MySQL\_for\_Python\_Albert\_c13

owing MySQL Metadata

In previous chapters, we have moved from the basics of selecting, inserting, creating,

and dropping in MySQL through Python. In this chapter, we will look at accessing

MySQL's metadata through Python. We have seen a lot of this incidentally in

preceding chapters. However, in this chapter, we will look at them in greater depth.

By the end of this chapter, we will see the following:

•What MySQL metadata is available to us and how to access it

•How to get a list of databases and tables

•Ways to switch databases on-the-fly

•How to get columnar information

This information is very useful for creating intermediate levels of database

management. At the end of the chapter, we will look at how to create a class

representation of a database.

Within a MySQL session, the easiest way to access metadata is by using the

INFORMATION\_SCHEMA pseudo-table. By switching to it with:

USE INFORMATION\_SCHEMA;

you can access a plethora of information. Doing this from within a Python program

increases your program's overhead. So in this chapter, we will look at how to access

metadata without switching databases.Showing MySQL Metadata

The primary way to get MySQL to tell you anything is the SHOW command. What

you get in return naturally depends on what you ask for. There are many different

arguments that you can pass to SHOW, some of which appear as follows:

Arguments

EVENTS

BINARY LOGS

BINLOG EVENTS

CHARACTER SET

COLLATION

COLUMNS

CONTRIBUTORS

CREATE

DATABASES

ENGINE

ENGINES

ERRORS

FUNCTION CODE

FUNCTION STATUS

GRANTS

INDEX

MASTER STATUS

OPEN TABLES

PLUGINS

PRIVILEGES

PROCEDURE CODE

PROCEDURE STATUS

PROCESSLIST

PROFILES

RELAYLOG EVENTS

SLAVE HOSTS

SLAVE STATUS

STATUS

TABLE STATUS

TABLES

TRIGGERS

VARIABLES

WARNINGS

These are basic options, most of which can be further nuanced with the use of LIKE

or other keywords to nuance or restrict the results. Some of these require different

privileges than the average user. While many of these will be invaluable in accessing

information about MySQL and the databases you access, many others are used

infrequently, if at all. In the following discussion, we will look at key information

about databases, tables, and MySQL system information that is most relevant to

Python programming.

MySQL's system environment

MySQL offers access to a wide array of information about the environment in

which your database is stored and the variables that impact your access to it. Of

particular import for programming is understanding MySQL's engines, profiles,

and system variables.

[ 370 ]Chapter 13

ENGINE

To understand the use of MySQL's ENGINE command, it is important to first

understand what MySQL means by a database storage engine. When you create a

database, MySQL defaults to storing data with the MyISAM database engine. It is a

transactionless database storage engine that uses the following three files:

1A format file (.frm).

2An index file (.MYI for MYIndex).

3Data file (.MYD for MYData).

MyISAM is sufficiently robust for most purposes, but there are following eight others

that are worth noting:

1InnoDB: A high-performance database engine for processing large volumes of

2IBMDBI: A transaction-capable database engine designed for IBM's DB2 table

format on IBM i servers.

3MERGE: A merger of two or more MyISAM tables to function as one.

4MEMORY (HEAP): An engine that stores all tables in memory; this is very fast

5

data with efficient CPU usage and the use of transactions.

but resource-intensive.

FEDERATED: Allows access to a remote database through a local MySQL

instance.

6ARCHIVE: An index-less storage engine that allows for efficient data storage.

7CSV: Stores data in comma-delimited files.

8BLACKHOLE: Receives data, but does not store it.

Oracle, the parent company of MySQL, has discussed changing the default to InnoDB

in MySQL 5.5, so the default engine may change before MySQL 5.5 makes it out of

beta. For other versions, the default engine remains MyISAM.

