MySQL\_for\_Python\_Albert\_c11

ECT Alternatives

We have already seen how to restrict the data that MySQL returns by using a WHERE

clause. Rather than retrieve the entire table and sort through it in Python, we passed

the burden onto the server. Using WHERE restricted the number of rows that match.

This is just like when we specify columns instead of using an asterisk after

SELECT—it saves us from receiving the entire record for every match.

WHERE causes MySQL to ignore any row that does not match our selection. Specifying

the columns then indicates to MySQL, which parts of the affected rows to return. In

addition to WHERE, MySQL supports other ways of narrowing one's returns. It also

allows us to match and complement the data using other tables, including combining

tables or results from different queries.

In this chapter, we will see:

•How we can restrict results using WHERE and HAVING, and what the

differences are between them

•When it is best to use WHERE and when one might use HAVING

•How to create temporary subtables from which to SELECT data

•How to join the results of two SELECT statements with UNION

•The various ways to JOIN tables

•The difference between an INNER JOIN and and OUTER JOIN, whether LEFT

or RIGHT

The project for this chapter will again be built on the web-based database

administration program that we have developed in preceding chapters. After seeing

new ways to restrict and complement the data that we retrieve from a table, we will

add some of that functionality into our application.SELECT Alternatives

HAVING clause

The HAVING clause has similar effect to the WHERE clause. The syntax is virtually

the same:

SELECT <some column(s)> FROM <table> HAVING <met a certain condition>;

Indeed, in some statements, one can be tempted to replace WHERE with HAVING

because the syntax is so similar, and sometimes one would not notice much, if

any, difference in the returned data. However, each has its purpose and is applied

differently, as discussed later in this chapter.

WHERE versus HAVING: Syntax

The HAVING clause can only be applied to columns that have been previously

indicated in the statement. For example, using the sakila database, let's say we

wanted every record from film that was updated since 2005. Trying to hack from

our knowledge of WHERE, we might try the following:

SELECT title FROM film HAVING YEAR(last\_update) > '2005';

But we would be wrong and would be greeted with an error:

ERROR 1054 (42S22): Unknown column 'last\_update' in 'having clause'

One way of resolving the problem is by using a universal quantifier:

SELECT \* FROM film HAVING YEAR(last\_update) > '2005';

The returned data will be the entire table because all of the records in the film were

updated last in 2006. This statement works because last\_update is implied in the

quantifying asterisk. However, it is far from ideal and leaves it to Python to sort

through the data. It is better to use this statement:

SELECT title, last\_update FROM film HAVING YEAR(last\_update) > '2005';

This statement will still return every record by giving us the information we want

and proof of the match.

WHERE versus HAVING: Aggregate functions

A common situation in which WHERE and HAVING differ in syntax, is in the use

of aggregate functions. As we have seen, a WHERE clause such as the following

is allowed:

SELECT title, MAX(film\_id), AVG(length), last\_update FROM film WHERE

length > '100' GROUP BY YEAR(last\_update);

[ 312 ]Chapter 11

Because all records in film were last updated in the same year, this query will return

exactly one row holding—the title of the first affected record (that is, of films with

length greater than 100 minutes), the highest ID number of the affected rows, the

average length of the same, and the timestamp for the last update of the first hit.

To use HAVING in a context such as this requires a different syntax. When using

HAVING with aggregate functions, it must be placed in the GROUP BY structure. The

precise reasons for this are addressed more fully in the next section, but the concise

cause is that HAVING is applied after aggregation is performed. So for example, to

get the same information for every title that begins with X, Y, or Z, we could use

the following:

SELECT title, MAX(film\_id), AVG(length), last\_update FROM film GROUP

BY title HAVING title > 'X';;

The results of this are as follows:

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

| title

| MAX(film\_id) | AVG(length) | last\_update

|

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

| YENTL IDAHO|995 |86.0000 | 2006-02-15 05:03:42 |

| YOUNG LANGUAGE|996 |183.0000 | 2006-02-15 05:03:42 |

| YOUTH KICK|997 |179.0000 | 2006-02-15 05:03:42 |

| ZHIVAGO CORE|998 |105.0000 | 2006-02-15 05:03:42 |

| ZOOLANDER FICTION |999 |101.0000 | 2006-02-15 05:03:42 |

| ZORRO ARK1000 |50.0000 | 2006-02-15 05:03:42 |

|

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

However, achieving the same results as the WHERE clause can give us further required

adaptations. As mentioned above, HAVING requires the column used in its condition

to have been mentioned previously in the SELECT statement. But the columns passed

as arguments to calculating functions do not count as having been mentioned

previously. In effect, MySQL does not see them when it evaluates a HAVING clause.

