A Biologists' Introduction to MySQL

Jacob Brown

Contents

Chapter 1

Preamble

What is MySQL?

MySQL is a database handling language which allows the user to establish, retrieve data from, and manipulate databases. It is one of the many variations of *Structured Query Language* (SQL).

Why should I use MySQL?

Assuming you are reading this, you already realise the potential of having a secure database, however the following are a few reasons why you should be using MySQL.

- Data is secure in its class - dates, times, strings, etc. are stored in defined ways that make it difficult to misinterpret
- Manipulating and handling data is ingrained in the language.
- Unlike other SQL languages (Oracle, Access, etc.) MySQL is free.
- It is easily used with analytical languages, such as R and Python.
- MySQL is immensely popular, troubleshooting issues are often quickly resolved with a quick search of the web.

What is this guide and is it for me?

This a a short guide to getting started with MySQL, it is aimed at the complete beginner with biological examples - although the general concepts are applicable in wider use. No prior knowledge of the MySQL language is required - however I'll assumed you have accessing a pre-established database.

What do I need to get started?

MySQL can run in the terminal, however a Graphic User Interface can be used - MySQL Workbench (stand-alone program) and phpMyAdmin (browser based) are good options.

Dataset Structure and Definitions

Datatables can be structured in a multitude of ways, one of the simplest being made of a series of *variables* (columns) and rows, with each cell containing a *value* - throughout this will be the structure.

Below are a few terms that you first should familerise with.

- **Query** - the code in which specifies what data should be returned
- **Function** - a built in feature that performs a pre-defined task
- **Clause** - a section of the query
- **Null** - an absent value
- **Alias** - the name given to a variable, table, subquery, etc.

Chapter 2

Selecting Data

Queries and their Structure

Queries specify which data is retrieved from the database, each can be thought of as requests made by the user to the database.

"Return all of the information on sample numbers between 300 and 350", might look like:

`SELECT * FROM tblSamples WHERE SampleNumber BETWEEN 300 AND 350`.

Each query has a pre-defined structure with mandatory and optional clauses, the defined order and function of each of the clauses is as follows.

```
SELECT `variable1`, `variable2`, `variable3` -- Which variables do you want?
FROM `table1` -- From where?
JOIN `table2` -- Attach another table of data
WHERE condition --  Filter by what?
GROUP BY `variable` -- Grouping on a variable
ORDER BY `variable` -- Arrange the data by
HAVING condition -- Filter by what?
```

`SELECT` and `FROM` are mandatory in MySQL, all else in the above are optional.

It is good practice to wrap aspects of a query with punctuation. Single quote marks for values, and back-ticks for variables and tables.

```
'values' -- Single quotation marks
`variables` -- Backticks
`tables` -- Backticks
```

Selecting Variables

An asterisk (*) denotes selecting all variables, `SELECT * FROM tblSamples` would return everything from tblSamples. `SELECT tblSamples.* FROM tblSamples` is synonymous with the previous statement - this will later come in use when joining multiple tables and using sub-queries. Alternatively, following the `SELECT` clause, specified variables can be stated - only they will be returned on running the query.

```
SELECT `SampleNumber`, `SampleDate`, `BatchNumber`
FROM `tblSamples`
```

The `SELECT` clause is not limited to available variables, calculations and functions can be included within the query. For example:

```
SELECT `SampleNumber`, `SampleDate`, `BatchNumber`, `BatchNumber`+20
FROM `tblSamples`
```

The above would return SampleNumber, SampleDate, and BatchNumber, with an additional variable that adds 20 to the BatchNumber.

Assigning an Alias

Renaming, or assigning an alias, can be applied to (and in some cases is mandatory for) variables, tables, and sub-queries. It uses `AS`, followed by the desired alias. Assigning alias can make the code more readable and prevent variables from having the same or similar names - which may result in an error when running the query.