[ 371 ]Showing MySQL Metadata

The most popular engines

While MySQL supports each of these engines, practice sees the following storage

engines are used most often: MyISAM, InnoDB, and MEMORY. MySQL's comparison of

these looks like the following:

MyISMInnoBMEMORY

Multi-statement transactions,

ROLLBACK-X-

Foreign key constraints-X-

Locking leveltablerowtable

BTREE indexesXX-

FULLTEXT indexesX--

HASH lookups-XX

Other in-memory tree-based index--X

GIS, RTREE indexesX--

UnicodeXX-

Merge (union views)X--

Compress read-only storageX--

Relative disk uselowhigh-

Relative memory uselowhighlow

Additionally, many MySQL installations use networked databases referred to as Network

Database (NDB), which combines several standard MySQL databases with the cluster-

oriented storage engine for which it is named. You can find more about clustering MySQL

databases at http://dev.mysql.com/doc/refman/5.0/en/mysql-cluster-

overview.html

Transactions

While many of these points of comparison are self-explanatory, the use of

transactions is worth further explanation because we have not used it much in this

book. For the purposes of database management, a transaction is treated as a unit of

work that can be compared against other units of work performed within a database

management system. Put plainly, a transaction is effectively a collection of queries

passed to MySQL in a particular order for a particular purpose.

Obviously, if you are retrieving data or inserting completely new information, the

order in which the statements are executed frequently does not matter. However,

if you use UPDATE or other statements that change the substance or structure of a

dataset, the order matters a great deal.

[ 372 ]Chapter 13

Using transactions allows one to change the state of the database back to what it

was before the commands were issued, and usually requires the use of the COMMIT

keyword to commit the changes once they are affected. In MySQL for Python, we

use the commit() method of the connection object after we have entered the data. Of

the engines shown in the previous table, only InnoDB supports multiple-statement

transactions and the ability to roll the state of the database back using ROLLBACK.

For more on using transactions in InnoDB, see the MySQL manual at

http://dev.mysql.com/doc/refman/5.5/en/commit.html

Specifying the engine

To specify which engine to use, simply append the option to the CREATE statement

for the table as follows:

CREATE TABLE mydb(myfield INT) ENGINE = <engine name>;

MySQL does allow each table of a database to use a different engine. For example:

mysql> CREATE TABLE t1(i1 INT) ENGINE=MyISAM;

Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE t2(i2 INT) ENGINE=INNODB;

Query OK, 0 rows affected (0.10 sec)

mysql> CREATE TABLE t3(i3 INT) ENGINE=MEMORY;

Query OK, 0 rows affected (0.00 sec)

This results in three tables each using a different engine.

For information on using table spaces with InnoDB tables, see:

http://dev.mysql.com/doc/refman/5.5/en/multiple-

tablespaces.html

Whichever engine is used for a given table will naturally have an impact over how

that data is stored and indexed. This can further have an impact on the choice

of what backup method you use with which table (for more on MySQL disaster

recovery, see the discussion in the next chapter).

ENGINE status

Each engine in MySQL is treated separately of the others. If you use InnoDB, MySQL

allows you to access the log and the status information from within MySQL. To do

so, we use the ENGINES keyword with SHOW.

mysql> SHOW ENGINE InnoDB STATUS;

[ 373 ]Showing MySQL Metadata

The output will likely run off your screen. If you are on Windows or a similar

graphic client to MySQL, this is usually not a problem. However, if you are using a

traditional terminal on a Unix or Linux server, you would normally need a mouse

to scroll through the data. A better way is to affix a \G at the end of the query. Or

you can set the MySQL pager to whatever text viewing application is installed on

the server that will receive piped input. On Unix machines, the text viewer more is

generally available; on Linux, less is available. To pass output to the latter, use the

PAGER command as follows:

mysql> PAGER less;

Now, less will be used to display output from MySQL, and the results of SHOW

ENGINE can be browsed easily. Note that, because we set the PAGER value within the

MySQL session, the assignment is only temporary and will refresh to the default

when you next log in. Making the change persistent can be affected with the—PAGER

flag, if you log in from a shell prompt (for more on this see: http://dev.mysql.

com/doc/refman/5.5/en/mysql-commands.html).

