CRUD

Monday, October 21, 2019 9:02 PM

In **computer programming**, create, read, update, and delete (**CRUD**) are the four basic functions of persistent storage. Alternate words are sometimes used when defining the four basic functions of **CRUD**, such as retrieve instead of read, modify instead of update, or destroy instead of delete.

mySQL example
Create
Read (Select…)
Update (Update …Set…)
Delete (Delete, Drop)

Difference Between DDL and DML in DBMS

Data Definition Language (DDL) and Data Manipulation Language (DML) together forms a Database Language. The basic difference between DDL and DML is that **DDL** (Data Definition Language) is used to Specify the database schema database structure. On the other hand, **DML** (Data Manipulation Language) is used to access, modify or retrieve the data from the database. Let us discuss the differences between DDL and DML, with the help of comparison chart shown below

Comparison Chart

BASIS FOR COMPARISON	DDL	DML
Basic	DDL is used to create the database schema.	DML is used to populate and manipulate database
Full Form	Data Definition Language	Data Manipulation Language
Classification	DDL is not classified further.	DML is further classified as Procedural and Non-Procedural DMLs.
Commands	CREATE, ALTER, DROP, TRUNCATE AND COMMENT and RENAME, etc.	SELECT, INSERT, UPDATE, DELETE, MERGE, CALL, etc.

DDL stands for **Data Definition Language.** The Data Definition Language defines the database **structure** or database **schema**. DDL also defines additional properties of the data defined in the database, as the domain of the attributes. The Data Definition Language also provide the facility to specify some constraints that would maintain the data consistency.
Let us discuss some commands of DDL:
CREATE is command used to create a new Database or Table.

ALTER command is used to alter the content in the Table.
DROP is used to delete some content in the database or table.
TRUNCATE is used to delete all the content from the table.
RENAME is used to rename the content in the database.
One can notice that DDL only defines the columns (attributes) of the Table. Like other programming languages, DDL also accept the command and produce output that is stored in the data dictionary (metadata).

Definition of DML (Data Manipulation Language)

DML stands for **Data Manipulation Language**. The schema (Table) created by DDL (Data Definition Language) is populated or filled using Data Manipulation Language. DDL fill the rows of the table, and each row is called **Tuple**. Using DML, you can insert, modify, delete and retrieve the information from the Table.
Procedural DMLs and **Declarative DMLs** are two types DML. Where Procedural DMLs describes, what data is to be retrieved and also how to get that data. On other hands, Declarative DMLs only describes what data is to be retrieved. It doesn't describe how to get that data. Declarative DMLs are easier as the user has only to specify what data is required.
The commands used in DML are as follow:
SELECT used to retrieve the data from the Table.
INSERT used to push the data in the Table.
UPDATE used to reform the data in the Table.
DELETE used to delete the data from the Table.
If we talk about SQL, the DML part of **SQL** is non-Procedural i.e. **Declarative** DML.

Key Differences Between DDL and DML in DBMS

The basic difference between DDL and DML is that DDL (Data Definition Language) is used to define the schema or the structure of Database which means it is used to create the Table (Relation) and the DML (Data Manipulation Language) is used to access, or modify the schema or Table created by DDL

DML is classified in two types Procedural and Declarative DMLs whereas the DDL is not classified further.

CREATE, ALTER, DROP, TRUNCATE, COMMENT and RENAME, etc. are the commands of DDL. On the other hand, SELECT, INSERT, UPDATE, DELETE, MERGE, CALL, etc. are the commands of DML.

Conclusion:

For forming a database language both DDL and DML is necessary. As they both will be required to form and access the database.

DB vs Spreadsheets

Saturday, August 3, 2019 9:13 PM

When you think of a database, the first thing that may come to your mind is your address book. Perhaps your address book is digital and looks something like this. Storing your address book in a spreadsheet is very convenient, but it is important to know that a database is not a spreadsheet. First, the similarities. Both databases and spreadsheets store data in rows and columns. Both can calculate new data based on existing data. And both can allow more than one person to view and change data. A spreadsheet may have fancy tools, but at its heart its a electronic ledger. It is a digital worksheet that can easily make nice graphs and charts. In a spreadsheet, data is stored in a cell. A cell can have raw data, like the number one or the string hello. A cell can also have a calculation. And that calculation can be standalone, or it can refer to other cells. A cell also contains formatting information like the font size or highlighting. You can format the type of data you intend to see in a cell, like specifying that the numbers have two decimal places. You can manipulate the data in a spreadsheet by visually moving or sorting the data. A database on the other hand, stores data in a record. And only stores values. Unlike a spreadsheet, formatting information like font size, highlighting, and even sort order are not stored in a database record. Usually that information is completely separate. Part of a program that displays the data, including sorting. By separating the data from the formatting, databases allow you to manipulate data without worrying that you are changing the underlying data. For example, let's say we have our address book stored as a spreadsheet. We just finished entering in information, and want to sort by last name so we can easily find an entry when we need it. So we sort by last name, but we did not sort properly and as a result, the last names were sorted but not in the context of the rest of the row. Now we have changed the data itself. John Smith became John Durand, and Noel Durand became Noel Sharma, and so on. Because the display information is separate from the data in a database, making a sorting mistake like this, does not actually change the underlying data like it does in a spreadsheet. In a database, you can give the data in a column a type. For example, you can say that the information in the birthday column is a date. In a spreadsheet, we can put anything we want in a cell. A calculation, or a word, and the spreadsheet allows that information in there. In a database, if we define a column as having a date type, it will not allow another type of data like a string. In a spreadsheet, everything you do results in data. If you do a calculation involving one cell, that calculation lives in another cell. In a database, you can do a calculation on a record and just show the calculation result without adding more data to the database. One of the biggest benefits of a database is that you can eliminate duplicate information. For example, in our address book we have eight people. Each of these people lives with another person, so the address and home phone numbers are listed twice. What happens if we make a mistake in one of the cells. In this example John and Paul are roommates living at the same address, but have two different home phone numbers. If you saw this, you would know that one of the numbers was wrong but you would not know which phone number is wrong. In a database, it is very easy to put the names in one place and say John and Paul both have that phone number. The same can be done with the address, so that when a family moves you only have to edit one database record. If a family moves, and your address book is in a spreadsheet, you have to update each address cell. Because of this, databases are more efficient at storing data. Changing data is easier and less error prone in a database. Spreadsheets are great for ledger type activities and lists, like showing bank transactions and calculating an account balance They are also good for creating charts and graphs of data. Databases are good for storing lots of information. Where you can specify that the data is a certain type. Databases separate the data from how it is displayed. Which can eliminate duplicate information and reduce data errors. Databases are also good at allowing more than one person at a time to look at and change the data without disturbing each other. Learning about databases is worth the effort.

Relational DB vs Flat Files

Wednesday, August 7, 2019 4:42 PM

When talking about relational databases versus spreadsheets, I explained a bit about what relational databases are. You may be thinking, I just put everything into a regular TXT file. If you are a power user, maybe you'll use GNU tools such awk, sed and grep to find, manage and change the data in your TXT file. There are definitely some similarities between flat files and relational databases. Both can be manipulated to show information in a new form without changing the original data. Both allow you to extract data out of certain fields. And both have powerful matching and filtering capabilities. Almost all text editors have some kind of global search and replace functionality and relational databases can do that too. Depending on your text program, you might have formatting capabilities like font size, and highlighting. Maybe you have your mother's phone number in big, bold letters so her number is very visible for when you want to call her. Sorting is also pretty tricking in a text file. A text file allows you to put anything into it. We can put a string, like Hello There, or we can put a number like 172, and our program won't complain, even if it's supposed to be the field for a date. If we want to be able to use tools to manipulate the data, we might have some rules. If we are using a tab-delimited file, the rule is that the pieces of data are separated by tabs and that they appear in a certain order. There's also nothing stopping us from using a tab to separate the data on one line, but using a comma to separate the data on the next. The program does not complain, but when we try to use tools to get the data we want, we won't be able to get our proper data. A database will separate the fields for us, a text file does not. We have to enforce a structure if we want our text file to work in a certain way. Like a spreadsheet, text editors have display information stored with the data itself. Text editors are very good for writing text, laying out text on a page, and moving text around. You can cut and paste and easily move a word, paragraph or even several pages of text in a text editor. They are designed for storing, viewing, and changing documents that consist mostly of text. A text file is sometimes called a flat file especially when comparing to a relational database. This is because a relational database is built to highlight how some data relates to other data. Flat files are great for writing and editing text. They are also good for moving lots of text around. Databases are good for storing information that you want to be a certain type. Databases separate the data from how it is displayed which can eliminate duplicate information and reduce data errors. Text files are handy, but a database is a much more powerful tool when manipulating data.

Normalizing DBs (needs improvement)

Wednesday, August 7, 2019 4:42 PM

Database normalization is a way to organize your fields and your tables. What we mean by organizing fields and tables is figuring out which fields should go with which tables. We could have a person table that has first name and last name fields. Do we put the address into the person table? Do we make the address one field? Or do we separate it into several fields by street address, city, province, country and postal code? These are the questions that database normalization helps to solve. Wikipedia has a great description of why database normalization is important. 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 using the defined relationships
For example, if a family's home address changes, you can change the address once in the Address table, instead of once for each person. In this diagram each circle is one record in an Address table, and each blue rectangle is an entry into a person table. The person table relates to the address table so that when you ask, where does Camille live, you get the proper address. There are many forms of normalization, but the accepted standard is Third Normal Form, also called 3NF. Third normal form includes first and Second Normal Form, so let's take a brief look at them. First Normal Form is achieved when each record has a single value. For example, if we had a column called Phone Number and put the mobile and home numbers together, this kind of a table is an example of a table that is not in First Normal Form. So the good news is, the example we already had is in First Normal Form. The requirements for Second Normal Form are first, that the table is in First Normal Form. The other requirement to achieve Second Normal Form is that all of the information in the table is dependent on what defines the row. This is the little complex, so I will explain it using our example. Imagine our address book was setup like this. Each phone entry had a different row in our spreadsheet. Let's look at what defines the row. If you wanted to find one particular record, what data would you need? For example, how would you find John Smith's home phone number? You need more than just the first name and last name or even more than the First Name, Last Name and Birthday just in case I know more than one person named John Smith. If you search for records containing John Smith or April 7th 1998,you'll find two records for phone number. So you would need John Smith and his birthday and phone type of home, to find the record you're looking for. Therefore the fields that defines a row are First Name, Last Name, Phone Type and Birthday. It is perfectly acceptable to have multiple fields define a row. In order to be in second normal form though, both the phone number and address must depend on all the row definition fields. One of the other fields is Phone Number. And that already depends on First Name, Last Name, Birthday, and Phone Type. That's how we figured out which fields were the defining field. The address, on the other hand, does not depend on all the defining fields. The address depends on first name, last name, and birthday. But it does not depend on the phone type. You can see this, because records that have the same First name, Last Name, and Birthday, but different phone types, all have the same address. So this table is not in second normal form because the address does not depend on all the defining fields of the table. In order to achieve Third Normal Form, the table must be in Second Normal Form, therefore it also must be in First Normal form, and all the non-defining fields are directly dependent on the defining fields. Let's take a look at our table and see what the defining field of our table are. It looks like first name and last name are defining field because given any first plus last name you can determine a record. However, let's also put Birthday as a defining field because I do know two people named Tony Cabral. One Tony Cabral is my husband, and another is my father-in-law. They have different birthdays. So let's take the remaining fields, Mobile Number, Home Number, and Address. They are all dependent on the defining fields so this table is in Second Normal Form. However, the Home Phone Number and Address are dependent on each other because they're both associated with the house. If there are different people at that address, the phone number is still the same. So, the Home Phone Number is indirectly dependent on the First Name, Last Name and Birthday. In order to have this table in Third Normal Form we need to split it out into two tables. The first table will be the First Name, Last Name, Mobile Number and Birthday. The defining fields of that table are First Name, Last Name and Birthday. There are no records of multiple

entries, so it's in First Normal Form. It's in Second Normal Form because the remaining field, Mobile Number depends on the defining fields. And it is in third normal form because it directly depends on the defining fields. The other fields, Address and Home Phone Number get put into a second table. There are records with multiple entries, so it is indeed in First Normal Form. It is in Second Normal Form, because the remaining field, Home Phone Number, depends on the defining field, which is address, and it is in third normal form because home phone number directly depends on the address. To complete our address book, we need to somehow link the two tables. Denormalization is when you intentionally ignore some of the normalization rules. This means that you bring back redundancy and dependency. Usually you do this so that some of your more common database operations are faster The point to remember is that normalization will have the best data, but sometimes you want to take a risk for another gain, like performance. Most of the time, databases are normalized to provide the optimal conditions for working with data. A normalized database has the fewest chances of problems with redundant data. Denormalization is a trick you can use to make things faster. And is used only when it is absolutely necessary. To learn more about normalization in greater detail, take a look at the foundations of programming databases course here on lynda.com, which goes over database normalization. It covers First, Second and Third Normal Form and when to denormalize your data.

