FULL STACK WEB DEVELOPER PART X

Relational Databases - MySQL

Learn SQL

Panos M.

www.techcareerbooster.com

Full Stack Web Developer Part X: Relational Databases MySQL

Panos Matsinopoulos

This book is for sale at
http://leanpub.com/full-stack-web-developer-part-x-relational-databases-mysql

This version was published on 2019-08-15

Contents

The Bundle and the TCB Subscription

Part of Bundle

This book is not sold alone. It is part of the bundle Full Stack Web Developer.

Goes with a TCB Subscription

When you purchase the bundle, then you have full access to the contents of the TCB Courses.

Each TCB Subscription Goes with the Bundle

Moreover, this goes vice-versa. If you purchase the subscription to the TCB Courses, then you are automatically eligible for the Full Stack Web Developer bundle.

Credits To Photo on Cover Page

We have designed the cover page, but the photo in the middle is the creation of our friend Telis Marin. He is an amateur photographer. He doesn't have an official Web site where you could find more of his amazing photos. Hence, if you want to see more of his work, the only way you can now do it is by making him an FB Friend here.

Telis Marin is an author, a publisher Edizioni Edilingua and a teacher trainer. He has written more than 20 books for learning Italian, which are used by schools and universities in over 80 countries.

About and Copyright Notice

Full Stack Web Developer - Part X - Relational Databases - MySQL 1st Edition, August 2019

by Panos M. for Tech Career Booster (https://www.techcareerbooster.com)

For more information: https://www.techcareerbooster.com

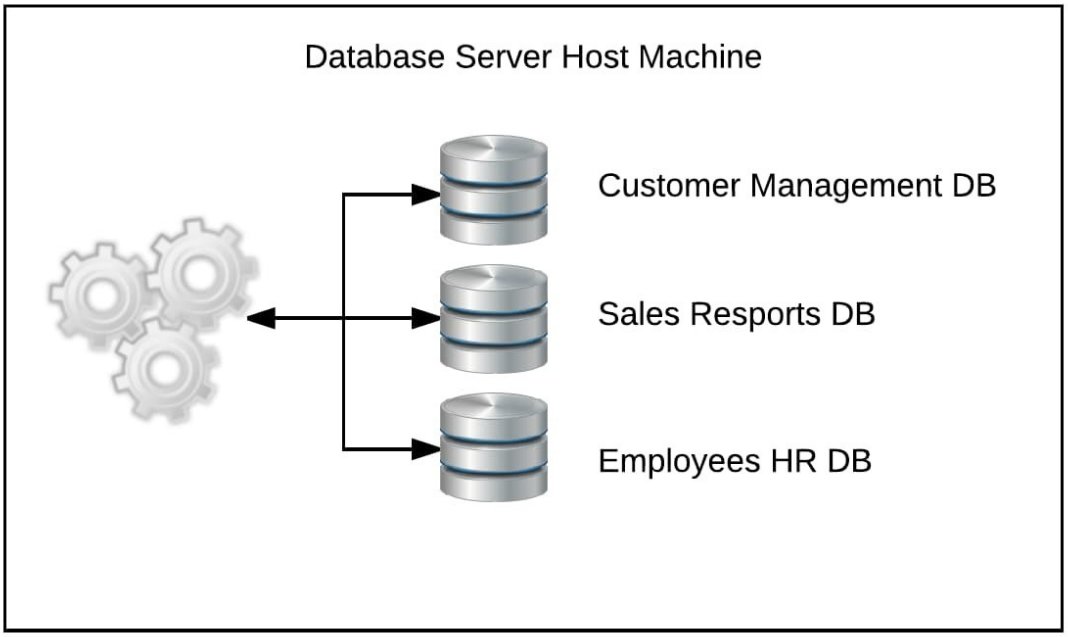
1 - Introduction to RDBMS

Summary

RDBMS is one of the most popular and important technology in software engineering. You must definitely become very good in SQL in order to be considered a good software engineer.

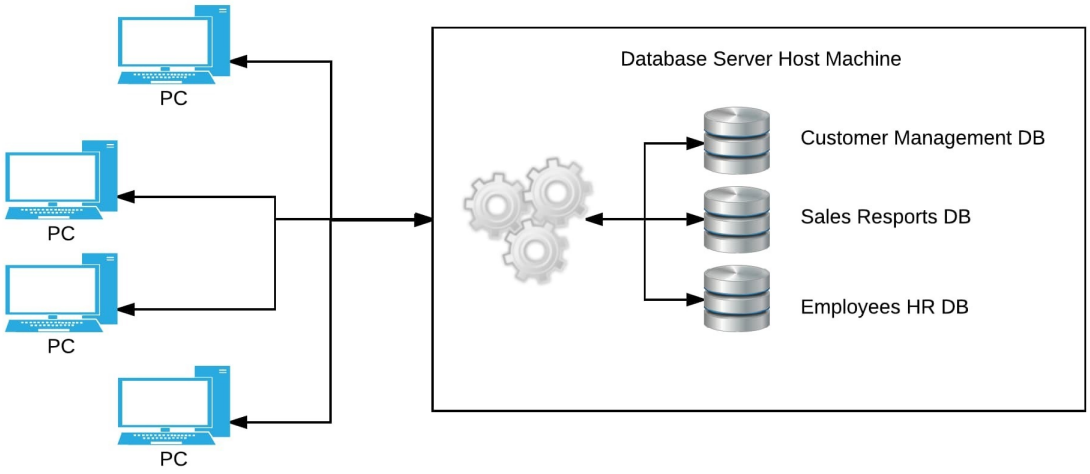
This chapter is an introduction to RDBM systems.

It will teach you how a server is managing many databases



Server Manages Many Databases

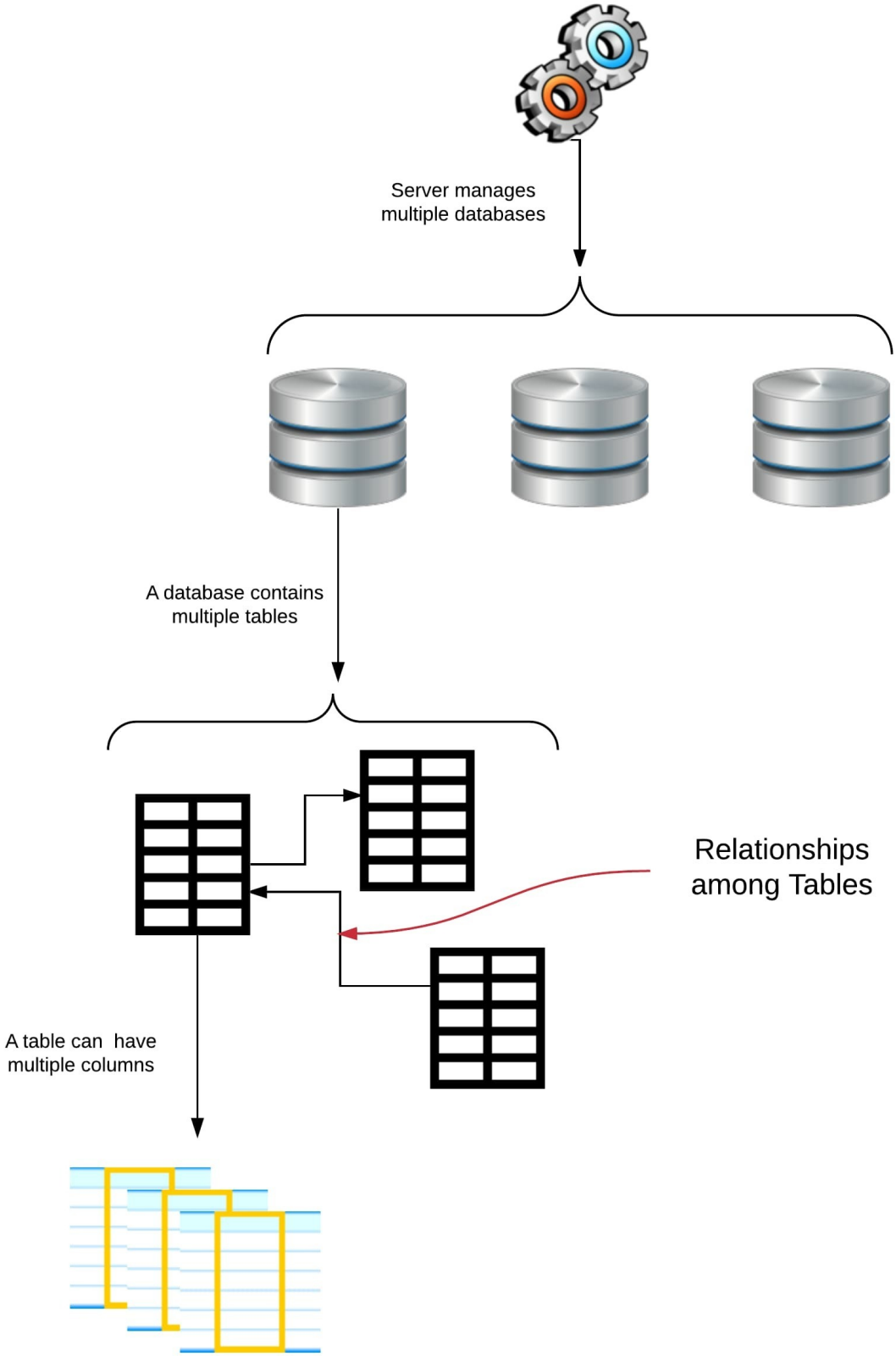
and serves many clients at the same time.



A Server serves Multiple Clients

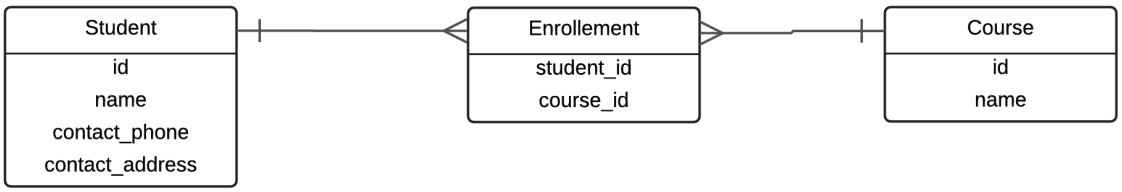
It will teach you about the client server model.

Also, it will teach you how the databases work and their fundamental components:



RDBMS Architecture

It will also teach you about the relationships between business entities.



Relationships Between Business Entities

Learning Goals

1. Learn what an RDBMS is.
2. Learn about the RDBM Server.
3. Learn about RDBMS clients.
4. Learn how clients access RDBM Server.
5. Learn about the client-server model.
6. Learn about SQL.
7. Learn about DDL.
8. Learn about DML.
9. Learn how we divide our database to tables and we create relationships between them.
10. Learn about the similarities and differences of an EXCEL to an RDBMS.
11. Learn about how we map business entities to tables.
12. Learn how we map business entity properties to table columns.
13. Learn about the conceptual model.
14. Learn about the relationships between between business entities.
15. Learn about how the actual database model will have some extra design details.
16. Learn which RDBMS we are going to use in next chapters to learn SQL.

Introduction

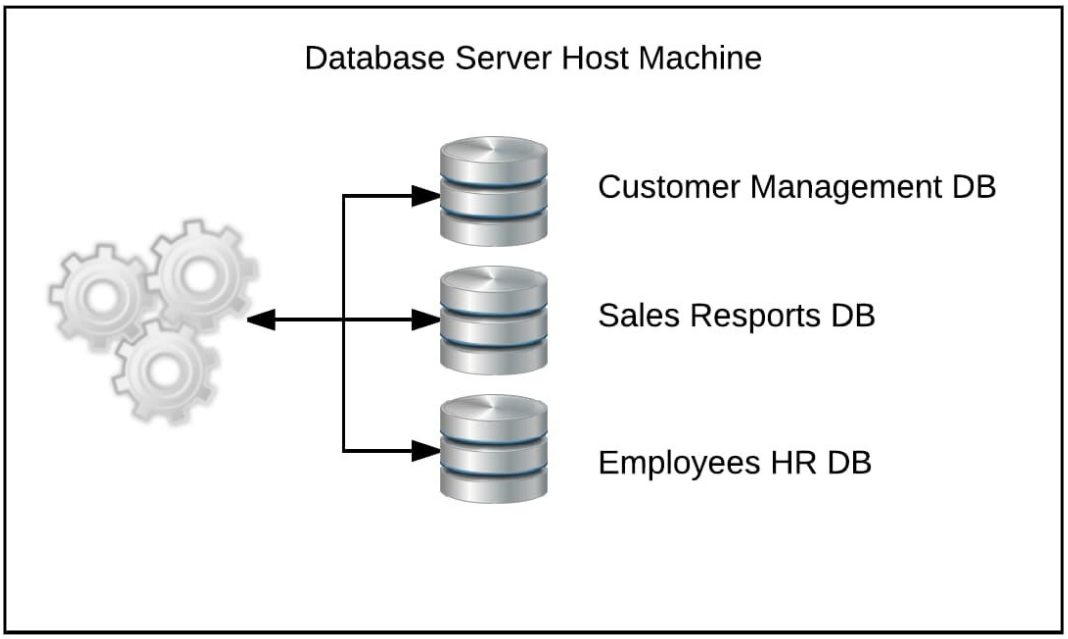
RDBMS stands for Relational Database Management Systems. These are the systems that are responsible to manage the database with our application data. Traditionally, all data management applications use an RDBMS. Although, latest years other types of database systems have flourished that fall into the NoSQL category.

As the name implies, an RDBMS is a system that manages data in a relational way. Sometimes the `S` stands for `Server` instead of `System`. In that case it refers to the executable process that runs in the background and takes care of our database management.

RDBMS Server

A server is a process. It runs on the background as a daemon. It makes sure that we can store our data safely on the hard disk.

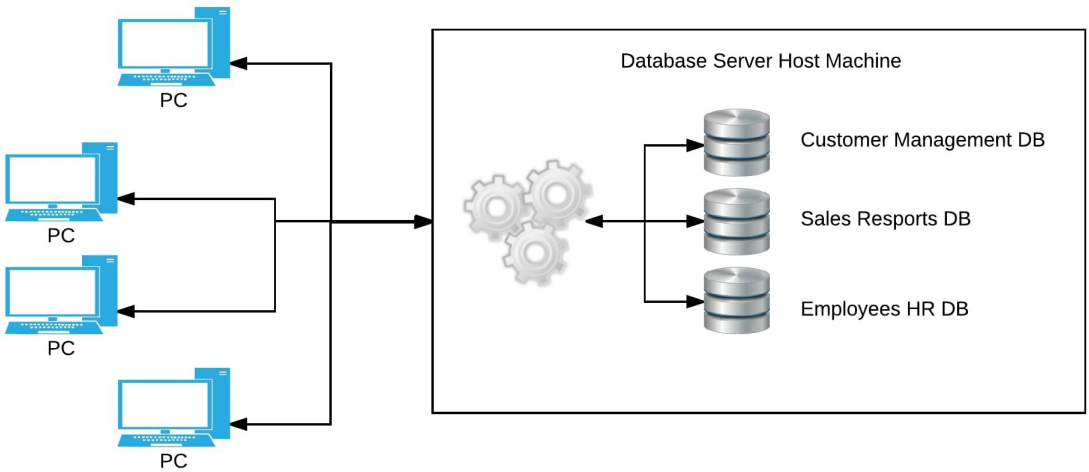
One server can manage multiple databases at the same time.



Server Manages Many Databases

So, we do not have to have one server per database. It is becoming very cost-effective and it is efficient to have one server managing many different databases at the same time.

Similarly, an RDBM Server can serve many different clients at the same time.

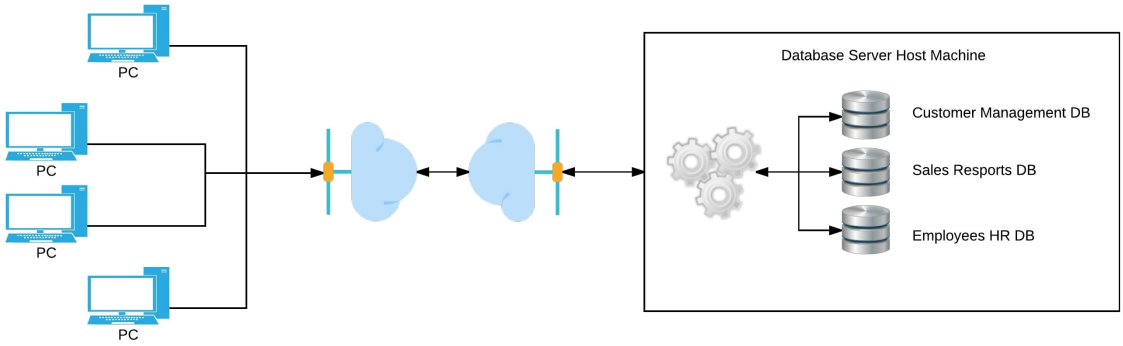


A Server serves Multiple Clients

The clients can be on the same client machine or on different client machines. They can even be on the same machine the RDBS server is running on.

In other words we may have multiple applications being served by the same server process.

Note that clients sending requests to an RDBM Server might be sending them over the Internet. They do not have to be on the same machine or same LAN as the server process.



Clients May Access Server over the Internet

Client - Server Model

An RDBMS is using the client - server model. In other words, RDBMS is a server process that responds to client requests. Whereas client is another process, another program, that sends requests to the server.

SQL

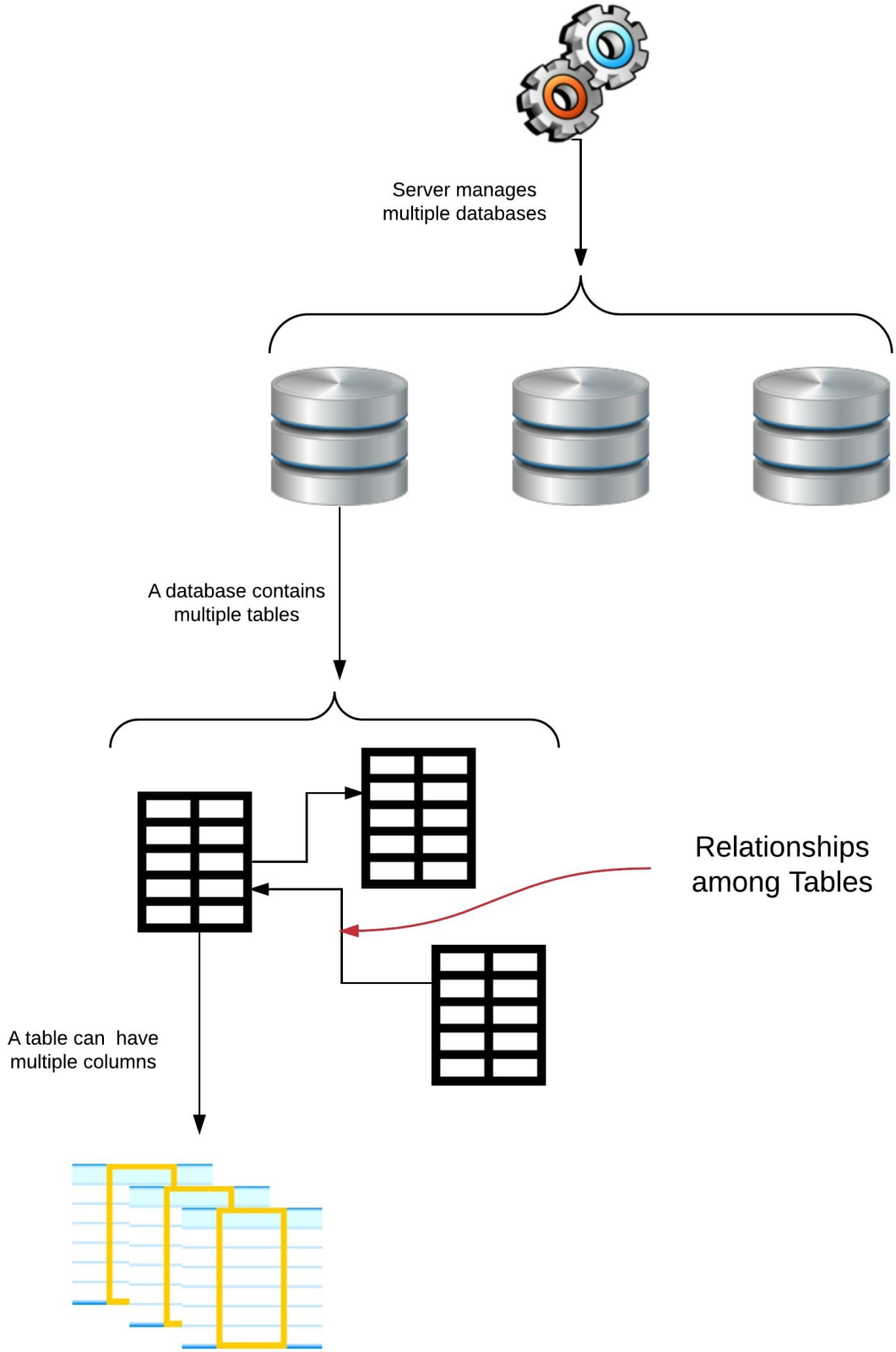
RDBMS client programs use SQL (Structured Query Language) to give instructions to server.

SQL is basically divided into 2 categories:

- DDL - Data Definition Language. This is the set of SQL statements that describe to the server the structure of the data. How the data have to be.
- DML - Data Manipulation Language. This is the set of SQL statements that tell to the server what we want to do with the data. Basically, create, retrieve, update, delete commands.

Data Modeling

Look at the following diagram. You need to understand that very well.

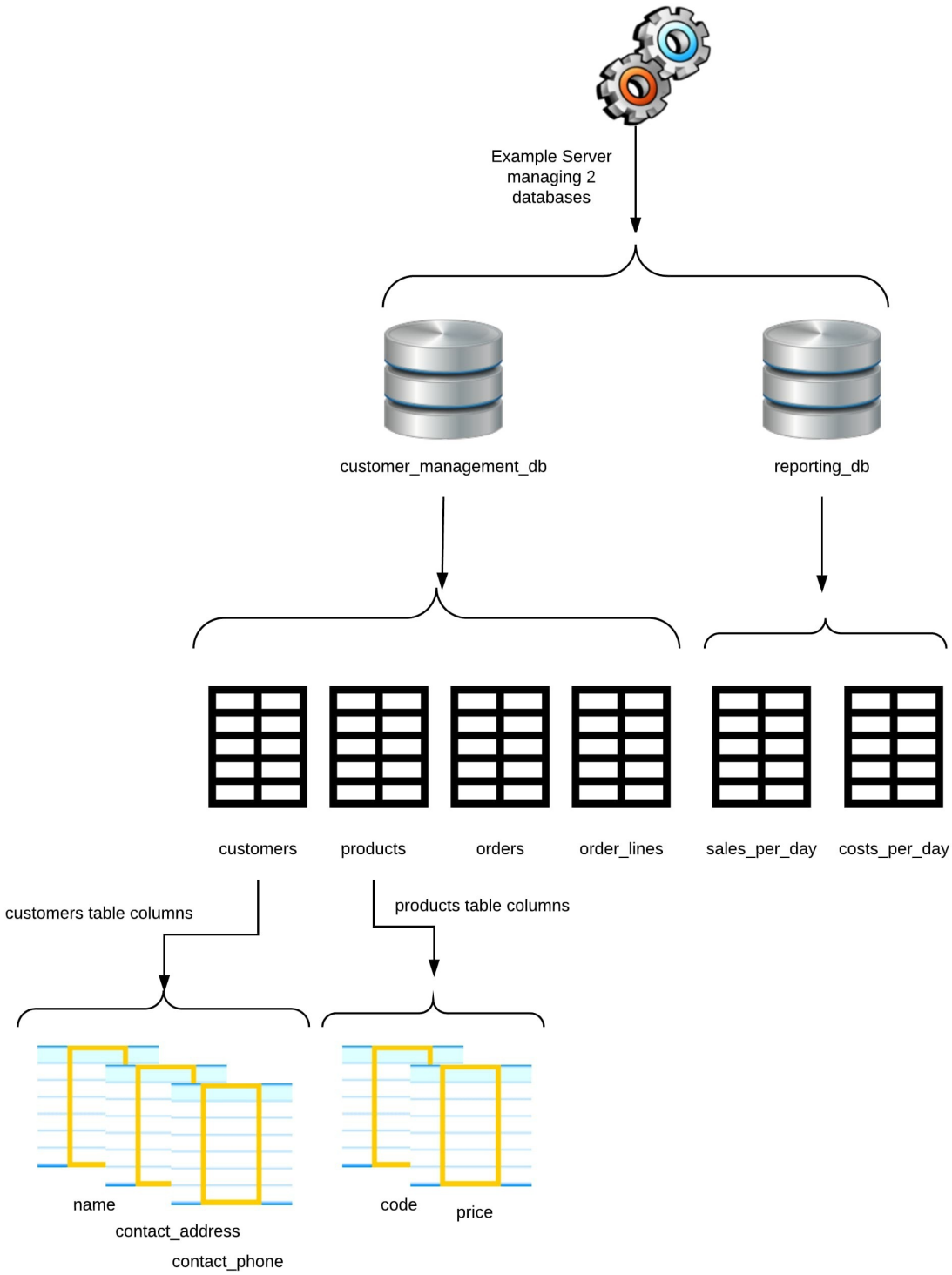


RDBMS Architecture

It basically says the following:

1. An RDBM server can manage multiple databases at the same time.
2. Each database contains one or more tables. A database with no tables is useless.
3. Each table contains one or more columns. A table without columns is useless too.
4. There are relationships between the tables of a database. That is why we call this system a Relational Database Management System.

Let's see an example of the above.



