MySQL\_for\_Python\_Albert\_c10

regate Functions

and Clauses

It is not overstating things to say that MySQL queries can return a significant

amount of data. As we have seen with the SELECT statement, using clauses such as

WHERE reduces the number of hits that MySQL returns. Similarly, one can reduce

the amount of data returned by specifying the columns to be returned. Such

features save us from having to write our own algorithms to weed through the

data. However, even after narrowing the data, we sometimes still need to perform

calculations with the data for purposes such as tallying a sum, averaging out results,

or finding the range of a dataset. MySQL saves us this trouble as well, by providing

functions that return the answers to our questions without bothering us with the

data. These summative operations are called aggregate functions.

In this chapter, we will see how:

•MySQL saves us time and effort by pre-processing data

•To perform several calculations using MySQL's optimized algorithms

•To group and order returned data by column

The project for this chapter will once again build on the project from the earlier

chapters. After becoming more conversant with MySQL's aggregate functions and

learning how to use them in Python, we will add some aggregate functionality to the

web-based administration program.Aggregate Functions and Clauses

Calculations in MySQL

Letting the database server take the burden of calculations can reduce latency across

a network. Most web servers, for example, not only serve up web pages, but also

perform the processing for any scripts used to form those pages. If the script needs

to perform calculations of large amounts of data, the processing time of the server

will increase accordingly. Similar bottlenecks can occur on LANs and even desktop

systems that are normally optimized for user-based interactions, not heavy

data processing.

Computer clusters are another matter altogether. Depending on the size

of the cluster, one can lay much heavier loads on individual machines

and the cluster will be able to bear the burden. While using a cluster for

processing data takes the load off the database server, this can still result

in latency as one part of the processing machinery (the database) may

operate much slower than the rest of the parts.

MySQL provides several optimized ways of performing calculations on data. Their

functionality spans from statistics to bitwise operations, but the most commonly

used functions are:

•COUNT()

•SUM()

•MAX()

•MIN()

•AVG()

A listing of the less frequently used numeric functions can be found

at http://dev.mysql.com/doc/refman/5.5/en/group-by-

functions.html

Each of these functions is called by inserting the function's name and argument

immediately after SELECT. The function call stands in lieu of the column names to

be returned.

In the subsequent sections, we will be using the film table of the sakila database.

[ 280 ]Chapter 10

COUNT()

The COUNT() function returns the number of records affected. It does not return the

data itself. Its basic syntax is:

SELECT COUNT(<column name>) FROM <table name>;

In practice, it looks like this:

SELECT COUNT(title) FROM film;

This query returns the following results:

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

| COUNT(title) |

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

|

1000 |

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

In contrast, consider a table such as follows:

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

| ID | activity

| year |

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

|1 | NULL| 2010 |

|2 | learn piano| 2009 |

|3 | climb Kilimanjaro | 2008 |

|4 | learn Python| 2005 |

|5 | tour outerspace| NULL |

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

If we run a similar query against these five records, we get different results:

SELECT COUNT(activity) FROM diarytb;

The preceding statement returns the following results:

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

| COUNT(activity) |

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

|

4 |

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

1 row in set (0.00 sec)

[ 281 ]Aggregate Functions and Clauses

The COUNT() function returns a count of only non-NULL values by default. Similar

results would therefore be returned for the following:

SELECT COUNT(year) FROM diarytb;

To return both non-NULL and NULL values, we must use a universal quantifier:

SELECT COUNT(\*) FROM diarytb;

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

| COUNT(\*) |

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

|

5 |

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

In the case of the film table from sakila, no record was NULL. Therefore, the number

of returned records was the same as the total number of records in the table.

As you may notice, COUNT() does not care if the data is alphabetic

or numeric, it merely counts affected records. Of all of the functions

mentioned, COUNT(), MAX(), and MIN() are the only ones that can

handle string data.

SUM()

As the name suggests, SUM() returns the total of a series of numeric values. The basic

syntax is as follows:

SELECT SUM(<column name>) FROM <table name>;

SUM() only works on numeric values. If one passes string values to it, MySQL will

throw an error.

Using sakila again, we can ascertain the length of all films in the table with the

following query:

SELECT SUM(length) FROM film;

[ 282 ]Chapter 10

The result is the cumulative length in whatever unit is presumed for the column, in

this case, minutes:

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

| SUM(length) |

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

|

115272 |

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

MAX()

The MAX() function takes a column as its argument and returns the largest value of

the query results. The basic syntax of MAX() is as follows:

SELECT MAX(<column name>) FROM <table name>;

In practice, it looks like this:

SELECT MAX(length) FROM film;

Which returns the following results from sakila:

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

| MAX(length) |

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

|

185 |

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

Obviously, MAX() returns the highest number for numeric values. For string values,

however, it returns the highest index value of a set of strings.

mysql> SELECT MAX(title) FROM film;

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

| MAX(title) |

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

| ZORRO ARK

|

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

1 row in set (0.00 sec)

For information on how MySQL indexes strings, see the MySQL manual:

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

html

[ 283 ]Aggregate Functions and Clauses

If no records are affected, MAX() returns NULL.

mysql> SELECT MAX(title) FROM film WHERE title='CARMAGEDDON';

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

| MAX(title) |

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

| NULL

|

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

MIN()

The MIN() function is the obvious complement to MAX() and performs similarly to

it. As with MAX(), MIN() follows the SELECT keyword and takes a column name as

an argument:

SELECT MIN(<column name>) FROM <table name>;

In sakila, we can get the minimum running time as follows:

mysql> SELECT MIN(length) FROM film;

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

| MIN(length) |

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

|

46 |

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

As with MAX(), MIN() also accepts strings and returns the string with the lowest

index value:

mysql> SELECT MIN(title) FROM film;

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

| MIN(title)

|

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

| ACADEMY DINOSAUR |

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

AVG()

The AVG() function returns the mean of a numeric range. It follows the same format

as the other aggregate functions in its syntax:

SELECT AVG(<column name>) FROM <table name>;

[ 284 ]Chapter 10

The results of AVG() are calculated by dividing the sum of the resulting numeric

series by the number of records affected.

mysql> SELECT AVG(length) FROM film;

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

| AVG(length) |

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

|

115.2720 |

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

The different kinds of average

It is important to note what kind of average is meant by MySQL's AVG() function.

