INTRODUCTION TO RELATIONAL DATABASES WITH MYSQL

CODE CAMP CHARLESTON

TRIP OTTINGER

CONTENTS

SECTION 1: CONNECTING TO MYSQL WITH THE COMMAND PROMPT/TERMINAL

The first class will define how a database is organized using a relational model. We will discuss the advantages of a relational database management system (RDBMS). We will explain the basics behind using SQL to manipulate data. An overview of the various MySQL components will be provided including the MySQL server, SQL Workbench, and the mysql console application. We will learn about how to use a terminal application to connect to the MySQL database server and immediately begin creating databases, building tables, and working with data types.

EXERCISE: CONNECTING TO MYSQL

In this exercise we will learn how to talk with the MySQL Server via the **`mysql`** program within a terminal window. A terminal, also referred to as a terminal emulator or a terminal app in OS X, is a purely text-based system and provides an environment for Unix shells, which allows the user to interact with the operating system of any Unix-like computer in a text-based manner through the command line interface to the operating system. MySQL works with Windows, as well. This means you can install the MySQL database server on a Windows machine, as well as, use the various client applications like **`mysql`** program within a command prompt (terminal window).

1. In OS X, open the Terminal. On OS X, open your Applications folder, then open the Utilities folder. Open the Terminal application. You may want to add this to your dock. I like to launch terminal by using Spotlight search in OS X, searching for "terminal". If you are using Windows, open a command prompt.

Your instructor has already installed the MySQL server and the associated client applications. He has added the necessary users and permissioned those users to access the database server for tasks such as creating databases and performing SQL queries to retrieve and manipulate data.

In the exercise below, we will need to connect to the MySQL Database Server, we will call upon the **`mysql`** program from a command prompt (Unix/MacOSX termal window).

```
$ mysql –h host_name –p –u user_name
```

`–h` this is the host where the MySQL Server application is running. In this course, the MySQL Database Server is your host (local) machine. Since the host is the same computer from where you are running the **`mysql`** program, you can omit the -h option.

`–u` the MySQL user name. If using Unix this is the same as the Unix login name. If you want to use the Unix login name as the user name, you can omit the –u option.

`–p` you *could* provide the password directly as part of the command (be careful! no spaces after **`–p)`** BUT for security purpose *don't do this*. By not providing the value for –p, **`mysql`** will prompt you for the password without echoing the password to the screen.

2. Let's connect! Ask your instructor for the following options for the mysql command
   - host_name – we are running mysql from the same computer as the MySQL Server, so we can omit this option.
   - user_name – ask your instructor
   - password – we don't want to display the value on the screen as we type. Just provide the –p option and mysql will prompt you for the password while protecting the information as you type.

     ```
     $ mysql –u user_name –p
     ```

   - Go ahead and type in the mysql command at the terminal window using the appropriate options and associated values for the command. Since each student is running the MySQL server locally, you can omit the –h option.

- After you connect you should see the mysql prompt

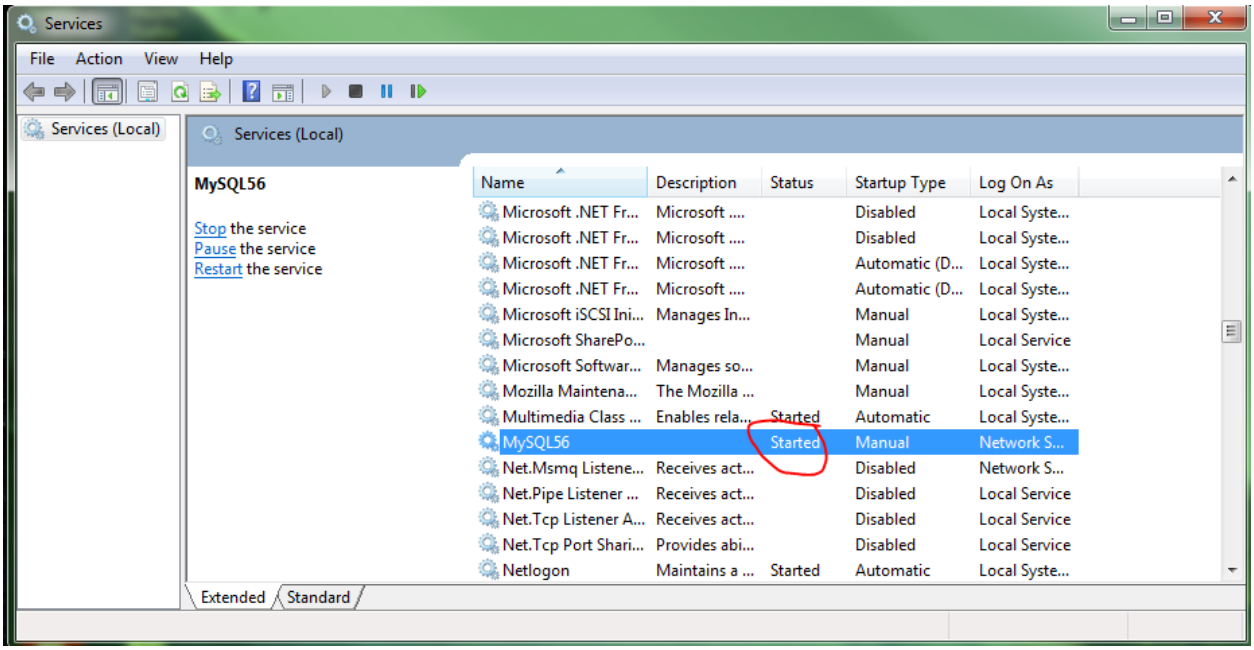
```
mysql>
```

- Let's quit the mysql application (not the server) by entering quit at the mysql> command prompt

```
mysql> quit
```

<END OF EXERCISE>

TIP: ON A WINDOWS BASED SYSTEM, MAKE SURE THE MYSQL SERVICE IS STARTED USING SERVICES. IN THE FIGURE BELOW, THE SERVICE IS NAMED MYSQL56L. USING SERVICES YOU CAN START, PAUSE, STOP AND RESTART THE MYSQL SERVER SERVICE.



NEXT STEPS: ON YOUR OWN TIME, CHECK OUT THE MACH OS X BASIC COMMAND LINE TUTORIAL ON YOUTUBE FOR MORE BASIC INFORMATION ON UNIX COMMANDS: HTTP://WWW.YOUTUBE.COM/WATCH?V=FTJOIN_OADC

FOR INSTRUCTIONS ON STARTING, STOPING, AND RESTARTING MYSQL FROM THE COMMAND LINE, CHECK OUT THE FOLLOWING ARTICLE: HTTP://COOLESTGUIDESONTHEPLANET.COM/START-STOP-MYSQL-FROM-THE-COMMAND-LINE-TERMINAL-OSX-LINUX/

EXERCISE: SIMPLE SQL COMMANDS

1. Open a Terminal - Applications\ Utilities\ Terminal

   Soon we will need to connect to the MySQL Server, we will call upon the **`mysql`** program from a Windowscommand prompt/Unix/MacOSX termal window.

   ```
   $ mysql –p –u user_name
   ```

2. Once connected to MySQL Server, type in the following command at the mysql> prompt. Notice the semicolon at the end! This tells mysql that the command is completed.

Heads Up! SEMICOLON! – You need to end your SQL statements with one!

```
mysql> SELECT NOW();
```

As an example, here is a screenshot of connecting to a local MySQL Server and running the SQL statement from a Windows prompt using the 'Training1' user:

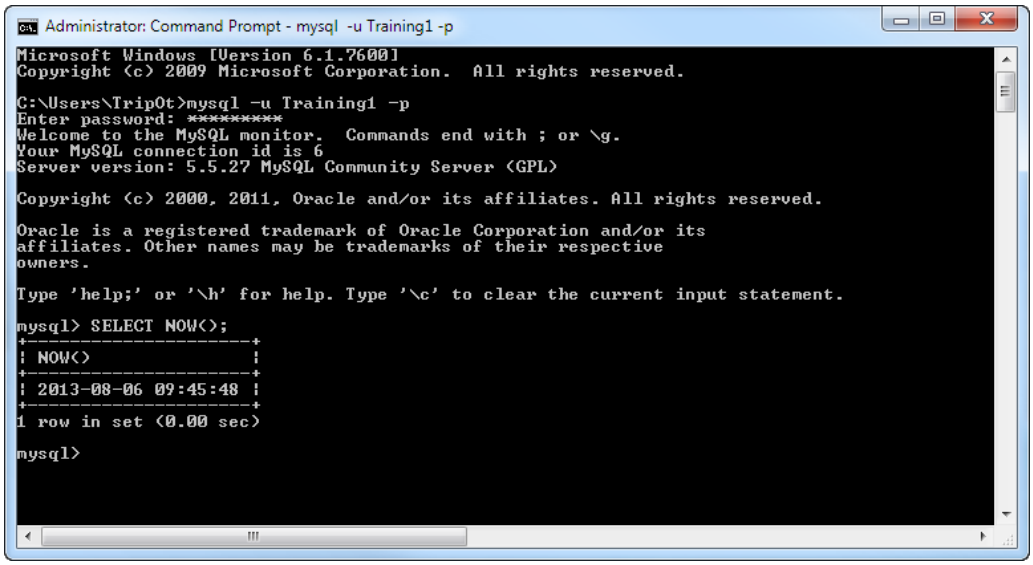


Figure: Starting the mysql application, entering a password, and executing a basic SQL statement

<END OF EXERCISE>

EXERCISE: CREATING AND DELETING DATABASES

Next we will create some databases on the MySQL Server. To create a database, we can issue a command that follows the correct syntax.

Heads Up!: In UNIX, the ***database_name*** *is case sensitive. On Windows systems, the* ***database_name*** *is NOT case sensitive.*

```
CREATE DATABASE | SCHEMA database_name
```

Example:

```
mysql> CREATE DATABASE Wassup;
```

Or this would also create the database:
Example:

```
mysql> CREATE SCHEMA Wassup;
```

1. Go ahead and create a database named 'Wassup'.
2. Create another database named 'WASSUP'.
3. Create a third database named 'wassup' using SCHEMA in the command instead of DATABASE.
4. Enter and run the following into the mysql> prompt to display a listing of databases on the server:
   ```
   mysql> SHOW DATABASES;
   ```

The **USE db_name** statement tells MySQL to use the **db_name** database as the default (current) database for subsequent statements. The database remains the default until the end of the session or another USE statement is issued:

5. Enter and run the following statement at the mysql> prompt to select the 'Wassup' database as the current database. Remember the semicolon at the end.

   mysql>USE Wassup;

6. Switch the current database to 'WASSUP'.
7. Use the **DROP DATABASE | SCHEMA *database_name*** to remove the 'WASSUP' database from the MySQL Server.

   mysql> DROP DATABASE WASSUP;

8. Drop the database named 'Wassup'. Don't forget the semi-colon at the end of the command.

9. Use **DROP SCHEMA** to delete the 'wassup' database. This time instead of a semi-colon you can terminate the command with a '/g' for example:

   mysql> DROP DATABASE wassup /g

10. With your databases dropped. End the **mysql** session by typing '**quit**' at the **mysql>** prompt. This will return you to the UNIX (**%)** prompt or Mac OSX prompt ($). The next time we want to connect to the MySQL server using the mysql program, we will have to issue another **mysql** command such as:

    $ mysql –p –u *user_name*

    <END OF EXERCISE>

QUICK REVIEW:

Some things to remember about creating database:

- Under Unix, database names are case sensitive. So, 'Wassup' is not the same as 'wassup' or 'WASSUP'.
- On Windows, database names are not case sensitive. So, you cant create a database named 'Wassup' If you already have a database named 'WASSUP'.
- If you get an error such as ERROR 1044 (42000): Access denied for user 'someuser'@'localhost' to database 'Wassup' when attempting to create a database, this means that your user account does not have the necessary privileges to do so
- CREATE SCHEMA is the same thing as CREATE DATABASE. There are two ways to end a SQL command. Either use a semi-colon (;) or /g

SECTION 2: CREATING TABLES AND DATA TYPES

Inarguably, over the last 20 years the most prevalent model for describing data has been the relational data model whereby the model is expressed as tables. Each table contains rows. It's very similar to a page in a spreadsheet with each spreadsheet page being similar to a single table. Each row within the table represents a thing of interest or *entity*. For example, a Customer table would hold rows of data with each row describing a single customer. The entity is described by the columns in the table with each column containing a single value. A Customer table may contain columns such as CustomerID, LastName, FirstName, Gender, and BirthDate.

Table Name:	**Customer**
Column Name	**Description**
CustomerID	A number that uniquely identifies a single customer. No other customer can use this number in this table.
FirstName	The customer's first name.
LastName	The customer's last name.
BirthDate	The customer's data of birth
Gender	A customer's gender: Male, Female, Unknown

You can have many tables within a relational database with the ability to relate tables to one another. For example, an Orders table could be related to the Customer table. We could express this relationship verbally by stating "One customer can place one to many Orders". Another way to say this would be "A single order is related to a single customer".

In order to relate the two tables together. we refer a column in a table to a column within the other table. Below we see the Orders table. See how we copy over the CustomerID from the Customer table to the Orders table? This begins to establish the relationship between the two tables.

Table Name:	**Orders**
Column Name	**Description**
OrderID	A number that uniquely identifies a single order
CustomerID	This column relates a customer to an order. A number that uniquely identifies a single customer. A single customer can have one to many orders. So, you may find this value repeats for each of the customers order.
OrderDate	The order date.

Below is some sample data from Customer and Orders. Again, note the CustomerID column's data within the Orders table. See how the values in this column match in the Customer table?

Customer Table				
CustomerID	FirstName	LastName	BirthDate	Gender
1	Jeff	Cave	1969-02-05	Male
2	Wes	Bridwell	1975-04-04	Male
3	Wendy	Jefferson	1980-10-13	Female
4	Pat	Jenkins	1972-09-01	Unknown
5	Leslie	Nelson	1948-05-23	Unknown
Orders Table				
OrderID	CustomerID	OrderDate		
10	1	2013-01-05		
20	3	2013-01-05		
30	5	2013-01-05		
40	3	2013-01-06		
50	3	2013-01-10		

A RELATION IS A SET OF TUPLES

The relational data model organizes data into a structure of tables and rows, or more properly, relations and tuples. In the relational model, a **tuple** is a set of name-value pairs and a **relation** is a set of tuples.

Relation = Table

Tuple = A rows in a Table

The Structured Query Language (SQL) deals with relations (tables), which are sets of rows (tuples). You can manage the rows of data using SQL. You INSERT rows using SQL. You retrieve (SELECT) rows with SQL. You edit (UPDATE) rows with SQL. You DELETE rows with SQL. Get the picture? The values returned by a SELECT SQL statement are a set of columns and rows. In the relational model you can think of these operations as operating on and returning tuples (rows of data). You can create a query that consists of a SELECT SQL statement to count up all the customers by state. You can give the query a name such as 'vCustomerCountByState'. We call this type of database object a *View*.

A View is virtual table in that it contains rows of data or a relation.

In a relational database system, we will have to define the structure of our tables first. This is known as defining a *schema*. It is assumed the data structure is known ahead of time. This is common across all different types of RDMBS like SQL Server, MySQL, Oracle, etc.

Nerd Alert: Recently, there's new a type database platform on the block. NoSQL databases operate without a schema, allowing you to freely add fields to database records without having to define a structure ahead of time. This is helpful when you can't assume the structure of your data ahead of time. Many newer applications which are designed today use a mixture of both relational and NoSQL technologies.

PRIMARY KEY

Did you notice something interesting about the schemas for the **Customer** and **Orders** tables? Each table has an ID column such as **CustomerID** for the **Customer** table and **OrderID** for the **Orders** table. These ID columns are used to uniquely identify each row in their respective table. We call these types of columns a *Primary Key*. A primary key is a column or combination of columns which uniquely specify a row (tuple). The values that make up the primary key must be unique, they cannot repeat in the table.

NORMALIZATION

According to Wikipedia, "Database normalization is the process of organizing the fields and tables of a relational database to minimize redundancy and 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 via the defined relationships…"

It's important when you model a table that you only place values that directly relate to the entity. For example, you could track the individual's favorite car in the **Individual** table but you would not want to track additional values about that car such as the car color, engine type, and body style. Rather, these fields would belong within a separate **Automobile** table.

INTRODUCING DATA TYPES

Databases manage data. How's that for profound? A database contains tables and a table contains column. Each column of data is associated with a *data type*. When you define the schema for a table you have to also define the columns in the table. When you define a column you will provide a column name, a data type, and possibly some additional values depending on the data type. Let's review a simple data type known as a ***String*** data type.

STRING DATA TYPE

For example, a person's last name would be a string. A person's last name would be a string. A person's city of birth would be a string. Their dog's name would be a string. There are different data types of strings with the most common being `CHAR` and `VARCHAR`. `CHAR` is short for character. `VARCHAR` is short of variable length character. With `CHAR` the length of the data type is fixed based upon a length of N. Values with a length shorter than N are padded to the length of N with trailing spaces and stored in the table. The padded spaces are removed when the data is retrieved. So `CHAR(30)` translates to a data type that holds up to 30 characters.

Unlike `CHAR`, `VARCHAR` values are not padded when they are stored. The basic rule of thumb is if your data length does not change in your column, it is more efficient to use `CHAR`. If there is variation to the length of your characters in a column, consider using `VARCHAR`. For example, a Zip code column would be `CHAR(5)` or `CHAR(9)` if you wanted to store those weird extra 4 digits at the end.

