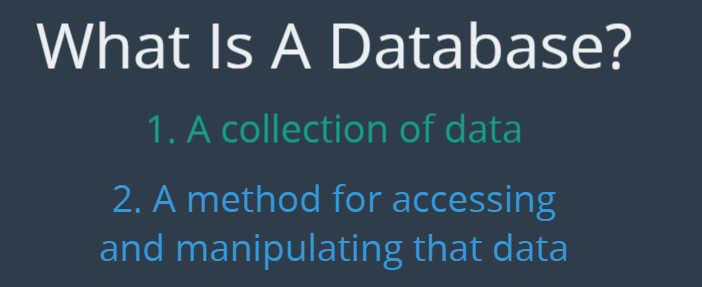
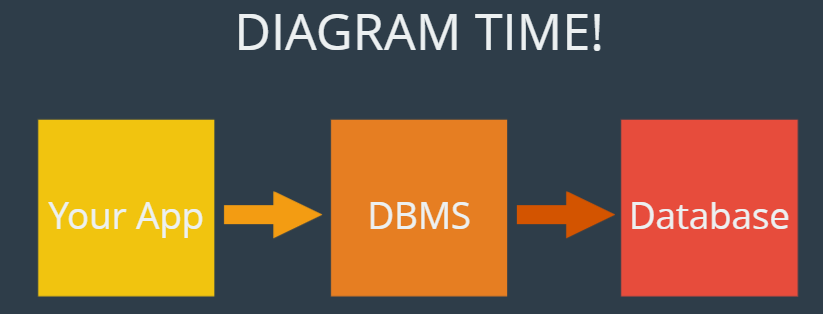
**Database**

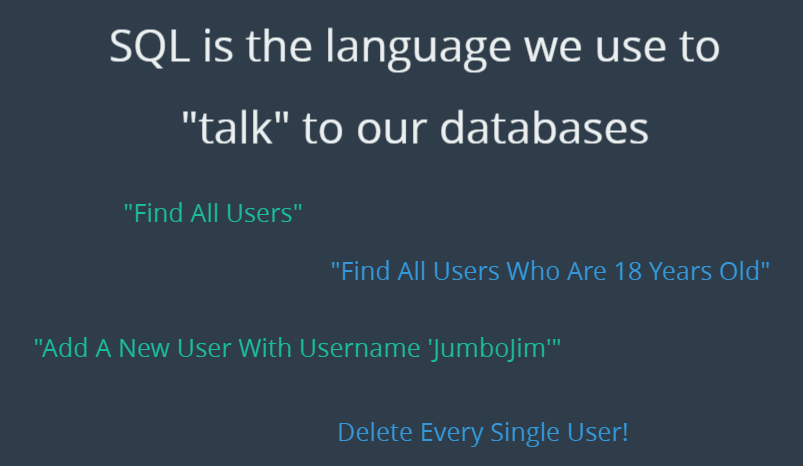
****

****

DBMS allows us to interface with the database (talks to our database for us). Hence, MySQL is a DBMS which allows us to interact with a database and perform operations on it (such as fetch some info, delete something etc). We give commands to this DBMS which then further sends those commands to the database itself.

Usually, people refer to DBMS and Database together as a database but its kind of different.

**SQL vs MySQL**

****

****

All these relational database management system uses the SQL standard. Whenever we use mysql, we will be writing sql all the time. SQL is a structured query language, and mysql is a DBMS that implements sql. SQL is how we interact with the data (add, delete fetch etc). We run queries and run them.

**Section 3: Creating Databases and Tables**



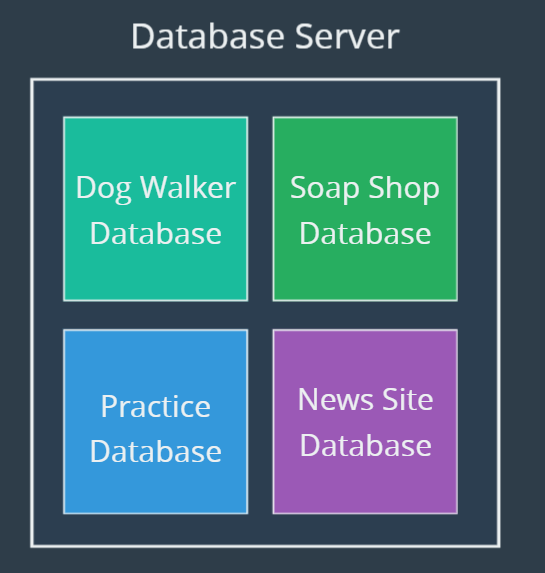
**show databases;** lists the current databases that exist in the mysql server. Used to create a database on the server.

**create database <name>;** use snake case while naming the databases, e.g. dogs\_app.

**drop database <name>;** deletes the database/table.

**use <database\_name>;** since we can multiple databases on our mysql server, we need to tell sql that which database we want to use at a particular time and switch using **use** command.

**select database();** tells the database being used currently.

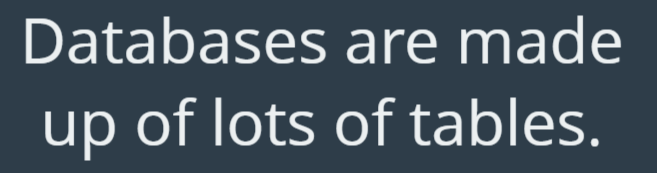


**TABLES**

A database is a bunch of tables. What goes in a database is just a bunch of tables and that’s where the data is stored.

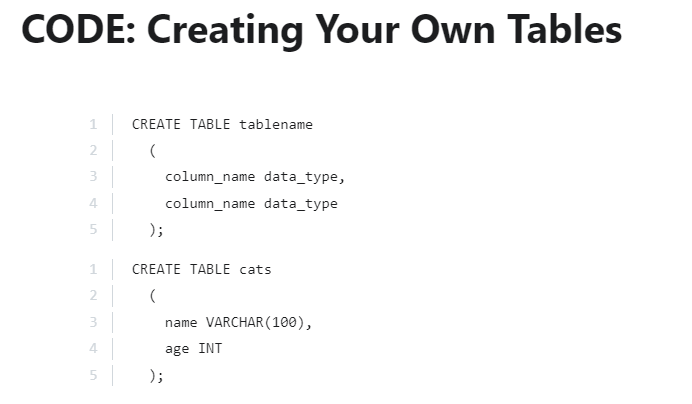
**Columns** in a table refers to the headers, the different component/pieces of data.

**Rows** are the actual data.



**DataTypes**

**Varchar** allows us to store variable length string whereas, **char** has a fixed length (everything in the column has to be of the same length). When we use varchar, we specify the maximum length aswell, e.g. **varchar(100)**.





**Show tables;** will show the current tables in our database.

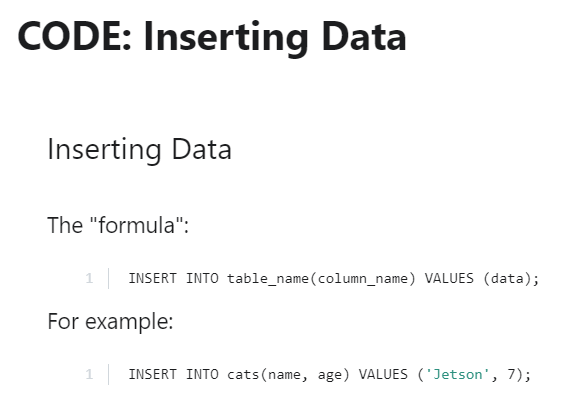
**Show columns from <tablename>;** we get to see the actual columns in the table.

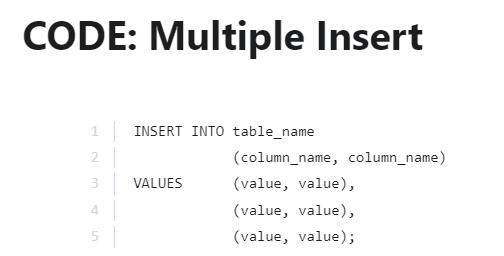
**DESC <tablename>;** we get to see the actual columns in the table.

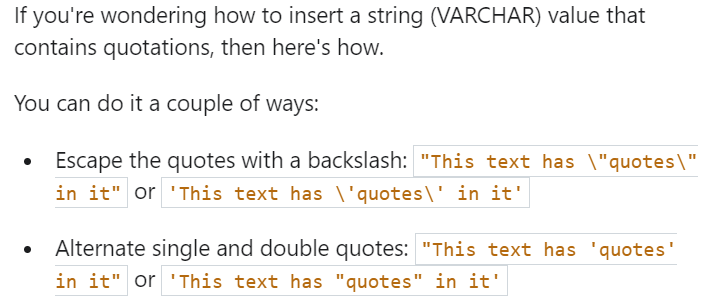


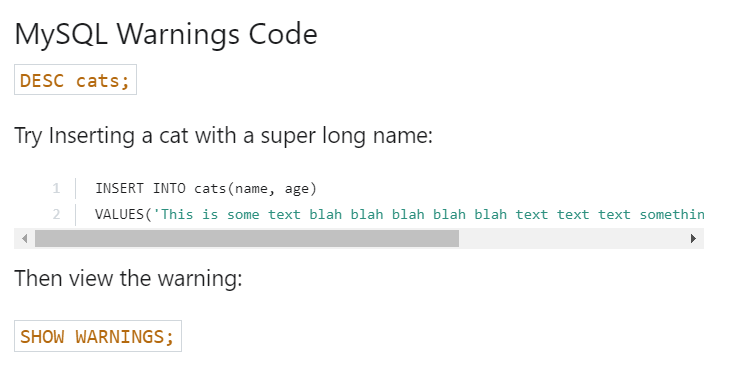
**Section 4: Inserting Data**

We can specify the columns in any order however, whatever we establish first, we have to use the same when we specify the values (so that it can match it).