SHOW ENGINES

Sometimes when working on a system that you did not set up, it is necessary to

confirm, which database engines are supported by the local installation. To do that,

use SHOW ENGINES. The output will be a table of all storage engines known to the

current installation and their support status. For example:

mysql> SHOW ENGINES;

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

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

| Engine

| Support | Comment

|+------------+----------+-----------------------------------------------

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

| MyISAM

| DEFAULT | Default engine as of MySQL 3.23 with great

performance

|

| MEMORY

| YES

| Hash based, stored in memory, useful for

temporary tables

|

| InnoDB

| YES

| Supports transactions, row-level locking, and

foreign keys

|

| BerkeleyDB | NO

| Supports transactions and page-level locking

|

| BLACKHOLE | YES

| /dev/null storage engine (anything you write to

it disappears) |

| EXAMPLE

| NO

| Example storage engine

|

| ARCHIVE

| YES

| Archive storage engine

|

[ 374 ]Chapter 13

| CSV

| YES

| CSV storage engine

|

| ndbcluster | DISABLED | Clustered, fault-tolerant, memory-based tables

|

| FEDERATED | YES

| Federated MySQL storage engine

|

| MRG\_MYISAM | YES

| Collection of identical MyISAM tables

|

| ISAM

| NO

| Obsolete storage engine

|

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

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

Note that this example was run against an earlier 5.x version of MySQL. MySQL 5.5,

for example, does not support the BerkeleyDB.

Profiling

Profiling is MySQL's form of user monitoring. When used, it tracks several aspects

of system performance and the resources used in a given session. By default, it is

switched off. To turn it on, key:

mysql> SET PROFILING = 1;

When profiling is switched on, you can view your session data using one of two

keywords.

SHOW PROFILE

To show your profile, simply enter:

mysql> SHOW PROFILE;

MySQL will then return several statistics that will look similar to the following:

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

| Status

| Duration |

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

| (initialization)| 0.000064 |

| Opening tables| 0.000025 |

| query end| 0.000008 |

| freeing items| 0.00001

|

| logging slow query | 0.000005 |

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

[ 375 ]Showing MySQL Metadata

SHOW PROFILES

However, if you want to view performance by query, you need to use the PROFILES

keyword. This will trace your history as you switched on profiling and recount the

execution times for each query you passed:

mysql> SHOW PROFILES;

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

| Query\_ID | Duration

| Query

|

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

|1 | 0.00017000 | SHOW ENGINES

|

|2 | 0.00100500 | SHOW ENGINE innodb STATUS |

|3 | 0.00011200 | SET PROFILING = 1

|

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

SHOW system variables

MySQL also provides access to a plethora of database system variables when you

issue the query such as:

mysql> SHOW VARIABLES;

If you are using MySQL's default pager, the results table will usually scroll off the

screen. One does not normally browse the system variables. On the contrary, one

uses matching to cull out a particular value. For example, if we wanted to know

which version of MySQL is in use, we would match against the value version:

mysql> SHOW VARIABLES LIKE 'version';

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

| Variable\_name | Value

|

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

| version

| 5.0.51a-3ubuntu5.4 |

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

[ 376 ]Chapter 13

If, however, we did not know the variable that we needed, we can use the following

wildcard matching just as easily:

mysql> SHOW VARIABLES LIKE '%version%';

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

| Variable\_name

| Value

|

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

| protocol\_version| 10

|

| version| 5.0.51a-3ubuntu5.4 |

| version\_comment| (Ubuntu)

|

| version\_compile\_machine | i486|

| version\_compile\_os|

| debian-linux-gnu

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

Exactly which variables are available will differ by installation and server version.

For a comprehensive list of variables with links to the import of each, see:

http://dev.mysql.com/doc/refman/5.5/en/server-system-variables.html

Be sure to adjust the 5.5 (version) to read according to your server version.