- **First Normal Form** – The information is stored in a relational table with each column containing atomic values. There are no repeating groups of columns.

- **Second Normal Form** – The table is in first normal form and all the columns depend on the table's primary key.

- **Third Normal Form** – the table is in second normal form and all of its columns are not transitively dependent on the primary key

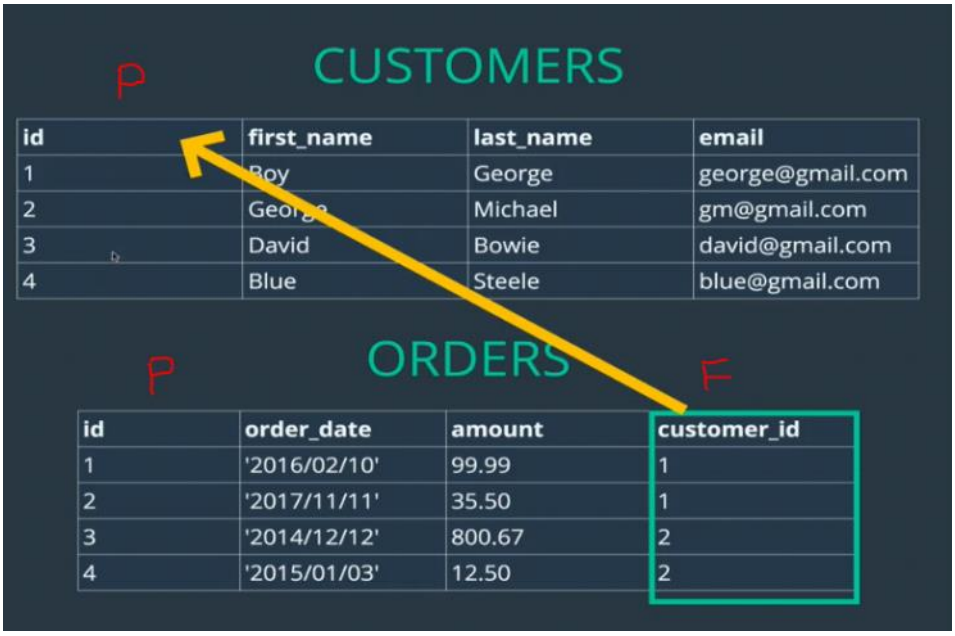
  Read more: https://www.essentialsql.com/get-ready-to-learn-sql-database-normalization-explained-in-simple-english/

Relationships (keys, cardinality, etc.)

Saturday, November 16, 2019 4:08 PM

Primary Key
a **primary key** is a single field or combination of fields that uniquely defines a record. None of the fields that are part of the **primary key** can contain a NULL value. A table can have only one **primary key**.

Foreign Key
A **foreign key** is a field (or a set of fields) in a table that uniquely identifies a row of another table. The table in which the **foreign key** is defined is called the "child table" and it (often) refers to the primary **key** in the parent table.



In any case, there are three possible values for the "Key" attribute:

1. `PRI`
2. `UNI`
3. `MUL`

The meaning of `PRI` and `UNI` are quite clear:

- `PRI` => primary key
- `UNI` => unique key

The third possibility, `MUL`, (which you asked about) is basically an index that is neither a primary key nor a unique key. The name comes from "multiple" because multiple occurrences of the same value are allowed. Straight from the MySQL documentation:

> If `Key` is `MUL`, the column is the first column of a nonunique index in which multiple occurrences of a given value are permitted within the column.

There is also a final caveat:

> If more than one of the Key values applies to a given column of a table, Key displays the one with the highest priority, in the order `PRI`, `UNI`, `MUL`.

As a general note, the MySQL documentation is quite good. When in doubt, check it out!

```
+-------------+--------------+------+-----+---------+----------------+
| Field       | Type         | Null | Key | Default | Extra          |
+-------------+--------------+------+-----+---------+----------------+
| id          | int(11)      | NO   | PRI | NULL    | auto_increment |
| order_date  | date         | YES  |     | NULL    |                |
| amount      | decimal(8,2) | YES  |     | NULL    |                |
| customer_id | int(11)      | YES  | MUL | NULL    |                |
+-------------+--------------+------+-----+---------+----------------+
```

Relationship types

You share many relationships with members of your family. For instance, you and your mother are related. You have only one mother, but she may have several children. You and your siblings are related—you may have many brothers and sisters and, of course, they'll have many brothers and sisters as well. If you're married, both you and your spouse have a spouse—each other—but only one at a time. Database relationships are very similar in that they're associations between tables. There are three types of relationships:

- **One-to-one:** Both tables can have only one record on either side of the relationship. Each primary key value relates to only one (or no) record in the related table. They're like spouses—you may or may not be married, but if you are, both you and your spouse have only one spouse. Most one-to-one relationships are forced by business rules and don't flow naturally from the data. In the absence of such a rule, you can usually combine both tables into one table without breaking any normalization rules.

- **One-to-many:** The primary key table contains only one record that relates to none, one, or many records in the related table. This relationship is similar to the one between you and a parent. You have only one mother, but your mother may have several children.
- **Many-to-many:** Each record in both tables can relate to any number of records (or no records) in the other table. For instance, if you have several siblings, so do your siblings (have many siblings). Many-to-many relationships require a third table, known as an associate or linking table, because relational systems can't directly accommodate the relationship.

Common DB acronyms

Wednesday, August 7, 2019 4:42 PM

There are a lot of acronyms in the database world. You can guess that DB means Database and that a database administrator is sometimes shortened to DBA. You might read articles that talk about a DBMS, but that just means a database management system. It's just the database program you are using. For example, MySQL, Postgres, Oracle or Microsoft SQL Server all are database management systems. In fact, as they are all relational databases, they are all relational database management systems or RDBMS. You may have already heard me talking about some acronyms. For example, SQL stands for structured query language, and it is how you interact with the database. It is sometimes pronounced sequel, and that's the preferred pronunciation for Microsoft SQL Server. There are some less obvious acronyms used in the database world, however. That are important to know about. There are two acronyms used when talking about changing information in a database. One is DDL or data definition language. Data definition language is the set of SQL commands you can use, to define tables and databases. DDL usually involves changing how the data is stored. You can use DDL to define new fields or change how a field works. You can even use DDL to get rid of an entire table or an entire database, so use DDL with caution.

Another acronym used a lot often in comparison to DDL is DML. DML stands for data manipulation language and is how you manipulate data or change it. Think of a database like you closet and your clothes are the data. DML is how you move clothes around to different places, removing them from the closet when you wear them, and then putting them back in the closet. DDL is how you organize your closet. Where the shirts go versus where the shoes go, and even whether or not you store wrapping paper or suitcases alongside your clothes. Just like wearing clothes, you probably will use DML to manipulate data frequently. DML is how you put data into a table, how you change it, and even how you delete data from a table. Using DDL to change how things are defined, is used much less frequently. Just like you only reorganize your closet, once in a while. People may describe an application as OLAP. OLAP stands for online analytical processing, with the key part being analytical. Think of this as an application that does a lot of reports. This is in contrast to OLTP which is online transactional processing. The key here is transactional. We are not going to go into what a transaction is here, so think of something like a sales transaction. An application that does a sales transaction has different needs from an application that reports on those same sales transactions. The sales themselves would need OLTP features and the reporting application to figure out how many widgets were sold in Germany needs another set of features, the OLAP features. Databases will say they are ideal for OLTP or OLAP applications and point out OLAP and OLTP features. You may run across the acronym CRUD, often pronounced CRUD. CRUD stands for create, read, update, and delete. And it refers to a set of ways to interact with your data. For example, you can create an address book entry, read an address book entry, update an address book entry or delete an address book entry. But CRUD goes a little deeper than just explaining how to interact with your data. If you write an application, and it cannot create, read, update, and delete, then your application is probably not complete. There are exceptions of course. For example, logging software probably does not delete or update many entries. But when writing logging software you would at least consider those possibilities. An address book would hardly be complete if you could not create entries, read entries, update entries and delete entries. Or think of an online store. Customers need to be able to create and delete. That is, put items into their virtual shopping cart and take them out. They also need to be able to read entries. Looking at all the details about an item before deciding to buy it. And they need to update information. Maybe they want to order more than one of a particular item or maybe they need to update their shipping address. CRUD represents one way of thinking about how you're going to interact with your data. It is just the basic level and how you decide to change data depends on the situation. For example, if you change an address because you made a typo, you would update the existing address. If an entire family moves, you would also update the address to the newer one. But if only one of the people in a family moves, you would keep the original address record for the remaining people and add a new address record for the one person who moved. You will come up with

different processes on what to do in certain cases, but in the end it all comes down to creating, reading, updating, and deleting information. You may find job descriptions that mention needing CRUD abilities, or you may come across software that implements CRUD for you. So it is critical to know that Crud means create, read, update and delete and it's how you interact with the database. We have learned about basic and intermediate acronyms in the database world. These acronyms are used throughout databases whether you are using MySQL or a different database system

Understanding MySql Terminoloy: table, row, colums, and field

Wednesday, August 7, 2019 4:43 PM

Let's take a look at some basic database terminology. We have been looking at an example for an address book. The address book contains names, phone numbers, addresses, and birthdays. In the context of a database, each of these is called a field or a column. You can use field or column interchangeably for description. Mobile number, is an example of a field or column. If mobile number is a column, you may have already guessed that we can use the word row to describe an entire entry. In fact, MySQL returns results with information of how many rows were returned. You can also call a row a record. So, you might talk about Paul Thompson's records. Or finding all records whose birthdays are in July. You can use rows and records interchangeably when talking about entries in a database. A collection of records makes up a table. A table stores the records that are described by the fields in their row. We may have a table that has name, mobile phone number and birthday as fields in the table. And we may have 0, 1, 10, 500, or more records in that table. We have talked about DDL, which is the Data Definition Language. DDL is how we define what fields are in a table. DML, or data manipulation language, is how we create, update, and remove records from a table. When we use DDL or DML on a table, we are really just sending statements to the table. When we read or search for information we are also sending statements to a table. Sending statements to a table is called querying the table, and the commands used are called queries. You can pronounce it either query or querry. You might have to write a query or optimize a query. When we normalized our data, we put all the information that relates only to one person and not anyone else into one table. Name, birthday, and mobile phone number. We put shared information, like address and home phone number in another table and related the tables to each other. This is how relational databases work, and in fact, in MySQL, a database is just a container that holds tables. This is not merely a convenient analogy, on Windows, Mac OS X, and Linux and Unix system, a MySQL database is just a folder or directory. All the information inside a database is stored underneath this folder or directory. A synonym for database is schema. And the plural of schema is schemata. Or you can just say schemas. When people talk about creating a schema or changing a schema, what they mean is they are changing the definition of what is in the database. Usually, they are changing a table definition. A database is a collection of tables. It can also be called a schema. A table is made up of records, which are also the rows of a table. The columns of a table are called fields. The way to interact with data is with queries.

