

INTRODUCTION TO RELATIONAL DATABASES WITH MYSQL

CODE CAMP CHARLESTON

TRIP OTTINGER

CONTENTS

DAY 1:	4
Connecting to mysql with the command prompt/terminal	4
Exercise 1: Connecting to MySQL	4
Exercise 2: Simple SQL Commands	6
Exercise 3: Creating and Deleting Databases	6
Relational Data Modeling	8
Creating Tables and data types	10
A relation is a set of tuples	11
Primary Key	12
Normalization	12
Introducing Data Types	12
String Data Type	12
Data Types and Performance	13
Exercise 4: Creating Tables	13
NOT NULL	14
Date and Time Data Types	14
Exercise 5: Creating Tables with date columns	14
INSERT Statements Add Rows to a Table	16
Exercise 6: DESCRIBE lists a Table's Structure	16
Modifying a Table's Schema	18
Exercise 7: ALTER TABLE	18
Exercise 8: Adding a Column as the Primary Key and using the INT data Type	18
Number Data Type	19
YEAR Data Type	19
ENUM Data Type	20
BOOL, BOOLEAN, TINYINT(1) Data Types for True/False	20
Exercise 9: Using the Year, Enum, and Boolean data types	20
Foreign Keys: Relating Tables Together	21

Modeling Foreign Key Relationships 22

 Exercise 10: Create an Album table. Relate the Album table to the Band table using a Foreign Key..... 22

Many to Many Relationships 24

 Exercise 11: Building a cross reference table to form the many to many relationship 29

DAY 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 1: 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 terminal 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

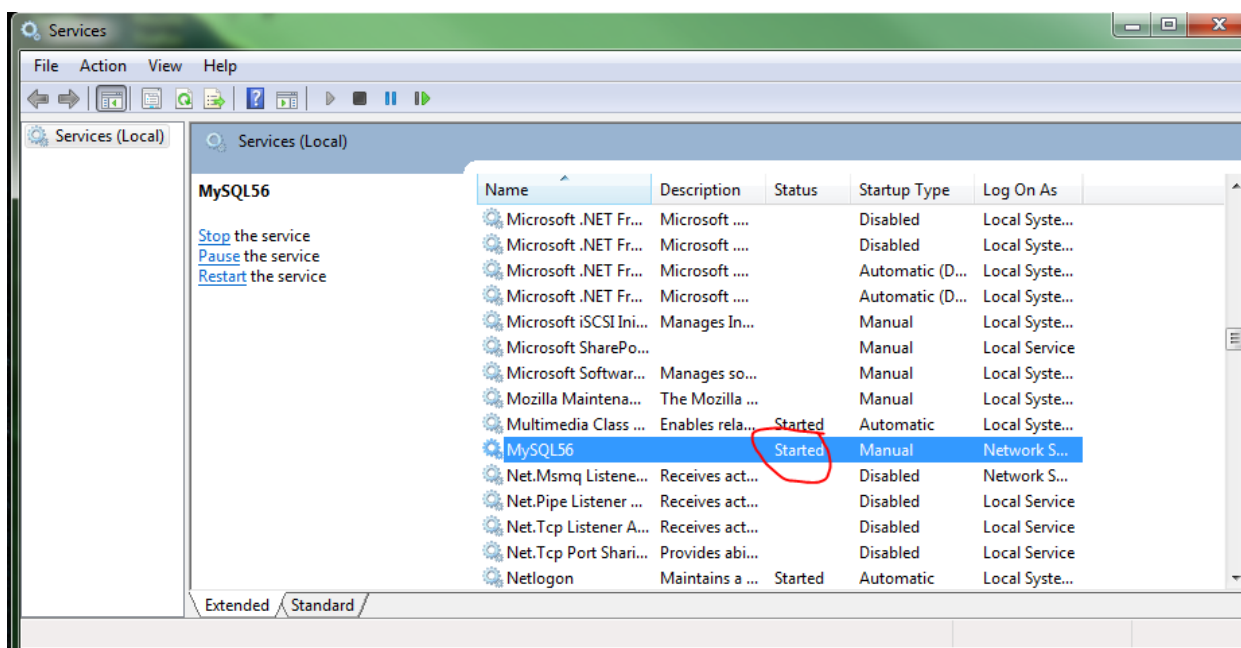
```
mysql>
```

- 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.