Accessing database metadata

MySQL provides access to several aspects of databases, tables, or extensions thereof.

Combining the various options creates the potential for interactive programs, as

we will see later in this chapter. As with the MySQL environmental commands, all

statements discussed here presume the use of SHOW.

DATABASES

If you are creating a wholly interactive system for database administration,

you will need to access the list of databases. To do this in MySQL, we use the

following command:

mysql> SHOW DATABASES;

The result is a single column table showing all known databases on the system to

which the user has access. Note that the appearance of a database on the list does not

indicate permission to access it. Rather, you would also need to query the privileges

of the user, as we will see when accessing user information in the next main section.

[ 377 ]Showing MySQL Metadata

Using the USE command

When you have a list of the available databases, you can access a database within the

current session using USE:

mysql> USE sakila;

Reading table information for completion of table and column names

You can turn off this feature to get a quicker startup with -A

Database changed

When creating a database connection object in MySQL for Python, it is not necessary

to indicate the database to be used in order to log in. Rather, one can leave that blank

as follows:

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

If we want to be more overt and show that we purposely are not declaring the

database, we can simply use a blank string:

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

or, even better:

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

user='skipper',

password='secret',

db='')

However, in order to do anything that uses a database table, one must issue a USE

statement through the execute() method of a cursor object:

statement = "USE sakila"

runit = cursor.execute(statement)

If in doubt over privileges, be sure to couch this part of your code in a

try...except structure.

Accessing metadata about tables

Once you indicate to MySQL about the database you are going to use, there are

several table-oriented operations that you can perform. You can also ascertain

several dynamics about any given table.

SHOW TABLES

Like DATABASES, the TABLES keyword will cause MySQL to return a list of all tables

in a database to which you have access. The syntax is simply:

mysql> SHOW TABLES;

[ 378 ]Chapter 13

SHOW TABLE STATUS

In addition to seeing the table names, we can also access several pieces of

metainformation for each table using TABLE STATUS.

mysql> SHOW TABLE STATUS;

Depending on how many of the tables in the database we are allowed to access, the

results can be quite long. Therefore, it is usually advisable to restrict the matches

with a WHERE clause (or LIKE clause) and a parameter of equality as appropriate.

Showing columns from a table

For any given table, we can access the name, type, and default value for every field in

a table. We can also see whether a NULL value is allowed, whether the field holds the

primary key for the table, and any extra options that apply to the table. For example,

using sakila, we can get the format of the city table as follows:

mysql> SHOW COLUMNS FROM city;

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

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

| Field

Extra

| Type

|

| Null | Key | Default

|

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

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

| city\_id

| smallint(5) unsigned | NO

auto\_increment || PRI | NULL|

| city

|| varchar(50)|| NULL|

| country\_id

|| smallint(5) unsigned | NO| MUL | NULL|

| last\_update | timestamp

|

|

| NO

| NO

|

| CURRENT\_TIMESTAMP

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

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

A synonym for COLUMNS FROM is DESCRIBE. Therefore, we could get the same data

with the following:

mysql> DESCRIBE city;

[ 379 ]Showing MySQL Metadata

FUNCTION STATUS

Using FUNCTION STATUS by itself will return a list of all available stored functions

known to the system. For each function, MySQL will return the following:

•The name of the database to which the function pertains

•The name of the database, its type

•The user who defined the function

•Its last modification date

•When it was created

•The role of the users who are associated with the function

•Any comments that are used to describe it

Depending on how your system is set up, this list could get quite long. So as with

other operations, we can nuance what is returned by using a parameter of equality

and, if necessary, a wildcard.

