Workshop 3 – Advanced SQL

In today's workshop we'll look to extend your SQL skills, advancing your data handling and analysis capabilities to enable you to work with any type of dataset. We'll focus on querying across multiple datasets, making use of temporary tables, and working with spatial and temporal datasets.

The tutorial is broken down into five main sections.

1. Joining tables and querying joined tables
2. Using temporary tables
3. Setting up new table indexes
4. Working with temporal data
5. Working with spatial data

We'll be working with MySQL Workbench once more, and with the data tables you've been working with already. Like last week, just work your way through the worksheet; your tasks have been highlighted in **bold font**.

1 Joining Tables

In this section we're going to run through the various ways in which you can join and query multiple tables. So before we start, **open up MySQL Workbench and connect to your database**. Remember to set your database as 'Default Schema' before running your queries.

We'll work across both the Cities and Countries datasets. Inspect the structure and contents of both tables. **Which shared attributes would be most appropriate? And how will that choice affect the resulting datasets?**

We can start to investigate this by experimenting with different types of join.

Let's start with the most simple, joining two tables through the WHERE clause. This exports all records where any two fields match. **Run the following command**.

```
SELECT * FROM cities a, countries b WHERE a.country_cd = b.country_cd;
```

What results does this query give you? Notice how the table aliases (`a` and `b`) are used to denote which field has being referenced where tables have matching names.

We can limit our results in the same way as any other query too. This just requires careful specification of the columns and table aliases. **Run this next command to see this in action**.

```
SELECT a.name, b.name, a.lat, a.lon FROM cities a, countries b WHERE a.country_cd = b.country_cd;
```

This pulls out the city name, country name, and the city location. Notice that we don't have to include the `country_cd` data in our results, despite it being the field we use to join the tables.

Nevertheless, the matching column names from this query could be a bit confusing – is it always clear which name attribute refers to country, and which one to city? To where does the latitude and longitude refer? **Add some aliases to the columns to make the results a bit more clear.**

Now, using these methods, and the skills you learn last week, **limit the results to export only the city name, the country name, and the proportion of the country's population that live in that city.** Remember to use the table aliases in the same way above to select the correct columns.

This query demonstrates how we can combine datasets to extract new understanding from our standalone tables.

Finally, we can add other clauses to the existing WHERE clause, to further specify the results returned. **Add another WHERE clause to limit the join to only cities in Europe**.

Remember, if you want to check your answer then ask the Slack AnswerBot! When you see the blue circle, use the question number to ask the AnswerBot for help or a hint.

Next, we'll use the JOIN syntax to join two tables. This works in the same way as the WHERE method above, but is a bit clearer (which is particularly important where joining more than two tables). **Try running this query.**

```
SELECT * FROM cities a
JOIN countries b ON a.country_cd = b.country_cd;
```

This query pulls up the same results as before. Let's try changing the columns on which the join is formed. **Change the query to link city name to the country's capital city** (which is spelt `captial` in the countries table, err, intentionally of course…).

This query returns only the cities that have a matching capital city name in the country table. However, is this actually what we want? **Have a careful look at the results, notice anything wrong?** (Hint: Order the results alphabetically by city name, and go to London).

The problem here is that we have *not been specific enough* in joining the tables. By just linking by name, we are including instances where multiple matches exist between records in each dataset.

To fix this, we need to add further clauses onto our JOIN, to ensure correct results. We do this like so, **try running this and check the results once more.**

```
SELECT * FROM cities a
JOIN countries b ON a.name = b.captial AND a.country_cd = b.country_cd;
```

Next, let's suppose we still want all of the records from the cities table, plus records from the countries table where a match exists. We do this using a LEFT JOIN. This is a really simple syntax change to that above. **Have a look back at the lecture notes and see if you can change the query above to the left join**.

See how the results change? We now receive back ALL of the city records, but only the country records where a match exists.

Now adjust this query to list all of the matching records first, followed by the unmatched city records.

One final thing to point out is that we can join tables to each other. This is known as a self-join, and is pretty useful in identifying duplicate records. **Let's try running this for the cities table**.

```
SELECT * FROM cities a
JOIN cities b ON a.name = b.name AND a.country_cd != b.country_cd;
```

Notice how this takes quite a long time to execute? Here we're comparing each of the 23000 records against all 23000 records, which works out to be a lot of operations.

The results give us a list of duplicate city records. If you're feeling extra daring, see if you can add in country data for both city tables (so using four JOINs in total). I'll leave this as an optional extra.

2 Using Temporary Tables

Temporary tables are very useful for when you're working with subsets of a larger table. Forming a straight join between two tables is computationally costly, as it runs through both tables to ensure all records are matched. As such, executing queries on joined datasets can become annoyingly lengthy process when dealing with large datasets.

