University of Sunderland

Technology Management for Organisations
Workshop 6
Relational Databases

Aims of the workshop

In lectures we have looked at Relational Databases, the relational model, and the ER Model as well as querying via SQL. We explored the role of a Database Management System including the client <-> server interface, as well as underlying storage engines and the principles of ACID Compliance. From this we then introduced ORMs as a programmatic way of interfacing with a RDBMS, and creating a mapping between Object-Oriented code, and Relational data within a RDBMS.

In this workshop we will extend our existing knowledge of List and Dictionary comprehensions, and introduce the Generator syntax. We will look at a common utility to Database Administration, PhpMyAdmin. We will further look at SQL, executing some preliminary queries on example database tables, and finally we will get hands-on experience with the ORM Library introduced in lectures: PonyORM.

You are encouraged to read around the topics introduced in today's workshop, many of these include documentation pages for both SQL and PonyORM.

Make sure to fill in your ePortfolio for Teaching Week 6 / Workshop 6 during/after the workshop.

Feel free to discuss the work with peers, or with any member of the teaching staff.

Reminder

We encourage you to discuss the contents of the workshop with the delivery team, and any findings you gather from the session.

Workshops are not isolated, if you have questions from previous weeks, or lecture content, please come and talk to us.

Exercises herein represent an example of what to do; feel free to expand upon this.

Exercises

With the exercises for this workshop, please discuss answers with a peer as well as a member of the delivery team.

For the exercises today we are hosting a database with the following credentials:

host: **europa.ashley.work**
username: **cetm50_user**
password: **iE93F2@8EhM@1zhD&u9M@K**
database: **cetm50**

Note, the environment we are using here is not typically done as we are exposing the database to the wider internet. Usually databases are restricted to only connections from localhost (127.0.0.1), the machine it is running on.
Note: For the more cybersecurity-curious amongst you, consider whether you can intercept the passwords being sent as part of the database connection.

Secure database connection / communications are beyond the scope of this particular workshop.

WARNING

Do NOT change the password for this user account whatsoever. These credentials are to be used by everyone.

Exercise 1 - Previously we have looked at List and Dictionary Comprehensions. Let's recap those briefly now.

Create the following List, this represents average overnight temperatures per day.

```
[ -1.3, 0.1, 2.1, 2.0, 1.5, 0.3, 0.2, 0.8, 0.1, -0.1, -0.9, -1.0, 0.5 ]
```

Create a List comprehension which filters this list for any temperature below, and including, 0.

Exercise 2 - Take the following table of stock names, prices, market cap, and Price-to-earnings ratios and write this as a dictionary.

You may wish to consider the following template as a hint:

```
{
      "UWMC" : {
          "price" : ...,
          ...
      },

      "QRTEA" : {
           ....
      },
      ...
}
```

	Price ($)	Market Cap ($B)	12-Month Trailing P/E Ratio
UWM Holdings Corp. (UWMC)	6.98	0.7	1.2
Qurate Retail Inc. (QRTEA)	10.36	4.2	3.1
Sage Therapeutics Inc. (SAGE)	41.42	2.4	3.3
Annaly Capital Management Inc. (NLY)	8.69	12.6	3.8
Sylvamo Corp. (SLVM)	27.64	1.2	4.2

Source: YCharts

Using a dictionary comprehension, make a new dictionary which provides the entire stock record for any stocks whose P/E ratio is greater than 2 and where their market cap is greater than 5 Billion USD.

Remember: A dictionary comprehension returns a new dictionary when evaluated.
E.g some_dict = { k: v for k, v in another_dict.items() }

Generator Expressions are similar to List and Dictionary Comprehensions. They follow the same general structure, but are defined using different brackets.

