Learn MySQL in Plain English

A beginner's guide to MySQL

TREVOR J PAGE

Table of Contents

What is a Database?

Managed Data Storage

The sole purpose of a database is to store information in an organized way.

That’s really as simple as it gets, and it is something that you should always come back to throughout this book.

We want to store our information in an organized way, such that we can retrieve the data quickly at some later date and time. We not only want to be able to retrieve our data at some point in the future, but we also want to get it VERY quickly.

I’m talking split second speed here!

In a test that I literally JUST did on my own computer as I was writing this sentence, my laptop was able to return over 110,000 rows of data in just 1 second.

Think about that…

I’m talking about the ability for my simple laptop computer to get the equivalent of ALL of the transactions you’ve ever made with your credit card and debit card combined over the course of your entire life… plus ALL of the emails you’ve ever sent in the past 5 years of your life, in under 1 second.

That’s a lot of speed!

So I guess it’s fair to say that modern databases are pretty darn efficient when it comes to storing and retrieving data.

And that’s exactly what you’re going to learn about in this book. You’re going to learn how to harness the power, speed, efficiency and sheer “coolness” of databases.

The Basics

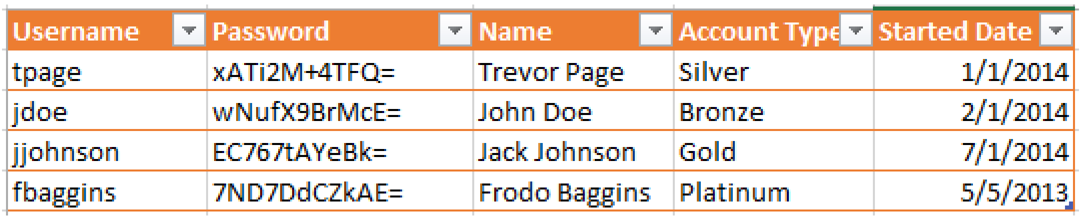
First thing’s first, let’s make sure you can “visualize” what a database is.

I’m all about visualization to aid in learning new concepts, so let’s start with a simple spreadsheet. You know, with rows and columns with boring figures and data points?

Simple spreadsheets aren’t far off from what modern databases are all about. It’s actually why some companies choose to store their precious information in a spreadsheet instead of relational databases. This, however, is a huge mistake if you hope to store any larger amounts of data.

In any case, just like spreadsheets have rows and columns, so do relational databases. Rows make up the unique data amount "something" in particular, and columns give us the ability to store more and more information about each unique row of data.

For example, let's take a look at a simple spreadsheet that outlines information about a User. Let's assume we want to store some information about Users so that we can have them login to an application.



Username	Password	Name	Account Type	Started Date
tpage	xATi2M+4TFQ=	Trevor Page	Silver	1/1/2014
jdoe	wNufX9BrMcE=	John Doe	Bronze	2/1/2014
jjohnson	EC767tAYeBk=	Jack Johnson	Gold	7/1/2014
fbaggins	7ND7DdCZkAE=	Frodo Baggins	Platinum	5/5/2013

We keep track of many things like their username, password, account type, and so on. Each one of these unique things we keep track of is a column. It tells us something of interest with regards to this particular spreadsheet.

Each row of data makes up the unique users stored in our spreadsheet. Each row is unique, as we never want to have a duplicate user. Imagine the chaos of storing two users with the same username… if one of the two people logged into the system, which data would you want to show them? You could accidentally show them the data that belongs to the OTHER user that shares the same username.

Maintaining unique rows of data is of paramount importance in databases. But we'll dive into that later.

For now I want you to concentrate on the concept of rows and columns… and tables.

What's a table?

Well a table is where we stored the rows and columns. You can think of a table like an individual spreadsheet. A table is where all of our data is actually stored in our database. It's kind of like a folder on your desktop. It helps us organize our data.

In our example of users, we would likely have our information stored in a "Users" table. We would name it that, because that name just makes a whole lot of sense. We're storing user information in the table, so we might as well name it the "Users" table.

Tables are typically named after nouns. A noun in the English language means a person, place or thing. If you have any object oriented programming experience, then you can think of a table as an Object. If you don't have any programming background, then no worries, I won't make too many nerdy programming references.

Alright, so you've just learned that these things called Tables are used to store information by following the format of rows and columns. And believe it or not, that's pretty much as complex as a database really gets in terms of how to "visualize" it. So just be sure to have those terms memorized before continuing, as I'll be using them a lot throughout this book.

Intro to SQL and Databases

Now let’s shift our discussion into the actual operations that a database carries out during the course of its life. Obviously databases were created to actually DO stuff right? We call the “work” that a database does an operation. It’s not brain surgery, but it does require some detailed knowledge of how to talk in the “language of databases”.

What the heck is the language of databases? Why it’s SQL of course! SQL is an acronym that stands for “Structured Query Language”.

I usually pronounce SQL as “sequel” (like the sequel to a movie). Some people don’t agree that it should be pronounced this way and insist on just saying the letters “Ess-Queue-Elle”. I’m a man who likes efficiency, and saying the letters in SQL takes longer than saying sequel, so you can guess how I pronounce it. In any case, knowing how to say the name of the language of databases isn’t why you have this book I presume.

So let’s learn how to speak in our database tongue shall we?

What’s all this CRUD about?

Create

Read

Update

Delete

This is the stuff at the heart of all databases and SQL. A database essentially carries out these four operations over and over again for the duration of its existence.

In this initial discussion, I talk about these four database operations in detail and the actual syntax that is used in a flavor of SQL known as MySQL.

MySQL

You can think of SQL as a kind of specification that needs implementation. In Java terms, you can think of SQL like an interface and MySQL like the class that implements it.

I chose to go into depth with MySQL as it’s a free implementation of SQL, it is popular and widely adopted with plenty of HOW-TO articles around the net.

If you wish to learn how to install the MySQL RDBMS and a GUI for manipulation your database, then please check out this video: *link required here*

Create

Let's talk syntax!

The first step in our CRUD operations is the Create operation. This is used to create data in the database. This can be accomplished using the `insert` keyword.

Let's assume you would like to `insert` a new `User` into your database. How would you accomplish this?

I like to approach these questions with the mindset of "What would I need to know if I was the database management system?" I would need to know **where** to insert the data, and specifically **what** data to insert.

Let's see what that looks like in the MySQL syntax:

```
insert into users (username, password) values ('tpage', 'password123');
```

In the SQL code above, we have satisfied the questions of **where** and **what**. The **where** is satisfied by specifying the name of the table in which the data is to be inserted, and also the columns we wish to populate. The **what** is satisfied by the actual data we are providing in the single quotes.

Read

This is by far the most commonly used operation in any database. The ability to retrieve/read data in a database is crucial to a properly functioning application. In order to read information from a database using MySQL you'll use the `select` keyword. I'm not sure why they didn't choose to go with a keyword like "read", but hey, what can you do!

Just like in the "Create" section, when "Reading" information from a database, you'll need to think of what the database management system would need to use to get you to the information that you want. In this case, it's mostly just a matter of "where" is the data that you want. Let's assume you want to retrieve the username and password for a particular user to validate that they are indeed a valid user. What would you need to specify? Well, you need to tell the database where this information exists in terms of the table and the columns. Here's an example:

```
select username, password from users;
```

Now this code will give you the `username` and `password` of **ALL** the records in the `users` table. This isn't exactly what we wanted, we specifically want to verify if ONE particular user exists in the database and what that particular user's `username` and `password` is. So how do we accomplish this with SQL? It's all about the "where"!

```
select username, password from users where username = 'tpage';
```

Don't forget that you can also keep 'chaining' the where question in MySQL using the `and` keyword like so:

```
select username, password from users

where username = 'tpage'

and password = 'password123';
```

This will only return a match if you have a user that exists in the `users` table with the `username = 'tpage'` which has a `password = 'password123'`.

Note: You can see that I broke up the above SQL statement with carriage returns so that it becomes a bit more readable. I separate each 'section' of the syntax with a line break.

Update

When you want to change the data for any existing row in a database, you need to invoke an Update using the `update` keyword.

Let's say that you wish to change a given user's email address… how would you do it?

```
update users

set email = 'trevor@javavideotutorials.net'

where email = 'info@howtoprogramwithjava.com';
```

This is kind of tricky, but it functions just like a Java statement. The database management system will essentially read this statement in chunks.

It will figure out what table will be changing (this is found in the first line of the script above)

It will figure out which row(s) that need to be updated

It will change the data to the value you've specified

In other words, the database management system (MySQL) is able to find the row of data that needs to be updated with the last line (`where email = 'info@howtoprogramwithjava.com'`), it will then take the row(s) that it finds and execute the update that you've specified to change the row(s) email to be 'trevor@javavideotutorials.net'.

Note: It's very important that you know the results of the `update` statement before it executes, as you don't want to accidentally update MORE rows than you intended. For this, I would recommend copy/pasting your `update` statement and changing it to reflect a `select` statement like so:

```
--update users

--set email = 'trevor@javavideotutorials.net'

--where email = 'info@howtoprogramwithjava.com';


select * from users

where email = 'info@howtoprogramwithjava.com';
```

This way you can run the `select` statement and ensure that it returns the expected rows. Once you're satisfied with the rows that are returned, you can run the `update` statement and it will then update those rows that you saw in the `select` statement.

Make sense?

Delete

The final main operation that MySQL can carry out is a `delete`. This is used to completely destroy a row (or multiple rows) of data. This is very similar to writing a `select` statement, in that you are narrowing in on the data that you wish to delete. Let's say we want to delete the row of data that pertains to the user with the email address 'info@howtoprogramwithjava.com'. It would look something like this:

```
delete from users

where email = 'info@howtoprogramwithjava.com';
```

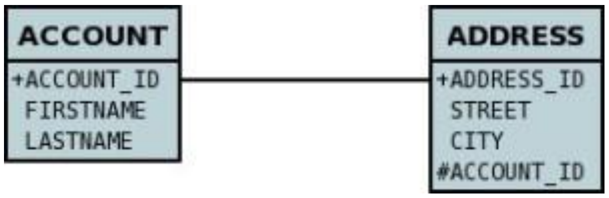
Easy peasy. The only thing that trips me up sometimes with the `delete` statement, is that sometimes I put an asterisk next to the `delete` keyword. It feels natural to want to say “delete ALL from users” by typing `delete * from users`, but this will in fact result in a syntax error.

Database Joins, Keys and Relationships

Modern databases are referred to as relational databases. This is a key thing to point out, because modern databases are all about maintaining relationships. Relationships are at the heart of what makes modern databases so great at storing data in a manageable way!

Let's say, for example, that you are designing a database that's intended to store information about a person's bank account and their home address. For reasons that I'll get into much later, it's best to store that information in two separate tables: an account table and an address table.

Here's a diagram that outlines this relationship:



There are probably two things you'll probably be a bit confused by in the diagram above. But before we get into that, let's take a look at what should make sense.

You'll notice that we have two boxes which represent two tables in a database. One is called "Account" the other "Address". This makes sense since we're going to be storing information about people's bank account and their addresses. No biggie.

You will also notice that we have given each table names for their rows. In the Account table, we have three column names: account_id, firstname and lastname. In the Address table we have four columns: address_id, street, city and account_id.

All of that should make some sense to you (more or less), but the confusing parts are why we have the columns "account_id" and "address_id", and why "account_id" appears twice (once in each table).

Well this is where we're going to start talking about relationships and keys.

Let's talk about keys

Before we can talk about relationships, you first need to understand the concept of keys.

Do you remember how I said that each row in a database should be unique? We were talking about what would happen if the Users table were to have two rows that had the same username and how crazy things would get if that were to happen.

