# MySQL Starter Pack:

**Introduction:**

MySQL is a relational database management program, which we store, manipulate and extract data using Structured Query Language. The full documentation is available with this link <https://dev.mysql.com/doc/refman/8.4/en/> for reference, but I will try and give a brief outline of the basics and any parts you will likely need to use.

In MySQL, a record, also known as a row, is a single entry in a table. It contains specific data, such as information about a product or employee. A table also has columns, which are vertical entities that contain information related to a specific field. Each entry on each field must only and can only hold 1 point of data.

Columns or fields, have datatypes like int for integer signalling how data needs to be stored, and constraints like Unique signalling that every entry in that column must be distinct. Two very important constraints are primary key and foreign key. The primary key uniquely identifies each record, and in reverse each piece of data in the record must directly relate to that primary key. Each record only has one primary key, but the primary key may be composed of more than one field. [Foreign keys](https://dev.mysql.com/doc/refman/8.4/en/create-table-foreign-keys.html) are used to link tables and records by pointing towards a different field (usually a primary key).

For instance, we have a table named “Site” where each record contains information about a different national trust site. This will have a primary key which we have named “IDSite”. We may also have an alternative table named “Sensors” with a foreign key field named “IDSite” linking it to the site table. If “Mount Stewart” in the Site table has an IDSite of “1” and the IDSite of Sensor ”NTUB-A” is also 1 then it means “NTUB-A” belongs to/is related to Mount Stewart. We use Primary keys and Foreign Keys due to database normalisation which is an annoying and complicated topic which you can read about [here](https://en.wikipedia.org/wiki/Database_normalization) if you’re interested.

**Insert Statement:**

Insert Statements allow you to add new data to a database.

It takes the form:

INSERT INTO table\_name (column1, column2, column3, ...)  
VALUES (value1, value2, value3, ...);

Where table\_name, column1, value1 etc are placeholders. For an insert statement to be accepted by the database, every field marked “NOT NULL” without a default value must contain a value and any field marked unique contain a new unique value or be empty.

For instance this is the set of columns and constraints for a record in the “data” table:A screenshot of a survey

Description automatically generated

You may notice that idSensor is marked unique, this is technically incorrect as it would mean that each sensor may only have a single record. In actuality idSensor and Time combined would have to be unique, there may be multiple records for sensor NTUB-A and multiple records at 12:00 on the 15th of June 2023, but there must only be 1 set of data for NTUB-A at 12:00 on the 15th of June 2023

An acceptable query might take the form:  
INSERT INTO data (IDsensor, Time)  
VALUES (8,'2023-06-30 15:56:00');

As Temperature, Humidity and DewPoint are able to be empty and “ID” has AI for “Auto increment” marked, which means that the database will automatically give it a value.

You also don’t necessarily need to specify the columns as long as you fill in all the values in order.

INSERT INTO data  
VALUES (1, '2023-06-30 15:56:00', 8, 19.960, 67.8, 13.8, 1, 1, 1);

However such queries become much harder to read in my opinion.

**Update Statements:**

Update statements allow you to alter existing records.

They take the form:

UPDATE Customers  
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'  
WHERE CustomerID = 1;

“ContactName” is a column name and “Alfred Schmidt” is the new value in this example. Update statements will not be accepted if they violate any constraints similar to Insert statements.

The “WHERE” clause is crucial as it limits how many fields will be accepted. Where clauses contain Boolean statement and you can concatenate them with the “AND” or “OR” connectives.

E.G

UPDATE Data  
SET TemperatureValidity = 0  
WHERE Temperature > 50 OR Temperature < -50;

Be careful as omitting the where clause will mean that you will affect every record in the table, and potentially the database.

**Delete Statements:**

Delete statements allow you to remove data from the database.

They take the form:

DELETE FROM table\_name

WHERE condition;

The where clause is the same as described above, and omitting it will remove all the data on the table.

**Select Statements:**

Select Statements allow you to query the database for a specific subset of the data.

They take the general form:

SELECT Column1, Column2, Column3

FROM table\_name

WHERE condition

ORDER BY Column1, Column2;

The order clause only works for columns that you have selected, e.g you cannot order by time unless you have also selected time. You can omit the “ORDER BY” clause if you don’t require the data to be in any specific order, and omitting the “WHERE” clause will mean that the entire table is queried.

You can also use \* in place of column names if you want to get the data for all the columns.

SELECT \* FROM table\_name;

Finally you can query for averages or counts by using aggregate functions, which take into account multiple values in the database.

Below is a query I have used to select the number of records where humidity is invalid for every day in the database:

SELECT*DATE(Time)* AS *date, COUNT(\*)* AS *count*

FROM *data*

WHERE *humidityvalidity = 0* AND *IDSensor* IN *(1,2,3,4,5,6,7,8,9,10)*

GROUP BY *date*

ORDER BY *date;*

First of all in the database time is stored using the DATETIME structure, which can be separated into subparts like just the hour component or month component. The DATE() function selects just the day component and the “AS date” part is just me temporarily renaming the columns for the duration of the query so they are a bit more readable.

Secondly “WHERE humidityvalidity = 0 AND IDSensor IN (1,2,3,4,5,6,7,8,9,10)” limits the number of selected rows to both where humidity is invalid and sensor ID is between 1 and 10 inclusive, which are the sensors at Mount Stewart.