The English word 'average' can mean any of three things: mean, median, or mode.

Only mean is implied by MySQL's AVG().

Mean

Mean is calculated by dividing the sum of a numeric series by the number of items in

that series. For the film table of sakila, this is 115272 ÷ 1000, or 115.2720.

Median

Median is the middle item of a series (for example, the sixth item of a series of eleven

items). If the number of items is odd, the middle item is the median. If the number

of items is even, the median is the average of the two numbers in the middle of the

series. A simple example of how to determine the median of an odd number of items

is the following query:

SELECT <column> FROM <table> ORDER BY <column> DESC LIMIT <half the

total number of items>, 1;

To determine the median of an even number of items we limit the results by two:

SELECT <column> FROM <table> ORDER BY <column> DESC LIMIT <half the

total number of items>, 2;

and then take the mean of the two results.

[ 285 ]Aggregate Functions and Clauses

In the case of the film table of sakila, this query will return two instances of

the same value. Instead, one must broaden the range until a different number is

returned. Then one takes the mean of the two unique values.

mysql> select length from film order by length limit 500, 5;

+--------+

| length |

+--------+

|114 |

|114 |

|114 |

|114 |

|115 |

+--------+

The median is then (114 + 115) ÷ 2, or 114.5.

Mode

Mode is the item that occurs most frequently in a series, of which there may be none

or more than one. If all values in the series are unique, there is no mode. If more

than one value has the highest rate of occurrence, there is more than one mode. As a

result, modes are tricky to ascertain through MySQL itself. However, the following

will work in many circumstances:

SELECT COUNT(value) AS mode FROM table GROUP BY value ORDER BY mode

DESC LIMIT 1;

If no results are to be returned, AVG() returns NULL.

SELECT AVG(release\_year) FROM film WHERE title='The Osterman Weekend';

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

| AVG(release\_year) |

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

|

NULL |

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

[ 286 ]Chapter 10

Trimming results

MySQL allows for two different ways of trimming results when using aggregate

functions. One is a keyword that removes redundant data. The other is a function

that pools the data into a single string value.

DISTINCT

The purpose of DISTINCT is to ensure that all results are unique. The examples above

do not discriminate between duplicate values. Each value is treated as a separate

record without quantitative comparison to the others, thus allowing for redundancy

in the results.

This redundancy works well when comprehensiveness is required but can otherwise

skew the results of a query. For example, if we want to know how many unique

ratings are used in the film table of sakila, we would not want to use this query:

SELECT COUNT(rating) FROM film;

The results show every record that has a value for the rating column.

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

| COUNT(rating) |

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

|

1000 |

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

Instead of retrieving the actual values and process 1000 records in our program, we

can pass the burden onto the MySQL server by using the DISTINCT keyword:

SELECT COUNT(DISTINCT rating) FROM film;

This results in a much smaller return:

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

| COUNT(DISTINCT rating) |

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

|

5 |

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

[ 287 ]Aggregate Functions and Clauses

If we want to return the ratings themselves, we can use DISTINCT without the

COUNT() function:

SELECT DISTINCT rating FROM film;

The results will be as follows:

+--------+

| rating |

+--------+

| PG|

| G|

| NC-17|

| PG-13|

| R|

+--------+

By default, the results are returned in the order they appear in the table. To sort them

otherwise, use the ORDER BY function that is detailed later in this chapter.

Of the functions discussed earlier in this chapter, DISTINCT can be used with

the following:

•COUNT()

•MAX()

•MIN()

•AVG()

While the DISTINCT keyword can be used with MAX() and MIN(), the benefits are

negligible. The MAX() function finds the highest value of a dataset and MIN() the

lowest. As DISTINCT simply weeds out the duplicates, there is no significant benefit

to be had in using it. The net results are the same.

When using DISTINCT with AVG(), however, the results can be dramatically affected.

For example, the raw average (that is, mean) length of the records in film is returned

as follows:

mysql> SELECT AVG(length) FROM film;

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

| AVG(length) |

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

|

115.2720 |

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

[ 288 ]Chapter 10

However, if we sort out the duplicate lengths, we get a different value.

mysql> SELECT AVG(DISTINCT length) FROM film;

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

| AVG(DISTINCT length) |

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

|

115.5000 |

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

Consequently, it is important to know which value you are seeking and also to form

the query accordingly.

GROUP\_CONCAT()

The GROUP\_CONCAT() function collates all of the results into a single string value and

returns it. Its basic syntax is:

SELECT GROUP\_CONCAT(<column name>) FROM <table name>;

In practice, it looks like this:

SELECT GROUP\_CONCAT(length) FROM film;

