

4.5: MySQL CRUD Actions

There are four basic actions that cover how we interact with the data and structures in our database. We can either create new data, read existing data, update something already in place, or delete it. These four actions are collectively referred to as CRUD, for Create Read Update and Delete, and represent the basic concepts behind data interaction. In MySQL, we will address these concepts through a collection of commands.

Opening SQL

From a Browser

If you are using an installation of WAMP, you will have access to PhpMyAdmin from your browser. This program includes a SQL tab where you can input commands to your server as well as use a graphic interface to interact with your databases. Typically you can get there by going to <http://localhost/phpmyadmin> or <http://127.0.0.1/phpmyadmin> (do not forget to add the port number if you changed yours from :80). Your credentials will be the same as described in the next section, and you will find a SQL tab that allows you to enter commands to follow along with our examples.

From a Command Prompt

Each MySQL installer includes a client access program to interact with your server. Due to the variety of operating systems and versions of MySQL, definitive directions for accessing your client program cannot be provided here, but can be found by searching online with the details of your particular system. In general, your programs list may contain a MySQL folder with a link to launch the client, or you can access it by using your system's terminal window. Once you have a terminal window open, you can log into MySQL by typing:

1. `mysql -u root -p`

This instructs your computer to start MySQL using the username (represented by `-u`) of root and asking it to prompt you for a password, represented by `-p`. Once you press enter, you will be prompted to enter the password. By default, MySQL uses root as the username. Your password could be root, password, or an empty password (type nothing at all) depending on your version. If you changed these values when you installed, use what you provided in place of this example.

Once logged in to your database, you can begin to take a look around to see what your server already contains. To see what is currently in your server, you can enter a simple command to ask for that list. In MySQL we need to end our statements with semi-colons just as we do with PHP. Our commands and interactions are structured to be sentence-like, which makes queries easier to understand. Let us take a look at what we have available:

1. Show databases;

In a new installation of MySQL, you should see a list similar to the following:

```
+-----+
|      Database      |
+-----+
| information_schema |
| mysql              |
+-----+
```

1. 2 rows in set (0.00 sec)

Now that we have designed our database, it is time to describe it to the server, and get some data into it, so we have a fully functioning system. First we need to get into MySQL:

1. create database music;

After pressing enter you should see a response from the server that gives either a completed message or an error message, along with other information such as the amount of time it took to complete the instruction. If you receive an error that the database already exists, you can use a different name. If something happened and you are starting over, or know you can get rid of the existing database, you can get rid of it and all its data by typing:

1. drop database music;

Keep in mind this command deletes everything—the database and all tables and data stored in it. There is no “undo” option or garbage bin to recover from, so if you do not have a backup you will not be able to recover from this action. Once you have a database created, tell the server that is the one you want to interact with by typing:

1. use music;

You will notice we consistently use semi-colons after each instruction. Just like in PHP, this is how the server can tell where one instruction ends and the next begins.

Create

The first step in building our database is to create the tables from our design. Before we do so, we need to create the database we want to put those tables in. Be sure you have a music database created as we saw above. We will keep using the database we just designed. The create command for a table involves specifying that we mean to create a table (as opposed to a database), defining our table name, and then in parenthesis defining each column. This done with a comma separated collection of values that includes the name, type, null option, and other features of the column. Here we will look at the command necessary to create our first table:

| | | | | | |
|-----------------------------------------|---------------------------------------------------|------------------------------------------------------------------------------|-----------------------------------------------------|------------------------------------------------------|----------------------------------------------------------------|
| Bands bandID bandName | | Albums albumID albumName releaseDate bandID producerID | | Songs songID title length albumID | Concerts2Artists id artistID concertID |
| Labels producerID producer | Locations locID city state zip | Artists artistID artistName locationID | Venues venueID venueName locationID | Concerts concertID venueID date | Bands2Labels id producerID bandID timestamp |

1. Create table bands (bandID int not null auto_increment primary key,
2. bandName varchar (40) not null
3.);

You will notice a few things about this statement. First, we only use commas to separate the entire definition of one column from another as opposed to separating each piece of information about a column. If you are using a command line interface, you can also use the enter key to format your statement as we did above, with one column on each line. This is due to the fact that the statement will not execute until the semicolon is present.