Rather, it just sees their results and finds no match. Therefore, to use those results

with a HAVING clause, we need to give them a name using AS. The last example could

be rewritten for ease of reading as follows:

SELECT title, MAX(film\_id) AS max, AVG(length) AS avg, last\_update

FROM film GROUP BY title HAVING title > 'X';

[ 313 ]SELECT Alternatives

And the headings of the results would be changed accordingly as:

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

| title

| max

| avg

| last\_update

|

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

| YENTL IDAHO|995 |

| YOUNG LANGUAGE|996 | 183.0000 | 2006-02-15 05:03:42 |

| YOUTH KICK|997 | 179.0000 | 2006-02-15 05:03:42 |

| ZHIVAGO CORE|998 | 105.0000 | 2006-02-15 05:03:42 |

| ZOOLANDER FICTION |999 | 101.0000 | 2006-02-15 05:03:42 |

| ZORRO ARK

| 1000 |

86.0000 | 2006-02-15 05:03:42 |

50.0000 | 2006-02-15 05:03:42 |

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

Using AS, we can achieve results similar to the WHERE clause that started this section.

Let's say, however, that we wanted to group by title, not the year of the last update.

The WHERE version would look like this:

SELECT title, MAX(film\_id), AVG(length), last\_update FROM film WHERE

length > '100' GROUP BY title;

But the HAVING version cannot merely switch the order of the clauses around

because, in the view of MySQL, the column length is never referenced as a column

in the SELECT statement. Rather, it is an argument to AVG(). Therefore, we need to

name the results of AVG() as length.

SELECT title, MAX(film\_id), AVG(length) AS length, last\_update FROM

film GROUP BY title HAVING length > '100';;

Note that HAVING requires the column of its condition to have been used

earlier in the clause, but GROUP BY does not.

To be sure, the use of AVG here is unnecessary. As we are grouping by a unique

quality, title, the average length will be the value initially passed to AVG(). The

average of a single value is that value. Consequently, the WHERE clause would be the

best version to use in this case. In the next section, we will look at when one should

use HAVING instead of WHERE.

WHERE versus HAVING: Application

The use of HAVING differs from WHERE in more than its syntax. However, the two

clauses also differ in how they are applied.

[ 314 ]Chapter 11

Whenever you pass a SELECT statement to MySQL, it forms a table of results which

is then optimized (that is, narrowed), based on the criteria passed to it. The condition

represented in WHERE is applied before the optimizations are carried out. But the

condition in HAVING is applied after the optimizations are performed, being affected

as a final filter for what should be reported back. The difference is subtle, but boils

down to application before the results are finalized (WHERE) and application after the

results table is created (HAVING). A further example may help to clarify the difference.

Suppose we wanted to search the Country table of the world database to find which

forms of government had the largest populations. We then quantify the population

size by those in excess of 100 million people. A basic, unquantified query might look

like this:

SELECT AVG(Population), GovernmentForm FROM Country GROUP BY

GovernmentForm ORDER BY Population DESC;;

The trouble with this query is that it gives us far more data than we need:

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

| AVG(Population) | GovernmentForm

|

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

| 1277558000.0000 | People'sRepublic|

|48596000.0000 | Islamic Republic|

|133954700.0000 | Federal Republic|

|38357333.3333 | Socialistic Republic|

|22720000.0000 | Islamic Emirate|

...

|

1000.0000 | Independent Church State

|

|0.0000 | Co-administrated|

|0.0000 | Dependent Territory of the US|

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

Using conditions, we are able to modify the results on the server. However, how we

narrow the query, will determine the results.

Using WHERE, we can narrow the results to countries having populations over 100

million people.

SELECT AVG(Population) AS Population, GovernmentForm FROM Country

WHERE Population > '100000000' GROUP BY GovernmentForm ORDER BY

Population DESC;;

[ 315 ]SELECT Alternatives

The resulting table is drastically smaller:

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

| AVG(Population) | GovernmentForm

|

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

| 1277558000.0000 | People'sRepublic|

|344114800.0000 | Federal Republic|

|165915000.0000 | Republic|

|126714000.0000 | Constitutional Monarchy |

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

The problem is that we are not looking for countries with populations larger then 100

million. We are looking for forms of government. For this, we can use the following

statement with a HAVING clause:

SELECT AVG(Population) AS Population, GovernmentForm FROM Country

GROUP BY GovernmentForm HAVING Population > '100000000' ORDER BY

Population DESC;;

These results are even smaller:

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

| Population

| GovernmentForm

|

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

| 1277558000.0000 | People'sRepublic |

|

133954700.0000 | Federal Republic |

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

If you want to check these results against the larger table, simply take out the

HAVING clause:

SELECT AVG(Population) AS Population, GovernmentForm FROM Country

GROUP BY GovernmentForm ORDER BY Population DESC;;

The query using WHERE limited the values submitted for averaging. The condition

was applied to the data before the average was calculated. Using HAVING, the average

for each type of government was calculated first and then the condition was applied

to the result of AVG.