Example of Databases, Tables and Columns

On the above example we have 1 RDMS managing 2 databases:

- `customer_management_db` and
- `reporting_db`

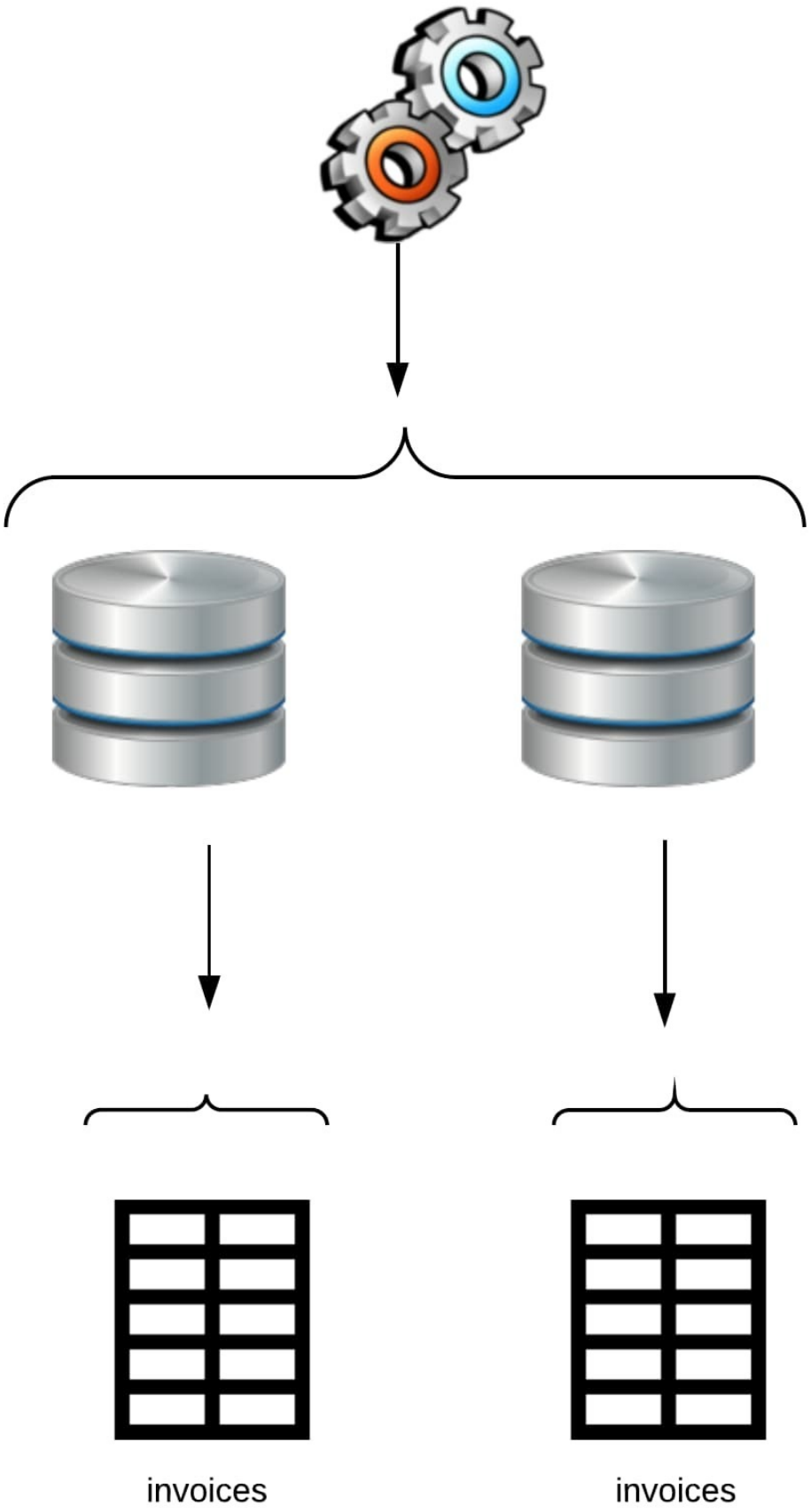
Each database has its own tables.

- `customer_management_db` has
  - `customers`
  - `products`
  - `orders`
  - `order_lines`
- `reporting_db` has
  - `sales_per_day`
  - `costs_per_day`

And each table has its own columns. For example:

- `customer` table has the columns:
  - `name`
  - `contact_address`
  - `contact_phone` and
- `products` table has the columns:
  - `code` and
  - `price`

Since each database has its own tables, that means that we can have the same name used by two different tables as long as those tables belong to different databases.

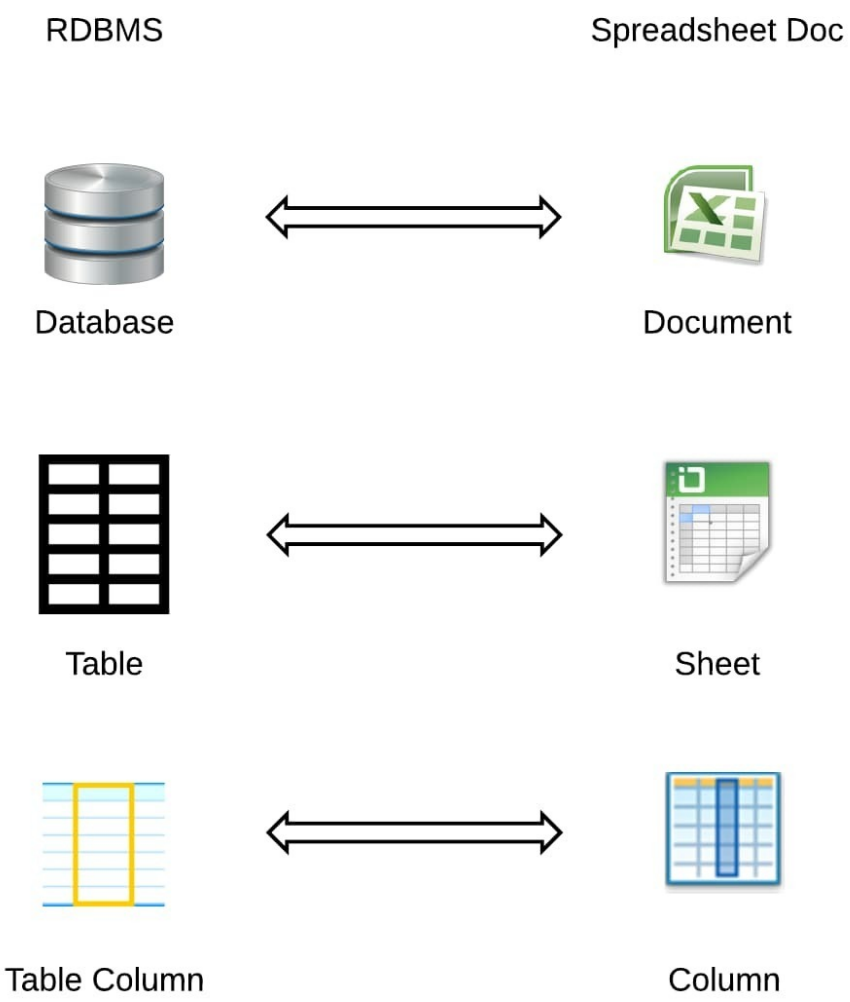


Two Tables Same Name Different Databases

Same goes for the columns. You cannot have two columns with same name belonging to same table. But, you can have two columns with same name belonging to different tables (even if those tables are in the same database).

Like sheets on EXCEL or Google spreadsheets

One can find similarities between an EXCEL or Google spreadsheets document and an RDBMS database.



Similarities Between RDBMS and Spreadsheets

1. An EXCEL file or a Google spreadsheets document can be considered similar to a RDBMS database.
2. A sheet can be considered similar to a database table.
3. A sheet column can be considered similar to a database table column.

However, there are some important differences between a spreadsheet and an RDBMS.

1. Unlike EXCEL, an RDBMS column can only hold same type of data. For example, you cannot store in the column `name` of table `customers` the string `"Peter"` and the integer `45`.
2. Unlike EXCEL, a table in an RDBMS database can hold huge amounts of data.
3. Unlike EXCEL, a table in an RDBMS database is related to another table. The RDBMS makes sure that business rules are enforced on these relationships and make sure that we do not store invalid data into our tables. For example, you cannot have an entry in the table `orders` that does not have corresponding entry in the table `customers`, or in other words, an entry in `orders` that does not belong to any `customer`.

Tables and Business Entities

When we design our database we usually use `1` table to store data for `1` business entity. For example, we use a table with name `customers` to store the occurrences, or the instances, of the business entity `Customer`. On another example, the table `products` is used to hold the instances of the business entity `Product`.

Since we usually have many occurrences, many instances of a business entity, each one with different characteristics and properties, we will create many rows inside the table that is used to store instances of that business entity.

Table: customers

3 Business Entities

name	contact_address	contact_phone
John Smith	5th JA str	8279921122
Mary Bright	15th Lincoln av.	3299188292
L. Brenard	27 Rue Pasteur	8294098292

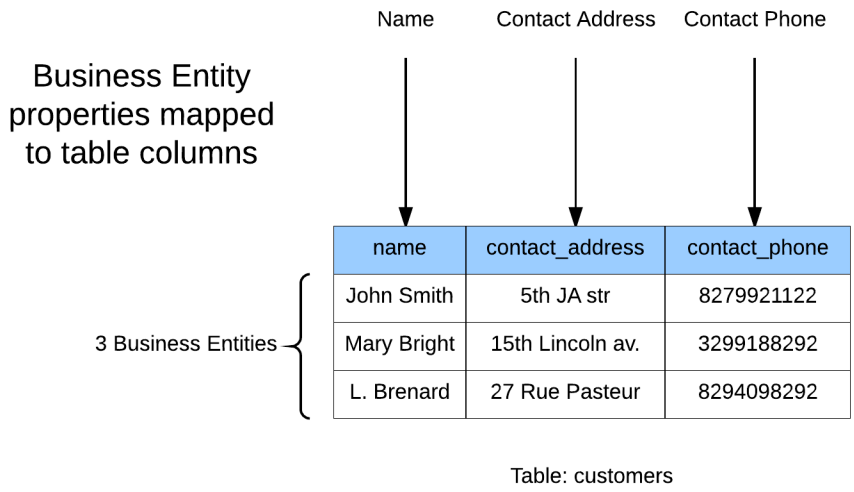
Many Business Entities Stored In Same Table

Above, you can see an example of a table `customers` storing `3` instances of the business entity `Customer`.

Also, in order to represent the properties, attributes, characteristics of a business entity, we usually do that with corresponding columns. Hence, having identified that a `Customer` has:

1. a Name
2. a Contact Address
3. a Contact Phone

we decide to create the corresponding columns on that table.



Business Entity Properties Mapped to Table Columns

Database Design

Conceptual Model

If you are going to implement a data management application you will probably need to design the database. You have to identify your business entities and you have to map them to tables. Having identified the business entities, then you have to identify the relationships between the business entities. Basically, there can be the following relationships between business entities.

- one-to-one (1 - 1). One business entity is related to another one business entity.
- one-to-many (1 - n). One business entity is related to many other business entities.
- many-to-many (m - n). Many business entities are related to many other business entities.

Information: The type of the relationship between two entities is also called cardinality

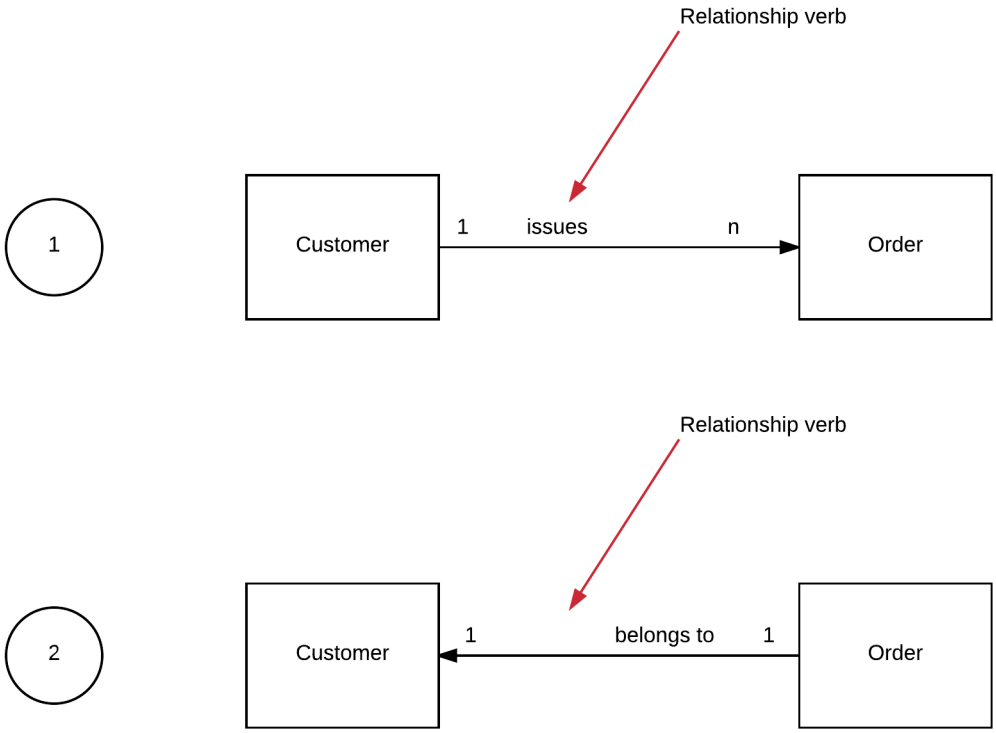
Let's see the relationships with some examples:

Customer and Orders - 1 - n

The above example shows the relationship of a `Customer` to an `Order`. This is a one - to - many relationship. One `Customer` can issue many `Orders`.

This is also a one - to - one relationship. This is because one `Order` is always associated with one `Customer`.

Hence, you understand that, the relationship between two entities is not always one. It can be more than one. It depends on where you start traversing the relationship from and where you end traversing the relationship to. Also, it depends on the actual *verb* of the relationship:

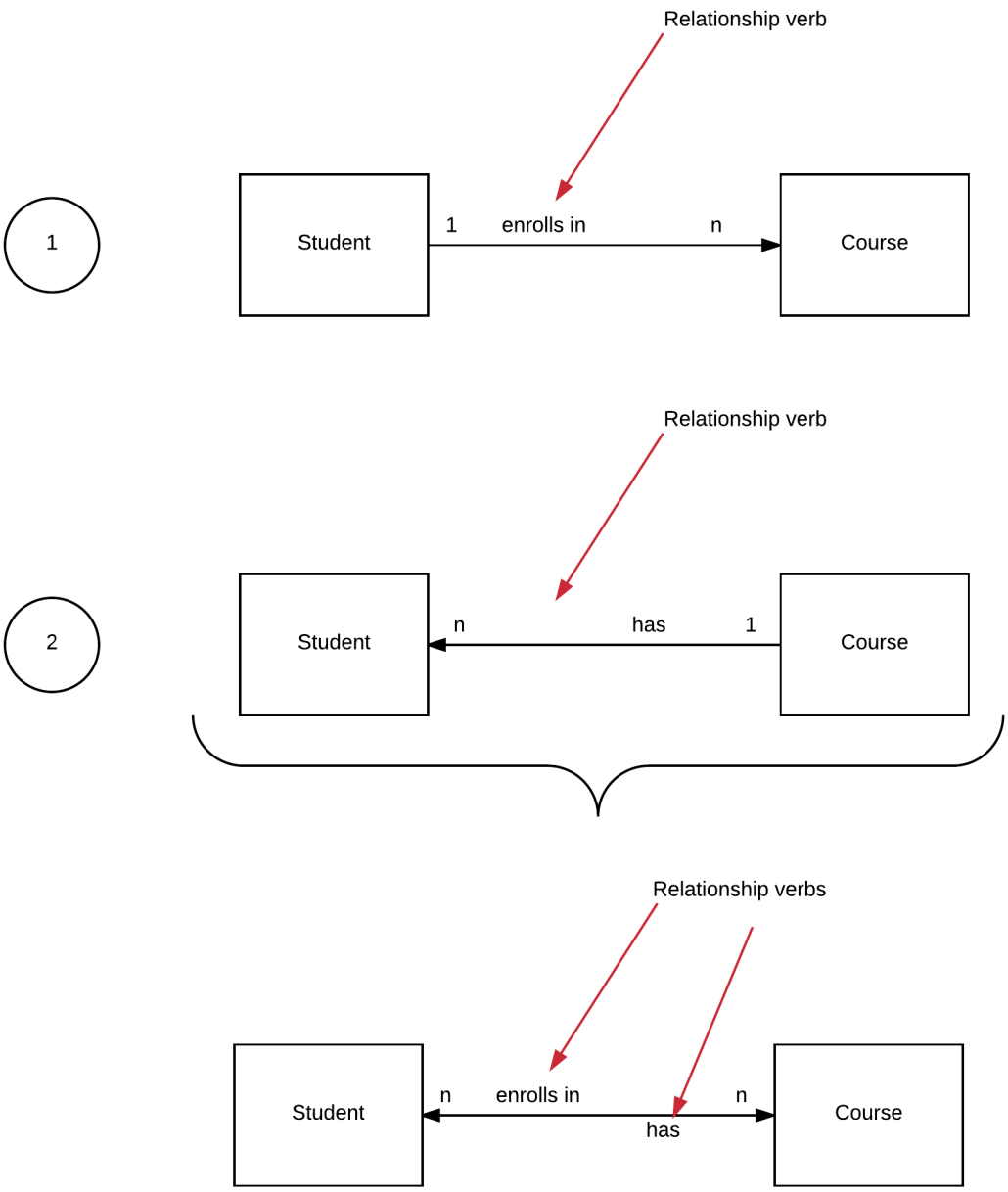


Relationship Direction and Semantics

As you can see on the above diagram, we can view a relationship between two entities in many different ways. Depends on the verb that we use and the direction that we follow.

1. Given a `Customer`, we can say that this `Customer` can issue many different `Orders` (one-to-many).
2. Given an `Order`, we can say that this `Order` belongs to a specific one `Customer` (one-to-one).

Let's see one more example with a many-to-many relationship:



Many-to-Many Relationship

On the previous picture you can see the relationships between business entity `Student` and business entity `Course`.

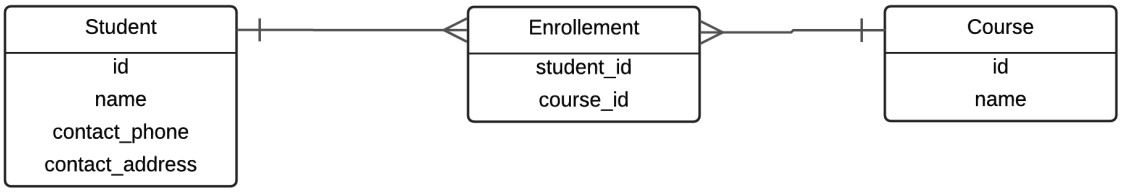
1. Given a specific `Student`, we can say that this `Student` can enroll in one or more, i.e. many, `Courses` (one-to-many), and
2. Given a specific `Course`, we can say that this `Course` can have one or more, i.e. many, `Students` (one-to-many).

When two business entities are related with a one-to-many relationship, on both directions, then we say that they have a many-to-many relationship.

Database Model

We have talked about the main types of relationships between business entities. The analysis and conceptual design is something that you start with, but you finally have to design the actual database model using a specific RDBM system.

The actual database model might be a little bit different to the conceptual model. For example, the many-to-many relationships are modeled with a table-in-the-middle:

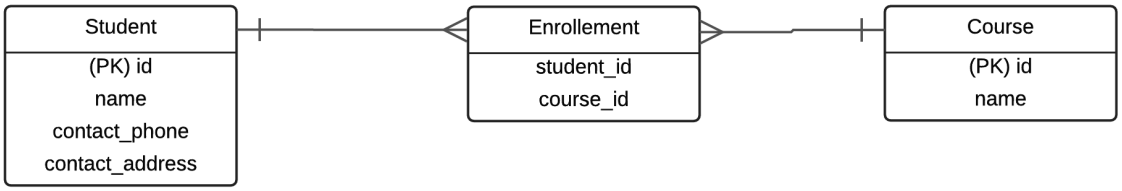


Many-to-Many With Table In the Middle

So, basically, the actual database design implementation uses two one-to-many relationships as depicted above. This is necessary in order to allow the storage of information a many-to-many relationship.

We will see specific examples in the next chapters.

Other important feature of the actual database design is the *primary key (PK)*. The primary key needs to uniquely identify each row in a table. For example, for the students table it can be a field named `id` that takes unique values. Same for the courses table.

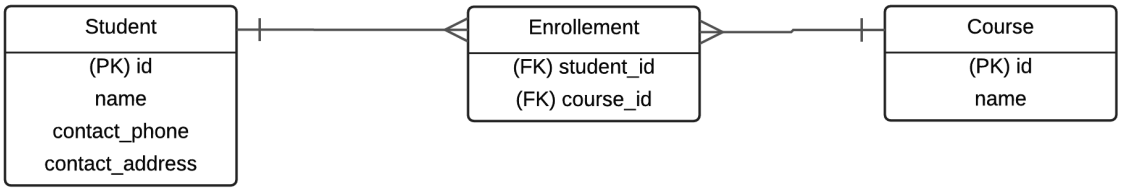


Primary Keys for Students and Courses

In other words, even if two students may have the same name and contact address and contact phone, they will never have the same id. Hence, they will be different entries in our students table.

The fact that the two tables have a primary key makes it possible to refer from table with enrollments to specific students and courses. We are referring to a student with a column named `student_id` and to the courses with the column `course_id`. Hence, any row in enrollments table specifies a student and a course without any doubt, and hence, we know which students have enrolled in which courses.

The fact that enrollments table uses columns to uniquely identify entries in other tables (students and courses), makes these column to be referred to as *foreign keys (FK)*.



With Foreign Keys

There are constructs and tools that we use to design a robust database model. Such as indexes and unique indexes. We will see all of that in the next chapters.

MySQL

In the next chapters we are going to do a lot of work with MySQL, one of the most popular relational database management systems. `mysql` is going to be the client program that we are going to use to send requests to MySQL server.

> *Note:* You may hear MySQL being pronounced `mycql`. Same goes for SQL. You can hear it as `cql`.

The example database that we will create, it will be a primitive electronic shop, with customers, orders and products.

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Quiz

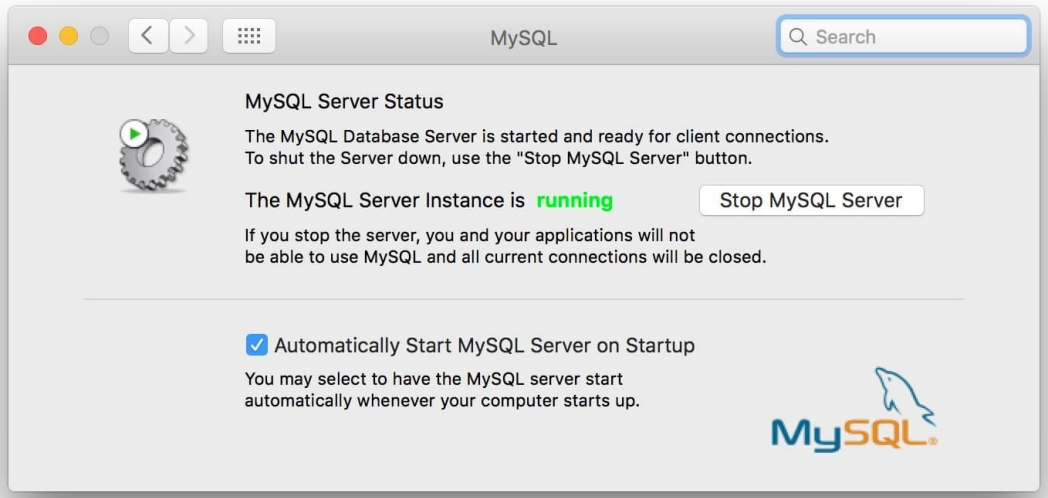
The quiz for this chapter can be found here

2 - Manage MySQL Server

Summary

This chapter will teach how to install MySQL server and how to check whether it is running or not.

You will also learn how to start, stop MySQL server, both using UI and command line interface.



MySQL Server reported as running

Finally, you will connect to MySQL server.

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 794
Server version: 5.7.14 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

(the above code snippet online)

Learning Goals

1. Learn how to install MySQL on your local machine.
2. Learn how to verify that MySQL is running.
3. Learn how to read the process list and to find out how MySQL has been started.
4. Learn how to locate the data directory, i.e. the directory where all the databases reside.
5. Learn how to locate the error log directory.
6. Learn how to locate the file that holds the MySQL process id.
7. Learn how to start and stop MySQL server.
8. Learn how to start a new session with MySQL server using the `mysql` command line interface.
9. Learn about the `mysql` command line interface prompt.
10. Learn how to exist the `mysql` command line interface.

Introduction

As we said in the previous chapter the RDBMS that we are going to use it's going to be MySQL.

MySQL Installation

Installing MySQL on your machine might differ depending on the operating system. This course is for Mac and Linux distributions.

This URL here explains how to install MySQL on OS X. The download page is here.

And this URL here explains how to install MySQL Debian Linux distribution. The download page is here.

Finally, this URL here explains how to install MySQL on Ubuntu Linux distribution. The download page is here.

> *Note*: While installing MySQL on Mac, the installation process assigns a random password to `root` user. I personally logged in using `mysql` as `root` user with the command `mysql -u root -p` giving that random password, and then issued the command `ALTER USER 'root'@'localhost' IDENTIFIED BY '';` to completely remove the `root` password. I find it that it is not necessary to have a password for `root` user on my local development machine.
>
> *Note:* On Mac, after installation start the MySQL preferences and start MySQL server. Also make sure that you have the check to start MySQL server on machine start up.

Verifying MySQL Running as Server

Let's start a terminal.

