# Databases Lab 3 (MySQL)

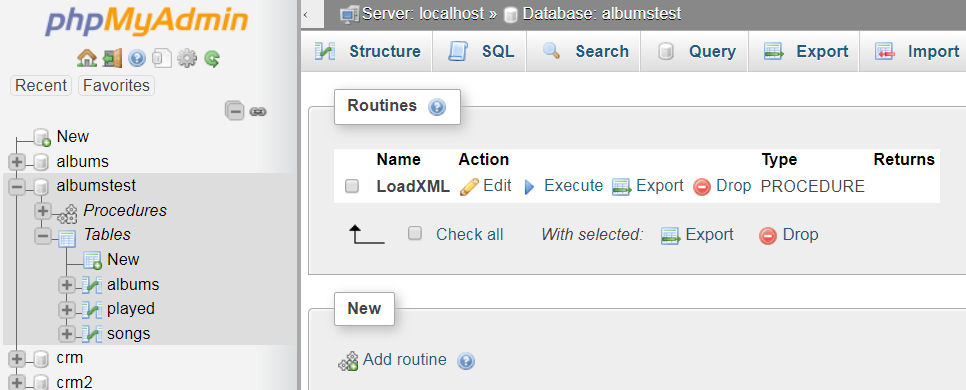
We are going to continue to practice some SQL queries. First we are going to upload and run an SQL script which will create some tables and then upload data to those tables.

## Exercise

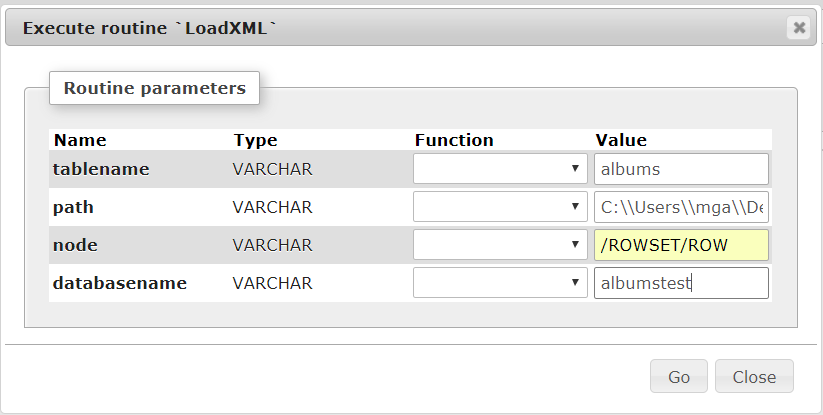
Run the supplied script as described in Lab 1 to create the database and tables.

To upload the data we are going to use a MySQL stored procedure to traverse the supplied XML data files and insert rows into the tables.

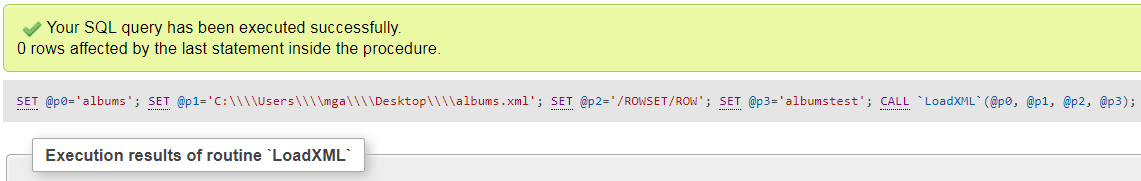
Select the **Procedures** for the database you have created:



Click the **Execute** option and enter values for the four parameters as detailed below:



You may have to modify the database name and the path to where your XML files are stored.

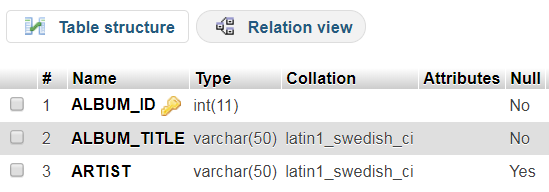


Run the procedure again to load the songs data.

## Exercise

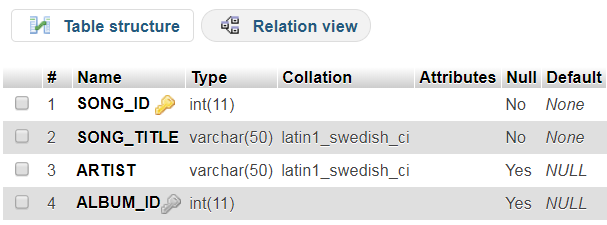
You can now view the created tables and uploaded data through the **Object Browser**:

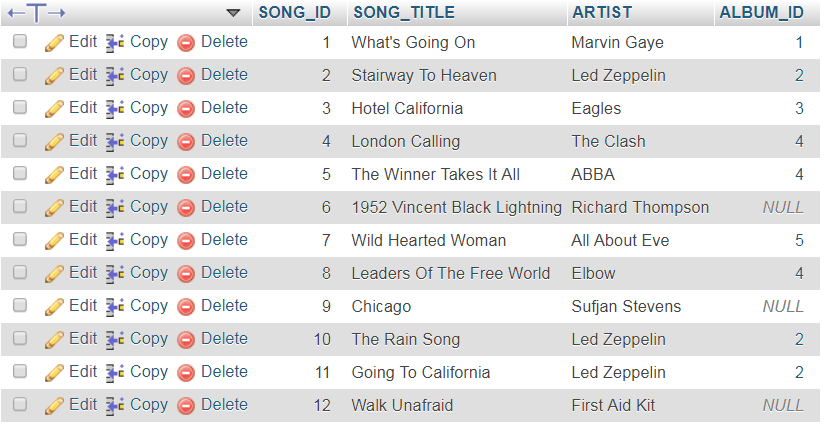
1. **Albums**





1. **Songs**





### Exercise

Enter the following SQL commands into the **SQL Editor** and check you see the results depicted; ensure you understand the queries:

1. **Display albums that have songs**

Recall that a relationship between the *Albums* and *Songs* table is through the primary key-foreign key relationship. An album can have many songs so each row in the Songs table can be linked to a single row in the *Album* table using the primary key value of the album (album\_id) as a foreign key value in the song (album\_id).

To achieve this we use an inner join which will return all rows where there is at least one match in both tables i.e. albums that have songs linked to them through the foreign key album\_id value:

**Select Album\_Title "Album", Albums.Artist "Album Artist",**

**Song\_Title, Songs.Artist "Song Artist"**

**From Albums**

**Inner Join Songs**

**On Albums.Album\_Id = Songs.Album\_Id**



An inner join will ignore *Album* rows where there are no *Songs* and *Songs* which are not linked to an *Album*.

1. A left join can be used to select all rows from the first (left) table and matches from the second (right) table.

Here we can use it to list all the albums and any songs linked to albums i.e. where the album\_id value for a song is not NULL:

**Select Album\_Title "Album", Albums.Artist "Album Artist",**

**Song\_Title, Songs.Artist "Song Artist"**

**From Albums**

**Left Join Songs**

**On Albums.Album\_Id = Songs.Album\_Id**



1. A right join will select all rows from the second (right) table and those rows from the first (left) table which have a match.

Here we are listing all *Songs* and any *Album* details which match:

**Select Album\_Title "Album", Albums.Artist "Album Artist",**

**Song\_Title, Songs.Artist "Song Artist"**

**From Albums**

**Right Join Songs**

**On Albums.Album\_Id = Songs.Album\_Id**



1. A full outer join will return all rows where is a match in one of the tables. Note in MySQL we need to use a UNION clause. Here we list all albums and songs:

**(Select Album\_Title "Album", Albums.Artist "Album Artist", Song\_Title, Songs.Artist "Song Artist"**

**From Albums**

**Left Join Songs**

**On Albums.Album\_Id = Songs.ALBUM\_ID)**

**UNION**

**(Select Album\_Title "Album", Albums.Artist "Album Artist", Song\_Title, Songs.Artist "Song Artist"**

**From Albums**

**Right Join Songs**

**On Albums.Album\_Id = Songs.ALBUM\_ID)**



1. A cross join (Cartesian product) will create rows by matching all rows from the second (left) table to each row of the first (right) table:

**Select Album\_Title "Album", Albums.Artist "Album Artist", Song\_Title, Songs.Artist "Song Artist"**

**From Albums**

**Cross Join Songs**



Each of the 12 rows in the *Songs* table is combined with each of the 7 rows in the *Albums* table to produce a result set of 84 rows.

1. A natural join will attempt to create a join by looking for the same column in each table; in this case, album\_id appears in both tables so those columns are used:

**Select \* From Albums**

**Natural Join Songs**



A left natural join:

**Select \* From Albums**

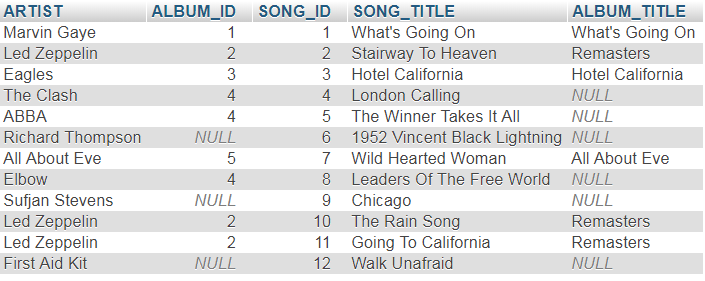
**Natural Left Join Songs**



A right natural join:

**Select \* From Albums**

**Natural Right Join Songs**



1. A **Union** clause is used to combine two SQL statements which retrieve the same columns:

**Select Artist From Albums**

**UNION**

**Select Artist From Songs**

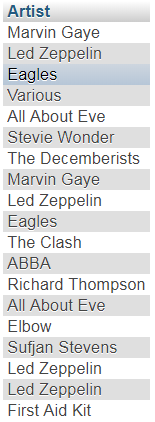


This will remove duplicates; use **ALL** to retrieve all matches:

**Select Artist From Albums**

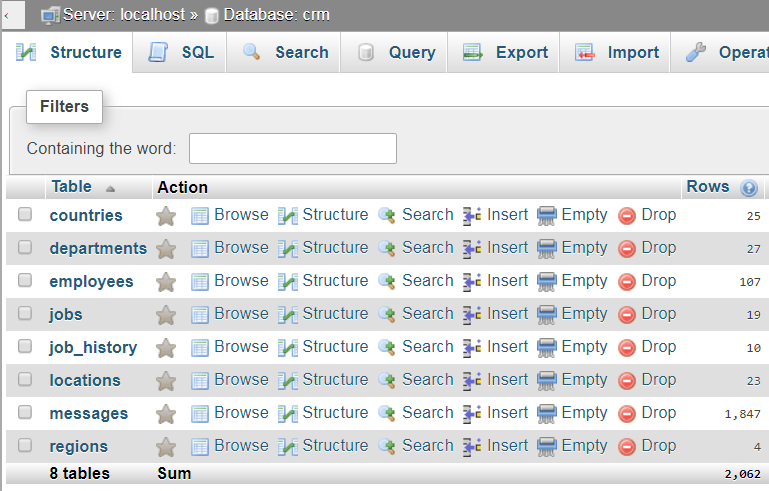
**UNION ALL**

**Select Artist From Songs**



### Exercise

1. Upload and run the *users\_script.sql* script, as described earlier, to create the following tables if they do not already exist in your database :
   1. Employees
   2. Departments
   3. Locations
   4. Countries
   5. Regions
   6. Jobs
   7. Job\_History



1. Note the **Foreign Key** constraints for each table and use these to construct an **Entity Relationship Diagram** for the tables above.
2. Upload the data for the following tables from the supplied XML files using the **LoadXML** routine (in the listed order): **Jobs**, **Regions**, **Countries** & **Locations**.
3. Note that for the remaining tables the **Foreign Key** constraints will cause issues in uploading the data therefore the stored procedure turns off this check. Upload the XML data for the remaining tables: **Departments**, **Employees** & **Job\_History**.

### Exercise

Write and test SQL queries to answer each of the requests below:

1. Retrieve the full name (concatenated and with a nice column header), job title, department id, department name and salary for all employees. (106 rows)
2. Retrieve the full name, job title, department id, department name and salary for all employees. Order the result by department id in ascending order (smallest department number first). (106 rows)
3. Same query as above but reverse the order (largest department number first). (106 rows)
4. Same query as above but now order the result in ascending order of the department name. (106 rows)
5. Reverse the order. (106 rows)
6. Same query as above but order the result first by department id (ascending) and then by salary (ascending) within departments. (106 rows)
7. Add the manager id to the previous result after the salary. Order the result first by department id (ascending), then by salary (descending) within departments and then by manager id within salary (ascending). (106 rows)
8. List the employee id, last name, department id and department name for all employees in departments 90 and 80. (37 rows)
9. If you haven’t done so run the above query using **JOIN ... ON....** What happens when the join condition is omitted?
10. List the department name, manager id and city. (27 rows)
11. Show the employee ids, last names, department ids and department names of all employees and all departments. Include employees that don’t have a department assigned to them yet as well as departments with no employees. Are there any employees without departments? Are there any departments without employees? (123 rows)
12. Show only the last names and department ids of all employees including those that don’t have a department assigned to them. (107 rows)
13. Show the last names and department ids (just to make sure) of **only** those employees that don’t have a department assigned to them. (1 row)
14. Show the department ids and names of only those departments without any employees. (16 rows)
15. For each department list the department id, department name and full name (first name and surname) of the manager. Note that some departments don’t have a manager. (27 rows)
16. The same as above but only show departments that have a manager. (11 rows)
17. List the last name, job id, department name, location id and city name for all employees in departments 90 and 80. (37 rows)
18. List the employee ids and names of all employees together with their manager id and the names of their managers. Rename the columns in such a way that makes the result easy to understand. (106 rows)
19. As above, but make sure that really every employee is shown. What’s the difference? (107 rows)
20. Show the names of all cities in which there aren’t any departments together with their location ids. (16 rows)