****

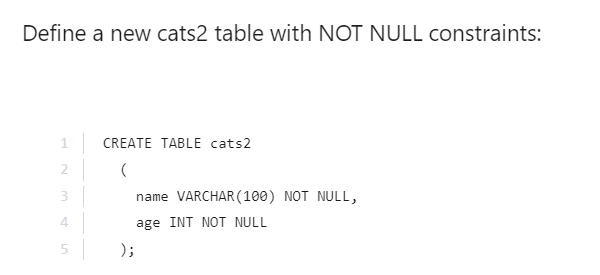
****

****

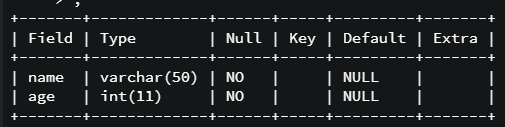
****

**NOT NULL**

When we use to view a table, if a field showed NULL as yes, that means we have permitted NULL to be a valid value for that field. We could type **insert into cats () values ();** and that would insert **NULL, NULL** in the table. Hence, in order to prevent this, we make use of **NOT NULL. Null** is not the same as 0 and by specifying not null, we are telling mysql that the field isn’t allowed to be NULL.



Hence, when we use **desc cats;** we see: (NULL is no)



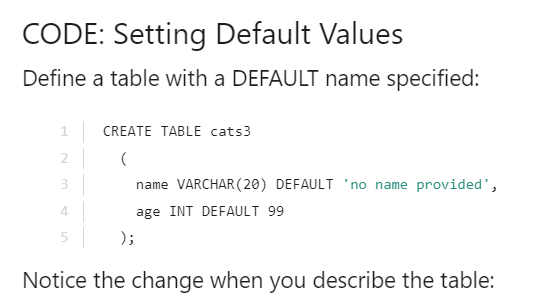
Earlier when we didn’t specify a value for a field and it was allowed to be NULL, it assumed NULL. However, now it cannot assume NULL and hence takes up the default value (if we don’t specify a value when inserting data) which can be set up by us or is 0 in case of an integer. Incase of a string, if we allow it to be NULL and forget to insert a value (while inserting data into the table) then the string takes up NULL however, if we say NOT NULL then the string will just show empty space.

**Setting Default Values**

If something is not specified then there will be a fallback to the default value.

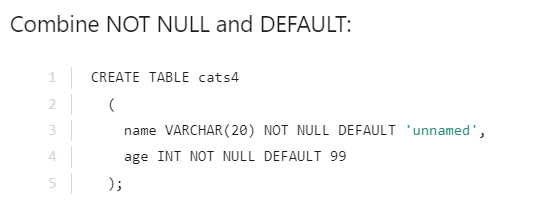
**\*\*\***<https://www.udemy.com/course/the-ultimate-mysql-bootcamp-go-from-sql-beginner-to-expert/learn/lecture/6912304#overview>**\*\*\***

If we set some default values and then forget to pass in a value, then that default value is given to the field.



If we allow NULL values (by not mentioning NOT NULL) and specify a default value but while inserting the values, we explicitly pass NULL as a value, then that value will be accepted.

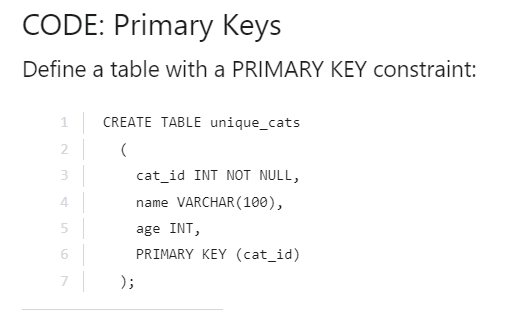
However, if we make a field NOT NULL and then try to pass NULL explicitly, we get an error. Hence, it is good to specify NOT NULL along with the default values so that passing of NULL value explicitly is denied.



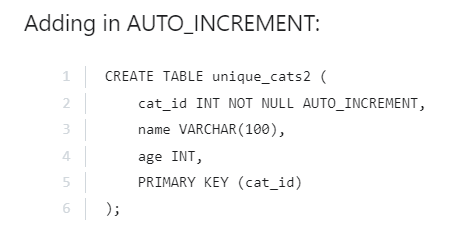
**Primary Keys**

Primary key refers to something that is a unique identifier in a row.

If we try adding a duplicate entry (same primary key value) then it wont be added to the database.



Since it’s a pain to keep incrementing the id and keep a check on where we last left, its best to use **AUTO\_INCREMENT** so that the id is updated automatically and we don’t have to provide it.



**Section 5: CRUD Commands (Create, Read, Update and Delete)**

**Read Command**

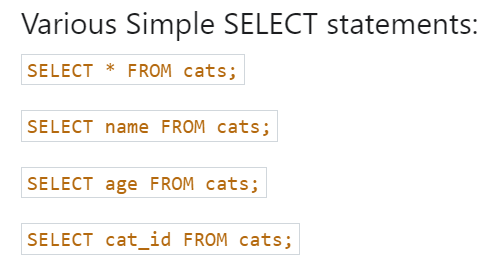
We specify which all columns we want.

Select \* is used to select all the columns and uses the default order to display the data in which we entered the values when we created the table.





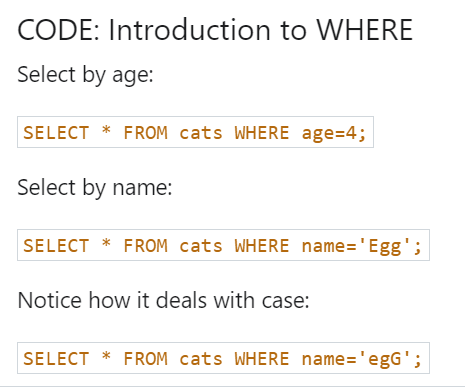




**WHERE Clause**

We use this when we want to select something particular or maybe update/delete something particular.

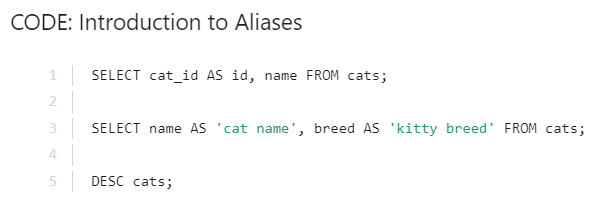
By default, when using the WHERE clause, it is case insensitive. (Where name = “lak” and Where name = “LAK” fetches the same result)



**Select name, breed from cats where age = id** // combining select expression & where clause

**Aliases**

Doesn’t change the original table but is used to change the column headers when we are reading the data (selecting).



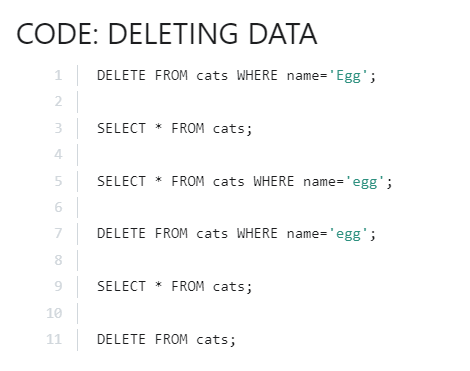
**UPDATE**



If we don’t specify the where clause then all the rows for that particular column will be updated.

**DELETE**

The ids remain the same even after deleting a particular row because it might be possible that the contents of this table (including the id) is used within another table and changing the ids can lead to malfunctioning.



**Line 11:** Since we didn’t select using a where clause, we kind of selected everything and hence it deleted everything.

Dropping a table gets rid of the table and its contents entirely whereas, **delete from cats** will only delete the entries within that table however the table is still retained.

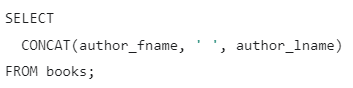
**Section 7: The world of string functions**

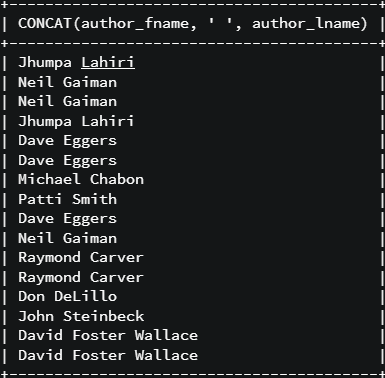
**mysql-ctl cli**

**source flename.sql**

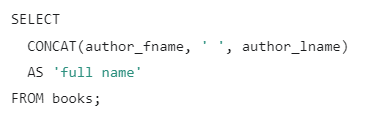
**CONCAT and CONCAT\_WS**

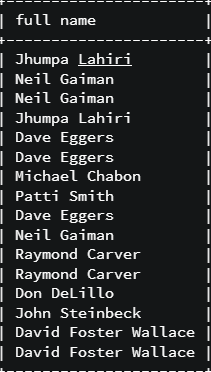
This wont change the original data in the database.