After we identified our first column as bandID, we identified it as an integer, referencing the data types we looked at earlier. We also stated that this column cannot be null, meaning it must have a value. We can apply this on any column that must be present in our table for a record to be considered useable. In this case, we want bandID to be entered for us by the database. Adding the auto_increment attribute will tell the database to assign each new record a value (we can also control the starting value if we ever wanted to start at something other than 1). Finally, we label this column as the primary key of our table, place a comma at the end of our definition, and move to the next column.

When we add our bandName to the table definition we needed to define far less. This time, we went with varchar (variable character) which means we expect the content to be text, but not necessarily long text like sentences or paragraphs. When we use varchar, we have to tell MySQL what the maximum number of characters is allowed in the field, so it knows how much space to reserve for each record. In this example, we have decided that our longest band name would be at most 40 characters. To complete this column we again specify that it cannot be null. In our example, it is the only field in the table outside of the id anyway. If we

wanted to allow a null value, we could include the word null in place of not null, or just drop that piece altogether (MySQL will assume null is valid unless told otherwise).

After we execute this statement, we can use the command “show tables” to see what is in our database:

```
1. mysql> show tables;
```

| bandID | bandName |
|--------|--------------|
| 1 | The Who |
| 2 | Moxy Fruvous |
| 3 | The Doors |
| 4 | Maroon 5 |

```
1. 1 row in set (0.00 sec)
```

To make sure everything was created as we intended, we can look more closely at the structure of the table we created with the command “show columns from”:

```
1. mysql> show columns from bands;
```

| Field | Type | Null | Key | Default | Extra |
|----------|-------------|------|-----|---------|----------------|
| bandID | int(11) | NO | PRI | NULL | auto_increment |
| bandName | varchar(40) | NO | | NULL | |

```
1. 2 rows in set (0.02 sec)
```

Here we can see the structure of what we just created. Neither field can be null, bandID is our primary key, will auto increment, and neither field has a default value. If we want to assign a default value, we would include the word default followed by the value in quotation marks when defining our column.

We will create one more table here as a second example, and then you can continue creating the rest on your own in order to practice. Now we should create the Albums table so we can see a data field in use, which will cover all of the data types we will need in this database. Keep in mind that when we create tables with foreign keys in our example here, we are not going to define them as such at the database layer like we did when we defined our primary key. This is an available feature however, and allows MySQL to help us maintain data integrity by giving us an error if we try to insert a record where a foreign key value does not exist. As an example, if we tried to create a concert record but our reference to artist #5 did not exist in the artist table, MySQL would return an error instead of allowing the record to be created. You would have to create artist #5 first, then go back and try your previous statement again. Since this complicates the order of table creation and data insertion, we will ignore it for now until you are more comfortable, but know it is available and useful for production systems.

1. Create table albums(albumID int not null auto_increment primary key,
2. albumName varchar(70) not null,
3. releaseDate date,
4. bandID int not null,
5. producerID int
6.);

Once we have created all of our tables, we will need to put some data in them so we have something to interact with. To do this, we use the insert command. Using insert involves specifying the table we want to interact with, passing the list of fields we intend to fill, and then passing the values for those fields. The fields, as well as each set (or record) of values are contained in a comma separated list enclosed in parenthesis. Multiple records can be added at once, assuming each record is using the same set of fields, by adding another set of data in parenthesis. We can see this in action by creating our first band:

1. Insert into bands(bandName) values ("The Who");

If we want to pass more than one record, we just keep tagging on more sets:

1. Insert into bands(bandName) values ("Moxy Fruvous"), ("The Doors"), ("Maroon 5");

Take note of the fact that we do *not* specify the bandID for these records. Before we can insert albums though, we need to know what each band's ID actually is. We will take a quick preview of the Read actions by using the following command to get that information:

1. Select * from bands;

This command should give you something like the following:

1. mysql> Select * from bands;

| +-----+-----+ | |
|---------------|--------------|
| bandID | bandName |
| +-----+-----+ | |
| 1 | The Who |
| 2 | Moxy Fruvous |
| 3 | The Doors |
| 4 | Maroon 5 |
| +-----+-----+ | |

