# The City Lit Institute

##### Department of Computing

## Keeley Street, Holborn, London WC2B 4BA

**MySQL**

**(with Apache server)**

**LECTURER** : **ALEXANDER ADU-SARKODIE**

MSc. Telematics (IT & Telecom), MSc. Eng., Dip. Russ. Lang., Teach. Cert, AMIAEng (UK), MBCS (UK), MIfL(UK)

**Profile :** [**http://goldhawk-college.com/istudio/profile.html**](http://goldhawk-college.com/istudio/profile.html)

**Email :** [**aadusarkodie1@gmail.com**](mailto:aadusarkodie1@gmail.com)

**Modules**

* **MySQL**
* Modelling Data
* What are Tables?
* Optimizers
* Primary Keys
* Foreign Keys
* Cardinality
* Why Relational?
* Types of Relations
* Normalization
* Data Types
* Create Tables in phpMyAdmin
* Adding Rows and Columns
* Using SQL Query Window in phpMyAdmin
* Command Lines
* Adding Data
* Updating Data
* Searching Data
* Removing Data
* Views
* Charts and Graphs
* Exporting and Importing data
* Setting up backup data
* **Trouble shooting**
* **Wildcards and Functions**
* **Exercises**
* **Lookup Queries**
* **Optimization**
* **Project work**
* **Introduction to MongoDB**
* **Working with PHP**
* **Summary**
* **Managing Authentication**

**MySQL**

MySQL is the world most used **relational database management system** (**RDBMS**). It runs as a **server** providing **multi-user access** to a number of databases.

As a relational database,MySQL is often used to **store** data for web sites working in conjunction with PHP.

The SQL phrase stands for **“Structured Query Language”**. Applications which use MySQL databases include: TYPO3, Joomla, WordPress, Drupal and other software built on the **XAMPP** or **LAMP** software stacks.

**Binaries** are provided for many operating systems and **source code** is provided that can be compiled on all other systems.

**Relational** means that **different tables** of the database can be **cross referenced** to one another.

MySQL was built using the **SQL base** and released as an open source database system.

Here are some of the advantages of using MySQL:

* Its **fast**: The main goal of developing this RDBMS was **speed**. Consequently the software was designed with speed in mind using ***optimisers***.
* It is **non-procedural.** No need to define access paths.
* It’s **inexpensive**: MySQL is free under the **open source GPL license**, and a fee for a commercial license is reasonable.
* It’s **easy to use**: You can build and interact with a MySQL database using a **few** **simple statements** in the SQL language.
* **It can run on many operating systems**: MySQL runs on a wide variety of operating systems – Windows, Linux, Mac OS, most varieties of UNIX (including Solaris, ,ADX and DEC UNIX), FreeBSD, OS/2, Irix and others.
* **Technical Support**: A large **base of users** provides **free support** via **mailing list**.

There are two ways of building resources for MySQL. One is by using the **SQL code** directly either through a **shell prompt** or through some sort of **a query window**.

MySQL is the database component of the popular **XAMPP** and **LAMP** application stacks, alongside Apache, and PHP/Perl/Python programming languages.

For our course lessons, we will be using **XAMPP** bundle in conjunction with **phpMyAdmin** client to build our **databases,** and **algorithms for web applications.**

**You will learn to:**

* **Create A Database:** You will learn to create a database. Once that is done, you can start **adding** information.
* **Create Tables:** Within the database, you can have many tables, and each table is like a big **grid** with **information** held in each of the **boxes** on the **grid**. You will need to create at **least** one table to hold data within your database.
* **Add Data:** Data within a table can be managed in many ways. You will learn to **add**, **edit**, **delete** and **search** the information in your database.
* **Get Relational:** The great thing about MySQL is that it is a relational database. This means the data **from one of your tables** can be used in conjunction with data on **another table**. We call this a ***Join***.

**Modelling your Data**

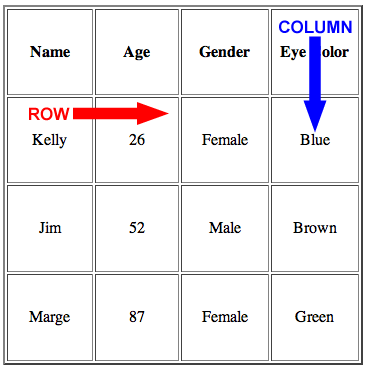
To design an efficient and structured database management system, it is required to have the following measures and procedures in place.

* **Name your database**
* **Identify the Objects –** An object is an entity and a representative of the table in the database, and binds attributes of the table to itself. Here, we begin **abstraction** for the table. High level aggregation for the table.