Well this is where keys come into play: specifically a primary key. There are a few different types of keys, but I want to focus on the most common, the primary key.

The Primary Key

You see, it's the job of the primary key to ensure that each row in a database table is unique! It accomplishes this by ensuring that a unique number is assigned to every row in any given table. This is usually done by starting with the number 1, and then incrementing by 1 for each new row that is inserted into the table.

This simple rule of always incrementing by 1 ensures that you will always assign a unique "key" to each row in a given table.

This way, if you were to accidentally assign the same username to two different rows in the User table, you'd at least be able to tell them apart by their primary keys.

So, you see that's what those two columns are in our Account and Address tables (account_id and address_id). They are primary keys: account_id is the primary key of the Account table and address_id is the primary key of the Address table.

But why does account_id appear twice? Once in the Account table, and once in the Address table?

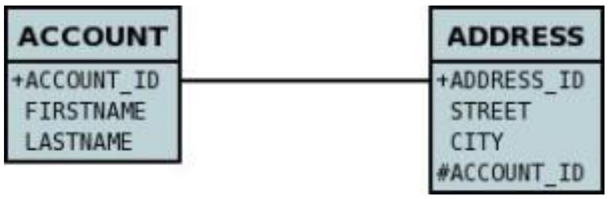
If you asked that question, you are quite observant (good for you). The answer is a bit tricky, but now is as good a time as ever to talk about another kind of key: the foreign key.

The Foreign Key

Foreign keys are what we use to create relationships. It's what we use to "tie" one database table to another database table.

When we tie two tables together (via a foreign key), this is known as a relationship.

In the case of our example tables (Account and Address) we have a relationship that can be identified by the foreign key "account_id". I'll re-display this relationship with the same graphic as we've already seen above (for your viewing pleasure):



As you can see here, we have the column of "account_id" appearing inside of the Address table. You should realize that the account_id field really has no business being inside of the Address table, as it doesn't have anything to do with someone's address now does it?

If I were to walk up to you on the street and ask you what your address was, would you at any point tell me what your account id was? I would hope not! Though I would also hope you wouldn't give me your address, because that's just creepy.

In any case, though it may seem like the account_id field is out of place inside of the Address table, it's actually there with good reason. It's what ties the two tables together… with a relationship.

This means that we are actually telling our database, in no uncertain terms, that the Account table and the Address table are in some way linked by an account_id.

What's important to note here is that the account_id is functioning as a primary key within the Account table, and as a foreign key inside the Address table. We call it a "foreign key" inside the Address table because it's the primary key of another table… thus it's foreign, ou-la-laaa.

So, having covered that, this relationship is what we use to be able to JOIN data from the "Account" table to the "Address" table.

Joins

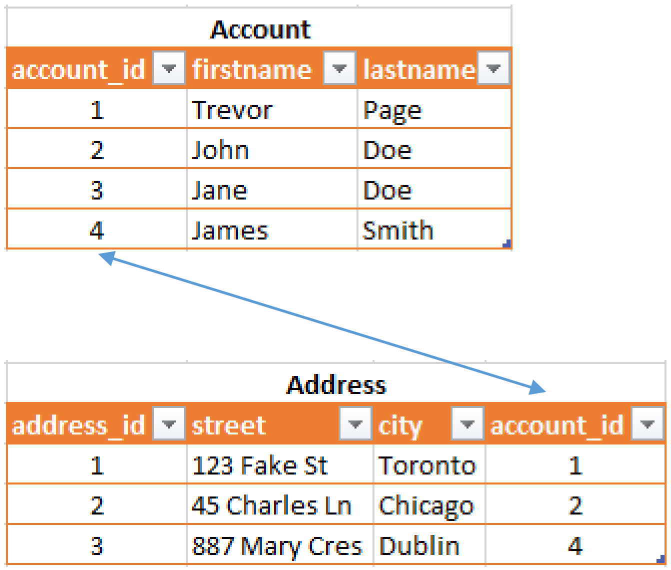
The topic of a join may be a bit of a head scratcher at first, but it's at the very base of the knowledge you need to gain to be a database and SQL pro, so pay attention here.

Joining is the process of matching rows between tables based on their relationships.

Read that line I just wrote above until it's etched into your memory… I'll wait.

If it doesn't make sense yet, then hopefully it will soon. In our example we have two tables (Account and Address) that maintain a relationship via the "account_id" foreign key in the Address table. What this really means is that for each row in the Address table, there should be a matching row in the Account table (based on the foreign key).

Let's take a look at a real world example. Let's have a look at some example data that could exist in both of these tables:



Account

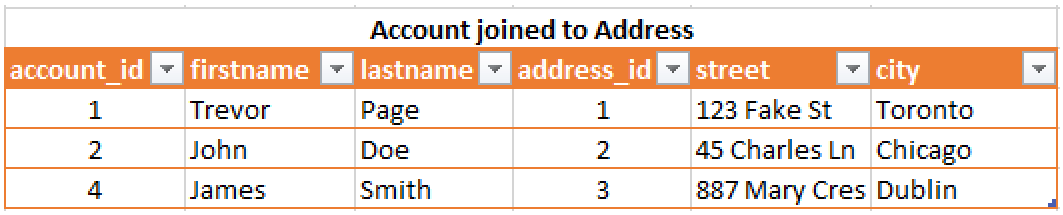
account_id	firstname	lastname
1	Trevor	Page
2	John	Doe
3	Jane	Doe
4	James	Smith

Address

address_id	street	city	account_id
1	123 Fake St	Toronto	1
2	45 Charles Ln	Chicago	2
3	887 Mary Cres	Dublin	4

Ah! Isn't that nice? A visualization of what I've been talking about. So what we can see in the image above is that there's a link between these two tables via the account_id. In the Account table, rows 1 and 2 match to rows 1 and 2 of the Address table, and row 3 of the Address table matches to row 4 of the Account table. I can say this because I'm just matching the account_ids together in both tables.

So if I were to "Join" these two tables together, what would that look like?



Account joined to Address

account_id	firstname	lastname	address_id	street	city
1	Trevor	Page	1	123 Fake St	Toronto
2	John	Doe	2	45 Charles Ln	Chicago
4	James	Smith	3	887 Mary Cres	Dublin

You see? It's just a plain old horizontal matching of the rows in each table based on their relationship (which is the account_id).

That is the foundational "behavior" of relational databases. With this simple technique of joining tables together, we can build fast, efficient, powerful and robust databases for our applications.

Joining Syntax

Here's an example of how we would read data from these tables by joining them together:

```
select * from Account

inner join Address on Account.account_id = Address.account_id;
```

This will actually join ALL the data from the "Account" table to the "Address" table just like we saw above… but what if we just want to get the information for a single account? Well we would just add a 'where' clause to our SQL like so:

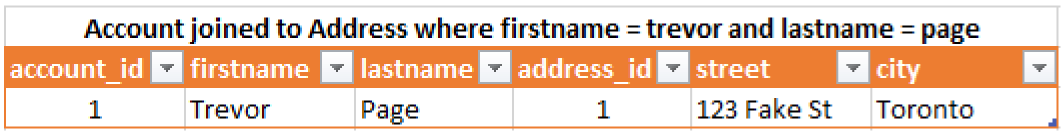
```
select * from Account

inner join Address on Account.account_id = Address.account_id

where Account.firstname = 'trevor'

and Account.lastname = 'page';
```

And here's the result of that query:



Account joined to Address where firstname = trevor and lastname = page

account_id	firstname	lastname	address_id	street	city
1	Trevor	Page	1	123 Fake St	Toronto

Flavors of SQL

It's important to note that there are many flavors of SQL out there. What I mean by that is there are different syntaxes available to use, and they depend on the type of relational database management system you're using.

Relational Database Management System (RDBMS)

When you "use" a database, you're really using a relational database management system (an RDBMS). The job of an RDBMS is to do exactly what it sounds like, manage your relational database. What that means specifically would be too lengthy to talk about at this moment, but what I can say is that your RDBMS is the system that will allow you to create a database on your computer and with the tables, rows and columns within it.

In the world of SQL, there are some big players trying to achieve the majority of the market share. This is the case for one typical reason: money! The more users a company can get, the more money they can make. Though different vendors have different business models, some of the biggest vendors of relational database management systems include:

- Oracle
- Microsoft SQL Server
- MySQL
- PostgreSQL

So, with all of these RDBMS to choose from, how can we possibly pick the right one? Well there are obviously advantages and disadvantages to all of them. Here's a quick breakdown of which would best match your situation…

Oracle: Extremely expensive, but you get what you pay for. Typically used in large enterprise applications by companies with money burning a hole in their pockets. It's a very well-crafted RDBMS with tons of bells and whistles.

Microsoft SQL Server: Also comes at a fairly hefty price, but more of a bargain than Oracle. But again, you get what you pay for. I've used this RDBMS for quite a while and it definitely gets the job done.

MySQL: One of the most popular choices of databases, for one good reason. It's free! The open source MySQL community server is widely used and there is a wealth of information available on its use. So if you ever run into any issues, I can almost guarantee there's an article on the internet that will help you solve your problem. Although this RDBMS is free, it also has a paid version that you can upgrade to if you need the extra bang for your buck. This is the RDBMS that we'll be using in this book.

PostgreSQL: Also a free RDBMS option, I've tried using it before but found that there was less support online and less tools suited for its use. I quickly went back to MySQL.

Tools

Before we can get too deep into databases and SQL, you'll need to arm yourself with the right tools. Well, thankfully in the world of SQL there's really only two things you'll really need to use. That's the actual MySQL RDBMS itself, the program that allows you to manage your databases and SQL in a nice visual way.

The "visual management" tool that I would suggest using is known as TOAD for MySQL. Now there are plenty of free tools out there to get you started, but I like TOAD the most as it supports more than just MySQL. It supports all the major players in the SQL industry, so when you learn MySQL in this book, you are already one step ahead of the game if you need to jump to another flavor.

To make this process as simple as possible, I created a video which walks you through the installation of the MySQL RDBMS as well as where to get your copy of TOAD for MySQL.

To lay it out in a very simple manner, here's the breakdown of what you'll need to install:

1. The MySQL Community Server (available via http://mysql.com)
2. TOAD for MYSQL (available via http://software.dell.com/products/toad-for-mysql)

And as I already mentioned, to make the process of installing these things easier, here's a link to a video that shows you how to install and run both of these things:

http://youtu.be/DQUlyZAuJww

Data Types

Storing data in databases are what all the cool kids are doing these days, but not all data is created equal. This is why we have data types.

We need to tell our database what **kind** of data we'll be storing in it. Unfortunately the databases (RDBMS) are not really smart enough to figure it out on their own.

If this concept seems strange to you, then consider the following. You've got three different types of data you want to store:

1. Your social security number
2. A quote from your favorite author
3. The date when you need to renew your driver's license

If you were to just treat all of this data the same, then you'd probably just store it as a sequence of alphanumeric characters (numbers and letters). Storing alphanumeric data in databases is usually referred to as VARCHAR data. We'll get into why it's called that in a minute.

Now when you store data as VARCHAR data, you can't perform any mathematical or date operations on this data. Mathematical operations are reserved for data stored as a numeric type. Date operations (like figuring out if one date is earlier or later than another) are reserved for data stored as a Date type.

So when you're specific with the type of data you're storing, you essentially open up a whole new world of being able to interact and modify that data.