Finding information about your database with SHOW

Monday, August 19, 2019 7:26 PM

Most Database systems have a way to find out what the schema is. That is, what Databases, Tables and Columns already exist. There is usually a way to query a special table to get that information. However, this presents a bit of a problem for someone new to Databases. How do you find out what your schema is like, if you do not yet know how to query? One of the greatest things about MySQL is that it is very user friendly. There is a way to write queries to figure out what the schema is like. But MySQL also offers easier options. I have MySQL installed here on my system, and if you don't have it set up yet. Make sure to get it installed for your platform. If you need some help getting MySQL installed on your system, check out installing Apache, MySQL, and PHP here in the lynda.com library. I'm now going to log in to the MySQL system. I type the command's name, MySQL and then give it a user with a dash u option. And then use the dash P option with nothing after it. It haven't asked me for a password. I type the password but you can't see what I'm typing and then I hit enter. And I am at the MySQL prompt, which you can tell because it says Mysql at the beginning of the line right where my cursor is. When you are on the MySQL command prompt, you may not know what to do first. What databases do you have? MySQL has made this easy, with the SHOW DATABASES command. When you run show databases, MySQL will return with a list of Databases on the system. From there, you might want to know about the tables that make up a certain database. Let's take a look at one of the special Databases that comes with MySQL called information Schema. To find out the tables in that Database, we can run SHOW TABLES FROM information_schema. And then we can see a list of all the tables in the information schema database. There's a lot of them. There's about 59 rows in set, which means there are 59 of them. There is another command called USE, which sets your default database. In order to see tables from a particular Database, we had to say show Tables from the Database name. But if we have a default database set, we do not have to specify a database. Let's set out default database to information schema. Now we can just type show tables and we get the same list of 59 rows that we got before. If SHOW_DATABASES is how you see databases on the system, and SHOW_TABLES is how you see tables, you might guess how you could see the columns in a table. If you guessed that there was a command called show columns, you're absolutely right. Let's find a table to look at. How about this Schemata Table. Remember a Schema is a synonym for Database and Schemata is the plural of schema. So the Schemata Table should have information about the Databases. Let's select the database first with USE information_schema, and then we can run show columns from SCHEMATA. Now we can see that there are five rows returned, which means there are five columns in the Schemata Table. What if we want to see columns from a table that is not in the default database? MySQL has a syntax called dot notation, which allows you to refer to another database without changing your current database. Let's go to the MySQL database with USE mysql, and try to SHOW COLUMNS FROM SCHEMATA again. You'll see that it says, Table mysql.schemata doesn't exist. Notice that it said mysql.schemata. That is dot notation in action. All you have to do is append the Database Name in front of the Table Name, and separate the names with a dot. So using that same format, we can run, SHOW COLUMNS FROM information_schema.SCHEMATA. And now we see the columns in that table. There are several other SHOW commands, which we will not go into the details of right now. But there are two resources you can use to find out more. The first is the command line help, or the help inside whatever program you're using to connect to MySQL. You can use the command help to get a little bit of help about a command. For example, help SHOW shows you the description of what SHOW does. SHOW has many forms that provide information about Databases, Tables, Columns or status information about the Server. Then, it shows a list of the commands you can get further help on. For example, you can do help SHOW DATABASES. And the end of help SHOW, you can see a URL. That URL goes to MySQL manual page for that particular command. The MySQL manual is very good and very detailed, and I absolutely recommend it as a supplemental source of information. It is not something you would read through like a tutorial. But it is extremely useful as a reference manual. Now you know how to see what databases exist, how to select a default database, and how to see what Tables and Columns exist. You also know how to use the command line help function, and that the MySQL online manual is a great reference tool. And you learned about dot notation which you can use in many parts of MySQL, to refer to a table outside of the default database.

Creating DBs and Tables

Monday, October 21, 2019 8:58 PM

Start mySQL CLI for Goorm IDE:
mysql-ctl cli;

Show DB
SHOW databases;

Create DB
CREATE database <name>;

Delete DB
DROP database <name>;

Using DB
USE <database name>;

This command will show the current db in use.
SELECT database();

mySQL Datatypes
mySQL datatypes: http://www.mysqltutorial.org/mysql-data-types.aspx

This will create a table inside a db
```
CREATE TABLE <table name>
(
	columnName dataType,
	columnName dataType,
	Sex varchar(2)
);
```

Shows the tables inside a db
SHOW TABLES;

This will show the columns in a table. Both of these commands have the same result
SHOW COLUMNS FROM <table name>;
DESC <table name>;

Delete a table
DROP TABLE <table name>;

Inserting data

Monday, October 21, 2019 8:59 PM

```
INSERT INTO <table>(<columns>, <columns>)
VALUES ("values", "values");

INSERT INTO <table>(name, age)
VALUES (Tom, 7);

You can also insert multiple values
INSERT INTO <table>(name, age)
VALUES (Tom, 7),
       (Rick, 3),
       (Sandy, 9);
```

Default Values

Monday, October 21, 2019 8:59 PM

Creating columns with
CREATE TABLE employees (name VARCHAR(100) DEFAULT 'unnamed',
age INT DEFAULT 1);

This will provide a default values for specific columns
INSERT INTO employees () VALUES ();
INSERT INTO employees (name) VALUES ("rob");
INSERT INTO employees (age) VALUES ("12");

Creating columns with no null values allowed
CREATE TABLE students (name VARCHAR(100) NOT NULL DEFAULT 'unnamed',
age INT DEFAULT 1);
INSERT INTO students (name) VALUES (); This code will cause an error since NULL values aren't allowed
INSERT INTO students (name, age) VALUES ("Richard" ,NULL); This code is fine as long as you explicitly state NULL for columns that allow null values

Primary Keys

Monday, October 21, 2019 8:59 PM

Assigning a **primary key** is essential in relational database. It serves as a unique identifier for a specific row

```
CREATE TABLE employees (
                    employee_id INT NOT NULL,
                    name VARCHAR(100) DEFAULT 'unnamed',
                age INT,
                    PRIMARY KEY (employee_id));

INSERT INTO employees(employee_id, name, age) VALUES
                (1, "Fred", 23),
                (2, "Deepak", 13);

INSERT INTO employees(employee_id, name, age) VALUES
                (1, "Rob", 43); This will generate an error because it has the same primary
                key


AUTO INCREMENT FEATURE FOR PRIMARY KEYS----------
CREATE TABLE employees (
                    employee_id INT NOT NULL AUTO_INCREMENT,
                    name VARCHAR(100) DEFAULT 'unnamed',
                age INT,
                    PRIMARY KEY (employee_id));

Notice that we don't specify the primary key field since we have assigned an auto increment feature to
the employee_id (primary key)
INSERT INTO employees(name, age) VALUES
                ("Fred", 23),
                ("Deepak", 13),
                ("Chicky", 23),
                ("Hamstry", 43),
                ("Gordy", 3);



This is an exercise----------
Notice that we have the PRIMARY KEY keyword here. This is the shorthand method.
CREATE TABLE employees (
                    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
                    last_name VARCHAR(100) NOT NULL,
                    first_name VARCHAR(100) NOT NULL,
                    middle_name VARCHAR(100),
                age INT NOT NULL,
                    current_status VARCHAR(100) NOT NULL DEFAULT "employed");

INSERT INTO employees(last_name, first_name, age) VALUES
                ("Dicky", "Jody", 43),
                ("John", "Hopkin", 53),
                ("James", "Puke", 77);
```

Foreign Keys

Saturday, November 16, 2019 4:17 PM

A **foreign key** is a column or group of columns in a relational database table that provides a link between data in two tables. It acts as a cross-reference between tables because it references the primary **key** of another table, thereby establishing a link between them.

Read more: https://dev.mysql.com/doc/mysql-tutorial-excerpt/5.7/en/example-foreign-keys.html

```
CREATE DATABASE customers_and_orders;

CREATE TABLE customers
(
id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
first_name VARCHAR(100),
last_name VARCHAR(100),
email VARCHAR(100)
);

CREATE TABLE orders
(
id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
order_date DATE,
amount DECIMAL(8,2),
customer_id INT,
FOREIGN KEY (customer_id) REFERENCES customers(id)
);

INSERT INTO customers (first_name, last_name, email)
VALUES ('Boy', 'George', 'george@gmail.com'),
    ('George', 'Michael', 'gm@gmail.com'),
    ('David', 'Bowie', 'david@gmail.com'),
    ('Blue', 'Steele', 'blue@gmail.com'),
    ('Bette', 'Davis', 'bette@aol.com');

INSERT INTO orders (order_date, amount, customer_id)
VALUES ('2016/02/10', 99.99, 1),
    ('2017/11/11', 35.50, 1),
    ('2014/12/12', 800.67, 2),
    ('2015/01/03', 12.50, 2),
    ('1999/04/11', 450.25, 5);
```

SQL Warnings

Monday, October 21, 2019 8:59 PM

MySQL Warnings Code
SHOW WARNINGS is a diagnostic statement that displays information about the conditions (errors, warnings, and notes) resulting from executing a statement in the current session. Warnings are generated for DML statements such as INSERT, UPDATE, and LOAD DATA as well as DDL statements such as CREATE TABLE and ALTER TABLE.

DESC cats;

Try Inserting a cat with a super long name:

```
INSERT INTO cats(name, age)
VALUES('This is some text blah blah blah blah blah text text text something about cats lalalalal meowwwwwwwwww', 10);
```

Then view the warning:

SHOW WARNINGS;

Try inserting a cat with incorrect data types:

INSERT INTO cats(name, age) VALUES('Lima', 'dsfasdfdas');

Then view the warning:

SHOW WARNINGS;

Aliases

Monday, October 21, 2019 7:35 PM

Introduction to Aliases

```
SELECT cat_id AS id, name FROM cats;
```

Notice that if there's space between the words, we enclose them by using quotation marks.

```
SELECT name AS 'cat name', breed AS 'kitty breed' FROM cats;
```

```
DESC cats;
```

Update

Monday, October 21, 2019 7:42 PM

Main syntax:
UPDATE <table> **SET** <column=value> WHERE <column=value>

Change tabby cats to shorthair:
UPDATE cats SET breed='Shorthair' WHERE breed='Tabby';

Another update:
UPDATE cats SET age=14 WHERE name='Misty';

UPDATE cats SET name="Rohelyo" WHERE name="Ringo";

Delete

Monday, October 21, 2019 8:00 PM

What is the DELETE Keyword?

The SQL DELETE command is used to delete rows that are no longer required from the database tables. It deletes the whole row from the table. Delete command comes in handy to delete temporary or obsolete data from your database. The DELETE command can delete more than one row from a table in a single query. This proves to be advantages when removing large numbers of rows from a database table.

Once a row has been deleted, it cannot be recovered. It is therefore strongly recommended to make database backups before deleting any data from the database. This can allow you to restore the database and view the data later on should it be required.

Delete command syntax

The basic syntax of the delete command is as shown below.

DELETE FROM `table_name` [WHERE condition];

ON DELETE CASCADE

SQL deletes the rows in the child table that is corresponding to the row deleted from the parent table. ON DELETE SET NULL : SQL Server sets the rows in the child table to NULL if the corresponding rows in the parent table are deleted.

Ex:

```
CREATE TABLE customers(
   id INT AUTO_INCREMENT PRIMARY KEY,
   first_name VARCHAR(100),
   last_name VARCHAR(100),
   email VARCHAR(100)
);

CREATE TABLE orders(
   id INT AUTO_INCREMENT PRIMARY KEY,
   order_date DATE,
   amount DECIMAL(8,2),
   customer_id INT,
   FOREIGN KEY(customer_id)
      REFERENCES customers(id)
      ON DELETE CASCADE
);

INSERT INTO customers (first_name, last_name, email)
VALUES ('Boy', 'George', 'george@gmail.com'),
      ('George', 'Michael', 'gm@gmail.com'),
      ('David', 'Bowie', 'david@gmail.com'),
      ('Blue', 'Steele', 'blue@gmail.com'),
      ('Bette', 'Davis', 'bette@aol.com');

INSERT INTO orders (order_date, amount, customer_id)
VALUES ('2016/02/10', 99.99, 1),
      ('2017/11/11', 35.50, 1),
      ('2014/12/12', 800.67, 2),
      ('2015/01/03', 12.50, 2),
      ('1999/04/11', 450.25, 5);
```

DELETE FROM customers WHERE email = 'george@gmail.com';

Without the ON DELETE CASCADE exception

```
mysql> DELETE FROM customers WHERE email = 'george@gmail.com';
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails (`customers_and_orders`.`ord
ers`, CONSTRAINT `orders_ibfk_1` FOREIGN KEY (`customer_id`) REFERENCES `customers` (`id`))
```

Select

Monday, October 21, 2019 7:22 PM

CODE: Official Introduction to SELECT

Various Simple SELECT statements:

```
SELECT * FROM cats;
SELECT name FROM cats;
SELECT age FROM cats;
SELECT cat_id FROM cats;
SELECT name, age FROM cats;
SELECT cat_id, name, age FROM cats;
SELECT age, breed, name, cat_id FROM cats;
SELECT cat_id, name, age, breed FROM cats;
```

DISTINCT

Thursday, October 24, 2019 10:01 PM

The SELECT DISTINCT statement is used to return only distinct (different) values.

```
SELECT DISTINCT author_lname FROM books;
```

```
SELECT DISTINCT CONCAT (author_lname, ' ',  author_fname) AS 'Full name' FROM books;
```

Where

Saturday, September 21, 2019 4:52 PM

A WHERE Clause

SELECT <columns> FROM <table> WHERE <condition>;

Equality Operator

Find all rows that a given value matches a column's value.