The results of this query will be a lot of MySQL formatting that sandwiches a series

of values:

| 86,48,50,117,130,169,62,54,114,63,126,136,150,94,46,180,82,57,113,

79,129,85,92,181,74,86,179,91,168,82,92,119,153,62,147,127,121,68, 99,

148,137,170,170,113,83,108,153,118,162,182,75,173,87,113,65,129,90,122

,160,89,175,106,73,151,100,53,77,122,85,142,100,93,150,162,163,103,61,

85,114,148,103,71,50,102,63,121,76,179,63,63,98,72,121,176,123,169,56,

73,136,161,73,60,133,119,125,61,63,67,89,53,52,120,75,61,167,70,135,85

,176,92,151,114,85,163,61,179,112,183,179,110,152,114,117,51,70,146,66

,71,114,87,185,122,142,61,124,107,101,132,150,101,143,90,165,81,150,16

4,143,124,58,65,70,95,139,55,70,149,109,67,76, 120,59,112,65,180,122,8

7,172,115,173,184,166,185,112,92,146,64,57,136,139,172,143,50,153,104,

112,69,112,184,56,133,176,161,84,106,58,144,121,89,99,130,165,185,104,

59,57,113,120,122,51,106,100,64,76,81, 76,56,88,143,87,107,63,147,141,

94,68,100,47,57,122,120,125,68,49, 139,177,154,47,177,176,170,133,135,

61,159,175,178,110,96,116,132, 171,119,154,101,168,141,140,96,98,85,14

8,153,107,67,85,115,126,155,152,177,143,92,85,77,51,1 |

When interacting with MySQL from Python, this can be very helpful for getting all of

the results in one go. If you parse the results, it is important to note that MySQL does

not insert spaces into the results. Rather, each comma follows or precedes each

value immediately.

[ 289 ]Aggregate Functions and Clauses

Specifying the delimiter

If commas are not your delimiter of choice, GROUP\_CONCAT() allows you to specify

the delimiter by using the SEPARATOR keyword followed by the delimiter in quotes.

mysql> SELECT GROUP\_CONCAT(length SEPARATOR ':') FROM film;

This command returns the same values separated by colons instead of commas.

It is important to note that the delimiter only goes between values, so we

cannot use SEPARATOR to format the results for printing. No delimiter

would go before the first value.

The GROUP\_CONCAT() function further allows multi-character delimiters.

For example:

mysql> SELECT GROUP\_CONCAT(length SEPARATOR ' and ') FROM film;

The command returns the following:

| 86 and 48 and 50 and 117 and 130 and 169 and 62 and 54 and 114 and

63 and 126 and 136 and 150 and 94 and 46 and 180 and 82 and 57 and 113

and 79 and 129 and 85 and 92 and 181 and 74 and 86 and 179 and 91 and

168 and 82 and 92 and 119 and 153 and 62 and 147 and 127 and 121 and

68 and 99 and 148 and 137 and 170 and 170 and 113 and 83 and 108 and

153 and 118 and 162 and 182 and 75 and 173 and 87 and 113 and 65 and

129 and 90 and 122 and 160 and 89 and 175 and 106 and 73 and 151 and

100 and 53 and 77 and 122 and 85 and 142 and 100 and 93 and 150 and

162 and 163 and 103 and 61 and 85 and 114 and 148 and 103 and 71 and

50 and 102 and 63 and 121 and 76 and 179 and 63 and 63 and 98 and 72

and 121 and 176 and 123 and 169 and 56 and 73 and 136 and 161 and 73

and 60 and 133 and 119 and 125 and 61 and 63 and 67 and 89 and 53 and

52 and 120 and 75 and 61 and 167 and 70 and 135 and 85 and 176 and 92

and 151 and 114 and 85 and 163 and 61 and 179 and 112 and 183 and 179

and 110 and 152 and 114 and 117 and 51 and 70 and 146 a |

Oops! MySQL has cut off our results in midstream. By default, GROUP\_CONCAT()

returns a maximum length of 1024 characters. When the results hit that limit,

MySQL truncates the value.

Customizing the maximum length

When dealing with large amounts of results, a limit of 1024 can be very frustrating.

One way around this is to set the maximum length of GROUP\_CONCAT() to a higher

value for your MySQL session. We do this by passing the following code inside of a

MySQL shell:

SET SESSION group\_concat\_max\_len = 10240;

[ 290 ]Chapter 10

Now, we can pass the same query as before and get the results in their entirety. Any

value can be passed here, but this sets the length to ten times the default. Whatever

value we set, the amount of data transported is still bound by the local value for

max\_allowed\_packet, which defaults to 1 GB. On MySQL versions prior to 5.084,

however, we can set that variable, too, with this command:

SET SESSION max\_allowed\_packet = 10000;

As of MySQL 5.0.84, max\_allowed\_packet is available on a read-only basis; changes

must therefore be affected via a configuration file (or via a command-line option,

if available).

For more on MySQL system variables, see: http://dev.mysql.com/

doc/refman/5.0/en/server-system-variables.html

If you do not know which version of MySQL you are using or are unsure of the

relevant configuration, the following command will show whether your change has

taken effect:

SHOW VARIABLES LIKE 'max\_allowed\_packet';

If you are uncertain what a system variable is called, you can show all of them:

SHOW VARIABLES;

To differentiate between what variables pertain to the current session

and, which pertain to the entire database environment, use the

keywords SESSION and GLOBAL, respectively:

SHOW SESSION VARIABLES;

SHOW GLOBAL VARIABLES;

Using GROUP\_CONCAT() with DISTINCT

The GROUP\_CONCAT() function only concatenates results, it does not sort them. As a

result, we can wind up with a lot more data than we need. To refine the data further,

we can use DISTINCT. Compare the results of these two queries:

mysql> SELECT GROUP\_CONCAT(length) FROM film;