To work more effectively with a subset of your data, you can create a temporary table from a simple SELECT query. This is pretty straightforward and is a similar syntax to what you would have seen in Workshop 1.

The following command creates a temporary table containing only UK city data.

```
CREATE TEMPORARY TABLE uk_cities AS (SELECT * FROM cities WHERE country_cd = 'GB');
```

As a temporary table, it will not appear in your Table list, **so execute a simple SELECT query to inspect the data.**

We can now use this table to simplify our joins. For example, remember how long the full self-join took between the two cities tables earlier? **Using that query, try changing one of the cities tables for the `uk_cities` tables**.

This query provides yields a list of cities from across the world whose name matches the name of a UK city. **Now – just because you can – find the average population of this subset of world cities and check the answer with AnswerBot.**

3 Setting Up Indexes

Indexes were mentioned in the first week. A lot can be said about indexes but they're basically look up tables that improve the speed at which queries and joins are executed.

It's worth you knowing how to add an index to a table, as it'll really help you in your future SQL endeavours. Plus MySQL Workbench makes this pretty easy.

Before we add an index, I want to demonstrate the impact an index can make. So, first, **run this query again**.

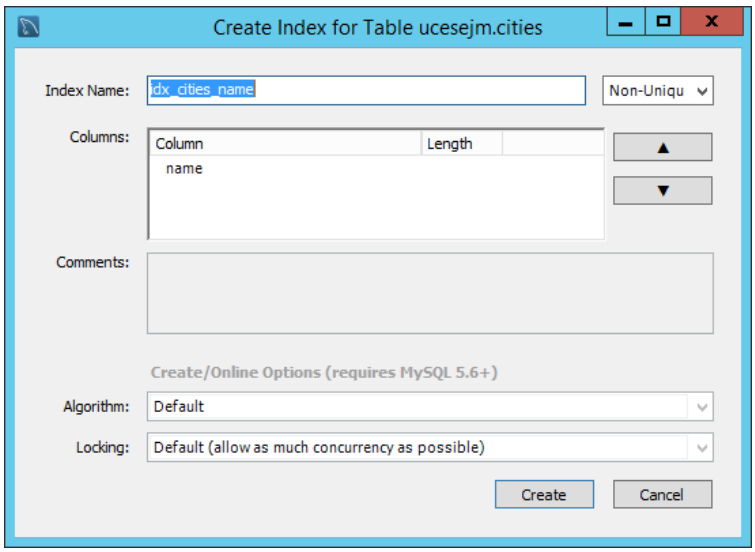
```
SELECT * FROM cities a
JOIN cities b ON a.name = b.name AND a.country_cd != b.country_cd;
```

Wait until it finishes and look in the Output window at the bottom of the MySQL Workbench screen. You'll see a column that reads 'Duration/Fetch'. **Find the row relating to the query above and note the results from the Duration/Fetch column**. I got back `17.469 sec / 85.547 sec`, indicating that the query took 85 seconds to run in total.

To speed this query up, we'll add two new indexes to the City table. These need to be added to the columns we are using within the WHERE section of the query (namely, the columns that we're searching through to form the join). To do this, **complete the following**:

1. Go into Table Inspector (Right Click – Table Inspector), and hit the Indexes tab. You'll see that there is an index already existing on the `id` column, which would not help for our query above.

2. Click on the `name` column in the lower window, and hit the 'Create Index for Selected Column…' button. The window below will appear.



   The main thing to pay attention to in this window is the drop-down list in the top-right of the window. The options hear allow you to specify the type of index this will be – whether it is unique or non-unique, or whether it is a spatial index. For now though, we'll use the default settings.

3. Click Create and the index will now be created.

4. Next, carry out the same procedure for the `country_cd` column.

Now we've created these indexes, try running the same query as above. How long does it take to run this time? I got 0.23 seconds this time :)

If you still don't believe it makes this much difference, go back into the Indexes page and remove the indexes (using the Drop Index button) again.

4 Working with Temporal Data

We'll now turn our attention back to the Flickr data you were working with in Workshop 1, and start playing with temporal and spatial data.