EXERCISE: CREATING TABLES

So you want to be a rock and roll star? Well listen now and learn how to build relational database tables. We will create some tables to model some data around popular music performers. Let's think about a rock show, what types of things can we track? How about the individual performers and their bands, let's model them first. We know that a solo act or a band could perform. A solo act is comprised on a single individual and a band which is comprised of many individuals. Let's start with the individuals. We need to create a table named **Individual** which tracks a series of values for each individual entity such as first name, last name, birth country, birth date, hometown, and a brief biography.

To create a table we use a CREATE TABLE statement. The syntax is:

```
CREATE TABLE table_name (column_listing)
```

1. Open a Terminal if it's not already open and connect to MySQL.
2. Create a database named **RockStar**. Remember that semi-colon!
3. Select the **RockStar** database as the default database for use.
4. For starters we will create the Individual table with just the last and first name columns. We will use the VARCHAR string data type for each of these two columns. We will prefix the table name (**Individual**) with the database name (**RockStar**). This part is

redundant if we selected the default database. This will ensure the table is created in the correct database. At the `mysql>` prompt, enter the following SQL statement to create the **Individual** table.

Remember, we can enter multiple lines at the command prompt but be sure to end the statement with a semicolon to tell mysql that you are done typing the statement.

```
mysql> CREATE TABLE RockStar.Individual
    ➔       (LastName varchar(50) NOT NULL
    ➔       , FirstName varchar(25));
```

5. Enter the **`SHOW DATABASES;`** statement to list your databases within the MYSQL server.
6. Enter the **`SHOW TABLES;`** statement to list your tables within the **RockStar** database. You should see the **Individual** table listed.
7. Enter the statement **`SHOW COLUMNS FROM Individual;`** Compare the results on your screen to the **`CREATE TABLE`** statement you entered earlier.
8. Don't grow too attached to your Individual table just yet. **`DROP`** the table by issuing the following statement at the mysql> prompt:

```
mysql> DROP TABLE Individual;
```

<END OF EXERCISE>

NOT NULL

Sometimes it is ok to have an unknown or missing value within a column in a table. Sometimes it isn't. When a column is marked at **`NOT NULL`**, we will not be able enter a **`NULL`** value into the column. You test for **`NULL`** values within your tables by using the **`IS NULL`** and **`IS NOT NULL`** operators within a **`SELECT`** statement. **`IS NULL`** will test whether a value is **`NULL`** while **`IS NOT NULL`** will test whether a value is not **`NULL`**.

In the **`CREATE TABLE`** statement above, notice that the **LastName** columns was marked with the **`NOT NULL`** operator. This means we have to enter something into this column. In contrast, the **FirstName** column will allow **`NULL`**.

DATE AND TIME DATA TYPES

There are different kinds of date and time data types: **`DATE, TIME, DATETIME, TIMESTAMP`**, and **`YEAR`**. A date such as July 4, 1776 would be entered into a column with a **`DATE`** data type as 1776-07-04 or 1776-7-4. MySQL might seem weird in how it portrays dates but it is in accordance with the ISO standard. Below is a brief table showing the various Date and Time data types:

Data Type	Format	Description	Example
DATE	CCYY-MM-DD	Year-Month-Day	1999-12-31
TIME	hh:mm:ss	Hour-Min-Sec	23:59:59
DATETIME	0000-00-00 00:00:00	Holds combined date and time values	1999-12-31 23:59:59
TIMESTAMP	0000-00-00 00:00:00	Date and time like DATETIME but stored as Universal Coordinated Time (UTC)	
YEAR	YEAR([M])	Represents a year value. Optionally specify a display width of either 4 or 2.	YEAR(2) - 74 YEAR(4) - 1974

So for a birth date, you could use a data type of **`DATE`**. If you wanted to just store the year for something, which is much more space efficient, you could use a data type of **`YEAR(4)`**. This would return the year value back as a 4 digit year.

Pro Tip: When designing your tables, it is VERY important to use as small of a data type as possible for each of your columns. Efficiently storing data into tables has a huge impact on the overall speed of your database system, especially as the number of records within your tables grow.

Next Steps: After the course, be sure to explore date data types further. There is much more detail to learn for date data types, especially for TIMESTAMP and YEAR data types.

EXERCISE: CREATING TABLES WITH DATE COLUMNS

Let's expand our **Individual** table by adding some date related columns.

1. At the `mysql>` prompt, enter the following `CREATE TABLE` statement to recreate the **Individual** table. We will add a **BirthDate** column and a **DateAdded** column. We use a data type of `DATE` for the birth date and a data type of `TIMESTAMP` for the **DateAdded**. **DateAdded** will help us track when a record was added into the table for auditing purposes. We will use the `DEFAULT` property on the `TIMESTAMP` column to designate that for new rows the column's value is set to the current timestamp if you do not specify a value when the row is added.

```
mysql> CREATE TABLE RockStar.Individual
         ->    (LastName varchar(50) NOT NULL
         ->    , FirstName varchar(25)
         ->    , BirthDate DATE NOT NULL
         ->    , DateAdded TIMESTAMP DEFAULT CURRENT_TIMESTAMP);
```

2. Enter the statement `SHOW COLUMNS FROM Individual;` Compare the results on your screen to the `CREATE TABLE` statement you entered earlier.

```
mysql> CREATE TABLE RockStar.Individual (LastName varchar(50) NOT NULL,
FirstName varchar(25) , BirthDate DATE NOT NULL , DateAdded TIMESTAMP DE
FAULT CURRENT_TIMESTAMP);
Query OK, 0 rows affected (0.16 sec)

mysql> SHOW COLUMNS FROM Individual;
+-----------+-------------+------+-----+-------------------+-------+
| Field     | Type        | Null | Key | Default           | Extra |
+-----------+-------------+------+-----+-------------------+-------+
| LastName  | varchar(50) | NO   |     | NULL              |       |
| FirstName | varchar(25) | YES  |     | NULL              |       |
| BirthDate | date        | NO   |     | NULL              |       |
| DateAdded | timestamp   | NO   |     | CURRENT_TIMESTAMP |       |
+-----------+-------------+------+-----+-------------------+-------+
4 rows in set (0.01 sec)

mysql>
```

<END OF EXERCISE>

INSERT STATEMENTS ADD ROWS TO A TABLE

Relational database systems use the Structured Query Language (SQL) to deal with relations, which are rows of data. With SQL there are different statements to do different types of operations on the data. Many SQL statements deal with manipulating the data within tables. One such statement is the `INSERT` statement which is used to add a row of data into a table. When you define the `INSERT` statement you start with the word `INSERT INTO` followed by the name of the table followed by the word `VALUES` followed by a series of column names. Below are examples of `INSERT` statements.

The `INSERT` statement below uses the following syntax. With this syntax, the values must contain a value for each column. And the values should be in the order in which you created the table.

```
INSERT INTO table_name VALUES (value1, value2,…);
```

An example would be:

```
INSERT INTO Individual Values ('Hendrix','Jimi','1942-11-27');
```

Notice in the example above we omitted the value for the **DateAdded** column and that's ok because of the default placed on the column. Omitting a value for a column will cause a NULL value to be entered which then causes the default value to be placed into the column.

Note: Go ahead and commit this concept to memory: "Omitting a value for a column will cause a NULL value to be entered which then causes the default value to be placed into the column. "

Here's another way to write an `INSERT` statement. It lists the column names after the table name:

```
INSERT INTO table_name (Column1, Column2, Column3,…) Values (value1, value2, value3,…);
```

An example would be:

```
INSERT INTO Individual (LastName, FirstName, BirthDate) VALUES ('Jagger', 'Mick', '1943-07-26');
```

Danger Ahead! Note the syntax for the second example above. When a column is not used in the list of columns, a default is assigned such as NULL. If a column does not allow NULLs and you don't specify the column in the column list, and the column does not have a default declared when you created the table then you will get an error.

Here is a third syntax for `INSERT`ing rows into table. This syntax allows you to add more than row at once.

```
INSERT INTO table_name VALUES (,,,),(,,,),(,,,),…;
```

```
And an example would be:
```

```
INSERT INTO Individual VALUES ('Jagger', 'Mick', '1943-07-26'), ('Zimmerman', 'Robert', '1942-05-25'),('Cobain', 'Kurt', '1967-02-20');
```

EXERCISE: DESCRIBE LISTS A TABLE'S STRUCTURE

Before you write an `INSERT` statement, it's sometimes helpful to refresh your knowledge of the table's schema. One way to do this is to use the `DESCRIBE table_name` statement.

1. Go ahead and run the following 2 statements:

```
mysql> USE RockStar;
mysql> DESCRIBE Individual;
```

```
mysql> USE RockStar;
Database changed
mysql> DESCRIBE Individual;
+-----------+-------------+------+-----+-------------------+-------+
| Field     | Type        | Null | Key | Default           | Extra |
+-----------+-------------+------+-----+-------------------+-------+
| LastName  | varchar(50) | NO   |     | NULL              |       |
| FirstName | varchar(25) | YES  |     | NULL              |       |
| BirthDate | date        | NO   |     | NULL              |       |
| DateAdded | timestamp   | NO   |     | CURRENT_TIMESTAMP |       |
+-----------+-------------+------+-----+-------------------+-------+
4 rows in set (0.04 sec)

mysql>
```

2. Now let's use the INSERT statement to add a single rock star into the Individual table. Here is one way to insert a row into a table. The syntax can be quite elaborate. There are more than 3 ways to write an INSERT statement. After the course, I suggest you review the different nuances of the syntax to perform INSERTs via the online MySQL guide: http://dev.mysql.com/doc/refman/5.5/en/insert.html. Enter the following INSERT statement and press ENTER to run the query. Note how the table name is qualified with the database name.

   ```
   mysql> INSERT INTO RockStar.Individual (LastName, FirstName, BirthDate) VALUES
   ('Jagger', 'Mick', '1943-07-26');
   ```

After you submit the command to the MySQL Server, you should see a message that says 'Query OK, 1 row affected…'

3. Now, let's check out the contents of the Individual table by issues a `SELECT` query. Enter the following statement at the `mysql>` prompt and press ENTER to run the query:

   ```
   mysql> SELECT * FROM Individual;
   ```

   If all went well, your output in the terminal should look something like this. Notice the value of the **DateAdded** column. It's in the table but we did not provide the value within our `INSERT` statement. Interesting…

```
mysql> INSERT INTO RockStar.Individual (LastName, FirstName, BirthDate) VALUES ('Jagger', 'Mick', '1943-07-26');
Query OK, 1 row affected (0.05 sec)

mysql> Select * FROM Individual;
+----------+-----------+------------+---------------------+
| LastName | FirstName | BirthDate  | DateAdded           |
+----------+-----------+------------+---------------------+
| Jagger   | Mick      | 1943-07-26 | 2013-02-09 02:15:55 |
+----------+-----------+------------+---------------------+
1 row in set (0.01 sec)

mysql>
```

4. Add a couple more rock stars into the table. You can enter the following values or INSERT whomever you wish.

   LastName: Harrison

   FirstName: George

   BirthDate: February 25, 1943

   LastName: Buck

   FirstName: Peter

   BirthDate: December 6, 1956

<END OF EXERCISE>

MODIFYING A TABLE'S SCHEMA

Unfortunately, some rock stars are no longer with us. We need to model this fact by adding a **DeceasedDate** to the Individual table with a data type of `DATE`. Since some rock stars will still be among living, we will include the NULL option on the new column. To modify a table we will use an `ALTER TABLE` statement. The `ALTER TABLE` statement can do a lot of things. You can use it to create or drop new indexes. You can rename the table. You can add or remove columns. You can modify column data types. Again, we can't cover everything in this course, so I recommend you study the `ALTER TABLE` statement on your own time.

EXERCISE: ALTER TABLE

The syntax for ALTER TABLE is:

```
ALTER TABLE table_name action [, action] …;
```

So, with `ALTER TABLE` you should include at least one action but have the option for additional actions. Before using the `ALTER TABLE` statement, it's a good idea to verify its current structure by issuing `SHOW CREATE TABLE` statement.

1. At the `mysql>` prompt, enter the following command to verify the structure of the Individual table:
```
mysql> SHOW CREATE TABLE INDIVIDUAL;
```

2. Now let's add the **DeceasedDate** column by using the `ADD` action:
```
mysql> ALTER TABLE Individual ADD DeceasedDate DATE NULL;
```

3. After successfully adding the column, inspect the table again using `SHOW CREATE TABLE` statement.

<END OF EXERCISE>

EXERCISE: ADDING A COLUMN AS THE PRIMARY KEY AND USING THE INT DATA TYPE

Let's add another column this time to our table. We need a way to avoid confusion between individuals. In other words, we need for the rows within the table to be unique. Using a data type of `INT` is helpful because it will store simple integer values. With integers, there isn't a fractional part. We don't want negative numbers and we must provide a value when INSERTing rows into the table. We want the database to provide the value when adding rows into the table; we would like for the database to figure that out for us. We also need for the column to be indexed to help speed retrieval of the rows when we want to look up rock stars or join the table with other tables to retrieve rows from more than one table.

That's a lot for a single column to do, but we can do it by adding a new column with the following data type and options:

- We will call the column **ID** and the data type will be `INT`. This will hold an integer value.
- We will add the `UNSIGNED` column attribute to the new **ID** column. This will prevent negative values in the column.
- We will add the `NOT NULL` column attribute which will prevent missing values in the column
- We will add the very important `AUTO_INCREMENT` option. This will tell the database server to generate unique numbers to identify each row in the table. It will generate sequential numbers automatically. Pretty cool!
- The column must be indexed. We will take care of this next by adding a `Primary Key` clause which indicates the column is indexed to allow fast lookups. It also sets up a constraint on the table which dictates that each value must be unique. This prevents us from entering the same ID value twice in the RockStar table.

Note: There can only be one column in each table that uses the `AUTO_INCREMENT` column attribute. The column cannot have a NULL value. We took care of this when we assigned the NOT NULL column attribute.

1. Now let's add the **ID** column by using the `ADD` action once again. This time we will issue an `ALTER TABLE` command on the Individual table to add a column named **ID** which is unsigned meaning it won't accept negative numbers. We also specify that the column should not accept missing values and the value in the column will auto increment. We will finish the column off by marking it as the primary key which adds an index to speed performance and a unique constraint to keep out duplicates:

   ```
   mysql> ALTER TABLE Individual ADD ID INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY;
   ```

2. By adding the `AUTO_INCREMENT` option on the column, the system will auto fill the column starting with a value of 1 and increasing the value by 1 for each existing row in the table. Enter a `SELECT` statement to retrieve the data back out of the **Individual** table.

   ```
   mysql>SELECT * FROM Individual;
   ```

3. Add a couple more records (rows) into the **Individual** table. We will add a row for Neil Young and another row for Levon Helm. But this time we will provide a value of 100 for the ID for Neil and a value of NULL for the ID for Levon. Oh, forgot to mention that Levon recently passed away, so you will need to provide a value for the DeceasedDate. How do you think the system will react to the value of 100 being placed for the ID column when it has AUTO_INCREMENT defined? And what about the NULL value for the ID for Levon? Do you think we will receive an error from the database server?

   ```
   mysql> INSERT INTO RockStar.Individual (ID, LastName, FirstName, BirthDate) VALUES
   (100, 'Young', 'Neil', '1945-11-12');

   mysql> INSERT INTO RockStar.Individual (ID, LastName, FirstName, BirthDate,
   DeceasedDate) VALUES (NULL, 'Helm, 'Levon', '1940-05-26','2012-04-19');
   ```
   Did the database complain?

4. And finally, use a `SELECT` statement to view the contents of the table. What primary key values did you find for Neil and Levon?

   <END OF EXERCISE>

NUMBER DATA TYPE

The value 123.45 is a number, specifically it's a fixed point number, and in MySQL it's data type is called DECIMAL. The value 56 is a number; more specifically it's an integer. Like t-shirts, integer data types in MySQL come in all sorts of sizes from extra small to extra-large such as TINYINT, SMALLINT, MEDIUMINT, INT, and BIGINT.

Tip: We won't cover all the different data types in this course. Instead, I recommend you check out the MySQL reference manuals on the Oracle web site at http://dev.mysql.com/doc/ or get a good book on MySQL such as the MySQL Developer's Library by Paul DuBois.

YEAR DATA TYPE

YEAR is a 1-byte type used to represent year values. It can be declared as YEAR(4) or YEAR(2) to specify a display width of four or two characters. The default is four characters if no width is given.

You can specify the value as either a number or a string. So a string value like '1999' will work just as well as a numerical value like 1999.

ENUM DATA TYPE

An ENUM is a string object with a numeric value chosen from a defined, static list of possible values. The strings you specify as input values are translated by the database server as numbers. The cool thing is the numbers are converted back to the corresponding strings in query results. The elements listed in the column specification are assigned integer numbers, beginning with 1. Here is an example:

```
CREATE TABLE DrinkMenu (

    name VARCHAR(40),

    size ENUM('12 oz', '16 oz', '24 oz', '32 oz')

);
```

BOOL, BOOLEAN, TINYINT(1) DATA TYPES FOR TRUE/FALSE

There is not an exact way to represent True and False in MySQL. You can get close, however. The BOOL, BOOLEAN, TINYINT(1) data types are ways to implement a true/false value within a table column. Each column can store a 0 for false or a 1 for true. … well sort of. BOOL and BOOLEAN get translated to TINYINT(1) which will store a very small integer value with a signed range is -128 to 127 . The unsigned range is 0 to 255. So if you have a 0 in the column, your application can interpret this as False and anything else as True.

