**MySQL and Database Design:**

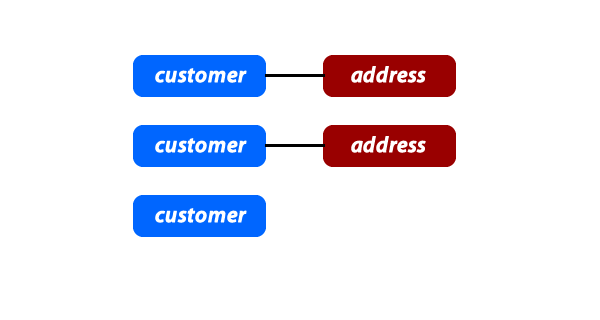
ERD: Entity Relationship Design

* ERD’s are visual blueprints for how your database looks and behaves. ERD’s and SQL work together very intimately.
* ERD is map of structure of how we want to store our data. SQL is the language we use to manipulate the data as per the relationships we defined in our ERD.

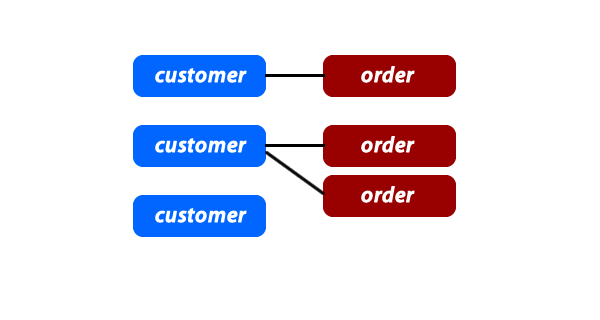
**Don’t Repeat Data!**

Database Relationships (remember to check both directions):

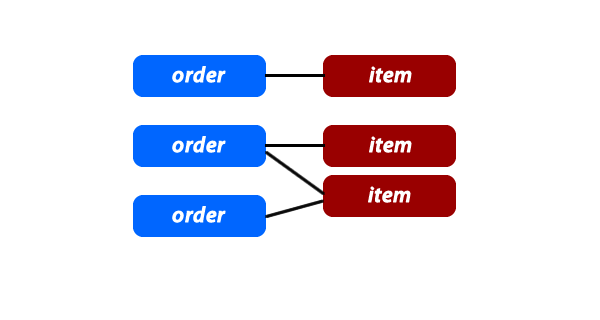
* One to One:
  + Customers and Credit Cards: Every Customer has one Credit Card, every Credit Card belongs to one Customer.
  + User and Email: Every User has one Email Address, every Email Address has one User.
  + Product and Image: Every Product has an Image, every Image is of a Product.



* One to Many
  + Messages and Comments - One Comment belongs to one Message, but one Message can have many Comments.
  + States and Cities - One City is only in one State, but one State can have many Cities.
  + Customers and Orders - One Order only has one Customer, but one Customer can have many Orders.



* Many to Many: **anytime you have a Many-to-Many, it will require some sort of joining table!**
  + Users and Interests - One User can have many Interests, one Interest can be applied to many Users.
  + Actresses and Movies - One Movie can have many Actresses, one Actress can be in many Movies.
  + Businesses and Cities - One Business can be spread across many Cities, one City can be home to many Businesses.



**Normalization:**

Database normalization is simply a convention for splitting large tables of data into smaller separate tables with the primary goal being to not repeat data. *It is possible to take normalization to an extreme.*

Some rules (in order of priority):

1. Each column in your table can only have 1 value.
2. Each column in your table that is not a key (primary or foreign) must have unique values.
3. You cannot have a non-key column that is dependent on another non-key column.

**Conventions:**

Reasons:

1. Developers can have a better understanding of your database if you are using a set of industry standards.
2. Developers can create software to automate a lot of the queries if some assumptions can be made.

**Guidelines:**

1. **Make the table name plural and ALL lowercase**
2. **Use “id” as the primary key**
3. **Name foreign keys with singular\_table\_name\_id**
4. **Use *created\_at* and *updated\_at* as columns for the timestamp in EVERY table you create**

DATA TYPES:

Simple Data Types:

* **VARCHAR(*number of characters*)**
  + Used to store non-numeric values that can be up to 255 characters. It is called a VARCHAR because it can store a variable number of characters and will only use the space required for each record that is stored in the database. VARCHAR should be used for values with different character lengths like an email, first\_name, or last\_name.
* **CHAR(*number of characters*)**
  + Also used to store non-numeric values, however, it will use up all space for the set number of characters regardless of what value is added. For instance, if I set CHAR(15), and I try to store the value "Coding", it will use up the equivalent of 15 characters even though "Coding" is only 6 characters long. Char is good to use for things that will always be a given number of characters. Char would work well for something like a state\_abreviation.
* **INT**
  + Used to store integers.
  + The columns that you will find mostly using the INT are things like a unique identifier for each table. The majority of rows in a table will not exceed 2.1 billion records. INT is good to use for most normal number values like a phone\_number or a zip\_code.
  + **unsigned** (positive numbers only) - can store numerical values from 0 up to 4294967295
  + **signed** (positive and negative numbers) - can store numerical values from -2147483648 up to 2147483647
* **BIGINT**
  + BIGINT would be used for columns that would need to store huge numbers. In most cases you wouldn't need BIGINT, but if you wanted to store something like a Facebook id when using Facebook's API, since they have over a billion users the id will need to be a data type of BIGINT.
  + **unsigned**(again positive numbers only) -can store numerical values from 0 up to 18446744073709551615
  + **signed**(positive and negative numbers) - can store numerical values from 9223372036854775807 to -9223372036854775808.
* **TINYINT**
  + TINYINT would be good to use for numbers that will be relatively small. A good example of something that would use a TINYINT is user level identifier (0 - inactive user, 1 - active user, 9 - admin).
  + **unsigned -**can store numerical values from 0 up to 255
  + **signed -**can store numerical values from -128 up to 127
* **FLOAT**
  + Used to store floating point numbers (numbers that need to have decimal places). An example column for this would be like an item\_cost.
* **TEXT**
  + Used to store a large amount of text, like a description, message, or comment. Use this for any text that VARCHAR() is too small to handle.
* **DATETIME**
  + used to store a date and time in the format *YYYY-MM-DD hh:mm:ss*

**MySQL Queries:**

SQL: Structured Query Language. Programming language designed for managing data in relational databases. SQL statements are used to perform tasks: SELECT data, SELECT data WHERE some conditions are true, INSERT data, UPDATE data, DELETE data, and JOIN different tables together.

MAMP (or WAMP) along with MySQL Workbench to interact with our database. **MAMP** plays important role by setting up two different servers: *web server* (called Apache) and *database server* (running MySQL Server). These servers each run on their own port.

Connecting to MySQL Server:

1. Open MAMP, specify which port you want the DB server to listen to, then start the server.
2. Open up MySQL Workbench, create a new connection to the port we specified in the previous step.

Importing Data:

1. If you have an SQL file, you can just copy and paste the commands into the editor and click run.
2. If you have an ERD diagram, you can forward engineer into MySQL workbench.

Inserting Records:

The SQL command pattern for INSERTing records is as follows:

INSERT INTO table\_name (column\_name1, column\_name2)

VALUES('column1\_value', 'column2\_value');

Updating Records:

The SQL command pattern for updating/editing records is as follows:

UPDATE table\_name SET column\_name1 = 'some\_value', column\_name2='another\_value' WHERE condition(s)

**IMPORTANT**: if WHERE condition is not added to the UPDATE statement, the changes will be applied to every record in the table.

Deleting Records:

The SQL command pattern for deleting/removing records is as follows:

DELETE FROM table\_name WHERE condition(s)

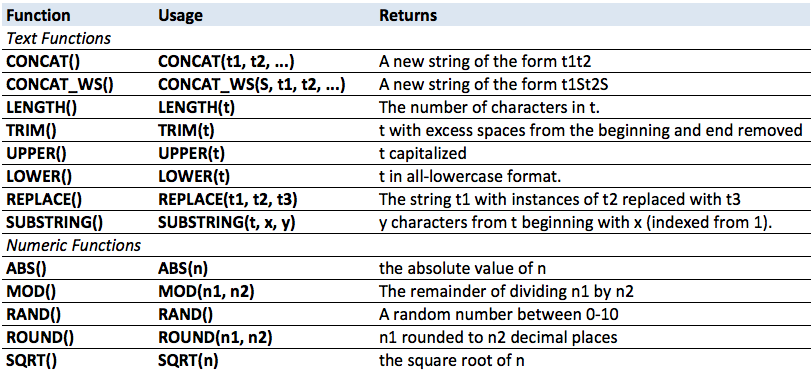
**IMPORTANT:** if WHERE condition is not added to the DELETE statement, it will delete all the records on the table.

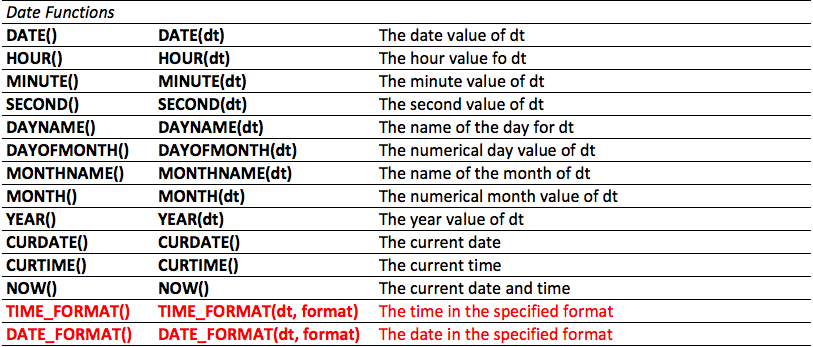
If unable to delete stuff from database, try running this command and delete again:

SET SQL\_SAFE\_UPDATES = 0;

Functions:  
When calling a function on a column, make sure that column is the appropriate Data Type for that function.