Common Data Types in MySQL

So in this book we'll be focusing on the MySQL relational database management system (RDBMS). Given that this is the case, we will be looking at the commonly used data types within MySQL. So here they are in no particular order:

- **INT** – a standard integer with range of (-2^{31}) to (2^{31} - 1)
- **BIGINT** – a larger integer with range of (-2^{63}) to (2^{63} - 1)
- **DECIMAL (M, D)** – where M = display length of number and D = number of decimals to include
- **DATE** – Just a date stored in the format YYYY-MM-DD
- **DATETIME** – Date with Time stored in the format YYYY-MM-DD HH:mm:SS
- **TIME** – Time stored in the format HH:mm:SS
- **VARCHAR (M)** – Represents a String type where M = length of String

INT

The INT data type is used to store whole numbers (so, no fractions in the INT type) and it's one of the most commonly used. It's typically used to keep track of things like counts, primary key IDs or just anytime you want to keep track of any kind of number that doesn't require decimal point accuracy.

Again, the range of the INT data type goes from (-2^{31}) to ($2^{31}-1$), and if you're unsure of what that looks like, that's -2,147,483,648 to 2,147,483,647.

BIGINT

The BIGINT data type is exactly the same as the INT data type, with one exception. It's BIG! Where the regular INT only ranges from the -2 billion mark to the 2 billion mark (so about a range of 4 billion on the number line) the BIGINT's range is from (-2^{63}) to ($2^{63}-1$). And again, for those of you who are interested, those numbers look like this:

From -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

I'm not even sure I know how to pronounce the number range the BIGINT gets into. So, for the majority of the cases, the plain old INT data type should do the trick.

Other INTs?

And for those of you who are astute, you might ask "If there's a BIGINT, does that mean there's SMALLINT?"

Why yes, there is a SMALLINT, there's even a MEDIUMINT and a TINYINT. You see, the INT stores 4 bytes of data, and each smaller increment stores one less byte. So the whole picture actually looks like this:

Data Type	Range
BIGINT	-2^{63} to $2^{63}-1$
INT	-2^{31} to $2^{31}-1$
MEDIUMINT	-2^{23} to $2^{23}-1$
SMALLINT	-2^{15} to $2^{15}-1$
TINYINT	-2^{7} to $2^{7}-1$

Just remarkable stuff isn't it?

Okay not really, but it's good to know that you can choose as small or as large a data type for whole number as you'd like.

The benefit of choosing a smaller data type is that it will take up less space in the database itself. The obvious disadvantage is: What happens if you need to store larger numbers? Well then you'll need to change the column's data type. We'll get into that later, so no worries!

DECIMAL (M, D)

Alright so the name of this data type should be pretty self-explanatory. If you need to store a number that requires decimal place accuracy, then this is your cup 'o tea. The only question is, what the heck are the "M" and "D" for?

The “M” stands for the maximum number of digits that this particular column will store. The “D” stands for the number of digits to the right of the decimal point.

This means that if you declare your column to be of type DECIMAL (5,2), that means that your number can have at most 5 digits, 2 of which are to the right of the decimal place. So that would mean you could go from -999.99 to 999.99. If you declared it as DECIMAL (10,4), then your range would be from -999,999.9999 to 999,999.9999.

The common uses for the DECIMAL (M, D) data type is with storing monetary values, but obviously it can be used to store any type of numbers that require a certain decimal point of accuracy.

Once again, just RIVITING stuff huh?

Moving right along…

DATETIME

This is another very commonly used data type in MySQL. As you may have guessed, it’s used to store dates and times.

The only catch to this data type in MySQL is that you’ll need to store your data in a particular format for MySQL to properly parse your input. MySQL retrieves and displays DATETIME in the format ‘YYYY-MM-DD HH:MM:SS’

You can also store fractions of a second up to 6 decimal places like so: ‘YYYY-MM-DD HH:MM:SS.mmmmmm’ i.e. ‘2014-07-01 12:23:00.123456’

It’s also good to note that if you are ONLY interested in storing a date or you’re ONLY interested in storing a time, this is also possible.

You’d just use either the DATE or the TIME data types. Each of which are subject to the same rules as the DATETIME data type.

VARCHAR (M)

The VARCHAR data type is one that we’ve already heard about, and that’s because it’s probably THE most commonly used data type in MySQL. VARCHARs store alphanumeric data, like sentences, names, license plates, driver’s licenses, almost anything you can imagine. If you have a programming background, VARCHARs are like Strings.

The “M” in the declaration of the VARCHAR data type stands for the maximum number of characters in the column.

Also, VARCHAR means variable length, so you can have anywhere from 0 to ‘M’ characters. For example VARCHAR (5) could be “Hey!”, “”, “DUDE.”, “12345”, “hmm” etc.

“M” can be set anywhere from 0 to 65,535

CHAR (M)

CHAR is similar to VARCHAR except that CHAR is fixed length. This means that a CHAR is right padded with spaces. So if you declared a data type of CHAR (5), then the equivalent examples you saw in the VARCHAR (5) would be stored in CHAR (5) as “Hey! ”, “   ”, “DUDE.”, “12345”, “hmm ” etc. The difference is that you’ll have some extra white spaces added to pad your data. Believe it or not, MySQL will actually chop off the extra white spaces when you retrieve the CHAR values from the database unless the PAD_CHAR_TO_FULL_LENGTH SQL mode is enabled. In any case, those are nitty gritty details that aren’t too important to cover at the moment, so let’s keep on trucking shall we?

Constraints in MySQL

The beauty of modern databases is that they not only get the job done quickly, but they also do a great job at maintaining data integrity.

What the heck do I mean when I say data integrity? I mean that the RDBMS maintains your data in such a way that it shouldn't become tainted or corrupt over the long term. It does this by enforcing things called constraints.

Constraints allow us to tell our database management system what kind of data makes sense, and what kind of data doesn't. The RDBMS uses a set of rules to ensure that the data being inserted, changed and deleted is always valid within the bounds of the constraints. It might sound like wizardry, but it's actually pretty straight forward. And believe it or not, you've already encountered a couple of these constraints!

Primary Key Constraint

In MySQL, when you declare a column as a primary key, you are automatically adding some primary key constraint logic to that column.

The main rule that the primary key constraint enforces is one of uniqueness. When you declare a primary key, the main task that the RDBMS carries out is to ensure that there are no duplicate rows in the given table. If at any point someone tries to insert a row of data with a primary key value that already exists in that same table, the RDBMS will throw an error and it won't insert any of the data.

Not Null Constraint

This constraint is also very straight forward. Its job as a constraint is to ensure that the values being inserted into the column that enforces the not null constraint, are indeed not null. This means that if at any point, someone tries to insert a null value (empty value) into the constrained column, the database management system will complain and stop the insertion (or alteration) operation.

Unique Constraint

The unique constraint in MySQL does not allow you to insert a duplicate value in a column. So if you were to apply the unique constraint to the "username" column, then MySQL would not allow two usernames to be the same in that table. You'd get some sort of error and MySQL would stop the insertion of the row with the duplicate username.

Also, more than one UNIQUE column can be used in a table. So you can go nuts with which columns you'd like to mark as being unique.

Note: MySQL ignores any null values and doesn't consider multiple null values to be in breach on the unique constraint. So the main difference between the unique constraint and the primary key constraint is that you're allowed to insert nulls with a unique constraint, but not a primary key constraint.

Foreign Key Constraint

The foreign key constraint is one that you're also already familiar with. This constraint allows us to establish relationships in our database tables. It achieves this by using a special rule unique to the foreign key constraint.

The insertion of a value into a column with a foreign key constraint is only considered valid if the corresponding value exists inside of the table that the foreign key references. This means that if you try to insert the value '5' into table B's foreign key constrained column, but the column in table A that the foreign key references doesn't contain any '5' values, the insertion will fail citing the foreign key constraint is not satisfied.

This rule ensures that you'll never be in a situation where you have "orphaned" rows.

This same rule is enforced if you were to try and delete a row from table A. The foreign key constraint will first check to make sure the "child" values that reference the foreign key are gone before allowing the parent row to be deleted. This is enforced because you'd never want to have a child row pointing to a parent that has been deleted, so you'll always need to delete the child rows FIRST, before deleting the parent… and MySQL ensures that this is always the case via the foreign key constraint.

Handy stuff!

Check Constraint

The Check constraint is a very powerful and flexible constraint that you can make use of. It allows you to supply your own set of logic to the constraint.

This is powerful because it is customizable. You can add check constraints to ensure that values are inside of specific ranges, like for example that dates being inserted are only in the future, or only in the past. You could also ensure that integer values are within certain ranges, or only positive or only negative. You can even specify that a column can only be one of a handful of VARCHAR values as well, handy when ensuring that when someone specifies a the name of a country, that it's a valid name.

Database Relationships – One to Many

We've talked about relational databases already, and we've learned why this type of database management really dovetails with the object oriented programming model (as Tables are very similar to Objects). So now I want to dive into the specifics when it comes to relationships.

Believe it or not, the relationship that we've looked at already is one of three different kinds of relationships. There are different kinds of relationships in relational databases because there are different kind of relationships in the real world.

The reason why it's important to talk about the different kinds of relationships is because it will have an effect on how you design your database tables. Each of the three different relationships defined in SQL will result in a different design for the tables which maintain the relationships with each other.

Let's have a deeper look into the relationship that you've already seen (but didn't know it). The One-to-Many relationship.

What are the different types of relationships in SQL?

There are three types of relationships you can have in SQL, they are:

1. One-to-Many
2. One-to-One
3. Many-to-Many

In this lesson we are going to be focusing on the One-to-Many relationship as it's the most commonly used in my opinion. Let's start off our talk by first exploring what a One-to-Many relationship looks like.

What does a One-to-Many relationship look like?

The typical example of a one to many relationship is when you're talking about Users and Addresses. Typically a User can have one or more addresses (perhaps a mailing address and a billing address). The key here is that (in this particular design) any ONE Address can only belong to ONE User and ONLY ONE User.

This means that, for any particular Address that you could pick from the database table, that Address will only belong (or map to) exactly one User. This is what makes the relationship a One-to-Many relationship.

Other real world examples could include:

- **One** Employer has **many** Employees
- **One** Guitar has **many** Guitar Strings
- **One** Car has **many** Seats

There are countless different real world examples that can be thought up for this One-to-Many relationship, the key thing to understand is when to choose this relationship in your program.

When to choose a One-to-Many relationship

You'll need to essentially ask yourself two questions to decide if you indeed want to implement a One-to-Many relationship:

Is there indeed a need for a "Many" side of the relationship? For example, what if your system only needs a User to have ONE Address? This would negate the use for a One-to-Many relationship, as any given User is required to input exactly one Address

Does the table (Object) on the "Many" side actually only map to 1 item in its related table? For example, what if you wanted to have any one particular address belong to multiple Users? Perhaps your system needs to keep track of which of its Users live at the SAME Address (like brother and sister, roommates or spouses). If this is the case, then you'd probably want to choose a Many-to-Many relationship

As you can see, there are many ways that you can implement a simple User -> Address relationship. You could make it a One-to-One, One-to-Many or Many-to-Many… it's all in the design of your application and how YOU want it to function.

These are the kinds of things you'll need to consider before committing to any particular table relationship.

How do you create a One-to-Many relationship in SQL?

Creating this One-to-Many relationship is all about your primary key <<**PK**>>. Let's say we have chosen to use the One-to-Many relationship for our User -> Address mapping.

We know that we need to have a primary key <<**PK**>> for **both tables**. So our database tables may look something like this:

Users
<<**PK**>> user_id [int (11)]
username [varchar(20)]
password [varchar(20)]

Address
<<**PK**>> address_id [int (11)]
street_address_1 [varchar(255)]
street_address_2 [varchar(255)]
region [varchar(50)]
zip_code [varchar(7)]
country [varchar(50)]

Important Note*: this hasn't yet defined our One-to-Many relationship! We still need to make use of something called our Foreign Key <<**FK**>>.*

To add our relationship to these tables, we'll need to add in a foreign key <<**FK**>>. This foreign key is what is used to create a "link" between our tables. The typical way this is done, is to insert your foreign key into the table that represents the "Many" side of the One-to-Many relationship.