EXERCISE: USING THE YEAR, ENUM, AND BOOLEAN DATA TYPES

1. Create a new table called **Band** within the **RockStar** database. The table should have the following columns and options. Be sure to read the column descriptions as they hold clues on how to define the table schema:

Table Name:	**Band**			
Column Name	**Data Type**	**Primary Key**	**NULLS ALLOWED?**	**Description**
ID	INT	Yes	NO	The primary key which uniquely identifies the band. It should auto increment.
Name	Varchar(25)		NO	The band name like 'The Beatles' or 'Mudhoney'.
YearFormed	Year(4)		NO	The year the band was founded or formed.
IsTogether	Boolean		NO	Is the band still together? yes or no, 0 or 1.
MusicGenre	Enum		NO	Choose from the type of music the band played. Rock, Blues, Pop, Hip-Hop

2. Once you have successfully created the **Band** table, use INSERT statements to add some bands into the **Band** table. Ideally, your individuals will belong to the bands that you add into the Band table. We will model the relationship between bands and individuals next

<END EXERCISE>

SECTION 3: FOREIGN KEYS

Foreign keys allow you to relate data together between two tables. You create a foreign key by creating a foreign key constraint which helps prevent orphaning records. Foreign keys are created using either the CREATE TABLE or ALTER TABLE statement. Foreign keys have the following characteristics:

- The foreign key references a parent table and a child table.
- Creating a foreign key requires relating a column within a child table to a column within a parent table. The values within the child table column must match the value within the parent table column.
- The FOREIGN KEY clause must be placed on the child table.
- Both tables referenced by the foreign key must be stored within the INNODB database engine.
- The two referenced columns must be of the same data type.
- InnoDB requires indexes on the columns within both tables referenced by the foreign key.
- Creating a row in the child table will be rejected if there is not a matching value in the parent table.

In the example below, we are creating a table named **Automobile** that contains an **ID** column for the primary key and a **Name** column for the name of the car. Notice also how we explicitly designate the storage engine for the table as InnoDB.

```
CREATE TABLE Automobile (

ID INT NOT NULL

, NAME VARCHAR(25) NOT NULL

, PRIMARY KEY (ID)

) ENGINE=INNODB;
```

Next we create a child table named **Engine** which tracks the engine options available for the **Automobile**. We use an **ID** column for the primary key, and an **AutomobileID** as the column that supports the foreign key which references the parent **Automobile** table's **ID** column. Note the index created on the **AutomobileID** column in the child **Engine** table. Remember in both tables, the columns involved in the foreign key relationship in each table require coverage with an index. The parent **Automobile** table received an index on its **ID** column when we designated this column as the primary key.

```
CREATE TABLE Engine

(ID INT NOT NULL

, AutomobileID INT NOT NULL

, EngineSize DECIMAL (2,1) NOT NULL

,INDEX AutomobileID_idx (AutomobileID),

FOREIGN KEY (AutomobileID) REFERENCES Automobile(ID)

                    ON DELETE CASCADE

) ENGINE=INNODB;
```

Looking back at the `CREATE TABLE` statement for the **Engine** table we can see the following pieces involved with creating a foreign key.

- Created a column in the child table with the same data type as the column in the parent table. In the **Engine** table we created the **AutomobileID** column.

- Created an index in the child table on the column that supports the foreign key. In the statement above, we created an index named **AutomobileID_idx**.
- Created a `FOREIGN KEY` clause that first references the column in the child table (**AutomobileID**) followed by `REFERENCES` statement followed by the parent table name (**Automobile**) followed by the column in the parent table (**ID**).
- Optionally, you can specify what happens when a row either deleted or updated on the parent table by specifying `ON DELETE` and `ON UPDATE`. Use either `ON DELETE` or `ON UPDATE` with one of the following options:
    - CASCADE – deleting or updating a parent id row on the parent cascades the action down to the matching rows within the child table. Updating a parent id causes matching rows to be updated with the same value in the child table. Deleting a parent row causes matching rows to be deleted in the child table.
    - SET NULL - set the foreign key child column to null. If you use this option be sure that you have not designated the foreign key child column as NOT NULL.
    - RESTRICT – Rejects the update or deletion on the parent record. In other words, you can't orphan the children.

NOTE: THE INNODB STORAGE ENGINE SUPPORTS FOREIGN KEYS WHILE OTHER STORAGE ENGINES SUCH AS MYISAM, MEMORY, CSV, AND ARCHIVE DO NOT. WE WILL BE BUILDING TABLES USING THE INNODB STORAGE ENGINE WITHIN THIS TRAINING COURSE.

MODELING FOREIGN KEY RELATIONSHIPS

In another example, a band could release many albums. In the example below, an optional relationship is shown between band and albums; the symbols closest to the **Album** entity represents zero, one, or many whereas an **Album** has one and only one **Band**. The former is therefore read as, a band releases "zero, one, or many" album(s).

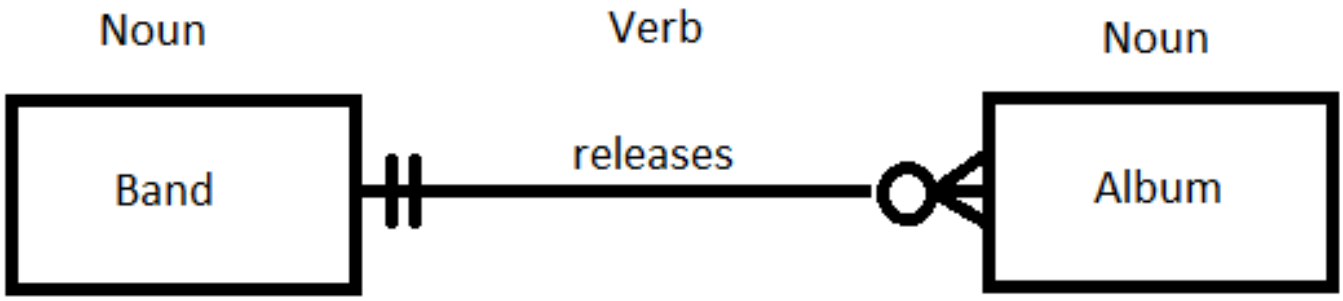


Figure: Another One-to-Many relationship

EXERCISE: CREATE AN ALBUM TABLE. RELATE THE ALBUM TABLE TO THE BAND TABLE USING A FOREIGN KEY

1. Let's create the Album table that tracks an ID for the primary key, Name, AlbumYear, and a BandID column to support the foreign key to the Band table. The table looks like this. The trick is the BandID column as it requires the same data type of the parent column that you are relating it to. If the parent column is an unsigned INT then child column will need to be an unsigned INT. You will need to add an index for the BandID column.

Table Name:	**Album**			
Column Name	**Data Type**	**Primary Key**	**NULLS ALLOWED?**	**Description**
ID	INT	Yes	NO	The primarykey for the Album. Auto increments
Name	VARCHAR(50)		NO	The album name.
AlbumYear	Year(4)		NO	The year the album was released.
BandID	INT		NO	The foreign key to the Band table's ID column.

```
CREATE TABLE Album (
ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY
, Name varchar(50) NOT NULL
, AlbumYear Year(4) NOT NULL
, BandID INT UNSIGNED NOT NULL
, INDEX BandID_IDX(BandID)
, FOREIGN KEY (BandID) REFERENCES BAND(ID) ON UPDATE CASCADE ON DELETE CASCADE
);
```

<END EXERCISE>

Our next goal is to model the relationship between rock stars (individuals) and the band that they belong to. Let's take a musician, say Paul McCartney, which band does he belong to? The Beatles, right? So we *could* model this by adding a new column on the **Individual** table that tracks the **Band** table's **ID** column. To accomplish this we could add a column named **BandID** to the **Individual** table. The **BandID** column would hold the **ID** value from the **Band** table. In this way we relate a row from the **Band** table to a row within the **Individual** table. The table would look something like this. Don't get too attached to this design because we are not done with it yet.

Table Name:	**Individual**		
Column Name	**Data Type**	**Primary Key**	**Description**
ID	INT	Yes	The primary key which uniquely identifies a rock star.
FirstName			The rock star's first name.
LastName			The rock star's last name.
BirthCountry			the birth country like "Scotland" or "USA"
BirthDate			the birth date
Hometown			The name of the home town. Example: "Portland"
Biography			A rather long note about the rock star.
BandID	INT		This column would hold a value from the Band table's ID column. Under this design, we are effectively saying that an Individual can belong to a single band. Hmmmm, we may have a problem.

And the sample data in both the Band and Individual tables would look like this:

Individual Table

ID	FirstName	LastName	BirthCountry	BirthDate	Hometown	Biography	BandID
1	Mick	Jagger	England	7/26/1943	Dartford	English musician, si	1234
2	George	Harrison	England	2/25/1943	Liverpool	English musician, si	1111
3	Neil	Young	Canada	11/12/1945	Toronto	He began performing	2222
4	Helm	Levon	United States	5/26/1940	Elaine, Arkansas	American rock musi	1222
5	Ringo	Starr	England	7/7/1940	Liverpool	English musician an	1111
6	Ronnie	Wood	England	6/1/1947	Hillingdon	English rock musicia	1234

Band Table

ID	Name	YearFormed	IsTogether	MusicGenre
1111	The Beatles	1969	No	Rock
1234	The Rolling Stones	1962	Yes	Rock
1222	The Band	1964	No	Rock
2222	CSNY	1968	No	Rock

Below we see a picture of this relationship between a **Band** and the **Individual**. We call this type of picture an Entity Relationship Diagram or ERD. An ERD is a way to describe the relationships between entities (Tables). There are different types of notation for

expressing an ERD. The diagram below is in "Crow Foot" notation. The symbols closest to the **Individual** entity represents zero, one, or many whereas an **Individual** has one and only one **Band**. The two vertical lines next to the **Band** entity represent the "one and only one" part of the relationship. This diagram states a band is comprised of zero, one, or many individuals. We call this a *One-to-Many* relationship. One band is comprised of many individuals. You could flip the entities around and state an individual belongs to a single band.

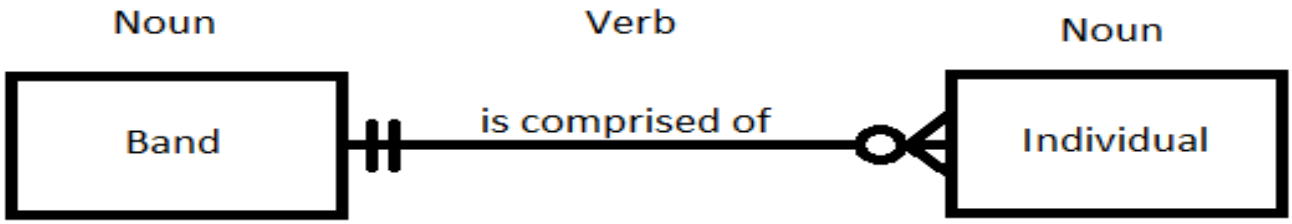


Figure: One-to Many ERD

MANY TO MANY RELATIONSHIPS

Up to this point we have talked about One-to-Many relationships. One individual performs many songs. One band is comprised of many individuals. Let's test the converse of the relationship between individuals and band in the real world. Is it true to say that one individual belongs to one and only one band? Back to Sir. Paul McCartney, Paul was a member of the Beatles and he was also a member of the band Wings. So, the individual can belong to many bands and a band is comprised of many individuals. We have ourselves a *Many-to-Many* relationship. We can *logically* express a Many-to-Many relationship with a model that looks like this:

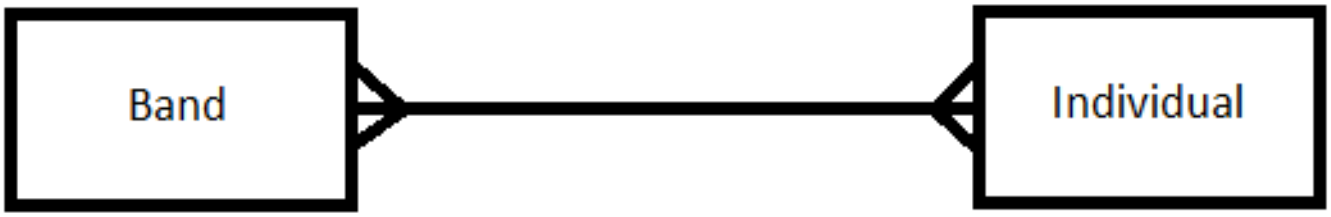


Figure: Many-to-Many ERD

But when it comes time to *physically* implement the logical model of a Many-to-Many relationship in the database, we will have a little more work to do. We have to add a table that sits in between the two entities to form the Many-to-Many relationship. This table is often called a junction table or a cross-reference table. In the ERD below the junction table is called **IndividualBand**.

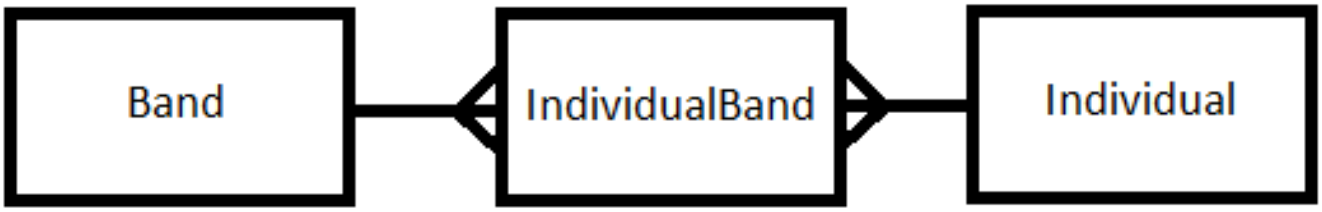


Figure: A junction or cross-reference table helps implement the Many-to-Many relationship between Band and Individual

So now we have to create the **IndividualBand** table. It will consist of a primary key named **ID**. We will copy over the primary key columns from the **Band** and **Individual** tables and add them to the **IndividualBand** table as the **BandID** and **IndividualID** columns. We will use the **BandID** and **IndividualID** columns as foreign key fields which provide data integrity. This will ensure we do not associate an **Individual** with a **Band** that does not exist or vice versa.

Table Name:	**Individual**			
Column Name	**Data Type**			
ID	INT			
FirstName				
LastName				
BirthCountry				
BirthDate				
Hometown				
Biography				
		Table Name:	**IndividualBand**	
		Column Name	**Data Type**	
		ID	INT	The primary key value for the IndividualBand table.
		IndividualID	INT	A foreign key field copied over from the ID column of the Individual table
		BandID	INT	A foreign key field copied over from the ID column of the band table
Table Name:	**Band**			
Column Name	**Data Type**			
ID	INT			
Name	Varchar(25)			
YearFormed	Year(4)			
IsTogether	Boolean			
MusicGenre	Enum			

Let's look at some sample data for the tables listed above. Below, notice how Neil Young is now associated with two bands: CSNY and Crazy Horse:

Individual Table						
ID	**FirstName**	**LastName**	**BirthCountry**	**BirthDate**	Hometown	Biography
1	Mick	Jagger	England	7/26/1943	Dartford	English music
2	George	Harrison	England	2/25/1943	Liverpool	English music
3	Neil	Young	Canada	11/12/1945	Toronto	He began per
4	Helm	Levon	United States	5/26/1940	Elaine, Arkansas	American roc
5	Ringo	Starr	England	7/7/1940	Liverpool	English music
6	Ronnie	Wood	England	6/1/1947	Hillingdon	English rock
Band Table						
ID	**Name**	**YearFormed**	**IsTogether**	**MusicGenre**		
1111	The Beatles	1969	No	Rock		
1234	The Rolling Stones	1962	Yes	Rock		
1222	The Band	1964	No	Rock		
2222	CSNY	1968	No	Rock		
3333	Crazy Horse	1969	Yes	Rock		
IndvidualBand Table						
ID	**IndividualID**	**BandID**				
100	1	1234				
200	2	1111				
300	3	2222				
400	3	3333				
500	4	1222				

EXERCISE: BUILDING A CROSS REFERENCE TABLE TO FORM THE MANY TO MANY RELATIONSHIP

We don't want to allow an entry of rows into the **IndividualBand** table unless the **Individual ID** and **Band ID** area known in the **Individual** and **Band** tables. To enforce this we can create 2 foreign key relationships. The first foreign key relationship will be between the **Individual** and **IndividualBand** tables and a second foreign key relationship will be placed between the **Band** and **IndividualBand** tables.

1. Using the table below as a guide, build the **IndividualBand** table. Each foreign key column will require an index. Also be sure to check the data types on the primary key columns on the parent tables, especially whether the INT data type is UNSIGNED or SIGNED. Each foreign key will require a cascade delete and cascade update. Good luck.

Table Name:	**IndividualBand**								
Column Name	**Data Type**	**Signed**	**Allow Nulls**	**Auto Increment**	**Primary Key**	**Index**	**Foreign Key Constraint**	**Cascase Delete?**	**Cascade Update?**
ID	INT	No	No	Yes	Yes				
BandID	INT	No	No	No		Yes	Yes	Yes	Yes
IndividualID	INT	NO	No	No		Yes	Yes	Yes	Yes

<END EXERCISE>

SECTION 4: BASIC SQL STATEMENTS

EXERCISE: CREATING A DATABASE FROM A .SQL SCRIPT FILE

Let's start this exercise by opening a terminal window/command prompt. We will create a new database named **RockStarDay2**, create tables within the database, and populate the tables with data. We will do this using a .sql script file that contains a series of SQL commands. We will pipe the instructions stored within a file named rockstar.sql. In this way, we can all work from the exact same database schema (structure) and data. The tables will look familiar. We have an Individual, Band, etc. tables in the database.