To return information about the function Capitalise(), defined in the previous

chapter, we would use the following query:

mysql> SHOW FUNCTION STATUS LIKE 'Capitalise';

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

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

| Db

| Name

Created

| Type

| Definer

| Security\_type | Comment |

| Modified

|

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

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

| javab | Capitalise | FUNCTION | skipper@localhost | 2010-03-24 11:29:13

| 2010-03-24 11:29:13 | DEFINER

|

|

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

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

CREATE (DATABASE/FUNCTION/PROCEDURE/TABLE/VIEW)

Sometimes, it can be difficult to know the reason an aspect of a MySQL database

functions as it does. While SHOW COLUMNS and SHOW TABLE STATUS offer a very good

picture, it is not unheard of for a developer to think the database was formed with

one definition when it was created with another. Therefore, MySQL allows us to see

how it understands things to be and does so by giving us the defining statements of

the object involved.

[ 380 ]Chapter 13

To view the code for the sakila database, for example, we would issue the following

query (using a postpended \G if needed):

mysql> SHOW CREATE DATABASE sakila;

and get these results:

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

| Database | Create Database

|

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

| sakila

| CREATE DATABASE `sakila` /\*!40100 DEFAULT CHARACTER SET

latin1 \*/ |

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

The same goes for a table. If we are using sakila, we can see the following creation

statement for the film table. Using \G will spare you a lot of the formatting, which

can become noise on a smaller screen.

mysql> SHOW CREATE TABLE film;

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

-------------------------------------...(ellipses for MySQL's formatting

dashes)...

-------------------------------------------------------------------------

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

| Table | Create Table

|

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

-------------------------------------...(ellipses for MySQL's formatting

dashes)...

-------------------------------------------------------------------------

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

| film

| CREATE TABLE `film` (

`film\_id` smallint(5) unsigned NOT NULL auto\_increment,

`title` varchar(255) NOT NULL,

`description` text,

`release\_year` year(4) default NULL,

`language\_id` tinyint(3) unsigned NOT NULL,

`original\_language\_id` tinyint(3) unsigned default NULL,

`rental\_duration` tinyint(3) unsigned NOT NULL default '3',

`rental\_rate` decimal(4,2) NOT NULL default '4.99',

[ 381 ]Showing MySQL Metadata

`length` smallint(5) unsigned default NULL,

`replacement\_cost` decimal(5,2) NOT NULL default '19.99',

`rating` enum('G','PG','PG-13','R','NC-17') default 'G',

`special\_features` set('Trailers','Commentaries','Deleted

Scenes','Behind the Scenes') default NULL,

`last\_update` timestamp NOT NULL default CURRENT\_TIMESTAMP on update

CURRENT\_TIMESTAMP,

PRIMARY KEY

(`film\_id`),

KEY `idx\_title` (`title`),

KEY `idx\_fk\_language\_id` (`language\_id`),

KEY `idx\_fk\_original\_language\_id` (`original\_language\_id`),

CONSTRAINT `fk\_film\_language` FOREIGN KEY (`language\_id`) REFERENCES

`language` (`language\_id`) ON UPDATE CASCADE,

CONSTRAINT `fk\_film\_language\_original` FOREIGN KEY (`original\_language\_

id`) REFERENCES `language` (`language\_id`) ON UPDATE CASCADE

) ENGINE=InnoDB AUTO\_INCREMENT=1001 DEFAULT CHARSET=utf8 |

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

-------------------------------------...(ellipses for MySQL's formatting

dashes)...

-------------------------------------------------------------------------

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

1 row in set (0.09 sec)

If we wanted to see the code for the Capitalise() function, we would issue:

mysql> SHOW CREATE FUNCTION Capitalise;

However, we must be in the appropriate database for that function. In sakila, we

get this message back:

ERROR 1305 (42000): FUNCTION Capitalise does not exist

If we switch to the database for which we defined Capitalise(), however, we will

get the function definition returned.

Similar operations can be performed for procedures and views.

[ 382 ]Chapter 13

Accessing user metadata

If you write a program that interacts with MySQL dynamically, it will need to adopt

its behavior based on the server setup and the characteristic of a user's account.