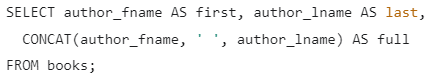


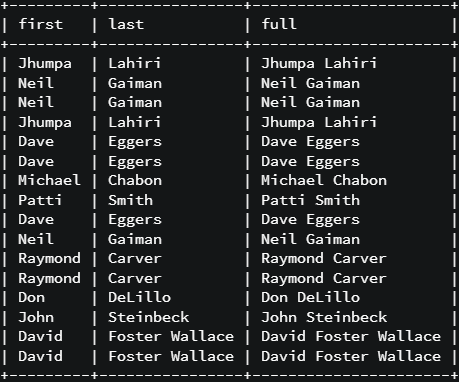


To remove the weird heading, we can use **AS**.



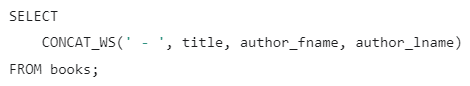








In the above syntax we can see that we wish to give **–** between each word. Hence, it becomes tedious to keep writing that and hence we can use **CONCAT\_WC** command which inserts the character between every two words.



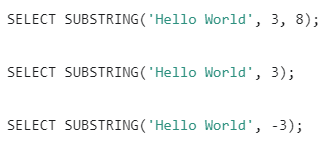


**SUBSTRING**

Allows us to select portions of a string.

Indexing starts from 1 and even blank space is counted. Both the start and end indices are inclusive. If we only provide 1 number then we get the entire string from that point till the end.

If we want to perform substring on some normal text then we can use the following syntax:

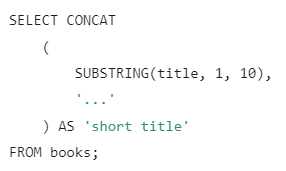


To perform substring operations on a string we can use the following syntax (using **AS** to avoid the weird column heading which is automatically generated by MySQL).



We can also use both concat and substring together (can use other string operators/functions aswell).

First substring runs, then concat runs, and then all this runs for the entire book table.

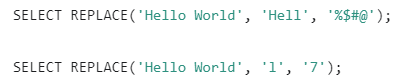


**REPLACE**

Replaces part of a string.

We just have to mention what string we want to perform the work on, what we want to remove, and with what it needs to be replaced with (it replaces all occurences).

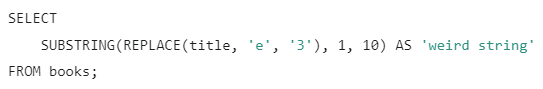
The following replaces all L with 7. It is case sensitive.



To perform this command on a table we can use the following syntax: (replaces all lower-case e to 3)



We can combine substring and replace together (can combine other things aswell). This will first replace the occurrences of e with 3 and then take the first 10 characters.



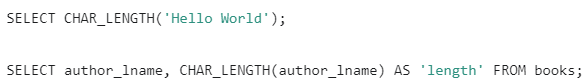
**REVERSE**

Reverses the String.



**CHAR\_LENGTH**

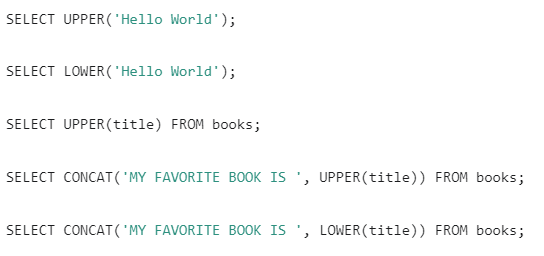
Counts characters in a given string.





**Changing case with UPPER and LOWER**

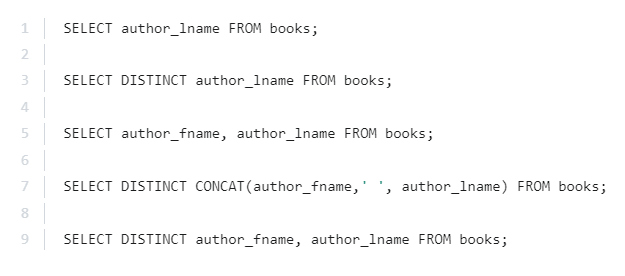
Change the case of a string.



**Section 8: Refining our Selections**

**DISTINCT**

Applies distinct to the entire row incase we type **DISTINCT fname, lname**.



**SORTING: ORDER BY**

Ascending by default.

Just add **DESC** at the end for descending order.



We can also use the following syntax. Here **2** is just used to replace the second column that we are selecting.

****

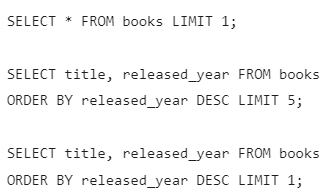
We can also use the following syntax so that we can ORDER BY 2 columns.



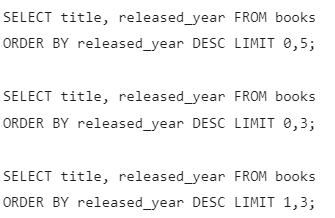
**LIMIT**

Specify the number of results that we want to select (eg to select top 10 highest selling books).

The limit command is executed at the last (basically everything else is performed first and then the top 5 or whatever number of rows we have specified are chopped off).



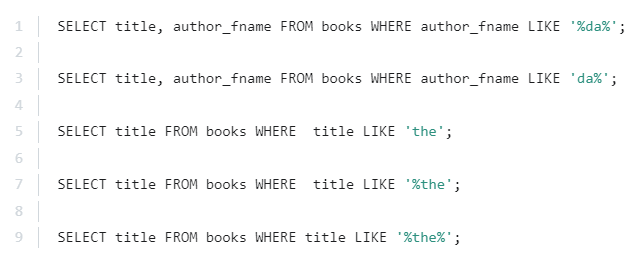
We can also use the following syntax wherein, first we specify the starting point (first row is 0) and then how many rows from there we travel ahead.



**LIKE**

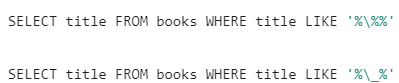
Allows better searching of the data. For exact matches we previously used **WHERE age = 10;** now we can use **WHERE age LIKE ‘%5’;**

**This is case insensitive.**

****

Wildcards are: **%**: 0 or more, **\_**: 1 character.

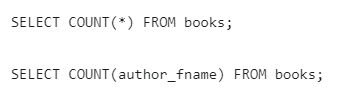
To search for a special wildcard character, we can use the following syntax:



**Section 9: The magic of aggregate functions**

Helps us to aggregate/combine data and get meaning out of it.

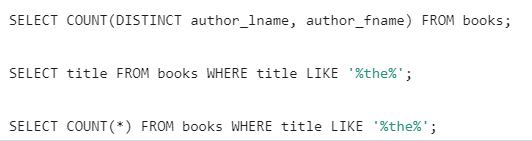
**COUNT Function**

****

If we want to calculate the distinct values then we can use the DISTINCT keyword:



We can also use the following syntax to get rows where the first name and last name are distinct:



**GROUP BY**

[**https://www.udemy.com/course/the-ultimate-mysql-bootcamp-go-from-sql-beginner-to-expert/learn/lecture/6915112#overview**](https://www.udemy.com/course/the-ultimate-mysql-bootcamp-go-from-sql-beginner-to-expert/learn/lecture/6915112#overview)

It summarizes or aggregates identical data into single rows.

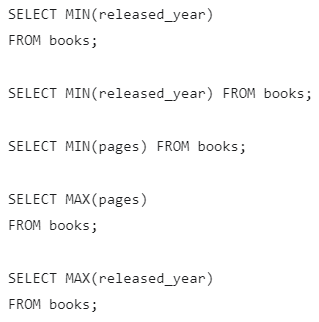


**The COUNT(\*) here is referring to the grouped rows, its going to the grouped rows and count how many rows have been grouped together.**

**Groups by wherever author\_fname and author\_lname are unique.**

**We cannot use GROUP BY with any random column, we can use it with aggregate functions like the count function (to see how many books any unique author has written).**

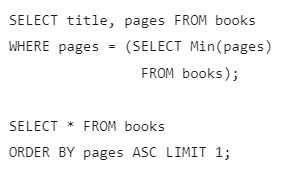
**MIN and MAX**

****

If we type **SELECT MAX(pages), title FROM books;** then since there can only be 1 max pages book, then we get the max number of pages. However, it outputs the first book in the table because we are only allowed 1 row. Hence, both of them aren’t interconnected. The max is first executed and theres only 1 row its preparing to print out and since only 1 spot is there, it prints the first row.

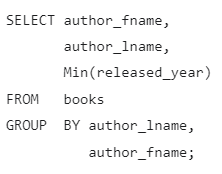
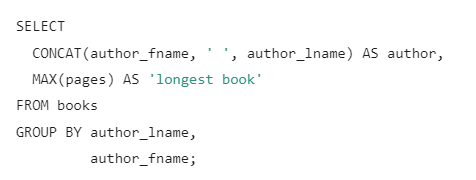
Let us say we want to access the name of the book which has the maximum number of pages, then we can do the following.

**Subqueries with MIN and MAX**

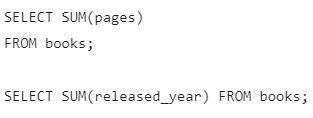
****

In the first query, we are using the concept of subqueries and there are 2 queries that are being run and hence it’s a bit slow. That’s the reason we can use the second approach.