List: []
Dictionary: {}
Generator: ()

Perform the following:

```
#Generator Expression

num_cube_lc=[n**3 for n in range(1,11) if n%2==0]    #List
Comprehension
num_cube_generator=(num**3 for num in range(1,11) if num%2==0)
#Generator Expression

print(f"List Comprehension = {num_cube_lc}")
print(f"Generator Expression = {num_cube_generator}")

#sum(num_cube_generator)
print(f"Sum = {sum(num_cube_generator)}")
```

You should obtain the following output:

```
Output:
-------
List Comprehension = [8, 64, 216, 512, 1000]

Generator Comprehension = <generator object <genexpr> at
0x00000238925E7848>

Sum = 1800
```

The difference between these is that Generators are **lazy** (technical term). List comprehensions return the entire sequence. For small numbers and data, this is not that impactful; for large datasets, this can be incredibly memory and cpu intensive. Imagine calculating cubes for 99 Quintillion numbers, we would have to calculate them all before returning anything.
Generators only evaluate what is needed there and then (Lazy Evaluation), as such we can iteratively step through them getting the next value only when it is suited.

Try change the List Comprehension range above from 1 to 10*1000. Notice how long the List Comprehension takes to return (Take care not to run out of memory!).

Do the same for the Generator Expression and delete the line "`print(f"Sum = {sum(num_cube_generator)}")`". How long did it take to return?

Note: The reason we remove the sum() call is this would get all of the numbers. Which defeats the purpose of using lazy evaluation. If removing this line, you should notice that the list comprehension takes a long time to assign to num_cube_lc, and the generator expression assigns straight away.

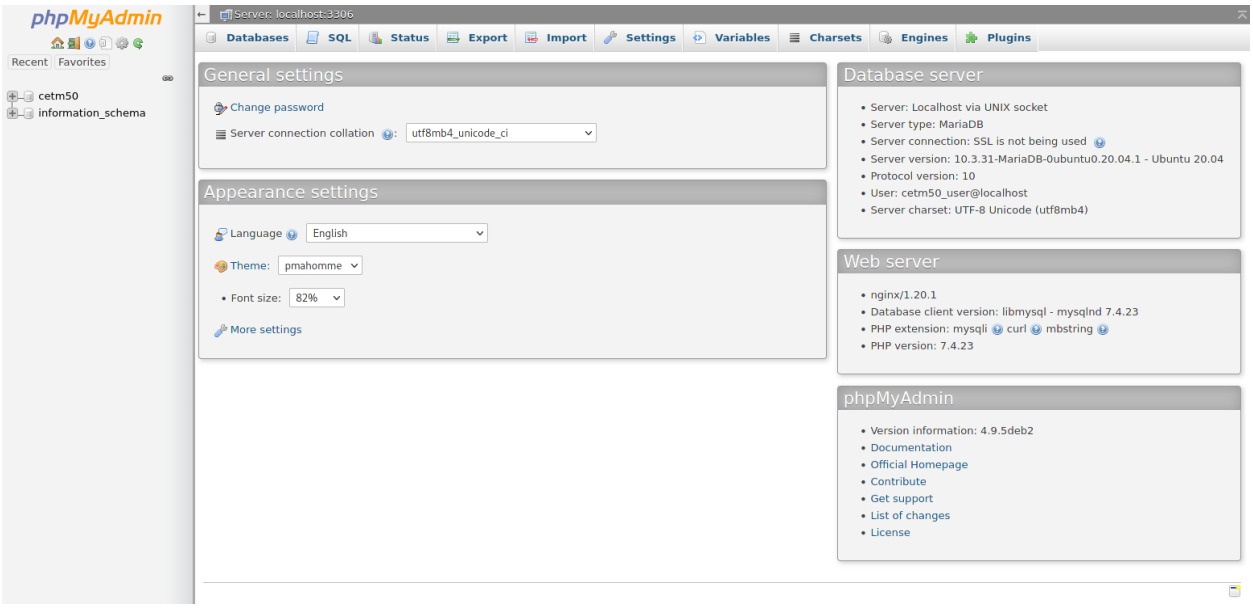
PhpMyAdmin is a Database Administrative web-portal written in PHP for MySQL/MariaDB Databases. It enables database users to log in and manage the database, tables, and any records therein. Additionally, it allows database maintenance operations, and user control.