To run the sql file we will run a command similar to the following:

```
$ mysql < filename.sql -u root -p
```

But, you will need to make sure you're in the directory where that script is kept or provide the full path to the script. Again, your path will vary from the one provided in the example below:

```
$ mysql < C:\Users\tripot\Dropbox\CharlestonCodes\MySQL\SQLScripts\RockStar\rockstar.sql
-u root -p
```

Create a folder on your mac to hold the rockstar.sql script file.

1. Create a folder on your mac to hold the rockstar.sql script file.
2. Ask the instructor for the location of the course files which will include the rockstar.sql file. Your instructor will provide you with the files for the course. Place the files on your local computer within the subfolder you created in the previous step.
3. Open the sql file (it's just a text file) and inspect its contents. Review the SQL statements within the script.
4. In OS X, open the Terminal. On OS X, open your Applications folder, then open the Utilities folder. Open the Terminal application. You may want to add this to your dock. I like to launch terminal by using Spotlight search in OS X, searching for "terminal".
5. Make sure you are in the directory where you placed your script. Or, make sure you know the entire path to the location of the rockstar.sql file. The directory names are case sensitive. You can change your directory with cd (short for change directory). If you pass it an argument, it will change your to that location, if it exists. Without an argument, it will take you to your home directory (~).

   Example: $ cd Documents

   Soon we will need to connect to the MySQL Database Server, we will call upon the **`mysql`** program from a command prompt (Unix/Mac OSX terminal window). In the example below, my rockstar.sql file resides within a directory named

   "C:\Users\tripot\Dropbox\CharlestonCodes\MySQL\SQLScripts\RockStar". Your path will vary. Check with your instructor.

`$ mysql < C:\Users\tripot\Dropbox\CharlestonCodes\MySQL\SQLScripts\RockStar\rockstar.sql -u root -p`

   a. **`-h`** this is the host where the MySQL Server application is running. In this course, the MySQL Database Server is your host machine. Since the host is the same computer from where you are running the **`mysql`** program, you can omit the -h option.
   b. **`-u`** the MySQL user name. If using Unix this is the same as the Unix login name. If you want to use the Unix login name as the user name, you can omit the –u option. Use the same user name as before.
   c. **`-p`** Leave this bland and you will be prompted for the password. You will use the same password as before.

6. Let's run the rockstar.sql script by connecting to the database server and piping in the sql script. Run the command at the command prompt. Make sure you are in the directory where you placed your script. Or, make sure you know the entire path to the location of the rockstar.sql file. The directory names are case sensitive. If you don't specify a password with the -p option, you will be prompted for your mysql user password.

```
$ mysql < rockstar.sql -u root –p
```

7. At the mysql prompt, enter the following command

```
mysql> USE RockStarDay2;
```

8. At the mysql prompt, select all the rows from the Band table

```
mysql> Select * from Band;
```

<END EXERCISE>

NEXT STEPS: ON YOUR OWN TIME, CHECK OUT THE FOLLOWING CONTENT WITHIN THE ONLINE MYSQL DOCUMENTATION FOR MORE GUIDANCE ON EXECUTING SQL STATEMENTS FROM A TEXT FILE:

HTTP://DEV.MYSQL.COM/DOC/REFMAN/5.0/EN/MYSQL-BATCH-COMMANDS.HTML

SELECT STATEMENTS

In our last session we spent a lot of time creating tables and relationships. You could say we defined the data structures. There is a technical name for this and it's called the Data Definition Language or DDL. We used the DDL portion of the Structured Query Language to define our data structures.

There is more than just building tables. Using SQL you can retrieve data and even change data within the database using SQL. These types of SQL statement are called Data Manipulation Language statements (DML). Since SQL is an ANSI standard language. You can use it on other relational database platforms like SQL Server, Oracle, Access, Sybase, DB2, and others. Each database vendor adheres to *most* of the ANSI standards for SQL which can lead to SQL that is not completely portable across different systems.

BEWARE! WHEN A VENDOR DEVIATES FROM THE STANDARD, YOU RUN THE RISK OF REWRITING SQL CODE IF YOU WANT TO SWITCH TO A DIFFERENT PLATFORM.

EXERCISE: BUILDING A SELECT STATEMENT

The SELECT statement is used to retrieve data from the database. Take the following `SELECT` statement as an example:

```
SELECT * FROM Individual;
```

The statement above almost reads like English, doesn't it? It simply states, `SELECT` for me all the columns `FROM` the `Individual` table. This SQL statement will retrieve all the rows from the table and all the columns. The '*' means return all the columns and since we are not retricting which rows to return, the statement will return all rows.

Tip: The `SELECT` statement is not case sensitive. So, `select * from Individual;` works too!

1. Within the terminal, use mysql to connect to the database server. We will be querying from the database named RockStarDay2. Be sure to use the USE <databasename> statement to point your SQL queries at the correct database. At the mysql> prompt, enter **`SELECT * FROM INDIVIDUAL;`** and hit enter to view the all the rows and columns in the table.

```
sugaree@ottUbuntuVirtualBox:~$ mysql -u root -pPassword1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 43
Server version: 5.5.29-0ubuntu0.12.04.1 (Ubuntu)

Copyright (c) 2000, 2012, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> USE RockStarDay2;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SELECT * FROM Individual;
+----+--------------+-----------+------------+---------------------+--------------+
| ID | LastName     | FirstName | BirthDate  | DateAdded           | DeceasedDate |
+----+--------------+-----------+------------+---------------------+--------------+
|  1 | Jagger       | Mick      | 1943-07-26 | 2013-02-21 20:25:20 | NULL         |
|  2 | Zimmerman    | Robert    | 1942-05-25 | 2013-02-21 20:25:20 | NULL         |
|  3 | Cobain       | Kurt      | 1967-02-20 | 2013-02-21 20:25:20 | NULL         |
|  4 | Harrison     | George    | 1943-02-25 | 2013-02-21 20:25:20 | NULL         |
|  5 | Buck         | Peter     | 1956-12-06 | 2013-02-21 20:25:20 | NULL         |
|  6 | Young        | Neil      | 1945-11-12 | 2013-02-21 20:25:20 | NULL         |
|  7 | Helm         | Levon     | 1940-05-26 | 2013-02-21 20:25:20 | 2012-04-19   |
|  8 | Cash         | Johnny    | 1932-02-26 | 2013-02-21 20:25:20 | 2012-09-12   |
|  9 | Presley      | Elvis     | 1935-01-08 | 2013-02-21 20:25:20 | 1977-08-16   |
| 10 | Plant        | Robert    | 1948-08-20 | 2013-02-21 20:25:20 | NULL         |
| 11 | Townsend     | Pete      | 1945-05-19 | 2013-02-21 20:25:20 | NULL         |
| 12 | Hendrix      | Jimi      | 1942-11-27 | 2013-02-21 20:25:20 | 1970-09-18   |
| 13 | Baker        | Ginger    | 1939-08-13 | 2013-02-21 20:25:20 | NULL         |
| 14 | Rotten       | Johnny    | 1956-01-31 | 2013-02-21 20:25:20 | NULL         |
| 15 | Strummer     | Joe       | 1952-08-21 | 2013-02-21 20:25:20 | 2002-12-22   |
| 16 | Francis      | Black     | 1965-05-06 | 2013-02-21 20:25:20 | NULL         |
| 17 | Deal         | Kim       | 1961-06-10 | 2013-02-21 20:25:20 | 2012-04-19   |
| 18 | Mills        | Mike      | 1958-12-17 | 2013-02-21 20:25:20 | NULL         |
| 19 | Bell         | John      | NULL       | 2013-02-21 20:25:20 | NULL         |
| 20 | Houser       | Michael   | NULL       | 2013-02-21 20:25:20 | NULL         |
| 21 | JoJo         | Hermann   | NULL       | 2013-02-21 20:25:20 | NULL         |
| 22 | Schools      | David     | NULL       | 2013-02-21 20:25:20 | NULL         |
```

THE SELECT CLAUSE

So, I guess you noticed the SELECT within the previous exercise. The word 'SELECT' in SQL is called a clause and is used to specify the columns to be returned by your query. Whenever you see the SELECT clause, you are retrieving data. The SQL statement 'Select * from Individual;' will return all the columns from the Individual table. You could also specify which columns to return. For example, you could author the following:

```
SELECT ID, FirstName, LastName, BirthDate FROM Individual;
```

The statement above will retrieve the ID column, FirstName column, LastName column, BirthDate column FROM the Individual table.

Pro Tip: For performance reasons, use only use the least amount of columns in your SELECT statement. It is generally a good idea to refrain from using * as this will retrieve all the columns which wastes computing resources like CPU.

2. Let's find out about all the columns within our Band table. Run the following `SHOW COLUMNS FROM Band;`
3. Next, retrieve the ID, Name, and Genre columns from the Band table. If you need to cheat, look at the screen shot below.

```
mysql> SHOW COLUMNS FROM Band;
+------------+-----------------------------------------------------------------------+------+-----+---------+----------------+
| Field      | Type                                                                  | Null | Key | Default | Extra          |
+------------+-----------------------------------------------------------------------+------+-----+---------+----------------+
| ID         | int(10) unsigned                                                      | NO   | PRI | NULL    | auto_increment |
| Name       | varchar(40)                                                           | YES  |     | NULL    |                |
| YearFormed | year(4)                                                               | NO   |     | NULL    |                |
| IsTogether | tinyint(1)                                                            | NO   |     | 1       |                |
| Genre      | enum('Rock','Alternative','Country','Funk','Grunge','Bluegrass') | YES  |     | NULL    |                |
+------------+-----------------------------------------------------------------------+------+-----+---------+----------------+
5 rows in set (0.02 sec)

mysql> Select ID, NAME, Genre from Band;
+----+--------------------------------+-------------+
| ID | NAME                           | Genre       |
+----+--------------------------------+-------------+
|  1 | Rolling Stones                 | Rock        |
|  2 | Beatles                        | Rock        |
|  3 | Traveling Wilburys             | Rock        |
|  4 | Nirvana                        | Grunge      |
|  5 | REM                            | Alternative |
|  6 | Crazy Horse                    | Rock        |
|  7 | Pixies                         | Alternative |
|  8 | Widespread Panic               | Rock        |
|  9 | Jouney                         | Rock        |
| 10 | P Funk AllStars                | Funk        |
| 11 | Def Leppard                    | Rock        |
| 12 | Cream                          | Rock        |
| 13 | George Jones                   | Country     |
| 14 | Garth Brooks                   | Country     |
| 15 | Alison Kraus and Union Station | Bluegrass   |
| 16 | Alan Jackson                   | Country     |
| 17 | Clint Black                    | Country     |
| 18 | Merle Haggard                  | Country     |
| 19 | Hank Williams                  | Country     |
| 20 | Waylon Jennings                | Country     |
| 21 | The Highwaymen                 | Country     |
| 22 | The Buckeroos                  | Country     |
| 23 | The Band                       | Country     |
+----+--------------------------------+-------------+
23 rows in set (0.01 sec)
```

<END EXERCISE>

RESTRICTING THE ROWS RETURNED

We will now introduce another clause into the mix, the WHERE clause. Use it along with an expression to tell the system which rows you do or do not want. There WHERE clause contains conditions to filter the returned rows. Rows which pass the filter conditions are allowed to be returned. Here are some examples of queries with a WHERE clause:

EXERCISE: TAKING THE WHERE CLAUSE FOR A SPIN

Let's get some practice using the WHERE clause. Using the WHERE clause will soon become second nature as it is an extremely useful thing! After you run each statement, try and verbalize each SELECT statement as a sentence. For example, the first statement below would read like this: "Grab every column of data from the Individual table where the Individual's last name is equal to 'Jennings'.

```
1. Select * FROM Individual WHERE LastName = 'Jennings'
2. Select ID, LASTNAME FROM Individual WHERE DeceasedDate IS NOT NULL;
3. Select ID, LastName, BirthDate FROM Individual WHERE Year(BirthDate) > 1940;
4. Select * FROM Individual WHERE ID IN (1,3,5,7,19);
```

<END EXERCISE>

EXERCISE: TRANSLATING REQUIREMENTS TO SQL STATEMENTS

So, after this course you will be able to answer some interesting data questions using a relational database. Let's prepare for that glorious day! Now, translate the following questions to SQL statements.

1. Provide a list of alternative bands that have broken up.

2. Give me the id and name of the band with an ID of 4

<END EXERCISE>

SQL WILDCARDS

You can use SQL Wildcards to help filter your data in the WHERE clause. Wildcards follow the LIKE operator. The LIKE operator allows you to apply wildcards to apply pattern matching to perform the filtering of data.

Wildcard	Description
%	Use % to match any string. Example: **`SELECT * FROM Band WHERE Name LIKE '%Stones';`** This query will return all band records that have a name that ends in 'Stones'. Here is another example. This query will return all individuals with the first name begins in 'L%' **`Select * from Individual WHERE FirstName LIKE 'L%';`**
_	The underscore or _ can be used as a substitution for a single character. For example, this query would return anyone with the first name of 'Rob' or 'Bob'. **`Select * from Individual WHERE FirstName Like '_ob';`**

EXERCISE: USING WILDCARDS IN A WHERE

Okay. Now it's your turn to try out some wildcard characters in your WHERE clauses. Let's warm up by running the following queries.

```
1. SELECT * FROM Band WHERE Name LIKE 'The%';
2. SELECT * FROM Band WHERE Name LIKE '%Stones';
3. SELECT * FROM Individual WHERE FirstName Like '__ck';
```

Now try your hand at deciphering the requirements into your own SQL statements.

4. Find all bands that have an 'and' in the middle of their band name like 'Jason and the Scorchers'.
5. Find people who have a three letter first name that is like 'Tim' or 'Kim' or 'Jim'.
6. Find all bands that end with 's' like 'The Beatles' or 'The Rolling Stones' or 'Pixies'.

<END EXERCISE>

SECTION 4: RELATING TABLES TOGETHER WITH JOINS

INNER JOIN

An INNER JOIN is the most common type of join. An INNER JOIN will output only rows that match between tables. An INNER JOIN clause is used to join two or more tables together based on a common field to produce a result set. Another way to say this is the query will return rows where the join condition is met. So, let's say you have two tables: team and batting. The team table displays a listing of MLB teams while the batting table lists batting leaders.

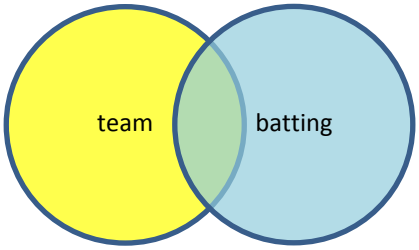
Here is the team table. Note the values in the ID column:

```
mysql> use baseball
Database changed
mysql> select * from team;
+----+-----------------------+------+--------+--------------+
| ID | TeamName              | ABBR | League | DivisionName |
+----+-----------------------+------+--------+--------------+
|  1 | Baltimore Orioles     | BAL  | AL     | AL EAST      |
|  2 | Boston Red Sox        | BOS  | AL     | AL EAST      |
|  3 | New York Yankees      | NYY  | AL     | AL EAST      |
|  4 | Toronto Blue Jays     | TOR  | AL     | AL EAST      |
|  5 | Chicago White Sox     | CHW  | AL     | AL CENTRAL   |
|  6 | Cleveland Indians     | CLE  | AL     | AL CENTRAL   |
|  7 | Detroit Tigers        | DET  | AL     | AL CENTRAL   |
|  8 | Kansas City Royals    | KAN  | AL     | AL CENTRAL   |
|  9 | Minnesota Twins       | MIN  | AL     | AL CENTRAL   |
| 10 | Houston Astros        | HOU  | AL     | AL WEST      |
| 11 | Atlanta Braves        | ATL  | NL     | NL EAST      |
| 12 | Miami Marlins         | MIA  | NL     | NL EAST      |
| 13 | New York Mets         | NYM  | NL     | NL EAST      |
| 14 | Philadelphia Phillies | PHI  | NL     | NL EAST      |
| 15 | Washington Nationals  | WAS  | NL     | NL EAST      |
| 16 | Chicago Cubs          | CHC  | NL     | NL CENTRAL   |
| 17 | Cincinnati Reds       | CIN  | NL     | NL CENTRAL   |
| 18 | Milwaukee Brewers     | MIL  | NL     | NL CENTRAL   |
| 19 | Pittsburgh Pirates    | PIT  | NL     | NL CENTRAL   |
| 20 | St. Louis Cardinals   | STL  | NL     | NL CENTRAL   |
| 21 | Arizona Diamondbacks  | ARI  | NL     | NL WEST      |
| 22 | Colorado Rockies      | COL  | NL     | NL WEST      |
| 23 | Los Angeles Dodgers   | LAD  | NL     | NL WEST      |
| 24 | San Diego Padres      | SD   | NL     | NL WEST      |
| 25 | San Francisco Giants  | SF   | NL     | NL WEST      |
| 26 | Los Angeles Angels    | LAA  | AL     | AL WEST      |
| 27 | Oakland Athletics     | OAK  | AL     | AL WEST      |
| 28 | Seattle Mariners      | SEA  | AL     | AL WEST      |
| 29 | Texas Rangers         | TEX  | AL     | AL WEST      |
| 30 | Tampa Bay Rays        | TAM  | AL     | AL EAST      |
+----+-----------------------+------+--------+--------------+
30 rows in set (0.43 sec)
```

And here is the batting table. Note the values in the TeamID column:

```
mysql> Select * from batting;
+----+----------+------+--------+--------+------+------+------------+
| ID | PlayerID | Rank | TeamID | AtBats | Runs | Hits | BattingAvg |
+----+----------+------+--------+--------+------+------+------------+
| 16 |      101 |    1 |     22 |    489 |   74 |  162 |        331 |
| 17 |       60 |    2 |     11 |    514 |   54 |  165 |        321 |
| 18 |      103 |    3 |     11 |    551 |   89 |  176 |        319 |
| 19 |      104 |    4 |     20 |    505 |   68 |  161 |        319 |
| 20 |      105 |    5 |     15 |    462 |   84 |  147 |        318 |
| 21 |      106 |    6 |     20 |    626 |  126 |  199 |        318 |
| 22 |      107 |    7 |     20 |    583 |   97 |  185 |        317 |
| 23 |      108 |    8 |     20 |    508 |   71 |  160 |        315 |
+----+----------+------+--------+--------+------+------+------------+
8 rows in set (0.08 sec)
```

Did you notice that the values within the batting.TeamID column match the columns within the Team's ID column? You can join these two tables using a SELECT statement with an INNER JOIN clause to discover the teams for the batting leaders by joining the team.ID column with the batting.TeamID column. The query will only return rows where the Team IDs match as represented by the overlap (green area) within the two circles. If we were to draw a picture it would look like this.