[ 291 ]Aggregate Functions and Clauses

The preceding command returns:

| 86,48,50,117,130,169,62,54,114,63,126,136,150,94,46,180,82,57,113,

79,129,85,92,181,74,86,179,91,168,82,92,119,153,62,147,127,121,68,

99,148,137,170,170,113,83,108,153,118,162,182,75,173,87,113,65,129,90,

122,160,89,175,106,73,151,100,53,77,122,85,142,100,93,150,162,163,103,

61,85,114,148,103,71,50,102,63,121,76,179,63,63,98,72,121,176,123,169,

56,73,136,161,73,60,133,119,125,61,63,67,89,53,52,120,75,61,167,70,135

,85,176,92,151,114,85,163,61,179,112,183,179,110,152,114,117,51,70,146

,66,71,114,87,185,122,142,61,124,107,101,132,150,101,143,90,165,81,150

,164,143,124,58,65,70,95,139,55,70,149,109,67,76, 120,59,112,65,180,12

2,87,172,115,173,184,166,185,112,92,146,64,57,136,139,172,143,50,153,1

04,112,69,112,184,56,133,176,161,84,106,58,144,121,89,99,130,165,185,1

04,59,57,113,120,122,51,106,100,64,76,81,76,56,88,143,87,107,63,147,14

1,94,68,100,47,57,122,120,125,68,49,139,177,154,47,177,176,170,133,135

,61,159,175,178,110,96,116,132,171,119,154,101,168,141,140,96,98,85,14

8,153,107,67,85,115,126,155,152,177,143,92,85,77,51,1 |

but

mysql> SELECT GROUP\_CONCAT(DISTINCT length) FROM film;

returns a concatenation of only the unique values, amounting to less than half

the length:

| 86,48,50,117,130,169,62,54,114,63,126,136,150,94,46,180,82,57,113,

79,129,85,92,181,74,179,91,168,119,153,147,127,121,68,99,148,137,170,8

3,108,118,162,182,75,173,87,65,90,122,160,89,175,106,73,151,100,53,77,

142,93,163,103,61,71,102,76,98,72,176,123,56,161,60,133,125,67,52,120,

167,70,135,112,183,110,152,51,146,66,185,124,107,101,132, 143,165,81,1

64,58,95,139,55,149,109,59,172,115,184,166,64,104,69,84,144,88,141,47,

49,177,154,159,178,96,116,171,140,155,158,174,138,97,131,156,80,145,11

1,128,157,78,105,134 |

To further sort the results of GROUP\_CONCAT(), we need to indicate the order by

which to sort the results. For this, we turn to server-side sorting in the next section.

Server-side sorting in MySQL

When sorting results, MySQL allows for two types of ordering: by group and by

item. Below, we look at each in turn in terms of their respective clauses:

•GROUP BY

•ORDER BY

Despite their syntactical similarities, each has its distinct applications and limitations.

[ 292 ]Chapter 10

GROUP BY

The GROUP BY clause is used to organize results according to the structure of the data

returned. The clause itself is appended to the end of the SELECT statement. The basic

syntax is:

SELECT <column name(s)> FROM <table name> GROUP BY <column name as

key>;

As a rule, GROUP BY cannot be used in conjunction with a universal quantifier ('\*')

instead of the column name(s). Rather, the column used as the key for sorting must

be stated among the column names indicated for the query.

Using the world database, we can ascertain what the database records as the official

language of each country in the world, with the following query:

SELECT CountryCode, Language FROM CountryLanguage WHERE IsOfficial='T'

GROUP BY CountryCode;

The results are indexed alphabetically according to the first column retrieved. As one

might imagine, they run quite long. If we wanted to trim the results to the language

of German, we can use a WHERE clause:

SELECT CountryCode, Language, Percentage FROM CountryLanguage WHERE

Language LIKE 'German' AND IsOfficial='T' GROUP BY CountryCode;

The results will be:

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

| CountryCode | Language | Percentage |

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

| AUT| German|92.0 |

| BEL| German|1.0 |

| CHE| German|63.6 |

| DEU| German|91.3 |

| LIE| German|89.0 |

| LUX| German|2.3 |

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

Using the sakila database, we can use GROUP BY to ascertain how long most films

are borrowed. The rental\_duration for any film is between three and seven

days. Therefore, we key the sorting to that column and count the number of

corresponding records.

SELECT rental\_duration, COUNT(title) FROM film GROUP BY rental\_

duration;

[ 293 ]Aggregate Functions and Clauses

The result is:

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

| rental\_duration | COUNT(title) |

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

|3 |203 |

|4 |203 |

|5 |191 |

|6 |212 |

|7 |191 |

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

We can tell from these results that most films are returned within four days. If we

had used the AVG() function, however, we would have received different results:

mysql> SELECT AVG(rental\_duration) FROM film;

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

| AVG(rental\_duration) |

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

|

4.9850 |

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

Therefore, we can see (again) that the mean is not always the best average to take.

ORDER BY

Unlike GROUP BY, the ORDER BY clause causes no categorization whatsoever. Rather, it

strictly sorts the records returned according to an indicated pattern. Where the 1000

titles of film will result in a single row if grouped by release\_year, the ORDER BY

clause will return 1000 records.

The basic syntax for this clause is:

SELECT <column name> FROM <table name> ORDER BY <column name>;

Using a universal quantifier

The ORDER BY clause also differs from GROUP BY in that it works with universal

quantification of the columns selected. So rather than specify the columns to be

returned, one can simply use the asterisk; however, one must still use a valid column

name as a key for the sorting. So to adopt the GROUP BY example, we can use:

SELECT rental\_duration, title FROM film ORDER BY rental\_duration;

[ 294 ]Chapter 10

We can get the rental duration and title from film sorted according to increased

rental periods. In addition, we can return all columns with the following:

SELECT \* FROM film ORDER BY rental\_duration;

Sorting alphabetically or from low-to-high

MySQL treats strings according to their index equivalents. Therefore, alphabetic

sorting is the same as sorting in ascending order. To do so, we use the ASC keyword

immediately after the column to be used as a key. The following returns the results

in alphabetical order by title:

SELECT title, rating, length FROM film WHERE title LIKE 'Y%' ORDER BY

title ASC;

To keep all the results on screen, this example uses a WHERE clause to reduce the

relevant results to film titles that begin with 'Y'. The result of this query is:

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

| title

| rating | length |

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

| YENTL IDAHO

| R

|

86 |

| YOUNG LANGUAGE | G

|

183 |

| YOUTH KICK

| NC-17 |

179 |

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

If we trade out the WHERE clause for a LIMIT clause at the end of the statement, we

can learn the top ten of a series very quickly. This query returns the ten shortest films

in the sakila database:

SELECT title, rating, length FROM film ORDER BY length ASC LIMIT 10;

The results are as follows:

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

| title

| rating | length |

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

| ALIEN CENTER

| NC-17 |

46 |

| IRON MOON

| PG

|

46 |

| KWAI HOMEWARD

| PG-13 |

46 |

| LABYRINTH LEAGUE

| PG-13 |

46 |

| RIDGEMONT SUBMARINE | PG-13 |

46 |

| DIVORCE SHINING

| G

|

47 |

| DOWNHILL ENOUGH

| G

|

47 |

| HALLOWEEN NUTS

| PG-13 |

47 |

| HANOVER GALAXY

| NC-17 |

47 |

| HAWK CHILL

| PG-13 |

47 |

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

[ 295 ]Aggregate Functions and Clauses

Reversing the alphabet or sorting high-to-low

Where ASC overtly indicates that sorting should be in ascending order, we can

reverse that sort with the DESC keyword. Adapting the last example, we can

ascertain the twenty longest films in the sakila database with the following query:

SELECT title, rating, length FROM film ORDER BY length DESC LIMIT 20;

The results are as follows:

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

| title

| rating | length |

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

| CHICAGO NORTH| PG-13|185 |

| CONTROL ANTHEM| G|185 |

| DARN FORRESTER| G|185 |

| GANGS PRIDE| PG-13|185 |

| HOME PITY| R|185 |

| MUSCLE BRIGHT| G|185 |

| POND SEATTLE| PG-13|185 |

| SOLDIERS EVOLUTION | R|185 |

| SWEET BROTHERHOOD| R|185 |

| WORST BANGER| PG|185 |

| CONSPIRACY SPIRIT| PG-13|184 |

| CRYSTAL BREAKING| NC-17|184 |

| KING EVOLUTION| NC-17|184 |

| MOONWALKER FOOL| G|184 |

| SMOOCHY CONTROL| R|184 |

| SONS INTERVIEW| NC-17|184 |

| SORORITY QUEEN| NC-17|184 |

| THEORY MERMAID| PG-13|184 |

| CATCH AMISTAD| G|183 |

| FRONTIER CABIN| PG-13|183 |

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

[ 296 ]Chapter 10

You will notice that all of the ten longest films have the same length. To

determine the twenty longest lengths in the database, we would need to

use a GROUP BY clause in conjunction with GROUP\_CONCAT:

SELECT GROUP\_CONCAT(title), rating, GROUP\_

CONCAT(DISTINCT length) FROM film GROUP BY length DESC

LIMIT 20;

The ORDER BY clause cannot be used in conjunction with the calculating

and concatenating functions.

Like sorting in ascending order, sorting in descending order uses index values for

strings. The index value of certain strings, however, will be determined by how the

table is created. For example, if we run this query against sakila:

SELECT title, rating, length FROM film WHERE title LIKE 'WO%' ORDER BY

rating ASC;

We get all film titles that begin with WO sorted by rating, supposedly in ascending

order. However, it is not in that order. The results are as follows:

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

| title

| rating | length |

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

| WORST BANGER| PG|185 |

| WON DARES| PG|105 |

| WONDERLAND CHRISTMAS | PG|111 |

| WORDS HUNTER| PG|116 |

| WORLD LEATHERNECKS| PG-13|171 |

| WOMEN DORADO| R|126 |

| WORKING MICROCOSMOS| R|74 |

| WORKER TARZAN| R|139 |

| WONKA SEA| NC-17|85 |

| WONDERFUL DROP| NC-17|126 |

| WOLVES DESIRE| NC-17|55 |

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

The NC-17 ratings follow after, not before the others. The reason for this is that

the rating column is of type enum. This is a type of string for which the values are

enumerated at the time the table is created and the column specified. The index value

for such types follows the order in which the options were specified. In the case of

film, we can learn the sequence from the description. Using DESCRIBE FILM, the data

type for rating is revealed to be:

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

[ 297 ]Aggregate Functions and Clauses

Therefore, MySQL's sorting of the ratings column will always be according to

this order.

Sorting with multiple keys

The ORDER BY clause can be used with multiple keys to provide a multi-tiered

sorting process.

SELECT rating, title, length FROM film ORDER BY rating, length;

It returns all columns of film sorted first by rating then by title.

To use the ASC and DESC keywords, we simply place them after the relevant key. So

rating can be in ascending order and length in descending with the following query:

SELECT rating, title, length FROM film ORDER BY rating ASC, length

DESC;

Putting it in Python

As with most programming languages, the more one knows about the data and

data structure, the better one can program to handle it. This is at least doubly true

for database programming with aggregate functions and clauses such as those which

we cover in this chapter.

Putting the SELECT statement into Python is not particularly complex, but handling

the results intelligently requires a knowledge of their format. Again, setting up a

basic database session in Python would look like this:

#!/usr/bin/env python

import MySQLdb

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

cursor = mydb.cursor()

For the statement to be run against sakila, we can use the following:

statement = """SELECT \* FROM film WHERE title LIKE 'Z%'

ORDER BY rating ASC"""

Then we execute it:

runit = cursor.execute(statement)

fetch = cursor.fetchall()

[ 298 ]Chapter 10

If, at this point, we just print the value of fetch, we get a lot of a database formatting:

print fetch

((999, 'ZOOLANDER FICTION', 'A Fateful Reflection of a Waitress

And a Boat who must Discover a Sumo Wrestler in Ancient China',

2006, 1, None, 5, Decimal("2.99"), 101, Decimal("28.99"), 'R',

'Trailers,Deleted Scenes', datetime.datetime(2006, 2, 15, 5, 3,

42)), (998, 'ZHIVAGO CORE', 'A Fateful Yarn of a Composer And a Man

who must Face a Boy in The Canadian Rockies', 2006, 1, None, 6,

Decimal("0.99"), 105, Decimal("10.99"), 'NC-17', 'Deleted Scenes',

datetime.datetime(2006, 2, 15, 5, 3, 42)), (1000, 'ZORRO ARK', 'A

Intrepid Panorama of a Mad Scientist And a Boy who must Redeem

a Boy in A Monastery', 2006, 1, None, 3, Decimal("4.99"), 50,

Decimal("18.99"), 'NC-17', 'Trailers,Commentaries,Behind the Scenes',

datetime.datetime(2006, 2, 15, 5, 3, 42)))

Obviously, this is less than ideal. If we did not know the format of the data, we

would be left to abstract the data handling. This would typically result in verbose

and inefficient code.