SELECT <columns> FROM <table> WHERE <column name> = <value>;

Examples:

SELECT * FROM contacts WHERE first_name = "Andrew";

SELECT first_name, email FROM users WHERE last_name = "Chalkley";

SELECT name AS "Product Name" FROM products WHERE stock_count = 0;

SELECT title "Book Title" FROM books WHERE year_published = 1999;

Inequality Operator

Find all rows that a given value doesn't match a column's value.

SELECT <columns> FROM <table> WHERE <column name> != <value>;

SELECT <columns> FROM <table> WHERE <column name> <> <value>;

The not equal to or inequality operator can be written in two ways != and <>. The latter is *less* common.

Examples:

SELECT * FROM contacts WHERE first_name != "Kenneth";

SELECT first_name, email FROM users WHERE last_name != "L:one";

SELECT name AS "Product Name" FROM products WHERE stock_count != 0;

SELECT title "Book Title" FROM books WHERE year_published

The following are the keywords that you may use after the WHERE clause

SELECT last_name FROM phone_book WHERE last_name

"XXX" AND "XXX";

"XXX" OR "XXX";

IN (7.99, 9.99, 11.99);

BETWEEN <lesser value> AND <greater value>;

LIKE <%pattern%>

IS NULL

IS NOT NULL;

ORDER BY

Thursday, October 24, 2019 10:12 PM

```
SELECT author_lname FROM books;

SELECT author_lname FROM books ORDER BY author_lname;

SELECT title FROM books;

SELECT title FROM books ORDER BY title;
SELECT author_lname FROM books ORDER BY author_lname DESC;

SELECT released_year FROM books;

SELECT released_year FROM books ORDER BY released_year;

SELECT released_year FROM books ORDER BY released_year DESC;

SELECT released_year FROM books ORDER BY released_year ASC;

SELECT title, released_year, pages FROM books ORDER BY released_year;

SELECT title, pages FROM books ORDER BY released_year;

SELECT title, author_fname, author_lname
FROM books ORDER BY 2;

SELECT title, author_fname, author_lname
FROM books ORDER BY 3;

SELECT title, author_fname, author_lname
FROM books ORDER BY 1;

SELECT title, author_fname, author_lname
FROM books ORDER BY 1 DESC;

SELECT author_lname, title
FROM books ORDER BY 2;

SELECT author_fname, author_lname FROM books
ORDER BY author_lname, author_fname;
```

LIMIT

Thursday, October 24, 2019 10:25 PM

```
SELECT title FROM books LIMIT 3;

SELECT title FROM books LIMIT 1;

SELECT title FROM books LIMIT 10;

SELECT * FROM books LIMIT 1;

SELECT title, released_year FROM books
ORDER BY released_year DESC LIMIT 5;

SELECT title, released_year FROM books
ORDER BY released_year DESC LIMIT 1;

SELECT title, released_year FROM books
ORDER BY released_year DESC LIMIT 14;

SELECT title, released_year FROM books
ORDER BY released_year DESC LIMIT 0,5;

SELECT title, released_year FROM books
ORDER BY released_year DESC LIMIT 0,3;

SELECT title, released_year FROM books
ORDER BY released_year DESC LIMIT 1,3;

SELECT title, released_year FROM books
ORDER BY released_year DESC LIMIT 10,1;

SELECT * FROM tbl LIMIT 95,18446744073709551615;

SELECT title FROM books LIMIT 5;

SELECT title FROM books LIMIT 5, 123219476457;

SELECT title FROM books LIMIT 5, 50;
```

Relational Operators

Sunday, October 13, 2019 7:10 PM

Relational Operators

There are several relational operators you can use:

- < less than
- <= less than or equal to
- > greater than
- >= greater than or equal to

These are primarily used to compare *numeric* and *date/time* types.

SQL Used

SELECT <columns> FROM <table> WHERE <column name> < <value>;
SELECT <columns> FROM <table> WHERE <column name> <= <value>;
SELECT <columns> FROM <table> WHERE <column name> > <value>;
SELECT <columns> FROM <table> WHERE <column name> >= <value>;
Examples:
SELECT first_name, last_name FROM users WHERE date_of_birth < '1998-12-01';
SELECT title AS "Book Title", author AS Author FROM books WHERE year_released <= 2015;
SELECT name, description FROM products WHERE price > 9.99;
SELECT title FROM movies WHERE release_year >= 2000;

AND & OR

Sunday, October 13, 2019 7:19 PM

You're not restricted to just using one condition, you can test rows of information against multiple conditions. You can choose whether you retrieve rows that match both of your conditions or either of them.

- Teacher's Notes
- Questions?3
- Video Transcript
- Downloads

SQL Used

You can compare multiple values in a WHERE condition. If you want to test that *both* conditions are true use the AND keyword, or *either* conditions are true use the OR keyword.

SELECT <columns> FROM <table> WHERE <condition 1> AND <condition 2> ...;
SELECT <columns> FROM <table> WHERE <condition 1> OR <condition 2> ...;

Examples:

SELECT username FROM users WHERE last_name = "Chalkley" AND first_name = "Andrew";
SELECT * FROM products WHERE category = "Games Consoles" AND price < 400;
SELECT * FROM movies WHERE title = "The Matrix" OR title = "The Matrix Reloaded" OR title = "The Matrix Revolutions";
SELECT country FROM countries WHERE population < 1000000 OR population > 100000000;

Filtering Dates

Sunday, October 13, 2019 7:25 PM

Relational Operators

There are several relational operators you can use:

- < less than
- <= less than or equal to
- > greater than
- >= greater than or equal to

These are primarily used to compare *numeric* and *date/time* types.

SELECT <columns> FROM <table> WHERE <column name> < <value>;
SELECT <columns> FROM <table> WHERE <column name> <= <value>;
SELECT <columns> FROM <table> WHERE <column name> > <value>;
SELECT <columns> FROM <table> WHERE <column name> >= <value>;

Examples:

SELECT first_name, last_name FROM users WHERE date_of_birth < '1998-12-01';
SELECT title AS "Book Title", author AS Author FROM books WHERE year_released <= 2015;
SELECT name, description FROM products WHERE price > 9.99;
SELECT title FROM movies WHERE release_year >= 2000;

IN keyword (Shorthand usage)

Monday, October 14, 2019 8:58 PM

The SQL IN Operator

The IN operator allows you to specify multiple values in a WHERE clause.
The IN operator is a shorthand for multiple OR conditions.

SQL Used

SELECT <columns> FROM <table> WHERE <column> IN (<value 1>, <value 2>, ...);
Examples:
SELECT name FROM islands WHERE id IN (4, 8, 15, 16, 23, 42);
SELECT * FROM products WHERE category IN ("eBooks", "Books", "Comics");
SELECT title FROM courses WHERE topic IN ("JavaScript", "Databases", "CSS");
SELECT * FROM campaigns WHERE medium IN ("email", "blog", "ppc");
To find all rows that are not in the set of values you can use NOT IN.
SELECT <columns> FROM <table> WHERE <column> NOT IN (<value 1>, <value 2>, ...);
Examples:
SELECT answer FROM answers WHERE id IN (7, 42);
SELECT * FROM products WHERE category NOT IN ("Electronics");
SELECT title FROM courses WHERE topic NOT IN ("SQL", "NoSQL");

Challenge Task 1 of 2

We have an e-commerce database. Inside the products table we have the columns of id, name, description and price.
Without using the OR keyword, find all products with the price of 7.99, 9.99 or 11.99.

SELECT * FROM products WHERE price IN (7.99, 9.99, 11.99);

BETWEEN keyword for specifying ranges

Monday, October 14, 2019 9:09 PM

Again, writing shorter queries can help with readability and tracking down mistakes in our SQL code. In this video we'll go over the syntax of handling ranges of values in SQL.

SQL Used

SELECT <columns> FROM <table> WHERE <column> BETWEEN <lesser value> AND <greater value>;
Examples:
SELECT * FROM movies WHERE release_year BETWEEN 2000 AND 2010;
SELECT name, description FROM products WHERE price BETWEEN 9.99 AND 19.99;
SELECT name, appointment_date FROM appointments WHERE appointment_date BETWEEN "2015-01-01" AND "2015-01-07;

NOT BETWEEN
I did a bit of playing around in the SQL Playground and the following query seems to work fine:

SELECT title, author FROM books WHERE first_published NOT BETWEEN 1800 AND 1899;

In the e-commerce database we have the products table with the columns id, name, description and price.
Find all the products in the database with the price including and between 10.99 and 12.99.

SELECT * FROM products WHERE price BETWEEN 10.99 AND 12.99;

We're back in our sports team database with the results table. The columns are id, home_team, home_score, away_team, away_score and played_on.
There are 30 days in September. Find all the games played in the results table in September 2015.

SELECT * FROM results WHERE played_on BETWEEN "2015-09-01" AND "2015-09-30";

LIKE (For wildcard search)

Monday, October 14, 2019 9:23 PM

Finding Data that Matches a Pattern

Placing the percent symbol (%) any where in a string in conjunction with the LIKE keyword will operate as a wildcard. Meaning it can be substituted by any number of characters, including zero!

Note: You need WHERE before using LIKE

SELECT <columns> FROM <table> WHERE <column> LIKE <pattern>;
Examples:
SELECT title FROM books WHERE title LIKE "Harry Potter%Fire";
SELECT title FROM movies WHERE title LIKE "Alien%";
SELECT * FROM contacts WHERE first_name LIKE "%drew";
SELECT * FROM books WHERE title LIKE "%Brief History%";

PostgreSQL Specific Keywords

LIKE in PostgreSQL is case-sensitive. To case-insensitive searches do ILIKE.
SELECT * FROM contacts WHERE first_name LIKE "%drew";

In the e-commerce database we have a products table. The columns are id, name, description and price.
Find all the products where the pattern 't-shirt' can be found anywhere in the product name.

SELECT * FROM products WHERE name LIKE "%t-shirt%";

In the users table we have the columns id, username, password, first_name and last_name.
Find all users with the first name starting with the letter "L".

SELECT * FROM users WHERE first_name LIKE "L%";

More samples

SELECT title, author_fname FROM books WHERE author_fname LIKE '%da%';

SELECT title, author_fname FROM books WHERE author_fname LIKE 'da%';

SELECT title FROM books WHERE title LIKE 'the';

SELECT title FROM books WHERE title LIKE '%the';

SELECT title FROM books WHERE title LIKE '%the%';

SELECT title, stock_quantity FROM books;

SELECT title, stock_quantity FROM books WHERE stock_quantity LIKE '____'; //an underscore represent a character

SELECT title, stock_quantity FROM books WHERE stock_quantity LIKE '__';

(235)234-0987 LIKE '(___)___-____'

SELECT title FROM books;

SELECT title FROM books WHERE title LIKE '%\%%'

SELECT title FROM books WHERE title LIKE '%_%'

NULL

Monday, October 14, 2019 9:35 PM

Filtering Out or Finding Missing Information using the NULL keyword

What is a NULL Value?

A field with a NULL value is a field with no value.
If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.
Note: A NULL value is different from a zero value or a field that contains spaces. A field with a NULL value is one that has been left blank during record creation!

How to Test for NULL Values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.
We will have to use the IS NULL and IS NOT NULL operators instead.