Example: You wish to rename a variable and the table from which it is found.

```
SELECT `Individual`, `Weight` AS `WeightKG`
FROM `tblWeights` AS `weights`
```

Filtering with Where and Having

Filtering is carried out through the use of `WHERE` and `HAVING` - both of which are optional clauses, but frequently used. In both clauses a condition is used to filter. The primary difference between the two is that `WHERE` runs simultaneously with `SELECT` and `FROM`, whereas `HAVING` runs after.

Example: From a data-table of birth-dates you wish to return individuals that are currently over 1 year old. `CURDATE()` is used to retrieve the current date and `DATEDIFF()` for calculating the difference between dates.

WHERE

```
SELECT `Individual`, `BirthDate`
FROM `tblIndividuals`
WHERE DATEDIFF(CURDATE(), `BirthDate`) >= 365
```

HAVING

```
SELECT `Individual`, `BirthDate`, DATEDIFF(CURDATE(), `BirthDate`) AS `Age`
FROM `tblIndividuals`
HAVING `Age` >= 365
```

The above `HAVING` clause assigns a new alias, from which the condition refers to - this structure would not work with `WHERE` unless additions to the query are made.

Conditions

Conditions are either met or not (true or false), here their use is highlighted through the use of `WHERE` and `HAVING`. If conditions are met on a particular variable, only the values matching the condition will be returned.

Example: Return only the individuals whose natal group 'HEAT'.

```
SELECT *
FROM `tblIndividual`
WHERE `NatalGroup` = 'HEAT'
```

Below are a few commonly used conditions.

```
= -- Equals
!= -- Doesn't equal
> -- Greater
< -- Less than
IS NULL -- Is null
IS NOT NULL --  Is not null
LIKE %value% -- Is similar to
NOT LIKE %value% -- Not similar to
```

I will not go into detail about string searches, however note that the presence or absence, and the placement, of `%` will determine matches for which characters and at specific locations within the string.

If multiple conditions are needed to be met, state `AND` between the conditions. Alternatively, `OR` can be used to return data matching either of the conditions.

```
WHERE condition1 AND condition2
```

```
WHERE condition1 OR condition2
```

If multiple different values are required to be met in a single variable, `IN()` may be uesd.

Example: You wish to return data from lion 1, 7, and 6.

```
SELECT *
FROM `tblIndividual`
WHERE `Individual` IN(1, 7, 6)
```

Ordering the Data

Ordering your data uses the `ORDER` clause, it precedes variables by which to sort. The default is to ascend (`ACS`) the values, to descend stating `DESC`. Multiple variables can be used to order the data, the first in the sequence takes priority - this comes in particular use for ordering within grouped data.

```
SELECT *
FROM `tblIndividuals`
ORDER BY `GroupName`, `BirthDate` DESC, `Sex`
```

Chapter 3

Dates and Times

Grasping a few date and time functions will likely be important, the following section describes a few. Many of the functions can be used in the `SELECT` statement, in addition to within conditions.

The below functions should have the parenthesis left empty.

```
NOW()    -- Current datetime
CURDATE() -- Current Date
CURTIME() -- Current Time
```

The below functions require a datetime, date, or time object within the parenthesis.

```
SECOND() -- To seconds
MINUTE() -- To minutes
HOUR()  -- To hours
DAY() -- To days
MONTH() -- To months
YEAR() -- To years
```

Difference Between Dates

Calculating differences between dates and times uses the below functions.

```
DATEDIFF('date1', 'date2')
TIMEDIFF('time1', 'time2')
TIMESTAMPDIFF(unit, 'datetime1', 'datetime2')
```

Datetime units are: `SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, YEAR`

Adding and Subtracting from a Datetime

Adding or subtracting from a particular date uses.

```
DATE_ADD('date', INTERVAL val unit)
DATE_SUB('date', INTERVAL val unit)
```

`TIMESTAMPADD(unit,INTERVAL,date)` is synonymous with the above `DATE_ADD()` function.