As part of the database installation at europa.ashley.work we have installed this web portal for you to test.

Caution: Please do not change the password, modify data, or delete data. This is a single table, and a single user account which everybody must share. For now, look, explore, but do not make changes.

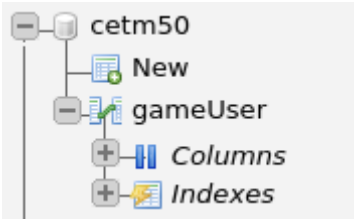
Exercise 3 - Navigate to https://europa.ashley.work/phpmyadmin you should be greeted with the following login prompt (if this is not the case, please contact a member of the delivery team).

Please login using the cetm50_user credentials from the beginning of the workshop. Once logged in you should see the following:



On the left-hand navigation panel you can see a list of databases which your current user has access to. As cetm50_user, you only have access to two: cetm50, and information_schema (a meta database).

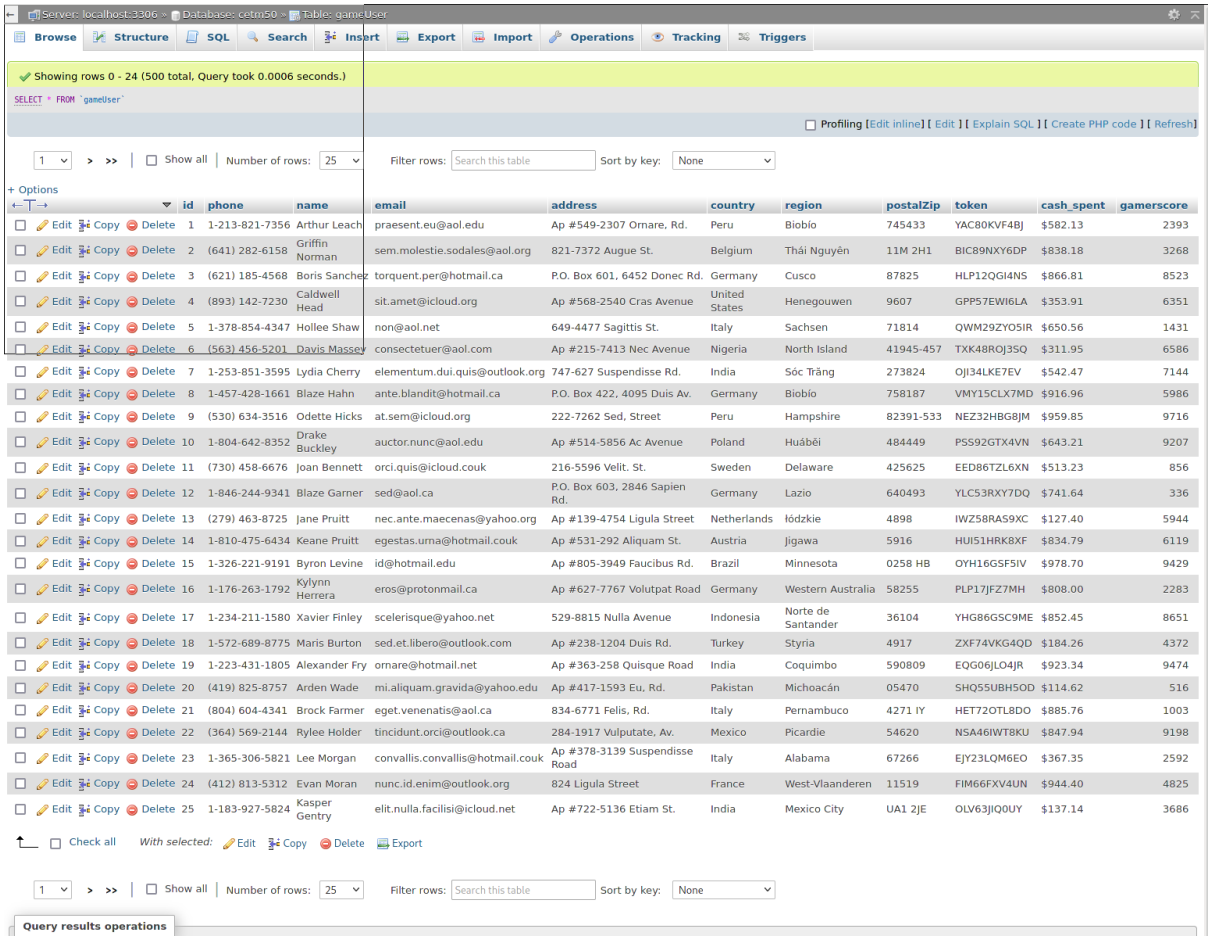
If you expand the cetm50 database you can see any tables which are defined under it. Likewise, expanding tables further shows properties of the table such as attributes.