How to Start a Terminal

Using the terminal, we are going to check whether MySQL server is running or not. In order to do that, we type in the following command.

```
ps -ef | grep 'mysql'
```

(the above code snippet online)

When you run that, you will see an output similar to this:

```
    74    96     1   0  9:08AM ??         0:00.44 /usr/local/mysql/bin/mysqld --us\
er=_mysql --basedir=/usr/local/mysql --datadir=/usr/local/mysql/data --plugin-dir\
=/usr/local/mysql/lib/plugin --log-error=/usr/local/mysql/data/mysqld.local.err -\
-pid-file=/usr/local/mysql/data/mysqld.local.pid
  501  1115   784   0  9:15AM ttys000    0:00.00 grep mysql
```

(the above code snippet online)

which means that the `ps -ef | grep 'mysql'` found 2 processes running (at the time the command was executed). The last one `grep mysql` refers to the command that we just typed in. The other process running refers to MySQL and verifies that MySQL server is running properly on our machine.

> *Note:* The above output will differ on your machine. Depends on the operating system you are running in too.

In any case, what is important to note here is that the above output

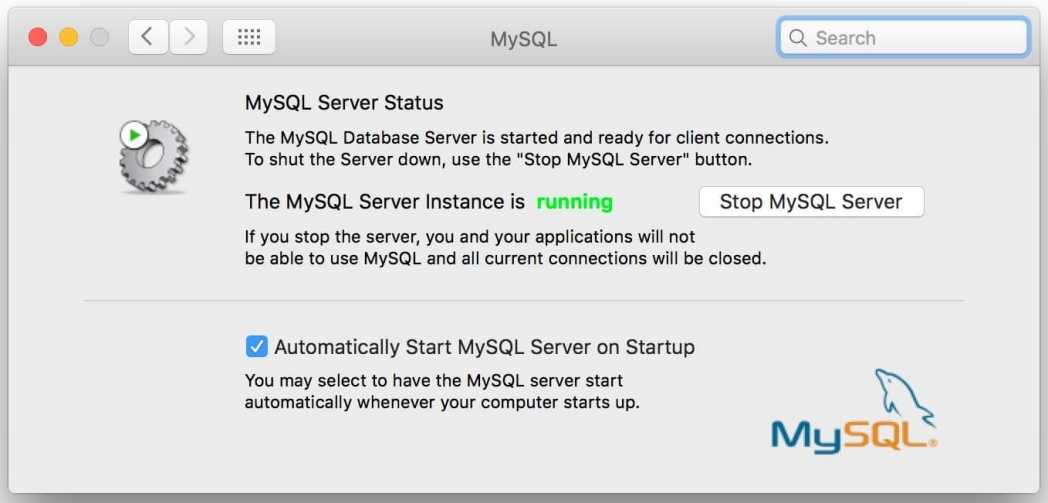
1. Tells us which user is used to run MySQL server. On my case it is `_mysql`. See `--user=_mysql`.
2. Tells where the base directory is. On my case it is `/usr/local/mysql`. See `--basedir=/usr/local/mysql`.
3. Tells which directory holds the data, i.e. the different databases this server is managing. On my case it is `/usr/local/mysql/data`. See `--datadir=/usr/local/mysql/data`.
4. Tells which file hold possible errors logged. On my case it is `/usr/local/mysql/data/mysqld.local.err`. See `--log-error=/usr/local/mysql/data/mysqld.local.err`.
5. Tells which file holds the MySQL server process id. On my case it is `/usr/local/mysql/data/mysqld.local.` See `--pid-file=/usr/local/mysql/data/mysqld.local.pid`. In fact, if one issues the command `sudo cat /usr/local/mysql/data/mysqld.local.pid` will get the number corresponding to the process id of MySQL server process. It is `96` for my case. Note that every time you stop and start the server again, a new server process is created with a new process id which is always recorded inside this file.

> *Note:* When we prefix the command with `sudo` we tell operating system that we want to run the command as system / administrator user. In that case we might be asked to give the system/administrator password. Alternatively, you can issue the command `su root`. The `su` means `switch user` and allows you to become another user. The `su root` command, then, allows you to become a `root` user. But, when you are root user you need to be very careful not to destroy any of your system files.

Starting and Stopping MySQL server

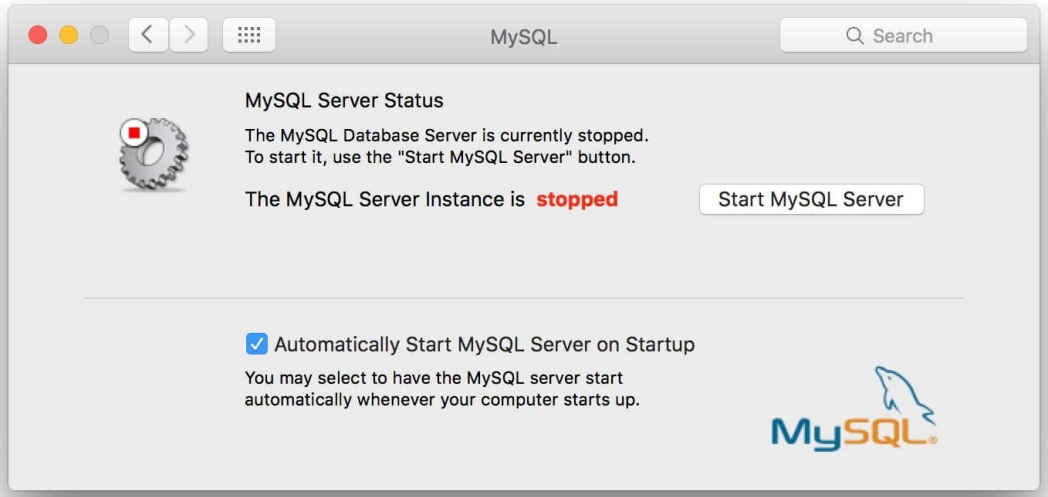
On Mac, there are two ways you can start and stop MySQL Server.

(1) From MySQL System Preferences. Open System Preferences and then locate MySQL. You will see this:



MySQL System Preferences - Stop MySQL

Then you can click on `Stop MySQL Server` button to stop MySQL server from running. You will be asked for your system password. After you do that you will see this on preferences dialog.



MySQL System Preferences - Server is Stopped

Now, if you go to a terminal window and type in the command `ps -ef | grep 'mysql'` you will not see the server running.

Obviously, if you click on the button `Start MySQL Server`, then you will have MySQL server back on running state.

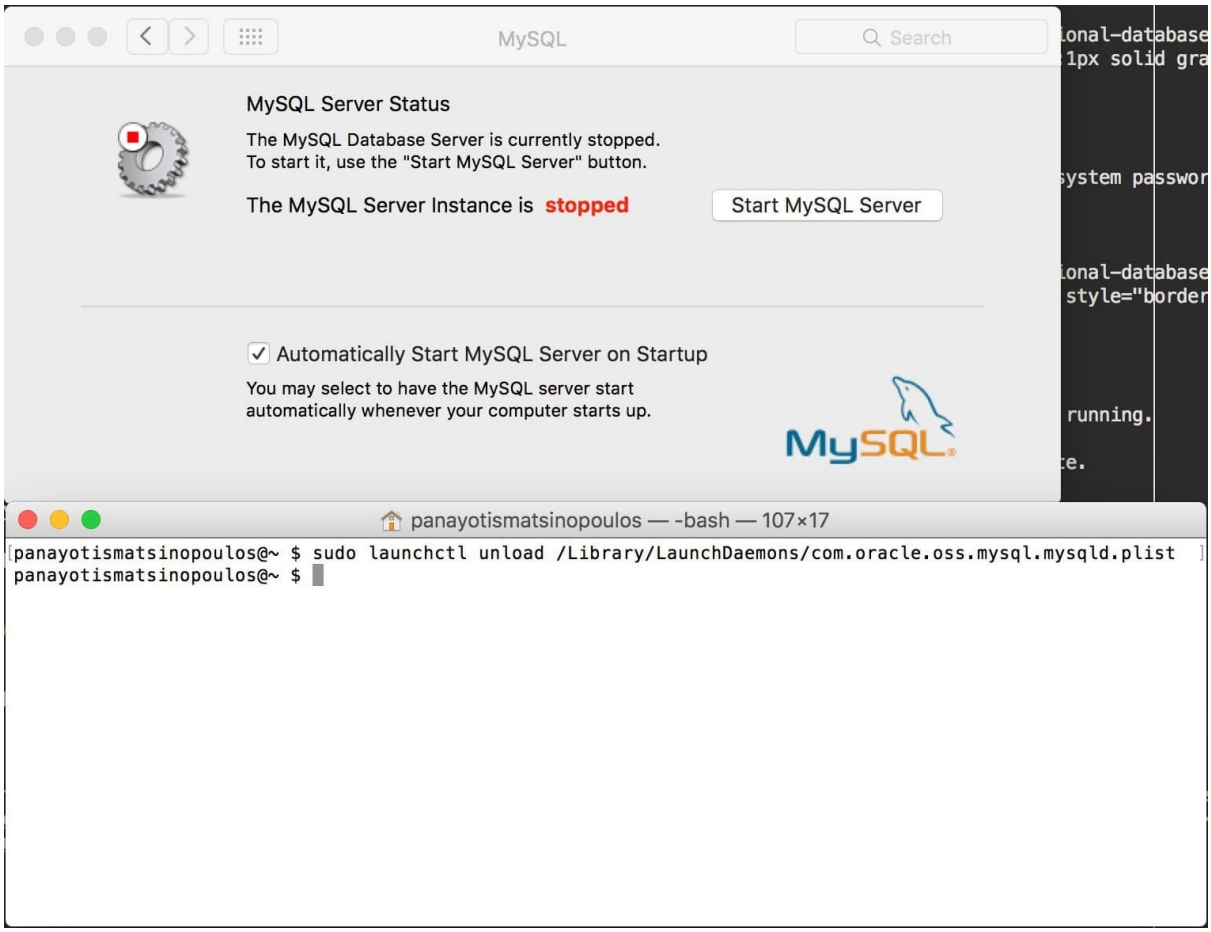
(2) Another way you can stop MySQL server is from the command line.

On Mac OSX El Capitan, this is done as follows:

```
1  sudo launchctl unload /Library/LaunchDaemons/com.oracle.oss.mysql.mysqld.plist
```

(the above code snippet online)

This will stop the server.

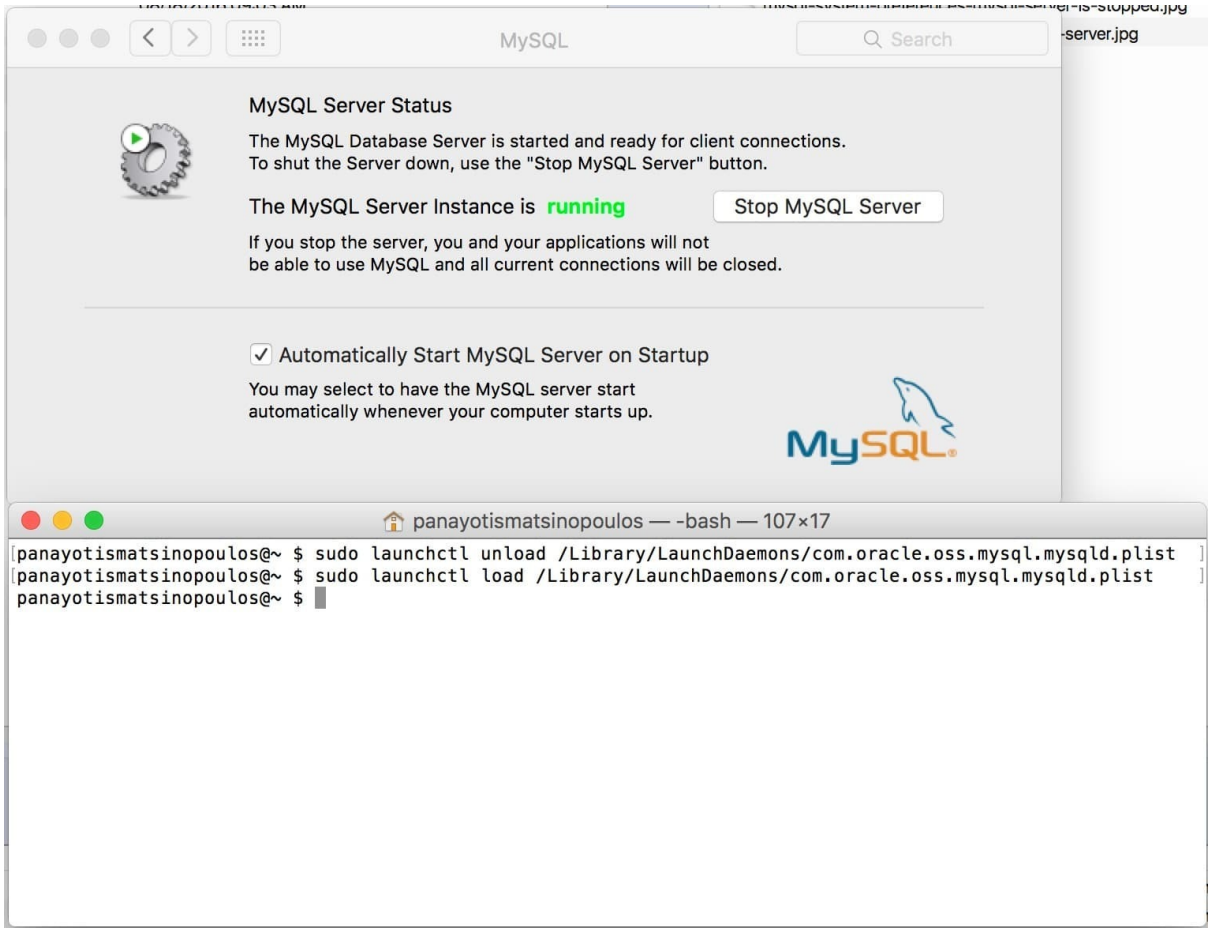


launchctl - Stop MySQL

Similarly, the `load` version can be used to start MySQL server:

```
1 sudo launchctl load /Library/LaunchDaemons/com.oracle.oss.mysql.mysqld.plist
```

(the above code snippet online)



launchctl - Start MySQL

> *Note:* Another command that you can use to stop MySQL server is the following: `bash mysqladmin -u root shutdown` (the above code snippet online)

> *Hint:* You need to have the `bin` folder of your MySQL installation added in your `PATH` environment variable. Personally, I have my `.bash_profile` file include the next three lines: `bash alias mysql=/usr/local/mysql/bin/mysql alias mysqladmin=/usr/local/mysql/bin/mysqla export PATH=$PATH:/usr/local/mysql/bin` (the above code snippet online)

The last line sets the `/usr/local/mysql/bin` into the `PATH`.

> *Hint:* If you update the `.bash_profile` file, then you need to either call `source ~/.bash_profile` or restart your terminal.

Communicating With MySQL Server

Now that we know that MySQL Server is running and we know how to stop and start it again, let's see that command line tool that we will be using to connect to MySQL Server.

This is `mysql`.

On your terminal, type in the following command:

```
mysql -u root -p
```

(the above code snippet online)

You will be asked to give the password of `root` user on your MySQL installation. On my installation there is no password. Hence on the prompt `Enter password:` I just hit the key <kbd>Enter</kbd>. If your `root` server has a password, then you need to give it here.

> *Hint:* If the `root` user does not have a password, you can start `mysql` with the command `bash mysql -u root` (the above code snippet online)

and you will not be prompted for password.

What you will see is something like the following:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 794
Server version: 5.7.14 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

(the above code snippet online)

The `mysql>` is the `mysql` command line interface prompt. In other words, you have left the operating system terminal prompt and you have started a new session with the MySQL server installed locally on your machine, which gives you a `mysql` specific prompt to type in SQL commands.

`mysql` prompt allows you to enter SQL commands. This means that if you try to issue, for example, an operating system command, it will fail.

```
mysql> ls -l;
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that c\
orresponds to your MySQL server version for the right syntax to use near 'ls -l' \
at line 1
```

(the above code snippet online)

If you want to exit `mysql` session and return back to your terminal prompt, you type in the `exit` command.

MySQL Settings

When MySQL server starts has some default settings. However, we can override the default settings by specifying values inside a file named `my.cnf`. On Mac, You can place that file in the following places:

```
/etc/my.cnf
/etc/mysql/my.cnf
/usr/local/mysql/etc/my.cnf
/usr/local/opt/etc
~/.my.cnf
```

(the above code snippet online)

Also, you can find a sample file here: `/usr/local/mysql/support-files/my-default.cnf`

Alternatively, you can edit the file `/Library/LaunchDaemons/com.oracle.oss.mysql.mysqld.plist` and set the start up options to your preferences.

One of the most important options is the `sql_mode`. You should turn strict mode on. This is done by setting `sql_mode` to include `STRICT_TRANS_TABLES`.

On my current installation, `STRICT_TRANS_TABLE` is on. I can verify that by executing the following query:

```
mysql> show variables where variable_name = 'sql_mode';
+---------------+-----------------------------------------------------------------\
-------------------------------------------------------------------------------+
| Variable_name | Value                                                           \
                                                                               |
+---------------+-----------------------------------------------------------------\
-------------------------------------------------------------------------------+
| sql_mode      | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_\
DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+---------------+-----------------------------------------------------------------\
-------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

(the above code snippet online)

As you can see, `sql_mode` includes the value `STRICT_TRANS_TABLES`, which means that strict mode is on.

If this is not the case with your MySQL installation, make sure that you edit the `my.cnf` file and add something like this (in the `[msqld]` section):

```
sql_mode = 'ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,E\
RROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION'
```

(the above code snippet online)

Ask your mentor for advice if you cannot make it.

Closing Notes

We have learned how to install, start and stop MySQL server. We have also learned how to start a new session with MySQL server using `mysql` command line interface. In the next chapters we will be working very hard with `mysql`.

> *Information:* If you had problems installing, starting and stopping MySQL server on your machine, you may want to arrange a pairing session with your mentor. He will help you setup your machine and MySQL.

> *Extra Video:* We have created a video that goes through the concepts of this chapter using a Debian (Linux) operating system. You may want to watch that too.

MySQL Server on a Debian (Linux) machine

Tasks and Quizzes

> **Before you continue**, **you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

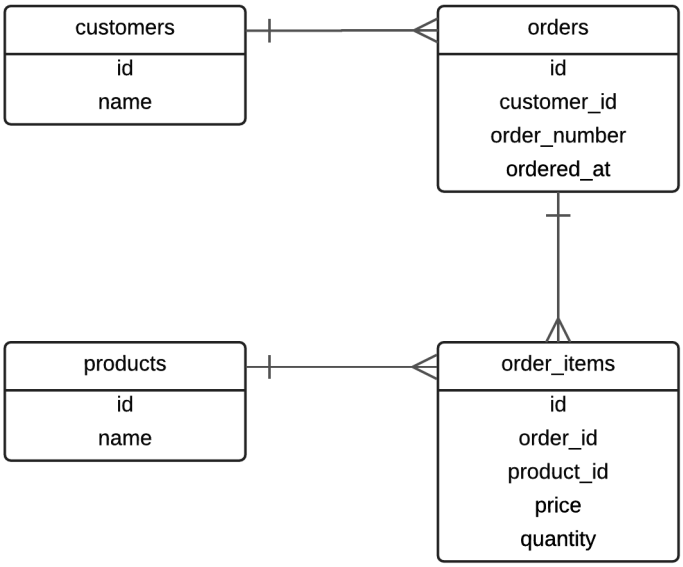
Quiz

The quiz for this chapter can be found here

3 - Basic SQL Commands

Summary

This chapter is good introduction to basic SQL commands. You are going to start creating a database such as this:



Primitive CRM - The ERD

After a recap on the entities between tables, you will learn all the necessary commands to create the database `customers_db` and its first table `customers`. Then you will manage (create, retrieve, update, delete) the data stored inside this table.

So, this chapter is an introduction to both DDL (Data Definition Language) and DML (Data Manipulation Language).

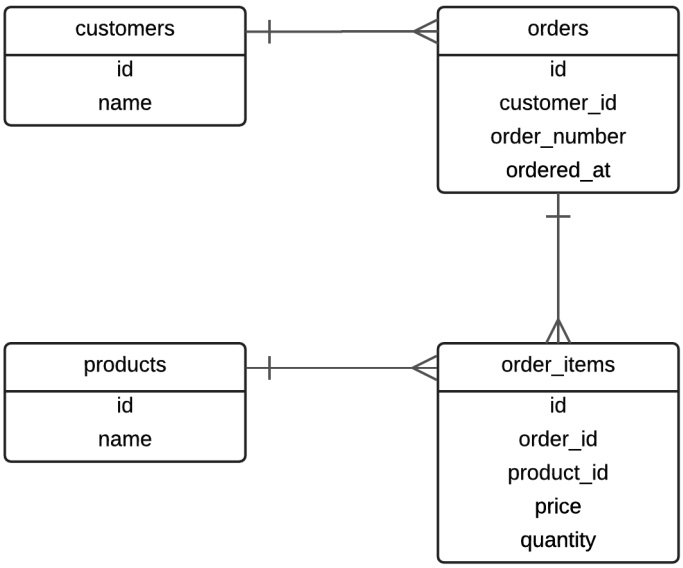
Learning Goals

1. Learn, again, about the basic concepts behind Entities and relationships between them.
2. Learn how to list the current databases your MySQL server is managing.
3. Learn how to create a brand new database.
4. Learn how to select the database to work with.
5. Learn how to enlist the tables of a database.
6. Learn about the various MySQL data types.
7. Learn how you can define non-null column types.
8. Learn how you can define columns with integer values that are auto incremented.
9. Learn how you can define the primary key of a table.
10. Learn how you can see the definition of the structure of an existing table.
11. Learn how to insert data into a table.

12. Learn how to select rows from a table.
13. Learn how to provide selection criteria.
14. Learn how to count the number of rows in a table.
15. Learn how to delete rows from tables.
16. Learn how you can update the rows of a table.
17. Learn about the `AND`, `OR` and `like` operators.

Introduction

We now start to design the database for a primitive Customer Relationship Management System. Here is the picture of the tables this database is going to have, alongside the relationships among them.



Primitive CRM - The ERD

Let's recap what we see on the above Entity Relationship Diagram (ERD). Also, let's give some rules about the columns and their values even if these are not obvious from the diagram:

1. We see 4 tables:
    1. customers
    2. orders
    3. products
    4. order_items
2. customers
    1. It has an `id`. This is also going to be the primary key of the table. I.e. all customers will have distinct, unique ids.
    2. It has a `name`. This is not unique. Two customers may have the same name. It is mandatory, i.e. we cannot have a customer without a `name`.
3. orders

1. It has an `id`. This is going to be the primary key of the table. I.e. all orders will have distinct, unique ids.
2. It has a column `customer_id`. This is going to be a reference to a corresponding row in `customers` table. It will indicate which customer this order belongs to. We call this a foreign key, because it corresponds to the primary key (`id`) of another table (`customers`). `customer_id` is mandatory, i.e. we cannot have an order without a reference to a customer.
3. It has a column `order_number`. This is a string type column and it is mandatory. This should be unique, i.e. no two or more orders can have the same `order_number`.
4. It has a column `ordered_at`. This is a timestamp and it is mandatory. It records when an order has been issued by the customer. If not given at order creation, it will take automatically the current date timestamp.
5. There is a relationship between `customers` and `orders`. Having *one* customer, I can find *many* orders referencing that customer. So, the relationship is *one-to-many*. Also, having *one* order, I can find *one* customer the order belongs to.

4. products
   1. It has an `id`. This is going to be the primary key of the table. I.e. all products will have distinct, unique ids.
   2. It has a `name`. This is a string type column and it is mandatory. I.e. a product cannot exist without a `name`. Also, this is going to be unique. No two or more products can have the same `name`.
5. order_items
   1. It has an `id`. This is the primary key of the table. I.e. all order_items will have distinct, unique ids.
   2. It has a reference to the order the order item belongs to. This is the column `order_id`. This is a foreign key because it references the primary key (`id`) of another table (`orders`). The `order_id` is mandatory, i.e. we cannot have an order item without an order.
   3. There is a relationship between `orders` and `order_items`. Having *one* order at hand, I can find *many* order items referencing that order. So, the relationship of `orders` to `order_items` is *one-to-many*.
   4. It has a reference to the product the order item refers to. The reference is the column `product_id` and it is a foreign key because it references the primary key (`id`) of another table (`products`). This is mandatory, because we cannot have an order item without reference to a product.
   5. There is a relationship between `products` and `order_items`. Having *one* product at hand, I can find *many* order items referencing that product. So, the relationship of `products` to `order_items` is *one-to-many*.
   6. There is a business rule which says that we cannot have the same product twice in the same order. We are using the `quantity` field to allow for many instances of the same product to exist in the same order. The `quantity` is an integer type column. It is mandatory, with default value 1.
   7. Table `order_items` is there to help model the properties of the *many-to-many* relationship between `orders` and `products`. As we have said in the previous chapter, a *many-to-many* relationship is modeled using a table in the middle. In our case, this

table is `order_items`, and we end up having two *one-to-many* relationships (one order many order items, one product many order items).

As you can understand from the above list, there a lot of details surrounding the data model of this simple CRM. And we have not enlisted them all. We will now proceed in the actual implementation of this database using MySQL Server.

Create the database

We will start by creating the CRM database.

Open a terminal window, and start `mysql` client, the command line interface to access MySQL server and issue commands.

```
mysql -u root
```

(the above code snippet online)

> *Note:* If your `root` user has a password, you need to start `mysql` with the command
> `bash mysql -u root -p`
> and give the password when prompt.

After connecting, you will see something like this:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.14 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

(the above code snippet online)

And you will be ready to issue MySQL commands.

As we have learnt in the previous chapter, MySQL can manage many databases. For that reason, there is always a way for us to see how many and which databases our MySQL server is currently managing.

Type the following command to find out the currently managed databases:

```
mysql> show databases;
```

(the above code snippet online)

> *Note:* Commands on `mysql` prompt need to be terminated with ; before hitting the <kbd>Enter</kbd> to send it out.

You will get an output like this:

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set (0.01 sec)
```

(the above code snippet online)

The MySQL installation on my Mac has the above 4 databases installed by default. These are databases used by MySQL server itself. Usually, client applications don't access these databases. We can definitely use them, but we need to be very careful with these databases and the SQL commands that we issue against them, because if we do a mistake we might prevent MySQL server from working properly.