1. 4 rows in set (0.00 sec)

Now we can use these values to try an insert that uses multiple columns. Keep in mind that if you try to insert a record without including a required field you will still get an error even if you do not include it in the fields you wish to pass! We also need to format our date to meet what MySQL expect, or need to use a MySQL function to convert it to something valid. Here we will format it ourselves. The default format is YYYY-MM-DD meaning four digit year, two digit month, two digit day, all separated by dashes:

1. Insert into albums(albumName, bandID, releaseDate) values ("Tommy", 1, "1969-05-23"), ("Bargainville", 2, "1993-07-20"), ("Full Circle", 3, "1972-07-17");

As you can see in this example, complex values like strings and dates need to be wrapped in quotation marks so MySQL knows where they start and end, but we can leave basics like integers as they are.

Read

Now that we have some sample data, we will look at some basic techniques to see what we have. This is done with use of the select command, which comes with an assortment of filters and qualifiers. This is where the power of an SQL database comes into play as we manipulate, combine, and alter the information into what we want to see. We already saw one example of this when we needed

to reference our bands table. The star that we used in “select * from bands” is a reserved character in MySQL that represents “all.” What we effectively asked was “select everything in the bands table.” We can drop the fields we do not want to see by specifying only the ones we want. For example, we can take a look at our albums table, but since we did not include producers yet and do not care about the record ID, we will just ask for certain columns:

1. Select albumName, bandID, releaseDate from albums;

| albumName | bandID | releaseDate |
|--------------|--------|-------------|
| Tommy | 1 | 1969-05-23 |
| Bargainville | 2 | 1993-07-20 |
| Full Circle | 3 | 1972-07-17 |

This gives us a more readable response. We will learn how to get the actual bandName soon, for now we will focus on how to change what we ask for. If we wanted to make this output more end-user friendly, we probably do not want to use the field names stored in the database. We can mask those by giving them an alias. We will also add some sorting in this example by applying the ascending sort (asc) in an order by clause to the album name (descending would be desc):

1. Select albumName as "Album", bandID as "Band", releaseDate as "Release Date" from albums order by albumName asc;

| Album | Band | Release Date |
|--------------|------|--------------|
| Bargainville | 2 | 1993-07-20 |
| Full Circle | 3 | 1972-07-17 |
| Tommy | 1 | 1969-05-23 |

We can also search for partial matches of text. Here we will use the “like” and “where” reserved words to further specify exactly what we want to see. You are probably noticing that most of our statements so far have been relatively human readable, meaning you can understand what is being done, just by reading the code. This is an intentional approach in structured query design as it makes it easier to design and debug more complex queries. Commas are used where more than one item is specified, however there is no comma after the last item in a list. You can see this where we do not include a comma after release date before moving on to specify the table we want in our “from” clause. You will also notice the use of % in the next example. This is another reserved character in MySQL, which represents a wild card, meaning anything found in that position is valid. In our example we will be searching for the word bargain, and because it is flanked on both sides by a % it will be considered a match anywhere in a string. If we only want things that start with bargain, we would only use the % after it.

1. Select albumName as "Album", bandID as "Band", releaseDate as "Release Date" from albums where albumName like "%bargain%" order by albumName asc;

| Album | Band | Release Date |
|--------------|------|--------------|
| Bargainville | 2 | 1993-07-20 |

You may have noticed that our search string had bargain lowercase, but MySQL still returned Bargainville even though it is capitalized. This case-insensitive search is the default on MySQL, but you can specify particular form of upper or lowercase if you want. If we want to match only a specific string, we can use = in place of like and %. In fact, we can use many of the operators we are already accustomed to when numbers are involved, such as greater than and less than.