Here we can see that we have a single table named **gameUser**.

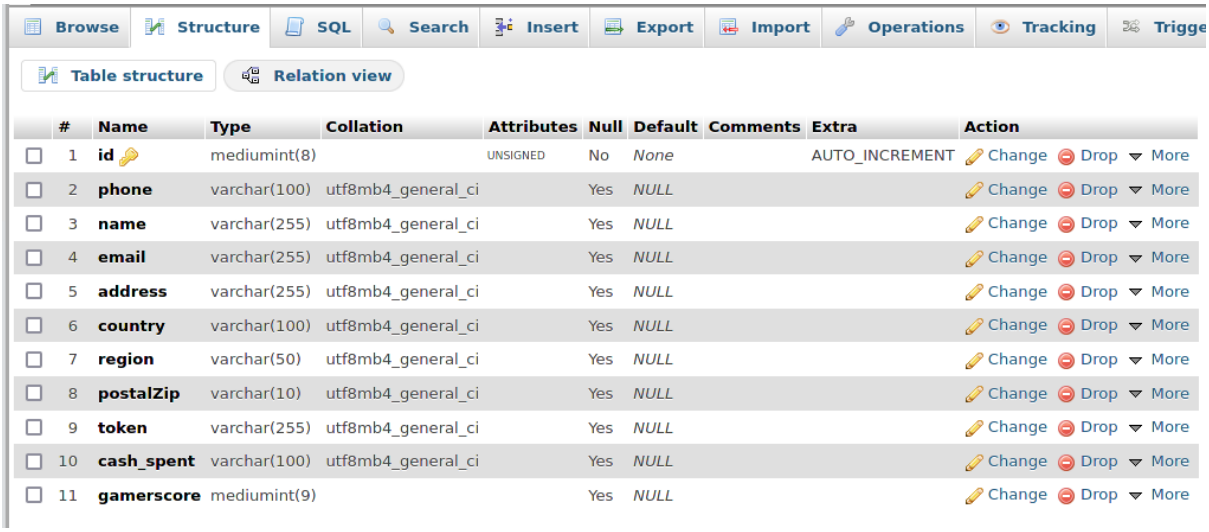
Exercise 4 - We can click on tables to view their structure, any data residing in them, as well as access any administrative actions which we're allowed to conduct.

Click on the **gameUser** table. By default we will see data which exists within that table. You can get back to this tab via the Browse Tab at the top of the Table view.



Here we are provided with the data residing in this table, as well as options to modify, copy, or delete specific records. Notice how MySQL provided an 'ID' column for each record, this was not input manually. This was created to act as a primary key, as one was not provided when this table was constructed and data input.

Moving along the tabs at the top, we can view the structure of the Table. This is equivalent to the Schema, what data must conform to in order to reside within this table.