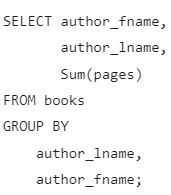
**MIN/MAX in conjunction with GROUP BY**

**** ****

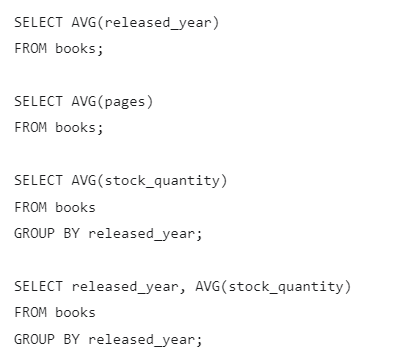
**SUM**

****

Using sum in conjunction with GROUP BY.



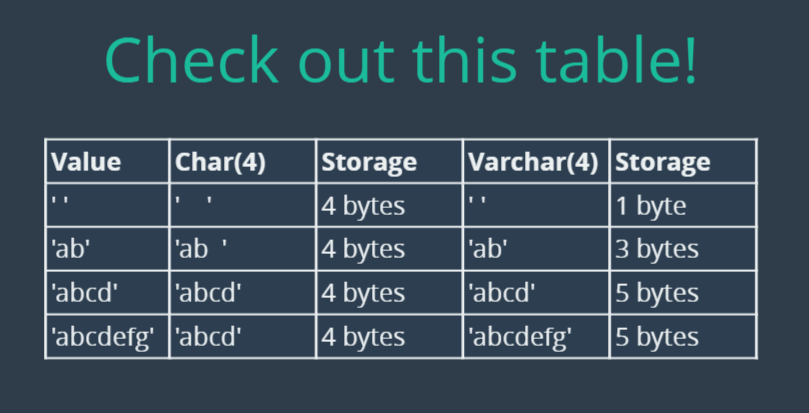
**AVERAGE**

****

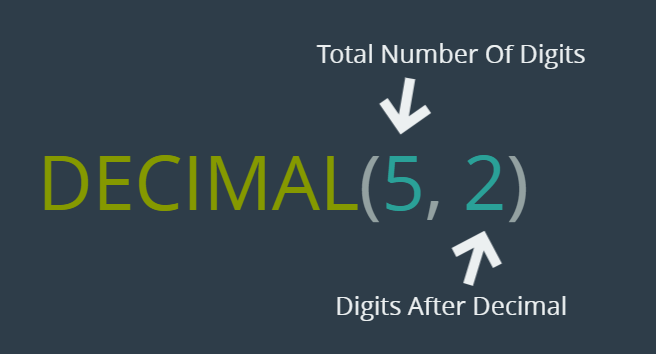
**Section 10: Revisiting Data Types**

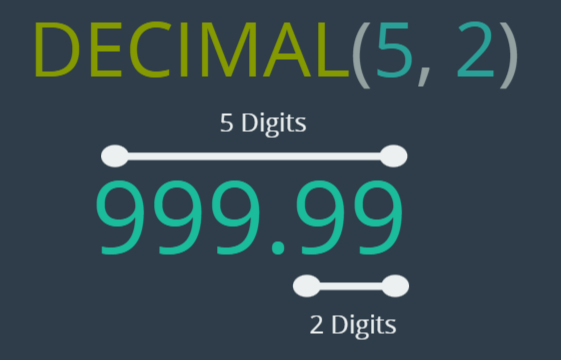
Both **CHAR and VARCHAR** store text but **CHAR** has a fixed length. A **CHAR** will always allocate the same amount of space for every row or every instance of that char (meaning if its acquiring less space then blanks will be added). However, when we retrieve it then we get only the text and not the blanks.

**CHAR** is faster for fixed length text.



**DECIMAL**

****

****

If we store a number like 3267326, then it will be displayed as 999.99 since that’s the max 5- digit number with 2 decimal places.

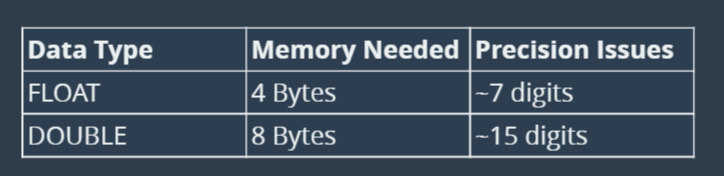
If we store a number like 2, then it will be displayed as 2.00.

If we enter 1.99999, then 2.00 will be displayed since we rounded it off to 2 decimal places.

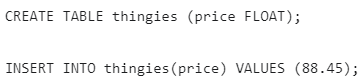
**FLOAT and DOUBLE**

**FLOAT** and **DOUBLE** can store larger numbers using less space however, **it comes at the cost of precision.**

A double is same as float (with the capacity to store larger numbers) however, they take up more space in memory.



We should use decimal unless we don’t mind losing some precision.

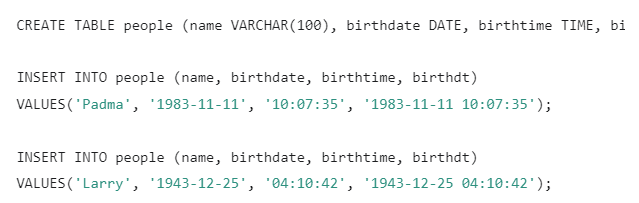


**DATE and TIME**





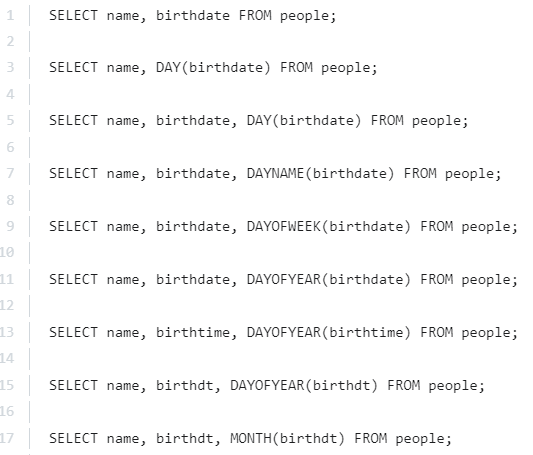


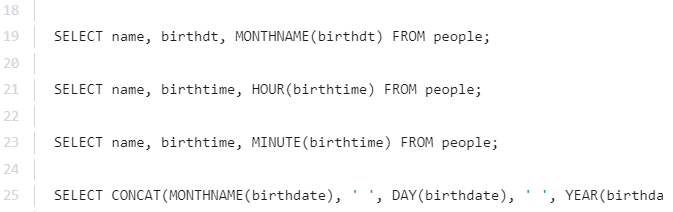


**CURDATE, CURTIME, and NOW**

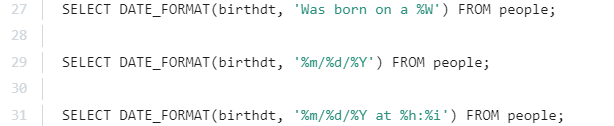
These 3 functions help to insert the curdate, curtime and curdatetime in a row.

**Formatting Dates**

****

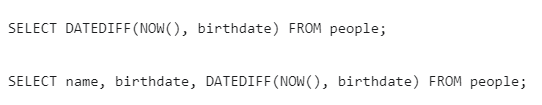
****

Instead of using 4 different functions, we can just use the following syntax and pass a **specifier** to it. This DATE\_FORMAT functions allows us to format things powerfully.

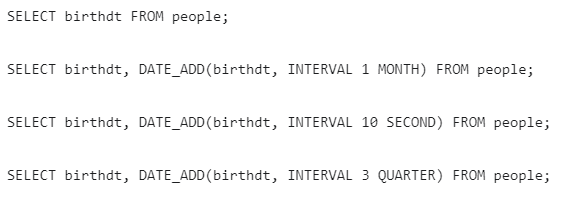


**DATE MATH**

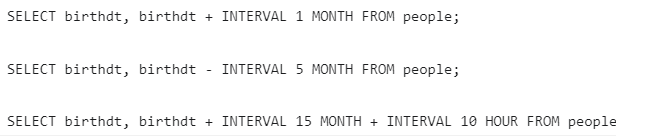
**DATEDIFF** calculates difference in dates. (only days)

****

We can add using the INTERVAL keyword.



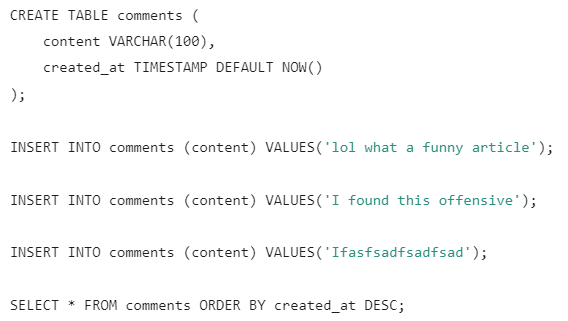
We should use **+,-** because with **DATE\_ADD** we cannot chain two additions or subtractions together (by chaining we mean adding months and seconds together) which we can using +,-.