You will then need to be able to access what privileges have been granted to a user

from within a MySQL session; you obviously cannot count on having administrator

access. For this reason, MySQL provides access to user information from within

a session.

SHOW GRANTS

In using SHOW GRANTS, we ask MySQL to return the GRANT statements used to grant

privileges to the user. The results will show the precise tables for which permission

has been granted. However, any passwords that were part of the original GRANT

statement are returned as a hash.

As a user mammamia, we can see the GRANT statements that pertain to that account as

shown below:

mysql> SHOW GRANTS;

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

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

| Grants for mammamia@localhost|

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

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

| GRANT USAGE ON \*.\* TO 'mammamia'@'localhost' IDENTIFIED BY PASSWORD

'\*41BAD3DEEB08D03DA99724882859C3188BAEC952' |

| GRANT SELECT ON `fish`.`menu` TO 'mammamia'@'localhost'|

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

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

These results can be parsed in Python and acted upon or just cataloged.

Depending on the user's level of access, you can show grants for other users by using

FOR clause added to SHOW GRANTS syntax as follows:

SHOW GRANTS FOR mammamia;

[ 383 ]Showing MySQL Metadata

PRIVILEGES

Of course, if the user is granted all privileges as indicated by the universal wildcard

(\*), simply parsing the results will still keep you guessing as to what those privileges

actually are. The privileges available to any user are naturally a subset of the

privileges supported by the server, which are themselves configurable and vary

from installation to installation. For this reason, MySQL also provides a way to learn

all of the privileges supported by the server and what they do. Simply use SHOW

PRIVILEGES.

What output you get for this statement depends on your server version and

how the server was configured.

Project: Building a database class

When you are new to a project or just unfamiliar with a database, getting up to speed

on the oral tradition of your team and (all too frequently) sparse documentation can

be quite frustrating. In such instances, being able to access the technical information

for a database can be quite helpful. The project for this chapter is therefore to build

a database class that gives you easy access to metainformation for a database of

your choosing.

As we have seen, MySQL results are not always easy to read in the aggregate.

Therefore, we will also need code that will digest the results and output them in

human-friendly format. The specification points for this project are thus:

•

•

•

•

Develop a class to access MySQL metadata

Return metainformation on a specified database using the class instance

Reformat tabular information to be more friendly to human readers

Print out a report of the information

We will save the boiler-plate code of creating the connection and checking the value

of \_\_name\_\_ until the very end. First, let's look at creating the class.

Writing the class

In principle, all that is necessary is to fulfill the first item and code the several

queries as methods of a class. Each method defines a statement that is then executed,

returning the results. The class definition begins as follows:

class Database:

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

"A class representation for MySQL database metadata"

self.database = []

[ 384 ]Chapter 13

Defining fetchquery() and some core methods

We could then write every method to execute each query within its own method like

the following:

def tables(self, cursor):

"Returns a list of the database tables"

statement = "SHOW TABLES"

header = ("Tables")

try:

runit = cursor.execute(statement)

results = cursor.fetchall()

except MySQLdb.Error, e:

results = "The query you attempted failed. Please verify

the information you have submitted and try again. The error message

that was received reads: %s" %(e)

return header, results

However, unnecessary repetition is the midwife of many code errors. Further, the

excess code created by such repetition serves to enlarge the resources needed for the

program. This may seem small, but the more the code is used the greater the drain

on resources it becomes. The best way to ensure the same operation is performed the

same way every time is to write it once and pass arguments to it.

It is therefore better to write an internal function and pass all statements to it. We can

write a method fetchquery() to serve this purpose. The revised code would look

like this:

# Execute straightforward queries

def fetchquery(self, cursor, statement):

"Internal method that takes a statement and executes the

query, returning the results"

try:

runit = cursor.execute(statement)

results = cursor.fetchall()

except MySQLdb.Error, e:

results = "The query you attempted failed. Please verify

the information you have submitted and try again. The error message

that was received reads: %s" %(e)

return results

def tables(self, cursor):

"Returns a list of the database tables"

statement = "SHOW TABLES"

header = ("Tables")

results = self.fetchquery(cursor, statement)

return header, results

[ 385 ]Showing MySQL Metadata

When retrieving data based on user information, there is always a chance of an

error. Therefore, we couch the execution lines of fetchquery() in a try...except

structure, returning the error message if it arises.

