:

statement = """SELECT title,actors FROM film\_list

WHERE title LIKE '%s'""" %("%" + value + "%")

results = self.execute(statement, sample)

return results

In either case, we check the value of self.type, form the appropriate query, and pass the resulting statement and the value of sample to the execute() method. When results are received from execute() we pass them back to the calling process.

**Executing the query**

The execute() method calls the connection() method and uses the returned cursor to execute the statement received from query() because its behavior still relies on the value of sample, its structure is similar to query().

def execute(self, statement, sample):

"""Attempts execution of the statement resulting from

MYSQLQuery.form()."""

while True:

try:

cursor = self.connection()

cursor.execute(statement)

if cursor.rowcount == 0:

print "No results found for your query."

break

elif sample == 1:

output = cursor.fetchone()

results = self.format(output, sample)

return results

else:

output = cursor.fetchmany(1000)

results = self.format(output, sample)

return results

Note that the argument to fetchmany() is the total number of records in Sakila. In real-life situations where multiple users are working against much larger databases, you would do well to iterate through the records.

Since the statement does not have to come from query(), but can be passed by a calling module, we again should be ready to handle any fallout from a failed query.

except MySQLdb.Error:

raise MySQLdb.Error

except MySQLdb.Warning:

pass

[ 154 ]Chapter 5

Here as we handle exceptions in a blanket fashion, it is advisable to handle exceptions in greater detail.

**Formatting the results**

As seen previously, the output of the search is passed to format() in order to be repackaged for the user. The value of output is a tuple and must be processed accordingly. Once again, how the data is processed is determined by whether it is a sample or not. The definition declaration thus reads:

def format(self, output, sample):

In the course of this method, we will use iteration often. Therefore, we create a blank object called results onto which we can add the parsed data.

results = ""

**Formatting a sample**

If a sample is needed, we then use an if...else loop to control program flow according to whether the user is searching for a drama or a film.

if sample == 1:

if self.type == "actor":

data = output[0] + " " + output[1] + ": "

titles = output[2]

entry = titles.split(';')

data = data + entry[0].split(':')[1]

results = results + data + "\n"

return results

else:

data = output[0] + ": "

actors = output[1]

data = data + output[1]

results = results + data + "\n"

return results

If an actor is sought, the output will follow this template:

<first\_name> <last\_name>: <first few titles>

For a film, the results would be:

<title>: <list of actors in a comma-delimited series>

**Formatting a larger set of results**

If a sample is not required, it is safe to assume that the full amount of results are to be processed. This forms the else part of the greater if...else loop of this method. Within it, we have another if...else loop to process the data by actor or title, as we did with the previous sample return.

else:

if self.type == "actor":

for record in output:

actor = record[0] + " " + record[1] + ": "

for item in xrange(2,len(record)):

names = record[item].split(';')

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

if i == 0:

titles = "\n " + names[i]

else:

titles = titles + '\n' + names[i]

data = actor + titles + '\n'

results = results + data + "\n"

else:

for record in output:

title = record[0] + ": "

for item in xrange(1, len(record)):

names = record[item].split(',')

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

if i == 0:

actor = "\n " + names[i]

else:

actor = actor + '\n' + names[i]

data = title + actor + '\n'

results = results + data + '\n'

return results

The results of either part of the loop will be in this format:

<full name of film or thespian>:

<list of actors in the film or films of the thespian>

If any part of a program's execution is unclear, the best way to figure out what's going on is to use print. Use it not only to print out values at different points of execution, but also the types of different variables. For example, if one inserts print output on the line after assigning results, Python will output that value. But if one inserts print type(output), Python tells us that the variable output is a tuple.

**The main() thing**

Having coded the MySQLQuery class, we can now write the main() function of the program. The first thing to do in main() is to test the value of status. We do this with a while loop that continues as long as status is equal to 0 (and break is not called).

while status == 0:

We then need to create an instance of MySQLQuery and try to execute the operational part of main(). This includes first assigning the type of query to MySQLQuery.type.

while status == 0:

request = MySQLQuery()

try:

if opt.actor:

request.type("actor")

value = opt.actor

elif opt.film:

request.type("film")

value = opt.film

Next, we query the database by calling the query() method. You will recall that query() passes the MySQL statement to execute(), which then passes the results to format(). The results are then returned down the chain. If you want to ensure that resources are relinquished before the next method is called, simply return the output of each method back to main() before calling the next one.

results = request.query(value, 1)

Here we pass the value to query() as well as a 1 to indicate the need for a sample. If results are returned, the sample should be output to the user for confirmation.

if results:

print "Sample returns for the search you requested are

as follows."

print results

confirm = raw\_input("Are these the kind of data that

you are seeking? (Y/N) ")

confirm = confirm.strip()

[ 157 ]Results Record-by-Record

The confirm variable should ideally begin with either a Y or an N. To ensure that the user has not accidentally hit the space bar before entering his or her response, we strip the whitespace out of the input value. If the answer is yes, we expect the first character then to be a capital Y, as indicated. If it is not, we default to a fatal break.

if confirm[0] != 'Y':

# if confirmation is not given,

then break.

print "\n\nSuitable results were not found.

Please reconsider your selection of %s and try again.\n" %(request.

type)

break

If confirmation is given, we then use the same object and send a second query call for the full listing of records.

if confirm[0] == 'Y':

results = request.query(value, 0)

print "\n\nResults for your query are as follows:\n\n"

print results

break

If there are no results, execute() tells the user so. We can therefore simply break.

else:

break

Finally, if trying to run the core of the main() method fails, we need to handle the fallout. Here we implement some general exception-handling that should be more robust in most applications of any size.

except MySQLdb.Error:

raise MySQL.Error

except MySQLdb.Warning:

pass

**Calling main()**

Finally, we call main() only if the program is called directly.

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

main()

[ 158 ]Chapter 5

**Running it**

If you save the file now as moviesearch.py, you can call it from the command-line with either an -a or -f flag for actor or film, respectively. Of course, there is also the helpful—h flag to explain the syntax.

**Room to grow**

The results of this project may appear to be marginally more complex than the example on retrieving from earlier in this book. However, this project can easily be extended and applied in several ways to form the basis of a fully-functional program. Some points for extension are as follows:

* Step through the results one-by-one, waiting for the user to indicate when to proceed to the next record
* Create a menu from the results returned and allow the user to select which record to return
* Allow the user to select multiple records to be either returned to the screen or output to a file
* Using the other tables in the database, develop reports of customer trends by actor and genre
* Create a web-based, point-of-sale (POS) interface that would allow users to input the DVDs being rented and register how much was being received from which customer
* For that same POS interface, write the necessary code for the program to recommend movies for the customer based upon the actors and genres of their past rentals

**Summary**

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

* Situations in which record-by-record retrieval is desirable
* How to use iteration to retrieve sets of records in smaller blocks
* How to create iterators and generators in Python
* When to use fetchone() and when to use fetchmany()

In the next chapter, we will look at how to handle multiple MySQL inserts.