[ 316 ]Chapter 11

Subqueries

The WHERE clause reduces the amount of data through a simple filtering process and

HAVING filters the results. But MySQL also provides more robust ways of narrowing

the data from which results are culled. Normally, MySQL processes a query against a

database that is resident on disk. Subqueries, however, are nested SELECT statements

that result in a table of results against which the main query is processed. Once the

main query is processed against the results of the subquery, the latter is purged

from memory.

Up to this point, if we needed to take the results from one query and use it as input

for another query, we might feel constrained to use two SELECT statements and

manually transfer the data. Here, however, subqueries do that for us.

For example, let's say that we wanted to find the title of every movie done by actors

with the surname CHASE. The sakila database does not provide this information in

one table. Using a series of SELECT statements, we would first need to retrieve the

actor\_id of every actor with a surname of CHASE:

SELECT actor\_id FROM actor WHERE last\_name='CHASE';

We would need to save those results and use them in a series of SELECT statements

against film\_actor to get the values for film\_id:

SELECT film\_id FROM film\_actor WHERE actor\_id =<actor\_id>;

We would take each of those film identifiers and query film to get the title:

SELECT title FROM film WHERE film\_id = <film\_id>;>;

Doing this manually in MySQL would be an enormous headache. We could do

it easily enough in Python by recycling the data into each query through loops.

But that requires writing code unnecessarily and increases our I/O, costing us in

speed and responsiveness. It is better to put the onus on the database server with a

subquery or a join (discussed later in this book).

To form a subquery, we define the condition of the WHERE clause in terms of the

results of a SELECT statement. The syntax looks like this:

SELECT <column reference> FROM <table 1> WHERE <a column from table 1>

<relational operator> (SELECT <a similar column from table 2> FROM

<table 2>);

An example that combines the last two of our earlier SELECT statements is:

SELECT title FROM film WHERE film\_id = (SELECT film\_id FROM film\_actor

WHERE film\_id='17' AND actor\_id='3');

[ 317 ]SELECT Alternatives

The result is as follows:

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

| title

|

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

| ACADEMY DINOSAUR |

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

MySQL requires that subqueries result in only one row by default. For this reason,

we have had to place two conditions in the subquery—a film\_id value of 17 and an

actor\_id value of 3. Had we not done so, MySQL would have thrown an error that

reads:

ERROR 1242 (21000): Subquery returns more than 1 row

A comprehensive discussion on errors pertaining to subqueries and how

to resolve them can be found in the MySQL documentation: http://

dev.mysql.com/doc/refman/5.5/en/subquery-errors.html

In order to have MySQL process for each row of a series in sequence, we need to

preface the subquery with one of two keywords—ANY or IN. However, ANY is used

on the right side of any relational operator to cause MySQL to process any True

value that is returned. IN is used to process an expression list. To dispense with the

hardwired value for film\_id, we can use either one:

SELECT title FROM film WHERE film\_id IN (SELECT film\_id FROM film\_

actor WHERE actor\_id='3');

SELECT title FROM film WHERE film\_id = ANY (SELECT film\_id FROM film\_

actor WHERE actor\_id='3');

The result will be a listing for every title in film that has an identifier of the same

value as those associated with actor\_id 3 in film\_actor. Those are 22 in number.

But that only gets us part of the way. The actor with actor\_id 3 is only one of those

with the surname CHASE. Instead of repeating the process manually, we can nest

another subquery inside of the subquery. We can specify to MySQL to draw only

from those records where the last\_name value is CHASE.

SELECT title FROM film WHERE film\_id = ANY (SELECT film\_id FROM film\_

actor WHERE actor\_id = ANY(SELECT actor\_id FROM actor WHERE last\_

name='CHASE'));));

[ 318 ]Chapter 11

The results will be 49 rows containing all actors with the specified surname. We can

nest subqueries like Matryoshka dolls as long as we want and as long as the data

allows, and MySQL will not care. The queries simply must make sense, return one

row at a time (or use the keywords ANY or IN), and feed into the WHERE clause of the

next query.

Unions

Sometimes, you may not want to limit results but, rather, combine results from

multiple queries. Rather than execute two different SELECT statements in Python,

you can pass the task to the server with UNION. A UNION is the combination of the

results from two SELECT statements into a single result set. Unlike JOINs (discussed

in the next section), a UNION does do not combine the results side-by-side, but one

after the other. So where the results from the first query end, the results from the

second query begin.

The basic syntax of a UNION is as follows:

(<SELECT statement 1>) UNION (<SELECT statement 2>);

Each SELECT statement is discrete as they are neither related nor can they rely on

each other's data. The number of columns returned by each SELECT statement must

be the same. Otherwise, MySQL will throw an error.