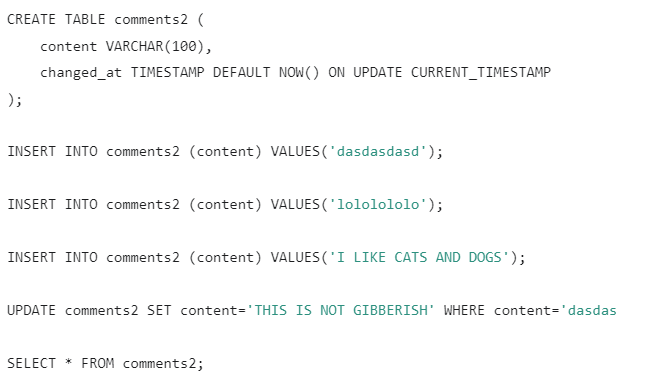
**Working with TIMESTAMPS**

The **TIMESTAMP** data type is just like the **DATETIME** datatype except the fact that it has a particular range of dates/time and takes up less amount of space.

The following schema is used so that whenever we enter a new value into a row, its date and time of addition is automatically updated.



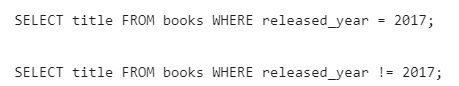
We can use the following query to track the time of whenever something is updated.



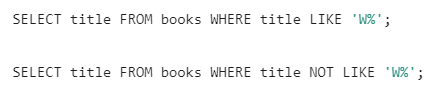
TIMESTAMPS are most oftenly used when we want to add meta-data about a row regarding the time it is updated.

**Section 12: The power of LOGICAL operators**

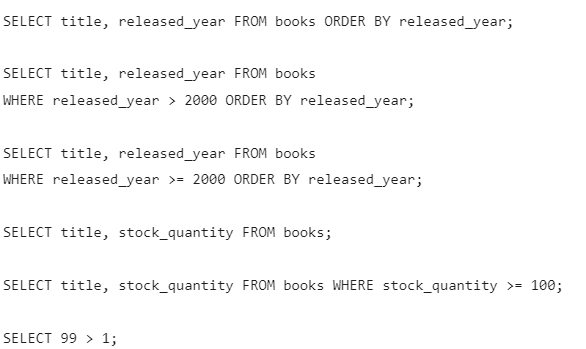
**NOT EQUAL**



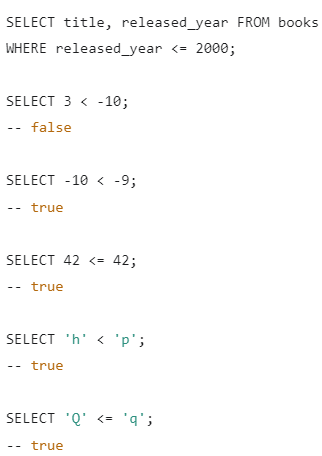
**NOT LIKE**

****

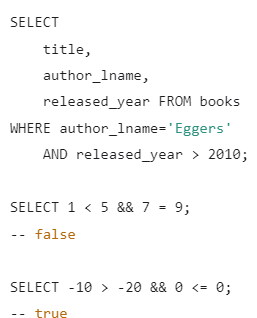
**GREATER THAN**

****

**LESS THAN**

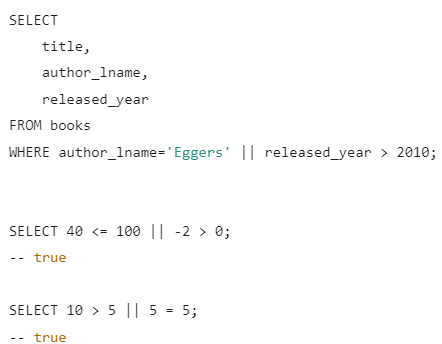
****

**AND**

****

However, we can only use **AND** and not **&&**.

**OR**

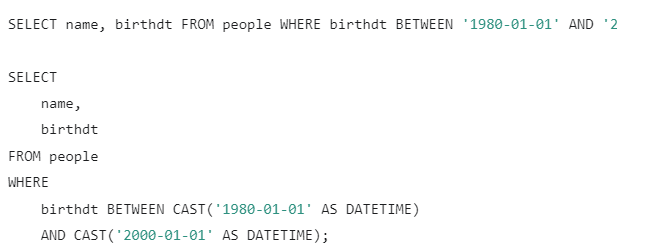
****

However, we can only use **OR** and not **||**.

**BETWEEN, NOT BETWEEN**

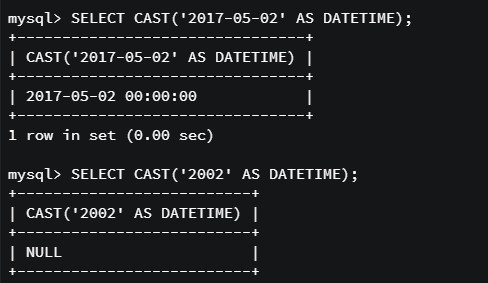
The AND keyword used here is a part of the between syntax.

When using BETWEEN, it’s best to cast the values to the desired data type.



**CASTING**

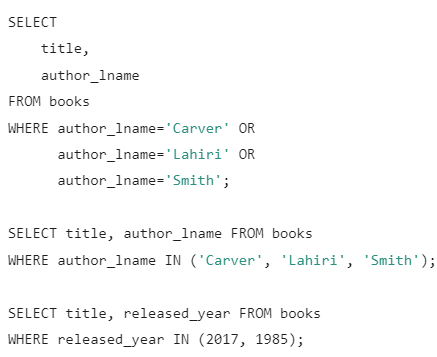
Whatever we past initially, is tried to be converted into the specified data type.



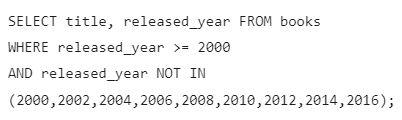
**IN, NOT IN**

We can provide a set of values and check if a given column is in that set.

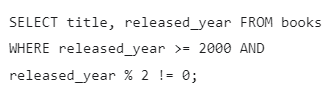
For example, if we wish to select all the books written by A, B or C then, we can either do it using the OR operator or simply use the IN operator.



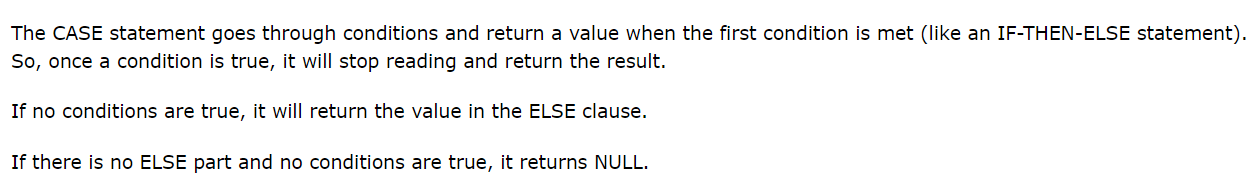
Use logical AND in conjunction with NOT IN.



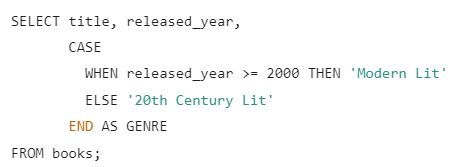
An even much better way is by using the **%** operator for not including the even years.

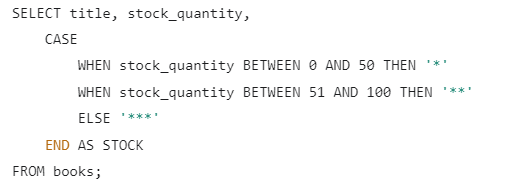


**CASE Statements**

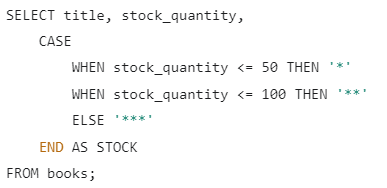
****

I’ve used **AS** then, the title would look completely weird.





We can remove the BETWEEN keyword and take advantage of the way how the things are executed.



**SELECT 1 IN (5,3) || 9 BETWEEN 8 AND 10;** // returns a true because 9 is between 8 and 10.

**Section 12: JOINS, One to Many**

**One-to-one:** customer and customer details.

**One-to-many:** one entity like BOOKS can be associated with another entity like REVIEWS wherein one book can have many reviews but those reviews belong to 1 book.

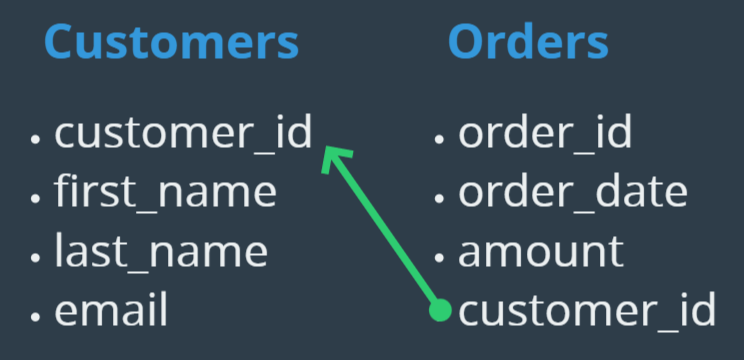
**Many-to-many:** authors and books because 1 author can write many books and a book can be written by many authors.

**ONE TO MANY**

To store details regarding a customer and the orders placed by him/her we could use 1 table as we have done below. However, this leads to a lot of duplication. Also its better to keep the customer and the orders data separate so that if we wish to interact with only a single table then, we don’t have to interact with one giant table.