Retrieving table status and structure

We can similarly write functions to retrieve the table statuses and structure. These

we call tbstats() and describe().

def tbstats(self):

"Returns the results of TABLE STATUS for the current db"

header = ("Name", "Engine", "Version", "Row\_format", "Rows",

"Avg\_row\_length", "Data\_length", "Max\_data\_length", "Index\_length",

"Data\_free", "Auto\_increment", "Create\_time", "Update\_time", "Check\_

time", "Collation", "Checksum", "Create\_options", "Comment")

statement = "SHOW TABLE STATUS"

results = self.fetchquery(statement)

return header, results

def describe(self, tablename):

"Returns the column structure of a specified table"

header = ("Field", "Type", "Null", " Key", "Default", "Extra")

statement = "SHOW COLUMNS FROM %s" %(tablename)

results = self.fetchquery(statement)

return header, results

While we name the method describe(), we use the MySQL call SHOW COLUMNS. This

is purely stylistic, and you could just as easily use the call DESCRIBE.

Retrieving the CREATE statements

Next, we can retrieve the CREATE statements for each of the database and tables. Just

like executing the query, we here need to use the same code repeatedly. Therefore,

we write a function to assemble the statement based on the information it receives

from the calling function. Let's call it getcreate():

# Retrieve CREATE statements

def getcreate(self, type, name):

"Internal method that returns the CREATE statement of an

object when given the object type and name"

statement = "SHOW CREATE %s %s" %(type, name)

results = self.fetchquery(statement)

return results

[ 386 ]Chapter 13

This method takes the type and name of the object and forms the SHOW CREATE

statement for it. That statement is then passed to fetchquery(), and the results of

that method are passed back as the results of this one.

For the creation statement of the database and the table, the method is:

def dbcreate(self):

"Returns the CREATE statement for the current db"

type = "DATABASE"

name = db

header = ("Database", "Create Database")

results = self.getcreate(type, name)

return header, results

def tbcreate(self, tbname):

"Returns the CREATE statement for a specified table"

type = "TABLE"

header = ("Table, Create Table")

results = self.getcreate(type, tbname)

return header, results

There are other metadata that we could retrieve, but these serve as illustrations.

Next, we'll look at calling these methods and handling the results.

Define main()—part 1

We need to define the main() function that will serve as the brains of the program.

For the moment, we will use main() simply to instantiate the class and run through

the methods, printing out their results. The beginning of the function thus reads:

def main():

mydb = Database()

To get an idea of what the output from these methods looks like, we can then write a

series of print commands as follows:

print mydb.tables()

print mydb.tbstats()

print mydb.dbcreate()

for i in mydb.tables()[1]:

print mydb.describe(i)

As you may note in looking at the methods, each returns a tuple in the same format

as: header and results. However, the describe() method requires a table as its

argument. So we pass the for loop the second part of the results from tables().

[ 387 ]Showing MySQL Metadata

A single run of this program will illustrate how unreadable the results are. MySQL

for Python serves us well by turning MySQL's tables into sequences, but these are

then unreadable by human users. We need to massage the data format a bit. For

this we have two options—either write a custom routine or abstract the handling

of results to a function. The first is illustrated, as follows, with regard to the results

of tables():

tables = mydb.tables()

print "Tables of %s" %(db)

for c in xrange(0, len(tables[1])):

print tables[1][c][0]

print '\n\n'

We simply use the length of the results to form a for loop that prints the results in

sequence. When it comes to other methods, however, it is not a bad idea to write a

separate function.