The data type of each column should be the same with respect to the columns of the

other statement. If it is not, you can get strange results. Consider this UNION of two

queries against sakila:

(SELECT actor\_id AS id, last\_name AS name FROM actor WHERE last\_

name='WRAY') UNION (SELECT actor\_id AS actorid, film\_id AS filmid FROM

film\_actor) limit 5;

The results are as follows:

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

| id | name |

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

| 63 | WRAY |

|

1 | 1|

| 10 | 1|

| 20 | 1|

| 30 | 1|

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

[ 319 ]SELECT Alternatives

We have limited the results to five for purposes of illustration. In the table actor,

there is only one record with a surname of WRAY. Consequently, by mixing data types

between the two SELECT statements, the surname of the actor becomes conflated

with the film\_id from film\_actor. Note that MySQL also does not specify where

one set of results leave off and the other take up. The aliases used in the second

SELECT statement are not applied.

Like subqueries, UNIONs can be applied for as long as system resources and datasets

allow. Simply surround each SELECT statement with parentheses and separate each

with the word UNION.

We can also specify whether to sort the results of a SELECT statement without

modifying the SELECT statement itself. This is done by following the use of UNION

with DISTINCT. This will also cause MySQL to sort out duplicates as it did with

aggregate functions. The opposite of DISTINCT is ALL. This is the default behavior

of UNION, but can nevertheless be used explicitly. For the effect of each, consider the

results of the following two queries:

(SELECT last\_name AS name FROM actor WHERE last\_name='WRAY') UNION

DISTINCT (SELECT film\_id AS filmid FROM film\_actor) LIMIT 20;

+------+

| name |

+------+

| WRAY |

| 1|

| 2|

| 3|

| 4|

+------+

and

(SELECT last\_name AS name FROM actor WHERE last\_name='WRAY') UNION ALL

(SELECT film\_id AS filmid FROM film\_actor) LIMIT 20;

+------+

| name |

+------+

| WRAY |

| 1|

| 1|

| 1|

| 1|

+------+

[ 320 ]Chapter 11

Joins

Joins are often one of the hardest topics for MySQL newcomers to grasp.

The entire concept pivots on the understanding of mathematical sets, unions,

intersections, and their resulting Venn diagrams. Another way of looking

at joins is to use tabular information. For an example of this, see:

http://en.wikipedia.org/wiki/Sql\_join.

In the preceding Venn diagram, the left circle may be seen to represent one set and

the right circle another. The overlapping area is called the intersection of the two sets.

The set of elements that encompasses the contents of both circles is called a union,

but should not be confused with the MySQL keyword UNION. With that information

in mind, we can then see how MySQL allows us to access different parts of the Venn

diagram through joins.

LEFT and RIGHT joins

In MySQL, each set is represented by a table. The first table referenced in the query

is the set on the left. The second is represented by the circle on the right. For MySQL

to know which is the primary set when forming the results, we use LEFT or RIGHT,

respectively. If neither LEFT nor RIGHT is specified in the SELECT statement, LEFT

is presumed. For reasons of portability, the MySQL manual further recommends

against using RIGHT joins if one can use a LEFT JOIN instead. LEFT and RIGHT joins

are types of OUTER joins, which are dealt with in the next section.

The basic syntax for LEFT and RIGHT JOIN is:

SELECT <columns to be returned> FROM <table 1> <LEFT or RIGHT> JOIN

<table 2> ON <key column from table 1> <relational operator> <key

column from table2>;

[ 321 ]SELECT Alternatives

Whichever table is specified as primary is returned in full by default (subject to

selection criteria). Where there is a disparity between the lengths of the two sets, a

LEFT JOIN will result in the left set being exhausted. A RIGHT JOIN will use the right

set similarly. If the non-primary set is shorter than the primary, a NULL value will be

returned as a complement to the primary sets values. If the primary is shorter, the

non-primary set will be truncated to fit.

Using the world database, we can perform a JOIN that returns every actor for

each film by the film identifier. Alternatively, we can return every film for each

actor by actor. Whether we join to the right or to the left determines which is

returned. A LEFT JOIN of the datasets looks like this (we use LIMIT here to

keep the example manageable):

SELECT actor.actor\_id AS id, actor.first\_name AS first, actor.last\_

name AS last, film\_actor.film\_id AS film FROM actor LEFT JOIN film\_

actor ON actor.actor\_id=film\_actor.actor\_id LIMIT 10;;