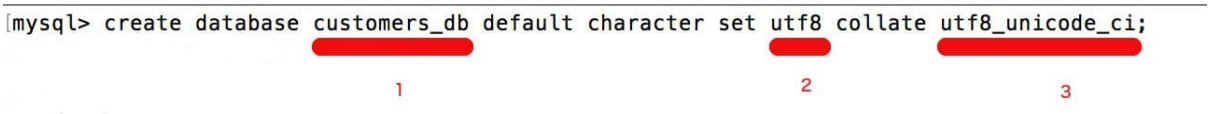
> *Note:* On your MySQL installation, the `performance_schema` and `sys` databases might be missing.

So, we are not going to work with these databases. We are going to design our own database. Let's call this database `customers_db`. The command to create our own database with name `customers_db` is given below. Give that command on `mysql` prompt.

```
mysql> create database customers_db default character set utf8 collate utf8_unico\
de_ci;
```

(the above code snippet online)

Let's analyze this command:



Create Database Command

1. This is the name of the database. On our example, it is `customers_db`.
2. This is the default character set. On our example, it is `utf8`. This means that we will be able to store both latin and non-latin characters, such as Greek, of French accented characters. This is very important because you might have customers whose language does not come from latin and they might want to type their names in their own language.
3. This is the collation. On our example, it is `utf8_unicode_ci` where `ci` stands for case insensitive. This is very important too, because, usually, you want to make sure that two strings that they only differ in their case, they are considered equal. Example: "John Smith" and "JOHN SMITH" should be considered the same.

So, type in the above command and hit <kbd>Enter</kbd> key on your keyboard. Don't forget to terminate the command with `;`. This will send the command to MySQL server. MySQL server is going to execute this command and return back to us a response with the result of the execution. This is going to be something like this:

```
mysql> create database customers_db default character set utf8 collate utf8_unico\
de_ci;
Query OK, 1 row affected (0.01 sec)

mysql>
```

(the above code snippet online)

The `Query OK, 1 row affected (0.01 sec)` means that everything went well and the database has been created. How can we verify that? With the command that we learnt earlier, i.e.

```
mysql> show databases;
```

(the above code snippet online)

You will get something like this:

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| customers_db       |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

mysql>
```

(the above code snippet online)

Do you see the database `customers_db`? This is the new database that we have just created.

We will continue now working on this particular database.

Select Database To Work With

In order to select the database you want to work with, you need to issue the command `use` as follows:

```
mysql> use customers_db;
```

(the above code snippet online)

When you do that, you will see the response being `Database changed`. This makes you ready to start working with the `customers_db` database. Or, in other words, the next commands that you will type, data definition or data manipulation commands, will be executed in the context of the `customers_db` database.

Creating Tables

As we have learnt in the introduction to RDBMSs, a database can have many tables. How can we get the list of tables of the current database? This is done with the issue of the command `show tables;`. If you issue this command you will see this:

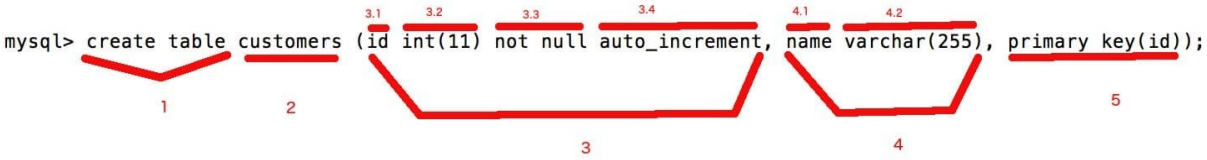
```
mysql> show tables;
Empty set (0.00 sec)

mysql>
```

(the above code snippet online)

Server tells you that `customers_db` does not have any table: `Empty set`. This is because `customers_db` is a database that we have just created and we have not created any table yet.

> *Side Note:* When you hit <kbd>Ctrl+L</kbd> while on `mysql` prompt, it will clear the terminal and `mysql` prompt will appear at the top. This is quite handy to clear the terminal window and proceed with new commands.

So, it is about time to design, to create our first database table. Below, we are giving you the command that would create the table `customers` inside the `customers_db` database. But, before you issue the command, it's better to study the notes that follow:



Command to Create Table customers

(1) `create table` is how we start the command to create a table.

(2) Then we give the name of the table.

(3) Then inside parentheses, we give the definition of the first column of the table.

(4) The second column definition. It is separated from the first using a comma.

(5) We optionally specify other details about the table columns, such as which column is the primary key.

On the above example, we will create the table with name `customers`. This is going to have `2` columns. The column `id` and the column `name`.

The column `id` is specified as follows:

```
id int(11) not null auto_increment
```

So,

(3.1) is the name of the column.

(3.2) is the type of the column. Here we use the type `int(11)`. This means that the column is going to hold integer numbers, with display width `11`. The display width is not affecting the range of the integer numbers that this column can hold. The range of the numbers is minimum `-2,147,483,648` and maximum `2,147,483,647`. You can read about all the MySQL data types here.

(3.3) optionally, we can tell whether the column can take `null` values of not. `null` in SQL represents the `absence of value`. Can we have a customer without an `id` value? No. In order to ask MySQL server to protect us from creating any customer without `id` value, we specify that the column needs to be `not null`. If we wouldn't use this specification, the column would have been `nullable`.

(3.4) with `auto_increment` we ask MySQL to use a unique number for `id`s every time we create a new customer. This number is created automatically by MySQL and assigned to each new row. It goes from `1` and it is incremented every time we insert a new customer. We don't have to worry about us assigning this number.

The column `name` is specified as follows:

```
name varchar(255)
```

So,

(4.1) is the name of the column.

(4.2) is the type of the column. With `varchar` we specify that this column is going to hold any variable-length character sequence, i.e. a string. However, the maximum length of the strings that can be stored there is `255`.

Note that for `name` column we have not specified the property `not null`, which means that we will be able to create customer entries without `name` value. This may sound like something we don't want. But this is how it is now. This create command given above, will create a table `customers` with column `name` being nullable. We will change that later on.

Finally, after the `name` column specification, we use a `,` in order to start an extra specification that tells MySQL which is going to be the primary key of the table.

```
primary key(id)
```

This is pretty much straightforward. It tells MySQL that the primary key of the table `customers` is going to be column `id`.

Nice! After having studied all that notes above, let's issue the command on the `mysql` prompt:

```
mysql> create table customers (id int(11) not null auto_increment, name varchar(2\
55), primary key(id));
Query OK, 0 rows affected (0.02 sec)

mysql>
```

(the above code snippet online)

You will get the `Query OK` response and your table would have been created.

How can you check that the table is there? It is the `show tables;` command, as we've learnt earlier:

```
mysql> show tables;
+------------------------+
| Tables_in_customers_db |
+------------------------+
| customers              |
+------------------------+
1 row in set (0.00 sec)

mysql>
```

(the above code snippet online)

As you can see from the result the table has been created inside the database.

Can we check what is the structure of a table? Yes, we can do that with the `show create table` command. Issue the following:

```
mysql> show create table customers;
+-----------+-------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------+
| Table     | Create Table                                                      \
                                                                               \

                                                                  |
+-----------+-------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------+
| customers | CREATE TABLE `customers` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
```

```
  `name` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-----------------------------------------------------------+
1 row in set (0.00 sec)

mysql>
```

(the above code snippet online)

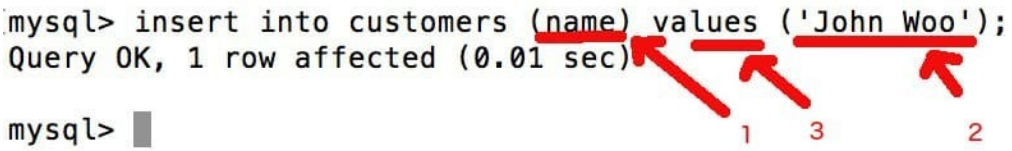
Perfect! MySQL server responds back and `mysql` server displays the structure of the table `customers`. It actually displays that using the `CREATE TABLE` command. There are some extra bits of information, if compared to the original `create table` command that we used, but you do not have to worry about that for now.

Inserting Data Into a Table

Now that we have our table, we can insert data into it. The command to insert data into a table is the `insert into <table_name>` command. Let's do that with the `customers` table. Issue the following command:

```
mysql> insert into customers (name) values ('John Woo');
Query OK, 1 row affected (0.01 sec)

mysql>
```

(the above code snippet online)



Insert Command

The `insert into customers` is used to insert rows inside the `customers` table. In order to succeed, we need to give the list of columns we will provide values for (1) and the actual values (2). The list of columns is given inside a comma separated list enclosed in parentheses (1). The list of values is given in another list (2), enclosed in parentheses too. The list of values needs to follow the `values` keyword (3).

Selecting Rows from Table

How do we really see that the customer `John Woo` has been stored inside the table `customers`? In order to do that we use the `select` command. Try to give the following command:

```
mysql> select * from customers;
```

(the above code snippet online)

This command will retrieve all the rows stored inside `customers` table and display them in a tabular format. You will see this:

```
+----+----------+
| id | name     |
+----+----------+
|  1 | John Woo |
+----+----------+
1 row in set (0.00 sec)
```

(the above code snippet online)

It gives you the values of all the columns for each row in the `customers` table. Currently, we only have 1 customer stored. Its `id` is `1` and has been assigned automatically by MySQL server (thanks to the `auto_increment` property of the `id` column). Its `name` is `'John Woo'` and has been assigned when we issued the `insert into` statement earlier.

The output also reports how long it took server to process the query and send the result back to `mysql` client.

Nice! This proves that our insert statement worked as expected.

Let's create another customer. Issue the following command on `mysql` prompt:

```
mysql> insert into customers (name) values ('John Papas');
```

(the above code snippet online)

This will create the customer with name `'John Papas'`. If we now issue the same `select` command we issued earlier we will get this:

```
mysql> select * from customers;
+----+------------+
| id | name       |
+----+------------+
|  1 | John Woo   |
|  2 | John Papas |
+----+------------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

As you can see, we now have 2 rows inside the `customers` table. The new customer, `'John Papas'`, has been created with an id of 2, which is the next increment of the previously inserted customer.

What about the * symbol used in the `select` statement? Between `select` and `from` we need to tell MySQL which columns of the table we want the result set to return back to us. With * we say that we want all the columns to be returned, a.k.a. `id` and `name` on our example.

If we want to include only the `name` in the result set, we need to explicitly specify that in our select command. Let's try that:

```
mysql> select name from customers;
+------------+
| name       |
+------------+
| John Woo   |
| John Papas |
+------------+
2 rows in set (0.01 sec)
```

(the above code snippet online)

You now see that the result set includes only the values for the `name` column. This feature might be quite useful because the table might have many columns and displaying all of them might not be easily readable, especially if we are only interested in a sub-set of the whole column set.

Let's ask only for the id:

```
mysql> select id from customers;
+----+
| id |
+----+
|  1 |
|  2 |
+----+
2 rows in set (0.00 sec)
```

(the above code snippet online)

As you can see, the result set now contains the values only for the `id` column.

If we want more than one column, explicitly defined, then we have to give their names in a comma separated list. For example, let's ask for the `id` and `name` columns explicitly:

```
mysql> select id, name from customers;
+----+------------+
| id | name       |
+----+------------+
|  1 | John Woo   |
|  2 | John Papas |
+----+------------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

Limiting the Rows Returned Using Criteria - where Clause

The previous select statements returned all the customers stored in our `customers` table. But this is not usually the case. Usually, the `customers` table contains many rows and we are never interested in enlisting all of them. We usually want to retrieve a sub-set of the customers, those that satisfy specific criteria.

Let's suppose that we want to retrieve the customer that has the name `'John Papas'`. In order to do that, we have to use the `where` clause. The `where` clause is given at the select statement after the name of the table which we retrieve data from. Try the following command:

```
mysql> select * from customers where name = 'John Papas';
```

(the above code snippet online)

You will get this:

```
+----+------------+
| id | name       |
+----+------------+
|  2 | John Papas |
+----+------------+
1 row in set (0.01 sec)
```

(the above code snippet online)

As you can see, the rows returned match the criteria that I have given: `name = 'John Papas'`. The criteria have been given after the `where` reserved word.

The criteria are usually composed of a column reference (`name` on our example), a comparison operator (= on our example) and a value (`'John Papas'` on our example). The column reference, the operator and the value compose a condition. On this particular example we have used only 1 condition. But this is not always the case. We might want to build more complex conditions, more complex criteria. This can be done by combining conditions with the use of boolean operators like `AND` and `OR`. Also, parentheses can be used to group conditions together in order to change the priority of evaluation. Note that `AND` is evaluated before `OR`.

On the previous example we have used the operator `=`. But we can use also operators like `>`, `>=` (greater than or equal), `<`, `<=` (less than or equal), `<>` (different from). Another useful operator is `like` and we are going to talk about that later on. We also have the unary operator `IS NULL`, which checks whether the column does not have value. And the negative form of it which is `IS NOT NULL`.

So, if we want to get the customer with `id` equal `2`, we can do that as follows:

```
mysql> select * from customers where id = 2;
+----+------------+
| id | name       |
+----+------------+
|  2 | John Papas |
+----+------------+
1 row in set (0.01 sec)

mysql>
```

(the above code snippet online)

Whereas the following statement:

```
mysql> select * from customers where id = 2 and name = 'John Woo';
```

(the above code snippet online)

is trying to retrieve all the customers that have `id` equal to `2` and `name` equal to `'John Papas'`. If you run that you will see that it will not return any row. There is no customer row satisfying those criteria.

> *Hint:* While on `mysql` prompt, you can always hit the <kbd>Up Arrow</kbd> button and go through your previous commands. This is handy because it will allow you to quickly issue a new command by only changing, a little bit, one of previous commands.

Now, let's change the previous command as follows: we will change the `and` operator to `or`.

```
mysql> select * from customers where id = 2 or name = 'John Woo';
+----+------------+
| id | name       |
+----+------------+
|  1 | John Woo   |
|  2 | John Papas |
+----+------------+
2 rows in set (0.01 sec)
```

(the above code snippet online)

You can see that it returns now two matching rows. The first one matching to criterion `name = 'John Woo'` and the second one to criterion `id = 2`. This is correct, because the `where` clause that we specified, instructs MySQL server to find all rows that have `id` equal to `2` and all rows that have `name` equal to `'John Papas'`.

Counting number of Entries

Some times we do not want to know the exact entries that a table contains, but only the number of the entries. In order to do that, we use the `select count(*) from <table_name> where <where_clause>` statement.

For example:

```
mysql> select count(*) from customers;
+----------+
| count(*) |
+----------+
|        2 |
+----------+
1 row in set (0.01 sec)
```

(the above code snippet online)

returns `2` telling us how many customers exist inside the `customers` table.

You can always use a `where` clause together with `count(*)`. In that case, the result will be the count of the rows that match the `where` clause criteria.

For example:

```
mysql> select count(*) from customers where name = 'John Papas';
+----------+
| count(*) |
+----------+
|        1 |
+----------+
1 row in set (0.00 sec)
```

(the above code snippet online)

would return `1`, because only 1 row / customer matches the criterion given, i.e. `name` being equal to `'John Papas'`.

`select count(*)` is very useful, especially when the result set is big. It does not bring data back to the client except from the counter. So, it is one of the best ways to calculate the number of rows inside tables with many entries.

Deleting Rows from Tables

We are now going to learn about the `delete` command. What if we wanted to delete the customer with name `'John Papas'`?

The command to delete this customer is:

```
mysql> delete from customers where name = 'John Papas';
Query OK, 1 row affected (0.01 sec)

mysql>
```

(the above code snippet online)

If you execute the above command you will see a `Query OK` and `1 row affected`. This confirms that the customer has been deleted. But we can still try to find the customer and double check that this customer does not exist:

```
mysql> select * from customers where name = 'John Papas';
Empty set (0.00 sec)
```

(the above code snippet online)

Or, you can list all of your customers (but only if they are few, otherwise it is useless) and visually inspect that the customer does not exist:

```
mysql> select * from customers;
+----+----------+
| id | name     |
+----+----------+
|  1 | John Woo |
+----+----------+
1 row in set (0.00 sec)
```

(the above code snippet online)

Our `customers` table now has only 1 customer. The customer with `id` 2 has been deleted, because it had the name equal to `'John Papas'` which was the name we used to delete: `delete from customers where name = 'John Papas';`.

The `delete` command is always followed by the pattern `from <table_name>`. This is because we need to tell MySQL which table we want to delete rows from. Then, we can optionally specify a `where` clause, like we did in previous `select` statements. All the rows that match the `where` clause will be deleted.

> **IMPORTANT**: Do not use `delete` without a `where` clause that specifies exactly which rows you want to delete, **unless you know what you are doing**. If you do not use a `where` clause, the `delete` command will delete all entries in your table, without any warning.
>
> *Hint:* I usually write a `select count(*) from <table_name> where <clause that will be used to delete>` statement to count the number of rows that would match the `where` clause to be used on next `delete` command. Hence, I know how many rows will be deleted before I actually delete them. Is that number the expected one? If not, I revise my `where` clause. When I am happy with the `where` clause on the `select` statement, then I copy the `where` clause from the `select` statement to the `delete` statement. That makes sure that I am going to `delete` the rows that I want.

Let's suppose now that we insert a new customer into our database `customers` table. Give the following command:

```
mysql> insert into customers (name) values ('Maria Foo');
Query OK, 1 row affected (0.00 sec)
```

(the above code snippet online)

You will get `Query OK`. Now, let's list all the customers again:

```
mysql> select * from customers;
+----+-----------+
| id | name      |
+----+-----------+
|  1 | John Woo  |
|  3 | Maria Foo |
+----+-----------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

We now see that we have a new customer. The new customer has the id `3`. This is because this is the first available / free id that can be used. The id `2` has been used earlier for customer `John Papas` and MySQL never reuses ids.

Updating Rows on a Table

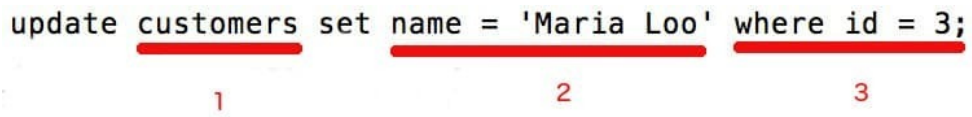
Let's say, now, that we want to change the value of an attribute for a specific customer. In order to do that we use the command `update`. If we want, for example, to change the name of `Maria Foo` to `Maria Moo`, then we give the following command:

```
mysql> update customers set name = 'Maria Moo' where id = 3;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

(the above code snippet online)

MySQL reports back with a `Query OK` and, also, with the number of rows that matched the `where` clause criteria, and the number of rows that have actually been changed.

The structure of the `update` command is more or less as follows:



Update Command

(1) After the `update` reserved word, we specify the name of the table that we want to update. Then we give the reserved word `set`.

(2) After the `set` word, we enlist in a comma separated list the columns and their new values: `<column> = <new_value>, <column> = <new_value> ...`

(3) Finally, we optionally specify a `where` clause to make sure that we update specific rows, unless we want to udpate all the rows in the table.

> **Important**: `update` command is equally dangerous like `delete` is. So, if you do not specify a `where` clause it will update all the rows in the table. Make sure that you use it with the same caution like the `delete` command.

Let's see the results of our update:

```
mysql> select * from customers;
+----+-----------+
| id | name      |
+----+-----------+
|  1 | John Woo  |
|  3 | Maria Moo |
+----+-----------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

You can see that the customer with id 3 has a new name: `'Maria Moo'`.

`like` operator

We will close this first SQL encounter, learning about the `like` operator, which is very useful in `where` clauses. Let's take an example.

Assume that we know that the name of our customer ends with `Woo`, but we do not really remember the exact name. How can we search for customers whose name ends with `Woo`. We will use the following statement:

```
mysql> select * from customers where name like '%Woo';
+----+----------+
| id | name     |
+----+----------+
|  1 | John Woo |
+----+----------+
1 row in set (0.03 sec)
```

(the above code snippet online)

It will return 1 customer, the one whose name ends with `Woo`. The string format `'%Woo'` given to `like` tells MySQL server to match the column value if it ends with `Woo`. The `%` means anything. Hence, the `%Woo` means anything to the left of `Woo`.

Now, let's take another example. Assume that we know that the customer we are looking for has a name that starts with `Maria`. In that case, we want the string to start with `Maria` and we don't care what follows. We are going to indicate that by positioning the `%` to the right of `Maria`. Like that:

```
mysql> select * from customers where name like 'Maria%';
+----+-----------+
| id | name      |
+----+-----------+
|  3 | Maria Moo |
+----+-----------+
1 row in set (0.00 sec)
```

(the above code snippet online)

Perfect! MySQL returned the single customer in our `customers` table that has name that starts with `Maria`.

Finally, note that the anything symbol, i.e. the `%` can be positioned in any position inside your search key as long as it makes sense for your search. For example, `'%opo%'` means any string that contains the sequence `opo` no matter what exists before or after it. Hence, it would equally match both `'Popolos'` and `'opolos'` and `'fopo'` and any other string that contains the sequence `opo`.

Closing Note

We have learnt how to create a database and our first table, `customers`. We have also learnt the basic SQL commands like:

- insert
- select
- update
- delete

If you want to watch this chapter in a video lecture you can watch this video here. The lecture has been compiled behind a Linux / Debian operating system. But all the concepts and hands on exercises presented there can be equally applied to any MySQL server.

SQL Basic Commands - Video Lecture

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Quiz

The quiz for this chapter can be found here

Task Details

Create a table with the following properties:

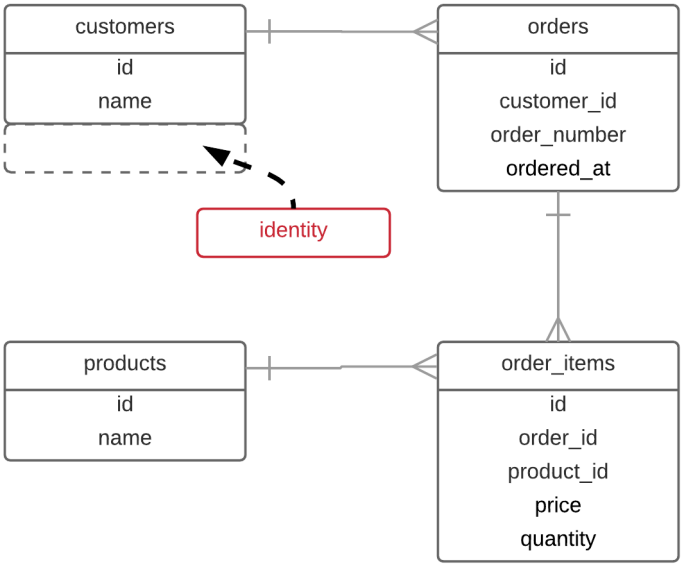
1. Table belongs to the database `school_db`. You have to create this database.
2. Name of the table is `students`.
3. Table has the following columns:
    1. column `id` which is an integer with display length 11. It should not be taking null values and it should be incremented.
    2. column `first_name` which is a string with maximum 50 characters. It should not be taking null values.
    3. column `last_name` which is a string with maximum 100 characters. It should not be taking null values.
4. Then insert the following data:
   `+------------+-----------+ | first_name | last_name | | John | Smith | | Mary | Foo | | Paul | Woo | +------------+-----------+`

Important: Your code (all SQL statements) needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

4 - Adding Columns And Indexes

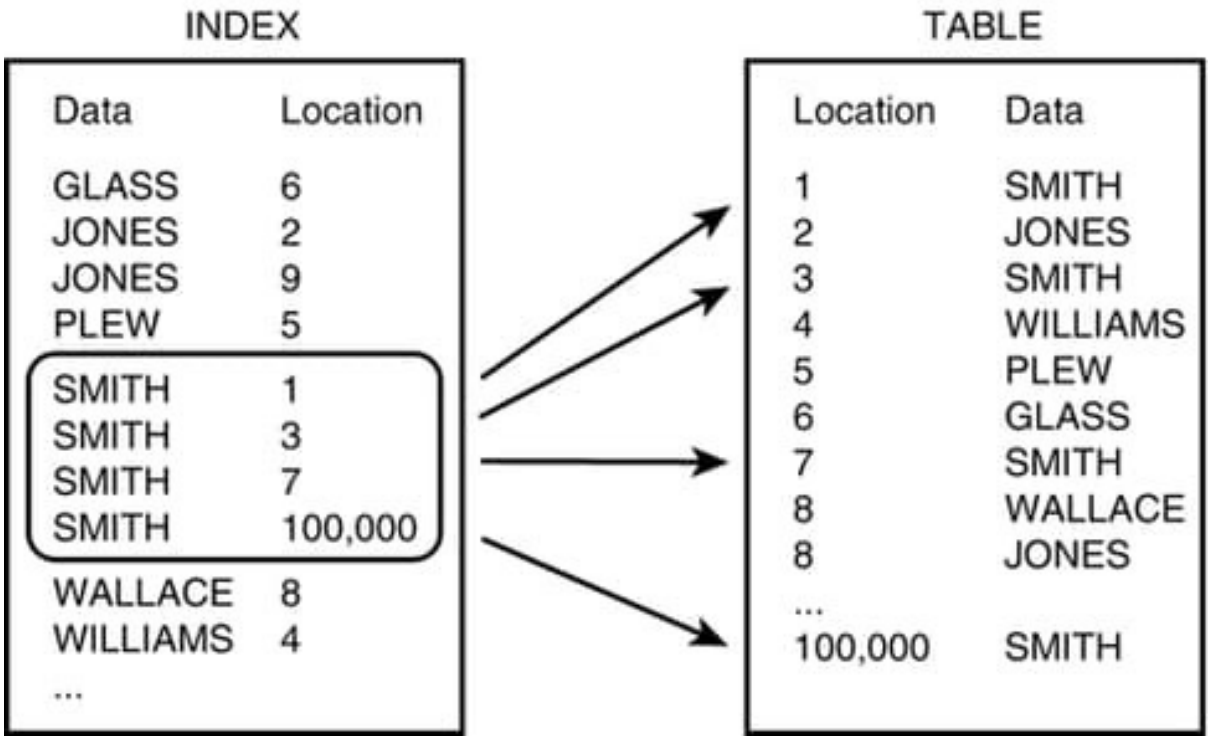
Summary

In this chapter you are going to learn how to add a new column to an existing table.