In this case, the "Many" side is the Address table. So we'll need to add a link to our User table into the Address table. This can be done by inserting the primary key of the User table into the Address table.

Since we know that the primary key (by definition) will always only point to ONE unique row of data, this will be perfect for keeping track of which Address row is related to which User row.

Let’s put our User table’s primary key into the Address table:

Users
<<**PK**>> user_id [int (11)]
username [varchar(20)]
password [varchar(20)]

Address
<<**PK**>> address_id [int (11)]
<<**FK**>> user_id [int (11)]
street_address_1 [varchar(255)]
street_address_2 [varchar(255)]
region [varchar(50)]
zip_code [varchar(7)]
country [varchar(50)]

There we have it! We now have a design for our database tables that incorporates the One-to-Many relationship using a foreign key!

Database Relationships – Many to Many / One to One

Alright, so we've already learned a whole bunch about the most common database relationship that exists, the one-to-many relationship.

In this lesson we will be expanding on the topic of database relationships and touch on two that are less common but just as useful.

Many-to-Many Relationship

The many-to-many database relationship is used when you are in the situation where the rows in the first table can map to multiple rows in the second table… and those rows in the second table can also map to multiple (different) rows in the first table.

If that quick and dirty explanation is a bit too general, let's take a look at a real world example!

Let's think of **books** and **authors** and decide what that relationship looks like.
The first question we ask is: Is there indeed a need for a "Many" side of the relationship?

Which means: can an author create "many" books? The answer is definitely **Yes**!
The second question we ask is: Does the table (Object) on the "Many" side actually only map to 1 item in its related table?

Which means: can a book only be written by one author? The answer here is **No** many books have been published by multiple authors!

So this means that we're definitely in the many-to-many arena with this relationship.

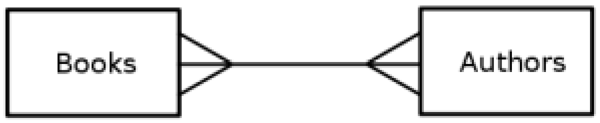


figure 1

How do you create a Many-to-Many relationship in SQL?

This is where things get slightly different from the more popular One-to-Many relationship.

A good design for a Many-to-Many relationship makes use of something called a **join table**. The term join table is just a fancy way of describing a third SQL table that only holds primary keys. You see, it's easy to draw out this relationship on paper, you can see an example of it in *figure 1* above. When it comes to creating this relationship in terms of SQL tables, just one more step is required.

First let's outline what the **author** and **book** tables could look like in SQL.

Author
<<**PK**>> author_id [int (11)]
first_name [varchar(20)]
last_name [varchar(20)]

Book
<<**PK**>> book_id [int (11)]
book_title [varchar(255)]
ISBN [varchar(255)]
version [varchar(10)]

Okay, so this is how the book and author tables could look like, but there's no relationship defined yet! So let's create one… since this is a Many-to-Many relationship and because I already mentioned that you'll need to use a join table when implementing a Many-to-Many relationship, let's see what this join table should look like.

Author_Book
<<**FK**>> author_id [int (11)]
<<**FK**>> book_id [int (11)]

A few things to note here:

By convention, the name of this join table is usually just the combination of the two tables of the many-to-many relationship. In this case it's just author_book, which implies that this is a join table since it's using the name of two existing tables joined by an underscore.

This join table only contains the primary keys from the author and book tables. Since this join table is referring to primary keys that **don't actually belong to the join table**, they are actually referred to as foreign keys.

Sometimes it's useful to assign a primary key column to a join table (i.e. `author_book_id [int (11)]`)

So now that we've created this join table, we will be able to easily create ANY relationship by inserting the appropriate rows into the join table. For example, if author "Trevor Page (author_id=14232)" created the book "How to Program with Java (book_id=9127329298)" then you could just insert the following row into the join table:

```
insert into author_book (author_id, book_id) values (14232, 9127329298);
```

So this will create a relationship between "Trevor Page" and "How to Program with Java", but let's say Trevor Page publishes another book (book_id=9315619872) and has some help from another author (author_id=14585) who also happens to have authored another book (book_id=8181225133), we can just insert those values into the join table to create that many-to-many relationship:

```
insert into author_book (author_id, book_id) values (14232, 9315619872);

insert into author_book (author_id, book_id) values (14585, 9315619872);

insert into author_book (author_id, book_id) values (14585, 8181225133);
```

So now we have author "Trevor Page" who owns two books. One of those books has a second author, and that second author also owns a book that "Trevor Page" does not.

Piece of cake right?

Not really I suppose, I had a lot of trouble figuring out the intricacies of the many-to-many relationship at first. So don't worry if you don't fully follow it, it'll come with time and practice!

One-to-One Relationship

Okay, so let's switch gears to the easiest relationship to understand. That's the One-to-One relationship. This one should hopefully be self-explanatory at this point, but if it isn't, I shall explain.

A One-to-One relationship means that you have two tables that have a relationship, but that relationship only exists in such a way that any given row from Table A can have **at most** one matching row in Table B.

A real world example of this could be the relationship between a person and a driver's license. Can one person have more than one driver's license? In the case of North America, the answer is no, any given person cannot have more than one driver's license. Well then, what's the reverse case? Can one particular driver's license be owned by more than one person? Again, in North America's case, the answer to that is no as well. One driver's license is assigned to one person, and ONLY one person.

So this means we can a One-to-One relationship. If I were to pick out ANY driver's license from a huge pile of driver's licenses, any individual license would point me back to ONE person in particular.

How do you create a One-to-One relationship in SQL?

The trick to creating a one-to-one relationship in SQL is to identify which table is on the "right hand side" or "child" of the relationship. This is usually done by deciding which object can exist without the other.

So ask yourself this question: Can a person exist without a driver's license? The answer is yes (I would hope)… then, can a driver's license exist without a person? I would say no, you cannot create a driver's license that doesn't belong to someone, it just wouldn't make sense.

So this is much like a parent/child relationship right? The parent in this case is the Person, and the child is the driver's license. You'll find that with the One-to-One relationship, this will be the case most of the time.

Since we've established that the driver's license is the "child" of this particular one-to-one relationship, we can move forward with our table design.

When designing the SQL tables for the one-to-one relationship, you'll need to make sure that the "child" table's primary key, is also the foreign key of the "parent" table. So this means that the driver's license table's primary key, should actually be the person table's key. It will look something like this:

Person
<<**PK**>> person_id [int (11)]
first_name [varchar(20)]
last_name [varchar(20)]

Drivers_License
<<**PK/FK**>> person_id [int (11)]
license_number [varchar(20)]
issue_date [datetime]
expiry_date [datetime]

So the important thing to note here is that the drivers_license table does **NOT** have its own `drivers_license_id` column, as that would break the design for a true one-to-one relationship.

How to Create a Table in Mysql

SQL Queries

You've learned all about how to create SQL queries to read, write, update and delete data… but you haven't yet learned how to create the tables where you'll be doing the reading, writing, updating and deleting.

So that's what today's lesson is all about, be sure to click the play button above this to listen to the show and then follow along with the notes below.

How to Create a MySQL Database

First things first, you need to make sure you've got some sort of application that you can use to manage your databases and your queries. I like to use TOAD for MySQL but it's obviously up to you which application you want to use. I even created a simple step by step video tutorial on how to install a database and to set up the TOAD program. I'd highly recommend watching this video even if you don't plan on using TOAD.

Once you've got your database installed and you've got your database management application up and running, it's time to create your first database table. Before we create a table, we need to create a database! Thankfully this is simple, all you need to do is write a one line script and run it within your database management application (i.e. TOAD):

```
create database test123;
```

So all that's going on here is that we're creating a new database, and it will be named `test123`, piece of cake! Once we run that code, we'll be ready to start creating tables inside of our new database.

How to Make a Database Table using MySQL

Next up is creating tables inside our new database. We're going to create two tables `Users` and `Address`, these tables have a one-to-many relationship. First let's create the `Users` table:

```
create table users

(

 user_id int(11) auto_increment primary key ,

 username varchar(20),

 password varchar(20)

);
```

Pretty straight forward, you'll notice that the `user_id` column has a few extra things added to it in the creation script. This is because it is the `primary key` of the `Users` table. When we use `auto_increment` it grants us the ability of not having to explicitly set a `user_id` every time we try to insert a row into that table… trust me, it's awesome.

Okay, so now let's create the `Address` table:

```
create table address

(

 address_id int(11) auto_increment primary key,

 user_id int(11),

 street_address_1 varchar(255),

 street_address_2 varchar(255),

 region varchar(50),

 zip_code varchar(7),

 country varchar(50),

 foreign key (user_id) references users (user_id)

);
```

This one is a bit more involved, but it still has the same concepts as the `Users` table with one exception. The `Address ` table declares a `Foreign Key` constraint.

This `Foreign Key` constraint will tell our database to automatically enforce our one-to-many relationship. What will happen now is that when we try to insert a row into the `Address` table, your new database will automatically check to make sure that the `user_id` you've specified **does indeed exist** inside the `Users` table.

Testing Foreign Key Constraint

There's a simple way to test that your new `foreign key` is set up correctly. Once you've run the table creation scripts above and your tables were successfully created inside your test database, you can try to insert data into the tables.
Try and run the following insert statement… **you should get an error**.

```
insert into address (
  user_id
 ,street_address_1
 ,street_address_2
 ,region
 ,zip_code
 ,country
) VALUES (
  1  -- user_id - IN int(11)
 ,'123 Fake St' -- street_address_1 - IN varchar(255)
 ,'Unit 283' -- street_address_2 - IN varchar(255)
 ,'Beverly Hills' -- region - IN varchar(50)
 ,'90210' -- zip_code - IN varchar(7)
 ,'USA' -- country - IN varchar(50)
);
```

If all goes well you'll see this error:

```
Lookup Error - MySQL Database Error: Cannot add or update a child row: a foreign key
constraint fails (`test123`.`address`, CONSTRAINT `address_ibfk_1` FOREIGN KEY
(`user_id`) REFERENCES `users` (`user_id`))
```

Or something similar to that message. This is your `foreign key` hard at work. It won't let you insert a row into the `Address` table without first having inserted a row into the `Users` table with the corresponding `user_id` of `1` (which is what we specified in the insert statement above).
So now to get things working properly, you should populate some data in the `Users` table first:

```
insert into users (
  username
 ,password
) VALUES (
  'username1' -- username - IN varchar(20)
 ,'password1234!' -- password - IN varchar(20)
);
```

Now that you've inserted a user into the `Users` table, now you can insert your address row:

```
insert into address (
  user_id
 ,street_address_1
 ,street_address_2
 ,region
 ,zip_code
 ,country
) VALUES (
  1  -- user_id - IN int(11)
 ,'123 Fake St' -- street_address_1 - IN varchar(255)
 ,'Unit 283' -- street_address_2 - IN varchar(255)
 ,'Beverly Hills' -- region - IN varchar(50)
 ,'90210' -- zip_code - IN varchar(7)
 ,'USA' -- country - IN varchar(50)
);
```

Done and done, you've now successfully created two tables, enforced the one-to-many relationship and populated data into both tables!

Congrats you smart little cookie you 😃

Assignment #1

You now have enough information to solve Assignment #1. To get access to all the assignments and bonus video content just go to http://dbvideotutorials.com/ebook-bonuses

You'll be emailed a copy of all the assignments and some bonus video content including the topics of database normalization and indexes. With this bonus content you'll learn the secrets behind how to ensure your database designs are rock solid, and how to ensure that your queries are as fast as they possibly can be!