The results

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

| id | first

| last

| film |

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

|1 | PENELOPE | GUINESS |1 |

|1 | PENELOPE | GUINESS |23 |

|1 | PENELOPE | GUINESS |25 |

|1 | PENELOPE | GUINESS |106 |

|1 | PENELOPE | GUINESS |140 |

|1 | PENELOPE | GUINESS |166 |

|1 | PENELOPE | GUINESS |277 |

|1 | PENELOPE | GUINESS |361 |

|1 | PENELOPE | GUINESS |438 |

|1 | PENELOPE | GUINESS |499 |

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

Clearly, we get every film for each actor. To get every actor for each film, we can

change the query to a RIGHT JOIN:

SELECT actor.actor\_id AS id, actor.first\_name AS first, actor.last\_

name AS last, film\_actor.film\_id AS film FROM actor RIGHT JOIN film\_

actor ON actor.actor\_id=film\_actor.actor\_id LIMIT 10;

[ 322 ]Chapter 11

and get the following results:

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

| id

| first

| last

| film |

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

|1 | PENELOPE

| GUINESS |

|10 | CHRISTIAN | GABLE|1 |

|20 | LUCILLE| TRACY|1 |

|30 | SANDRA| PECK|1 |

|40 | JOHNNY| CAGE|1 |

|53 | MENA| TEMPLE|1 |

|108 | WARREN| NOLTE|1 |

|162 | OPRAH| KILMER|1 |

|188 | ROCK| DUKAKIS |1 |

|198 | MARY| KEITEL1 |

|

1 |

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

As with every other part of MySQL, whether the results you get are valid for the

question you need to answer depends on how well you phrase your query.

OUTER joins

As the name implies, OUTER joins differ from INNER joins in what they encompass;

the latter is discussed in the next section. The OUTER JOIN is so called because it

always includes one of the tables in addition to the intersection of the two. The table

to be included is indicated by the terms LEFT and RIGHT. The resulting joins are

called LEFT [OUTER] JOIN and RIGHT [OUTER] JOIN, respectively.

[ 323 ]SELECT Alternatives

In the preceding Venn diagram, a LEFT OUTER JOIN would include the intersection

labeled JOIN as well as Table 1. A RIGHT OUTER JOIN includes the intersection and

the contents of Table 2. Now we can see how it makes sense to call these OUTER joins

because they contain data that are outside the region where the two tables are joined.

The syntax of an OUTER JOIN is as follows:

SELECT <column reference> FROM <table 1> <LEFT or RIGHT> OUTER JOIN

<table 2> ON (<key column from table 1> = <key column from table

2>);>);>);>);

Against the sakila database, for example, we can run this JOIN:

SELECT actor.\*, film\_actor.\* FROM film\_actor LEFT OUTER JOIN actor ON

(YEAR(actor.last\_update) = YEAR(film\_actor.last\_update)) limit 5;

The syntax of a RIGHT OUTER JOIN would be the same save for the obvious use of

RIGHT instead of LEFT. For the effective difference of each, see the previous section

on LEFT and RIGHT joins.

INNER joins

In MySQL, an INNER JOIN is simply the intersection of the two sets to the exclusion

of anything that does not overlap. A diagram of it is as follows:

As seen here, the INNER JOIN gives us the inner part of the diagram.

To affect an inner join, MySQL naturally requires you to specify which tables are to

be compared. It also requires the specific columns from each table to be specified as

well as the key values to be used for forming the intersection. The basic syntax of the

INNER JOIN is as follows:

SELECT <columns to be returned> FROM <table 1> INNER JOIN <table 2>

ON <key column from table 1> <relational operator> <key column from

table2>;

[ 324 ]Chapter 11

So for example, if we wanted to use the world database and find every city that lies

in a country of the same name, we might try to cull out all the city names from City

and all the country names from Country. Of course, the only way to relate the two

with certainty is to cull out the CountryCode from City and match it to Code from

Country. All of this requires a lot more processing than needed. The better option

would be to use an INNER JOIN between the City and Country tables:

SELECT City.Name AS Name, Country.Name AS Country, Country.Region AS

Region FROM City INNER JOIN Country ON City.Name = Country.Name;

As different column names may carry different meanings in different tables, it is not

a bad idea to use AS again to create custom headings. The results of this query are

as follows:

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

| Name

| Country

| Region

|

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

| Djibouti| Djibouti| Eastern Africa

| Mexico| Mexico| Central America |

|

| Gibraltar| Gibraltar| Southern Europe |

| Armenia| Armenia| Middle East

|

| Kuwait| Kuwait| Middle East

|

| Macao| Macao| Eastern Asia

|

| San Marino | San Marino | Southern Europe |

| Singapore

| Singapore

| Southeast Asia

|

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

We can further apply conditions to the data of the join by appending a WHERE clause

to the statement.