The syntax for an INNER JOIN (JOIN) looks like this:

```
Select <Column List>

FROM <Table_A> INNER JOIN <TABLE_B> ON <Table_A>.<Column_Name> = <TableB>.<Column_Name>
```

Here is an example of an INNER JOIN in action. Note the use of "INNER JOIN" and "ON" to form the join condition:

```
mysql> Select team.ID, team.TeamName,
    -> batting.TeamID, batting.BattingAvg
    -> FROM team
    -> INNER JOIN batting ON batting.TeamID = team.ID;
+----+----------------------+--------+------------+
| ID | TeamName             | TeamID | BattingAvg |
+----+----------------------+--------+------------+
| 22 | Colorado Rockies     |     22 |        331 |
| 11 | Atlanta Braves       |     11 |        321 |
| 11 | Atlanta Braves       |     11 |        319 |
| 20 | St. Louis Cardinals  |     20 |        319 |
| 15 | Washington Nationals |     15 |        318 |
| 20 | St. Louis Cardinals  |     20 |        318 |
| 20 | St. Louis Cardinals  |     20 |        317 |
| 20 | St. Louis Cardinals  |     20 |        315 |
+----+----------------------+--------+------------+
8 rows in set (0.18 sec)
```

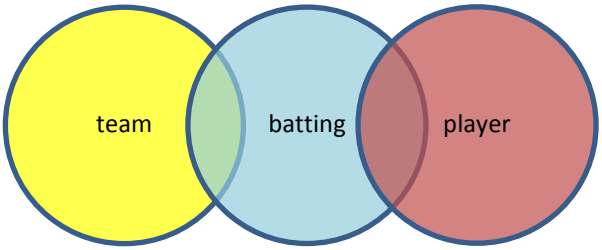
Here is the batting table again. Note the **PlayerID** column:

```
mysql> Select * from batting;
+----+----------+------+--------+--------+------+------+------------+
| ID | PlayerID | Rank | TeamID | AtBats | Runs | Hits | BattingAvg |
+----+----------+------+--------+--------+------+------+------------+
| 16 |      101 |    1 |     22 |    489 |   74 |  162 |        331 |
| 17 |       60 |    2 |     11 |    514 |   54 |  165 |        321 |
| 18 |      103 |    3 |     11 |    551 |   89 |  176 |        319 |
| 19 |      104 |    4 |     20 |    505 |   68 |  161 |        319 |
| 20 |      105 |    5 |     15 |    462 |   84 |  147 |        318 |
| 21 |      106 |    6 |     20 |    626 |  126 |  199 |        318 |
| 22 |      107 |    7 |     20 |    583 |   97 |  185 |        317 |
| 23 |      108 |    8 |     20 |    508 |   71 |  160 |        315 |
+----+----------+------+--------+--------+------+------+------------+
8 rows in set (0.00 sec)
```

The **PlayerID** column relates to the **ID** column of the player table. Here are a few columns from the player table:

```
mysql> Select ID, FirstName, LastName, Throws, Bats from Player;
+-----+-----------+-------------+--------+------+
| ID  | FirstName | LastName    | Throws | Bats |
+-----+-----------+-------------+--------+------+
|   5 | Starlin   | Castro      | R      | R    |
|  10 | Nate      | Schierholtz | R      | L    |
|  15 | Logan     | Watkins     | R      | L    |
|  20 | Anthony   | Rizzo       | L      | L    |
|  25 | Kyuji     | Fujikawa    | R      | L    |
|  30 | Kris      | Medlen      | R      | R    |
|  35 | Craig     | Kimbrel     | R      | R    |
|  40 | Alex      | Wood        | L      | R    |
|  45 | Mike      | Minor       | L      | R    |
|  50 | Evan      | Gattis      | R      | R    |
|  55 | Jason     | Heyward     | L      | L    |
|  60 | Chris     | Johnson     | R      | R    |
|  65 | Justin    | Upton       | R      | R    |
|  70 | Clay      | Buchholz    | R      | L    |
|  75 | John      | Lackey      | R      | R    |
|  80 | Ryan      | Dempster    | R      | R    |
|  85 | David     | Ross        | R      | R    |
|  90 | Dustin    | Pedroia     | R      | R    |
|  95 | David     | Ortiz       | L      | L    |
| 101 | Michael   | Cuddyer     | R      | R    |
| 103 | Freddie   | Freeman     | L      | R    |
| 104 | Yadier    | Molina      | R      | R    |
| 105 | Jayson    | Werth       | R      | R    |
| 106 | Matt      | Carpenter   | L      | R    |
| 107 | Andrew    | McCutchen   | R      | R    |
| 108 | Allen     | Craig       | R      | R    |
+-----+-----------+-------------+--------+------+
26 rows in set (0.06 sec)
```

We can add another INNER JOIN to relate the player table to the batting table.



We will need to join the batting.PlayerID column to the player.ID column like this:

```
SELECT batting.PlayerID, batting.Rank, player.FirstName, player.LastName
, batting.BattingAvg, batting.TeamID, team.TeamName
FROM team INNER JOIN batting on team.ID = batting.TeamID
INNER JOIN player ON batting.PlayerID = player.ID;
```

```
mysql> SELECT batting.PlayerID, batting.Rank, player.FirstName, player.LastName
    -> , batting.BattingAvg, batting.TeamID, team.TeamName
    -> FROM team INNER JOIN batting on team.ID = batting.TeamID
    -> INNER JOIN player ON batting.PlayerID = player.ID;
+----------+------+-----------+-----------+------------+--------+----------------------+
| PlayerID | Rank | FirstName | LastName  | BattingAvg | TeamID | TeamName             |
+----------+------+-----------+-----------+------------+--------+----------------------+
|      101 |    1 | Michael   | Cuddyer   |        331 |     22 | Colorado Rockies     |
|       60 |    2 | Chris     | Johnson   |        321 |     11 | Atlanta Braves       |
|      103 |    3 | Freddie   | Freeman   |        319 |     11 | Atlanta Braves       |
|      104 |    4 | Yadier    | Molina    |        319 |     20 | St. Louis Cardinals  |
|      105 |    5 | Jayson    | Werth     |        318 |     15 | Washington Nationals |
|      106 |    6 | Matt      | Carpenter |        318 |     20 | St. Louis Cardinals  |
|      107 |    7 | Andrew    | McCutchen |        317 |     20 | St. Louis Cardinals  |
|      108 |    8 | Allen     | Craig     |        315 |     20 | St. Louis Cardinals  |
+----------+------+-----------+-----------+------------+--------+----------------------+
8 rows in set (0.00 sec)
```

EXERCISE: USING AN INNER JOIN

Let's create and populate a new database named 'baseball'. Your instructor has provided a sql script file named 'baseball.sql'. Use this file to create the database.

1. You will need to exit out of the mysql terminal via the 'exit' command.
2. Open the baseball.sql file within a text editor and examine its contents. Note the structures of the tables, its foreign keys and the data that is placed into the tables. Note the CREATE VIEW statements, too.
3. Reconnect to mysql with a command that looks like this. Below, note how I am providing the full path to the sql file since I am not currently in the directory that houses this sql file. Be sure to use the correct user name and associated password.

```
C:\Users\tripot>mysql < C:\Users\tripot\Dropbox\CharlestonCodes\MySQL\SQLScripts\RockStar\baseball.s
ql -u root -p
Enter password: ********
```

4. RE-connect to mysql
5. Once connected, use the 'show databases;' command to list the databases. You should see the new 'baseball' database.
6. Make the baseball database the default database by issuing 'use baseball;' command.
7. Use the 'show tables' command to list the tables in the database.
8. Create a separate SELECT statement to retrieve all the rows and columns from each of the tables.
    a. Ex: Select * from player;
    b. Ex: Select * from team;
    c. Ex: Select * from batting;
    d. Ex: Select * from roster;
9. Create a SELECT statement that joins the batting table to the team table like this:

```
Select team.ID
, team.TeamName
, batting.TeamID
, batting.Rank
, batting.BattingAvg
FROM team
INNER JOIN batting on team.ID = batting.TeamID;
```

10. Create a SELECT statement that joins the batting table to the player table.

INSERTING ROWS INTO A TABLE

To add a row or rows of data into a table use the INSERT INTO statement. With INSERT INTO you provide the name of the table, the table columns, the VALUES keyword followed by a comma delimited list of values. The order of the values should correspond to the column names. The syntax looks like this

```
INSERT INTO tablename (Col1, Col2, Col3, ...) VALUES (Value1, Value2, Value3, …).
```

Exercise: Add an Individual to a Band

1. Use a SELECT statement to list the contents of the IndividualBand table:

   ```
   Select * from IndividualBand;
   ```

2. The contents of IndividualBand is kind of hard to decipher, right? Luckily, a ***view*** has been provided that joins together the Individual, IndividualBand, and Band tables into a view that relates the 3 tables together. A ***view*** is the result set of a stored query. A view defines a virtual table. You can use a SELECT statement to query a VIEW in the same way, you query a regular database table. The view's name is BandMembers. Go ahead and query BandMembers by running the following SQL statement:

   ```
   Select * from BandMembers;
   ```

3. Let's play around with the INSERT INTO statement by associating a Band with an Individual. The IndividualBand table helps to relate bands to individuals. Using the RockStarDay2 database, add Eric Clapton (ID = 31) to the band Cream (ID = 12) by executing the following SQL DML statement.

   ```
   INSERT INTO IndividualBand (BandID, IndividualID) Values (12,31);
   ```

4. After you have added Eric Clapton to the band Cream, re-query the BandMembers view by issuing a SELECT statement on the BandMembers view.
5. Are you curious about the contents of the BandMembers view? Would you like to see how it is defined? Run the following SQL statement and view the entrails of the BandMembers view:

   ```
   SHOW CREATE VIEW BandMembers;
   ```

   The results are a little hard to decipher but its good experience, anyway.

6. More practice. Use the table below to add the individuals to their respective bands:

Individual (ID)	Band (ID)
Buck Owens (47)	The Buckeroos (22)
Johnny Cash (8)	The Highwaymen (21)
Waylon Jennings (39)	The Highwaymen (21)

EXERCISE: MORE THAN ONE WAY TO INSERT INTO

Let's use a form of the INSERT INTO statement where we can add multiple records at once by including multiple sets of values. Each data set is wrapped in parenthesis and separated by a comma. Here is an example of an INSERT INTO statement that adds multiple rows into the Band table.

```
INSERT INTO Band

(Name,YearFormed,IsTogether,Genre)

VALUES
```

```
('Rolling Stones', '1962', 1, 'Rock')

, ('Beatles', '1960', 0, 'Rock')

, ('Traveling Wilburys', '1988', 0, 'Rock')

, ('Nirvana', '1987', 0, 'Grunge');

, ('REM', '1980', 0, 'Alternative');
```

1. Go ahead and build 3 separate INSERT INTO statements to add some of your favorite rock stars or into the Individual table, Band table, and IndividualBand tables.

EXERCISE: INSERT INTO SELECT

The INSERT INTO SELECT syntax allows you to SELECT rows from a table or tables and use those rows to INSERT INTO a table. Let's use a predefined SQL script file which creates a new table named **Ramones** and populates that table with rock stars. Once the table has been created, we can author an INSERT INTO SELECT data manipulation language (DML) SQL statement to SELECT from Ramones and INSERT INTO the existing table named Individual. Here is an example of the syntax:

```
INSERT INTO TableA (ID, Name, Description)

  SELECT TableB.ID, TableB.Name, TableB.Desc

  FROM TableB WHERE TableB.Price > 9.99 ORDER BY TableB.NAME;
```

1. Ask the instructor for the **ramones.sql** file.
2. Open the sql file (it's just a text file) and inspect its contents. Review the SQL statements within the script. The file is named **ramones.sql.**
3. In OS X, open a Terminal. If you already have the terminal open and you are connected to mysql, use the exit command to quit mysql.
4. Within the Terminal, ensure you are in the directory where you placed your script. The directory names are case sensitive.
5. Connect to the MySQL Database Server and run the script, we will call upon the **`mysql`** program from a command prompt (Windows/Unix/MacOSX terminal window). This will create a table named **Ramones** and populate the table with some individuals.

   ```
   $ mysql < ramones.sql -u root -p
   ```

6. Connect to mysql again with the following command:

   ```
   $ mysql -u root –p
   ```

7. Once you are at the mysql> prompt, switch to the RockStarDay2 database and author a SELECT statement that retrieves the rows form the new **Ramones** table. The screenshot below is from a MS Windows command window but you get the idea:

```
D:\Files\Personal\Dropbox\CharlestonCodes\MySQL\SQL Scripts\RockStar>mysql < ramones.sql -u root -pPassword1

D:\Files\Personal\Dropbox\CharlestonCodes\MySQL\SQL Scripts\RockStar>mysql -u root -pPassword1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7
Server version: 5.5.27 MySQL Community Server (GPL)

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> USE RockStarDay2;
Database changed
mysql> Select * from Ramones;
+----+----------+-----------+------------+---------------------+--------------+
| ID | LastName | FirstName | BirthDate  | DateAdded           | DeceasedDate |
+----+----------+-----------+------------+---------------------+--------------+
|  1 | Ramone   | Joey      | 1951-05-19 | 2013-04-12 16:44:41 | 2001-04-15   |
|  2 | Burke    | Clem      | 1955-11-24 | 2013-04-12 16:44:41 | NULL         |
|  3 | Ramone   | Johnny    | 1948-10-08 | 2013-04-12 16:44:41 | 2004-09-15   |
|  4 | Ramone   | Dee Dee   | 1951-09-18 | 2013-04-12 16:44:41 | NULL         |
+----+----------+-----------+------------+---------------------+--------------+
4 rows in set (0.00 sec)
```

With the **Ramones** table created and populated, we can author the **`INSERT INTO SELECT`** statement. We want to copy the records from the **Ramomes** table into the **Individual** table. We start with "**`INSERT INTO Individual`**" to state the table we are adding rows into. Next we need to list the columns for the Individual table. At this point your sql statement should look something like this. Notice we have left off the **ID** field since its value will auto increment. We also leave out the **DateAdded** since that data will be defaulted with the current date for each row inserted.

```
INSERT INTO Individual (LastName, FirstName, BirthDate, DeceasedDate)
```

We are not yet done. We use add a **`SELECT`** statement to grab the data from the **Ramones** table. And be sure to get the order correct on the columns from the **Ramones** table. At this point your SQL statement should look like this:

```
INSERT INTO Individual (LastName, FirstName, BirthDate, DeceasedDate)
SELECT LastName, FirstName, BirthDate, DeceasedDate
FROM Ramones
ORDER BY DateAdded;
```

8. Fire it up! Run the SQL statement to insert the rows!

```
mysql> Use Rockstarday2;
Database changed
mysql> SELECT * from Ramones;
+----+----------+-----------+------------+---------------------+--------------+
| ID | LastName | FirstName | BirthDate  | DateAdded           | DeceasedDate |
+----+----------+-----------+------------+---------------------+--------------+
|  1 | Ramone   | Joey      | 1951-05-19 | 2014-02-05 19:44:24 | 2001-04-15   |
|  2 | Burke    | Clem      | 1955-11-24 | 2014-02-05 19:44:24 | NULL         |
|  3 | Ramone   | Johnny    | 1948-10-08 | 2014-02-05 19:44:24 | 2004-09-15   |
|  4 | Ramone   | Dee Dee   | 1951-09-18 | 2014-02-05 19:44:24 | NULL         |
+----+----------+-----------+------------+---------------------+--------------+
4 rows in set (0.00 sec)

mysql> INSERT INTO Individual (LastName, FirstName, BirthDate, DeceasedDate)
    -> SELECT LastName, FirstName, BirthDate, DeceasedDate
    -> FROM Ramones
    -> ORDER BY DateAdded;
Query OK, 4 rows affected (0.03 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

9. Once you have obtained success, select the rows from the Individual table and confirm.