Ideally, we should modify our SELECT statement to return only the data that we

want. In this case, we are only interested in the rating, title, and length.

So we should change the SELECT statement accordingly:

statement = """SELECT rating, title, length FROM film

WHERE title LIKE 'Z%'

ORDER BY rating ASC"""

This gives us more managable data. Each record is now three fields long. However,

the results are still obtuse; being formatted with parentheses and commas obfuscates

the data. Better then to use a for loop to prepare the data:

for i in fetch:

print i[0],"\t", i[2],"\t", i[1]

While we are at it, we can preface that for a loop with a print statement that reflects

the column headers:

print "Rating \tLength \tTitle"

The result is easier on the eyes and therefore more user-friendly:

RatingLengthTitle

R101ZOOLANDER FICTION

NC-17105ZHIVAGO CORE

NC-1750ZORRO ARK

[ 299 ]Aggregate Functions and Clauses

Project: Incorporating aggregate

functions

Building support for aggregate functions into our web application requires some

revision of code that we have already written. As shown in this chapter, all aggregate

functions work with the SELECT keyword. We will thus need to change how we

support that kind of query. By the end of this section, we will build the following

functionality into the web application:

•Support for all calculating functions

•Support for use of DISTINCT in conjunction with calculating functions

•Allowance for sorting using either ORDER BY or GROUP BY

•Return results in a tabular format

The order of development when revising a project should be inside out. Revise the

relevant function and develop outward, through the main() function. After we code

the initial variable assignments, we can then move on to revise the web interface to

support the new functionality. The virtue of this process is that we do not introduce

variables without support—to do so would seed security and stability issues.

Adding to qaction()

As all of our new functionality deals with the SELECT statement, we will need to

revisit the qaction() function of the application. Currently, the code looks like this:

def qaction(qact, db, tb, columns, values, user, password):

"""Forms SELECT and INSERT statements, passes them to execute(),

and returns the affected rows."""

cursor = connection(user, password, db)

tname = tb + "("

columns = columns.split(',')

values = values.split(',')

cols = ""

vals = ""

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

col = columns[i].strip()

val = values[i].strip()

cols = cols + col

vals = vals + "'" + val + "'"

if i != len(columns)-1:

cols = cols + ", "

[ 300 ]Chapter 10

vals = vals + ", "

if qact == "select":

statement = "SELECT \* FROM %s WHERE %s = %s" %(tb, cols, vals)

results = execute(statement, cursor, 'select')

elif qact == "insert":

statement = "INSERT INTO %s (%s) VALUES (%s)" %(tb, cols,

vals)

results = execute(statement, cursor, 'insert')

return results

If you find seven arguments to be too much, you can use

\*\* args instead.

To introduce additional support in this function, we need to add new variables and

instructions for how to handle them without losing the original SELECT functionality.

New variables

For the calculating functions, we will use a new variable calc. As all calculating

functions require a columnar key on which to operate, we will also use a variable

colkey. So every calculation will have this format:

calc(colkey)

The value of calc will be allowed to be any one of the calculating functions. In the

HTML form, we can specify them to be uppercase so they are easier to read in the

resulting SQL statement. We could use str.upper(), but it is better to exploit our

control over the program input by structuring the HTML according to what we

ultimately need.

The use of DISTINCT is essentially a switch. It takes no arguments per se, but is

either used or not. Therefore, we can use a variable distinct as a boolean, in effect.

In the HTML, we can support the value as either yes or no. If distinct is yes, the

calculation function would read:

calc(DISTINCT colkey).

If no, we will pass over it (as addressed further).

Finally, support for sorting is provided by sort. Naturally, if we are going to sort,

we need to indicate the column by which sorting should take place. That value is

represented by variable key. So the sorting clause will read in the format.

sort key

[ 301 ]Aggregate Functions and Clauses

To pass these values into the function, we could merely append them to the

function definition line (again, using \*\*args could be used in lieu of our

arguments here):

def qaction(qact, db, tb, columns, values, user, password, calc,

colkey, distinct, sort, key):

But that gets unwieldy. Another option, and one that could easily be employed on

the other arguments to this and the other functions, is to use a convention reserved

for variable-length arguments. If we say that all variables related to aggregate

functions can be part of a tuple called aggregates, we can use the following:

def qaction(qact, db, tb, columns, values, user, password,

\*aggregates):

For purposes of illustration, we will use the variable-length convention as shown

here. However, how variable you make the length of the arguments is your choice.

We can then assign those variables in the function:

calc = aggregates[0]

colkey = aggregates[1]

distinct = aggregates[2]

sort = aggregates[3]

key = aggregates[4]

By default, we will plan on all variables equating to the string NONE. This is purely for

the sake of pattern matching and could alternatively be dealt with by testing the value

for equating to None or for setting up a test for it being True. A further alternative is to

use try...except structures defaulting to NONE or a similar string.

New statement formation

The new functionality needs to support new options in two different places.