SELECT City.CountryCode AS Code, City.Name AS Name, Country.Name AS

Country, Country.Region AS Region FROM City INNER JOIN Country ON

City.Name = Country.Name WHERE Country.Region LIKE 'South%';

This returns only those results where the region begins with South:

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

| Code | Name

| Country

| Region

|

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

| GIB| Gibraltar

| Gibraltar

| SMR| San Marino | San Marino | Southern Europe |

| SGP| Singapore

| Singapore

| Southern Europe |

| Southeast Asia

|

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

[ 325 ]SELECT Alternatives

However, INNER joins are quite powerful. They are unsurprisingly very common in

MySQL programming.

NATURAL joins

A NATURAL JOIN combines the two tables based on their commonalities. If neither

LEFT nor RIGHT is specified, then neither is given precedence, which often results

in an empty set. However, the table which is given precedence is reproduced in the

results. For every record of the precedent table, a record is produced in the results.

Where the columns have overlapping data, only one record will be reproduced.

If the precedent table is longer or fuller than the other set, the results from the latter

will be represented with NULL values. If the precedent table is shorter, the other table

is truncated to match.

The basic syntax of NATURAL joins is as follows:

SELECT <columns> FROM <table 1> NATURAL <LEFT or RIGHT> JOIN <table

2>;

In practice, it would look like this query of the world database:

SELECT \* FROM City NATURAL LEFT JOIN Country;

In the results, you will see that everywhere Country has no column whose name is

the same as that of the value in City, a NULL value is inserted. Using NATURAL RIGHT

JOIN instead gives results that defer to the second table over the first.

Another example that shows the way MySQL automatically sorts the data on a

NATURAL JOIN is seen with the sakila database:

SELECT \* FROM actor NATURAL LEFT JOIN film\_actor LIMIT 5;

We limit the results for sake of space, but they illustrate that actor\_id, being

common to both tables, is only reproduced once.

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

| actor\_id | last\_update

| first\_name | last\_name

| film\_id |

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

|1 | 2006-02-15 04:34:33 | PENELOPE| GUINESS|NULL |

|2 | 2006-02-15 04:34:33 | NICK| WAHLBERG|NULL |

|3 | 2006-02-15 04:34:33 | ED| CHASE|NULL |

|4 | 2006-02-15 04:34:33 | JENNIFER| DAVIS|NULL |

|5 | 2006-02-15 04:34:33 | JOHNNY| LOLLOBRIGIDA |NULL |

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

[ 326 ]Chapter 11

CROSS joins

Where INNER joins give us the intersection of the two sets and OUTER joins give us

the intersection with one of the sets, the CROSS JOIN gives us both sets. The basic

syntax is as follows:

SELECT <columns from table 1>, <columns from table 2> FROM <table 1>

CROSS JOIN <table 2>;

An example using the world database is:

SELECT City.\*, Country.\* FROM City CROSS JOIN Country;

Without a conditional clause, every record of table 2 will be returned for every

record of table 1. If each table has 1000 records, the table returned will be the

product of those two sets, or one million records. We can limit the results with a

WHERE clause as follows:

SELECT City.\*, Country.\* FROM City CROSS JOIN Country WHERE City.Name

LIKE 'C%';

This query restricts the number of results by quantifying the value of the Name

column in the City (table 1) set of data. Even so, we still get 67,159 records

returned.

Depending on which columns you specify, you can retrieve all or few of the fields for

each record. In the following example, we simply retrieve the city and country name

for each row, again quantifying the results with a WHERE clause.

SELECT City.Name, Country.Name FROM City CROSS JOIN Country WHERE

City.Name LIKE 'C%';

Greater precision in forming a query naturally leads to faster processing. Results

vary on different servers, but the unqualified query takes over 12 minutes to return

where the second query takes around 80 seconds and the third 20 seconds.

Doing it in Python

As we have seen in previous chapters, processing a SELECT query in Python is as

simple as execute() and fetchall(). However, Python also allows us to build

statements dynamically, and this applies to joins, unions, and subqueries as well.

[ 327 ]SELECT Alternatives

Subqueries

If we want column from table1, but the column reference for the subquery is

colref, from both table1 and table2, we can write the following:

#!/usr/bin/env python

import MySQLdb

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

cursor = mydb.cursor()

table1 = 'film'

table2 = 'film\_actor'

column = 'film\_id, title'

colref = 'film\_id'

statement = "SELECT %s FROM %s WHERE %s IN (SELECT %s FROM %s)"

%(column, table1, colref, colref, table2)

cursor.execute(statement)

results = cursor.fetchall()

for i in results: print i[0], '\t', i[1]

The results obviously will be the title and identifier for each title in film. We can

further nuance this for reader input to allow searches by the name of the actor.