TIP: TO START, STOP, RESTART THE MYSQL SERVER ON A MAC YOU CAN TRY ONE OF THE FOLLOWING:

Start

```
$ mysql.server start
```

OR you can Invoke mysqld directly. This works on any platform.

```
$ mysqld
```

Stop

```
$ mysql.server stop
```

Restart

```
$ mysql.server restart
```

For more information see [Starting and Stopping MySQL Automatically](#) and [Restart, Start, Stop MySQL from the Command Line Terminal, OSX, Linux](#)

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](http://www.youtube.com/watch?v=FTJOIN_OADC)

EXERCISE 2: 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.

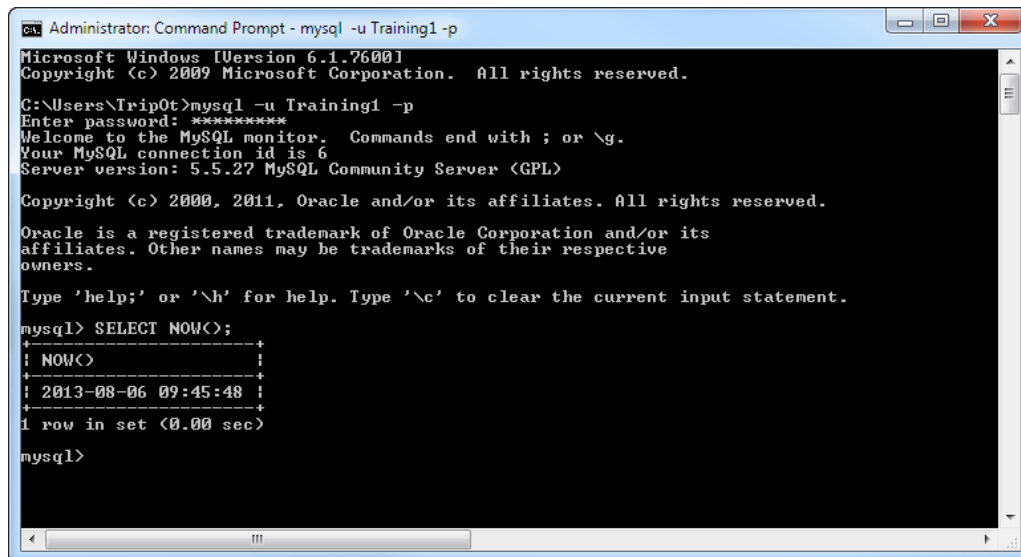
```
$ mysql -p -u user_name
```

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:



```
Administrator: Command Prompt - mysql -u Training1 -p
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Trip0t>mysql -u Training1 -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 6
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> SELECT NOW();
+-----+
| NOW() |
+-----+
| 2013-08-06 09:45:48 |
+-----+
1 row in set (0.00 sec)

mysql>
```

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

<END OF EXERCISE>

EXERCISE 3: 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 can't 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

RELATIONAL DATA MODELING

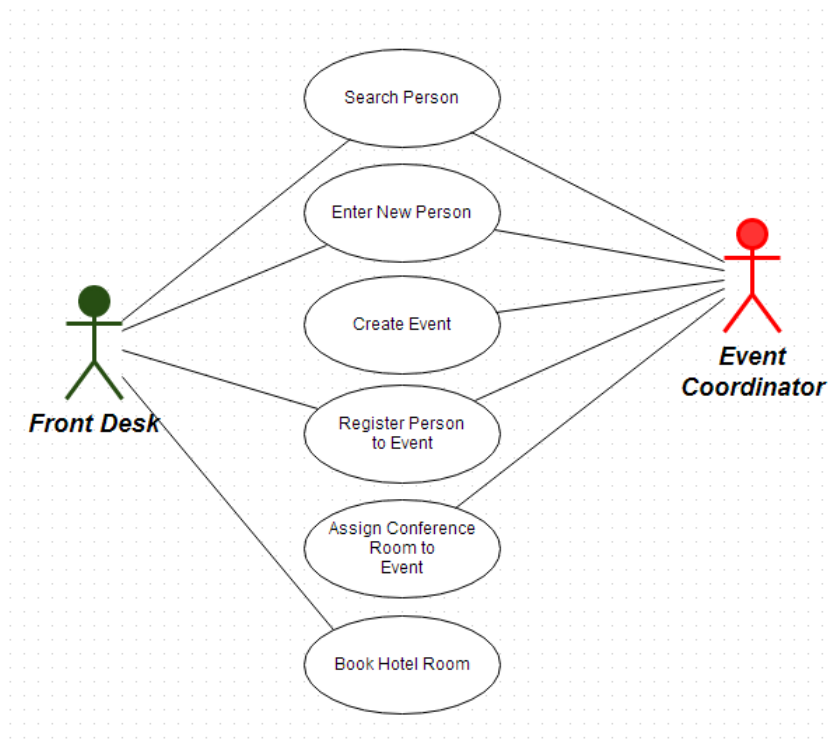
"A data model is a precise description of the data stored for a real-world problem.... The data model is not a complete nor exact description of a real situation... it will always be based on definitions and assumptions" (Churcher, 2007)