The object list could be

* + an id
  + forename
  + surname
  + city
  + postcode
  + A picture
  + email
  + address
  + cost
  + category
* **Define and name the table for each object**. For example postal subscribers
* **Identify the attributes for each object in the table** For example,
  + an id
  + forename
  + surname
  + city
  + postcode
  + picture
  + email
  + address
  + cost
  + category
* **Define and name columns**. For example,
  + user\_id
  + forename
  + surname
  + post\_code
  + picture
  + email
  + price
  + category
* **Identify the Primary Key**. Say, user\_id for the Postal Subscribers table and pet\_ID for the pets table. These keys will make each record unique as they persist.
* **Define the defaults**
* **Identify columns with required data**. These columns should never be allowed to be empty

**What are MySQL tables?**

[](http://0.tqn.com/d/php/1/0/A/-/-/-/table.gif)

A MySQL database can be made up of **many tables**. A table in a database consists of **intersecting columns and rows that form a grid**.

A good way to think about this is to imagine a **checker board**. Along the top row of the checkerboard there are **labels** for the data you wish to store, for example **Name**, **Age**, **Gender**, **Eye Colour**, etc. In all the rows below**, information** is **stored**. Each row is **one entry.**  All the data in a **single row, belongs to the same person** in this case and each column contains a **specific type of data** as indicated by its label.

When using MySQL information is stored in **Relational Database**. This is a **collection of tables**. A table is a structured **data consisting of vertical columns and horizontal rows**. See example 1 below:

MySQL tables are **two dimensional,** consisting of a **specified number of columns,** and a **variable number of unordered rows**. A row is an **entity**. Each row is characterised by a number of **attributes** called **columns**. Each column is defined by its **data type** to contain **certain types of values**. Below is an example of a simple table: **department**  table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Deptno** | **Deptname** | **loc** | **mgr** |
| 10 | Accounting | London | 200 |
| 20 | Research | Paris | 105 |
| 30 | Sales | Berlin | 409 |

Relational database is easy to learn because there is only one simple data structure: the **table**.

**Optimisers**

RDBMS is **non - procedural** because you **specify what needs to be done or achieved and not how to do it**. You don’t **need to define paths**. The system **automatically** finds it. The system has **optimisers** to find the data. The optimisers will **automatically search** for the **most efficient path to retrieve the desired results without any intervention from the user**.

**Primary Keys**

A **primary key** is a property of an element or record of a table in a database that makes it **unique**.

Further down are two databases the **Department** and **Employee** tables. The Employee table has a **deptno** column which is **common** to both tables. Each table in addition has a **unique** number that **identifies each record**. These are the **deptno** for Department and **empno** for Employee. Both fields are used as **unique identifiers** and are referred to as **primary keys.**

**Foreign Keys**

In the context of relational databases, a **foreign key** is a field (or collection of fields) in one table that uniquely identifies a row of another table.

For example, consider a database with two tables, a CUSTOMER table that includes all customer data and an ORDER table that includes all customer orders. Suppose that the business requires that each order must refer to a single customer. To reflect this in the database, the primary key (e.g., CUSTOMERID) in the CUSTOMER table is added to the ORDER table, where it is called a foreign key. Since CUSTOMERID in the ORDER table uniquely identifies a row of the CUSTOMER table, it says which customer placed the order.

**Cardinality**

It is the number of unique values (normally for the index on a table). For a table with a single primary key column, the cardinality should normally be equal to the number of rows in the table.

**department**

|  |  |  |  |
| --- | --- | --- | --- |
| **Deptno** | **deptname** | **Loc** | **Mgr** |
| 10 | Accounting | London | 200 |
| 20 | Research | Paris | 105 |
| 30 | Sales | Berlin | 109 |
| 40 | Manufacturing | Madrid | 210 |
| 50 | Shipping | Rome | 215 |

**employee**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Empno** | **name** | **job** | **Salary** | **comm** | **deptno** |
| 100 | Wilson | Clerk | 17000 |  | 10 |
| 101 | Smith | Salesman | 25000 |  | 40 |
| 103 | Raymon | Analyst | 35000 | 13000 | 30 |
| 105 | Michael | Manager | 45000 |  | 30 |
| 109 | Allen | Manager | 38000 | 8000 | 40 |
| 110 | Turner | Clerk | 18000 |  | 50 |

**Why Relational?**

So what is a **'relational'** database, and how does it use these tables?

A relational database lets us **'relate'** data from **one table** to **another**. Let's say for example we were making a database for a car dealership. We could make **one table** to hold all of **the details for each of the cars we were selling**. However, the **contact information** for 'Ford' would be the **same** for all of the cars they make, so we do **not** **need** to **type that data more than once**.