Enforcing Database Relationships with SQL

In this lesson we're going to learn how to enforce our SQL relationships that we've already learned about. We're going to be tackling the one-to-one and many-to-many relationships and we're going to learn how to write the code to enforce these relationships in our database.

If you're not familiar with database relationships at all, I'd suggest you go back to the previous lesson where I introduce the concept of the database relationship.

Many-to-Many SQL Code

As outlined in this lesson, we are going to be focusing on the many-to-many relationship with the `author` and `book` example. Remember that one author can publish many books, and one book can be written by many authors. This indicates a many-to-many relationship and I'm going to show you how to enforce that relationship in your database.
Here's the outline of how you would create these two tables:

```
create table author

(

  author_id int(11) auto_increment primary key,

  first_name varchar(20),

  last_name varchar(20)

);


create table book

(

  book_id int(11) auto_increment primary key,

  book_title varchar(255),

  ISBN varchar(10),

  version int(11)

);
```

As you can see, there's no indication of any relationships between these two tables. The `author` table only contains information about the authors and the `book` table only contains information about books.

So where does the enforcement of the many-to-many relationship come from? Well, it comes from a third table which is known as the **join table**.

When creating the join table, it's good practice to give it the name of the two combining tables… so since we're combining the `author` and the `book` tables, the join table should be called `author_book`. Here's what the table creation code looks like:

```
create table author_book

(

  author_id int(11),

  book_id int(11),

  foreign key (author_id) references author (author_id),

  foreign key (book_id) references book (book_id)

);
```

There are two things that need to be mentioned here. The first thing that you should notice is that we did not assign either an `auto_increment` or a `primary key` property to either of the two columns in this join table. The reason for this is that these two columns are `foreign key`s and you should never assign an `auto_increment` to a foreign key. It also wouldn't make sense to assign the `primary key` property to either of these columns, as neither one of them on their own would make up a unique identifier for the table… you'll see why in a minute.

The second thing you should take notice of in this table definition is that we are assigning TWO `foreign key`s in the same table. I don't think I explicitly said this, but it is completely legal to have more than one `foreign key` in a table. What's not legal is having more than one `primary key`.
Cool? Cool!

Alright, now let's take a look at what kind of data we can put into these tables. I'm going to create a scenario where we have three different authors and four books. Here's what that could look like:

```
insert into author (first_name,last_name) values ('Trevor','Page');

insert into author (first_name,last_name) values ('Jane','Doe');

insert into author (first_name,last_name) values ('Jack','Johnson');
```

```sql
insert into book (book_title, ISBN, version) values ('How to Program with Java 2nd Edition', '1857283449', 2);

insert into book (book_title, ISBN, version) values ('How to Program with Java', '1841268719', 1);

insert into book (book_title, ISBN, version) values ('Cooking for Dummies', '1195348425', 1);

insert into book (book_title, ISBN, version) values ('How to Play Guitar', '0584237850', 3);


insert into author_book (author_id, book_id) values (1,1);

insert into author_book (author_id, book_id) values (1,2);

insert into author_book (author_id, book_id) values (2,3);

insert into author_book (author_id, book_id) values (3,4);
```

The inserts for the first two tables are pretty straight-forward and I don't think I need to say much about them. The only thing that you need to know is that the primary keys will be automatically assigned for the two tables with each insert. So when we insert the three authors, each will receive a unique id… if this is the first time we're doing inserts on the `author` and `book` tables, then they will receive ids starting with number `1`. This means that `Trevor Page` will receive `author_id = 1`, `Jane Doe` will be assigned `author_id = 2`, and `Jack Johnson` will get `author_id = 3`. The same logic will apply to the `book` table entries (i.e. `book_id 1, 2, 3, 4`).

Knowing that each row will receive a unique identifier, we can now see what the entries in the `author_book` are doing. We are assigning author `Trevor Page` to books `How to Program with Java 2nd Edition` and `How to Program with Java`. Author `Jane Doe` is being assigned to book `Cooking for Dummies` and author `Jack Johnson` is assigned to `How to Play Guitar`.
Make sense?

Also, for those of you who are following along with this example by creating tables and inserting this data into your own database, here's the SQL script that you would use to query the tables by joining to the join table:

```sql
select author.first_name, author.last_name, book.book_title From author

join author_book on author.author_id = author_book.author_id

join book on book.book_id = author_book.book_id;
```

This query will allow you to see all of the authors and the books that they wrote by making use of our join table.

One-to-One SQL Code

Okay, so let's move onto the one-to-one relationship and how to recreate this in our database.

First thing to remember is that one-to-one means exactly what it says, only one record from one table can map to one record from another. In our case we are going to use the example of a `person` and a `drivers_license`. A `person` can either have zero or one `drivers_license`, a `person` cannot have MORE than one `drivers_license`.
Having laid out those rules, let's take a look at what we would need to do to create these two tables and enforce the relationship:

```
create table person

(

  person_id int(11) auto_increment primary key,

  first_name varchar(20),

  last_name varchar(20)

);


create table drivers_license

(

  person_id int(11) primary key,

  license_number varchar(20),

  issue_date datetime,

  expiry_date datetime,

  foreign key (person_id) references person (person_id)

);
```

The most important thing to note here is the `drivers_license` table's `primary key`. You may have expected to see the `primary key` be a "drivers_license_id", but in our case we don't even have that as a

column. This is a special property of the one-to-one relationship… to enforce this relationship, we assign a `foreign key` as the `primary key`.
You should also note that we didn't assign an `auto_increment` to the `drivers_license` table. This is the same reason as in our many-to-many example, we don't assign an `auto_increment` to a `foreign key`.
Let's take a look at what this would look like with some real data:

```
insert into person (first_name,last_name) values ('Trevor','Page');

insert into person (first_name,last_name) values ('Jane','Doe');

insert into person (first_name,last_name) values ('Jack','Johnson');


insert into drivers_license (person_id, license_number, issue_date, expiry_date) values (1, 'P7293-28745-92387', '2007-12-18', '2022-12-18');

insert into drivers_license (person_id, license_number, issue_date, expiry_date) values (2, 'D4982-12684-48795', '2009-02-28', '2015-02-28');

insert into drivers_license (person_id, license_number, issue_date, expiry_date) values (3, 'J8984-49822-32156', '2012-07-19', '2017-07-19');
```

Here we're using the same people as we did in the "author" example… don't worry about the fact that the names match, this was just for the sake of reusing the same names and saving myself from thinking up three new names. As far as SQL is concerned, these three people are completely different people (as they exist in a completely separate table).

What is actually interesting to note, is that these three people will be automatically assigned unique IDs (just like in our many-to-many example above). So this means that the "people" will have IDs `1,` `2` and `3` assigned to them from first insert to last insert.
Knowing this, we can figure out what's going on when we look at the insert statements for the `drivers_license` table. We can see that `person_id` `1` is receiving `license_number` `P7293-28745-92387`. This means that `Trevor Page` will be associated with the `license_number` `P7293-28745-92387` (and so on).
To have a look at the results of this data, just execute the following query:

```
select person.first_name, person.last_name, drivers_license.license_number from drivers_license

join person ON person.person_id = drivers_license.person_id;
```

Assignment #2

You **may** have enough information to solve Assignment #2. In order to complete Assignment #2, you'll need to have read all the material in this book (up to this point) as well as viewed the bonus video on database normalization. To get access to all the assignments and bonus video content (including the video on database normalization needed to complete Assignment #2) just go to http://dbvideotutorials.com/ebook-bonuses

You'll be emailed a copy of all the assignments and some bonus video content including the topics of database normalization and indexes.

SQL Joins

There are three categories of joins that you can make use of in SQL:

1. Inner Join
2. Outer Join
3. Cross Join

But before we dive into the categories of joins, we first need to have an understanding of what a join really is.

Joins are used in SQL to bring together all the relevant data from multiple database tables. Remember that we've broken data down into multiple tables and established relationships between the tables.
An example we've used in the past were the author and book tables. We are going to switch things up in this lesson and change the relationship of the author and book tables to be a one-to-many relationship (as opposed to the previous many-to-many relationship)

So what does the relationship between author and book look like as a one-to-many relationship?

Author
<<**PK**>> **author_id** [int (11)]
first_name [varchar(20)]
last_name [varchar(20)]

Book
<<**PK**>> book_id [int (11)]
<<**FK**>> **author_id** [int (11)]
book_title [varchar(255)]
ISBN [varchar(255)]
version [varchar(10)]

The relationship between these two tables is based on the author_id column. So how does this relate to the topic of joins?

Let's say we're in a situation where we want to ask the question "Which books did which author create?" We answer this question with a join. When we join the tables together, we marry the data together (smash it together) based on the existing relationship the tables have with each other. If there is no relationship, then we can't join the tables… make sense?

SQL Inner Join

So, the first type of join I want to talk about is one of the most commonly used joins in SQL, and that's the inner join. The inner join matches data from both tables with each other, and if there isn't a match, then there is no data returned. This is an important concept to understand, as it's what separates the inner join from the other types. So I'll say it again, the SQL inner join will not return a row if there is no match from one table to the other.

Here's an example that should illustrate this concept nicely:

Author

Author_id	First_Name	Last_Name
1	Trevor	Page
2	Jane	Doe
3	Jack	Johnson

Book

Book_id	Author_id	Book_title
1	1	How to Program with Java 2nd Edition
2	1	How to Program with Java 1st Edition
3	3	How to Play Guitar

Let's assume that the data above is what's stored in our database (in the `author` and `book` tables). What would happen if we invoked an inner join like so:

```
select * from author
inner join book on author.author_id = book.author_id;
```

The results of this query will follow the rules that I outlined above… if there is no match (based on the relationship) then it won't return any results. So because of this fact, there will only be three rows returned (instead of 4) because the `author` "Jane Doe" doesn't have a corresponding `book`.
Let's see what we get when we run this SQL script:

Author_id	First_Name	Last_Name	Book_id	Book_title
1	Trevor	Page	1	How to Program with Java 2nd Edition
1	Trevor	Page	2	How to Program with Java 1st Edition
3	Jack	Johnson	3	How to Play Guitar

So as you can see, since we've used an inner join, author `Jane Doe` doesn't even show up as a result because she doesn't have a matching row in the `book` table.
Make sense?

SQL Outer Join

Okay, so now let's move onto the second most used join type, the SQL outer join.

Let's say that we are in a situation where we want to see all the rows from the author table as well as any pertinent "joined" data from the `book` table. How can we accomplish that?
Outer joins of course!

SQL outer joins actually break down into three categories:

Left Outer Join

Right Outer Join

Full Outer Join

The most commonly used of these three (by far) is the left outer join, so that's the one that we will be focusing our attention on.

As I mentioned above, the left outer join is useful when you don't want to exclude any rows from one table if there doesn't happen to be any data to join to. So let's say we want to see all the authors (including the authors that haven't yet published a book), and we also want to see the book titles that belong to those authors who HAVE published a book. This requirement demands the use of a left join. Here's what it would look like:

```
select * from author
left join book on author.author_id = book.author_id;
```

Here's what the results of this query would be:

Author_id	First_Name	Last_Name	Book_id	Book_title
1	Trevor	Page	1	How to Program with Java 2nd Edition
1	Trevor	Page	2	How to Program with Java 1st Edition
2	Jane	Doe	null	null
3	Jack	Johnson	3	How to Play Guitar

Can you spot the difference in the results that were returned from the left join vs the inner join?

The difference is that with the left join we actually get the `Jane Doe` row to display… however, you'll notice that instead of displaying information about Jane's book, you see `null`. This is because Jane doesn't have any matching rows in the `book` table, so there's nothing for SQL to display in these `book` columns.
So to sum up, the when there's no matching data, the left join will display `null`s but still return a result, whereas the inner join will exclude the entire row altogether.