****

We use the following one-many approach:



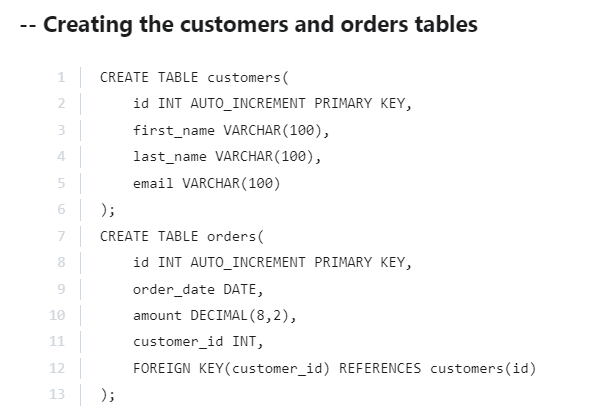


customer\_id is **primarykey** in the **customers** table (has to be unique otherwise the whole point of orders table is lost as it wont be able to track that to which customer a particular order belongs (considering customers have duplicate customer ids)). order\_id is the **primarykey** in the **orders** table which is the primary way to reference rows in that table.

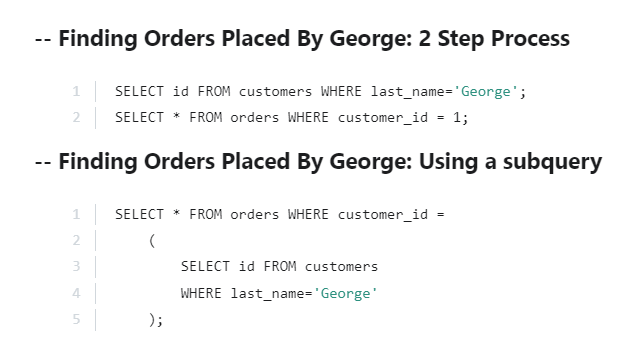
****

**Foreign keys** are references to another table within a given table. We need to specify it as a foreign key explicitly because we have to enforce that whatever customer\_id is present in the orders table, should be present in the customers table.

A foreign key means that it is referencing to something outside the table.

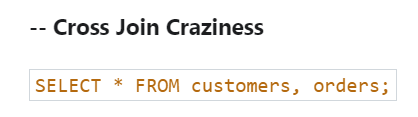


Now if we wanted to select orders placed by George then, we can either use the given syntaxes or JOINS.

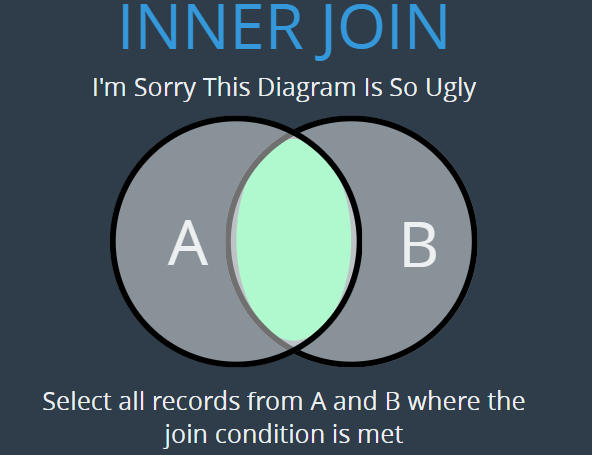
****

**1) CROSS JOIN**

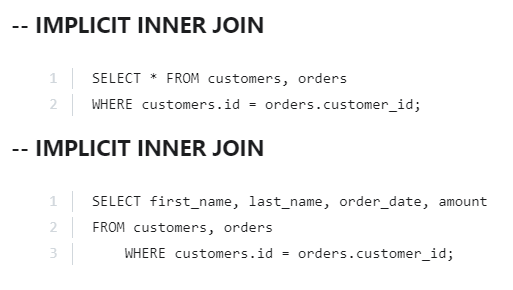
It just gives us the cartesian product.

****

**2) INNER JOIN**

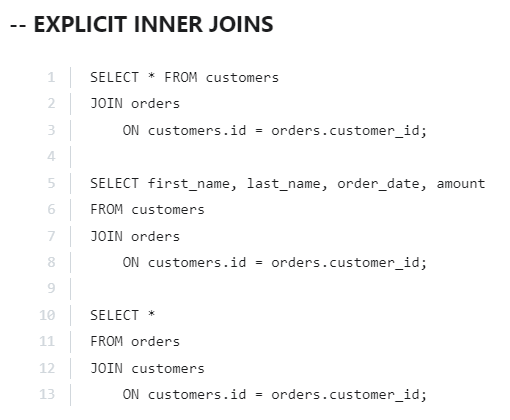
****

IMPLICIT INNER JOIN



EXPLICIT INNER JOIN **(explicit because of the fact that we use JOIN)**

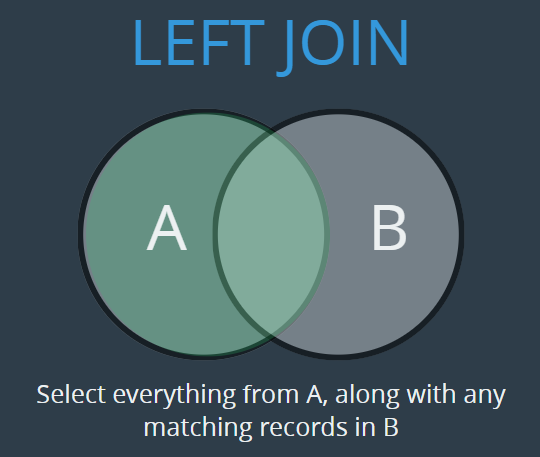
**ON** is used to tell where we want to join the tables.



**IN THE CASE OF A CROSS/INNER JOIN, THE ORDERING OF TABLES DOES NOT MATTER (ONLY VISUAL REPRESENTATION IS AFFECTED)**

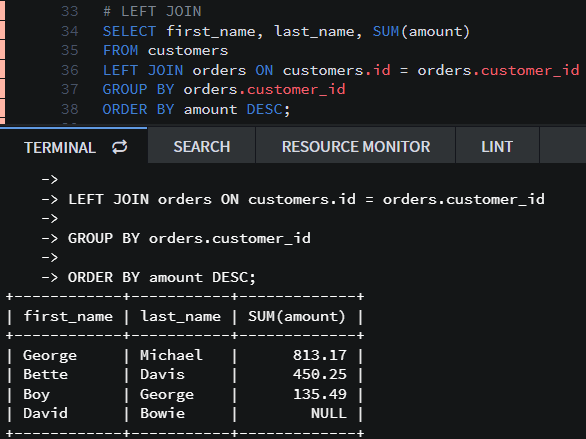
**We can either write INNER JOIN or JOIN (inner is taken by default).**

**3) LEFT JOIN**

****

Takes everything from the left table and from the right it will add the matching records.

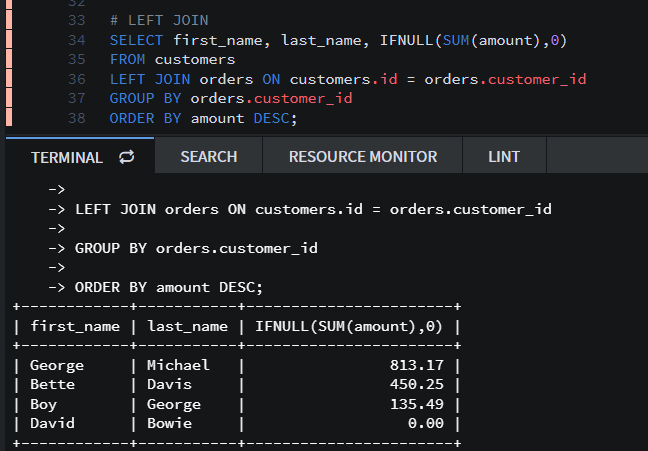
Why would you want to have access information like ‘Person A’ who hasn’t placed any order. And it depends on what you're doing. And in this case, maybe I previously I wanted to see, OK, all the orders that have been placed and names next to them. But now what if I want to tabulate the high spenders on our site, but I want to include people who haven't spent anything. So I just want to be able to see for every user how much have they spent. And I'd be able to go through and say, you know, maybe send an email to the people who have spent a lot. OK, thanks for being a loyal customer versus people who haven't spent. I could go through and send an email. Here's a coupon for your first purchase. Ten percent off, something like that. Right where I want to basically get some insight where it involves knowing everything about all users and some of them have corresponding orders and some of them don't.