What we can do is **create a second table**, called ***manufacturers***. In this table we could list **Ford**, **Volkswagen**, **Chrysler -**  the parent car names, etc. Here you could list the **address, phone number and other contact information** for each of these companies. You could then **dynamically** call the **contact information** from our **second table** for every car in our **first table**. You would only ever have to **type** this information **once** despite it being accessible for **every car in the database**. This not only **saves time** but **also valuable database space as no piece of data need be repeated**.

**Types of Relationships**

## one-to-one (unusual)one-to-one

* in a one-to-one relationship, each record in Table A can have only **one** **matching** record in Table B, and each record in Table B can have only one matching record in Table A

## one-to-many (most common)one-to-many

* in a one-to-many relationship, a record in Table A can have **many matching** records in Table B, but a record in Table B has **only one matching** record in Table A
* note that your table properties provide you advice about how to ensure the relationships are based on the proper field properties.

## many-to-many (less common)many-to-many

* in a many-to-many relationship, a record in Table A can have **many matching records in Table B, and a record in Table B can have many matching records in Table**

**Normalization**

Normalization is the process of **organizing data in a database**. This includes creating tables and establishing relationships between those tables according to rules designed both to **protect the existing data** (accidental deletions or amendments) and to **make the database more flexible** by eliminating redundancy and inconsistent dependency.

Normalization usually involves **dividing large tables into smaller (and less redundant) tables and defining relationships between them**. The objective is to isolate data so that additions, deletions, and modifications of a field can be made in just one table and then propagated through the rest of the database using the defined relationships.

Redundant data **wastes disk space** and **creates database maintenance problems**. If data that exists in more than one place must be changed, the data must be changed in exactly the same way in all locations. A customer address change is much easier to implement if that data is stored only in the Customers table and nowhere else in the database.

There are a few rules for database normalization. Each rule is called a "**normal form**". If the first rule is observed, the database is said to be "**in first normal form**". If the first three rules are observed, the database is considered to be "**in third normal form**". Although other levels of normalization are possible, third normal form is considered the highest level necessary for most applications.

As with many formal rules and specifications, real world scenarios do not always allow for perfect compliance. In general, normalization requires **additional tables** and some designers find this first difficult and then cumbersome. If you decide to violate one of the first three rules of normalization, make sure that your application anticipates any problems that could occur, such as redundant data and inconsistent dependencies.

### **First Normal Form**

* Eliminate repeating groups in individual tables.
* Create a separate table for each set of related data.
* Identify each set of related data with a primary key.

**Do not use multiple fields in a single table to store similar data. For example, to track an inventory item that may come from two possible sources, an inventory record may contain fields for Vendor Code 1 and Vendor Code 2. Also, what happens when you add a third vendor? Adding a field is not the answer; it requires program and table modifications and does not smoothly accommodate a dynamic number of vendors. Instead, place all vendor information in a separate table called Vendors, then link inventory to vendors with an item number key, or vendors to inventory with a vendor code key.**

### **Second Normal Form**

* Create separate tables for sets of values that apply to multiple records.
* Relate these tables with a foreign key.

Records should not depend on anything other than a table's primary key (a compound key, if necessary). For example, consider a customer's address in an accounting system. The address is needed by the Customers table, but also by the Orders, Shipping, Invoices, Accounts Receivable and Collections tables. Instead of storing the customer's address as a separate entry in each of these tables, store it in one place, either in the Customers table or in a separate Addresses table.

### **Third Normal Form**

* Eliminate fields that do not depend on the key.

**Values in a record that are not part of that record's key do not belong in the table. In general, any time the contents of a group of fields may apply to more than a single record in the table, consider placing those fields in a separate table.**

For example, in an Employee Recruitment table, a candidate's university name and address may be included. But you need a complete list of universities for group mailings. If university information is stored in the Candidates table, there is no way to list universities with no current candidates. Create a separate Universities table and link it to the Candidates table with a university code key.

### **Other Normalization Forms**

Fourth normal form, also called Boyce Codd Normal Form (BCNF), and fifth normal form do exist, but are rarely considered in practical design. Disregarding these rules may result in less than perfect database design, but should not affect functionality.

**MySQL Data Types**

Each **column** can only contain **one type of data** which we must define. An example of what this means is; in our **age** column we use a **number**. We could not change Kelly's entry to "**twenty-six**" if we had **defined that column to be a number**. The main data types are **numbers, date/time, text, and binary**. Although these have many subcategories, we will just touch on the most **common** types that you will use in this tutorial.

**INTEGER**- This stores ***whole numbers***: both **positive**, and **negative**. Some examples are 2, 45, -16 and 23989. In our example, the age category could have been integer.

**FLOAT** - This stores numbers when you need to use ***decimals***. Say heights. Some examples would be 2.5, -.664, 43.8882, or 10.00001.