SQL Aggregate Functions

In this lesson, you'll be learning all about the aggregate functions that exist in SQL.

What the heck is an aggregate function? Well that's what I'm going to try and teach you, and I promise, it's not a difficult concept to grasp. Just think of an aggregate function as a method that you're calling that will process data in your database and return a value. Obviously the returned value will depend on which of the aggregate functions you choose to use.

So that begs an obvious question, what are the aggregate functions that we can use in SQL? I'm glad you asked, here's the ones that I use all the time in MySQL:

`MAX`

`MIN`

`SUM`

`AVG`

`COUNT`

Okay, great! So now we know what the names of these functions are, now let's see some examples of them in use!

Well, before we jump into what all these functions do, we need to have a table to test all this stuff on right? So let's create a table that just holds some simple data points:

```
create table data_points
(
  data_points_id int(11) auto_increment primary key,
  data_point int
);


insert into data_points (data_point) value (1);
insert into data_points (data_point) value (2);
insert into data_points (data_point) value (3);
insert into data_points (data_point) value (4);

insert into data_points (data_point) value (5);

insert into data_points (data_point) value (6);

insert into data_points (data_point) value (7);

insert into data_points (data_point) value (8);
```

Sweet! Now we've got a table that will hold some simple data points (in the form of `ints`). Now let's talk about our aggregate functions.

SQL Max

What this particular function does is to identify the maximum value in a given set of values (in this case our set of values is whatever we're choosing to select from in our database table).

So, given that we have our `data_points` table, let's try and see what the `MAX` value is in that particular table:

```
select max(data_point) from data_points;
```

When we run this code, we get one value returned, the value `8`. This makes sense, as the number 8 is the largest value we have stored in the table. Pretty straight forward right?
So that should go without saying that we can find the minimum value just as easily by running this code:

```
select min(data_point) from data_points;
```

And as you would guess, we get the value `1` returned when we run the script above.
Piece of cake right? Well, let's get a little bit more complex. What if our table had more than just numeric values? What if our table had `varchar` values in a particular column? Let's test that scenario by adding a `varchar_data_point`column to our table. We'll just write a script to `alter` the existing table to add an additional column:

```
alter table data_points

add column varchar_data_point varchar(20);
```

This is a new piece of code, we haven't seen how to edit a table just yet. All we've seen is how to create and delete tables. Well it's pretty straight forward to edit a table, you just tell it what table you want to edit, then tell it what you want to add or remove from the table. We'll be diving into this later in the book, so don't worry too much about the syntax just yet.

Now, let's get back on track and focus on putting some `varchar` values in the table and run our `MAX` aggregate function on it.

```
insert into data_points (varchar_data_point) value ('Trevor');

insert into data_points (varchar_data_point) value ('Bob');

insert into data_points (varchar_data_point) value ('Zoolander');

insert into data_points (varchar_data_point) value ('Eric');

insert into data_points (varchar_data_point) value ('Christina');

insert into data_points (varchar_data_point) value ('Vanessa');
```

Okay, so now we've got some real data in the `varchar_data_point` column. So let's take a look at what happens when we try to find the max value:

```
select max(varchar_data_point) from data_points;
```

When we run the above script, the result is `'Zoolander'`. You see, SQL realizes that this particular column is of type `varchar` so it changes the algorithm for finding the maximum value by first sorting the values in alphabetical order and picking the last one in the list.
Neat!

So then we can probably figure out what we'll get when we look for the `MIN` value right?

```
select min(varchar_data_point) from data_points;
```

This query returns `'Bob'`, which makes sense given the fact that it'll sort the column alphabetically. Alright, so we've got a good understanding of how to use SQL Max and SQL Min, now let's look at the other functions.

SQL Sum and SQL Avg

These two functions are also very easy to understand. Performing a SQL Sum just means it's going to add up all the values in the column that you give to it, likewise the SQL Avg function will add up all the values and then divide by the number of values added.

So let's take a look at the syntax for running these functions:

```
select sum(data_point) from data_points;
```

Unsurprisingly, the result of this is `36`, which is just the addition of each of the data points that we inserted at the beginning of this lesson (the numbers 1 through 8).
Let's take a look at finding the average of all these numbers:

```
select avg(data_point) from data_points;
```

When you run the average, all it does is divide the sum by the number of values, which is 36 divided by 8. So the resulting value is `4.5000`, piece of cake!
The only special thing to note here is that in MySQL, it only keeps 4 digits of accuracy in the fraction part of the number. If you want to change the accuracy of the returned value, you can tweak it with the `cast()` function. Here's an article on it if you're interested: www.w3resource.com/sql/aggregate-functions/avg-decimal-places-using-cast-within-avg.php
Also, one last thing to note is that if you don't specify a column with numeric values, you'll likely just get the value of `0`returned to you… as there were no numbers to sum up.

SQL Count

This brings us to the final aggregate function that we'll talk about today. The SQL Count function is quite useful, as it can quickly count the number of rows that exist in a table (either specifically by column name, or even as generic as ALL of the columns in the table).

Here's an example of counting the entries of JUST one specific column in our table:

```
select count(data_point) from data_points;
```

This query will return the value of `8`. Now if you had a keen eye, you may have guessed that it would return a value of`14` because we actually have 14 rows in total in our `data_points` table. Eight of those rows have values in the `data_point` column and six have values in the `varchar_data_point` column (for a total of 14). But, I assure you that it will return `8` as the count.
So why is this?

It's because we specifically asked it to count the `data_point` column by specifying `select count(data_point)`! Now if we changed it to count the `varchar_data_point` values, what would it return?

```
select count(varchar_data_point) from data_points;
```

As expected, it returns `6`, since there are only six varchar data points in our table.
Great, so now how do we count ALL the rows in the table? We use an asterisk in place of our column name like so:

```
select count(*) from data_points;
```

This query will count and return the TOTAL number of rows in the table, which in this case is `14`!

Now it's your turn to play around with aggregate functions

Be sure to use all the scripts I've included throughout this lesson and create your own table with data. Mess around with the data and play with the aggregate functions to see if you can get it to do anything unexpected.

SQL Group By

After having talked about all the SQL Aggregate functions, there's one more topic that goes hand in hand with what we've already learned… The `group by` keyword.

This particular keyword allows us to take a bunch of data and mash it all together into matching groups and then perform aggregate functions on those groups (like `sum` and `avg`).

You might ask yourself why you'd want to "mash together" a bunch of data. The answer to this is best explained with an example, but let me try to put it in regular words before we jump into our example. Grouping data together allows us to look at aggregate data in relation to unique piece of data (or rows), a typical use case would be to group all the matching data together so you can get a count of the number of occurrences of specific data. An example related to grouping and counting could be a presidential election, you'll have all the votes in a database and you'll want to group that data together to get the total votes for each unique candidate.

Grouping and counting is a very simple use case for the `group by` keyword, so the example that we'll be looking at in this lesson will be related to bank accounts and transactions. With bank accounts and transactions within those bank accounts, we can start to use the more complex grouping methodologies (which we'll be talking about later).

Why use SQL Group By?

Grouping data together is extremely valuable in many ways. You'll see a few examples here that relate to bank accounts and transactions within those bank accounts.

Let's assume we have three different bank accounts and each has both credit and debit transactions. We would also like to see a quick summary of all the credits and all the debits for each bank account. This kind of "requirement" lends itself perfectly for grouping.

So let's first set up our example case in our database. We'll create the tables and columns as follows:

```
create table bank_account
(
  bank_account_id int(11) auto_increment primary key,
  bank_account_number varchar(20)
);

create table transaction
(
  transaction_id int(11) auto_increment primary key,
  bank_account_id int(11),
  transaction_date date,
  debit_amount numeric(10,2) default 0.0,
  credit_amount numeric(10,2) default 0.0
);
```

Okay, so with our two tables created, now we need to input some test data to start demonstrating how the SQL `group by` keyword works.

```
insert into bank_account (bank_account_number) values ('A-283748293'); --
this will become bank_account_id 1
insert into bank_account (bank_account_number) values ('B-174984638'); --
this will become bank_account_id 2
insert into bank_account (bank_account_number) values ('C-927461738'); --
this will become bank_account_id 3

insert into transaction (bank_account_id, credit_amount, transaction_date) values (1, 250.12, '2014-
06-10');
insert into transaction (bank_account_id, debit_amount, transaction_date) values (2, -123.93, '2014-
06-10');
insert into transaction (bank_account_id, credit_amount, transaction_date) values (3, 832.11, '2014-
06-11');
insert into transaction (bank_account_id, debit_amount, transaction_date) values (1, -100.32, '2014-
06-10');
insert into transaction (bank_account_id, credit_amount, transaction_date) values (2, 322.33, '2014-
06-11');
```

```
insert into transaction (bank_account_id, credit_amount, transaction_date) values (3, 131.92, '2014-06-11');
```

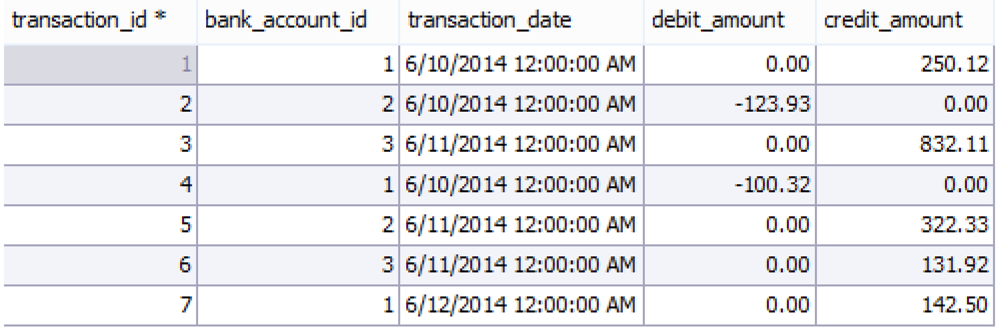
```
insert into transaction (bank_account_id, credit_amount, transaction_date) values (1, 142.50, '2014-06-12');
```

The SQL script above has just inserted three bank accounts into the `bank_account` table, and then it inserted a few transactions (both debit and credit) into each of the three bank accounts.
Alright, so with all of this data, it's easy to get a bit lost in the details of what's going on, right? Wouldn't it be nice if we could just get a breakdown of the balance of each account, based on all the transactions in our database table?