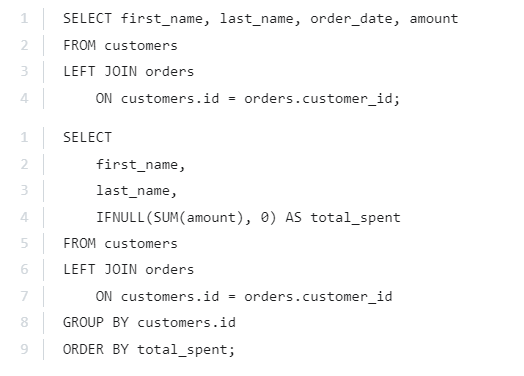


Now if we want to order by the amount each particular user has spent and for the users who have placed NULL orders, we want the amount to be 0, then we can either use a switch case or the better option is to use **IFNULL**. **IFNULL(agr1,agr2); IFNULL(SUM(amount),0)**

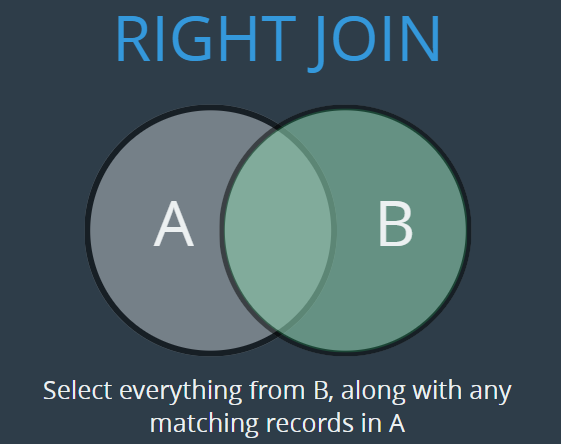
**arg1 =** thing we want to check if its null or not.

**arg2 =** if the first thing is null then replace it with this.



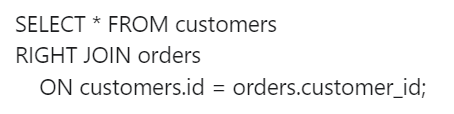


**4) RIGHT JOIN**

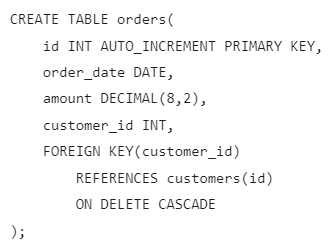


Takes/displays all orders regardless of whether they have a customer or not (incase we delete a custom row by mistake). **However, deleting a customer row is not possible unless we delete the order for that customer because we have made a foreign key and as per the constraints of the foreign key, it doesn’t let us create an order with a faulty or non-existent customer id. Hence, it doesn’t allow us to delete a custom if its dependent.**

**Also note that we cannot delete the customers table unless we have deleted the orders table.**

****

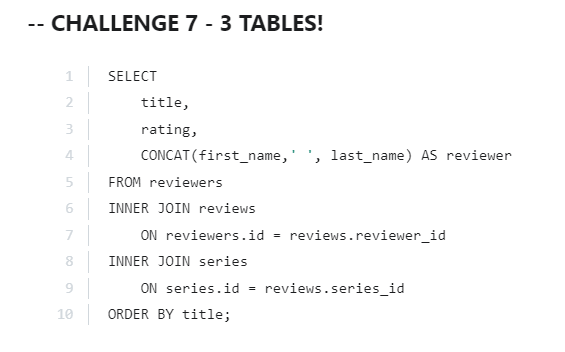
**New constraint which we can add to our schema so that when something is deleted that has a foreign key dependent on it, then it will automatically delete the dependent record.**

****

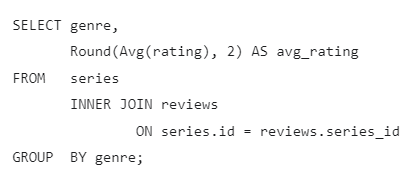
[**https://www.udemy.com/course/the-ultimate-mysql-bootcamp-go-from-sql-beginner-to-expert/learn/lecture/6965514#overview**](https://www.udemy.com/course/the-ultimate-mysql-bootcamp-go-from-sql-beginner-to-expert/learn/lecture/6965514#overview)

**Section 13: Many To Many**

**Round Function**

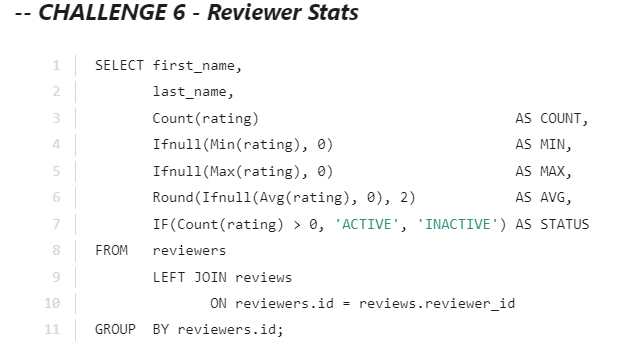
****

What we want to round and upto how many decimal places.



**IF Function**

For simpler case statements, we can use the IF function. **Line 7**



**HAVING**

[**https://www.udemy.com/course/the-ultimate-mysql-bootcamp-go-from-sql-beginner-to-expert/learn/lecture/7026262#overview**](https://www.udemy.com/course/the-ultimate-mysql-bootcamp-go-from-sql-beginner-to-expert/learn/lecture/7026262#overview)

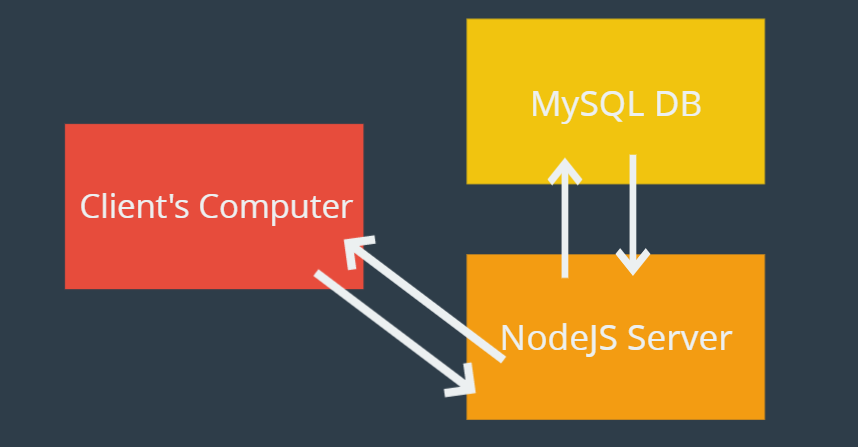
**WHERE clauses go before GROUP BY telling it what the data we’d like to select to group versus what we’d like to select from the grouped data. Hence, we use HAVING which takes our grouped data (end result) and allow us to filter based off a clause/condition. (basically we can first group and then filter out some values using HAVING).**

****

**Section 16: Introducing NODE**

How we take a programming language and combine it with mysql to build web apps.

Request is sent to the NODE JS server which further contacts the MYSQL database.





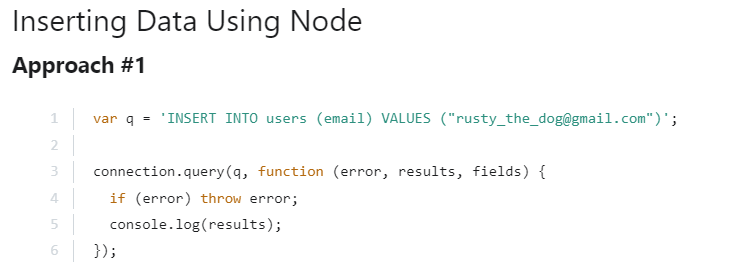
One other thing to address is what what is node if you're not familiar with it? And it's actually a pretty common question. There's a bit of confusion about Node Narges, which is what it's called, and regular JavaScript and what the differences is. Basically, JavaScript is a programming language. It was created first. It's been around for a while and it's existed as a language that you can use on the client side. So what that means is that back on this diagram, there's a line right here, not right now, but imagine a line. This is the client said so I could write JavaScript code that did things over on someone like on my computer. You could write code that would animate something or make a game or just add some pretty effects. That's not to say that it's not powerful, but traditionally it was used kind of to decorate pages, if you will, and it had nothing to do with databases. It had nothing to do with servers. You would use another language over here and you would use JavaScript on the client side of the front end. But then Node came around about five years ago now. And what it is, is an implementation of JavaScript written so that you can use it on the back end, so that you can do things like this. You can create a server with JavaScript instead of having to do it with Ruby and it can talk to a database like any other server-side language. So to summarize, that node is JavaScript.

It uses basically the exact same syntax, but you can use it on the back end, which is what we'll be doing in this course.

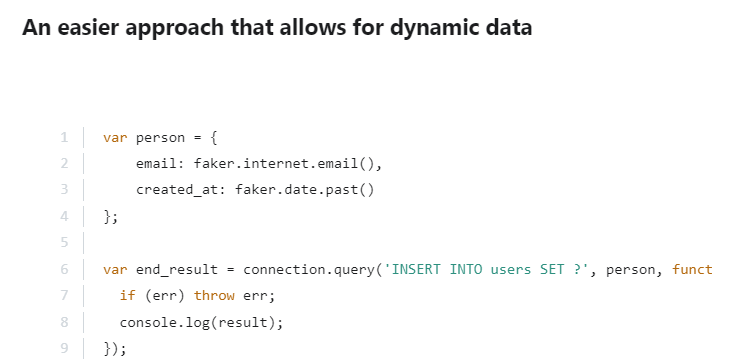
We will be using **Faker** package so that we don’t have to write all the code ourselves.