Currently, the SELECT statement is formed according to this template:

SELECT \* FROM <table> WHERE <column> = '<value>'

However, the calculating functions need to go where the asterisk is currently located,

and the sorting clause needs to follow on the end of the statement. We should

therefore be able to support both of the following options:

SELECT <calculating function> FROM <table> WHERE <column> =

'<value>'

SELECT \* FROM <table> WHERE <column> = '<value>' <sorting

[ 302 ]

clause>Chapter 10

To add these parts, we will use a series of if...else clauses.

The only time that we want to change the original statement is if we have new

variables to handle. Better is to test for a positive rather than a series of negatives.

However, our program needs do not allow for that easily. In which case, we might

set up a conditional structure as follows:

if calc != "NONE" or distinct != "NONE" or sort != "NONE" or

key != "NONE":

<form one of the new statements>

else:

statement = "SELECT \* FROM %s WHERE %s = %s" %(tb, cols,

vals)

So we default to the formation of the original statement value but allow for the

creation of the alternatives if any of our new variables have been set.

We then need to build in ways of handling the variables and forming the new

statements within the if clause of the explained structure. For the specification we

have set ourselves, this will fall into two inner if...else structures.

Each will handle the value of one option in the alternative statements discussed. At

the end of the outer if clause, the results of the inner if...else structures will be

combined as building blocks to form the statement.

The first if...else structure will quantify any calculating functions and determine

whether DISTINCT should be included. For this first variable part of the SELECT

statement, we will create a new string called selection. It will hold either an

asterisk, as per the original statement, or the verbage for the calculating function.

The code reads as follows:

if calc != "NONE":

if distinct == "yes":

selection = "%s(DISTINCT %s)" %(calc, colkey)

else:

selection = "%s(%s)" %(calc, colkey)

else:

selection = "\*"

Recall that, by design, calc can only equate to one of the calculating functions or to

NONE. If it equates to NONE, we default to the universally-quantifying asterisk.

If calc is not NONE, it must equate to one of the calculating functions. We then test

the value of distinct. If distinct is yes, we account for that in the assignment of

selection. Otherwise, we omit it.

[ 303 ]Aggregate Functions and Clauses

The second if...else structure determines the ordering and grouping by the value

of sort. The resulting building block is called sorting.

if sort != "NONE":

sorting = "%s %s" %(sort, key)

else:

sorting = ""

In the HTML of the page, we will hardwire sort to equate to either ORDER BY or GROUP

BY. This value is then added to the MySQL statement formed by the program.

It is worth noting that these values should not go unchallenged in real-life

deployments. We do not do it here for the sake of space. However, one

should never execute code directly from a public-facing web page because

of the inherent insecurity of such an implementation. For more on this,

see Wikipedia's articles or similar ones on SQL injection.

Once the two building blocks, and the selection and sorting are formed, we can

then combine them together with the rest of the SELECT statement template. The

assignment line reads as follows:

statement = "SELECT %s FROM %s WHERE %s = %s %s"

%(selection, tb, cols, vals, sorting)

With that completed, the resulting statement is again passed to execute() and the

affected rows assigned to results, which is subsequently returned by the function.

After these changes, the code for qaction() reads as follows:

def qaction(qact, db, tb, columns, values, user, password,

\*aggregates):

"""Forms SELECT and INSERT statements, passes them to execute(),

and returns the affected rows."""

cursor = connection(user, password, db)

calc = aggregates[0]

colkey = aggregates[1]

distinct = aggregates[2]

sort = aggregates[3]

key = aggregates[4]

tname = tb + "("

columns = columns.split(',')

values = values.split(',')

cols = ""

vals = ""

[ 304 ]Chapter 10

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

col = columns[i].strip()

val = values[i].strip()

cols = cols + col

vals = vals + "'" + val + "'"

if i != len(columns) - 1:

cols = cols + ", "

vals = vals + ", "

if qact == "select":

if calc != "NONE" or distinct != "NONE" or sort != "NONE" or

key != "NONE":

if calc != "NONE":

if distinct == "yes":

selection = "%s(DISTINCT %s)" %(calc, colkey)

else:

selection = "%s(%s)" %(calc, colkey)

else:

selection = "\*"

if sort != "NONE":

sorting = "%s %s" %(sort, key)

else:

sorting = ""

statement = "SELECT %s FROM %s WHERE %s = %s %s"

%(selection, tb, cols, vals, sorting)

else:

statement = "SELECT \* FROM %s WHERE %s = %s" %(tb, cols,

vals)

results = execute(statement, cursor, 'select')

elif qact == "insert":

statement = "INSERT INTO %s (%s) VALUES (%s)" %(tb, cols,

vals)

results = execute(statement, cursor, 'insert')

return results

Revising main()

Now that the functionality is built into qaction(), we can ammend main()

accordingly. At present, main() reads as follows:

def main():

"""The main function creates and controls the MySQLStatement

instance in accordance with the user's input."""

output = ""

[ 305 ]Aggregate Functions and Clauses

while 1:

try:

cursor = connectNoDB(opt.user, opt.password)

authenticate = 1

except:

output = "Bad login information. Please verify the

username and password that you are using before trying to login

again."

authenticate = 0

if authenticate == 1:

errmsg = "You have not specified the information necessary

for the action you chose. Please check your information and specify

it correctly in the dialogue."

if opt.dbact is not None:

output = dbaction(opt.dbact, opt.dbname, cursor)

elif opt.tbact is not None:

output = tbaction(opt.tbact, opt.tbdbname, opt.tbname,

opt.columns, opt.values, opt.user, opt.password)

elif opt.qact is not None:

output = qaction(opt.qact, opt.qdbname, opt.qtbname,

opt.columns, opt.values, opt.user, opt.password)

elif opt.uact is not None:

if opt.uact == "create":

act = "create-user"

output = uaction(opt.user, opt.password, act, opt.