```
mysql> INSERT INTO Individual (LastName, FirstName, BirthDate, DeceasedDate)
    -> SELECT LastName, FirstName, BirthDate, DeceasedDate
    -> FROM Ramones
    -> ORDER BY DateAdded;
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> Select * from Individual;
+----+---------------+-----------+------------+---------------------+--------------+
| ID | LastName      | FirstName | BirthDate  | DateAdded           | DeceasedDate |
+----+---------------+-----------+------------+---------------------+--------------+
|  1 | Jagger        | Mick      | 1943-07-26 | 2013-04-12 15:54:15 | NULL         |
|  2 | Zimmerman     | Robert    | 1942-05-25 | 2013-04-12 15:54:15 | NULL         |
|  3 | Cobain        | Kurt      | 1967-02-20 | 2013-04-12 15:54:15 | NULL         |
|  4 | Harrison      | George    | 1943-02-25 | 2013-04-12 15:54:15 | NULL         |
|  5 | Buck          | Peter     | 1956-12-06 | 2013-04-12 15:54:15 | NULL         |
|  6 | Young         | Neil      | 1945-11-12 | 2013-04-12 15:54:15 | NULL         |
|  7 | Helm          | Levon     | 1940-05-26 | 2013-04-12 15:54:15 | 2012-04-19   |
|  8 | Cash          | Johnny    | 1932-02-26 | 2013-04-12 15:54:15 | 2012-09-12   |
|  9 | Presley       | Elvis     | 1935-01-08 | 2013-04-12 15:54:15 | 1977-08-16   |
| 10 | Plant         | Robert    | 1948-08-20 | 2013-04-12 15:54:15 | NULL         |
| 11 | Townsend      | Pete      | 1945-05-19 | 2013-04-12 15:54:15 | NULL         |
| 12 | Hendrix       | Jimi      | 1942-11-27 | 2013-04-12 15:54:15 | 1970-09-18   |
| 13 | Baker         | Ginger    | 1939-08-13 | 2013-04-12 15:54:15 | NULL         |
| 14 | Rotten        | Johnny    | 1956-01-31 | 2013-04-12 15:54:15 | NULL         |
| 15 | Strummer      | Joe       | 1952-08-21 | 2013-04-12 15:54:15 | 2002-12-22   |
| 16 | Francis       | Black     | 1965-05-06 | 2013-04-12 15:54:15 | NULL         |
| 17 | Deal          | Kim       | 1961-06-10 | 2013-04-12 15:54:15 | 2012-04-19   |
| 18 | Mills         | Mike      | 1958-12-17 | 2013-04-12 15:54:15 | NULL         |
| 19 | Bell          | John      | 1962-04-14 | 2013-04-12 15:54:15 | NULL         |
| 20 | Houser        | Michael   | 1962-01-06 | 2013-04-12 15:54:15 | 2002-08-10   |
| 21 | JoJo          | Hermann   | NULL       | 2013-04-12 15:54:15 | NULL         |
| 22 | Schools       | David     | 1964-12-11 | 2013-04-12 15:54:15 | NULL         |
| 23 | Perry         | Steve     | 1949-01-22 | 2013-04-12 15:54:15 | NULL         |
| 24 | Schon         | Neal      | 1954-02-27 | 2013-04-12 15:54:15 | NULL         |
| 25 | Clinton       | George    | 1941-07-22 | 2013-04-12 15:54:15 | NULL         |
| 26 | Collins       | Bootsy    | 1951-10-26 | 2013-04-12 15:54:15 | NULL         |
| 27 | Elliot        | Joe       | 1959-08-01 | 2013-04-12 15:54:15 | NULL         |
| 28 | Allen         | Rick      | NULL       | 2013-04-12 15:54:15 | NULL         |
| 29 | Clark         | Steve     | 1960-04-23 | 2013-04-12 15:54:15 | 1991-01-08   |
| 30 | Bruce         | Jack      | 1943-05-14 | 2013-04-12 15:54:15 | NULL         |
| 31 | Clapton       | Eric      | 1945-03-30 | 2013-04-12 15:54:15 | NULL         |
| 32 | Jones         | George    | 1931-08-12 | 2013-04-12 15:54:15 | NULL         |
| 33 | Brooks        | Garth     | 1962-02-07 | 2013-04-12 15:54:15 | NULL         |
| 34 | Jackson       | Alan      | 1958-10-17 | 2013-04-12 15:54:15 | NULL         |
| 35 | Kraus         | Allison   | 1971-07-23 | 2013-04-12 15:54:15 | NULL         |
| 36 | Black         | Clint     | 1962-02-04 | 2013-04-12 15:54:15 | NULL         |
| 37 | Haggard       | Merle     | 1937-04-06 | 2013-04-12 15:54:15 | NULL         |
| 38 | Williams      | Hank      | 1923-09-17 | 2013-04-12 15:54:15 | 1953-01-01   |
| 39 | Jennings      | Waylon    | 1937-06-15 | 2013-04-12 15:54:15 | 2002-02-13   |
| 40 | Lynn          | Loretta   | 1932-04-14 | 2013-04-12 15:54:15 | NULL         |
| 41 | Nelson        | Willie    | 1933-04-30 | 2013-04-12 15:54:15 | NULL         |
| 42 | Cline         | Patsy     | 1932-09-08 | 2013-04-12 15:54:15 | 1963-03-05   |
| 43 | Kristofferson | Kris      | 1936-06-22 | 2013-04-12 15:54:15 | NULL         |
| 44 | Campbell      | Glen      | 1936-04-22 | 2013-04-12 15:54:15 | NULL         |
| 45 | Seger         | Bob       | 1945-05-06 | 2013-04-12 15:54:15 | NULL         |
| 46 | Nugent        | Ted       | 1948-12-13 | 2013-04-12 15:54:15 | NULL         |
| 47 | Owens         | Buck      | 1929-08-12 | 2013-04-12 15:54:15 | 2006-03-25   |
| 48 | Nelson        | Willie    | 1933-04-30 | 2013-04-12 15:54:15 | NULL         |
| 49 | Ramone        | Joey      | 1951-05-19 | 2013-04-12 17:06:44 | 2001-04-15   |
| 50 | Burke         | Clem      | 1955-11-24 | 2013-04-12 17:06:44 | NULL         |
| 51 | Ramone        | Johnny    | 1948-10-08 | 2013-04-12 17:06:44 | 2004-09-15   |
| 52 | Ramone        | Dee Dee   | 1951-09-18 | 2013-04-12 17:06:44 | NULL         |
+----+---------------+-----------+------------+---------------------+--------------+
52 rows in set (0.00 sec)

mysql>
```

EXERCISE: INSERTING A ROW AND DISCOVERING THE VALUE FOR THE LAST INSERTED ID

After a statement that successfully inserts an automatically generated **AUTO_INCREMENT** value, you can find that value by using the **LAST_INSERT_ID()** function. The value returned is on a per connection basis which means this value isn't affected by other clients, even if they generate **AUTO_INCREMENT** values of their own. This is a good thing because it means you can retrieve your value with the LAST_INSERT_ID() function and not be interfered with by others' activity. Let's try out!

1. In the **RockStarDay2** database, the **Individual** table has an auto incremented ID column. Go ahead and verify this with the following statement:
2. **`DESCRIBE INDIVIDUAL`**;

3. Run the following **`SELECT`** statement to check the ID values for all the Individuals. Note the highest value in the ID column.
   `SELECT ID, FIRSTNAME, LASTNAME FROM INDIVIDUAL;`

4. Run the following SELECT statement to grab the biggest ID value from the table:
   `SELECT MAX(ID) as LargestValue FROM INDIVIDUAL;`

5. Now that we have verified the **ID** column is marked as **auto_increment**, we can insert a row with an INSERT statement and then follow up with a call to the **LAST_INSERT_ID()** function. Let's write an **INSERT** statement to add Duane Allman into the Individual table. In 2003, Rolling Stone magazine ranked Allman at #2 in their list of the 100 greatest guitarists of all time, second only to Jimi Hendrix. Duane's birthdate was 11/20/1946 and he died in a motorcycle accident on October 29, 1971. He was 24.

6. After you have successfully added ol' Skydog to the database, use the following statement to discover his ID value.

```
SELECT LAST_INSERT_ID();
```

REMOVING RECORDS FROM A TABLE WITH THE DELETE STATEMENT

Use the **DELETE** statement to remove rows from a table. The syntax is fairly straightforward. In its simplest form you supply a statement with the following syntax:

```
DELETE FROM <table_name>;
```

WATCH OUT!: THE COMMAND **DELETE FROM Individual;** WILL ATTEMPT TO DELETE ***ALL*** THE ROWS FROM THE INDIVIDUAL TABLE. THIS IS ***VERY EASY*** TO DO. THIS IS ***VERY DESTRUCTIVE*** TO DO. USE A **WHERE** CLAUSE TO SPECIFY WHICH ROWS TO DELETE. AS A RESULT, THE **WHERE** CLAUSE WILL LIMIT THE AMOUNT OF ROWS DELETED.

The key to using the **DELETE** statement is to be *double darn* sure you know EXACTLY which rows you want to delete. This is when the **WHERE** clauses comes into play. Use the **WHERE** clause to specify exactly which rows you want to delete. Don't execute the following; but, If you were to issue the following statement:

```
DELETE FROM Individual
WHERE LastName = 'Ramone';
```

Would you know exactly how many rows the statement would delete? To be sure, it's a good idea to test out how selective the **WHERE** clause is by using it with a **SELECT** statement before you attempt to remove the data with the **DELETE** statement. So, let's say you issue the following **SELECT** statement:

```
SELECT * FROM Individual
WHERE LastName = 'Ramone';
```

And as you can see below, it selects 3 rows:

```
mysql> Select * from Individual WHERE LastName = 'Ramone';
+----+----------+-----------+------------+---------------------+--------------+
| ID | LastName | FirstName | BirthDate  | DateAdded           | DeceasedDate |
+----+----------+-----------+------------+---------------------+--------------+
| 49 | Ramone   | Joey      | 1951-05-19 | 2013-08-28 20:21:14 | 2001-04-15   |
| 51 | Ramone   | Johnny    | 1948-10-08 | 2013-08-28 20:21:14 | 2004-09-15   |
| 52 | Ramone   | Dee Dee   | 1951-09-18 | 2013-08-28 20:21:14 | NULL         |
+----+----------+-----------+------------+---------------------+--------------+
3 rows in set (0.02 sec)
```

In summary, the key to using **WHERE** clauses on DELETE statements is to very precise in the way you select the rows for deletion. I like to get the primary key column involved in my WHERE clauses. In this way, you can really be specific. So the SQL statement:

```
DELETE FROM Individual

WHERE ID = '49';
```

Would end up delete the row for Joey Ramone. Poor Joey.

EXERCISE: USING THE DELETE STATEMENT

1. Create a single **SELECT** statement to identify the ID value for Dee Dee Ramone.

2. Use the ID value for Dee Dee to construct a **DELETE FROM** statement to delete Dee Dee's row from the **Individual** table. Go ahead and run the **DELETE** statement. See ya, Dee Dee.

EXERCISE: USING THE IN OPERATOR TO DELETE MULTIPLE RECORDS

Now it's time to get fancy by using the **IN** operator within the **WHERE** clause to identify multiple rows. The **IN** operation is used to test whether or not a value is 'in' the list.

Create the following **SELECT** statement to see how many rows are 'in' the list:

```
SELECT ID, FirstName, LastName

FROM Individual

WHERE LastName IN ('Ramone', 'Jennings','Presley');
```

3. Using the **SELECT** statement above, change the **SELECT** statement to a **DELETE** statement. Execute the **DELETE** statement.

4. If a corresponding row existed for an individual within the **IndividualBand** table, what would happen to the row in the **IndividualBand** table when the row was deleted in the **Individual** table? Would the Foreign Key constraint defined on the **IndividualBand** table prevent the row from being deleted? Hmmmm.
   How can we answer this question? We need a way to view table metadata. We can use the **SHOW** statement to display database metadata. It's helpful for obtaining a picture of how a table was constructed.
   Go ahead and run the following **SHOW** statements and observer their results:

```
SHOW TABLES FROM RockStarDay2;
SHOW COLUMNS FROM IndividualBand;
SHOW  CREATE TABLE IndividualBand;
```

USING A SUB QUERY TO IDENTIFY THOSE BEAUTIFUL ROWS… AND THEN MERCILESSLY DELETE THEM.

UPDATING DATA

"The results of a query that returns multiple rows for the target row to be updated, will update that row with unpredictable results.

Often times I've seen developers that fail to properly analyze the queries they write, most specifically when it comes to determining whether sufficient criteria has been included to ensure the uniqueness of the rows in the output set. "

http://www.sqlservercentral.com/articles/T-SQL/101464/

Updating columns in a table is straighforward. A `where` clause identifies the row (or rows) to update. The `set` clause identifies the columns and their new values.

In the next series of exercises, we will focus on updating some tables in a new database named 'baseball'.

Exercise1; Blindly update all the rows of a table.

Exercise 2; Use a where clause to identify the row or rows to update

Exercise 3: Attempt update that violates a not null constraint. Did the data update?

USING THE DISTINCT KEYWORD FOR COLUMN LISTS WITHIN A DATABASE VIEW

To specify the just the columns you are interested in within your query results, use a comma separated list of column names. So, let's say you want to only select the Name and Genre from the Band table. The query would look like this:

mysql> SELECT Name, Genre FROM Band;

```
C:\Users\TripOt>mysql -uroot -pPassword1
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.5.27 MySQL Community Server (GPL)

Copyright (c) 2000, 2011, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> USE ROCKSTARDAY2;
Database changed
mysql> Select Name,Genre from Band;
+-------------------------------+-------------+
| Name                          | Genre       |
+-------------------------------+-------------+
| Rolling Stones                | Rock        |
| Beatles                       | Rock        |
| Traveling Wilburys            | Rock        |
| Nirvana                       | Grunge      |
| REM                           | Alternative |
| Crazy Horse                   | Rock        |
| Pixies                        | Alternative |
| Widespread Panic              | Rock        |
| Jouney                        | Rock        |
| P Funk AllStars               | Funk        |
| Def Leppard                   | Rock        |
| Cream                         | Rock        |
| George Jones                  | Country     |
| Garth Brooks                  | Country     |
| Alison Kraus and Union Station | Bluegrass   |
| Alan Jackson                  | Country     |
| Clint Black                   | Country     |
| Merle Haggard                 | Country     |
| Hank Williams                 | Country     |
| Waylon Jennings               | Country     |
| The Highwaymen                | Country     |
| The Buckeroos                 | Country     |
| The Band                      | Country     |
+-------------------------------+-------------+
23 rows in set (0.00 sec)

mysql>
```

Now let's write a query that pull rows from a View which was created when the generated the **RockStarDay2** database from the script file. You can create or replace an existing View with the CREATE VIEW statement. To create a view you would use the following syntax:

CREATE VIEW view_name AS select_statement

According to the online MySQL database reference for creating a view "The select_statement is a SELECT statement that provides the definition of the view. (When you select from the view, you select in effect using the SELECT statement.)select_statement can select from base tables or other views.

The view definition is "frozen" at creation time, so changes to the underlying tables afterward do not affect the view definition. For example, if a view is defined as SELECT * on a table, new columns added to the table later do not become part of the view."

You can think of a View as a stored SQL **SELECT** query that has a name. The view is stored within MySQL as a database object. The view is named **bandmembers**. We can **SELECT** the values from this view as if it was an actual table. The bandmembers view will join the individual, band and individualband tables together to provide a list of individuals who are in a band.

5. Run the following SQL statement to create a table, enter some sample data, and then create a view based on the table. The CREATE VIEW statement below also includes an expression that provides a calculation from the columns in the table.

```
mysql> CREATE TABLE inventory (quantity INT, price INT);

mysql> INSERT INTO inventory VALUES(5, 50);

mysql> CREATE VIEW v AS SELECT quantity, price, quantity *price AS value FROM
inventory;

mysql> SELECT * FROM v;

+-----------+-------+-------+

| quantity  | price | value |

+-----------+-------+-------+

|    5      |    50 |   250 |

+-----------+-------+-------+
```

6. Go ahead and write a SELECT statement from the bandmembers view. So, if you execute the following SELECT statement:

```
mysql> SELECT GENRE, NAME, LastName, FirstName, BirthDate FROM bandmembers;
```

   The SELECT statement within the view will join the band, individual, and individualband tables together and provide a list of individuals who are in bands.

If we wanted to extract a list of bands that have band members, we could shorten the column list in the previous query to something like this:

7. Go ahead and run the following query. Notice the redundant names of bands have multiple band members.