IS NULL Syntax

SELECT *column_names*
FROM *table_name*
WHERE *column_name* IS NULL;

IS NOT NULL Syntax

SELECT *column_names*
FROM *table_name*
WHERE *column_name* IS NOT NULL;

We're back on the smartphone, but our phone_book is a mess. There's a phone_book table but there's missing information in a couple of the columns.
The phone_book has the following columns id, first_name, last_name and phone.
Find all contacts in the phone_book where the phone number is missing so we can go and ask them for their number.

```
SELECT * FROM phone_book WHERE phone IS NULL;
```

We're still using the phone_book, with the columns id, first_name, last_name and phone.
Imagine we're implementing the autocomplete feature for a search facility on the phone where a user can start typing a last name and suggestions will appear. Write a query to retrieve all values from the last name column where the last name value is present. Only retrieve the last_name column.

```
SELECT last_name FROM phone_book WHERE last_name IS NOT NULL;
```

IFNULL(SUM(column1), 0)
1st param is the column being check if null
2nd param will replace the value of the column if true or null

Running SQL files

Wednesday, October 23, 2019 6:32 PM

```
CREATE TABLE cats
(
cat_id INT NOT NULL AUTO_INCREMENT,
name VARCHAR(100),
age INT,
PRIMARY KEY(cat_id)
);

mysql-ctl cli

use cat_app;

source first_file.sql

DESC cats;



INSERT INTO cats(name, age)
VALUES('Charlie', 17);

INSERT INTO cats(name, age)
VALUES('Connie', 10);

SELECT * FROM cats;

source testing/insert.sql
```

CONCAT - string function

Wednesday, October 23, 2019 6:57 PM

CONCAT (x, y, z);

CONCAT(colmun, "texthere" , ' ', column)

SELECT CONCAT('Hello', 'World' , ' ', '!'); NOTE That you need to use select to run this function
SELECT CONCAT(author_fname, ' ', author_lname) AS 'Full name' FROM books;

Use CONCAT_WS if you are combining a lot of data, and you would like to use a specific character separator such as - or space between the items being concatenated.

SELECT CONCAT_WS (' - ', author_fname, author_lname) AS PUTA FROM books;
More string function in mySQL
Source: https://dev.mysql.com/doc/refman/8.0/en/string-functions.html

SUBSTRING - string function

Wednesday, October 23, 2019 7:18 PM

```
SELECT SUBSTRING('Hello World', -3);

SELECT SUBSTRING('Hello World', -7);

SELECT title FROM books;

SELECT SUBSTRING("Where I'm Calling From: Selected Stories", 1, 10);

SELECT SUBSTRING(title, 1, 10) FROM books;

SELECT SUBSTRING(title, 1, 10) AS 'short title' FROM books;

SELECT SUBSTR(title, 1, 10) AS 'short title' FROM books;
```

Here we are combining the CONCAT and the SUBSTRING functions

```
SELECT CONCAT
   (
      SUBSTRING(title, 1, 10),
      '...'
   )
FROM books;

source book_code.sql

SELECT CONCAT
   (
      SUBSTRING(title, 1, 10),
      '...'
   ) AS 'short title'
FROM books;

source book_code.sql
```

REPLACE - string function

Wednesday, October 23, 2019 9:01 PM

The replace function is case sensitive! The REPLACE() function, as well as the other string functions, only change the query output, they don't affect the actual data in the database.

Syntax
SELECT REPLACE ('texthere', 'textTobeReplaced', 'character/s')

Examples

```
SELECT REPLACE('Hello World', 'o', '*');

SELECT REPLACE('Hello World', 'Hell', '%$#@');

SELECT REPLACE('Hello World', 'l', '7');

SELECT REPLACE('Hello World', 'o', '0');

SELECT REPLACE('HellO World', 'o', '*');

SELECT
REPLACE('cheese bread coffee milk', ' ', ' and ');

SELECT REPLACE(title, 'e ', '3') FROM books;

-- SELECT
--    CONCAT
--    (
--       SUBSTRING(title, 1, 10),
--       '...'
--    ) AS 'short title'
-- FROM books;

SELECT
SUBSTRING(REPLACE(title, 'e', '3'), 1, 10)
FROM books;

SELECT
SUBSTRING(REPLACE(title, 'e', '3'), 1, 10) AS 'weird string'
FROM books;
```

REVERSE - string function

Wednesday, October 23, 2019 9:07 PM

```
SELECT REVERSE('Hello World');

SELECT REVERSE('meow meow');

SELECT REVERSE(author_fname) FROM books;

SELECT CONCAT('woof', REVERSE('woof'));

SELECT CONCAT(author_fname, REVERSE(author_fname)) FROM books;
```

CHAR_LENGTH - string function

Wednesday, October 23, 2019 9:09 PM

This function returns the length of the text.

```
SELECT CHAR_LENGTH('Hello, Tom');

SELECT CHAR_LENGTH('Hello, Tom') AS Tite;

SELECT REVERSE(CHAR_LENGTH('Hello, Tom')) AS Tite;

SELECT CHAR_LENGTH('Hello World');

SELECT author_lname, CHAR_LENGTH(author_lname) AS 'length' FROM books;

SELECT CONCAT(author_lname, ' is ', CHAR_LENGTH(author_lname), ' characters long') FROM books;
```

UPPER() and LOWER() - string functions

Wednesday, October 23, 2019 9:26 PM

```
SELECT UPPER('Hello World');

SELECT LOWER('Hello World');

SELECT UPPER(title) FROM books;

SELECT CONCAT('MY FAVORITE BOOK IS ', UPPER(title)) FROM books;

SELECT CONCAT('MY FAVORITE BOOK IS ', LOWER(title)) FROM books;

SELECT CONCAT(LOWER("i am nothing without God!"), ' ', 'I''m not kidding!') AS message;
```

Hi All,
Before you move onto the next lecture, please remember that order is important when dealing with combining/wrapping certain string functions.
For example:
This works:

```
SELECT UPPER(CONCAT(author_fname, ' ', author_lname)) AS "full name in caps"
FROM books;
```

While this does not:

```
SELECT CONCAT(UPPER(author_fname, ' ', author_lname)) AS "full name in caps"
FROM books;
```

UPPER only takes one argument and CONCAT takes two or more arguments, so they can't be switched in that way.
You could do it this way, however:

```
SELECT CONCAT(UPPER(author_fname), ' ', UPPER(author_lname)) AS "full name in caps"
FROM books;
```

But, the first example above would be better (more DRY*) because you wouldn't need to call UPPER two times.
*Don't Repeat Yourself

Basic Data types

Monday, September 2, 2019 9:14 PM

Types of Data

Here's a slightly larger list of data types than in the video.

- Text Type Examples
  - TEXT
  - VARCHAR
- Numeric Type Examples
  - INT
  - INTEGER
- Date Type Examples
  - DATETIME
  - DATE
  - TIMESTAMP

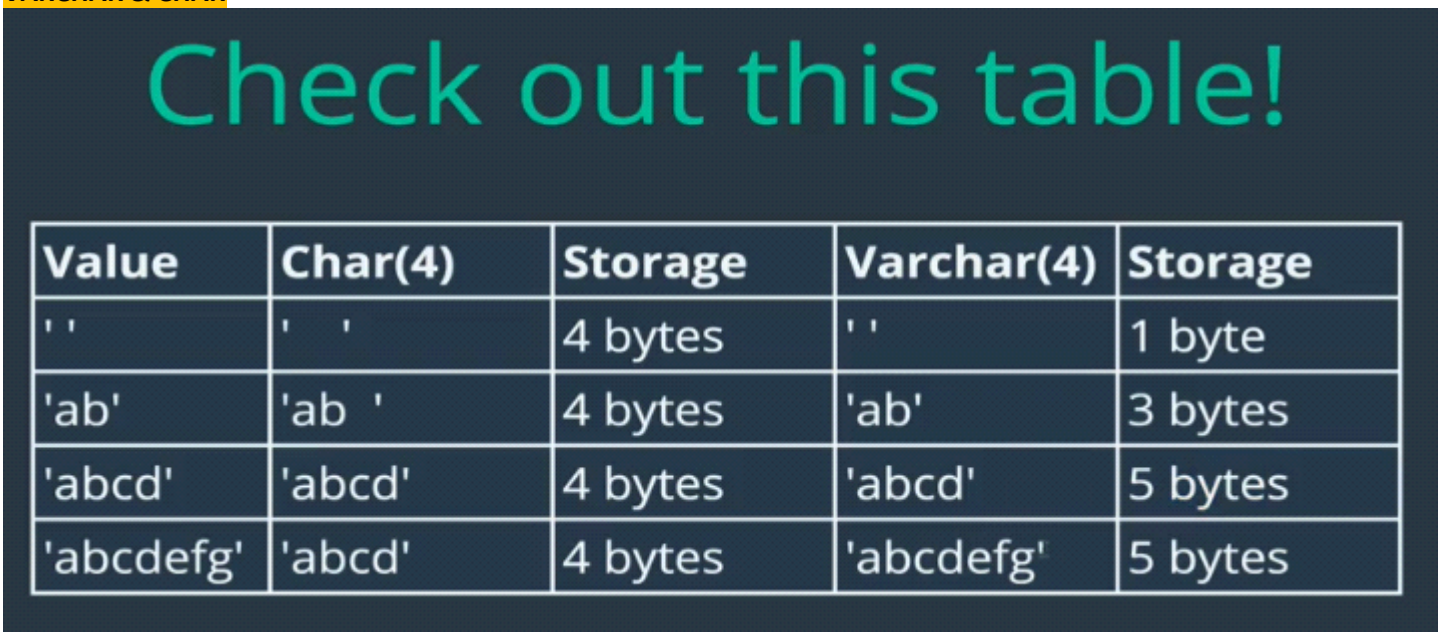
Here's documentation sites for some other databases where you can see the similarities and differences in data types.

- MySQL Data Types
- SQLite Data Types
- PostgreSQL Data Types
- Microsoft SQL Data Types

More Data types

Monday, November 11, 2019 9:22 PM

VARCHAR & CHAR



CHAR is faster if you have a fixed length and the storage size is fixed.
On the other hand, VARCHAR's storage size is dynamic.

DECIMAL

Read more: http://www.mysqltutorial.org/mysql-decimal/

The MySQL DECIMAL data type is used to store exact numeric values in the database. We often use the DECIMAL data type for columns that preserve exact precision e.g., money data in accounting systems.
To define a column whose data type is DECIMAL you use the following syntax:

```
1       column_name  DECIMAL(P,D);
```

In the syntax above:

- P is the precision that represents the number of significant digits. The range of P is 1 to 65.
- D is the scale that that represents the number of digits after the decimal point. The range of D is 0 and 30. MySQL requires that D is less than or equal to (<=) P.

The DECIMAL(P,D) means that the column can store up to P digits with D decimals. The actual range of the decimal column depends on the precision and scale.
Besides the DECIMAL keyword, you can also use DEC, FIXED, or NUMERIC because they are synonyms for DECIMAL.
Like the INT data type, the DECIMAL type also has UNSIGNED and ZEROFILL attributes. If we use the UNSIGNED attribute, the column with DECIMAL UNSIGNED will not accept negative values.
In case we use ZEROFILL, MySQL will pad the display value by 0 up to display width specified by the column definition. In addition, if we use ZERO FILL for the DECIMAL column, MySQL will add the UNSIGNED attribute to the column automatically.

The following example defines amount column with DECIMAL data type.

```
1       amount DECIMAL(6,2);
```

In this example, the amount column can store 6 digits with 2 decimal places; therefore, the range of

the amount column is from 9999.99 to -9999.99.

MySQL allows us to use the following syntax:

```
1       column_name DECIMAL(P);
```

This is equivalent to:

```
1       column_name DECIMAL(P,0);
```

In this case, the column contains no fractional part or decimal point.
In addition, we can even use the following syntax.