Adding New Column To Existing Table

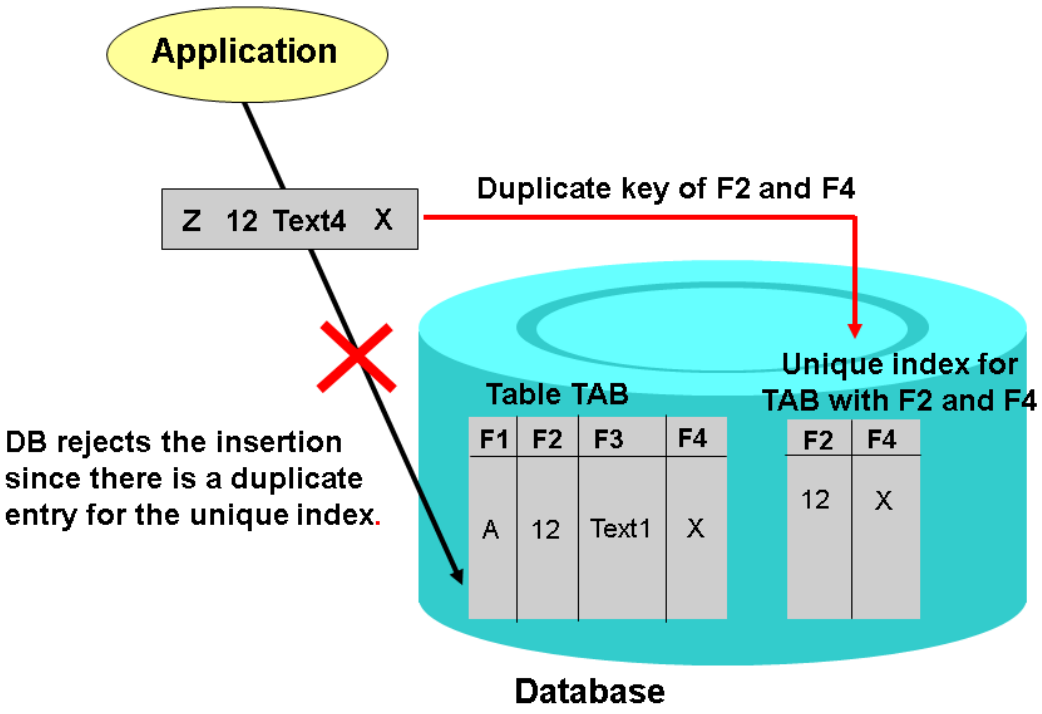
Then you are going to learn about indexes and how useful they are to MySQL good performance.



Database Index

You will also learn how to remove an index.

Finally, the very important concept of uniqueness is going to be explained.



Database Unique Index

Learning Goals

1. Learn how to add a new column to an existing table.
2. Learn what is an index.
3. Learn when and why an index is necessary.
4. Learn how to create an index.
5. Learn how to verify that an index has been created or not.
6. Learn which part of the table structure definition denotes the presence of an index.
7. Learn how you can remove an index.
8. Learn how can you verify that the index has been removed?
9. Learn about the unique indexes and when they are useful.
10. Learn how to verify that a unique index has been created.
11. Learn what response you get when you try to violate the rules of a unique index.

Introduction

In the previous chapter we created the database `customers_db` and the table `customers`. Also, we have learnt basic SQL commands, like, `insert`, `update`, `delete` and `select`.

Let's start our `mysql` client if it is not yet started.

```
mysql -u root
```

(the above code snippet online)

You should see something like this, which makes sure that you are ready to type in MySQL commands.

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.7.14 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

(the above code snippet online)

Let's switch to the `customers_db` with:

```
mysql> use customers_db;
```

(the above code snippet online)

Next, let's list the tables this database has:

```
mysql> show tables;
```

(the above code snippet online)

The result is:

```
+------------------------+
| Tables_in_customers_db |
+------------------------+
| customers              |
+------------------------+
1 row in set (0.01 sec)
```

(the above code snippet online)

which contains the single table that we have in our db, the table `customers`.

Let's also see the structure of this table:

```
mysql> show create table customers;
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------+
| Table     | Create Table                                                             \
                                                                                 \
                                                                                 |
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------+
| customers | CREATE TABLE `customers` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

(the above code snippet online)

Adding a New Column

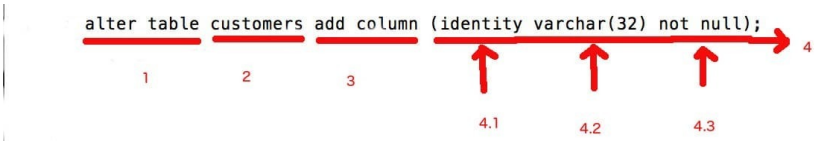
Let's suppose now that we want to introduce the column with name `identity`. We assume that each customer has a unique identity number and we want to introduce this concept on our data model.

How are we going to add the column? Here it is how. We will execute the following `alter table` command:

```
mysql> alter table customers add column (identity varchar(32) not null);
```

(the above code snippet online)

Before you actually execute that command, let's see the elements that are used to compose such a command.



Alter Table Command

(1) We add a new column to an existing table, by using the command `alter table`. Generally, `alter table` can be used to apply various table alteration. One of these is the addition of a collumn. We will see some other cases later on.

(2) Then we give the name of the table that we want to alter.

(3) Then we add the `add column` reserved words since we want to add a new column.

(4) In order to add the new column we need to specify the properties of the new column. The specification is given inside parentheses `( ... )`. What we put inside is similar to what we put when we create a table and we specify its columns.

- (4.1) First we give the name of the new column. It is `identity` on our case.
- (4.2) Then we give the type of the column. We give the type `varchar` and we specify the number 32. This means that the column is going to hold strings with 32 characters maximum.
- (4.3) With `not null` we specify that the column should always have a value. So, customers without an `identity` will not be accepted.

We are ready now to execute this command:

```
mysql> alter table customers add column (identity varchar(32) not null);          \
                                                                                   \

Query OK, 0 rows affected (0.13 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql>
```

(the above code snippet online)

We got back `Query OK`. Was the column added? Let's check the structure of the table with `show create table` command:

```
mysql> show create table customers;
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-----------------------------------------------------+
| Table     | Create Table                                                             \
                                                                                    \
                                                                                    \
                                                    |
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-----------------------------------------------------+
| customers | CREATE TABLE `customers` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `identity` varchar(32) COLLATE utf8_unicode_ci NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
```

```
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-----------------------------------------------------+
1 row in set (0.00 sec)

mysql>
```

(the above code snippet online)

As you can see, the identity column has been added.

Let's see this column on our data too.

```
mysql> select * from customers;
+----+-----------+----------+
| id | name      | identity |
+----+-----------+----------+
|  1 | John Woo  |          |
|  3 | Maria Moo |          |
+----+-----------+----------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

Because of the fact that the customers table already had some rows, before the addition of the new column, MySQL decided to assign a default value to existing rows. Hence, our 2 customers have the same `identity` value `''`.

Let's update the `identity` of our existing customers;

```
mysql> update customers set identity = 'JW0001' where id = 1;
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> update customers set identity = 'ML0001' where id = 3;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql>
```

(the above code snippet online)

We used 2 update commands. First to update the `identity` of the customer with `id` equal to `1`. Second, to update the `identity` of the customer with `id` equal to `3`. Let's fetch the customers again in order to verify the updates:

```
mysql> select * from customers;
+----+-----------+----------+
| id | name      | identity |
+----+-----------+----------+
|  1 | John Woo  | JW0001   |
|  3 | Maria Moo | ML0001   |
+----+-----------+----------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

As you can see both `identity` rows have been updated.

Obviously, you can retrieve the customers by `identity` now. For example:

```
mysql> select * from customers where identity = 'JW0001';
+----+----------+----------+
| id | name     | identity |
+----+----------+----------+
|  1 | John Woo | JW0001   |
+----+----------+----------+
1 row in set (0.00 sec)
```

(the above code snippet online)

How can we search for a customer if we only remember that his identity starts with `'JW'`? Easy stuff. We are using `like`:

```
mysql> select * from customers where identity like 'JW%';
+----+----------+----------+
| id | name     | identity |
+----+----------+----------+
|  1 | John Woo | JW0001   |
+----+----------+----------+
1 row in set (0.00 sec)
```

(the above code snippet online)

Indexes

Searching for a particular customer inside a table that has 2 customers is very quick. As you can see, the response is returned within `0.00 sec`. But, is that equally fast when table has 2 million customers? No way. When you issue the select statement

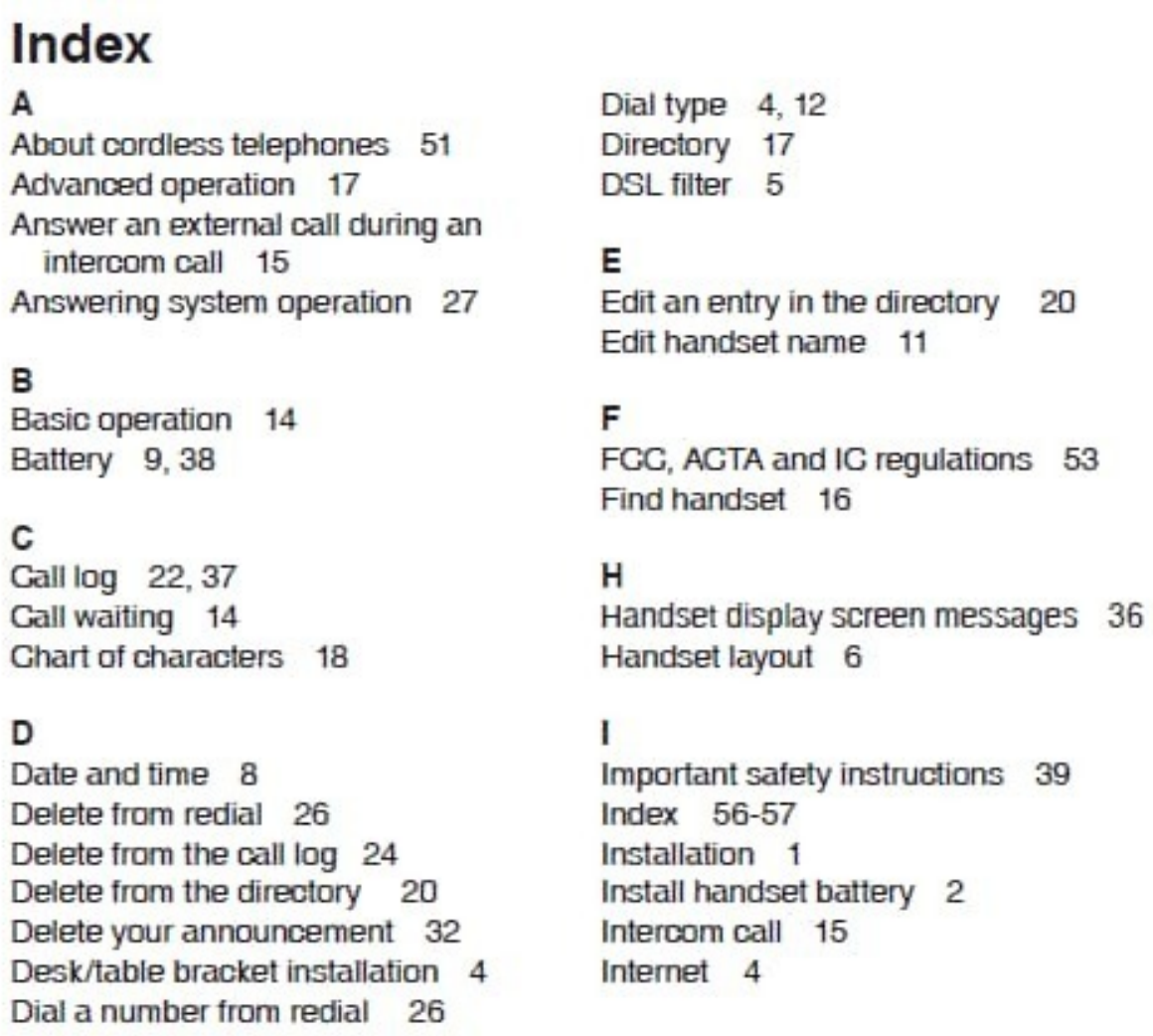
```
mysql> select * from customers where identity = 'ML0001';
```

(the above code snippet online)

for example, then MySQL will scan the whole table, starting from the 1st row, and taking one by one all the rows, until it matches the row with the required `identity` value. This is going to be very slow when table has a lot of rows. Actually, does not have to have millions of rows. Even with some thousands of rows, such queries will experience delays in returning back the results.

In order to speed up the response on such queries, we need to use *indexes*. Indexes is a tool that will help us search for specific data into a table very quickly. The index is being used by MySQL at the background, in order to retrieve data quickly.

Index is like the index one can find at the end of a book, with terms and pages the term can be found in, that helps reader quickly locate the page that references a term.



Index

Book Index

Whenever we insert a new row into an indexed table, this new row is automatically added to the index, i.e. the data this row contains are being analyzed and added to the corresponding index. Then whenever we use a `where` clause that references an indexed column, then rows matching the criteria given are quickly located.

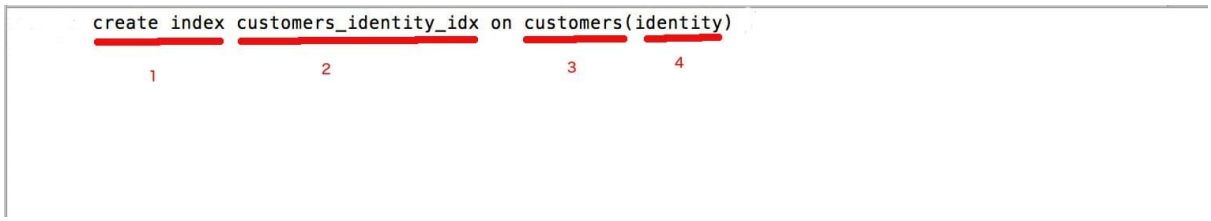
Creating an Index

Let's see an example. We want to create an index on the column `identity` of the table `customers`. The command is:

```
1  mysql> create index customers_identity_idx on customers(identity);
```

(the above code snippet online)

Before we actually execute that, let's analyze its format:



Create Index Command

(1) The create index command should start with `create index`.

(2) Then, we should give the name of the index. Each index needs to have a name and it can be anything. But, I am suggesting that you follow a pattern when naming your indexes. I follow the pattern `<table_name>_<column>_idx`. In other words, I construct the name of the index from the table name and the column names that are going to be indexed. Then I suffix with `_idx`. On our example the name of the index I decide to use is `customers_identity_idx`. This is because the index is going to index the column `identity` of the table `customers`.

(3) Then we specify the table that is going to be indexed.

(4) Finally, inside parentheses, we specify a comma separated list of the columns that are going to be indexed. On our example, we will build an index on a single column, the `identity`. There might be cases in which you might want to build an index on multiple columns. This is true when you query data using `where` clauses that reference more than 1 column. For example, if you find yourself executing a query such as `select * from customers where name = 'John Smith' and identity = 'JS0001'` then you may want to have defined an index on both columns (`create index customers_identity_idx on customers(name,identity);`)

Let's proceed with issuing the create index command:

```
1  mysql> create index customers_identity_idx on customers(identity);
2  Query OK, 0 rows affected (0.12 sec)
3  Records: 0  Duplicates: 0  Warnings: 0
```

(the above code snippet online)

You will see `Query OK` meaning that the command has been successfully executed by MySQL server.

How can we verify that the index has indeed been created? We can issue the `show create table` command:

```
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
--------------------+
| Table     | Create Table                                                             \
                                                                                     \
                                                                                     \
                                                                                     \
                    |
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
--------------------+
| customers | CREATE TABLE `customers` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `identity` varchar(32) COLLATE utf8_unicode_ci NOT NULL,
  PRIMARY KEY (`id`),
  KEY `customers_identity_idx` (`identity`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
--------------------+
1 row in set (0.01 sec)
```

(the above code snippet online)

The new entry here is the line:

```
KEY `customers_identity_idx` (`identity`)
```

(the above code snippet online)

The `KEY` indicates the definition of an index. You can see the name of the index next to the `KEY` word: `customers_identity_idx`. Then, in parentheses, you can see the columns that are indexed, i.e. `identity` on our example.

I guess that you have already identified the `KEY` word inside `PRIMARY KEY` specification too. Yes, you are right. `PRIMARY KEY` essentially defines an index. But this index is a special index that makes the column specified to be unique too. Also, this makes sure that you cannot have NULL values on the primary key column. Also, the primary key column is used to reference entries of the table from other tables, something that will be later explained when we will talk about foreign keys.

You need to know that you shouldn't add an index to a table if you do not really want it. The presence of an index on a column (or more) renders the DML commands (insert, update, delete) a little bit slower. This is because except from the fact that the actual table content needs to be amended, it is also, now, necessary to update the content of the corresponding indexes.

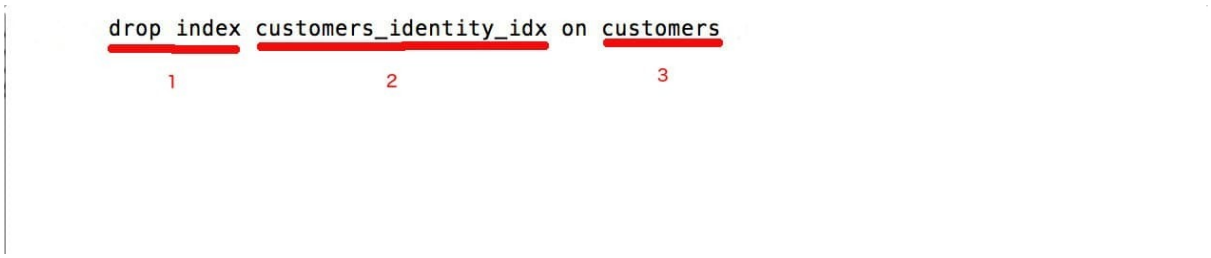
Removing an index

How can we remove an index that we do not need anymore? We can use the `drop index` command as in the following example:

```
1  mysql> drop index customers_identity_idx on customers;
```

(the above code snippet online)

But, before actually executing this command, let's see its constituent parts:



Drop Index Command

(1) The removal of an index can be accomplished with the command `drop index`.

(2) After `drop index` we need to give the name of the index.

(3) Then we need to specify the name of the table the index belongs to.

Let's execute the command:

```
1  mysql> drop index customers_identity_idx on customers;
2  Query OK, 0 rows affected (0.02 sec)
3  Records: 0  Duplicates: 0  Warnings: 0
4
5  mysql>
```

(the above code snippet online)

We can see `Query OK` which means that the command has been successfully executed by MySQL server. But how can we verify that the index does not exist anymore? With the `show create table` command:

```
mysql> show create table customers;
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-----------------------------------------------------+
| Table     | Create Table                                                             \
                                                                                     \
                                                                                     \
                                                    |
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-----------------------------------------------------+
| customers | CREATE TABLE `customers` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `identity` varchar(32) COLLATE utf8_unicode_ci NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+-----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-----------------------------------------------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

As you can read above, only the primary key index exists. The key index on `identity` column is no longer there.

Unique Indexes

Let's proceed now to another concept related to indexes.

First, let's see again the current customers:

```
mysql> select * from customers;
+----+-----------+----------+
| id | name      | identity |
+----+-----------+----------+
|  1 | John Woo  | JW0001   |
|  3 | Maria Moo | ML0001   |
+----+-----------+----------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

Now, let's execute the following command. This will try to create a new customer `'John Boo'`, but with `identity` value equal to the `identity` of an existing customer (that of `'John Woo'`).

```
mysql> insert into customers (name, identity) values ('John Boo', 'JW0001');
Query OK, 1 row affected (0.01 sec)
```

(the above code snippet online)

Here, we are doing a business error on purpose. We have inserted the new customer with the wrong identity, that of an existing customer. But, alas! Query has been successfully executed and the new customer has been inserted without any warning or error:

```
mysql> select * from customers;
+----+-----------+----------+
| id | name      | identity |
+----+-----------+----------+
|  1 | John Woo  | JW0001   |
|  3 | Maria Moo | ML0001   |
|  4 | John Boo  | JW0001   |
+----+-----------+----------+
3 rows in set (0.00 sec)
```

(the above code snippet online)

That is not good. Can we do something that would prevent us from such errors in the future? We need to make sure that MySQL server does not allow us to insert a new customer that has identity equal to the identity of another / existing customer. Or, in other words, the `identity` column should be `unique`.

The tool that we can use here is the `unique index`. The unique index is exactly like an index but does not allow the same value twice.

We will do that on our `customers` table. First, let's remove the *invalid* customer.

```
mysql> delete from customers where id = 4;
```

(the above code snippet online)

And then select to see that we have deleted the invalid customer:

```
mysql> select * from customers;
+----+-----------+----------+
| id | name      | identity |
+----+-----------+----------+
|  1 | John Woo  | JW0001   |
|  3 | Maria Moo | ML0001   |
+----+-----------+----------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

Let's create, now, a new index that is going to be unique index. The command is almost the same like the simple index case. Issue the following command:

```
mysql> create unique index customers_identity_uidx on customers(identity);
Query OK, 0 rows affected (0.04 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

(the above code snippet online)

The unique index is going to do 2 things at the same time. It's going to create an index on the `identity` column. And it will prevent any customers having the same `identity` value.

See, also, how I suffix the name of the index with `_uidx`, in order to indicate that this is the name of a unique index.

We can verify that the unique index has been created using the `show create table` command:

```
mysql> show create table customers;
+-----------+-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------------\
----------------------------+
| Table     | Create Table                                                                   \
                                                                                           \
                                                                                           \
                                                                                           \
                           |
+-----------+-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------------\
----------------------------+
| customers | CREATE TABLE `customers` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) COLLATE utf8_unicode_ci DEFAULT NULL,
  `identity` varchar(32) COLLATE utf8_unicode_ci NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `customers_identity_uidx` (`identity`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+-----------+-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------------\
----------------------------+
1 row in set (0.01 sec)
```

(the above code snippet online)

Do you see the

```
UNIQUE KEY `customers_identity_uidx` (`identity`)
```

(the above code snippet online)

line? This is the proof that `customers` table now has a unique index on the `identity` column.

But, we can also try to insert a duplicate identity and see how MySQL now behaves:

```
mysql> insert into customers (name, identity) values ('John Boo', 'JW0001');
ERROR 1062 (23000): Duplicate entry 'JW0001' for key 'customers_identity_uidx'
```

(the above code snippet online)

As you can see, an `ERROR` is returned by MySQL server. It clearly explains that you are trying to insert a duplicate entry for the unique index `customers_identity_uidx`. This means also that the insertion was not carried out. (Try to list the customers to verify that).

Now, the database design is such that the database server is protecting us. There is no way we can insert two customers with the same identity value.

Closing Note

We would like to close this chapter by giving you a link to video / screencast with the chapter content. You may want to watch it as an alternative source of learning.

Chapter Video / Screencast - Adding Columns and Indexes

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

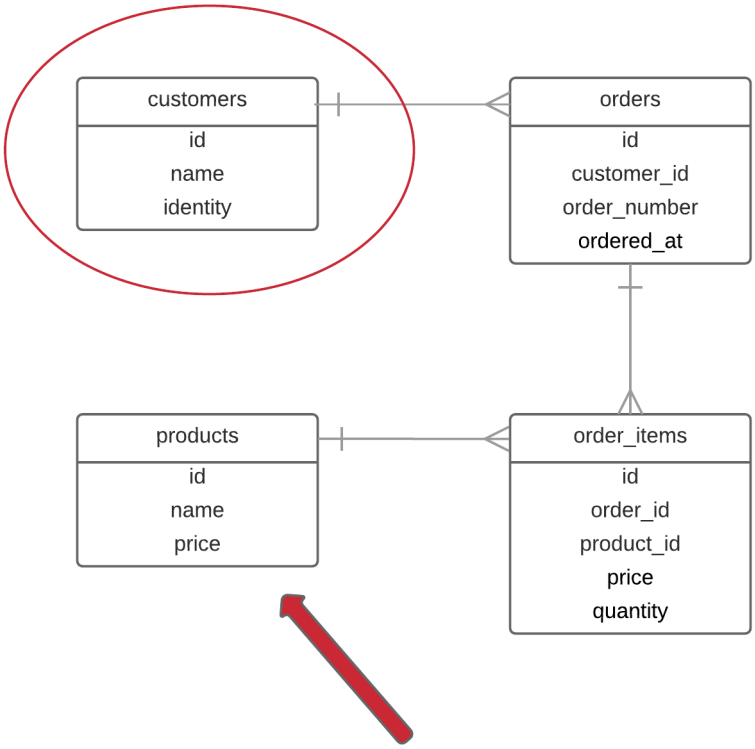
Quiz

The quiz for this chapter can be found here

5 - Counting And Sorting Records - Limiting Results

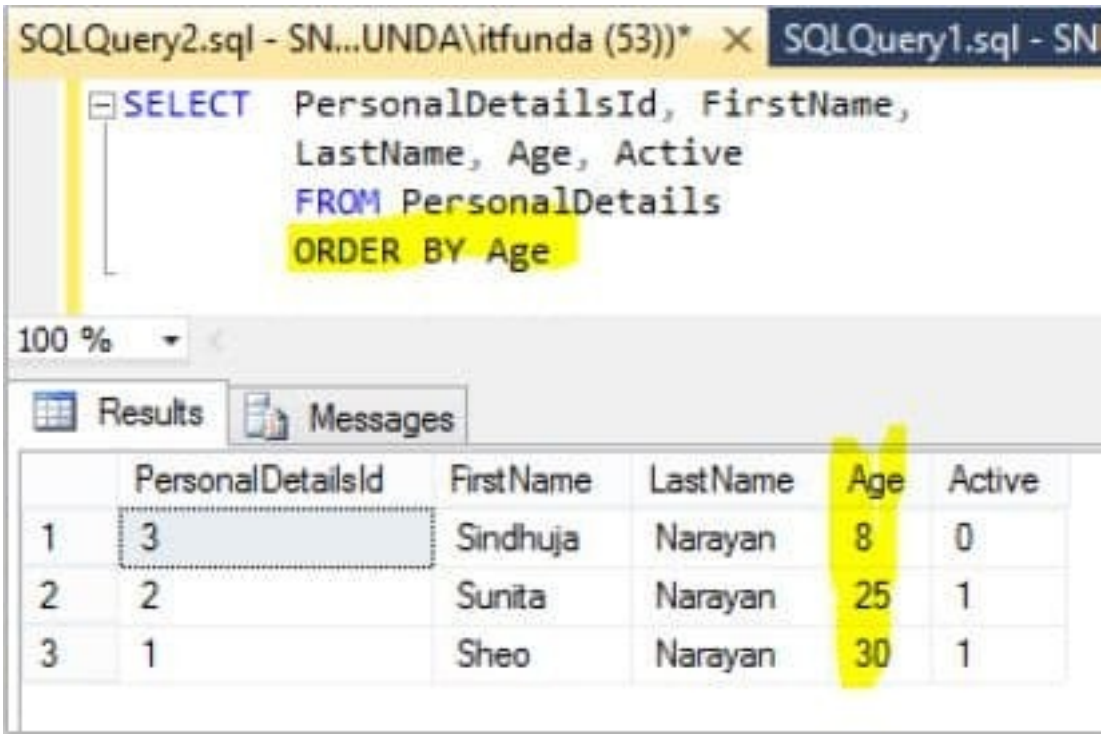
Summary

In this chapter, we are going to create a new table `products`, as we progress with the implementation of our primitive customer relationship management system.



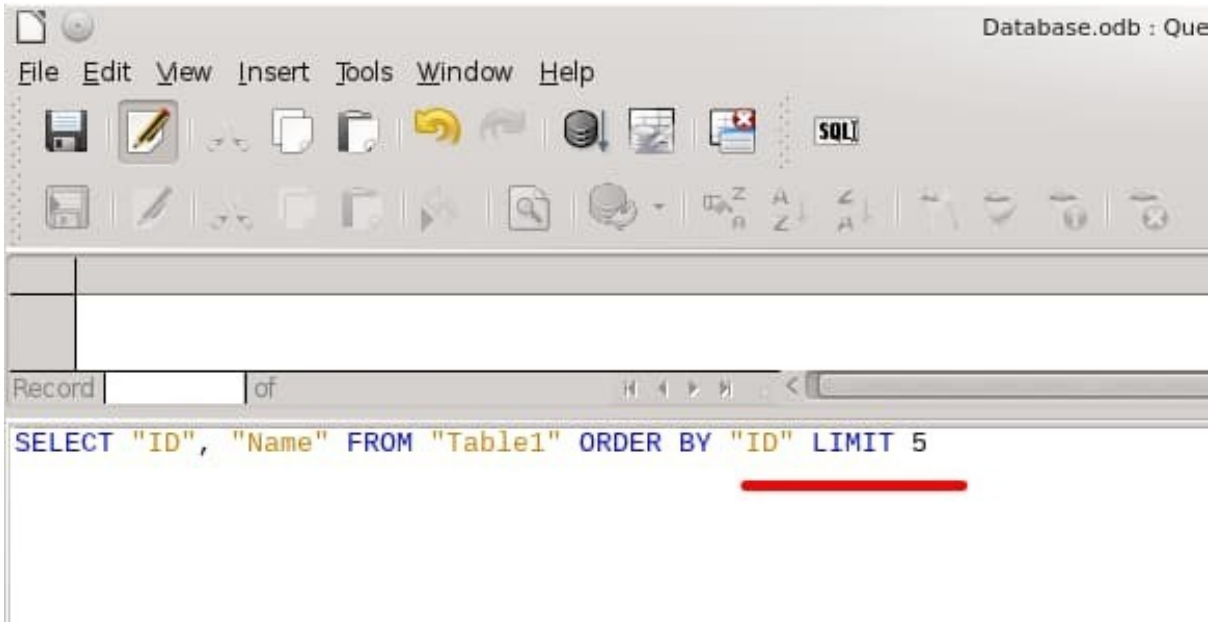
We'll Implement Table products

Besides that, you will learn how to sort records



Sorting Records

and how to limit the number of records returned.



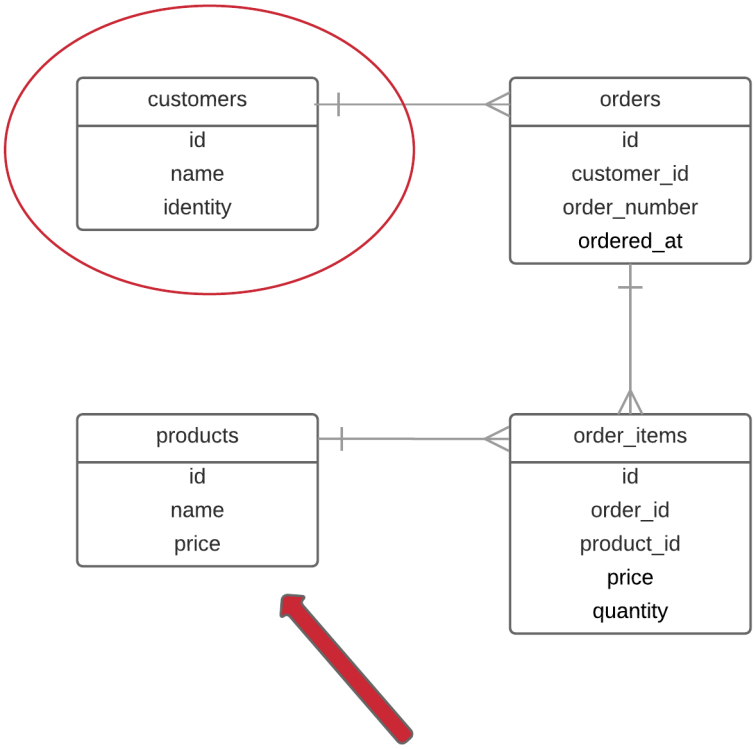
Limiting Number Of Records

Learning Goals

1. Learn how to specify a real number data type for a table column.
2. Learn how you can sort/order the results returned by any table column.
3. Limiting the number of records returned to a specific number.
4. Learn how to combine `where`, `order by` and `limit` in one select statement.

Introduction

In the previous chapters we have started implementing our primitive customer relationship management system. We have already implemented the `customers_db` and the table `customers`. In this chapter we are going to implement the table `products`.



We'll Implement Table products

Table products:

- Will have a primary key `id`. This will be integer, not null and auto incremented.
- It will have a `name`, which is going to be a string, not null and unique.
- It will have a `price`, which is going to be a decimal number, not null.

Let's start.

Initially, we open the terminal and we start `mysql` client:

```
$ mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2548
Server version: 5.7.14 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

(the above code snippet online)

We then switch to the `customers_db` database:

```
mysql> use customers_db;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

(the above code snippet online)

Let's see the tables that exist inside our database:

```
mysql> show tables;
+------------------------+
| Tables_in_customers_db |
+------------------------+
| customers              |
+------------------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

Creating Table products

We will now create the table `products`. You already know the command to create a table. Let's see its incarnation for the table `products`.

```
mysql> create table products (id int(11) not null auto_increment, name varchar(25\
5) not null, price numeric(15, 2) not null, primary key(id));
Query OK, 0 rows affected (0.05 sec)
```

(the above code snippet online)

The only new thing to you here is the `numeric(15, 2)` which is the type for the `price` column. It denotes that this column is going to hold real numbers, or decimal numbers as we sometimes call them. These numbers are going to have 2 decimal digits and their maximum number of digits is going to be 15.

Now, if you type `show tables` you will see the new table in the list of tables for the current database:

```
mysql> show tables;
+------------------------+
| Tables_in_customers_db |
+------------------------+
| customers              |
| products               |
+------------------------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

Let's also see the structure of this new table:

```
mysql> show create table products;
+----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
--------+
| Table    | Create Table                                                             \
                                                                                     \
                                                                                     \

        |
+----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
--------+
| products | CREATE TABLE `products` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  `price` decimal(15,2) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+----------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
```

```
----------------------------------------------------------------------------\
--------+
1 row in set (0.01 sec)
```

(the above code snippet online)

You can see that this new table has exactly 3 columns with the specifications that we have given while entering the `create table` command.

Unique Name

Now, we need to make sure that no 2 or more products have the same name. The name should be unique. Also, we need to make sure that we can quickly search for products using the name as criteria. In order to do that, we need to create a unique index on the `name` column.

You already know how to create a unique index. Let's try that:

```
mysql> create unique index products_name_uidx on products(name);
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

(the above code snippet online)

As you can see, we've got back `Query OK`. Let's verify the existence of the new unique index:

```
mysql> show create table products;
+----------+----------------------------------------------------------------------\
----------------------------------------------------------------------------\
----------------------------------------------------------------------------\
-----------------------------------------------------+
| Table    | Create Table                                                          \
                                                                                \
                                                                                \
                                                    |
+----------+----------------------------------------------------------------------\
----------------------------------------------------------------------------\
----------------------------------------------------------------------------\
-----------------------------------------------------+
| products | CREATE TABLE `products` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  `price` decimal(15,2) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `products_name_uidx` (`name`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+----------+----------------------------------------------------------------------\
----------------------------------------------------------------------------\
```

```
23 ------------------------------------------------------------------------------\
24 ------------------------------------------------+
25 1 row in set (0.00 sec)
```

(the above code snippet online)

As you can see, there is a new row in the specification:

```
1 UNIQUE KEY `products_name_uidx` (`name`)
```

(the above code snippet online)

which specifies the unique index on column `name`.

Inserting a Product

Having the table ready, let's insert a new product into the table:

```
1 mysql> insert into products (name, price) values ('Game of Thrones - S01 - DVD', \
2 50.0);
3 Query OK, 1 row affected (0.01 sec)
```

(the above code snippet online)

As you can see above, in the `insert` statement, we first specify the columns we will give values for, and then we give the values themselves.

Let's double check that the product has been created:

```
1 mysql> select * from products;
2 +----+-----------------------------+-------+
3 | id | name                        | price |
4 +----+-----------------------------+-------+
5 |  1 | Game of Thrones - S01 - DVD | 50.00 |
6 +----+-----------------------------+-------+
7 1 row in set (0.01 sec)
```

(the above code snippet online)

As you can see, the product has been created. Its `id` has been set to `1`, automatically. The `price` is a real number with 2 decimal digits.

Let's create one more product:

```
1 mysql> insert into products (name, price) values ('Of Mice and Men', 19.80);
2 Query OK, 1 row affected (0.01 sec)
```

(the above code snippet online)

And check that is has been successfully inserted:

```
mysql> select * from products;
+----+-----------------------------+-------+
| id | name                        | price |
+----+-----------------------------+-------+
|  1 | Game of Thrones - S01 - DVD | 50.00 |
|  2 | Of Mice and Men             | 19.80 |
+----+-----------------------------+-------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

Nice. The new product has `id` 2, which has been assigned automatically.

Let's insert one more product:

```
mysql> insert into products (name, price) values ('A Nice Story', 5.00);
Query OK, 1 row affected (0.00 sec)
```

(the above code snippet online)

and double check that it has been inserted:

```
mysql> select * from products;
+----+-----------------------------+-------+
| id | name                        | price |
+----+-----------------------------+-------+
|  1 | Game of Thrones - S01 - DVD | 50.00 |
|  2 | Of Mice and Men             | 19.80 |
|  3 | A Nice Story                |  5.00 |
+----+-----------------------------+-------+
3 rows in set (0.00 sec)
```

(the above code snippet online)

We now have 3 products in our product catalogue.

Counting

Suppose that we have a very long product catalogue and we want to count the number of products in our catalogue. The command to do that is the following:

```
mysql> select count(*) from products;
+----------+
| count(*) |
+----------+
|        3 |
+----------+
1 row in set (0.00 sec)
```

(the above code snippet online)

What if we wanted to count the products that have price greater than 10?

```
mysql> select count(*) from products where price > 10;
+----------+
| count(*) |
+----------+
|        2 |
+----------+
1 row in set (0.01 sec)
```

(the above code snippet online)

This returned 2 because we have only 2 products with price greater than 10.

Sorting - Ordering Results

There are many times that we want the results to be returned in a specific order. For example, let's suppose that we want to get the list of products in ascending order of name. This is how we do that:

```
mysql> select * from products order by name;
+----+-----------------------------+-------+
| id | name                        | price |
+----+-----------------------------+-------+
|  3 | A Nice Story                |  5.00 |
|  1 | Game of Thrones - S01 - DVD | 50.00 |
|  2 | Of Mice and Men             | 19.80 |
+----+-----------------------------+-------+
3 rows in set (0.00 sec)
```

(the above code snippet online)

As you can see, the results were now returned in an order different from the order returned by the `select * from products` command. The first returned product has a name which starts with `A`. The second starts with `G`. The third starts with `O`.

Similarly, if we wanted to sort by price, then we would issue the following command:

```
mysql> select * from products order by price;
+----+-----------------------------+-------+
| id | name                        | price |
+----+-----------------------------+-------+
|  3 | A Nice Story                |  5.00 |
|  2 | Of Mice and Men             | 19.80 |
|  1 | Game of Thrones - S01 - DVD | 50.00 |
+----+-----------------------------+-------+
3 rows in set (0.00 sec)
```

(the above code snippet online)

You can see that we first get the product with price `5.00`, then the product with price `19.80` and finally the product with price `50.00`.

If we want to list the products with the most expensive product at the top and the cheapest product at the bottom, then we would have to use the predicate `desc`, in order to specify that we want a descending order by name:

```
mysql> select * from products order by price desc;
+----+-----------------------------+-------+
| id | name                        | price |
+----+-----------------------------+-------+
|  1 | Game of Thrones - S01 - DVD | 50.00 |
|  2 | Of Mice and Men             | 19.80 |
|  3 | A Nice Story                |  5.00 |
+----+-----------------------------+-------+
3 rows in set (0.00 sec)
```

(the above code snippet online)

Limiting Number Of Records Returned

Sometimes we want to return the first row that matches the select criteria, even if more rows might match. We can do that using the `limit` at the end of the select statement.

For example, let's suppose that we want to get the most expensive product in our catalogue. This is returned with the following select statement:

```
mysql> select * from products order by price desc limit 1;
+----+-----------------------------+-------+
| id | name                        | price |
+----+-----------------------------+-------+
|  1 | Game of Thrones - S01 - DVD | 50.00 |
+----+-----------------------------+-------+
1 row in set (0.00 sec)
```

(the above code snippet online)

As you can see, we only have 1 record returned, thanks to the `limit 1` predicate.

Similarly, if we wanted to get the cheapest product only, we could do it with the following command:

```
mysql> select * from products order by price limit 1;
+----+--------------+-------+
| id | name         | price |
+----+--------------+-------+
|  3 | A Nice Story |  5.00 |
+----+--------------+-------+
1 row in set (0.00 sec)
```

(the above code snippet online)

Combining `where`, `order by` and `limit`

`where`, `order by` and `limit` can be combined together in one select statement, as long as it makes sense of course. For example:

```
mysql> select * from customers where name like '%oo' order by name limit 2;
```

(the above code snippet online)

Will bring the first 2 customers order by ascending order of name, whose name ends with `'oo'`.

Closing Note

We would like to close this chapter by giving you a link to a video / screencast with the chapter content. You may want to watch it as an alternative source of learning.

Chapter Video / Screencast - Counting and Sorting Records - Limiting Results

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

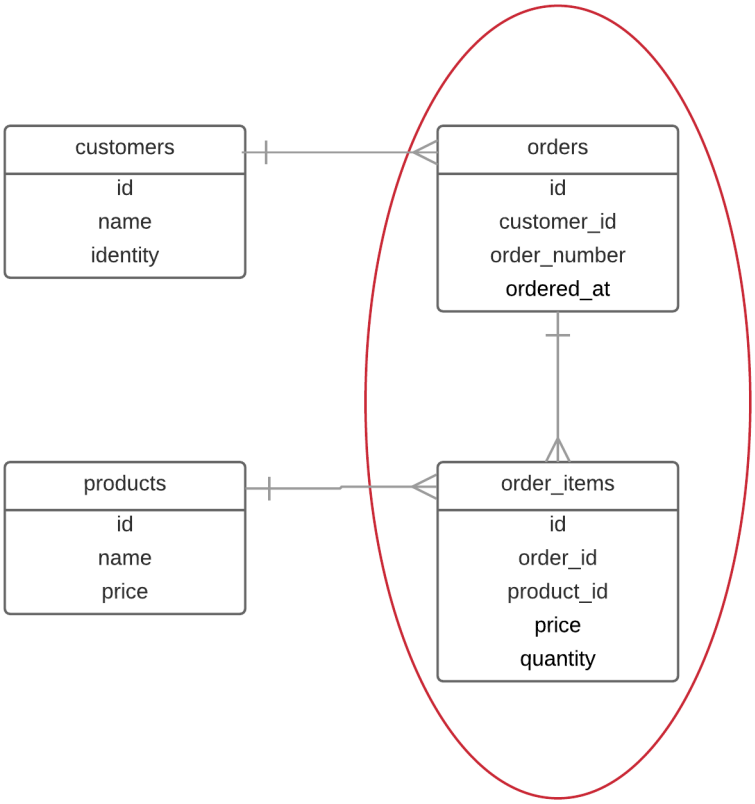
Quiz

The quiz for this chapter can be found here

6 - Foreign Keys

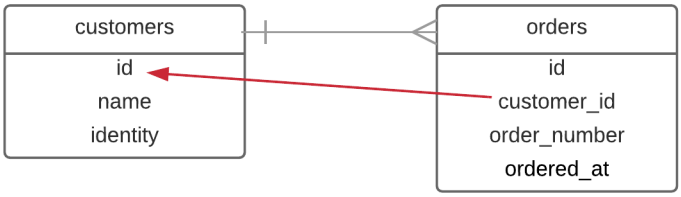
Summary

In this chapter you continue the database schema design for the primitive customer relationship management system.

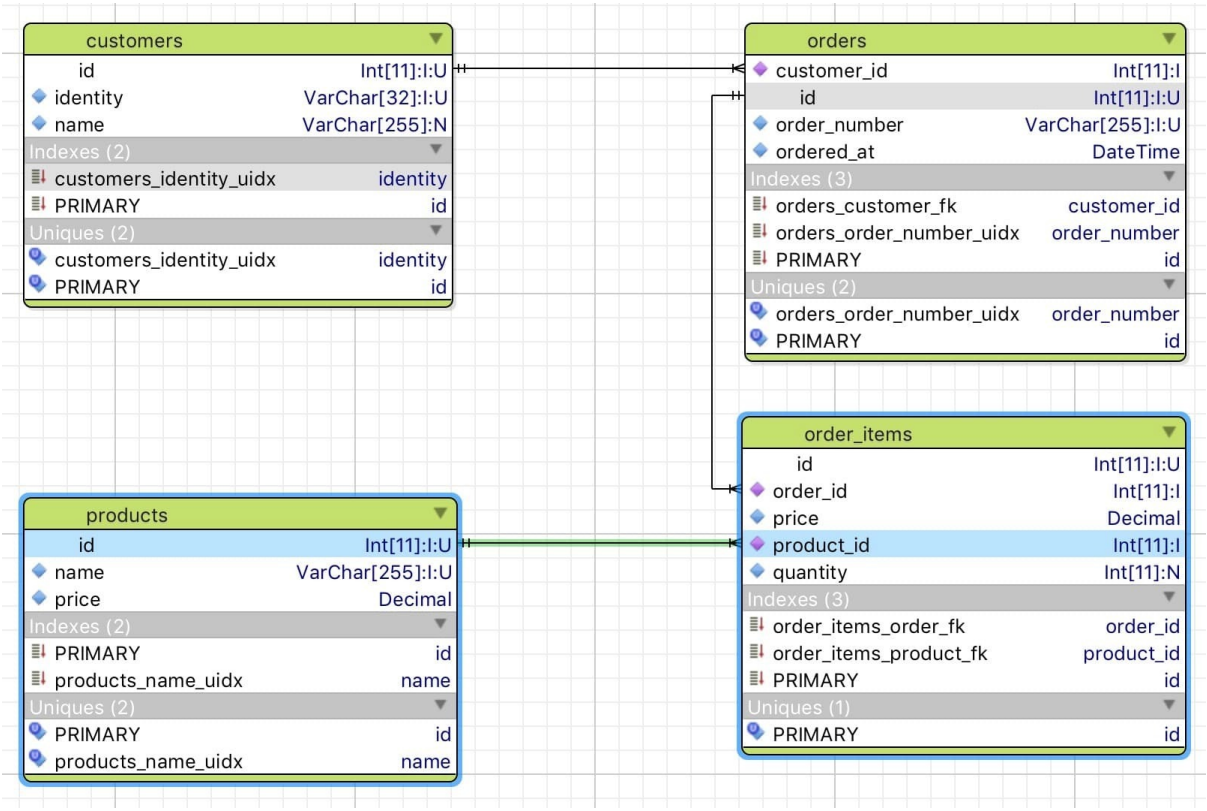


Creating Orders And Order Items

The main objective is for you to learn and practice creating relationships between tables, using foreign key constraints.

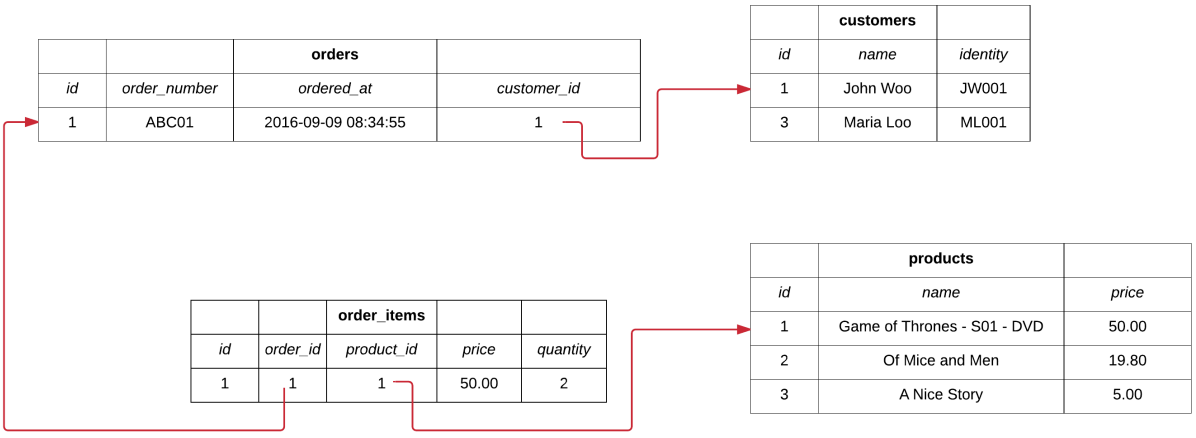


Relationships Between Tables



Final db Schema With All the Relationships Between Tables

Relationships between tables essentially mean data references from one table to another, like on the following example:



		orders	
id	*order_number*	*ordered_at*	*customer_id*
1	ABC01	2016-09-09 08:34:55	1

	customers	
id	*name*	*identity*
1	John Woo	JW001
3	Maria Loo	ML001

		order_items		
id	*order_id*	*product_id*	*price*	*quantity*
1	1	1	50.00	2

	products	
id	*name*	*price*
1	Game of Thrones - S01 - DVD	50.00
2	Of Mice and Men	19.80
3	A Nice Story	5.00

Data Relationships

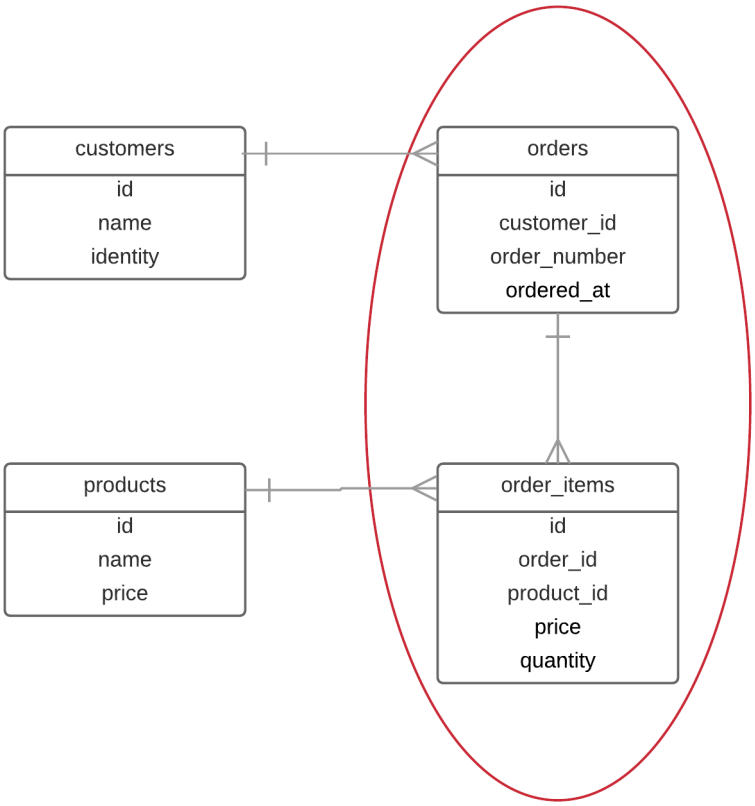
Learning Goals

1. Learn about the `datetime` type.
2. Learn about the `current_timestamp`.
3. Learn why a foreign key constraint is useful.
4. Learn how to create a foreign key constraint.

5. Learn how you should set the types of foreign keys.
6. Learn how to create an index that combines the values of two columns.

Introduction

We continue the creation of our primitive customer relationship management system. We have already created the tables `customers` and `products` and we are now ready to create 2 more tables, `orders` and `order_items`.



Creating Orders And Order Items

Model Details

`orders` table is holding the header of the order. It has a reference to the owning customer, by the `customer_id` column. This is going to be a foreign key as we call it. Given 1 order, we can locate the single 1 customer that the order belongs to. That is why the order-to-customer relationship is a 1-to-1 relationship. On the other hand, given 1 specific customer, we can locate more than 1, many, orders that belong to the particular customer. Hence, the relationship customer-to-orders is a 1-to-many relationship.

Besides that, `orders` also has the `order_number`. This is going to be mandatory and it's going to have a string representation. For example `ORD123`. It is also going to be unique. I.e. each order

number will be unique within the table and not two or more orders will have the same order number. Also, it will be used to quickly locate an order by order number.

There is also another column called `ordered_at` that will be used to record the date and time an order has been placed. This is going to be mandatory too.

`order_items` table is holding the details of an order. Since each entry in this table belongs to an order, we need the `order_id` column to be a reference to the owning order. This is another foreign key in our database. Also, given 1 order item we can locate the single 1 order the item belongs to, hence, the order_item-to-order relationship is a 1-to-1 relationship. On the other hand, given 1 order we can locate more than 1, many, order items belonging to this particular order. Hence, the order-to-order_items relationship is a 1-to-many relationship.

Moreover, each order item needs to specify which product it is referring to. We see here the existence of another foreign key, the `product_id`. This is a reference to the product that the order item refers to. Given 1 specific order item, then we can locate the single 1 product that this order item belongs to. Hence, the relationship order_item-to-product is a 1-to-1 relationship. On the other hand, given a specific 1 product, we can locate many order items referring to that particular product. Hence, the relationship product-to-order_items is a 1-to-many relationship.

`order_items` table also has a `price` column. This is going to be a number with 15 digits length and 2 decimal digits rounding. It is supposed to be holding the price of the product sold at the moment the order item is created. This is going to be mandatory.

For educational reasons, we have decided to prevent the same product appearing in the same order more than once. For that reason, if an order were to include the same product more than once, then there would be an order item referencing that particular product and having `quantity` equal to the number of instances of the particular product being in the order. Having said that, how can we ask RDBMS help in implementing this constraint? Maybe a unique index? Is this were to be on the table `order_items` on `product_id`? Would that be ok? No, because it wouldn't allow the same product to be sold on two different orders. So, the uniqueness needs to be the combination of the `product_id` and `order_id`. We will see how we are going to create a unique index on that.

Implementation

We will start with the `orders` table. We start `mysql`:

```
$ mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.14 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

```

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

(the above code snippet online)

Then we switch to `customers_db`:

```
mysql> use customers_db;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

(the above code snippet online)

And let's see the tables that our database has:

```
mysql> show tables;
+------------------------+
| Tables_in_customers_db |
+------------------------+
| customers              |
| products               |
+------------------------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

As you can see, we have the 2 tables, `customers` and `products`.

Create `orders` table

Let's now create the `orders` table. The command to create this table is the following:

```
create table orders (id int(11) not null auto_increment, order_number varchar(255\
) not null, ordered_at datetime not null, customer_id int(11) not null, primary k\
ey(id));
```

(the above code snippet online)

Before we actually execute this command, let me tell you that there is only 1 new thing for you here. It is the `datetime` type that is used for the `ordered_at` column. This type is used to define columns that will be holding a date and time stamp.

Let's execute the command:

```
mysql> create table orders (id int(11) not null auto_increment, order_number varc\
har(255) not null, ordered_at datetime not null, customer_id int(11) not null, pr\
imary key(id));
Query OK, 0 rows affected (0.04 sec)
```

(the above code snippet online)

We get back `Query OK` which means that the table has been created.

Let's see the list of tables:

```
mysql> show tables;
+------------------------+
| Tables_in_customers_db |
+------------------------+
| customers              |
| orders                 |
| products               |
+------------------------+
3 rows in set (0.00 sec)
```

(the above code snippet online)

This means that the table `orders` has been created. And let's see the actual structure of the new table:

```
mysql> show create table orders;
+--------+-----------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-----------------------------------------------+
| Table  | Create Table                                                          \
                                                                               \
                                                                               \
                                               |
+--------+-----------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-----------------------------------------------+
| orders | CREATE TABLE `orders` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `order_number` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  `ordered_at` datetime NOT NULL,
  `customer_id` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+--------+-----------------------------------------------------------------------\
-------------------------------------------------------------------------------\
```

```
23 ------------------------------------------------------------------------------\
24 ----------------------------------------------+
25 1 row in set (0.00 sec)
```

(the above code snippet online)

This table is missing something that will physically connect it to the `customers` table. Currently, the fact that we have a column `customer_id` does not enforce any rules, any constraints at the database level. In other words, MySQL does not really know that there is a relationship between the table `orders` and the table `customers`. But, we are going to deal with that a little bit later.

Inserting Orders

Let's see our customers:

```
1 mysql> select * from customers;
2 +----+-----------+----------+
3 | id | name      | identity |
4 +----+-----------+----------+
5 |  1 | John Woo  | JW0001   |
6 |  3 | Maria Moo | ML0001   |
7 +----+-----------+----------+
8 2 rows in set (0.01 sec)
```

(the above code snippet online)

Now, we are going to insert a new order in the `orders` table. We will use the following command:

```
1 insert into orders (order_number, ordered_at, customer_id) values ('ABC001', curr\
2 ent_timestamp, 1);
```

(the above code snippet online)

Before we actually execute this command, let see the new stuff here. It is the `current_timestamp`. This is used as a value for the `ordered_at` column. It will ask MySQL to set the current date and time stamp. So, we do not have to specify any precise date and time ourselves.

Also, see the number value `1` for the `customer_id`. This is the id of the customer this new order is going to be linked to. In other words, we want the new order created to be owned by the customer with id `1`, which is the customer `John Woo`.

As you can see, we are using the integer numbers, the ids, to reference a `customers` entry from an entry inside `orders` table. The integer values, the ids of the customers are stored inside the column `customer_id` of the `orders` table. That is why, the type of the column `customer_id` is the same as the type of the column `id` on the `customers` table (they are both `int(11)`).

Now, let's execute the command:

```
mysql> insert into orders (order_number, ordered_at, customer_id) values ('ABC001\
', current_timestamp, 1);
Query OK, 1 row affected (0.00 sec)
```

(the above code snippet online)

As you can see, we got a `Query OK` back. Let's confirm the insertion of the new order:

```
mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  1 | ABC001       | 2016-09-09 08:34:55 |           1 |
+----+--------------+---------------------+-------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

The new order has been inserted. Its `id` is given automatically and it is `1`. Also, as you can see the `current_timestamp` has been automatically converted to the current date and time value, thanks to MySQL doing the work for us.

> *Note:* The value for `ordered_at` at your order will be different, depending, of course, at the date and time you executed this command, your current date and time stamp, in your time zone.

Inserting Wrong Data

Now, let us try something else. Let us try the following command:

```
insert into orders (order_number, ordered_at, customer_id) values ('ABC001', curr\
ent_timestamp, 3);
```

(the above code snippet online)

Before we execute the command, let me underline the following. Here, we are trying to insert a new order with the same order number as the existing one. This is a business mistake. There shouldn't be two or more orders with the same order number. In other words, the order number should be unique within the whole orders table.

But, even if this order is supposed to be invalid, let's try to do the insertion:

```
mysql> insert into orders (order_number, ordered_at, customer_id) values ('ABC001\
', current_timestamp, 3);
Query OK, 1 row affected (0.00 sec)

mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  1 | ABC001       | 2016-09-09 08:34:55 |           1 |
|  2 | ABC001       | 2016-09-09 08:44:44 |           3 |
+----+--------------+---------------------+-------------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

We get `Query OK` and if we see the list of the orders, we will see the new order inserted without a problem.

But, since this is not a valid business case, we will ask the help of the database to protect us from inserting such bad data.

Unique order number

Let's first delete the wrong order, the order with id `2`.

```
mysql> delete from orders where id = 2;
Query OK, 1 row affected (0.01 sec)

mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  1 | ABC001       | 2016-09-09 08:34:55 |           1 |
+----+--------------+---------------------+-------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

We will now create a unique index on the `order_number` of the `orders` table. This will both make my search queries (ones using order number) quick and will also protect me from inserting the same order number twice.

This is the command that will create the unique index. Let's execute it:

```
mysql> create unique index orders_order_number_uidx on orders(order_number);
Query OK, 0 rows affected (0.04 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

(the above code snippet online)

This is not something new to you. Note the convention on the name of the index: `orders_order_number_uidx`.

We got the `Query OK`. Let's examine the structure of the `orders` table to make sure that the index has been created:

```
+--------+-----------------------------------------------------------------------\
---------------------------------------------------------------------------------\
---------------------------------------------------------------------------------\
---------------------------------------------------------------------------------\
----------------------------------------+
| Table  | Create Table                                                          \
                                                                                 \
                                                                                 \
                                                                                 \
                                        |
+--------+-----------------------------------------------------------------------\
---------------------------------------------------------------------------------\
---------------------------------------------------------------------------------\
---------------------------------------------------------------------------------\
----------------------------------------+
| orders | CREATE TABLE `orders` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `order_number` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  `ordered_at` datetime NOT NULL,
  `customer_id` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `orders_order_number_uidx` (`order_number`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+--------+-----------------------------------------------------------------------\
---------------------------------------------------------------------------------\
---------------------------------------------------------------------------------\
---------------------------------------------------------------------------------\
----------------------------------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

As you can see we now have the specification:

```
UNIQUE KEY `orders_order_number_uidx` (`order_number`)
```

(the above code snippet online)

which denotes the presence of a unique index on the `order_number` column.

Now, let's try to insert the problematic order, the one that has same order number as the existing one:

```
mysql> insert into orders (order_number, ordered_at, customer_id) values ('ABC001\
', current_timestamp, 3);
ERROR 1062 (23000): Duplicate entry 'ABC001' for key 'orders_order_number_uidx'
```

(the above code snippet online)

If you try to do that, you now get an `ERROR` with the description `Duplicate entry 'ABC001' for key 'orders_order_number_uidx'`, which is precisely the protection that we needed, protecting us from inserting different orders with same order number.

Inserting Wrong Customer Id

Let's see our customers and our orders again:

```
mysql> select * from customers;
+----+-----------+----------+
| id | name      | identity |
+----+-----------+----------+
|  1 | John Woo  | JW0001   |
|  3 | Maria Moo | ML0001   |
+----+-----------+----------+
2 rows in set (0.00 sec)

mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  1 | ABC001       | 2016-09-09 08:34:55 |           1 |
+----+--------------+---------------------+-------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

As you can see, the order with `id 1` refers to the customer with id `1` via the `customer_id` column.

Now, I am going to try to insert a new order that references a non-existing customer. Try the following command:

```
mysql> insert into orders (order_number, ordered_at, customer_id) values ('ABC002\
', current_timestamp, 4);
Query OK, 1 row affected (0.01 sec)

mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  1 | ABC001       | 2016-09-09 08:34:55 |           1 |
|  4 | ABC002       | 2016-09-09 08:55:20 |           4 |
+----+--------------+---------------------+-------------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

Ooops! The order has been inserted without any problem (error or warning). That's not good. Orders that refer to non-existing customers are not good. They are invalid data. This is where the statement *MySQL does not really know that there is a relationship between `orders` and `customers` table* applies.

We now really need to make this relationship explicit and make MySQL protect us from inserting such invalid data.

In order to make the relationship explicit, we need to create a database level constraint, which is called *foreign key constraint*.

Ok. Let's do that. But first, let's delete the bad order:

```
mysql> delete from orders where id = 4;
Query OK, 1 row affected (0.00 sec)

mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  1 | ABC001       | 2016-09-09 08:34:55 |           1 |
+----+--------------+---------------------+-------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

The command to create the foreign key constraint is the following:

```
alter table orders add constraint orders_customer_fk foreign key(customer_id) ref\
erences customers(id);
```

(the above code snippet online)

Before we actually execute that, let's study its constituent parts:

```
alter table orders add constraint orders_customer_fk foreign key(customer_id) references customers(id);
```

1 2 3 4 5 6

Add Foreign Key Constraint

(1) The command that we use to create a foreign key constraint on an existing table is a version of the `alter table` command.

(2) Then we give the name of the table.

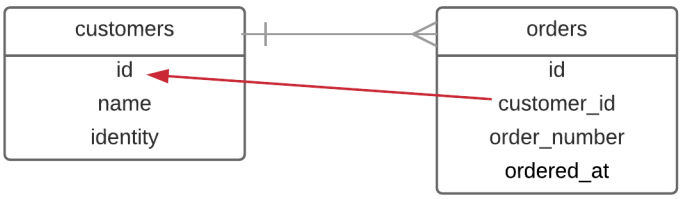
(3) Then we give `add constraint`. Remember that the last time we used the `alter table` command was to add a new column. In that case we used `add column`.

(4) We give the name of the constraint. This can be anything, but I am suggesting that you follow a convention. When it is a foreign key constraint[*], I follow the convention `<altered_table_name>_<foreign_table_name_in_singular_or_plural_form>_fk`. Hence, for our example, we use `orders_customer_fk`.

> Note: There may be other types of constraints, which are out of the scope of this course.

(5) We add `foreign key(<name_of_the_column_holding_the_reference/foreign_value_ids>)`. So, for our example, it is `foreign key(customer_id)`, because it is the `customer_id` column that is going to hold the reference/foreign value identifiers.

(6) We add `references <referenced_table_name>(<referenced_column_name>)`. For our example, the `referenced_table_name` needs to be `customers`, because this is the table we are referencing with `customer_id` foreign key. And the `referenced_column_name` needs to be `id` because this is the referenced column of `customers` table that will be referenced by the `customer_id` column.



Orders table customer_id column referencing id column of table customers

Let's now execute the command:

```
mysql> alter table orders add constraint orders_customer_fk foreign key(customer_\
id) references customers(id);
Query OK, 1 row affected (0.04 sec)
Records: 1 Duplicates: 0 Warnings: 0
```

(the above code snippet online)

As you can see, the command has been executed successfully. We can see the `Query OK`.

Let's see how this has affected the structure of the table `orders`:

```
mysql> show create table orders;
+--------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
---------------+
| Table  | Create Table                                                            \
                                                                                \
                                                                                \
                                                                                \
                                                                                \
                                                                                \

                |
+--------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
---------------+
| orders | CREATE TABLE `orders` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `order_number` varchar(255) COLLATE utf8_unicode_ci NOT NULL,
  `ordered_at` datetime NOT NULL,
  `customer_id` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `orders_order_number_uidx` (`order_number`),
  KEY `orders_customer_fk` (`customer_id`),
  CONSTRAINT `orders_customer_fk` FOREIGN KEY (`customer_id`) REFERENCES `custome\
rs` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=5 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+--------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
```

```
----------------------------------------------------------------------------\
----------------------------------------------------------------------------\
----------------------------------------------------------------------------\
---------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

There are 2 new things in the structure of the table:

(1) a new index with name `orders_customer_fk` on the column `customer_id`:

```
KEY `orders_customer_fk` (`customer_id`)
```

(the above code snippet online)

This is **automatically** added by MySQL every time we add a foreign key constraint. This is done in order to speed up the SQL queries that combine information from the two tables, `orders` and `customers` based on the relationship built between `customer_id` of `orders` and `id` of `customers`. We will come back to that later on, when we will talk about joins.

> **Important**: Not all RDBMS systems create this index automatically. MySQL does, but, for example PostgreSQL does not. We basically believe that this index is necessary. So, if you work with an RDBMS that does not add this index automatically when you add the foreign key constraint, then go ahead and add it yourself, with the use of the command `create index`.

> **Important**: If you first create the indexes necessary to support queries and unique constraints and then you create your foreign keys, MySQL might decided that it is not necessary to create an extra index to pair the foreign key, if the creation of that extra index is redundant, i.e. when there is already another index that could support the queries that join two tables using this foreign key constraint.

and

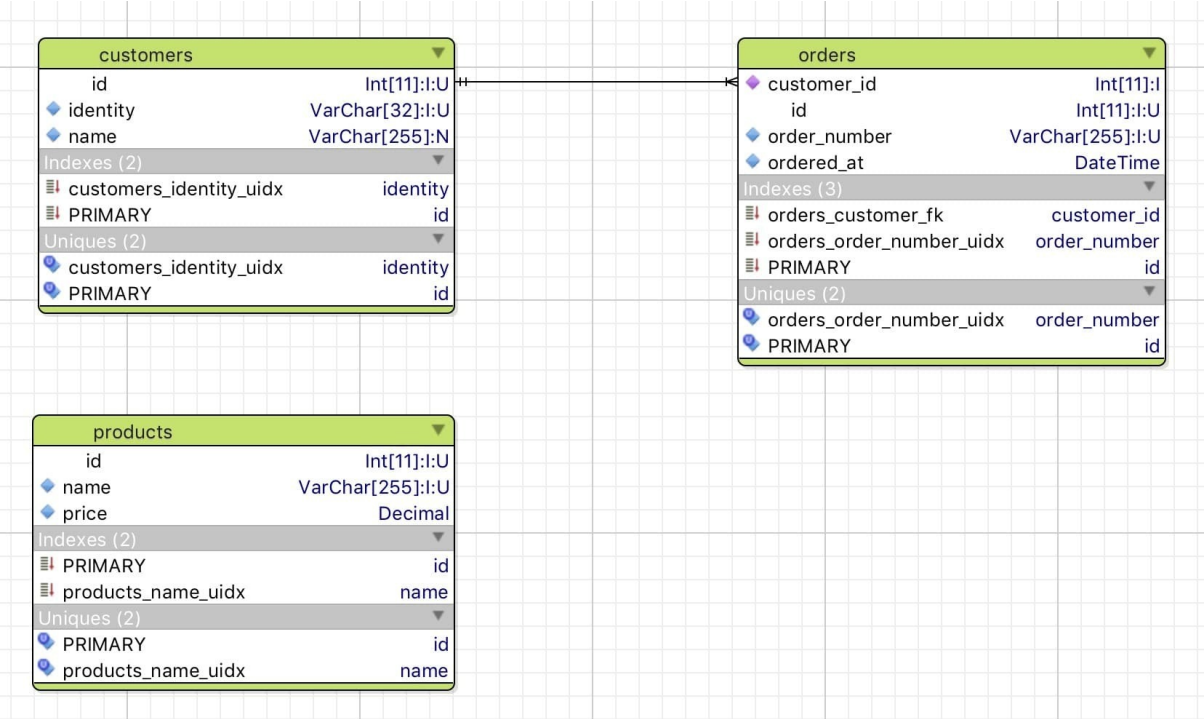
(2) a new constraint of type foreign key:

```
CONSTRAINT `orders_customer_fk` FOREIGN KEY (`customer_id`) REFERENCES `customers\
` (`id`)
```

(the above code snippet online) I guess this was expected, since it matches the command `alter table` that we issued.

Nice. We now have a real relationship between the table `orders` and the table `customers`. This relationship is also reversed engineered by tools that build ERD (Entity Relationship Diagrams) after reading the internals of the database schema.

One such tool is Valentina Studio which is free to install and works on many platforms. Here is the diagram that Valentina has built for our `customers_db` database:



Valentina Studio Diagram for customers_db

[*] At the end of this chapter you will find a video explaining how one can create an ERD using Valentina Studio

Try to Insert Bad Data Again

You remember that we have managed to create bad data (an order referencing a non-existing customer), and because of this, we have now created the constraint, in order to ask MySQL to protect us from this wrong action.

We will now try to insert the bad data again and see how MySQL behaves.

Let's see our customers.

```
mysql> select * from customers;
+----+-----------+----------+
| id | name      | identity |
+----+-----------+----------+
|  1 | John Woo  | JW0001   |
|  3 | Maria Moo | ML0001   |
+----+-----------+----------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

Let's try to add a new order that refers to a non-existing customer with id 4:

```
mysql> insert into orders (order_number, ordered_at, customer_id) values ('ABC002\
', current_timestamp, 4);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint fa\
ils (`customers_db`.`orders`, CONSTRAINT `orders_customer_fk` FOREIGN KEY (`custo\
mer_id`) REFERENCES `customers` (`id`))
```

(the above code snippet online)

Great! This one now failed, with `ERROR 1452` and descriptive message clearly mentioning that the error is due to the fact that a foreign key constraint rule fails. It also gives you the definition of the constraint that failed in order for you to clearly understand what was the root cause of the problem.

This is the `R` part of the `RDBMS` in action. `Relational`. We can see how MySQL, our RDBMS, is using the relationships between the tables to keep our data valid.

Now the `orders` table is finally ready.

Creating `order_items` table

`order_items` table is going to be holding the details of an order.

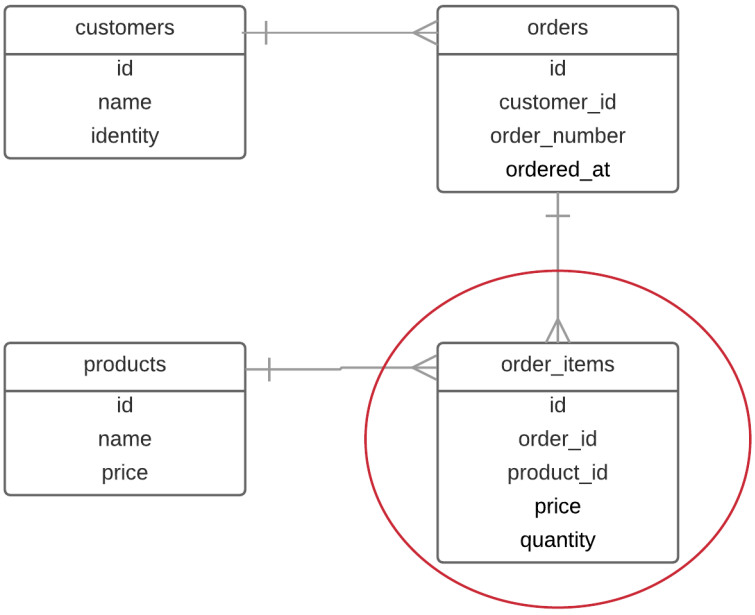


Table order_items with Order Details

For that table and in order to enforce the referential integrity between the tables, we will need to create 2 foreign keys. One foreign key that would relate table `order_items` to table `orders`. A second foreign key that would relate table `order_items` to table `products`.

But first, let's create the table. We will add the foreign key constraints later on, with `alter table` commands.

```
mysql> create table order_items (id int(11) not null auto_increment, order_id int\
(11) not null, product_id int(11) not null, price numeric(15, 2) not null, quanti\
ty int(11), primary key(id));
Query OK, 0 rows affected (0.02 sec)
```

(the above code snippet online)

The command returned `Query OK`. Let's keep some notes about the above command:

(1) It has an `id`, which is the primary key of the table, it is mandatory (`not null`) and it is automatically assigned, incrementally (`auto_increment`).

(2) The columns that are going to be used for referential integrity between table `order_items` and tables `orders` and `products` are `order_id` and `product_id` respectively. The type for both of these columns is `int(11)` and matches the type of `id` column of the corresponding table. I.e. The `id` on `orders` and the `id` on `products` they both have type `int(11)`. This is a pre-requisite in order to later build the foreign key constraints.

(3) Note that both `order_id` and `product_id` are mandatory (`not null`) because we cannot have an `order` entry without referencing an `order` and a `product`.

(4) `price` is mandatory.

(5) `quantity` is optional. But it may have been mandatory to improve the constraints on the design. For the time being, an `order` without a `quantity` will assume that quantity is `1`.

Let's get the list of the tables:

```
mysql> show tables;
+------------------------+
| Tables_in_customers_db |
+------------------------+
| customers              |
| order_items            |
| orders                 |
| products               |
+------------------------+
4 rows in set (0.01 sec)
```

(the above code snippet online)

You can see that `order_items` table is part of the list returned.

And let's get the structure of the new `order_items` table:

```
mysql> show create table order_items;
+-------------+-----------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-----------------------------------------------------------+
| Table       | Create Table                                                           \
                                                                                       \
                                                                                       \
                                                           |
+-------------+-----------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-----------------------------------------------------------+
| order_items | CREATE TABLE `order_items` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `order_id` int(11) NOT NULL,
  `product_id` int(11) NOT NULL,
  `price` decimal(15,2) NOT NULL,
  `quantity` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+-------------+-----------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-----------------------------------------------------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

There is nothing new here for you.