```
mysql> SELECT NAME FROM bandmembers;
```

We would like to remove the redundant band names. Use the DISTINCT keyword to remove the duplicate names from the results.

8. Go ahead and run the following query:

```
mysql> SELECT DISTINCT NAME FROM bandmembers;
```

```
mysql> SELECT NAME FROM bandmembers;
+--------------------+
| Name               |
+--------------------+
| Rolling Stones     |
| Traveling Wilburys |
| Traveling Wilburys |
| Nirvana            |
| REM                |
| Crazy Horse        |
| Pixies             |
| The Band           |
| Widespread Panic   |
| Widespread Panic   |
| Widespread Panic   |
| Widespread Panic   |
| Pixies             |
| Jouney             |
| Jouney             |
+--------------------+
15 rows in set (0.00 sec)

mysql> SELECT DISTINCT NAME FROM bandmembers;
+--------------------+
| Name               |
+--------------------+
| Rolling Stones     |
| Traveling Wilburys |
| Nirvana            |
| REM                |
| Crazy Horse        |
| Pixies             |
| The Band           |
| Widespread Panic   |
| Jouney             |
+--------------------+
9 rows in set (0.00 sec)
```

9. Now change the previous SQL statement to include the LastName column. Run the query. How did the results change? Are there still distinct band name or did the band names repeat?

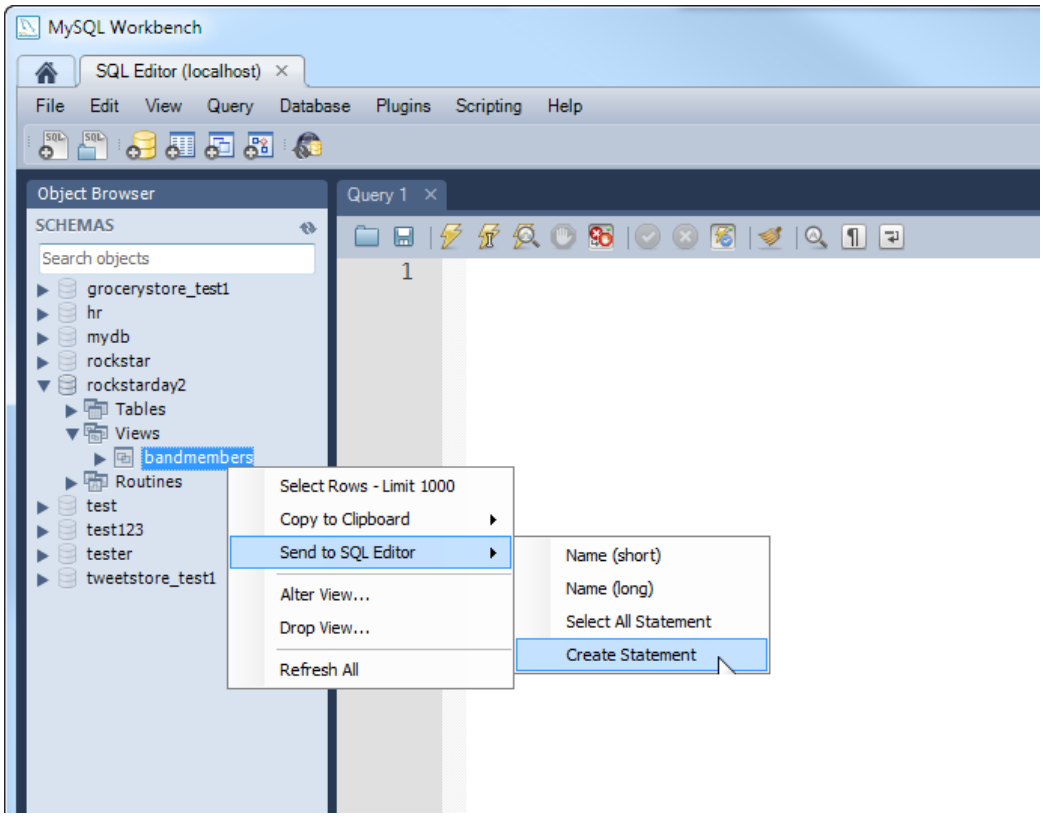
```
mysql> SELECT DISTINCT NAME, LastName FROM bandmembers;
```

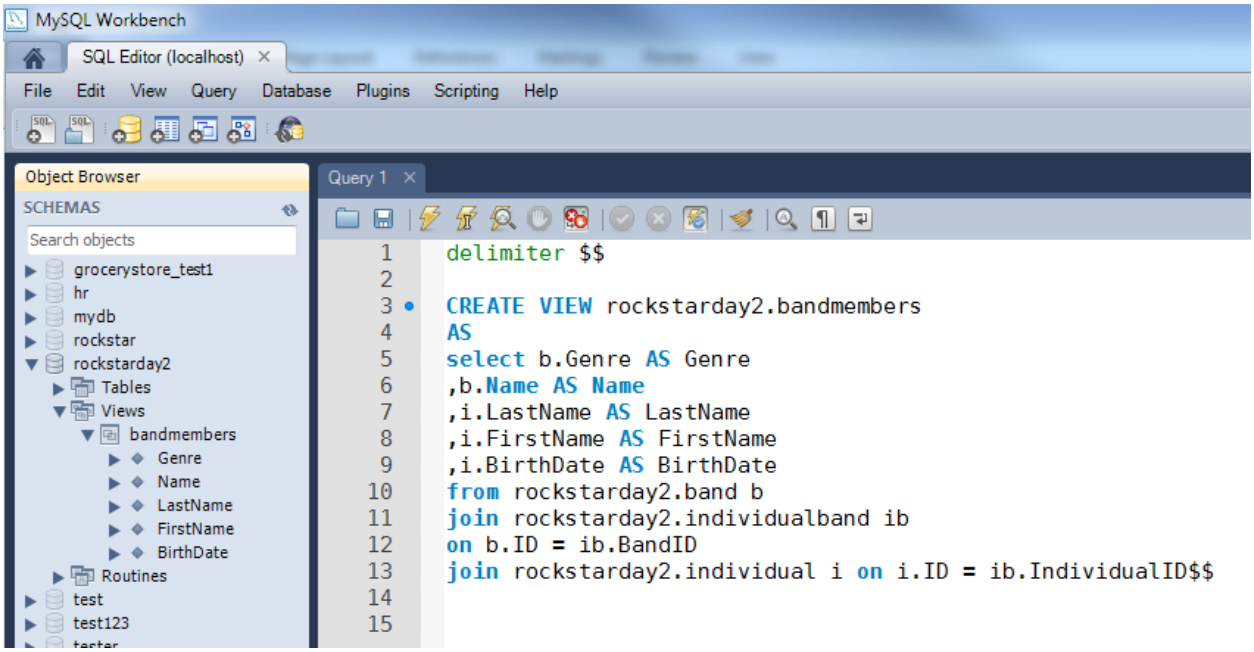
Let's take a quick detour and at some investigative techniques on how to explore the internals of a view such as the columns provided by the view and the SELECT statement that comprises the view.

10. Let's take a look at the CREATE VIEW statement that was used to create the **bandmembers** view.

```
mysql> SHOW CREATE VIEW bandmembers;
```

The output is a little hard to read in the terminal. Here's an easier view of the view using SQL Workbench: Don't worry about opening MySQL Workbench right now, just check out the screen shot below.

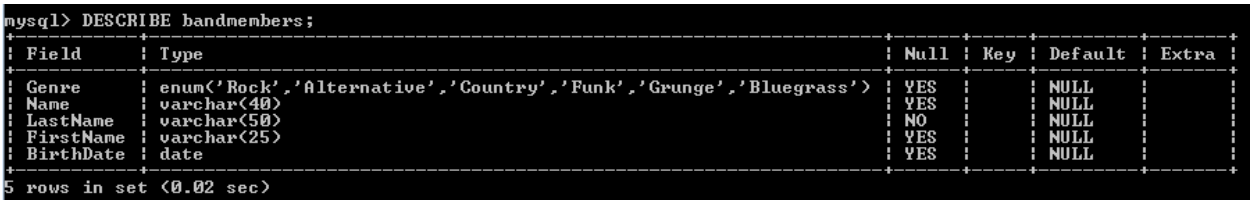




11. Using the **DESCRIBE** statement

Let's look at that view another way by using the **DESCRIBE** statement followed by the name of the view or table. Execute the following SQL:

```
mysql> DESCRIBE bandmembers;
```



```
mysql> DESCRIBE bandmembers;
+-----------+-----------------------------------------------------------------+------+-----+---------+-------+
| Field     | Type                                                            | Null | Key | Default | Extra |
+-----------+-----------------------------------------------------------------+------+-----+---------+-------+
| Genre     | enum('Rock','Alternative','Country','Funk','Grunge','Bluegrass') | YES  |     | NULL    |       |
| Name      | varchar(40)                                                     | YES  |     | NULL    |       |
| LastName  | varchar(50)                                                     | NO   |     | NULL    |       |
| FirstName | varchar(25)                                                     | YES  |     | NULL    |       |
| BirthDate | date                                                            | YES  |     | NULL    |       |
+-----------+-----------------------------------------------------------------+------+-----+---------+-------+
5 rows in set (0.02 sec)
```

The DESCRIBE and EXPLAIN statements are in fact synonymous.

12. Using the SHOW COLUMNS statement. SHOW COLUMNS also displays information about the columns in a table or views. Go ahead and try the following and compare to the DESCRIBE statement:

```
mysql> SHOW COLUMNS FROM bandmembers;
```

13. Now modify the SHOW COLUMNS statement my using the FULL keyword like this:

```
mysql> SHOW FULL COLUMNS FROM bandmembers;
```

IDENTIFYING COLUMNS

Database, table, index, column, and alias names are identifiers. You can qualify an identifier by using the period character (“.”). You can identify a column using any one of the following patterns:

Identifier pattern	Description
column_name	Use the column as identified by the column_name from the table or view.
table_name.column_name	Use the column_name from the specified table_name from the current database.
database_name.table_name.column_name	Use the column_name from the table_name of the provided database_name.

Column names are not case sensitive. It doesn’t matter whether it’s Windows, Unix, or Apple (OSx).

Table names and column names that contain special characters or share a reserved word will require quoting. The quote character is the backtick (“`”).

Example: `mysql> SELECT * FROM `delete`.id > 1000;`

NEXT STEPS: ON YOUR OWN TIME CHECK OUT THE FOLLOWING DOCUMENTATION FOR MORE INFO ON NAMING AND INDENTIFIER CASE SENSITIVITY:

HTTP://DEV.MYSQL.COM/DOC/REFMAN/4.1/EN/IDENTIFIERS.HTML

HTTP://DEV.MYSQL.COM/DOC/REFMAN/4.1/EN/IDENTIFIER-CASE-SENSITIVITY.HTML

TABLE AND COLUMN ALIASES

MySQL supports both column aliases and table aliases. Column aliases help make the output of a query more readable. You can use a column alias to provide a name that makes more sense. Use the **AS** keyword followed by the alias name for the column. The syntax goes something like this:

```
SELECT [column name | expression] AS 'easy to understand alias name' FROM table
```

EXERCISE: CREATING A SELECT STATEMENT WITH AN EXPRESSION THAT USES A COLUMN ALIAS.

1. Let's start with a simple query that retrieves every column from the bandmembers view.

```
mysql> select * from bandmembers;
```

2. Now let's alter the SELECT statement to combine the FirstName and LastName columns as a new column aliased as fullname. We'll use the CONCAT function to concatenate two columns together to form a single string. Try this:

```
mysql> SELECT lastname, firstname, CONCAT(firstname, ' ' , lastname) as fullname
         FROM bandmembers;
```

3. Now, let's use the YEAR() function to return the year for a bandmember's birthdate. We will alias the new column as 'birthyear'.

```
mysql> SELECT lastname, firstname, YEAR(birthdate) as birthyear
         FROM bandmembers;
```

USING MYSQL OPERATORS AND FUNCTIONS

Comparison operators will result in a value of True , False, or Null.

SELECT 5 <> 3; -- this will evaluate to True (1).

SELECT 5 = 3; -- this will evaluate to False (0).

Let's try some other comparison operators with the following exercises:

USING SOME BASIC COMPARISON OPERATORS

Comparison operators can work on numbers and strings. Below is a listing of some basic comparison operators:

- greater than (>)
- less than (<)
- equals to (=)
- not equal to (<>) (!=)
- greater than or equal to (>=)
- less than or equal to (<=)

EXERCISE: TRY SOME BASIC COMPARISON OPERATORS AND WATCH THE AUTOMATIC CONVERSION OF STRINGS TO NUMBERS

Try out the following SQL SELECT statements:

```
1. SELECT 'A' < 'B';
2. SELECT 'BA' < 'C';
```

Strings are automatically converted to numbers and vice versa as necessary.

```
3. SELECT '2' > 1;
4. SELECT 3 = (2+1);
5. SELECT '3' = (2+1);
```

USING THE IS NULL AND IS NOT NULL COMPARISON OPERATORS

IS NULL tests whether a value is null. It's really useful to discover columns are values that are NULL. If something is null then a 1 (true) is returned. For example, the following will return a value of 1:

```
mysql> SELECT NULL IS NULL;
```

You can use IS NULL in WHERE clause to determine a missing value. It will return the rows if the expression in the WHERE clause evaluates to true. For example:

```
mysql> SELECT * FROM Individual WHERE BIRTHDATE IS NULL;
```

You can IS NOT NULL to determine values that aren't NULL. For example:

```
mysql> SELECT * FROM Individual WHERE BirthDate IS NOT NULL;
```

USING THE BETWEEN COMPARISON OPERATOR

Use **BETWEEN** to evaluate the value of something to see if its greater than or equal to a minimum value AND less than or equal to a maximum value. If the value is **BETWEEN** then a 1 is returned, otherwise a 0 is returned.

The following expressions will evaluate to true:

```
mysql> SELECT 'y' BETWEEN 'x' and 'z';
mysql> SELECT 'ABC' BETWEEN 'AAA' and 'BBB';
mysql> SELECT 7 BETWEEN 7 and 10;
mysql> SELECT 1 BETWEEN .5 and 10;
```

You can use BETWEEN in the WHERE clause to restrict the rows returned where the expression in the where clause evaluates to true:

```
mysql> SELECT * FROM Individual WHERE LASTNAME BETWEEN 'A' AND 'Cobain' ORDER BY
LASTNAME;
```

As you can see **BETWEEN** is equivalent to the expression (minimum value <= expr AND expr <= maximum value)

EXERCISE: TRY OUT THE BETWEEN, IS NULL, AND IS NOT NULL COMPARISON OPERATORS

1. Create some SELECT statements that explore the Individual table. Each SELECT statement should pull the ID, FIRSTNAME, LASTNAME, BIRTHDATE, and DECEASEDDATE columns. Each SELECT statement should accomplish the following:
    a. Retrieve people who aren't dead.
    b. Retrieve people who are missing a birth date.
    c. Retrieve people who are NOT missing a birth date.
    d. Retrieve people who were born was in the 1940's. Hint: Use the YEAR() function.
    e. Retrieve people whose last name starts with the letters 'A' through 'D'. Hint: The last name 'Deal' is greater than 'D'.

USING THE IFNULL() AND COALESCE() FUNCTIONS

IFNULL takes two expressions and if the first expression is NOT NULL, it will return the first expression. If the first expression is NULL then it will return the second expression.

The following example will return the second expression:

```
mysql> SELECT IFNULL(NULL, 5);
+-----------------+
| IFNULL(NULL, 5) |
+-----------------+
|               5 |
+-----------------+
```

COALESCE will return the first non-null value in a list of values. It will return NULL if there are no non-null values.

```
mysql> SELECT COALESCE(NULL,'Rabbit');
+------------------------+
| COALESCE(NULL,'Rabbit') |
+------------------------+
| Rabbit                 |
+------------------------+
mysql> SELECT COALESCE(NULL,NULL, 5, 'Rabbit');
+---------------------------------+
| COALESCE(NULL,NULL, 5, 'Rabbit') |
+---------------------------------+
| 5                               |
+---------------------------------+
mysql> SELECT COALESCE(NULL, NULL, NULL, NULL);
+----------------------------------+
| COALESCE(NULL, NULL, NULL, NULL) |
+----------------------------------+
|                             NULL |
+----------------------------------+
```

EXERCISE: USING THE IFNULL() FUNCTION

1. Using the Individual table, create a query that returns three columns. The first two columns will be the ID and LASTNAME columns. In the third column returned, evaluate whether a person is alive. If they are alive, then return 'Alive' otherwise return the date they died.

Logical Operators
AND
NOT
OR
XOR

Control Flow Functions

CASE()
IF ()
ISNULL()
NULLIF ()

VARIABLES

i. Declaring variables
ii. Using Variables within a select statement

b.
c. Converting Data Types
d. CASE
e. Aggregate

7. Filtering Data
    f. Joins
        i. Outer Joins
        ii. Inner Joins
        iii. Other Types of Joins

    g. Subqueries
8. Grouping Data
    h. HAVING
9. Ordering Data
10. Table Aliases

CLASS 4: STORED ROUTINES

Stored procedures and functions can be referred together as stored routines. A stored routine contains SQL statements that are stored on the MySQL server and given a name. You can execute either on demand. Doing so causes the SQL statements defined within the stored routine to be run on the server.

With stored routines you as the database developer have access to programming constructs such as

Conditional statements and such as If-then

Loops

WHEN SHOULD YOU USE A STORED FUNCTION?

Use a function when you only need to return a discrete value, such as converting meters to inches or Centigrade to Fehrenheit. If you need to return multiple values or a set of data, use a stored procedure.

When to use a procedure?

Use to accomplish something without having to return a value.

What can stored functions do?

Stored functions return a result from a calculation.

Stored functions can be used within an expression

STORED ROUTINE VERSUS "REAL PROGRAMMING LANGUAGES" DEBATE RAGES

Pros

Code in one place; available to different types of client applications, written in different languages

Security

Performance

Encapsulate code and store a consistent set of code libraries in one place.

Error Handling

Cons

Performance – more work for the database to accomplish. code executed in a single place

You can create code classes in “real” programming languages with greater capabilities than the SQL language.

What if you need to switch database platforms? You will need to rewrite a major chunk of our application logic…ouch!

Note: Stored procedures and functions were added to the 5.0 MySQL release. You can determine the version of your MySQL server by executing the following SQL Statement:

```
mysql>  SELECT @@Version;
```

SYNTAX

To Create a Function you supply the following syntax:

```
CREATE FUNCTION db_name.sp_name
([param1 type][,param2 type][,param2 type][…]}
RETURNS Type
[characteristic ...]
routine_body
```

FUNCTION NAME

By default, a routine is associated with the default database. To associate the routine explicitly with a given database, specify the name as ***db_name.sp_name*** when you create it.

PARAMETERS

TO DO

RETURN TYPE

TO DO

CHARACTERISTICS

Several characteristics provide information about the nature of data use by the routine. In MySQL, these characteristics are advisory only. The server does not use them to constrain what kinds of statements a routine will be permitted to execute.

EXERCISE: BUILDING A SIMPLE DETERMINISTIC STORED FUNCTION

Let's create a simple function that will convert Celsius temperatures to Fahrenheit. While we are learning how to create a function, we will also learn how to denote it as either `DETERMINISTIC` or `NOT DETERMINISTIC`. We use either `DETERMINISTIC` or `NOT DETERMINISTIC` as a characteristic in the function definition. Place the characteristic after the return type. Use Deterministic if the function always produces the same result for the same input parameters, and "not deterministic" otherwise. If neither `DETERMINISTIC` nor `NOT DETERMINISTIC` is given in the routine definition, the default is `NOT DETERMINISTIC`. To declare that a function is deterministic, you must specify `DETERMINISTIC` explicitly.

Let's create a deterministic function named **`fnCelsiusToFahrenheit`** that contains a single parameter of type integer. The parameter represents a value for Celsius. The function is marked with the characteristic of **`DETERMINISTIC`** because it will return the same result in Fahrenheit for the same input parameter value in Celsius.

1. Within the terminal, use mysql to connect to the database server. We will be adding our stored routines to the database named RockStarDay2. Be sure to use the USE <databasename> statement to point to the correct database. At the mysql> prompt, enter

```
mysql> CREATE FUNCTION fnCelsiusToFahrenheit (celsius INT)
RETURNS INT DETERMINISTIC
RETURN (1.8 * celsius) + 32;
```

2. Let's invoke the function by using it within a **`SELECT`** statement. Go ahead and run the following statement at the `mysql>` prompt. Run it several times each time with a different value for Celsius.