#!/usr/bin/env python

import MySQLdb

import sys

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

cursor = mydb.cursor()

table1 = 'film'

column = 'film\_id, title'

colref = 'film\_id'

sub1ref = 'film\_id'

table2 = 'film\_actor'

sub2ref = 'actor\_id'

table3 = 'actor'

[ 328 ]Chapter 11

firstname = sys.argv[1]

surname = sys.argv[2]

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

= ANY (SELECT %s from %s WHERE first\_name='%s' AND last\_name = '%s'))"

%(column, table1, colref, colref, table2, sub2ref, sub2\

ref, table3, firstname, surname)

cursor.execute(statement)

results = cursor.fetchall()

for i in results: print i[0], '\t', i[1]

Unions

As with subqueries, we can create the two statements of a UNION and join them

dynamically. For example:

#!/usr/bin/env python

import MySQLdb

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

cursor = mydb.cursor()

statement1 = "SELECT actor\_id FROM actor"

statement2 = "SELECT film\_id FROM film"

union = "(%s) UNION (%s)" %(statement1, statement2)

cursor.execute(union)

results = cursor.fetchall()

for i in results: print i[0]

Joins

As mentioned in the earlier section on joins, which JOIN you use will depend on

the results you want. Here an INNER JOIN will be illustrated, but the same process

applies to the formation of the other kinds of joins.

#!/usr/bin/env python

import MySQLdb

[ 329 ]SELECT Alternatives

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

cursor = mydb.cursor()

table1 = 'City'

table2 = 'Country'

col1 = "CountryCode"

col2 = "Region"

colref1 = table1 + "." + col1

colref2 = table2 + "." + col2

colref = colref1 + ", " + colref2

keyref1 = 'Name'

keyref2 = 'Name'

key1 = table1 + "." + keyref1

key2 = table2 + "." + keyref2

statement = "SELECT %s FROM %s INNER JOIN %s ON %s = %s" %(colref,

table1, table2, key1, key2)

cursor.execute(statement)

results = cursor.fetchall()

for i in results: print i[0], '\t', i[1]

After opening a connection to the database and creating a cursor, we define the

parts of the statement, concatenating strings as necessary. We then insert the sundry

values into the value of statement.

Remember that the statement remains a string until it is passed to cursor.

execute() as an argument. It can be then amended at any time. If you find it too

difficult to manage the variables that are to be inserted into the statement, simply

break the statement down into parts and combine it at the end. Should a problem

with the statement persist, you can use a print statement just before the statement is

executed to find out what is being passed to MySQL.

Project: Implement HAVING

For the project of this chapter, we will introduce support for HAVING under sorting

in our basic database administration web application. Basically, a full-fledged

web-based administration application should have support for JOIN and, to a

certain extent, subqueries. However, this functionality will be left for room to grow.

[ 330 ]Chapter 11

The goals of this project are straightforward:

•Implement support for HAVING in the Python back-end of the application

•Code the HTML front-end to allow HAVING in conjunction with sorting

When we are done, we will also look at some ways that this application could

(and should) be dressed up.

Revising the Python backend

Before implementing a user interface for any functionality, one naturally needs

to code support into the program itself. In the case of HAVING, we need to do the

following in pymyadmin.py:

•Revise the qaction function to insert HAVING into the MySQL statement that

it passes to execute()

•Revise the qaction function call in main()

•Code support for appropriate option-handling—whether as CGI or as a

command-line option passed by PHP

Revising qaction()

Currently, the qaction() function looks like this:

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 = ""

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

col = columns[i].strip()

val = values[i].strip()

[ 331 ]SELECT Alternatives

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

In order to add facility for HAVING, we need to add some internal variables to carry

the value of the column and value keys, the arguments for HAVING. We do not need

to change the variable arguments of the function definition because the HAVING

arguments will be passed as part of aggregates[].

After the assignment key = aggregates[4], seen previously, we should add:

hcol = aggregates[5]

hval = aggregates[6]

[ 332 ]Chapter 11

The need for HAVING will be qualified by the presence of a value for hcol. Find the

test for sorting with the following if clause:

if sort != "NONE":":

## or: if sort is not None:

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

else:

sorting = ""

To assign a value for HAVING, we need to test whether sorting is desired and then

whether hcol has any value. If sort and hcol both have a value other than NONE,

we then prepare a value having. The if...else statement looks like this:

if hcol != "NONE" and sort != "NONE":

having = "HAVING %s = '%s'" %(hcol, hval)

else:

having = ""

With a value for having, we revise the SELECT statement to include having:

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

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

Just as with sorting, nothing is added if no value is to be appended.

Revising main()

With support for HAVING implemented in qaction(), we can build support for it in

main(). It is worth noting that we implemented HAVING in qaction in a way that

does not require its use. If we wanted, we could create separate calls for each in

main(), or another calling function.