Example: Adding one hour to the current time

```
DATE_ADD(NOW(), INTERVAL 1 HOUR)
```

First and Last Days of the Month

The first day of the month for a given date is calculated as the following.

```
DATE_SUB('date',INTERVAL DAYOFMONTH('date')-1 DAY) AS `StartDay`
```

The last day of the month is slightly easier to calculate - as it has its own function.

```
LAST_DAY('date') AS 'LastDay'
```

Creating Dates

The basis of creating new dates is to create a new string. Two functions are used within the `SELECT` clause - `CONCAT()` joins strings, `LPAD()` ensures sections of the sting are a set length.

Example: You wish to create a new date, with the year, month, and day from existing date strings.

```
CONCAT('2018', '-', MONTH('2017-01-10'), '-', DAY('2017-01-10'))
```

The above returns 2018-1-10, note however that the month is in the incorrect format, this is where `LPAD()` becomes useful - it allows the user to state the number of characters a string should be and if the length isn't met what character should be used in place.

```
CONCAT('2018', '-',
       LPAD(MONTH('2017-01-10'), 2, '0'), '-',
       LPAD(DAYOFMONTH('2017-01-10'), 2, '0'))
```

The returned date would be 2018-01-10.

A variation of `CONCAT()` that is particularly useful when creating dates is `CONCAT_WS()`, instead of repeating the separator (hyphen in the above examples), it is stated once within the function.

```
CONCAT_WS('-', '2018', '01', '01')
```

The returned date would be 2018-01-10.

Chapter 4

Summaries and Grouping

Summary functions are used in conjunction with the `GROUP BY` clause. The data is grouped on one or multiple variable(s), and summary functions are applied to the specified variables in the `SELECT` clause.

Example: Return the number of times an individual was seen.

```
SELECT COUNT(`SightingRef`), `IndividualID`
FROM `tblObservations`
GROUP BY `IndividualID`
```

Below are a few common examples of summary functions.

```
COUNT()
AVG()
MAX()
MIN()
SUM()
DISTINCT `variable1` -- eliminates duplicates and returns values
COUNT(DISTINCT `variable1`) -- counts unique values from variable 1
```

The `GROUP BY` clause should be used with caution, as grouping by a particular variable doesn't necessarily ensure that all of the returning data is correct. When stating variables in the `GROUP BY` clause, the remaining variables should appear in the `SELECT` statement with a summary function (or not appear at all).

Example: You wish to total the time spent observing each pride of lions.

Correct use

```
SELECT SUM(`Duration`), `GroupName`
FROM `tblObservations`
GROUP BY `GroupName`
```

Incorrect use

```
SELECT `Duration`, `GroupName`
FROM `tblObservations`
GROUP BY `GroupName`
```

In both of the above examples Duration and GroupName will be returned, however in the incorrect use example Duration will return the value from the uppermost row, rather than the desired total. Although a crude example, it demonstrates the importance of structuring your queries and being aware of what you are retrieving.

Chapter 5

Joining and Sub-Queries

You may want to join two (or more) tables of data, in order to do so left, right, and inner join are used. There may not always be an exact match for each row, depending on the type of join choosen, will have either a filtering effect or *NULL* values being generated. There are two broad types of join used in MySQL, *inner* and *outer joins*, of the latter there are two - *left* and *right joins*.

Example: Imagine two tables, one of individual lions (tblIndividual) and another table of current dominant lions (tblDominant) - you wish to join tblDominant to tblIndividual.

A *inner* or *right join* of tblDominant onto tblIndividual, would result in only the dominant individuals being returned. Whereas a *left join* would return all the information from tblIndividual, in addition to the dominance information of those that are dominant (i.e. those in tblDominant) - non-dominant individuals would have *NULL* values for the tblDominant varibles.

Joins require a variable to join the two tables on (think of it as a match between values) - the functions `ON()` and `USING()` are used. `ON()` matches variabables with different alias, however is not limited to, and `USING()` matches on variables with the same alias.

```
ON(`IndividualID` = `DominantID`)
```

IndividualID and DominantID have matching values.

```
USING(`IndividualID`)
```

In each table there is a common variable name (IndividualID) that is joined on.