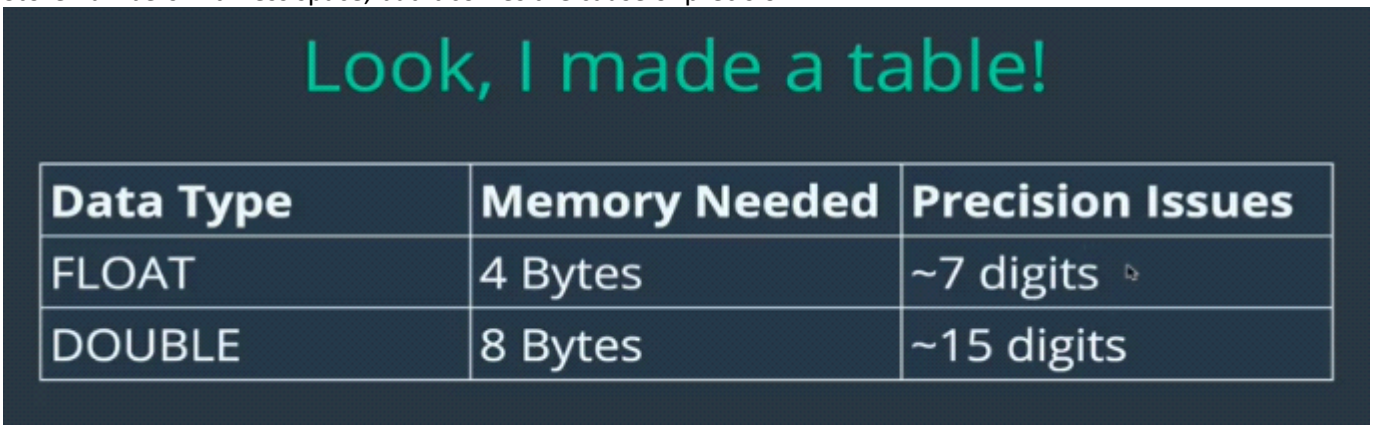
```
1       column_name DECIMAL;
```

The default value of P is 10 in this case.

```
INSERT INTO dogs(name, breed) VALUES ("Nats", "Scalion");
ALTER TABLE dogs
ADD price DECIMAL(4,2) AFTER breed;
INSERT INTO dogs(price) VALUES(99.99);
```

FLOAT & DOUBLE

Store numbers with less space, but it comes the cause of precision.




```
CREATE TABLE thingies (price FLOAT);

INSERT INTO thingies(price) VALUES (88.45);

SELECT * FROM thingies;

INSERT INTO thingies(price) VALUES (8877.45);

SELECT * FROM thingies;

INSERT INTO thingies(price) VALUES (8877665544.45);

SELECT * FROM thingies;
```

DATE & TIME

The DATETIME type is used for values that contain both date and time parts. MySQL retrieves and displays DATETIME values in 'YYYY-MM-DD hh:mm:ss' format. The supported range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'

```
CREATE TABLE people (name VARCHAR(100), birthdate DATE, birthtime TIME, birthdt DATETIME);
```

```
INSERT INTO people (name, birthdate, birthtime, birthdt)
VALUES('Padma', '1983-11-11', '10:07:35', '1983-11-11 10:07:35');

INSERT INTO people (name, birthdate, birthtime, birthdt)
VALUES('Larry', '1943-12-25', '04:10:42', '1943-12-25 04:10:42');

SELECT * FROM people;
```

Some built-in functions

CURDATE() - gives current date
CURTIME() - gives current time
NOW() - give current datetime

Ex:

```
INSERT INTO people (name, birthdate, birthtime, birdthdt) VALUES
("Azucal", CURDATE(), CURTIME(), NOW());
```

Formatting dates

Read more built-in functions here: https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html
Date format: https://dev.mysql.com/doc/refman/8.0/en/date-and-time-functions.html#function_date-format

```
SELECT name, birthdate FROM people;

SELECT name, DAY(birthdate) FROM people;

SELECT name, birthdate, DAY(birthdate) FROM people;

SELECT name, birthdate, DAYNAME(birthdate) FROM people;

SELECT name, birthdate, DAYOFWEEK(birthdate) FROM people;

SELECT name, birthdate, DAYOFYEAR(birthdate) FROM people;

SELECT name, birthtime, DAYOFYEAR(birthtime) FROM people;

SELECT name, birthdt, DAYOFYEAR(birthdt) FROM people;

SELECT name, birthdt, MONTH(birthdt) FROM people;

SELECT name, birthdt, MONTHNAME(birthdt) FROM people;

SELECT name, birthtime, HOUR(birthtime) FROM people;

SELECT name, birthtime, MINUTE(birthtime) FROM people;

SELECT CONCAT(MONTHNAME(birthdate), ' ', DAY(birthdate), ' ', YEAR(birthdate)) FROM people;

SELECT DATE_FORMAT(birthdt, 'Was born on a %W') FROM people;

SELECT DATE_FORMAT(birthdt, '%m/%d/%Y') FROM people;
```

```
SELECT DATE_FORMAT(birthdt, '%m/%d/%Y at %h:%i') FROM people;

SELECT CONCAT(name, " was born on ", DAYNAME(birthdate), " at ", TIME_FORMAT(birthtime, '%h:%
i:%s')) AS "Info" FROM people;
```

Date Math

This is a bunch of predefined functions with mathematical functions that you may use for date and time.

```
SELECT * FROM people;

SELECT DATEDIFF(NOW(), birthdate) FROM people;

SELECT name, birthdate, DATEDIFF(NOW(), birthdate) FROM people;

SELECT birthdt FROM people;

SELECT birthdt, DATE_ADD(birthdt, INTERVAL 1 MONTH) FROM people;

SELECT birthdt, DATE_ADD(birthdt, INTERVAL 10 SECOND) FROM people;

SELECT birthdt, DATE_ADD(birthdt, INTERVAL 3 QUARTER) FROM people;

SELECT birthdt, birthdt + INTERVAL 1 MONTH FROM people;

SELECT birthdt, birthdt - INTERVAL 5 MONTH FROM people;

SELECT birthdt, birthdt + INTERVAL 15 MONTH + INTERVAL 10 HOUR FROM people;
```

TimeStamps

The `DATETIME` type is used for values that contain both date and time parts. MySQL retrieves and displays `DATETIME` values in `'YYYY-MM-DD HH:MM:SS'` format. The supported range is `'1000-01-01 00:00:00'` to `'9999-12-31 23:59:59'`.

The `TIMESTAMP` data type is used for values that contain both date and time parts. `TIMESTAMP` has a range of `'1970-01-01 00:00:01'` UTC to `'2038-01-19 03:14:07'` UTC.

```
CREATE TABLE comments (
    content VARCHAR(100),
    created_at TIMESTAMP DEFAULT NOW()
);

INSERT INTO comments (content) VALUES('lol what a funny article');

INSERT INTO comments (content) VALUES('I found this offensive');

INSERT INTO comments (content) VALUES('lfasfsadfsadfsad');

SELECT * FROM comments ORDER BY created_at DESC;

CREATE TABLE comments2 (
    content VARCHAR(100),
    changed_at TIMESTAMP DEFAULT NOW() ON UPDATE CURRENT_TIMESTAMP
);
```

ON UPDATE will update the current timestamp in the event that the content field Is changed.

```
INSERT INTO comments2 (content) VALUES('dasdasdasd');

INSERT INTO comments2 (content) VALUES('lolololololo');

INSERT INTO comments2 (content) VALUES('I LIKE CATS AND DOGS');

UPDATE comments2 SET content='THIS IS NOT GIBBERISH' WHERE content='dasdasdasd';

SELECT * FROM comments2;

SELECT * FROM comments2 ORDER BY changed_at;

CREATE TABLE comments2 (
  content VARCHAR(100),
  changed_at TIMESTAMP DEFAULT NOW() ON UPDATE NOW()
);
```

AGGREGATE FUNCTIONS

Monday, November 4, 2019 3:45 PM

COUNT
Return the total count of rows.

```
SELECT COUNT(*) FROM BOOKS;

SELECT COUNT(DISTINCT author_fname) FROM books;

SELECT COUNT(title) FROM books WHERE title LIKE "%the%";
```

GROUP BY
Summarizes or aggregates identical data into single rows.
The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".
The GROUP BY statement is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.

```
SELECT title, author_lname FROM books;

SELECT title, author_lname FROM books
GROUP BY author_lname;

SELECT author_lname, COUNT(*)
FROM books GROUP BY author_lname;

SELECT title, author_fname, author_lname FROM books;

SELECT title, author_fname, author_lname FROM books GROUP BY author_lname;

SELECT author_fname, author_lname, COUNT(*) FROM books GROUP BY author_lname;

SELECT author_fname, author_lname, COUNT(*) FROM books GROUP BY author_lname, author_fname;

SELECT released_year FROM books;

SELECT released_year, COUNT(*) FROM books GROUP BY released_year;

SELECT CONCAT('In ', released_year, ' ', COUNT(*), ' book(s) released') AS year FROM books GROUP BY
released_year;
```

MIN & MAX
The **MIN**() function returns the smallest value of the selected column. The **MAX**() function returns the **largest** value of the selected column. If you use it for strings, it might sort the data in ASC order.

```
SELECT MIN(released_year)
FROM books;

SELECT MIN(released_year) FROM books;
```

```
SELECT MIN(pages) FROM books;

SELECT MAX(pages)
FROM books;
```

31. SELECT MAX(CustomerID) as "Greatest ID" FROM Customers;

```
SELECT MAX(released_year)
FROM books;
```

SELECT MAX(pages), title FROM books; <=this will only return the first row. This is a problem and you may use **subqueries** to fix this.

```
SELECT * FROM books
WHERE pages = (SELECT Min(pages)
FROM books);

SELECT title, pages FROM books
WHERE pages = (SELECT Max(pages)
FROM books);

SELECT title, pages FROM books
WHERE pages = (SELECT Min(pages)
FROM books);

SELECT * FROM books
ORDER BY pages ASC LIMIT 1;

SELECT title, pages FROM books
ORDER BY pages ASC LIMIT 1;

SELECT * FROM books
ORDER BY pages DESC LIMIT 1;
```

Using Min and Max with Group By

```
SELECT author_fname,
author_lname,
Min(released_year)
FROM   books
GROUP  BY author_lname,
author_fname;

SELECT
author_fname,
author_lname,
Max(pages)
FROM books
GROUP BY author_lname,
author_fname;

SELECT
CONCAT(author_fname, ' ', author_lname) AS author,
```

```
MAX(pages) AS 'longest book'
FROM books
GROUP BY author_lname,
author_fname;
```

The Sum Function

```
SELECT SUM(pages)
FROM books;

SELECT SUM(released_year) FROM books;

SELECT author_fname,
author_lname,
Sum(pages)
FROM books
GROUP BY
author_lname,
author_fname;

SELECT author_fname,
author_lname,
Sum(released_year)
FROM books
GROUP BY
author_lname,
author_fname;
```

The AVG function

```
SELECT AVG(released_year)
FROM books;

SELECT AVG(pages)
FROM books;

SELECT AVG(stock_quantity)
FROM books
GROUP BY released_year;

SELECT released_year, AVG(stock_quantity)
FROM books
GROUP BY released_year;

SELECT author_fname, author_lname, AVG(pages) FROM books
GROUP BY author_lname, author_fname;
```

Logical Operators (!=, LIKE, < > =, &&, ||, BETWEEN, IN, NOT, CASE Statements)

Thursday, November 14, 2019 9:56 PM

https://dev.mysql.com/doc/refman/8.0/en/logical-operators.html

As per mySQL documentation, The &&, operator is a nonstandard MySQL extension. As of MySQL 8.0.17, this operator is deprecated and support for it will be removed in a future MySQL version. Applications should be adjusted to use the standard SQL AND operator.