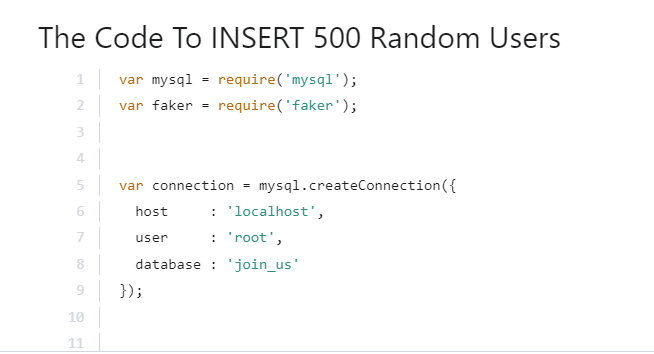
<https://www.udemy.com/course/the-ultimate-mysql-bootcamp-go-from-sql-beginner-to-expert/learn/lecture/6996806#questions/16585932>

<https://www.udemy.com/course/the-ultimate-mysql-bootcamp-go-from-sql-beginner-to-expert/learn/lecture/6996812#questions/16585932>



The above approach 1) doesn’t allow us to add dynamic data and 2) we have to specify all the column names when we insert the data whereas, in the syntax given below we just create an object and it automatically figures out the corresponding key-value pairs. **(MySQL package will take this object and figure out the key-value pairs as column-value pairs)**



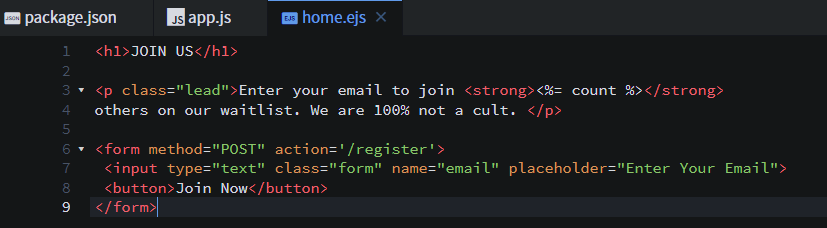


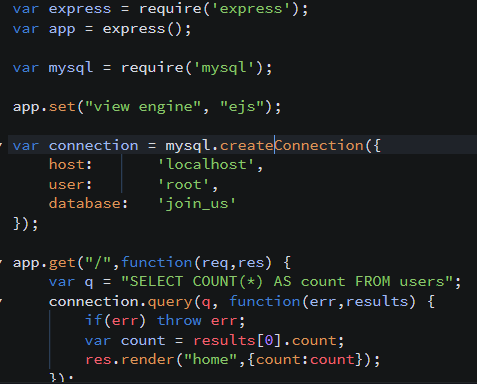
**Express and EJS**

Rendering home.ejs file which it automatically looks for in the views directory.

**<%=dat%>** special way of saying this is not html, we can put javascript here.





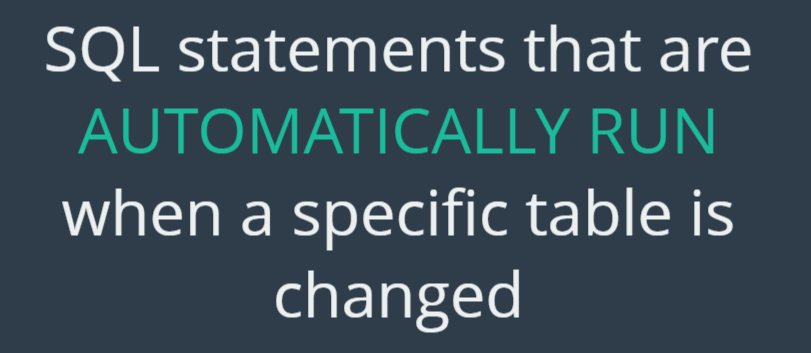


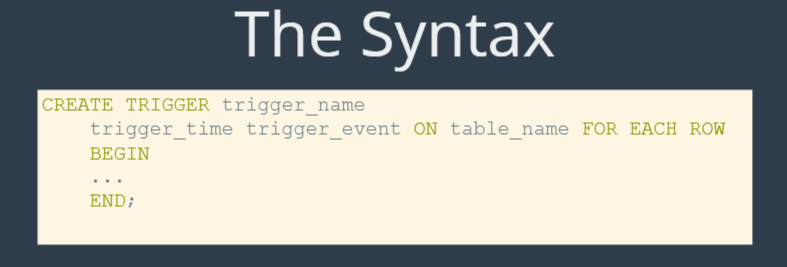
**POST Request**

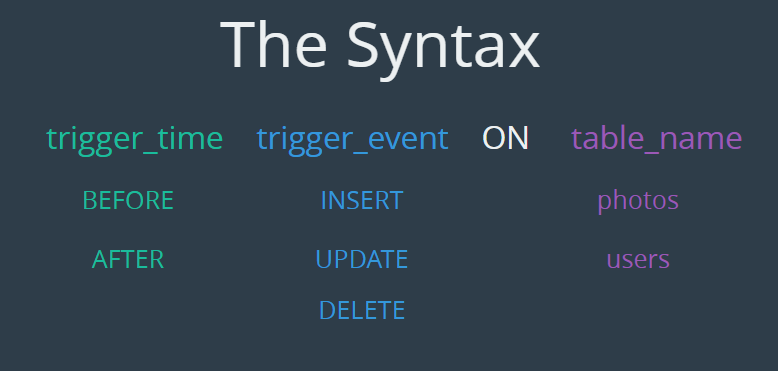
**Name** is the label under which data is being sent to the route.

**Bodyparser** will turn the giant piece of data that we pass into javascript which can then be used by us.

**Section 18: Database Triggers**







**Trigger\_time:** when we want the code to run.

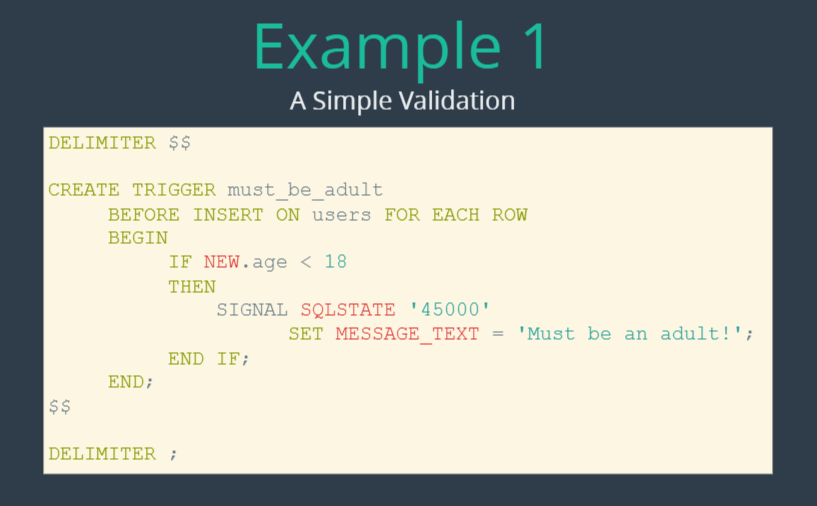
**Trigger\_event:** what event we want it to run for.

**Table\_name:** has infinite choice.

So we could run some code before we insert data into the users table.

1) Can be used for data validation. (run code before inserting data into the table) However it is better to do this validation on the client side and not on the server side where we are waiting for the database to reject the entry.

2) Manipulating other tables based on actions performed on 1 table.

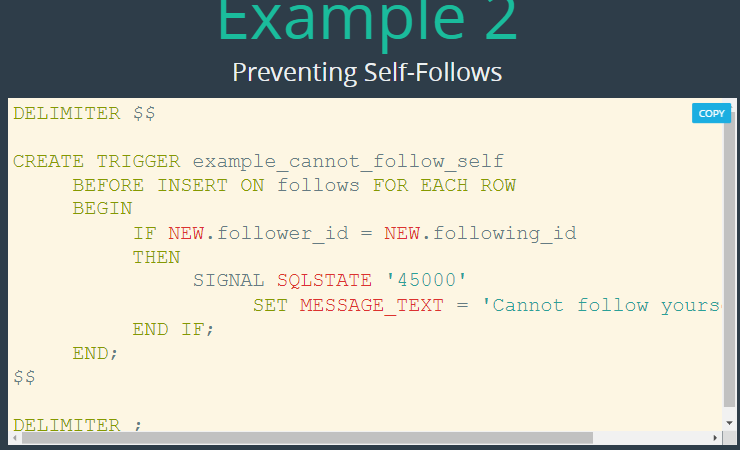


**NEW** is just a placeholder and refers to the new data (user) that is being inserted into the table. (Incase we deleted some data and want to have access to it we can use **OLD**).

So 45000 is sort of like a wild card state that is generic and it represents an unhandled user defined exception. So it's kind of just left out there for us to to use it for developers to return 45000 as a way of saying, hey, this is something I've come up with. This is not a MySQL, it's not a sequel thing like, you know, unknown table or a syntax error or something. This is something that I've defined and I've set the message. So signal skill state in quotes four or five zero zero zero is what we'll have any time we want to throw an error.

**WATCH FROM 9:00 – 10:00 to learn about delimiter:** <https://www.udemy.com/course/the-ultimate-mysql-bootcamp-go-from-sql-beginner-to-expert/learn/lecture/7517220#overview>

Now allowing someone to follow him/herself. We can do this on the client side by not displaying the button at first place and not bothering the database by telling it to insert and then waiting for the request to be rejected.



**Dependencies to install**

**1) npm init**

**2) npm install mysql –save**

**3) npm install express –save**

**4) npm install ejs –save**

**5) npm install body-parser --save**

**6)**  **npm install @faker-js/faker**