Inner Join

Matches data from the *left* table to the *right* table, if a mismatch is present (i.e. a *NULL* value is present in one or both of the tables) the row isn't returned.

```
SELECT *
FROM `tblIndividual`
INNER JOIN `tblDominant`
    ON(`IndividualID` = `DominantID`)
```

IndividualID	NatalGroup	DominantID	StartDomDate
1	1	1	2017-01-10
2	2	2	2001-01-01

Left Join

Matches data from the *left* table to the *right* table, when a mismatch is present from the table on the *left* the row from the *right* table is retained, resulting in *NULL* values being generated.

```
SELECT *
FROM `tblIndividual`
LEFT JOIN `tblDominant`
    ON(`IndividualID` = `DominantID`)
```

IndividualID	NatalGroup	StartDomDate	DominantID
1	1	2017-01-10	1
2	2	2001-01-01	2
3	1	NA	NA
4	2	NA	NA
5	1	NA	NA
6	2	NA	NA
7	1	NA	NA
8	2	NA	NA
9	1	NA	NA
10	2	NA	NA

Right Join

Similar to the *left join* however, works in the opposite direction - matching data from the *right* table to the *left* table. Below, if the individual is not found in tblDominant the row isn't returned.

```
SELECT *
FROM `tblIndividual`
RIGHT JOIN `tblDominant`
    ON(`IndividualID` = `DominantID`)
```

IndividualID	NatalGroup	DominantID	StartDomDate
1	1	1	2017-01-10
2	2	2	2001-01-01

N.B. A *right join* of tblIndividual onto tblDominant has the same effect as the above *left join*, with the dominance variables appearing in the first columns - see below.

```
SELECT *
FROM `tblDominant`
RIGHT JOIN `tblIndividual`
    ON(`DominantID`=`IndividualID`)
```

DominantID	StartDomDate	IndividualID	NatalGroup
1	2017-01-10	1	1
2	2001-01-01	2	2
NA	NA	3	1
NA	NA	4	2
NA	NA	5	1
NA	NA	6	2
NA	NA	7	1
NA	NA	8	2
NA	NA	9	1
NA	NA	10	2

A Cautionary Note on Joins

By joining one table to another it is possible to duplicate data, this will occur if one table has multiple references to join on. Duplication may not necessarily be incorrect however, imagine that you have two tables one for individual lions within a population and another

table for their sightings. Both tables contain IDs, however the table of individuals contains one reference whereas the sightings table may contain multiple entries - this is known as a one-to-many relationship. Joining the two tables on the ID may return duplications of the individual lions (one for each of the corresponding sighting). Although the duplication is not necessarily incorrect, it should be noted prior to analysis.

Sub-Queries

Sub-queries break up queries into manageable chunks - joins are used to nest additional queries. The code becomes more readable and troubleshooting becomes more efficient. Each sub-query requires a new alias - it is good practice to give it a descriptive name of the query within.

The following contains two subqueries, one of which uses a `GROUP BY` statement.

```
SELECT *
FROM `tblIndividual`

INNER JOIN (SELECT *
            FROM `tblGroups`) AS `grp`
      ON(`grp`.`GroupName` = `tblIndividual`.`CurrentGroup`)

LEFT JOIN (SELECT `IndividualID`, MAX(`SeenDate`) AS `LastSeen`
               FROM `tblObservations`
               GROUP BY `IndividualID`) AS `lastseen`
        USING(`IndividualID`)
```

N.B. how the new alias for grp is used in the `ON()` statement.

Union

A union is used to join two (or more) queries by vertically joining the datasets. The `SELECT` clause should be structured equally, i.e. same number and order of returning variables.

```
SELECT `IndividualRef`, `NatalGroup`, `BirthDate`
FROM `tblFemales`

UNION

SELECT `IndividualRef`, `NatalGroup`, `BirthDate`
FROM `tblMales`
```