We should add a few more records to see some more of the options we have at our disposal when selecting data:

1. Insert into albums(albumName, bandID, releaseDate) values ("Strange Days", 3, "1967-10-16"), ("Live Noise", 2, "1998-05-19");

Now we have a couple bands with more than one album. We can actually infer more data than we are storing in the database by using MySQL functions to manipulate the data and results in real time. Try using the count() function to find out how many albums we have for each artist. We need to specify what we want to count (in this case, records, so we can just use *) and what piece of the record we want to group to be counted, in this case the bandID:

1. Select bandID, count(*) as "Albums" from albums group by bandID;

| bandID | Albums |
|--------|--------|
| 1 | 1 |
| 2 | 2 |
| 3 | 2 |

We are not actually storing the total number of albums in any of our tables—MySQL is tracking the total number of each time a bandID occurs from the group by statement.

Maybe we want to answer a question, like which album was released most recently? If this were an excel document, we would just sort the table by the release date and look at the first record. We can do the same thing in MySQL by adding a limit in the number of records we want back. While we could certainly take the whole table result and read just the first row, when your data set gets larger you do not want to send your user more data than they want or need, because it is wasteful of resources and can degrade the user experience (as well as increase demands on your server).

1. Select * from albums order by releaseDate desc limit 1;

| albumID | albumName | releaseDate | bandID | producerID |
|---------|------------|-------------|--------|------------|
| 5 | Live Noise | 1998-05-19 | 2 | NULL |

As a final example of some of the vocabulary available to us, we can also use the words “and” and “or” to add additional conditions to a statement:

1. Select * from albums where albumName like "%noise%" or albumName like "%circle%";

| albumID | albumName | releaseDate | bandID | producerID |
|---------|-------------|-------------|--------|------------|
| 3 | Full Circle | 1972-07-17 | 3 | NULL |
| 5 | Live Noise | 1998-05-19 | 2 | NULL |

Keep in mind we are just touching the surface of the power of MySQL here. There are more reserved words, actions, abilities, and a whole library of functions that allow you to do even more.

Update

Now that we have practiced a bit of reading, we will try the next CRUD method, updating. Maybe instead of leaving the producerID as null we decide that we want it to say unknown by default. Since the producerID field is a foreign key reference, we cannot just change the value to text (we could use the Alter command to change the table structure, but this is out of our scope and would break our normalization). Since we only want to change the records where we do not have a producerID (even though in our case it is all of them) we need to specify that we want to change the column where the field is null. For update, we need to specify the table, define what we want our field set to, and the condition(s) required for the update to occur. Before we do this, we need a record in our Labels table where the producer name is “unknown.” We will pretend it is our first record:

1. Update albums set producerID=1 where producerID is null;

Now, all of our records will show “unknown” when we begin to join tables. This keeps us from having to create extra code in our site to adjust output when the field would otherwise be empty.

If you want to change your values back, you can reset the whole column by setting the value without adding a where clause:

1. Update albums set producerID=null;

Delete

The final CRUD method, delete, is as final as it sounds. Use with extreme caution! There is no “undelete” or “undo” function at our disposal. The vast majority of the time, it is best practice to never allow your users to delete records. Instead, add flags to your records that will *hide* the data from ever appearing again, by adding a Boolean “disabled” column to each table or creating a disabled table that tracks records that should not be shown (just a couple examples, there are even more ways to do this!).

As partial protection to the fast-fingered typists, MySQL splits delete functions out to two keywords, delete and drop. Delete is reserved for row-level actions, while drop is reserved for table and database level actions. Dropping a table or database is as simple as typing “drop table [your table name here]” or “drop database [your database name here].” There will be no “are you sure” prompt either. If the value exists, it will be removed. In terms of deleting rows, the same holds true. We define the table we want to

interact with, and the conditions that identify rows we want deleted. We will remove any albums with “live” in their name as an example:

1. Delete from albums where albumName like “%live%”;

You should receive a response that says one row was affected, and if you review your whole table you will see that the Live Noise album is now gone.

Learn more

Keywords, search terms: CRUD, structure query languages, SQL, MySQL

MySQL Functions List: <http://dev.mysql.com/doc/refman/5.0/en/func-op-summary-ref.html>

Beginner Tutorials: <http://beginner-sql-tutorial.com/sql.htm>

4.5: MySQL CRUD Actions is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

- **4.5: MySQL CRUD Actions** by [Michael Mendez](#) has no license indicated.