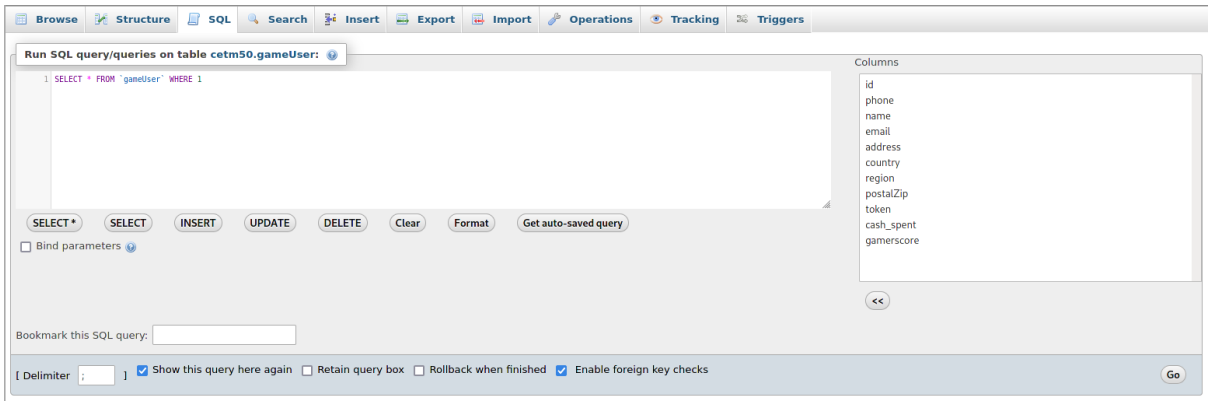


#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	mediumint(8)		UNSIGNED	No	None		AUTO_INCREMENT	Change Drop More
2	phone	varchar(100)	utf8mb4_general_ci		Yes	NULL			Change Drop More
3	name	varchar(255)	utf8mb4_general_ci		Yes	NULL			Change Drop More
4	email	varchar(255)	utf8mb4_general_ci		Yes	NULL			Change Drop More
5	address	varchar(255)	utf8mb4_general_ci		Yes	NULL			Change Drop More
6	country	varchar(100)	utf8mb4_general_ci		Yes	NULL			Change Drop More
7	region	varchar(50)	utf8mb4_general_ci		Yes	NULL			Change Drop More
8	postalZip	varchar(10)	utf8mb4_general_ci		Yes	NULL			Change Drop More
9	token	varchar(255)	utf8mb4_general_ci		Yes	NULL			Change Drop More
10	cash_spent	varchar(100)	utf8mb4_general_ci		Yes	NULL			Change Drop More
11	gamerscore	mediumint(9)			Yes	NULL			Change Drop More

We are provided with options to change, or remove attributes. Changes here would be specifying the data type of each attribute. Notice how different ‘string’ fields contain different lengths of varchar. Some are length 10, others 50, some 100, some 255. If we know that a field has a fixed maximum length, we should not allocate more space than required. E.g If each record reserve 255 characters for the PostCode, then we waste 245 characters worth of empty space for every single record we enter into the database!

Calculate how much wasted space this would be for our table here, you may have to look at browse to figure out how many records we have.

Finally, we have the SQL tab. This is where we can directly execute SQL queries against this specific table. PhpMyAdmin handles the context for us in this case.



Execute the query which is filled in for you. The ‘Go’ button is in the bottom right.
Note: PhpMyAdmin will parse and warn you of any invalid SQL entered. It will NOT correct semantics.

Exercise 5

As mentioned in the lectures, SQL is the standard for querying and manipulating databases.

If you are interested in the full grammar of the language you may find the following a useful resource:
https://gists.rawgit.com/GuntherRademacher/ad6a447fdc0d0c331c6ba24cc39882c4/raw/bc69f06fe4eb9f243b69e889657ca8763c6441ab/sql_yacc.xhtml

This outlines diagrammatically the language structure in the form of a railroad diagram. Elements here are interactable (clickable) so you can follow their definitions.

Alternatively you may find https://mariadb.com/kb/en/sql-statements/ useful as this breaks down SQL into the areas outlined from the slides.

Lookup all statements covered in the lecture slides in full from their documentation page.

Exercise 6 - Using the SQL Query tab from PhpMyAdmin on our gameUser table perform the following:
Note - the Query tab has GUI elements for helping you select and construct your query appropriately

1. Select only the e-mail addresses from the table
2. Select all users, where their gamerscore exceeds 9000
3. Select all users, where their gamescore exceeds 9000, ordering the result based on this gamerscore.