Well, that's actually pretty simple to do, as you would just perform a `SUM` operation on the `credit_amount` and `debit_amount` columns. You've already learned how to do this in the last SQL tutorial lesson on aggregate functions. But we also need to add the grouping logic in there to see the balance of each account.
Before we do that, let's take a look at how the data looks "as is" **without** grouping anything together. We'll execute a simple `select` statement on the `transaction` table:

```
select * from transaction;
```

After running this query, here's what the resulting values should look like in the `transaction` table:



transaction_id *	bank_account_id	transaction_date	debit_amount	credit_amount
1	1	6/10/2014 12:00:00 AM	0.00	250.12
2	2	6/10/2014 12:00:00 AM	-123.93	0.00
3	3	6/11/2014 12:00:00 AM	0.00	832.11
4	1	6/10/2014 12:00:00 AM	-100.32	0.00
5	2	6/11/2014 12:00:00 AM	0.00	322.33
6	3	6/11/2014 12:00:00 AM	0.00	131.92
7	1	6/12/2014 12:00:00 AM	0.00	142.50

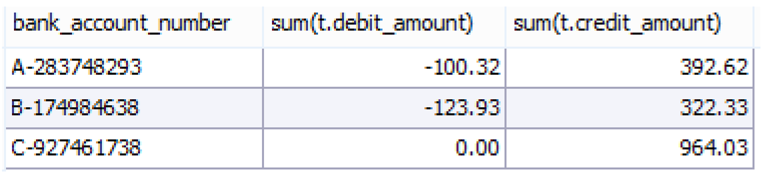
Do you see how the data is stored in the `transaction` table? Each individual transaction is given a date, it's associated with a specific bank account, and it's either marked as a debit or a credit based on which of the two columns are filled in. If it's a credit transaction, then the corresponding `credit_amount` column will have a positive dollar amount inserted, also the `debit_amount` column is set to 0.0 by default (as is seen in the table's definition where we set a default value for both the `credit_amount` and `debit_amount`)
But, like I said, this data is a bit too granular for our liking, so let's take a look at how we can group it together. Let's group this data by bank account number by using the `group by` keyword:

```
select b.bank_account_number, sum(t.debit_amount), sum(t.credit_amount) from bank_account b

join transaction t on t.bank_account_id = b.bank_account_id

group by b.bank_account_number;
```

After running this SQL script, here's what the resulting data looks like from the select:



bank_account_number	sum(t.debit_amount)	sum(t.credit_amount)
A-283748293	-100.32	392.62
B-174984638	-123.93	322.33
C-927461738	0.00	964.03

*****CRITICALLY IMPORTANT THINGS to note about the SQL script above are*****:
Since we are grouping by `bank_account_number` we are allowed to put it in the list of columns to `select` from.

Since we are using the `group by` keyword, we're also allowed to use the aggregate functions in our list of columns to `select` from.

We should not `select` any columns that aren't in the group by list of columns, unless of course we're using the aggregate functions. If we try to include data in our grouped query that doesn't exist in the grouping criteria, then our results will likely be strange and make no sense.

What this means is that because I've stated to `group by bank_account_number`, this means I **should not** also show values for something like the `transaction_date` as the value displayed won't necessarily make sense.

Grouping By Other Columns

Let's say that we're not interested in knowing the debits and credits for each bank account, let's say that we're only interested in knowing the total of both the credits and debits for any given date. How would we accomplish this?

Well, there are two requirements that need to be address in that previous statement. We'll need to:

`Group by transaction_date`

Add up the credits with the debits for each individual `transaction_date`

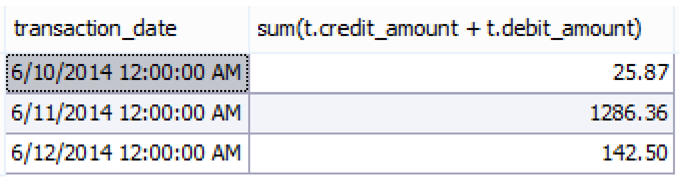
So, let's follow through on our new requirements with this SQL script:

```
select t.transaction_date, sum(t.credit_amount + t.debit_amount) from bank_account b

join transaction t on t.bank_account_id = b.bank_account_id

group by t.transaction_date;
```

You should note that we changed the `group by` to be `t.transaction_date` (instead of `b.bank_account_number`) and we've also used the plus (+) symbol inside of our `SUM` aggregate function to add up the debits and credits.
Okay, so what do the results of this query look like?



transaction_date	sum(t.credit_amount + t.debit_amount)
6/10/2014 12:00:00 AM	25.87
6/11/2014 12:00:00 AM	1286.36
6/12/2014 12:00:00 AM	142.50

So as you can see here, our database has taken all the data in question, grouped it together by the transaction dates that match, and then once it had them all grouped into their proper "buckets" it performs the aggregate `SUM` function to give us our final result.

How to Visualize the SQL Group By Command

So to summarize this tutorial, I'd like to leave you with the trick I use to help me understand what's going on when grouping data. Whenever you have a column listed in the `group by` area, just think of having buckets for that particular group of data. You'll have a new "bucket" for each unique piece of data inside of the column you're grouping on.

So if we're grouping by `bank_account_id`, then I would picture three different buckets (since we have three different bank accounts). Then for every transaction that belongs to the first bank account, we'll "throw" that transaction into the first bucket. For every transaction that belongs to the second bank account, we'll "throw" it into the second bucket, etc.

Once we're done putting our transactions into metaphorical buckets, we look at all of the transactions in each bucket and perform any aggregate functions on each… and voila! That's really all there is to grouping. For the sake of completion, I'd like to note that grouping can get more complex when grouping by multiple columns, so for example `group by b.bank_account_number, t.transaction_date`. But to figure this out, I'd just picture a bucket inside of a bucket. So first you determine which bank account a transaction belongs to and place it over the appropriate bank account bucket, but then look at the transaction date for it and place it into the appropriate `transaction_date` bucket that's within the `back_account_number` bucket.

Having Clause

The `having` clause is used to allow filtering on aggregate functions.

What happens when you want to only return results from a query where the sum of some amount is greater than, say, 80? You actually can't put an aggregate function inside of there `where` clause, you'll get an error from MySQL.

The situation is resolved by using the `having` keyword.

To learn all about this special filtering keyword, I've created a great video for you that you can access for free when you go to http://dbvideotutorials.com/ebook-bonuses

You'll receive an email with not only the video explaining how to use the `having` clause, but you'll also receive a bunch of other things like assignments and other bonus video content. It's all yours and it's all free!

Assignment #3

Now that you've received all of your bonus videos, go ahead watch the video on the `having` clause that I mentioned above. You'll need to learn how to do this before you attack assignment #3.

And as always, assignment #3 will be emailed to you once you've visited the special bonus section for this book via http://dbvideotutorials.com/ebook-bonuses

Database Views

A view is a special kind of statement that used to read data from your database. Think of a View as being a table in a database, only it's not an actual physical table. You can `select` data from this imaginary table, but you can't `insert` into it or `delete` from it. Views can be set up to read data from multiple tables as one select statement. This means that if you had a particularly helpful query that had to do a bunch of joins across several tables, this query could be simplified into a single view.

Views are especially helpful when creating reports about your data. This is the case because most of the time reports will combine data across numerous tables (a process known as denormalizing data). This can naturally lead to large queries that can become cumbersome to deal with. So these large queries can be condensed into a single view for ease of use.

Views are also helpful if you want to "hide" columns from a particular table. Views can be used to only return a subset of columns from a table (as opposed to every column in a table). So a view could be used to hide sensitive column data from certain people or users.

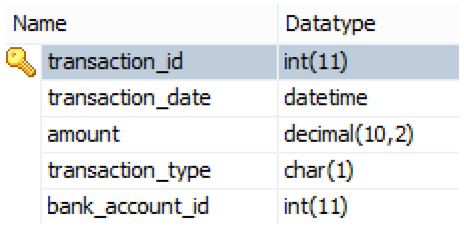
View Syntax

So let's take a look at the syntax involved in creating a simple view:

```
create view name_of_view as
select select_statement;
```

Not very complex, but it does need some explaining. You see, when you create a view, you're really just condensing a `select` statement into the name of the view. So this way you'll be able to `select` right from the view as if it were a table.

Let's take a look at a real world example. Let's assume that I have a table called `transactions` and it looks something like this:



Name	Datatype
transaction_id	int(11)
transaction_date	datetime
amount	decimal(10,2)
transaction_type	char(1)
bank_account_id	int(11)

So within this table we hold data related to the transactions within everyone's bank accounts. You can relate it to when you go out and buy a cup of coffee with your debit card, when you purchase that coffee, there will be a transaction inserted into the `transactions` table with information relevant to that purchase.

Here's a quick breakdown of each of the columns:

- transaction_id: the primary key of the transactions table
- transaction_date: a `datetime` type keeping track of the date and time of each transaction

- amount: a positive number with two fractional digits of accuracy
- transaction_type: ‘c’ for credit, ‘d’ for debit
- bank_account_id: foreign key pointing to the `accounts` table

Armed with this knowledge, let’s now assume that we’re in a position where we want to calculate the cash flow for each account. We want to be able to know at a quick glance if we’re in positive cash flow or negative cash flow. To figure this out we’ll need a complex query like so:

```
select
sum(if (transaction_type = 'D', amount*-1, amount)) as total,
bank_account_id from transactions
group by bank_account_id;
```

This `select` statement is a bit complex as it contains some conditional logic. What this query will do is add up all of the `amount` values for each bank account. It adds up the `amount` values in a specific way, as we know from the definition of the `amount` column above, it only holds positive numbers. So this wouldn’t be a good way to determine cash flow, as we’d always end up with a positive number if we just added up all the values. Instead, we have some logic that will multiply the `amount` by -1 if the `transaction_type` is ‘D’ (for debit). This way when we add up all the values, we’ll know a true tally of debits vs credits for each bank account.

Now having explained this query, you can see that it’s a bit of a pain to write out. It’s fairly complex and is a bit lengthy, so how can we take this query and make it easier to run?

We turn it into a view of course!

Let’s do that now:

```
create or replace view account_balances as
select
sum(if (transaction_type = 'D', amount*-1, amount)) as total,
bank_account_id from transactions
group by bank_account_id;
```

So you see here, we’ve just taken the complex query and plugged it into a “`create or replace view`” operation.

The reason why I’ve used “create or replace” is because you cannot create a view if it’s already been created once in your database. So if you ever need to make a simple modification to the definition of the view, you’ll need to drop the view, then re-create it. This operation can be simplified using the “`create or replace`” syntax.

So now once we run this view creation script, we’ll be able to `select` from the view just like it was a real table like so:

```
select * from account_balances;
```

How cool is that? All of that complex logic in the query is now simplified into one simple `select` statement!

Things to Note about Views

The view definition is “frozen” at the time of creation, so changes to the underlying tables afterward do not affect the view definition. For example, if a view is defined as `select *` on a table, new columns added to the table later do not become part of the view.

Also, the select statement within a view’s definition cannot contain a subquery, as that will cause issues with the execution of the view. If you don’t know what a subquery is, don’t worry, you’ll learn about it later in this book.

Altering Existing Tables

So you know how to create tables and populate them with columns and data types. But we haven't yet gone into detail about how to edit an existing table.

This is an important topic, as I'm sure you already realize. The whole reason why we'd like to edit tables is because, nobody is perfect! I will guarantee you that at some point in your life, you'll need to make a change to a table that you (or someone else) created in the past.

Thankfully, it's really not that difficult to make changes. The only thing you really need to know about is what exactly CAN be changed on a table? Here's a list of the most popular things you can do to an existing table in SQL:

1. Adding a new column
2. Deleting a column
3. Changing the name and data type of a column
4. Adding a Constraint
5. Removing a Constraint

Alright, so now that you know what you can do to modify existing tables, let's take a look at the actual syntax involved in each of these operations.

Adding a New Column

Adding a column is very straight forward, here's the generic syntax you'll need:

```
alter table table_name
add column column_name data_type;
```

Here's a real world example of adding a new column called `transaction_details` to an existing `transactions` table:

```
alter table transactions
add column transaction_details varchar(100);
```

Deleting a Column

Let's assume you've made an error in your design and you want to delete a column from your table. No big deal, here's the generic syntax:

```
alter table table_name
drop column column_name;
```

Let's take a look at a real example, let's say we now want to delete the `transaction_details` column we added above, here's the code you'd have to write:

```
alter table transactions
drop column transaction_details;
```

Rename Column and/or Change Data Type

Sometimes you find yourself in the position of needing to change the name of an existing column, this could be the case for many reasons:

- The column has a typo and needs fixing
- The column's name is really long and you've decided to shorten it
- The column wasn't named well and is confusing other programmers

Whatever the case is, renaming a column can sometimes be a bit dangerous. If you're using your database as part of a larger application, then there may be references to that column by its old name… thus if you change it, you could break the application. So whatever the reason is for changing a column name, just be sure to be careful.