NOT EQUAL !=

```
SELECT title FROM books WHERE released_year = 2017;

SELECT title FROM books WHERE released_year != 2017;

SELECT title, author_lname FROM books;

SELECT title, author_lname FROM books WHERE author_lname = 'Harris';

SELECT title, author_lname FROM books WHERE author_lname != 'Harris';
```

NOT LIKE

```
SELECT title FROM books WHERE title LIKE 'W';

SELECT title FROM books WHERE title LIKE 'W%';

SELECT title FROM books WHERE title LIKE '%W%';

SELECT title FROM books WHERE title LIKE 'W%';

SELECT title FROM books WHERE title NOT LIKE 'W%';
```

GREATER THAN >

```
SELECT title, released_year FROM books ORDER BY released_year;

SELECT title, released_year FROM books
WHERE released_year > 2000 ORDER BY released_year;

SELECT title, released_year FROM books
WHERE released_year >= 2000 ORDER BY released_year;

SELECT title, stock_quantity FROM books;

SELECT title, stock_quantity FROM books WHERE stock_quantity >= 100;

SELECT 99 > 1;

SELECT 99 > 567;

100 > 5
-- true

-15 > 15
```

```
-- false

9 > -10
-- true

1 > 1
-- false

'a' > 'b'
-- false

'A' > 'a'
-- false

'A' >=  'a'
-- true

SELECT title, author_lname FROM books WHERE author_lname = 'Eggers';

SELECT title, author_lname FROM books WHERE author_lname = 'eggers';

SELECT title, author_lname FROM books WHERE author_lname = 'eGGers';
```

LESS THAN <

```
SELECT title, released_year FROM books;

SELECT title, released_year FROM books
WHERE released_year < 2000;

SELECT title, released_year FROM books
WHERE released_year <= 2000;

SELECT 3 < -10;
-- false

SELECT -10 < -9;
-- true

SELECT 42 <= 42;
-- true

SELECT 'h' < 'p';
-- true

SELECT 'Q' <= 'q';
-- true
```

LOGICAL AND &&

```
SELECT title, author_lname, released_year FROM books
WHERE author_lname='Eggers';

SELECT title, author_lname, released_year FROM books
WHERE released_year > 2010;
```

```
SELECT
   title,
   author_lname,
   released_year FROM books
WHERE author_lname='Eggers'
   AND released_year > 2010;

SELECT 1 < 5 && 7 = 9;
-- false

SELECT -10 > -20 && 0 <= 0;
-- true

SELECT -40 <= 0 AND 10 > 40;
--false

SELECT 54 <= 54 && 'a' = 'A';
-- true

SELECT *
FROM books
WHERE author_lname='Eggers'
   AND released_year > 2010
   AND title LIKE '%novel%';
```

LOGICAL OR ||

```
SELECT
   title,
   author_lname,
   released_year
FROM books
WHERE author_lname='Eggers' || released_year > 2010;


SELECT 40 <= 100 || -2 > 0;
-- true

SELECT 10 > 5 || 5 = 5;
-- true

SELECT 'a' = 5 || 3000 > 2000;
-- true

SELECT title,
     author_lname,
     released_year,
     stock_quantity
FROM   books
WHERE  author_lname = 'Eggers'
          || released_year > 2010
OR     stock_quantity > 100;
```

BETWEEEN

```
SELECT title, released_year FROM books WHERE released_year >= 2004 && released_year <= 2015;

SELECT title, released_year FROM books
WHERE released_year BETWEEN 2004 AND 2015;

SELECT title, released_year FROM books
WHERE released_year NOT BETWEEN 2004 AND 2015;

SELECT CAST('2017-05-02' AS DATETIME);

show databases;

use new_testing_db;

SELECT name, birthdt FROM people WHERE birthdt BETWEEN '1980-01-01' AND '2000-01-01';
```

As per mySQL documentation, it's encouraged to use CAST() so that the string of data can be converted to a real DATETIME format

```
SELECT
   name,
   birthdt
FROM people
WHERE
   birthdt BETWEEN CAST('1980-01-01' AS DATETIME)
   AND CAST('2000-01-01' AS DATETIME);
```

IN or NOT IN

```
SELECT title, released_year FROM books
WHERE released_year NOT IN
(2000,2002,2004,2006,2008,2010,2012,2014,2016);

SELECT title, released_year FROM books
WHERE released_year >= 2000
AND released_year NOT IN
(2000,2002,2004,2006,2008,2010,2012,2014,2016);

SELECT title, released_year FROM books
WHERE released_year >= 2000 AND
released_year % 2 != 0;

SELECT title, released_year FROM books
WHERE released_year >= 2000 AND
released_year % 2 != 0 ORDER BY released_year;
```

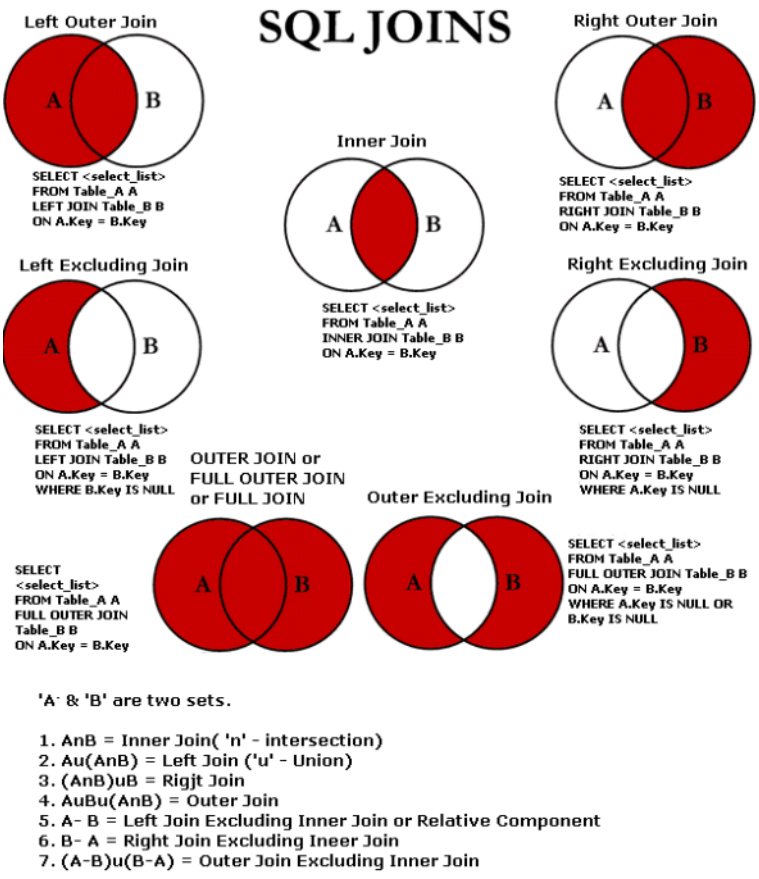
CASE STATEMENTS

```
SELECT *, //the tables
CASE //this is the start of the condition
WHEN id >= 3 THEN "You suck ass!" //the condition
ELSE "You rule!" //if the primary condition is not met, do this instead
END AS Result //print the result in this aliased table
FROM customers; //the source table
```

```
SELECT id, first_name, last_name, email,
CASE
WHEN id BETWEEN 1 AND 2 THEN "*"
WHEN id BETWEEN 3 AND 4 THEN "**"
ELSE "***"
END AS Result
FROM customers;
```

JOINS!

Saturday, November 16, 2019 5:10 PM



An SQL join clause - corresponding to a join operation in relational algebra - combines columns from one or more tables in a relational database. It creates a set that can be saved as a table or used as it is. A JOIN is a means for combining columns from one or more tables by using values common to each

Download:

customer_a
nd_orders

Cross Join

This is the most useless and basic type of join. A **cross join** is **used when** you wish to create combination of **every row** from two tables. All row combinations are included in the result; this is commonly called **cross product join**.

SELECT * FROM customers, orders;

As you can see, the fk customer_id doesn't correspond to the actual customer that ordered the product.

id	first_name	last_name	email	id	order_date	amount	customer_id
1	Boy	George	george@gmail.com	1	2016-02-10	99.99	1
2	George	Michael	gm@gmail.com	1	2016-02-10	99.99	1
3	David	Bowie	david@gmail.com	1	2016-02-10	99.99	1
4	Blue	Steele	blue@gmail.com	1	2016-02-10	99.99	1
5	Bette	Davis	bette@aol.com	1	2016-02-10	99.99	1
1	Boy	George	george@gmail.com	2	2017-11-11	35.50	1
2	George	Michael	gm@gmail.com	2	2017-11-11	35.50	1
3	David	Bowie	david@gmail.com	2	2017-11-11	35.50	1
4	Blue	Steele	blue@gmail.com	2	2017-11-11	35.50	1
5	Bette	Davis	bette@aol.com	2	2017-11-11	35.50	1
1	Boy	George	george@gmail.com	3	2014-12-12	800.67	2
2	George	Michael	gm@gmail.com	3	2014-12-12	800.67	2
3	David	Bowie	david@gmail.com	3	2014-12-12	800.67	2
4	Blue	Steele	blue@gmail.com	3	2014-12-12	800.67	2
5	Bette	Davis	bette@aol.com	3	2014-12-12	800.67	2
1	Boy	George	george@gmail.com	4	2015-01-03	12.50	2
2	George	Michael	gm@gmail.com	4	2015-01-03	12.50	2
3	David	Bowie	david@gmail.com	4	2015-01-03	12.50	2
4	Blue	Steele	blue@gmail.com	4	2015-01-03	12.50	2
5	Bette	Davis	bette@aol.com	4	2015-01-03	12.50	2
1	Boy	George	george@gmail.com	5	1999-04-11	450.25	5
2	George	Michael	gm@gmail.com	5	1999-04-11	450.25	5
3	David	Bowie	david@gmail.com	5	1999-04-11	450.25	5
4	Blue	Steele	blue@gmail.com	5	1999-04-11	450.25	5
5	Bette	Davis	bette@aol.com	5	1999-04-11	450.25	5

Inner Join

The following (implicit inner join) syntax will do the job if you don't prefer using an explicit inner join clause

SELECT * FROM customers, orders WHERE customers.id = orders.customer_id;

Using an explicit inner join clause

SELECT * FROM customers
JOIN orders **ON** customers.id = orders.customer_id;

OR

SELECT * FROM customers
INNER JOIN orders **ON** customers.id = orders.customer_id;

id	first_name	last_name	email	id	order_date	amount	customer_id
1	Boy	George	george@gmail.com	1	2016-02-10	99.99	1
1	Boy	George	george@gmail.com	2	2017-11-11	35.50	1
2	George	Michael	gm@gmail.com	3	2014-12-12	800.67	2
2	George	Michael	gm@gmail.com	4	2015-01-03	12.50	2
5	Bette	Davis	bette@aol.com	5	1999-04-11	450.25	5

Left Join

SELECT * FROM customers
LEFT JOIN orders **ON** customers.id = orders.customer_id;

This join returns null values.

id	first_name	last_name	email	id	order_date	amount	customer_id
1	Boy	George	george@gmail.com	1	2016-02-10	99.99	1
1	Boy	George	george@gmail.com	2	2017-11-11	35.50	1
2	George	Michael	gm@gmail.com	3	2014-12-12	800.67	2
2	George	Michael	gm@gmail.com	4	2015-01-03	12.50	2
5	Bette	Davis	bette@aol.com	5	1999-04-11	450.25	5
3	David	Bowie	david@gmail.com	NULL	NULL	NULL	NULL
4	Blue	Steele	blue@gmail.com	NULL	NULL	NULL	NULL

Right Join


```
SELECT
    IFNULL(first_name,'MISSING') AS first,
    IFNULL(last_name,'USER') as last,
    order_date,
    amount,
    SUM(amount)
FROM customers
RIGHT JOIN orders
    ON customers.id = orders.customer_id
GROUP BY first_name, last_name;
```

This join returns null values.

first_name	last_name	order_date	amount
Boy	George	2016-02-10	99.99
Boy	George	2017-11-11	35.50
George	Michael	2014-12-12	800.67
George	Michael	2015-01-03	12.50
Bette	Davis	1999-04-11	450.25
NULL	NULL	2017-11-05	23.45
NULL	NULL	2017-04-26	777.77

Query Quizzes

Monday, October 14, 2019 9:49 PM

Fill in the missing operator. Today is 19th October 2019. I want to find all matches happening today and in the future.
SELECT * FROM football_matches WHERE event_date >= "2019-10-19";

If I wanted to return rows that match **both** conditions, which keyword would I use?
SELECT <columns> FROM <table> WHERE <condition 1> AND <condition 2>;

Inequality operator
SELECT <columns> FROM <table> WHERE <column> != <value>;

I want to categorize products by price on a website. Cheap is defined by the prices from 0.01 and 9.99. Enter the missing keywords.
SELECT name, description FROM products WHERE price BETWEEN 0.01 AND 9.99;

If I wanted to return rows that match **either** conditions, which keyword would I use?
SELECT <columns> FROM <table> WHERE <condition 1> OR <condition 2>;