username, opt.passwd)

elif opt.uact == "drop":

act = "drop-user"

output = uaction(opt.user, opt.password, act, opt.

username)

elif opt.uact == "grant" or opt.uact == "revoke":

output = uadmin(opt.user, opt.password, opt.uact,

opt.username, opt.privileges, opt.acldb, opt.acltb)

else:

output = errmsg

printout = HTMLPage()

printout.message(output)

output = printout.page()

print output

break

[ 306 ]Chapter 10

The part of main() that needs changing, and the part in which the code of this

section will be placed, is the second elif clause:

elif opt.qact is not None:

output = qaction(opt.qact, opt.qdbname, opt.qtbname,

opt.columns, opt.values, opt.user, opt.password)

Into this clause, we will quantify each of the new variables. We will further assign

complementary values, such as colkey, based on calc also being indicated by the

user. The code reads as follows:

elif opt.qact is not None:

if opt.calc is not None:

calc = opt.calc

colkey = opt.colkey

else:

calc = "NONE"

colkey = "NONE"

if opt.distinct is not None:

distinct = opt.distinct

else:

distinct = "NONE"

if opt.sort is not None:

sort = opt.sort

key = opt.key

else:

sort = "NONE"

key = "NONE"

output = qaction(opt.qact, opt.qdbname, opt.qtbname,

opt.columns, opt.values, opt.user, opt.password, calc, colkey,

distinct, sort, key)

For safer computing, we could put the main block of this clause into a try...except

clause. Otherwise, we leave ourselves open to possible errors if the user omits

important data.

Finally, we pass the data to qaction() and assign the results to output. You will

notice that we pass the variables overtly. Because the aggregates argument of

qaction() is a tuple, we could also pass all of the added values (from calc

onward) as a tuple. The choice is yours, but Python can handle either.

As before, the resulting output is passed to HTMLPage.message() later. The

resulting HTML page is then printed as output.

[ 307 ]Aggregate Functions and Clauses

Setting up the options

Having set up the new functionality, we next need to provide support for it in the

options of the program. If you coded this application as a CGI program, the values

will be available automatically as part of cgi.FieldStorage(). In PHP, however,

we need to add each option as a flag. The code is as follows:

opt.add\_option("-C", "--calculate",

action="store",

type="string",

help="which calculating function to employ",

dest="calc")

opt.add\_option("-K", "--colkey",

action="store",

type="string",

help="column to use when calculating",

dest="colkey")

opt.add\_option("-I", "--distinct",

action="store",

type="string",

help="whether to return distinct results",

dest="distinct")

opt.add\_option("-S", "--sort",

action="store",

type="string",

help="how to sort results",

dest="sort")

opt.add\_option("-k", "--key",

action="store",

type="string",

help="key to use when sorting",

dest="key")

We can then call the program from the command line to test the HTML output. If

the program is called pymyadmin.py, we can call it as follows (changing the syntax

according to your operating system):

./pymyadmin.py -U 'skipper' -P 'secret' -q 'select' -Z sakila -Y film

-c length -v '114' -C 'COUNT' -K title

Here, HTML will output the following to the screen:

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN" "http://

www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"

dir="ltr">

<head>

<title>PyMyAdmin 0.001</title>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

</head>

[ 308 ]Chapter 10

<body>

<h1>PyMyAdmin Results</h1>

<br> 10 <br>

<br>

</body>

</html>

This is the same output that will be passed to Apache. You can test it by exporting

the output to an output file. However, if you used CGI for this application, you

would need to hard-wire the values into the code temporarily.

Changing the HTML form

As with the Python program, the HTML page will support the aggregate functions

within the section for queries. That section of the form currently reads:

<div>QUERIES</div>

<input type="radio" name="qact" value="select"> SELECT

<br>

<input type="radio" name="qact" value="insert"> INSERT

<br>

Database name: <input type="text" name="qdbname" value="">

<br>

Table name: <input type="text" name="qtbname" value="">

<br>

Columns (comma-separated): <input type="text" name="columns" value="">

<br>

Values (comma-separated): <input type="text" name="values" value="">

<br><br>

We need to change it to provide support for the functionality we have introduced

into the Python program.

As mentioned previously, the values of calc will be hard-wired into the HTML.

With the determination of that value, we will also include a text field in which the

user is to specify the value of colkey. The code follows the field for Values

and reads:

Calculations: <select name="calc">

<option value="COUNT">COUNT</option>

<option value="SUM">SUM</option>

<option value="MAX">MAX</option>

<option value="MIN">MIN</option>

<option value="AVG">AVG</option>

</select>

(<input type="text" name="colkey" value="">)

<br>

[ 309 ]Aggregate Functions and Clauses

You may note that the dialogue for colkey is sandwiched between parentheses. The

visual effect of this code is that the user will see the SQL formatting in the HTML.

Following this, we offer the user the ability to indicate the value of distinct.

DISTINCT?

<input type="radio" name="distinct" value="yes">Yes

<input type="radio" name="distinct" value="no">No

<br>

Finally, we need to provide the facilities for sorting. With the choice of whether to

order by item or by group, we also provide a text field for the key by which to sort

the values.

Sorting: <select name="sort">

<option value="ORDER BY">ORDER BY</option>

<option value="GROUP BY">GROUP BY</option>

</select>

<input type="text" name="key" value="">

<br><br>

Summary

In this chapter, we have covered how to use aggregate functions and clauses in

Python. We have seen:

•How to use MySQL's optimized functions to calculate certain aspects of

data results

•The ways that MySQL provides for sorting the data before it is returned

•How to tell MySQL render unique values

•Some of the ways that discrete values return different results than

non-discrete ones

•How we can instruct MySQL to return alternative averages to the mean and

its default for the function AVG

In the next chapter, we will see how to create joins (using JOIN) in MySQL

for Python.

[ 310 ]SELE