Exercise 7 - Using the Table Schema as a basis, construct a fake record.

1. Come up with some valid values for each of the fields/attributes you would need.
2. Using SQL and the INSERT keyword, form an SQL string which will insert your new person into the table.
    a. You may construct and execute this within the SQL Query tab of PhpMyAdmin; consider pressing the ‘INSERT’ button for a helping hand.
3. Verify that your new record has been added to the Table
    a. You may do this manually via Browse
    b. Or, you can construct a SELECT statement WHERE you can identify them via their e-mail address. E.g I might do WHERE e-mail = “uos@ashley.work”

Exercise 8 - Using UPDATE, modify just your fabricated record, and change one or more fields to some new values. E.g You may want to improve your gamerscore, and reduce the money you’ve spent.

Exercise 9 - Using SQL, create a DELETE SQL Statement which will only delete the fabricated data you made from the previous exercise.

Hint: You may wish to tick ‘Rollback when finished’ as a dry-run to ensure you delete the right number of records. Un-check it when you are certain.

PonyORM

Exercise 10 - Before we can use PonyORM to connect to our database, we must first ensure we have installed the relevant library.

Install pymysql and pony via PIP. (https://pypi.org/project/PyMySQL/ and https://pypi.org/project/pony/)

Exercise 11
In order to do anything with the database we must first create a connection. This involves creating a Database object, one for each database we want to talk to (for now, we just want the one). Ensure you have imported Database from pony.orm

```
from pony.orm import Database

db = Database()
```

At the moment, this doesn't do anything, as no connection is tested. In order to test our connection, let's attempt to bind to our Database. Usually this will take any Data Models we have defined, and map them to the Relational Model; however, for now this will just test the connection. If anything is wrong, host unreachable, credentials incorrect, database driver missing, this step will error. If nothing seems to happen, then it's working!

```
# MySQL
db.bind(provider='mysql', host=' ', user=' ', passwd=' ', db=' ')
```

Make sure to populate the arguments here with the information at the top of the workshop! Run your Python and you should have absolutely nothing happen.

Exercise 12
As nothing is happening, let's use this bound Database connection to execute some SQL manually.

Whenever we are dealing with querying and database entities, we should wrap our code in a context manager called **db_session**

Add **from pony.orm import db_session** to the top of your file (or where your other imports are)
After our db.bind call, add the following:

```
with db_session:
      my_query_result = db.select("SELECT * FROM gameUser LIMIT
10;")
```

```
print(my_query_result)
```

You should obtain a list of results, where each element is a tuple in the order of the attributes within the table.

Verify the data types and structure hierarchy of the results provided, including the number of elements returned.

Exercise 13
Using this manual SQL execution approach, perform the SQL queries you previously executed within PhpMyAdmin.

Where you fabricated data, you should create Python variables (List, Dict, etc) to represent your 'User'.

For inserting variables into a string, we can use Python's **f-strings** for this. This is arguably one of the best additions to Python in recent years, and came into the language with Python 3.6. A primer on f-strings is available here: https://realpython.com/python-f-strings/

f-strings (format strings) enable us to directly put variables into strings without having to concatenate multiple strings together in an unreadable complex way.

my_name = "Ashley"
print("My name is " + my_name)

Format strings allow us to directly place them in the string. First we must prepend the string with an **f**. E.g "My name…" becomes f"My name…". Once we have a format string, anytime we have a matching pair of curly braces { }, we treat that as a substitution.

E.g print(f"My Name is {my_name}")

This will print **My Name is Ashley**. It will lookup the variable my_name within the current frame (local and global scopes), and insert it directly into the string. Notice how the { } themselves aren't included.

For Example, we could change the LIMIT from the previous SQL Query to some variable LIMIT.

```
with db_session:
      some_limit = 25
      my_query_result = db.select(f"SELECT * FROM gameUser LIMIT
{some_limit};")
      print(my_query_result)
```

END OF EXERCISES