When designing a data model, you should devote time to define the tables in your system. Ensure the tables and their relationships reflect the reality of your problem. Define the definitions of your tables. Define the assumptions. Assumptions and requirements will change over time, especially if your project is successful. After all, software that does not change is not being used. A data model that accurately reflects reality will be resilient to change. "To develop software of lasting quality, you have to craft a solid architectural foundation that's resilient to change." (Churcher, 2007)

Define the requirements of what the system is going to accomplish. You may be trying to manage people attending an event, inventory for maintaining airplanes, or statistical results for a manufacturing process. I always start by getting the main stakeholders in a room with a laptop and a projector. Together we form several paragraphs that specifically define the problem. This helps to shape the problem we are going to solve and the problems we are not going to solve. I also begin listing a series of needs and wants from the system. I also find it helpful to begin ranking the needs and wants. This helps to ease the pain when time and resources are not available to fulfill all the potential of the project.

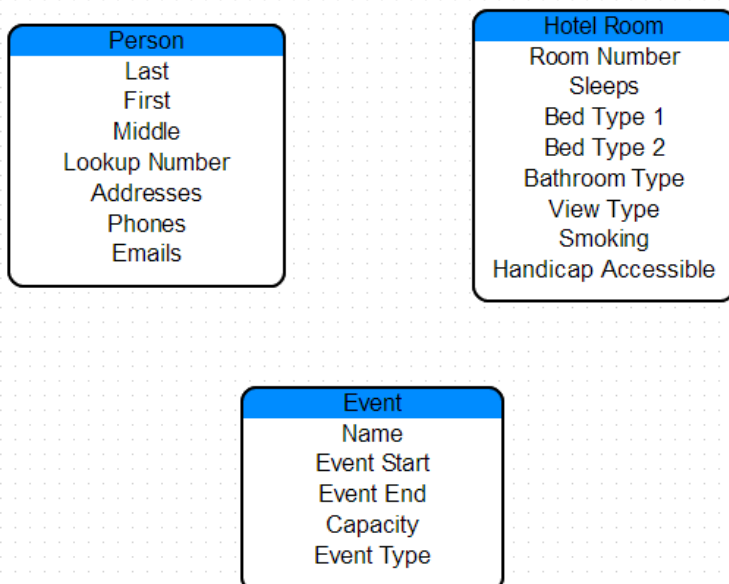
As the problem that you are trying to solve becomes clearer, you will want to move onto a more formal method of defining the interactions between the users and the software. UML is a popular choice for defining the requirements and designing the overall system. When it comes time to defining the system at a very high level, UML can be used to create use cases. Use cases track the various types of users (actors) that interact with the system. While this course doesn't cover requirements gathering or UML, a couple of good books for UML are [Writing Effective Use Cases](#) by Alistair Cockburn, Unified Modeling Language User Guide by Grady Booch, James Rumbaugh, and Ivar Jacobson, and UML Distilled: A Brief Guide to the Standard Object Modeling Language by Martin Fowler.

Below is an example of a use case diagram using free on-line software www.draw.io. The use case diagram is simple by design. You begin by identifying the people and other things that interact with the software. These are called actors and are represented by the stick figures. Each ellipsis represents an individual use case and depicts the objective or goal the user (actor) wants to achieve with the system. If you plan on using a use case diagram, keep it simple and don't get bogged into the various meanings of the graphics and relationships types. Think of it as a cave painting. Use cases begin with a verb. The use case diagram is meant to show a high level listing of all the objectives/use cases of the system. It displays the big picture. That's it.