* **Text Functions Data Types** (VARCHAR, TEXT, CHAR etc.)
* **Numeric Functions Data Types** (INT, BIGINT, FLOAT etc.)
* **Date and Time Functions Data Types** (DATETIME)
* **SELECT FUNCTION** (column) **FROM** table\_name





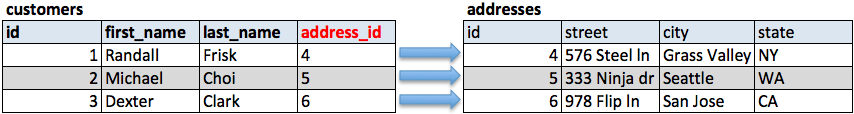
Joins:

The following below are “*INNER JOIN”*:

**One to One:**

SELECT \* FROM customers

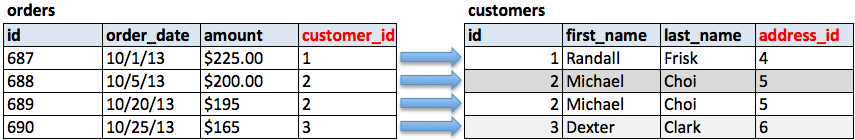
JOIN addresses ON addresses.id = customers.address\_id;



**One to Many:**

SELECT \* FROM orders

JOIN customers ON customers.id = orders.customer\_id;

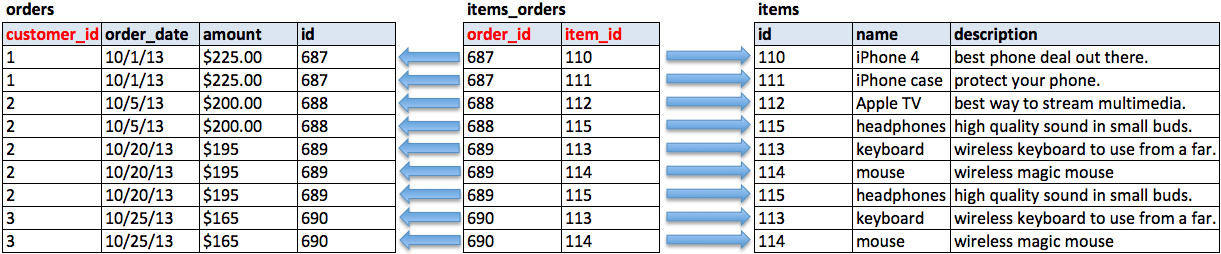


**Many to Many:**

SELECT \* FROM orders

JOIN items\_orders ON orders.id = items\_orders.order\_id

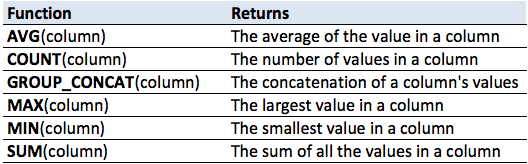
JOIN items ON items.id = items\_orders.item\_id;



Grouping Results:

In the previous section, we saw how we could use functions to manipulate a single value in a single row. With *GROUP BY*, we will group multiple rows together, by performing a function to combine the values of those rows. Because this results in a single result for the group, it will combine those grouped rows into a single resultant row.

As you can imagine, there are many different ways that we might combine multiple values into a result. Below are a list of the most common ones, often called **Grouping Functions or Aggregate Functions**.



Left Join vs Join:

When SQL uses the keyword **JOIN**, it **only includes those records that have matches on both sides**. It will omit any records that don't have a 'partner'. On the other hand, **LEFT JOIN** will **include all the records from the first table** (the 'left' table, the first one mentioned when reading from left to right), regardless of whether that record has a matching foreign key in the (right) table that we are trying to join to it. To summarize, JOIN will only include the intersection of the two tables, whereas LEFT JOIN will include all records from the first table, plus the records from the second table that correspond. This is why the JOIN is sometimes called the INNER JOIN, while all the other joins (including LEFT JOIN) are referred to as OUTER JOINs.

Exporting Database:

1. Select Server tab and choose “Data Export”
2. Select “Export to Self-Contained File” and set the directory and file name.
3. Include “Create Schema” (i.e. on top of file it’ll have the line “Create Database”).