To create support for HAVING in main(), we create another if...else clause in the

while loop. Currently, for usage with PHP, the while loop reads as follows:

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:

[ 333 ]SELECT Alternatives

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:

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)

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

[ 334 ]Chapter 11

printout = HTMLPage()

printout.message(output)

output = printout.page()

print output

break

The variable names used here would be different if the application is run under CGI

because of the CGI modules. See Chapter 7, Creating and Dropping for more on the

differences between using CGI and PHP and how to work with each.

It is possible to create a program that will run with both CGI and PHP by

either testing how the program is called or by using a try...except

structure for the options. However, coding such as that would only come

into play if you needed to deploy the system in both environments and

wanted to conserve on code.

We need to insert our if...else structure just after the if...else structure for

the assignment of sort and key. When we are done, that code section should read

as follows:

if opt.sort is not None:

sort = opt.sort

key = opt.key

else:

sort = "NONE"

key = "NONE"

if opt.hcol is not None and opt.hval is not None:

hcol = opt.hcol

hval = opt.hval

else:

hcol = "NONE"

hval = "NONE"

In addition to testing the value of opt.hcol, we should also test for None in hval.

Otherwise, we can get an error.

After the values of hcol and hval are assigned, we need to append each to the

arguments for qaction() in appropriate order. If we conflate them, qaction()

will assign their values in the wrong order.

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

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

distinct, sort, key, hcol, hval)

[ 335 ]SELECT Alternatives

With that, HAVING is supported by the main() function. We still need to build

support into the options of the program.

Revising the options

As with previous chapters, how you code support for the options, depends on

whether you are coding for CGI or PHP. See Chapter 7 for the difference on each.

Heretofore, we have been coding for PHP due to its relative simplicity in executing,

debugging and because PHP tends to execute faster than CGI when using default

configurations.

Currently, the beginning of our options-handling for sorting looks like this

(including the introductory code for optparse):

import optparse

# Get options

opt = optparse.OptionParser()

...

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")

To add option support for HAVING, we need to support hcol and hval. Therefore, we

need to add the following:

opt.add\_option("-H", "--hcol",

action="store",

type="string",

help="column to use for HAVING",

dest="hcol")

opt.add\_option("-V", "--hval",

action="store",

type="string",

help="value to use for HAVING",

dest="hval")

[ 336 ]Chapter 11

To be sure, Python does not care where in the list of options you put the assignment.

As long as it is before the assignment of opt and args, it will be assigned with all

of the other option values. However, for the sake of maintenance, it is best to add it

after the sorting options are handled:

opt.add\_option("-H", "--hcol",

action="store",

type="string",

help="column to use for HAVING",

dest="hcol")

opt.add\_option("-V", "--hval",

action="store",

type="string",

help="value to use for HAVING",

dest="hval")

With that, we finish implementing support for HAVING in our administration

application. However, we still need to create support for it in the HTML interface.

Revising the HTML interface

Currently, the section for queries in the HTML code reads as follows:

<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>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>

DISTINCT?

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

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

<br>Sorting:

[ 337 ]SELECT Alternatives

<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>

To finalize our implementation of HAVING for sorting, we need to solicit values

for hcol and hval from the user. After the value of key is assigned and before the

double line-breaks, we should insert the following dialogue:

<br>

HAVING <br>

Column: <input type="text" name="hcol" value="">

<br>

Value: <input type="text" name="hval" value="">

Then the dialogue is finished. As is illustrated in the following section Room to grow,

there are several things that we can and should add to this application before we

consider it complete.

Room to grow

With HAVING implemented for sorting in our administration application, we will

leave this project for others in subsequent chapters. As has been alluded several

times, this application could use more coding to make it more useful and usable

than it is at present. In particular, some issues that you should address are:

•How would you implement the various kinds of joins and subqueries?

•As HAVING is used outside of sorting, how would you allow a user to use it in

a generic SELECT statement?

•Currently, the tabular information returned by SELECT is not formatted. How

would you allow for its formatting while being sensitive to the fact that not

all statements will be SELECT statements?

•Aside from being monochrome, the page is now much more verbose than

when we started. How could you use CSS to create a tab-based menu system

instead of the sub-headers to simplify the interface and keep everything on

one screen?

[ 338 ]Chapter 11

Summary

In this chapter, we have covered how to use HAVING clauses, subqueries, and joins in

Python. We have also seen:

•How to restrict results using HAVING

•How HAVING differs from WHERE

•When it is best to use the following—WHERE or HAVING

•How to narrow data even further with subqueries

•Using UNION to concatenate two results sets before they are returned

•The various ways to join tables, including the difference between a LEFT JOIN

and a RIGHT JOIN as well as the differences between INNER joins, OUTER joins,

NATURAL joins, and CROSS joins

In the next chapter, we will look at several more of MySQL's powerful functions and

how to use them best in Python.

[ 339 ]Strin