Writing resproc()

What we want in resproc() is a method that parses the output from the Database

methods and returns a formatted output. The first thing we need to do upon defining

the function is to assign the two parts of the tuple input to different variables for ease

of processing.

def resproc(finput):

"Compiles the headers and results into a report"

header = finput[0]

results = finput[1]

Now we are in a position to split both parts of the input and compile them into

something more meaningful to the human eye. First, we split the results and

create a new dictionary out of the two called output.

output = {}

c = 0

for r in xrange(0, len(results)):

record = results[r]

outrecord = {}

for column in xrange(0, len(header)):

outrecord[header[column]] = record[column]

output[str(c)] = outrecord

c += 1

[ 388 ]Chapter 13

Next, we create a string for the results of the function. We then walk through our

new dictionary and add the results to the string for a nicely formatted report.

orecord = ""

for record in xrange(0, len(results)):

record = str(record)

item = output[record]

for k in header:

outline = "%s : %s\n" %(k, item[k])

orecord = orecord + outline

orecord = orecord + '\n\n'

return orecord

It should be noted that the formatting used here is just for a text file. We could just

as easily make it suitable for HTML or any other document format by including the

necessary formatting code.

Define main()—part 2

With resproc() defined, we can continue with the handling of other results. For

example, we can now handle the results of tbstats()as follows:

tablestats = mydb.tbstats()

print "Table Statuses"

print resproc(tablestats)

print '\n\n'

just as easily as we handle the results of dbcreate():

dbcreation = mydb.dbcreate()

print "Database CREATE Statement"

print resproc(dbcreation)

print '\n\n'

There are, of course, other methods in the class we defined earlier. But these are done

as an introduction.

The preamble

Now that we have the class and functions written, we need to include some

introductory code to the head of the program. Beside the modules we use, we

also need to assign a few variables and incorporate some administrative code.

[ 389 ]Showing MySQL Metadata

Modules and variables

We obviously will need MySQLdb. However, the specification that we set ourselves

previously says the user should be able to designate the database. We therefore need

the sys module as well. The beginning of the program thus reads:

#!/usr/bin/env python

import sys

import MySQLdb

host = 'localhost'

user = 'skipper'

passwd = 'secret'

The database can then be set with sys.argv[] as:

db = sys.argv[1]

Login and USE

Next, we include the login and USE statements as part of a try...except structure.

If we cannot login or use the specified database, we want to fail softly.

try:

mydb = MySQLdb.connect(host, user, passwd)

cursor = mydb.cursor()

statement = "USE %s" %(db)

cursor.execute(statement)

except MySQLdb.Error, e:

print "There was a problem in accessing the database %s with the

credentials you provided. Please check the privileges of the user

account and retry. The error and other debugging information follow

below.\n\n%s" %(db, e)

Closing out the program

Finally, we need to check whether the program has been called directly. As usual, we

do this with the following if clause:

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

main()

Running the program will then produce a long report which can be ported to a file

using a pipe or similar shell convention.

[ 390 ]Chapter 13

Room to grow

In this project, we have seen how to create a class that returns metainformation about

a database. While it does what we set out to do, it does not fulfill the intent of that

specification. As an exercise, the following have been left to be done:

•Create the option of an output file

•Implement the availability of options using the optparse module, allowing

the user to specify which part of the metainformation should be returned

•

Code support for handling the results from the methods that are not called in

main()

•Build facility for retrieving the same metainformation from a remote server

as from a local one

•Implement the ability to get server variables and user information

Summary

In this chapter, we have covered how to build a Python class for retrieving MySQL

metainformation. We have seen:

•How to retrieve information on the engines used by MySQL

•Which system variables we can retrieve

•The several pieces of metadata about a given table that we can access

•How to retrieve user privileges and the grants used to give them

In the next chapter, we will look at one of the most important aspects of database

administration—disaster recovery.

[ 391 ]Disas