Here's the relevant generic syntax for changing a column name:

```
alter table table_name
change column existing_column new_column_name new_datatype;
```

Now for a real world example. Let's assume we want to change the name of the `transaction_type` column, it needs to be changed to `the_transaction_type`. How would we accomplish this?

```
alter table transactions
change column transaction_type the_transaction_type char(1);
```

This will change both the name and data type of the existing column at the same time. If the existing column's data type was previously `char(1)` then nothing really changes, but if it was something different, then MySQL will try to change its data type as well.

Adding a Constraint

Sometimes you realize after you've created a table that you need to add new constraints. Constraints are things like foreign keys and primary keys, they constrain what data can be inserted into a particular table based on some predefined rules. Here's how to add them after you've already created the table without them:

```
alter table table_name
add constraint [constraint_type] [constraint_name] (column_name)
references another_table (another_column);
```

The constraint type and name are optional in this statement, but if you know the type of constraint you want to add, then you'll need to put in the appropriate value for it to be enforced. So for example, if you wanted to add a foreign key constraint to an existing column, then your syntax would look something like this:

```
alter table transactions
add constraint foreign key bank_account_id__fk (bank_account_id)
references bank_account (bank_account_id);
```

Deleting a Constraint

Sometimes you realize after you've created a table that you need to delete constraints. Here's how to delete them after you've already created the table with them:

```
alter table table_name
drop [constraint_type] constraint_name;
```

The only tough part about deleting a constraint is that you'll need to know the name of the constraint. This is why I'd always suggest specifying a name for your constraints when you create them, otherwise MySQL will pick one for you.

In the event that you allowed MySQL to pick the name of the constraint for you, you'll need to go digging to find out what that name actually is. If you're using the TOAD software you'll find the names of your constraints inside of the Object Explorer's "indexes" tab. You can then double click on any of the indexes/constraints in the list to find out what their names are.

The SQL Subquery

Now that you've learned about SQL Joins, aggregate functions and the group by keyword, it's time we moved on to our final topic in our SQL tutorial series. Today you'll be learning all about SQL Subqueries, how to use them and when you should use a SQL subquery.

So, what is a subquery?

First and foremost, let's get the jargon out of the way. A subquery can also be referred to as a nested query. It's just like having a nested `if` statement in your Java code. Essentially what you're doing with a subquery is you are constructing a regular old query (`select` statement) which could be run all by itself if you wanted to, but instead of running it all by itself, you're jamming it into another query (`select` statement) to give you more specific (filtered) results.
What's very important to note here is that the SQL subquery can almost always be re-written as a `join` with a `where` clause attached to it.
The advantage to using a subquery is that they tend to be easier to write and thus easier to understand if you're coming in as a new coder looking at an existing system. Of course, as is always universally true, when you have a piece of code that's easier to write and understand, it means that it's likely also less efficient. Unfortunately that statement holds true here as well. You can easily get carried away with SQL subqueries and start to write very inefficient scripts.

The rule I always live by is to write it so it works first, then refactor it to run faster and be more efficient. This is the test driven development mantra 😃

What does a SQL Subquery look like?

I love to teach by example, so let's take a look at one.

Let's say we're working with a database table that deals with people and addresses. A `Person` has a `name` and an `Address` has all the pertinent fields associated with it like so:

```
drop table if exists address;

drop table if exists person;


create table person

(

  person_id int(11) auto_increment primary key,
  first_name varchar(20),
  last_name varchar(20)
);

create table address
(
  address_id int(11) auto_increment primary key,
  person_id int(11),
  address_line_1 varchar(100),
  address_line_2 varchar(50),
  city varchar(50),
  region varchar(50),
  zipcode varchar(10),
  country varchar(50),
  foreign key (person_id) references person (person_id)
);

insert into person (first_name, last_name) values ('Trevor', 'Page');
insert into person (first_name, last_name) values ('Jane', 'Doe');
insert into person (first_name, last_name) values ('Jack', 'Johnson');
insert into person (first_name, last_name) values ('Jack', 'Black');
insert into person (first_name, last_name) values ('Dave', 'Matthews');
insert into person (first_name, last_name) values ('John', 'Doe');

insert into address (person_id, address_line_1, city, zipcode, region, country) values (1, '16 Dunwatson
 Dr', 'Toronto', 'Ontario', 'M1C 3L8' ,'Canada');
insert into address (person_id, address_line_1, city, zipcode, region, country) values (2, '1129 Bannock
 St', 'Denver', 'CO', '80204' ,'USA');
insert into address (person_id, address_line_1, city, region, country) values (3, 'Serna 307', 'Santa An
a', 'SON' ,'Mexico');
```

```
insert into address (person_id, address_line_1, city, zipcode, region, country) values (4, '462 Duke St', 'Glasgow City', 'G31 1QN', 'Glasgow','UK');

insert into address (person_id, address_line_1, address_line_2, city, zipcode, region, country) values (5, '1584 Fischer Hallman St', 'Kitchener', 'Unit 301', 'N2R 1C1', 'Ontario' ,'Canada');

insert into address (person_id, address_line_1, city, zipcode, region, country) values (6, '155 Northampton St', 'Rochester', 'New York', '14606' ,'USA');
```

Okay, so that script will create our database tables and populate it with some random information…
Now onto the fun stuff, the subqueries!

From the information stored in our database, let's say we want to isolate the people that live in certain countries. Let's say that we only want to see people who live in Canada, what would that query look like?

Here's the big reveal!

```
select * From person where person_id in (select person_id from address where country = 'Canada');
```

Fairly unexciting stuff. What you should note here is the use of the `in` keyword.
The `in` keyword is **one of the main keywords** that we can use to create our subqueries.

How does the SQL Subquery work?

Let's breakdown the query above: we are searching for any `Person` who lives in Canada. This is accomplished by writing two individual `select` statements:

```
select * from person
select person_id from address where country = 'Canada'
```

We then "join" these two queries together using the `in` keyword (thus utilizing the subquery structure).
Again, we want to know WHO lives in WHICH country. So that's why we have one query looking at the `Person` table and one query looking at the `Address` table. Make sense?
The next important thing to note about this subquery is the specific use of the `person_id` key. You'll notice that we said `where person_id in (select person_id from ...)`, when you're constructing subqueries and using the `in` keyword, you need to make sure that you're matching IDs together (just like if you were doing a join statement).

Actually, speaking of joins, **how about I show you what the equivalent query would look like as a join instead of a subquery?**

```
select person.* from person
join address on address.person_id = person.person_id
where address.country = 'Canada';
```

As I mentioned at the beginning of this lesson, you're usually able to write out a typical subquery as a join too, but sometimes the 'join' query can get a bit confusing. If you were to run the script directly above this paragraph, you'll see that the results are the same as the subquery script we were looking at earlier.

You should see this as a good thing, because if you can understand the 'join' script above, then you should understand how the subquery works. The 'join' statement invokes a join based on the `person_id` from both the `Person` and the `Address` tables… which is exactly what our subquery does using the `in` keyword… ie. `select * from person where person_id in (select person_id from address)`

Comparing SQL Subqueries to Joins

For the sake of clarity, how about I write the same query in both formats (as a join and as a subquery) and highlight which parts are functionally equivalent?

Sounds like a great idea, here you go:

```
select * from person
where person_id in
(select person_id from address
where country = 'Canada');
```

```
select person.* from person
join address on address.person_id = person.person_id
where address.country = 'Canada';
```

So as you can see above, the black parts are where we choose what we actually want to see once all the filtering is done. The blue parts are used to join the two tables together, and the red parts are what we use to filter our result set.

Multiple Subqueries

Alright, so hopefully you get the basic idea of how subqueries work… now how do we use multiple subqueries in the same query?

Well, let's say we want to know all the people who live in North America (i.e. USA and Canada combined). How would we go about doing this? We could just use a subquery inside of a subquery like so:

```
-- people who live in North America
select * From person where person_id in
 (select person_id from address where country in ('Canada', 'USA'));
```

Note the use of two different `in` keywords. This means that we have two subqueries running in one script, they are just nested within each other. You can nest your subqueries many times over (I think up to 16 or something ridiculous like that)… but with great power comes great responsibility. The deeper you go with nested subqueries, the more strain you could be putting on the database system. Of course, this kind of strain is all relative, because if you have a very small dataset, then you can likely get away with inefficient scripts and not have to pay the price. Typically what happens is that you do your best to write great code/scripts, and as your system grows, the inefficient pieces will make themselves known and you'll fix them as they come up… not the greatest approach, but that's just how the real world works.

Other Types of Subqueries

So we've talked a lot about using the `in` keyword, but there are others that can be used… for example, the `not in` keyword can be helpful.
What if you want to find all the people who live outside of North America?

```
-- people who live outside of North America
select * From person where person_id in
 (select person_id from address where country not in ('Canada', 'USA'));
```

Note the use of the `not in` keyword in this query… this will find all address rows that are not associated with Canada or USA, then it will join with the `Person` table via the `person_id` and return their names.

Easy peasy!

There are a few other examples of subqueries, but I don't tend to use them very often (if at all), but if you're interested in learning more, there's a great article on subqueries here for your reading pleasure. This article focuses on MySQL (just like mine does) and will give you a brief overview on all the different ways to write subqueries in MySQL.

Final Assignment

Ladies and gentlemen, now that you've made it through this entire eBook, it's time to present you with your final assignment.

As always, you can access this assignment (and all the others), plus bonus video content by going to http://dbvideotutorials.com/ebook-bonuses. Once you've signed up for free, you'll receive all the necessary documents via email.

I will tell you that this assignment is pretty tough. It's only one question, but it's one that I even struggled with for a bit.

But not to worry, I've included a "big hint" at the bottom of the assignment if you are at the point of tearing out your hair because you can't figure it out. I'd recommend waiting until JUST before you actually pull out any hair before peaking at the hint, as it is a fairly big one.

In any case, I hope you like the assignment!

Next Steps

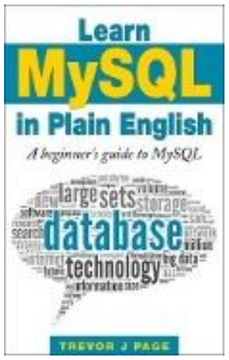
Step 1

Now that you've read this eBook and (hopefully) completed all the assignments, where do you go from here?

Well first thing's first, I want to thank you for picking up a copy of this book and actually reading through it! I commend you good sir or madam for sticking with it and making it to the end… truly commendable!

If you loved this eBook and found it to be of great help to you, I would love it if you could leave a positive review on Amazon. These reviews help as MASSIVE amount for getting this eBook in the hands of your fellow peers who are also dying to learn about Databases and MySQL.

Just go to Amazon.com and search for my book titled "Learn MySQL in Plain English" and click on the one with the cover that looks something like this:



Step 2

Are you really, really excited about what you've just learned?

Well I have a surprise for you!

I've created a Facebook group just for you, and people like you. You can join for free via this link: http://dbvideotutorials.com/facebook

Hop on in that group and share your biggest take-away from this eBook. What's the one thing that you learned in this eBook that you believe was the best "nugget" of knowledge? I'm sure other people in the group would love to hear from you, as it's an opportunity for them to learn something new as well!

Plus I'm in the group too, and I'll be participating in the discussions, so feel free to say "HI" to me too ☺

Step 3

Have fun with your new found skills and think of a project that you could do on your own to put these new skills to the test!

What’s a common problem in your life that you’re facing that you could potentially solve with a database?

Give it a try! There’s no better way to learn than to throw yourself into a fun project.

And hey, let us know what you’re building in the Facebook group, I’d love to hear about what you’re up to.