**DATETIME**- This stores a ***date and time*** in the format ***YYYY-MM-DD HH:MM:SS***

**VARCHAR** - This stores a **limited** amount of text or **single characters**. In our example, the name column could have been varchar (short for variable character)

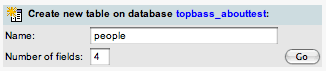
**BLOB**- This stores ***binary*** data other than text, for example file **uploads**.

**Create Tables in phpMyAdmin**

One of the easiest way to create a table is through **phpMyAdmin**, which is available on most hosts that offer MySQL databases. First you need to **login** to phpMyAdmin.

On the left hand side you will see "phpMyAdmin" logo, some small icons, and below them you will see your **database name**. You may have to **create** your database, if it doesn’t already exist. **Click** on your database name. Now on the right hand side any tables you may have in your database will be **displayed**, as well as a box labeled "**Create new table on database**"

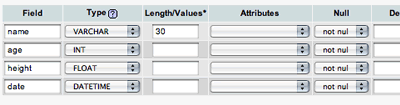
Click this and create a new table as we have in the diagram below.

[](http://0.tqn.com/d/php/1/0/5/-/-/-/MySQL1.gif)

**Adding Rows and Columns**

Let's say we work in a doctor's office and wanted to make a database called **persons** with simple table of people with a person's **name**, **age**, **height**, and the **date of birth** we collected this information. On the previous page we entered "**people**" as the name of our table, and chose to have **4** fields. This brings up a new **phpMyAdmin** page where we can fill in the **fields** and their **types** to add **rows** and **columns**.

We have filled in the field names as: **name**, **age**, **height**, and **date**. We have set the data types as **VARCHAR, INT (INTEGER), FLOAT and DATETIME**. We set a **length** of **30** on the **name**, and have left all other fields blank.

[](http://0.tqn.com/d/php/1/0/6/-/-/-/MySQL2.gif)

**Using SQL Query window in phpMyAdmin**

Perhaps a **quicker** way to add a table is by clicking on the small "**SQL**" **button** on the left hand side below the phpMyAdmin logo. This will bring up a **query window** where we can type our **commands**. You should run this command:

**CREATE TABLE people (name VARCHAR(30), age INTEGER, height FLOAT, date DATETIME)**

As you can see, the command "**CREATE TABLE**" does exactly that, creates a table which we have called "**people**". Then inside the **brackets**, we tell it what **columns** to make. The first is called "**name**" and is **VARCHAR**, the **30** indicates we are allowing up to **30 characters**. The second, "**age**" is an **INTEGER**, the **third** "**height**" is a **FLOAT** and the forth "**date**" is **DATETIME**. Note the **white space** that separates them.

Regardless of which method you chose, if you would like to see a breakdown of what you just did click on the "**people**" link that **now appears on the left hand side of your screen**. On the right you should now see the fields (data structure) you added, their data types, and other information

**Using Command Lines**

If you prefer you can also run **commands** from a **command line** as an option to create a table, you may have to install MySQL **locally**, then you will need to **login** to your MySQL database. Using your **password**, **username** and **database** name.Then you can run the command:

**CREATE TABLE people (name VARCHAR(30), age INTEGER, height FLOAT, date DATETIME);**

**Using Insert into SQL – Add Data**

Once you have created the table, you now need to **add data** **into it**. If you are using phpMyAdmin, you can **manually** enter in this information. First click on "people", the name of your table listed on the left hand side. Then on **the right hand side**, click the tab called "**insert**" and type in the data as shown below.

You can view your work by clicking people, and then the **browse** tab.

A quicker way is to add in data from the **query window**. Click the SQL **icon** in phpMyAdmin) or a command line by typing:

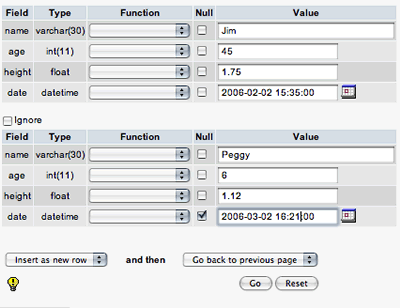
**INSERT INTO people VALUES ( "Jim", 45, 1.75, "2006-02-02 15:35:00" ), ( "Peggy", 6, 1.12, "2006-03-02 16:21:00" )**

This inserts the data directly into the table "people" in the order shown.

If you are **unsure** what order the fields in the database are, you can use this line instead:

**INSERT INTO people (name, date, height, age) VALUES ( "Jim", "2006-02-02 15:35:00", 1.27, 45 )**

Here we first tell the database the **order** we are sending the **column names**, and then the **actual values**.

[](http://0.tqn.com/d/php/1/0/7/-/-/-/MySQL3.gif)