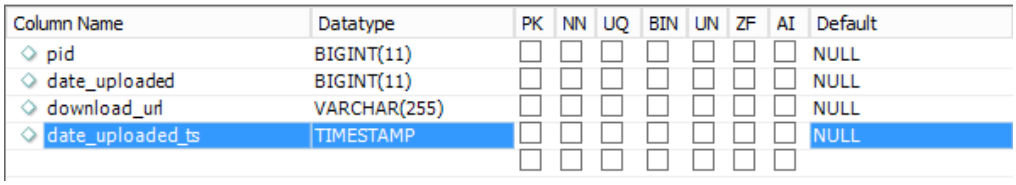
Open the Photos data and look at the `date_uploaded` field. You'll see that this data doesn't look like a date or time format you would have seen before. This is because it is Unix time (which is actually the number of seconds past 01/01/1970, but don't worry about that for now). You may well come across Unix time where handling datasets, so it's worth knowing how to make use of it. SQL has a nice method for converting it back to something more understandable.

Helpful Tip #564133 It would actually be more efficient to just work with the Unix date and time data directly, by including the `FROM_UNIXTIME(date_uploaded)` in our queries either in the SELECT or WHERE sections. But today we'll create a new column, even if it does duplicate the data.

The method we need for handling Unix time is called `FROM_UNIXTIME()`. We'll use this function to first convert our date and time data to something useful before proceeding. To do this, **following these steps**.

1. Right Click on the Photos table, and go to 'Alter Table…'. This window allows you to amend the table structure without dealing directly with the SQL.

2. Next we want to add a new column to the Photos table, to contain the date and time data in a more useful format. Click on the empty row at the bottom of the list of columns, add the Column Name '`date_uploaded_ts`' and change the Datatype to Timestamp (you can find it in the drop down list). Your new column list should now look like this.



Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Default
pid	BIGINT(11)	☐	☐	☐	☐	☐	☐	☐	NULL
date_uploaded	BIGINT(11)	☐	☐	☐	☐	☐	☐	☐	NULL
download_url	VARCHAR(255)	☐	☐	☐	☐	☐	☐	☐	NULL
date_uploaded_ts	TIMESTAMP	☐	☐	☐	☐	☐	☐	☐	NULL
		☐	☐	☐	☐	☐	☐	☐	

   Click the Apply button and confirm the change.

3. Now we have the new column, we need to populate it. We'll do this with the `UPDATE` command you learnt last week. Simply run the following command to populate the new column:

```
UPDATE photos
SET date_uploaded_ts = FROM_UNIXTIME(date_uploaded);
```

4. Open the Photos table and take a look at the new column.

Next we can start working with the new timestamp field, and use it within our queries. Given your knowledge of SQL queries to this point, this should be quite straightforward.

You can treat a timestamp like a number, in that you can use simple logic conditions within the `WHERE` clause. An example would be the following, **try running it**.

```
SELECT * FROM photos WHERE date_uploaded_ts <= '2010-01-17';
```

Note that, even though the timestamp contains both the date and time elements, we can are using only the date within this query.

Next, try changing the query to extract only photos taken on the 17th January 2010 after 19:00 and before 21:00, and order them in increasing time. How many records do you get back?

By the way, you can have a look at some of the photos by pasting the URL into a web browser.

5 Working with Spatial Data

In this last section we'll have a quick play with MySQL's spatial data formats. As you might have seen in the Flickr `photo_locations` table a few weeks ago, the table contains `POINT()` data for the locations at which the photographs were taken.

MySQL has a number of spatial analysis functions that allow you to quickly and easily extract records based on their spatial data. For this workshop, we'll just focus on the `Contains(g1, g2)` function, but all of the functions work basically in the same way, each comparing two geometries (denoted `g1` and `g2` above). The `Contains()` function takes two geometries, and tests whether the second geometry is contained within the first geometry.

For this example, we'll query whether a photo location falls within a `POLYGON` geometry. To do this we need to define a polygon within which to search. The creation of a new polygon from a set of coordinates requires a particular function, which shown below, taking a number of coordinates (as text inputs) and produces a `POLYGON` geometry object.

This particular set of coordinates will create a complete square around UCL.

```
ST_GeomFromText('POLYGON((-0.1463 51.5333, -0.1222 51.5333, -0.1463 51.5171, -0.1222 51.5171, -0.1463 51.5333)) ');
```

We can then embed this text within the `Contains()` function to create one of our geometries. The second geometry will be our photo location.

Bringing these functions together gives us the following command. **Run this to find all of the photos taken around UCL.**

```
SELECT * FROM photo_locations
WHERE ST_Contains (ST_GeomFromText('POLYGON((-0.1463 51.5333, -0.1222 51.5333, -0.1463 51.5171, -0.1222 51.5171, -0.1463
51.5333)) '), coords);
```

You should have a returned dataset containing data relating to 308 photos.

As a final task, **join these results to the photos table through the shared `pid` value, using whatever join method you like**. Check out some of the photo URLs and see if you find anywhere familiar.

Homework – Complete the second SQL Quiz on Moodle
Don't worry, it's not assessed, it's really easy and very short.