Exercise:

```
CREATE TABLE books
	(
		book_id INT NOT NULL AUTO_INCREMENT,
		title VARCHAR(100),
		author_fname VARCHAR(100),
		author_lname VARCHAR(100),
		released_year INT,
		stock_quantity INT,
		pages INT,
		PRIMARY KEY(book_id)
	);

INSERT INTO books (title, author_fname, author_lname, released_year, stock_quantity, pages)
VALUES
('The Namesake', 'Jhumpa', 'Lahiri', 2003, 32, 291),
('Norse Mythology', 'Neil', 'Gaiman',2016, 43, 304),
('American Gods', 'Neil', 'Gaiman', 2001, 12, 465),
('Interpreter of Maladies', 'Jhumpa', 'Lahiri', 1996, 97, 198),
('A Hologram for the King: A Novel', 'Dave', 'Eggers', 2012, 154, 352),
('The Circle', 'Dave', 'Eggers', 2013, 26, 504),
('The Amazing Adventures of Kavalier & Clay', 'Michael', 'Chabon', 2000, 68, 634),
('Just Kids', 'Patti', 'Smith', 2010, 55, 304),
('A Heartbreaking Work of Staggering Genius', 'Dave', 'Eggers', 2001, 104, 437),
('Coraline', 'Neil', 'Gaiman', 2003, 100, 208),
('What We Talk About When We Talk About Love: Stories', 'Raymond', 'Carver', 1981, 23, 176),
("Where I'm Calling From: Selected Stories", 'Raymond', 'Carver', 1989, 12, 526),
('White Noise', 'Don', 'DeLillo', 1985, 49, 320),
('Cannery Row', 'John', 'Steinbeck', 1945, 95, 181),
```

('Oblivion: Stories', 'David', 'Foster Wallace', 2004, 172, 329),
('Consider the Lobster', 'David', 'Foster Wallace', 2005, 92, 343);

```
SELECT Reverse(Upper('Why does my cat look at me with such hatred?')) AS mess
age
;

SELECT Replace(title, ' ', '->') AS title
FROM   books;

SELECT author_fname          AS forwards,
       Reverse(author_lname) AS backwards
FROM   books;

SELECT Upper(Concat(author_fname, ' ', author_lname)) AS 'full name in caps'
FROM   books;

SELECT Concat(title, ' was released in ', released_year) AS blurb
FROM   books;

SELECT title,
       Char_length(title) AS character_count
FROM   books;

SELECT Concat(Substring(title, 1, 10), '...')  AS 'short title',
       Concat(author_lname, ',', author_fname) 'author',
       Concat(stock_quantity, ' in stock') AS 'quantity'
FROM   books
ORDER  BY title;
```

```
SELECT title
FROM   books
WHERE  title LIKE '%stories%';

SELECT title,
    pages
FROM   books
ORDER  BY pages DESC
LIMIT  1;

SELECT Concat(title, ' - ', released_year) AS 'summary'
FROM   books
ORDER  BY released_year DESC
LIMIT  3;

SELECT title,
    author_lname
FROM   books
WHERE  author_lname LIKE '% %';

SELECT title,
    released_year,
    stock_quantity
FROM   books
```

```
ORDER  BY stock_quantity DESC
LIMIT  3;

SELECT title,
     author_lname
FROM   books
ORDER  BY author_lname;

SELECT Upper(Concat_ws(' ', 'my favorite author is', author_fname, author_lname,
          '!'))
     AS yell
FROM   books
ORDER  BY author_lname;
```

Aggregate functions

```
SELECT COUNT(*) FROM books;

SELECT released_year, COUNT(*) FROM books GROUP BY released_year;

SELECT SUM(stock_quantity) FROM books;

SELECT author_fname, author_lname, AVG(released_year) FROM books GROUP BY released_year;

SELECT CONCAT(author_fname, ' ',author_lname), pages FROM books WHERE pages = (SELECT
MAX(pages) from books);

SELECT released_year AS 'year', COUNT(*) AS 'books', AVG(pages) AS 'avg pages' from books GROUP BY
released_year ORDER BY year ASC;
```

Logical Operators

```
SELECT title FROM books WHERE released_year < 1980;
SELECT title, CONCAT(author_fname, ' ', author_lname) AS "Name" FROM books WHERE author_lname
= "Eggers" OR author_lname = "Chabon";
SELECT * FROM books WHERE pages BETWEEN 100 AND 200;
SELECT author_lname FROM books WHERE author_lname LIKE 'C%' OR  author_lname LIKE 'S%';

SELECT title,
CASE
WHEN title LIKE '%stories%' THEN "Short stories"
WHEN title LIKE '%just kids%' THEN "Memoir"
WHEN title LIKE '%heartbreaking work%' THEN "Memoir"
ELSE "Novel"
END AS Result
FROM books;

SELECT author_fname, author_lname,
   CASE
      WHEN COUNT(*) = 1 THEN '1 book'
      ELSE CONCAT(COUNT(*), ' books')
   END AS COUNT
FROM books
```

```
GROUP BY author_lname, author_fname;
```

More data types

varchar is dynamic when it comes to size. char is faster

```
CREATE TABLE inventory (item_name VARCHAR(100), price DECIMAL(8,2), quantity INT);
```

timestamp has a range from 1972 to 2038 something like that. timestamp is faster

```
select curtime();

select curdate();

SELECT DAYOFWEEK(CURDATE());

SELECT DATE_FORMAT(CURDATE(), '%m/%d/%Y') AS 'Current Date';

SELECT CONCAT(DATE_FORMAT(CURDATE(), '%M %D'), ' at ', TIME_FORMAT(CURTIME(),'%h:%i %p')) AS
'Current Time';

CREATE TABLE tweets_table (content VARCHAR(255), username VARCHAR(140), time_created
TIMESTAMP DEFAULT NOW()ON UPDATE CURRENT_TIMESTAMP);

INSERT INTO tweets_table (content, username) VALUES ('first comment, lol!', 'Johny');
INSERT INTO tweets_table (content, username) VALUES ('Sup, pus boui?', 'ray');
INSERT INTO tweets_table (content, username) VALUES ('fuck off!', 'Johny');

select * from tweets_table ORDER BY time_created DESC;
```

Joins

```
CREATE DATABASE class;

CREATE TABLE students (
id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
first_name VARCHAR(100)
);

CREATE TABLE papers (
title VARCHAR(100),
grade INT,
student_id INT,
FOREIGN KEY (student_id) REFERENCES students(id)
);

  INSERT INTO students (first_name) VALUES
  ('Caleb'),
  ('Samantha'),
  ('Raj'),
  ('Carlos'),
  ('Lisa');
```

```
INSERT INTO papers (student_id, title, grade ) VALUES
(1, 'My First Book Report', 60),
(1, 'My Second Book Report', 75),
(2, 'Russian Lit Through The Ages', 94),
(2, 'De Montaigne and The Art of The Essay', 98),
(4, 'Borges and Magical Realism', 89);

SELECT first_name, title, grade FROM students
INNER JOIN papers ON students.id = papers.student_id
ORDER BY grade DESC;

SELECT first_name, title, grade FROM students
LEFT JOIN papers ON students.id = papers.student_id;

SELECT
first_name,
IFNULL (title, "Missing"),
IFNULL (grade, "0")
FROM students
LEFT JOIN papers ON students.id = papers.student_id;

SELECT
first_name,
IFNULL (AVG(grade), "0") AS "average"
FROM students
LEFT JOIN papers ON students.id = papers.student_id
GROUP BY first_name
ORDER BY grade DESC;

SELECT
first_name,
IFNULL (AVG(grade), "0") AS "average",
        CASE
        WHEN grade >= 75 THEN "PASSING"
        ELSE "FAILING"
        END AS passing_status
FROM students
LEFT JOIN papers ON students.id = papers.student_id
GROUP BY first_name
ORDER BY grade DESC;
```

OR TRY THIS ALTERNATE SOLUTION

```
SELECT first_name,
       Ifnull(Avg(grade), 0) AS average,
       CASE
         WHEN Avg(grade) IS NULL THEN 'FAILING'
         WHEN Avg(grade) >= 75 THEN 'PASSING'
         ELSE 'FAILING'
       end               AS passing_status
FROM   students
       LEFT JOIN papers
```

```
        ON students.id = papers.student_id
GROUP  BY students.id
ORDER  BY average DESC;
```

More TV Joins

```
CREATE TABLE reviewers (
    id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(100),
    last_name VARCHAR(100)
);

CREATE TABLE series(
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(100),
    released_year YEAR(4),
    genre VARCHAR(100)
);

CREATE TABLE reviews (
    id INT AUTO_INCREMENT PRIMARY KEY,
    rating DECIMAL(2,1),
    series_id INT,
    reviewer_id INT,
    FOREIGN KEY(series_id) REFERENCES series(id),
    FOREIGN KEY(reviewer_id) REFERENCES reviewers(id)
);

INSERT INTO series (title, released_year, genre) VALUES
    ('Archer', 2009, 'Animation'),
    ('Arrested Development', 2003, 'Comedy'),
    ("Bob's Burgers", 2011, 'Animation'),
    ('Bojack Horseman', 2014, 'Animation'),
    ("Breaking Bad", 2008, 'Drama'),
    ('Curb Your Enthusiasm', 2000, 'Comedy'),
    ("Fargo", 2014, 'Drama'),
    ('Freaks and Geeks', 1999, 'Comedy'),
    ('General Hospital', 1963, 'Drama'),
    ('Halt and Catch Fire', 2014, 'Drama'),
    ('Malcolm In The Middle', 2000, 'Comedy'),
    ('Pushing Daisies', 2007, 'Comedy'),
    ('Seinfeld', 1989, 'Comedy'),
    ('Stranger Things', 2016, 'Drama');


INSERT INTO reviewers (first_name, last_name) VALUES
    ('Thomas', 'Stoneman'),
    ('Wyatt', 'Skaggs'),
    ('Kimbra', 'Masters'),
    ('Domingo', 'Cortes'),
    ('Colt', 'Steele'),
    ('Pinkie', 'Petit'),
```

```
    ('Marlon', 'Crafford');


  INSERT INTO reviews(series_id, reviewer_id, rating) VALUES
    (1,1,8.0),(1,2,7.5),(1,3,8.5),(1,4,7.7),(1,5,8.9),
    (2,1,8.1),(2,4,6.0),(2,3,8.0),(2,6,8.4),(2,5,9.9),
    (3,1,7.0),(3,6,7.5),(3,4,8.0),(3,3,7.1),(3,5,8.0),
    (4,1,7.5),(4,3,7.8),(4,4,8.3),(4,2,7.6),(4,5,8.5),
    (5,1,9.5),(5,3,9.0),(5,4,9.1),(5,2,9.3),(5,5,9.9),
    (6,2,6.5),(6,3,7.8),(6,4,8.8),(6,2,8.4),(6,5,9.1),
    (7,2,9.1),(7,5,9.7),
    (8,4,8.5),(8,2,7.8),(8,6,8.8),(8,5,9.3),
    (9,2,5.5),(9,3,6.8),(9,4,5.8),(9,6,4.3),(9,5,4.5),
    (10,5,9.9),
    (13,3,8.0),(13,4,7.2),
    (14,2,8.5),(14,3,8.9),(14,4,8.9);

--inner join--
  SELECT
    title,
    rating
  FROM series
  JOIN reviews
    ON series.id = reviews.series_id;

  SELECT
    title,
    AVG(rating) as avg_rating
  FROM series
  JOIN reviews
    ON series.id = reviews.series_id
  GROUP BY series.id
  ORDER BY avg_rating;

  SELECT
    first_name,
    last_name,
      rating
  FROM reviewers
  INNER JOIN reviews
    ON reviewers.id = reviews.reviewer_id;

      SELECT genre,
                Round(Avg(rating)) AS "avg rating"
      FROM   series
                INNER JOIN reviews
        ON series.id = reviews.series_id
      GROUP  BY genre;

  SELECT title,
    rating,
    Concat(first_name, ' ', last_name) AS "reviewer"
  FROM   reviewers
```

```
    INNER JOIN reviews
            ON reviewers.id = reviews.reviewer_id
    INNER JOIN series
            ON series.id = reviews.series_id
  ORDER  BY title; --double inner joins--

--left join--
  SELECT
          title,
        rating
  FROM series
  LEFT JOIN reviews
    ON series.id = reviews.series_id
  WHERE rating IS NULL;

  SELECT first_name,
        last_name,
        Count(rating)                       AS COUNT,
        Ifnull(Min(rating), 0)                AS MIN,
        Ifnull(Max(rating), 0)                AS MAX,
        Round(Ifnull(Avg(rating), 0), 2)          AS AVG,
        IF(Count(rating) > 0, 'ACTIVE', 'INACTIVE') AS STATUS --if,else shorthand--
  FROM   reviewers
        LEFT JOIN reviews
              ON reviewers.id = reviews.reviewer_id
  GROUP  BY reviewers.id;
```