```
mysql> SELECT fnCelsiusToFahrenheit(0);
```

3. Let's delete the function by using the **`DROP FUNCTION`** statement like this.

```
mysql> DROP FUNCTION fnCelsiusToFahrenheit;
```

4. Once you have dropped (deleted) the function, try executing the `SELECT fnCelsiusToFahrenheit(0);` again.

5. Now let's comment our function by describing what the function accomplishes. We can use the `COMMENT` characteristic to describe the stored routine. Recreate the function with an additional comment like this:

```
mysql> CREATE FUNCTION fnCelsiusToFahrenheit (celsius INT)
RETURNS INT
COMMENT 'Converts Celsius temperatures to Fahrenheit.'
DETERMINISTIC
RETURN (1.8 * celsius) + 32;
```

6. Now let's use the `SHOW CREATE FUNCTION` statement to display our comments.

```
mysql> SHOW CREATE FUNCTION fnCelsiusToFahrenheit;
```

7. Use the following logic as the basis for a new function named fnFahrenheitToCelcius. In the formula below, C is Celsius and F is Fahrenheit.

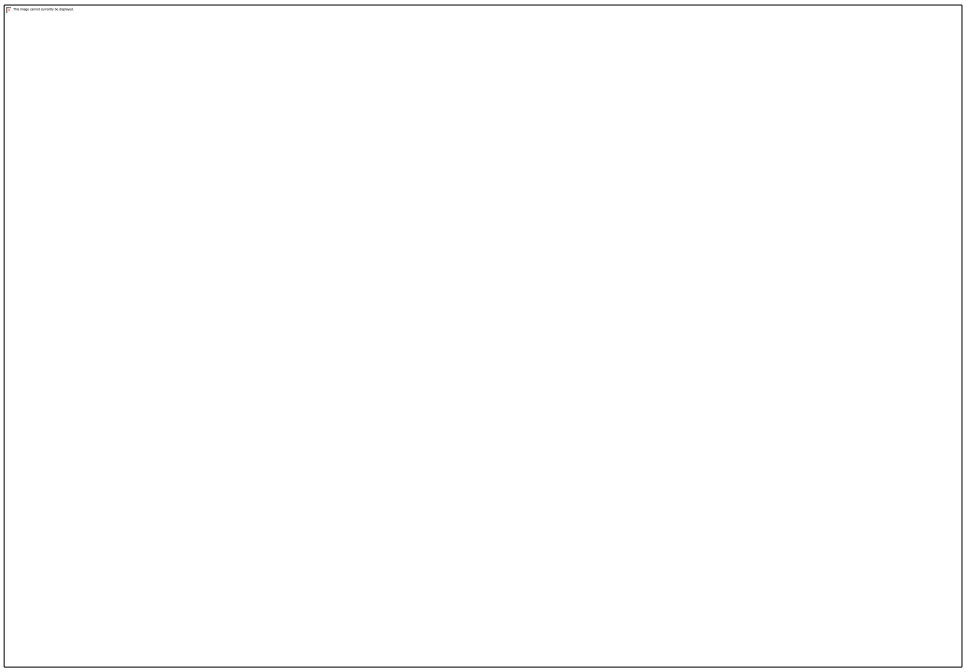
   C = 5/9 (*F*-32)

EXERCISE: CONNECTING TO SQL WORKBENCH

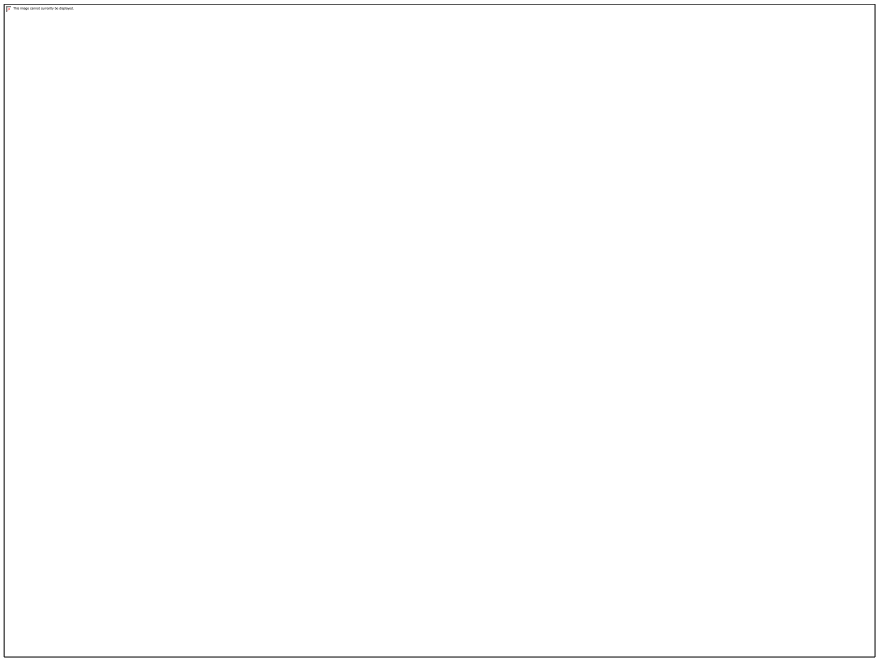
Our functions and procedures will become more and more sophisticated as we work through our exercises. It would be nice to have a sophisticated graphical user interface to help us manage the MySQL Server, create databases, manage connections to database servers, run SQL queries enter data, build procedures, create tables, create tables... Thankfully, SQL Workbench allows you do all these things and more.

You should have SQL Workbench installed on your local computer. Let's get started!

1. Let's start by launching SQL Workbench. Open the Applications folder in the Finder, then double-click MySQL Workbench. You should see the Home Window.



2. The first thing you will need to do is create a server instance. On the right hand side of the Home Window, select New Server Instance. You will see a Create New Server Instance Profile wizard. You will use the wizard to connect to a locally installed MySQL server on your mac. Select the localhost radio button:



3. Next you will set up a connection. Use the default values displayed below. although if your MySQL Server has a password set for the `root` account, you can enter it here by clicking Store in Vault. This enables you to connect to the server without needing to enter a password each time. It is also possible to use a different account to connect to the server by setting the user name and password here, if required.

USING THE IF STATEMENT

```
The IF statement allows you to conditionally branch your execution logic.  IF allows you
to say

IF <some condition> THEN <Perform Some Statements>

END IF


You can also add an ELSEIF statement to create another branch in your logic such as "IF
something is true THEN do A  ELSEIF something is true THEN do B.
```

EXERCISE: USING IF,THEN, ELSEIF, ELSE AND END IF

Let's create another weather related function to complement our original `fnCelsiusToFahrenheit` function we created earlier. Our new function will be called fnHotWarmCold. It will accept a single parameter of data type INT which represents the temperature in Celsius. The function will utilize our `fnCelsiusToFahrenheit` function and determine the temperature in Fahrenheit. The function will use and IF statement to evaluate the converted temperature. IF the temperature is less than 60 degrees then return "Cold". IF the temperature is greater or equal to 60 but less than 76 then return "Warm". IF the temperature is equal to or greater than 76 then return "Hot". The return values will be of data type Char(5).

Within SQL Workbench, select the RockStarDay2 database in the list of database on the left hand side of the window. Right click the database and select 'Set as Default Schema' from the popup context menu.

Within the database, select and right click 'Routines' followed by clicking on 'Create Routine…' from the popup context menu. This will open a new window. Within the window, you will see a stubbed out new routine. How nice!

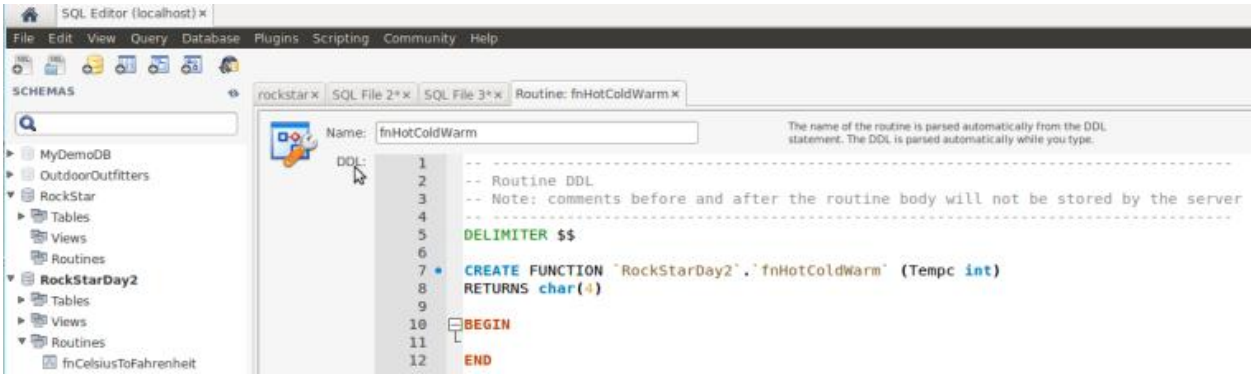
Notice the DELIMETER statement. We need a way to redefine how we end the SQL script. We normally use a semi colon to end a statement. But, we will be using several semi columns within the body of the script. So, we need a way to tell mysql that we are through defining the script. For this we use the DELIMETER statement. The system suggests using $$. We will just roll with it.

Now, let's name the routine. Replace 'new_routine' with 'fnHotWarmCold'.

Within the parenthesis, let's add the parameter named TempC. The datatype should be 'int'.

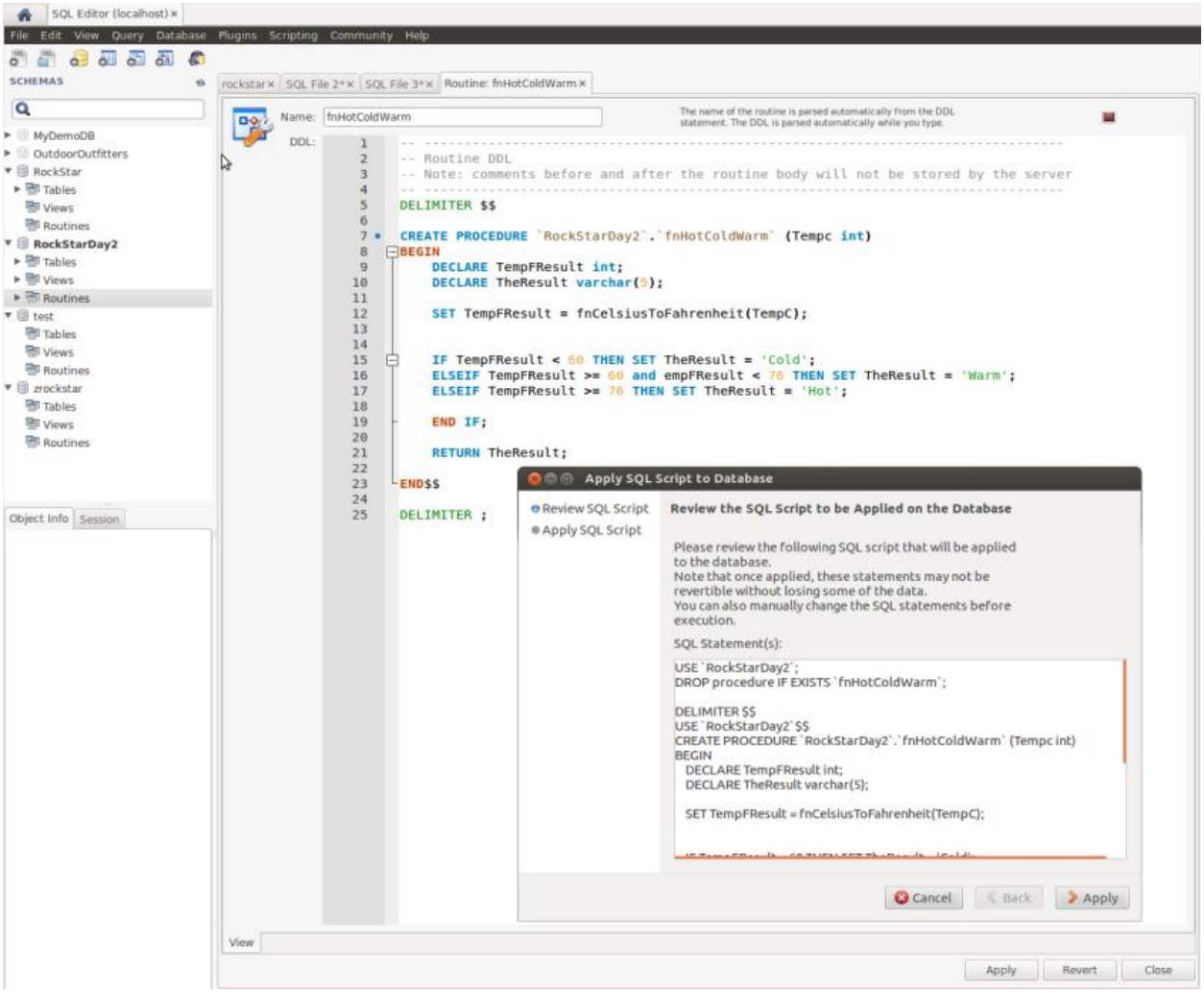
Next, we need to change the CREATE PROCEDURE statement to a CREATE FUNCTION statement. Go ahead and do that now. At this point your script should look like this:

Next, we need to declare the return type for the function. Since the function is going to return either 'Hot', 'Warm', or 'Cold', we need to return a char(4).



< place the script from Dropbox here and explain it step by step>

After you have finished hacking out the sql script, select the Apply button. An 'Apply SQL Script to Database' confirmation screen will appear. Select the 'Apply >' button to confirm the running of the script to create your new function. The screen shot below is from my home laptop running Ubuntu Linux. You script and confirmation screen should look something like this:



EXERCISE: USING A CONTROL FLOW FUNCTION: CASE

EXERCISE: USING A CONTROL FLOW FUNCTION: IFNULL

Stored Functions versus Stored Procedures

Stored Function	Stored Procedure
Calculates a single value	Can be used to calculate a single value, return a set of data, return multiple sets of data, return multiple values, perform an action and return no value.
Used within expressions such as SELECT fnCalculateAgeInDays('2001-12-12');	Not used in expressions. Executed as standalone operations using CALL statement.
Use the CREATE FUNCTION statement to create a function	Use the CREATE PROCEDURE statement to create a procedure
Uses a RETURNS clause to indicate the data type to return.	A stored procedure doesn't have a RETURNS clause.
Must include a RETURN statement to return a value.	Instead it can return the result as a row or rows of data. Or, it can return a result by assigning the value using an OUT parameter.
A stored function belongs to a database.	A stored procedure belongs to a database.
A stored function does not have an IN parameter type.	A stored procedure uses an IN parameter to accept in a parameter value from the caller. The caller cannot see any changes to the parameter value.
A stored function does not have an OUT parameter type.	With an OUT parameter, the stored procedure sets the value of the parameter. The parameter value can be utilized by the caller of the procedure.
A stored function does not have an INOUT parameter type.	With an INOUT parameter the caller can pass in a value into the procedure. The caller can receive a value from the procedure.

EXERCISE: CREATE A SIMPLE STORED PROCEDURE WITH AN IN PARAMETER

In this exercise we will create our first stored procedure that accepts a single parameter value into the stored procedure which returns a row set of data. The procedure will accept in a band name parameter and turn all bands that *start* with the band name parameter.

EXERCISE: CREATE A SIMPLE STORED PROCEDURE WITH AN IN AND OUT PARAMETERS

In this exercise, we will create a stored procedure named uspFahrenheitToCelcius that accepts IN a value for temperature in Fahrenheit and return the value using an OUT parameter in Celsius. Utilize the appropriate function created earlier to perform the heart of the calculation.

EXERCISE: REWRITE USING INOUT PARAMETER

In this exercise, you will rewrite the previous