Finally COUNT(column) is an aggregate function that counts the number of rows where the column is contains a value. COUNT(\*) counts the number of rows where any column contains a value. This works in conjunction with the “GROUP BY” clause which divides the data into smaller groups before they are counted. Like “ORDER BY”, “GROUP BY” can only work with column names already selected. If you omit the “GROUP BY” section then it will simply apply count to all the data selected. “count(\*) AS count… GROUP BY date” as a whole groups the data by day before counting any row with at least 1 non null value, more simply it counts the number of selected rows per day.

In totality, this query selects the counts how many records have invalid humidity within Mount Stewart for every day and then orders them by day.

There are many useful aggregate functions such as finding the maximum, average, variance etc and you can find the full list of them [here](https://dev.mysql.com/doc/refman/8.4/en/aggregate-functions.html).

**Sub-Queries and JOIN clauses:**

In the above query we used “IDSensor IN (1,2,3,4,5,6,7,8,9,10)” to limit the data to only be records from Mount Stewart, however this approach has a few issues. First of all it’s not immediately clear and readable what relation sensors 1 to 10 have to one another and secondly if I asked you to adjust this code to get the same data but for the sensors in Sheffield Park, you would need to know the exact sensor ID for every sensor in Sheffield Park. Currently this isn’t an issue as there are only 2 sites and 24 sensors, but maybe when there are 5 sites and 60 sensors it will become a bit trickier.

One way might be first querying for the site ID using the name “Sheffield Park”, then querying for the list of Sensor IDs using the site ID, and finally using the list of Sensor IDs to query for whatever data you want, but clearly this is a bit of a hassle.

Instead you can either use Sub-Queries or JOIN clauses to do the entire thing in with one query;

Sub queries are queries within a query, the 2 queries you need to get the list of sensor IDs are:

SELECT idSite FROM site WHERE name = ‘Sheffield Park’;

and

SELECT idSensor FROM sensors WHERE idsite = ?;

You can combine them into 1 query like this:

SELECT idSensor FROM sensors

WHERE idsite in (SELECT idSite FROM site WHERE name = ‘Sheffield Park’);

The advantage of sub querying is reducing the overhead compared to making 2 separate queries. Subqueries can generally be any valid select query so you can have a query with a subquery that contains a subquery. Therefore a valid solution to adjusting the above code is something like this:

SELECT*DATE(Time)* AS *date, COUNT(\*)* AS *count*

FROM *data*

WHERE *humidityvalidity = 0* AND *IDSensor* IN

*(*SELECT idSensor FROM sensors

WHERE idsite in (SELECT idSite FROM site

WHERE name = ‘Sheffield Park’)*)*

GROUP BY *date* ORDER BY *date;*

Alternatively you can use the JOIN clause instead sub queries, JOIN combines 2 tables together so you can make queries using columns from both tables.

SELECT *s.idSensor* FROM *sensors s*

JOIN *site t* ON *s.idSite = t.idSite*

Where *t.name = 'Sheffield Park';*

So the SQL code above joins the tables together using the idSite columns on both, and then filters any sensors that aren’t Sheffield park. As column names between tables can overlap we have to distinguish them using “s.idSite” instead of just “idSite” denoting that it is the idSite that belongs to table “s” for sensors. S and T are arbitrary nicknames I have used for readability and the code below is also valid:

SELECT *sensors.idSensor* FROM *sensors*

JOIN *site* ON *sensors.idSite = site.idSite*

Where *site.name = 'Sheffield Park';*

JOIN clauses are generally better in terms of readability, scalability and efficiency than sub queries, especially so if indexing is involved.

We can now use the query above in a subquery, but since you can join more than 2 tables together this is a much better solution:

SELECT*d.DATE(Time)* AS *date, COUNT(\*)* AS *count*

FROM *data d*

JOIN *sensors s* ON *d.IDSensor = s.idSensor*

JOIN *site t* ON *s.idSite = t.idSite*

WHERE *humidityValidity = 0* AND *s.name = ‘Sheffield Park’;*

GROUP BY *date* ORDER BY *date;*

**Python:**

To connect to mysql with python you need the MySQL.Connector module. I used an older version of this module and worked in python 3.6 because the more recent version caused issues in my environment, but maybe the later versions will work better on your machines.

**import** mysql.connector

# Establish the connection

conn = mysql.connector.connect(

host="your\_host", # e.g., "localhost" or your MySQL server's IP

user="your\_username", # e.g., "root"

password="your\_password",

database="your\_database" # e.g., "test\_db"

)

# Create a cursor object

cursor = conn.cursor()

# Execute a simple query

cursor.execute("SELECT \* FROM your\_table")

# Fetch and print the results

**for** row **in** cursor.fetchall():

**print**(row)

# Close the cursor and connection

cursor.close()

conn.close()

So above is the basic code you would need to implement to connect to an SQL database and execute a simple query. The comments explain what is going on quite will I think and I have included a blank-ish version of this code named blankQuery.py which adds some error and exception handling.

The only bit of information I would add is that you can separate the data and the query if you want to like this:

query = "select \* from data where idSensor in (%s,%s,%s)"

cursor.execute(query, [1,2,3])

So here we define a query with multiple “%s” and pass it into the cursor.execute function with our list of data. Inside the execute function the %s is replaced by the data in the list in sequential order so the query that ends up being executed is "select \* from data where idSensor in (1,2,3)". If there is only 1 “%s” and 1 data value, you will still need to pass in the data as a list.