The next step is for each ellipsis in the Use Case Diagram is to write a good text-based use case. A use case outlines the detailed steps to achieve the desired outcome or goal. For example, for the Enter New Person use case, the person must be logged in. The system must provide a data entry screen for entering the last name, first name, address, email, and phone information, etc. You will also want to describe the alternative paths that could occur. Think about the contingency plan. List the steps that should occur for the actor when things go wrong. Using these steps, write a user story using concise, easy to understand language

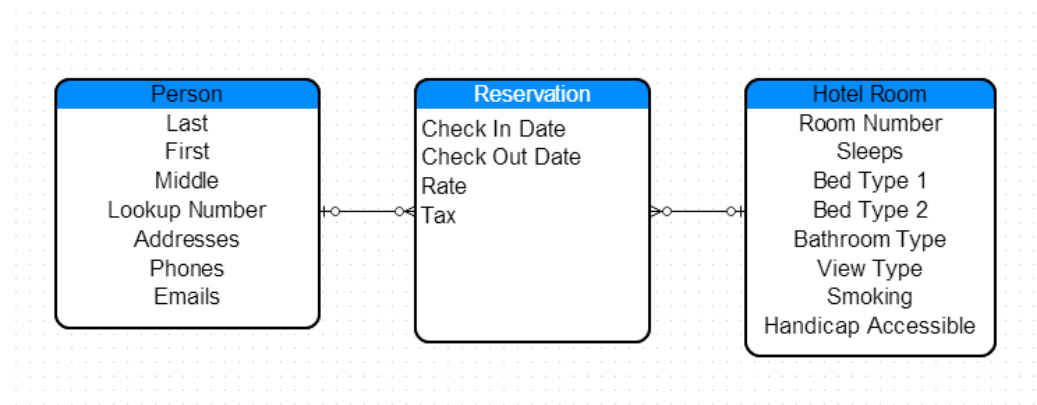
During the process of defining the use cases, begin tracking the names of the things (logical entities) you are tracking. For example, if you are building an event management system you start with a logical model containing a person, an event that people attend, and the person's hotel room. You should also begin mapping the attributes for each entity. For the person entity this would be their name, address, phone, etc. There are different ways and notations that you could use to model your system. For simplicities sake, you could simply sketch a box with the name of the entity:



After you have established some basic entities and attributes, you should begin thinking through the relationships between the entities. Let's take Person and Hotel Room. We know that a person can 'Book' a hotel room. But we need more detail. In real life can a person book more than one room? Does a person have to book a room? Let's for argument sake state the assumption that a person can book more than one room. We will also not require that person stay at our hotel to attend the event so a person doesn't have to book a room.

Let's look at the nature of the relationship between Person and Room from the Room's point of view. Does a room have to be occupied by a Person or can it be vacant? Over time can a room be booked by more than one person? We know that it is very likely that a room will not be booked all the time, we can have a room without it being booked. We also know that over time a room will be booked by different people. So a person can book may rooms and a room can be booked by people over time.

When a room is booked, what do we want to track in the booking/reservation? The reservation dates? The rate? The lodging tax? Which entity do we want to use to track the attributes for dates, rate, and tax? Do these attributes belong with the Person entity or the Hotel Room entity or an entirely new table? It seems we are uncovering a new entity: Reservation. It is within the Reservation entity that attributes such as the person booking, dates, rate, and tax should be kept.



The a logical model will ultimately be translated into a physical model. The physical model will contain the design for the database tables, data types, relationships, indexes, constraints, etc.

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 its attributes which translate to columns in the table. Each column contains 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 RDBMS 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 first 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.

DATA TYPES AND PERFORMANCE

- Choosing the correct type to store your data is crucial to getting good performance (Schwartz, Zaitsev, & Tkachenko, 2012)
- Use the smallest data type that can correctly store and represent your data
- Choose the smallest one that you don't think you'll exceed
- Smaller data types are usually faster, because they use less space on the disk, in memory, and in the CPU cache .
- Fewer CPU cycles to process

EXERCISE 4: 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 5: 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 `INSERTing` 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 6: 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>
```

- 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...'

- 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>
```

- 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 7: 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 8: 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 `INSERT`ing 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/](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 9: 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>

FOREIGN KEYS: RELATING TABLES TOGETHER

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).

