# SQL GUIDE

## Databases

If you’re unfamiliar with what a database is, then read this brief overview to familiarise yourself:

“A database is made up of multiple tables. Just like Excel tables, database tables consist of columns and rows. Each column corresponds to an attribute, and each row corresponds to a single record. Each table must have a unique name in a database.

For example, consider a database table that contains names and telephone numbers. You would probably set up columns named “FirstName,” “LastName” and “TelephoneNumber.” Then you would simply start adding rows underneath those columns that contain the data. In a table of contact information for a business with 50 employees, we’d wind up with a table that contains 50 rows.

An important aspect of a table is that each must have a primary key column, so that each row (or record) has a unique field to identify it.

One of the most powerful features of a database is the ability to create relationships between tables using foreign keys. For example, you might have a Customers table and an Orders table. Each customer can be linked to an order in your Orders table. The Orders table, in turn, might be linked to a Products table. This kind of design comprises a relational database, and simplifies your database design so that you can organize data by category, rather than trying to put all the data into one table, or just a few tables.”

Taken from: <http://databases.about.com/od/specificproducts/a/whatisadatabase.htm>

## Language

In R, we typically use dataframes to store and anlayse data (most of the homework in the Stats course in term1 stored the data in dataframes). A SQL table is equivalent to a dataframe, and as such its layout should be very familiar to you.

The term ‘row’ is used interchangeably with ‘entry’ or ‘record’. Much like in a normal dataframe, a row is a single entry. It typically contains data on one instance from the dataset (e.g. one transaction from a shop, or one tweet from a user on Twitter).

Compared to R, SQL syntax is very easy to use and understand. Each command has a clearly defined function, and there is normally just one way to solve a problem.

## Useful Commands in SQL:

*Note: as in Quarry, the output is color coded with Red and Purple. I have also boldened the part of the clause which contains a new command. When you write SQL in Quarry you don’t need to (and, indeed, won’t be able to) use bold for anything.*

1. **SHOW DATABASES**;

This shows the databases available in the ‘SQL Schema’ (which is just a fancy term for the list of databases in the server). In Quarry, there are ~1,870 databases available.

1. **USE** database1;

To specify which of the available databases we want to work with, use the USE command followed by the name of the database.

Note: larger databases take exponentially longer to run than smaller databases. So, if you use a large database in Quarry (such as enwiki\_p) your queries could take a very long time to run. For this reason, you should use a smaller database, like simplewiki\_p, in the classes.

1. **SHOW TABLES**;

The SHOW TABLES command returns the names of the Tables within the Database. For the simplewiki\_p database there are 59 tables.

1. **DESCRIBE** table1;

Once we know the name of a table we want to find out more about, we can use the DESCRIBE command. The DESCRIBE command lists all of the fields in the table and the data format of each field.

1. **SELECT \* FROM** table1

This is the syntax for extracting values from a table. Using the SELECT tells SQL that we want to extract some data and FROM table1 specifies which table we want to use.

Using \* specifies that we want to take all of the available columns. We can also just specify one or many columns individually: SELECT var1, var2, var3 FROM table1

1. SELECT \* FROM table1 **LIMIT 1000**

If we don’t specify a limit to the number of entries that we want returned then everything will be returned. Sometimes, this is fine. Other times, such as when we are working with a huge database like Quarry, taking all of the entries will take unfeasibly long. Use LIMIT 1000 to cap how many entries are returned.

1. SELECT \* FROM table1 ORDER BY var1 **ASC** LIMIT 1000;

ORDER BY specifies that we want the results ordered by one of the variables from within the table. Here, we have also passed the command ASC to specify that we want the results in ascending order. If we pass it DESC then the results will be ordered in descending order. If we don’t pass it either ASC or DESC then the default is ASC.

If you want to randomly select entries then use RAND() e.g. SELECT \* FROM table1 ORDER BY RAND() limit 1000;

1. SELECT table1.var1, table1.var2, table2.var3 FROM table1 **JOIN table2 ON table1.var1 = table2.var1** ORDER BY var3 ASC LIMIT 10;

A very commonly used, and powerful, command in SQL is JOIN. Because it is so important, let’s go through this syntax bit by bit.

First: SELECT table1.var1, table1.var2, table2.var3

This clause stipulates which variables we want to take. If all of the variables in each table are unique (i.e. they don’t share any Field names) then you can drop the ‘table1’ part of the variable and just use ‘var1’, ‘var2’, ‘var3’.

Second: FROM table1 JOIN table2

This clause stipulates which two tables we want to take the data from. It can be useful to think of this clause as comprised of two separate sub-statements: [FROM] [table1 JOIN table2] i.e. we are taking values [FROM] the combined [table1 JOIN table2] entity.

Third: ON table1.var1 = table2.var1

This clause specifies which variables we want to join the two tables on. The variables do not need to share a name, but they must contain values found in both tables. A good example is in the class hand-out – we join on the user\_id variable and the rev\_user variable as both contain userids.

Fourth: ORDER BY var3 ASC LIMIT 10

This clause provides further information to the database so that the output which is returned is in a readable format.

The default JOIN command is an INNER JOIN; you can also specify LEFT JOIN, RIGHT JOIN or OUTER JOIN. See here to find out more: <http://www.w3schools.com/sql/sql_join_inner.asp>

Finally, we can also specify a double join, which is where we conjoin three tables within a single command (see the class handout for an example of this).

1. SELECT \* FROM table1 **WHERE var1 = value1**;

The WHERE command is a conditional statement – it’s a bit like ‘if’ in Python and R. Only those records which fulfil a specified criterion are returned.

If value1 is recorded as a numeric variable, then it can just be written as it is: WHERE var1= 1. If value1 is a character, then it should be enclosed in quotation marks: WHERE var1 = ‘string1’ (note: annoyingly, numbers are sometimes encoded as strings in SQL databases).

1. SELECT **COUNT(var1)** from table1;

COUNT() is a SQL function – as such the variable must be enclosed in brackets, as in the example here. COUNT returns a count of the number of rows in a table. For a list of other functions which you will regularly use (Max, Min, etc.) see here: https://www.tutorialspoint.com/sql/sql-useful-functions.htm

It can be used in conjunction with a WHERE statement. The syntax is:

SELECT COUNT(var1) FROM table1 WHERE var1 = value1;

COUNT can also be used with the DISTINCT command. DISTINCT returns only unique values. This is useful if, for instance, we have a table of n tweets produced by k users, and we want to extract the number of unique users (k). The syntax for using DISTINCT is:

SELECT COUNT DISTINCT(var1) FROM table1;

Often, we use the AS command with COUNT. AS lets us rename the variable directly within the output. This is particularly useful if the commands we have used (such as COUNT()) produce ugly variable names. The syntax is:

SELECT DISTINCT COUNT(var1) AS unique\_count FROM table1;

NOTE: COUNT DISTINCT (var1) is different to DISTINCT COUNT(var1)

In the first case we take only the distinct values of var1 and then count them (this is useful for getting the number of e.g. unique users in a dataset). In the second case we count every user and then only keep the distinct totals. This may be useful in some cases, but you will probably use it less frequently

1. SELECT COUNT(var1), var2, var3 FROM table1 **GROUP BY var2** LIMIT 10;

GROUP BY lets you group results from a table together. It is typically used in conjunction with aggregate functions, such as COUNT(). It is really useful if you have selected many rows from a dataframe and want to summarise the results based on one of the variables. So, for instance, you might want to summarise the number of Wikipedia entries produced by different users. A GROUP BY command would do this quite easily.

1. SELECT var1 FROM table1 WHERE var1 **LIKE ‘pattern’**;

LIKE is used with the WHERE command to search for a specified pattern within a variable. This differs from the use of WHERE in number 9 (above) as in that case there was a precise value which the entries had to equal. Here, we are just searching for a string within the entry.

If you know exactly what pattern you want to search for then you can stipulate it directly in the LIKE command: e.g. WHERE var1 LIKE ‘Kanye’

Normally, LIKE is used in conjunction with wildcard characters. Wildcard characters can be used to substitute for other character(s) in a string. To find out more go here: <http://www.w3schools.com/sql/sql_wildcards.asp> The key operators are:

1. \_ Is a substitute for a single character. So, if you think there is a common mis-spelling of Kanye than you can use the \_ to identify it: WHERE var1 LIKE ‘K\_nye’
2. % is a substitute for zero or more characters. You can use % if you are interested in returning at least one specific part of the word: WHERE var1 LIKE ‘%ye’

OR you can use multiple % in a clause: WHERE var1 LIKE ‘%any%’

This is particularly useful if there is a term which could occur anywhere within the string – for instance, “%bot%” in the user\_name of Wikipedia editors.

1. SELECT var1 FROM table1 WHERE var1 **NOT** LIKE ‘pattern’;

The NOT command is very straightforward. In this case, it specifies to return all cases which do NOT match ‘pattern’. NOT is a logical Boolean operator, the same as in Python and R.

1. SELECT var1 FROM table1 WHERE var1 NOT LIKE ‘pattern1’ **AND** NOT LIKE ‘pattern2’;

The AND command is also straightforward. It stipulates a second criterion that must also be met for the entries can be returned. Records will be returned only if both the first condition AND the second condition are true. AND is the exact same as in Python and R.

1. SELECT var1 FROM table1 WHERE var1 NOT LIKE ‘pattern1’ **OR** NOT LIKE ‘pattern2’;

Just like AND and NOT the OR command is very simple, and can be interpreted in the exact same way as equivalent commands in Python and R. Records will be returned if either the first criterion OR the second criterion are met.

AND and OR can be easily combined by using brackets. In the code below, only entries where var1 equals value1 AND var2 equals either value2 OR value3 are returned:

SELECT \* FROM table1 WHERE var1 = ‘value1’ AND (var2 = ‘value2’ OR var2= ‘value3’);

\*\*\*\*END OF SQL GUIDE \*\*\*\*

Please email [bertie.vidgen@oii.ox.ac.uk](mailto:bertie.vidgen@oii.ox.ac.uk) if you have any queries, see any errors, or think of any improvements.