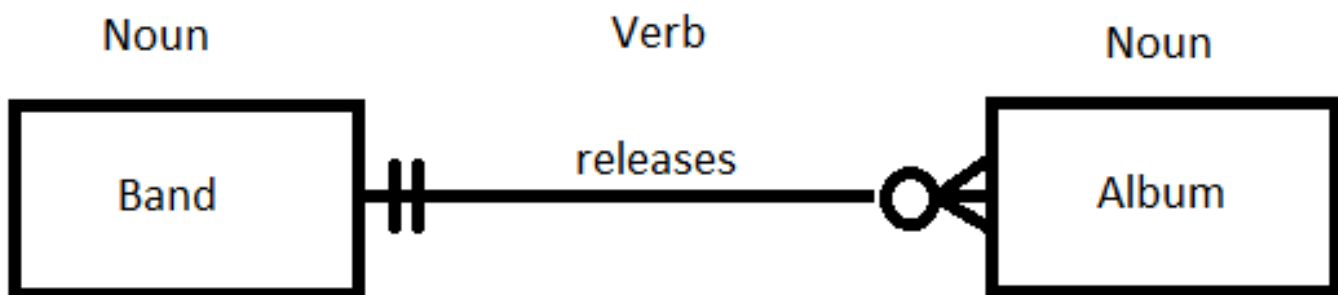


Figure: Another One-to-Many relationship

EXERCISE 10: 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. Hmmm, 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 musica	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.

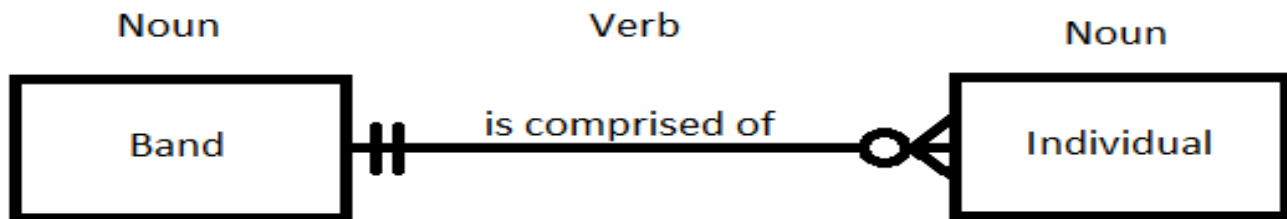


Figure: One-to Many ERD

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**.

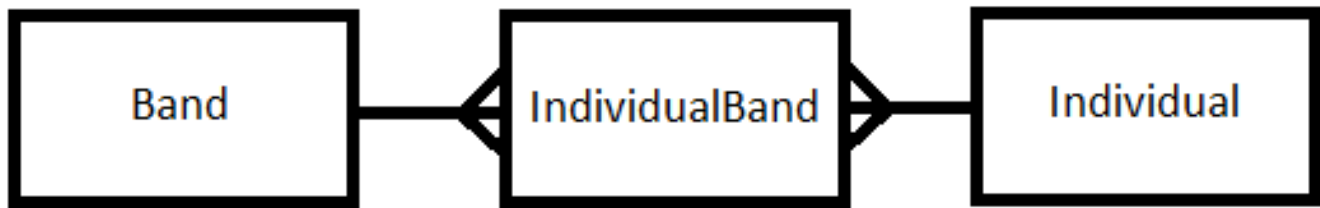


Figure: A junction or cross-reference table helps implement the Many-to-Many relationship between Band and Individual

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
	1	Mick	Jagger	England	7/26/1943	Dartford
	2	George	Harrison	England	2/25/1943	Liverpool
	3	Neil	Young	Canada	11/12/1945	Toronto
	4	Helm	Levon	United States	5/26/1940	Elaine, Arkansas
	5	Ringo	Starr	England	7/7/1940	Liverpool
	6	Ronnie	Wood	England	6/1/1947	Hillingdon
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	
IndividualBand Table						
	ID	IndividualID	BandID			
	100	1	1234			
	200	2	1111			
	300	3	2222			
	400	3	3333			
	500	4	1222			

EXERCISE 11: 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** are 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	Cascade 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>