Referential Integrity between `orders` and `order_items`

Let's first connect the table `order_items` with the table `orders`. You must be in position to understand and execute the next order:

```
mysql> alter table order_items add constraint order_items_order_fk foreign key(or\
der_id) references orders(id);
Query OK, 0 rows affected (0.03 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

(the above code snippet online)

We got `Query OK`. This creates the foreign key constraint `order_items_order_fk` and the index on the column `order_id`. The index will also have the same name.

Let's verify the new structure of the table:

```
mysql> show create table order_items;
+-------------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
---------------------------+
| Table       | Create Table                                                             \
                                                                                       \
                                                                                       \
                                                                                       \
                                                                                       \
                           |
+-------------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
---------------------------+
| order_items | CREATE TABLE `order_items` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `order_id` int(11) NOT NULL,
  `product_id` int(11) NOT NULL,
  `price` decimal(15,2) NOT NULL,
  `quantity` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `order_items_order_fk` (`order_id`),
  CONSTRAINT `order_items_order_fk` FOREIGN KEY (`order_id`) REFERENCES `orders` \
(`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+-------------+-------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
-------------------------------------------------------------------------------------\
---------------------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

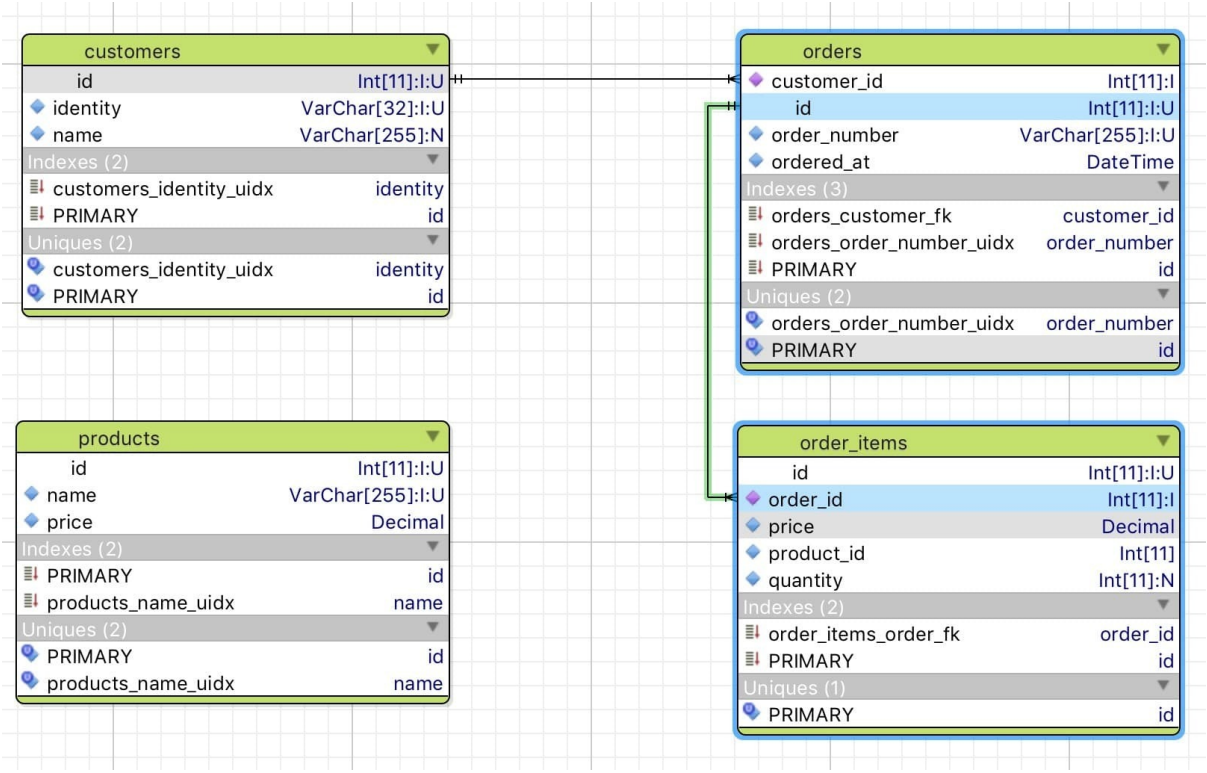
As you can see, we have the 2 new definitions:

```
KEY `order_items_order_fk` (`order_id`),
CONSTRAINT `order_items_order_fk` FOREIGN KEY (`order_id`) REFERENCES `orders` (`\
id`)
```

(the above code snippet online)

The first is for the new index, due to the fact that we have created the foreign key constraint. The second is for the foreign key constraint itself.

If you have downloaded the Valentina Studio or if you have other tool to automatically create the ERDs of a database schema, you will now see the relationship appearing between the `order_items` and the `orders` tables:



New Relationship Between order_items and orders

This relationship secures the referential integrity between the `order_items` and `orders`.

Referential Integrity between `products` and `order_items`

Let's now connect the table `order_items` with the table `products`. You must be in position to understand and execute the next order:

```
1 mysql> alter table order_items add constraint order_items_product_fk foreign key(\
2 product_id) references products(id);
3 Query OK, 0 rows affected (0.03 sec)
4 Records: 0  Duplicates: 0  Warnings: 0
```

(the above code snippet online)

We got `Query OK`. This creates the foreign key constraint `order_items_product_fk` and the index on the column `product_id`. The index will also have the same name.

Let's verify the new structure of the table:

```
mysql> show create table order_items;
+-------------+-------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
-------+
| Table       | Create Table                                                                  \
                                                                                             \
                                                                                             \
                                                                                             \
                                                                                             \
                                                                                             \
                                                                                             \
       |
+-------------+-------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
-------+
| order_items | CREATE TABLE `order_items` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `order_id` int(11) NOT NULL,
  `product_id` int(11) NOT NULL,
  `price` decimal(15,2) NOT NULL,
  `quantity` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `order_items_order_fk` (`order_id`),
  KEY `order_items_product_fk` (`product_id`),
  CONSTRAINT `order_items_order_fk` FOREIGN KEY (`order_id`) REFERENCES `orders` \
(`id`),
  CONSTRAINT `order_items_product_fk` FOREIGN KEY (`product_id`) REFERENCES `prod\
ucts` (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+-------------+-------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
--------------------------------------------------------------------------------------------\
```

```
47 -------+
48 1 row in set (0.00 sec)
```

(the above code snippet online)

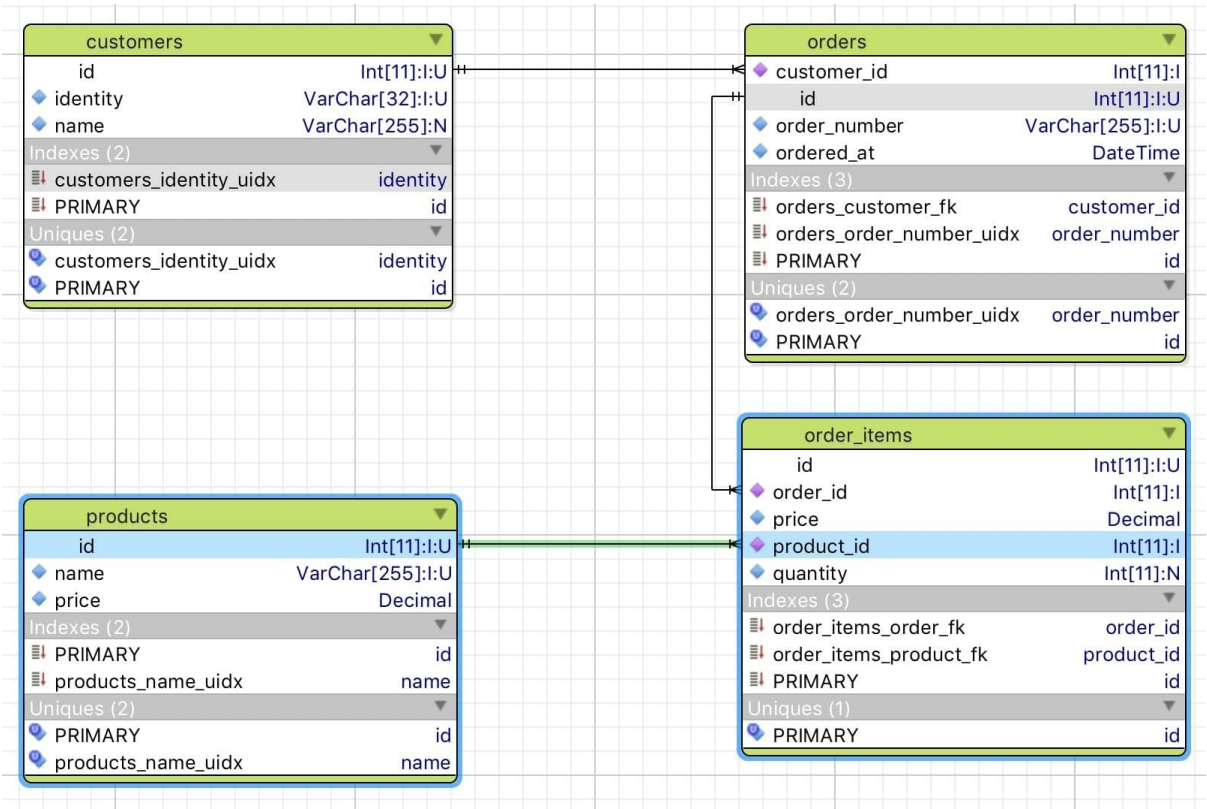
As you can see, we have the 2 new definitions:

```
1 KEY `order_items_product_fk` (`product_id`),
2 CONSTRAINT `order_items_product_fk` FOREIGN KEY (`product_id`) REFERENCES `produc\
3 ts` (`id`)
```

(the above code snippet online)

The first is for the new index, due to the fact that we have created the foreign key constraint. The second is for the foreign key constraint itself.

Valentina Studio diagram will show again the ERD and the new relationship between `order_items` and `products`:



New Relationship Between order_items and products

This relationship secures the referential integrity between the `order_items` and `products`.

Inserting Data Into The Details

Let's see the data that we have in our orders:

```
mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  1 | ABC001       | 2016-09-09 08:34:55 |           1 |
+----+--------------+---------------------+-------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

Let's also see our products:

```
mysql> select * from products;
+----+-----------------------------+-------+
| id | name                        | price |
+----+-----------------------------+-------+
|  1 | Game of Thrones - S01 - DVD | 50.00 |
|  2 | Of Mice and Men             | 19.80 |
|  3 | A Nice Story                |  5.00 |
+----+-----------------------------+-------+
3 rows in set (0.00 sec)
```

(the above code snippet online)

Let's now insert the details of the first order, the one with `id` `1`. We will sell 2 DVDs of the `Game of Thrones - S01 - DVD` product:

```
mysql>  insert into order_items (order_id, product_id, price, quantity) values (1\
, 1, 50.00, 2);
Query OK, 1 row affected (0.00 sec)
```

(the above code snippet online)

We get back `Query OK`. The command is not new to you. You only have to make sure that you give the correct reference values. So, for the `order_id` we give the value `1`, because we want the new details entry (the new entry in `order_items`) to be referring to the 1st order that has `id` `1`. We also set the `product_id` to have the value `1`, because it is the product id `1` that corresponds to the product `Game of Thrones - S01 - DVD`.

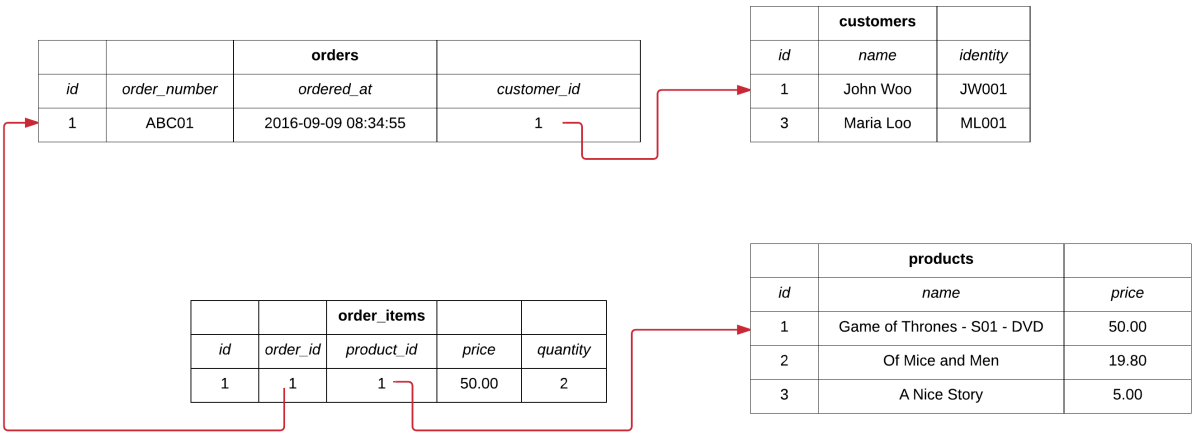
Let's now see the contents of the `order_items` table:

```
mysql> select * from order_items;
+----+----------+------------+-------+----------+
| id | order_id | product_id | price | quantity |
+----+----------+------------+-------+----------+
|  1 |        1 |          1 | 50.00 |        2 |
+----+----------+------------+-------+----------+
1 row in set (0.00 sec)
```

(the above code snippet online)

The new order item has been created successfully.

So, basically, we have created the following data relationships in our database.



Data Relationships

Unique Index On Multiple Columns

There is something left to be implemented with regards to our primitive customer relationship management system. This is the constraint:

The same product cannot be added to the same order more than once.

This business rule assumes that if one wanted to add the same product twice in the same order, then he would have to set the quantity to 2.

How can we ask the help of MySQL to prevent user from inserting the same product more than once in the same order?

This is very simple. Since all that we want is any combination between `order_id` and `product_id` to be unique, we only have to create a unique index that encompasses both columns. Let's do that:

```
mysql> create unique index order_items_order_product_uidx on order_items(order_id\
, product_id);
Query OK, 0 rows affected (0.04 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

(the above code snippet online)

We got back `Query OK`. Watch out how we build the name of the index. It is a pattern that we follow, although we could name it anything that we liked. Then watch out the `on order_items(order_id, product_id)` specification. This is the trick here. Our new index now combines 2 columns. So, the combined value of them needs to be unique. Not either of them. Both of them.

Let's see the structure of our table:

```
mysql> show create table order_items;
+-------------+-------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-----------------------------------------------------+
| Table       | Create Table                                                      \
                                                                                \
                                                                                \
                                                                                \
                                                                                \
                                                                                \
                                                                                \
                                                    |
+-------------+-------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-------------------------------------------------------------------------------\
-----------------------------------------------------+
| order_items | CREATE TABLE `order_items` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `order_id` int(11) NOT NULL,
  `product_id` int(11) NOT NULL,
  `price` decimal(15,2) NOT NULL,
  `quantity` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `order_items_order_product_uidx` (`order_id`,`product_id`),
```

```
  KEY `order_items_product_fk` (`product_id`),
  CONSTRAINT `order_items_order_fk` FOREIGN KEY (`order_id`) REFERENCES `orders` \
(`id`),
  CONSTRAINT `order_items_product_fk` FOREIGN KEY (`product_id`) REFERENCES `prod\
ucts` (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci |
+-------------+-----------------------------------------------------------------\
-----------------------------------------------------------------------------\
-----------------------------------------------------------------------------\
-----------------------------------------------------------------------------\
-----------------------------------------------------------------------------\
-----------------------------------------------------------------------------\
-----------------------------------------------------------------------------\
-----------------------------------------------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

As you can see, there is a new `KEY` specification, and in fact, a `UNIQUE KEY` one. This combines the columns `order_id` and `product_id`:

```
UNIQUE KEY `order_items_order_product_uidx` (`order_id`,`product_id`)
```

(the above code snippet online)

Let's see how we can verify that our unique index works as expected. Let's try to insert another order item for the same order and same product.

First list the current order items:

```
mysql> select * from order_items;
+----+----------+------------+-------+----------+
| id | order_id | product_id | price | quantity |
+----+----------+------------+-------+----------+
|  1 |        1 |          1 | 50.00 |        2 |
+----+----------+------------+-------+----------+
1 row in set (0.00 sec)
```

(the above code snippet online)

The next command will try to insert a new order item for the **same order** and the **same product**. We expect this to fail.

```
mysql> insert into order_items (order_id, product_id, price, quantity) values (1,\
 1, 50.00, 3);
ERROR 1062 (23000): Duplicate entry '1-1' for key 'order_items_order_product_uidx'
```

(the above code snippet online)

and it does. The error mentions that the combination `1-1` violates the unique index `order_items_order_product_uidx`.

Perfect! The unique index does not allow us to insert an order item with same order and same product.

On the other hand, let's try to insert an order item for same product but for different order. Let's see the orders first:

```
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  1 | ABC001       | 2016-09-09 08:34:55 |           1 |
+----+--------------+---------------------+-------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

Let's create another order:

```
mysql> insert into orders (order_number, ordered_at, customer_id) values ('ABC002\
', current_timestamp, 1);
Query OK, 1 row affected (0.01 sec)

mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  1 | ABC001       | 2016-09-09 08:34:55 |           1 |
|  6 | ABC002       | 2016-09-10 16:10:43 |           1 |
+----+--------------+---------------------+-------------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

Now we have 2 orders for the same customer, but they have different order number and ordered date and time stamp.

If we try to insert a new order item for this new order but for the old product with id `1`, this will not fail:

```
mysql> insert into order_items (order_id, product_id, price, quantity) values (6,\
 1, 50.00, 5);
Query OK, 1 row affected (0.00 sec)
```

(the above code snippet online)

Nice! The unique index, although it does not let us add the same product to the same order, it does let us add the same product to different order.

```
mysql> select * from order_items;
+----+----------+------------+-------+----------+
| id | order_id | product_id | price | quantity |
+----+----------+------------+-------+----------+
|  1 |        1 |          1 | 50.00 |        2 |
|  3 |        6 |          1 | 50.00 |        5 |
+----+----------+------------+-------+----------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

Closing Note

Video / Screencast with Content of Chapter

We would like to close this chapter by giving you a link to a video / screencast with the chapter content. You may want to watch it as an alternative source of learning.

Foreign Keys

Foreign Keys <a href="https://player.vimeo.com/video/194

Valentina Studio

This video shows how one can create a diagram of the tables and the relationships using Valentina Studio.

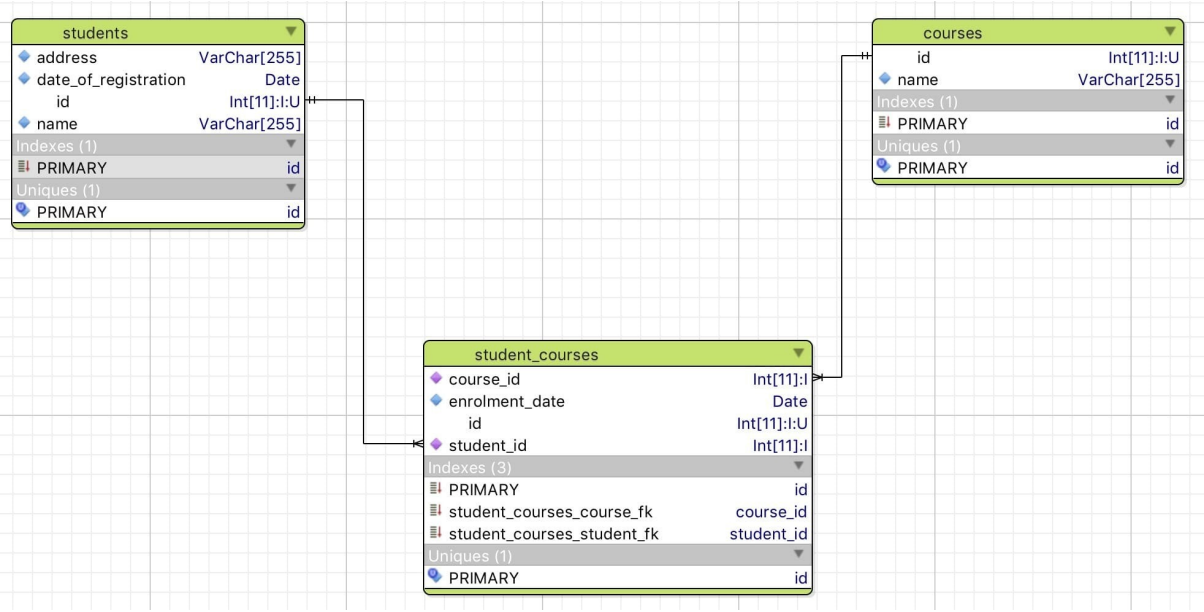
Chapter Video / Screencast - Create ERD using Valentina Studio

Tasks and Quizzes

> **Before you continue, you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Task Details

Create a students database following the next entity relationship diagram.



Students Database ERD

Some things that may not be obvious from the diagram are:

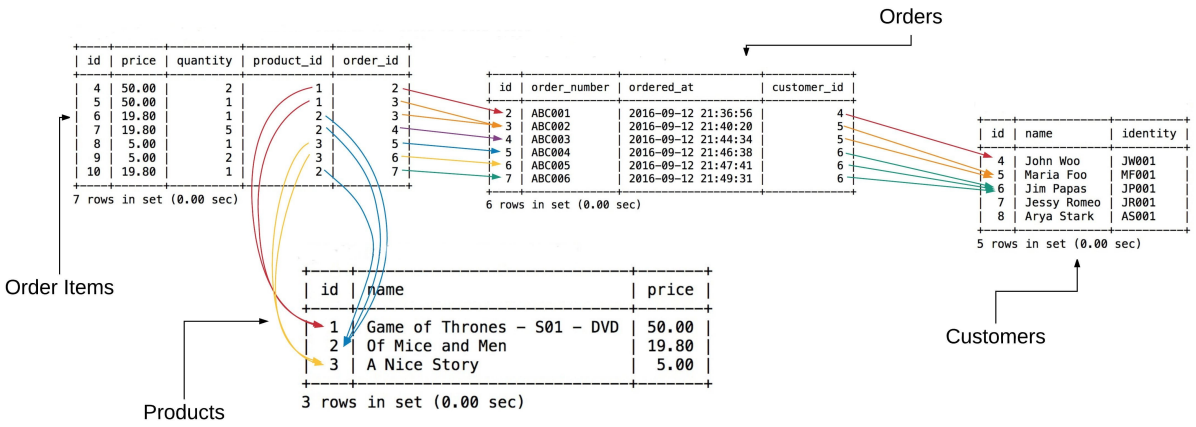
1. All tables `id` columns are mandatory and primary keys that take values automatically using increments.
2. Table `students`:
    1. `name` should be mandatory with maximum 255 characters. It should also be unique.
    2. `address` should be mandatory with maximum 255 characters.
    3. `date_of_registration` should be mandatory. It should be of type `date` and not `datetime`.
3. Table `courses`:
    1. `name` should be mandatory with maximum 255 characters. It should also be unique.
4. Table `student_courses`:
    1. `student_id` should be mandatory.
    2. `course_id` should be mandatory.
    3. `enrolment_date` should be mandatory. It should be of type `date` and not `datetime`.
    4. The same student should not be allowed to enrol on the same course more than once.

Important: Your code (all SQL statements) needs to be uploaded into a new project on your Github account. Invite your mentor to review your code online.

7 - Joins, Left Joins And Subqueries

Summary

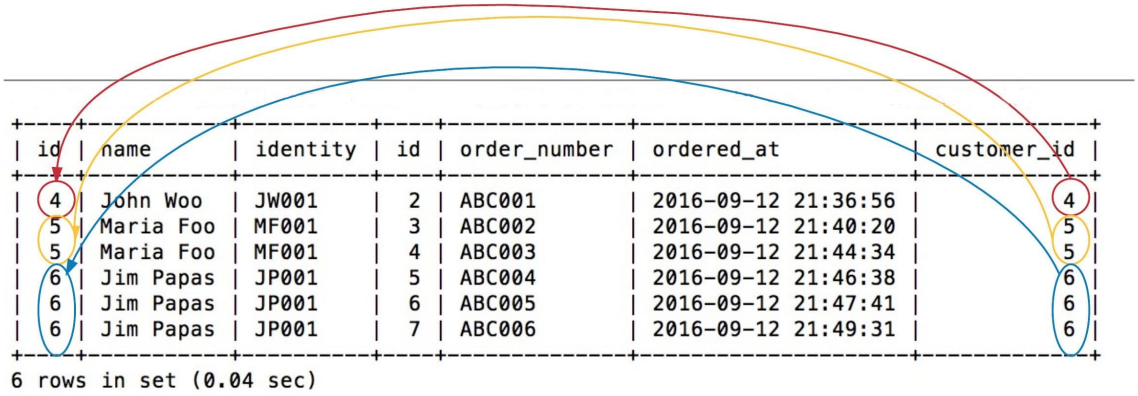
In previous chapters we have designed our primitive customer relationship management system. In this chapter we are setting up our data as follows:



Set up of our Data

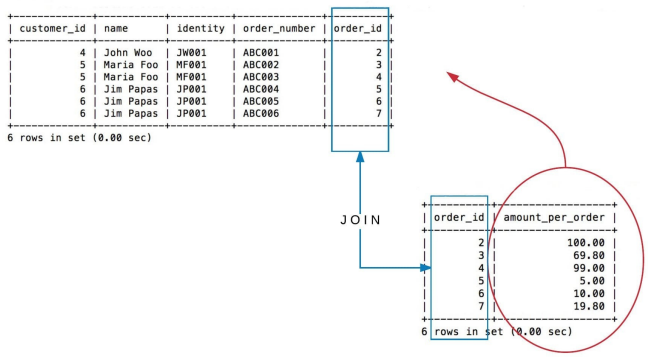
and we are trying to query and get useful information out of it.

We are going to learn how to combine data from two or more different tables:



Combining Data From Different Tables

Combining of information will not be restricted to information coming from tables only, but it will be done using sub-queries too:



Combining Query Results

We will also learn about the aggregate function `sum()` and how it can be used to get useful insights into out data.

Aggregate - Sum

Learning Goals

1. Learn about the `join` clause and how you can combine the information that exists in more than one table.
2. Learn how you can attach labels to column headers and make those headers easier to read.
3. Learn how to select to display specific columns from a wide, many-columns result set.
4. Learn to count distinct values in order to get accurate count results.
5. Learn about the `left join` and how you can combine information from two tables even if rows on first are not referenced by rows on the second table.
6. Understand the concept of *no value* and how it is represented in SQL.
7. Learn how to limit the result set to the columns of 1 table, even if SQL query joins many tables.

8. Learn how to display a dynamically-evaluated column (a derived-value column).
9. Learn about the `sum()` aggregate function and the `group by` clause.
10. Learn how to join information from 2 different queries.

Introduction

Let's start by opening a terminal window and connecting to MySQL server using `mysql`.

```
$ mysql -u root
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.7.14 MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

(the above code snippet online)

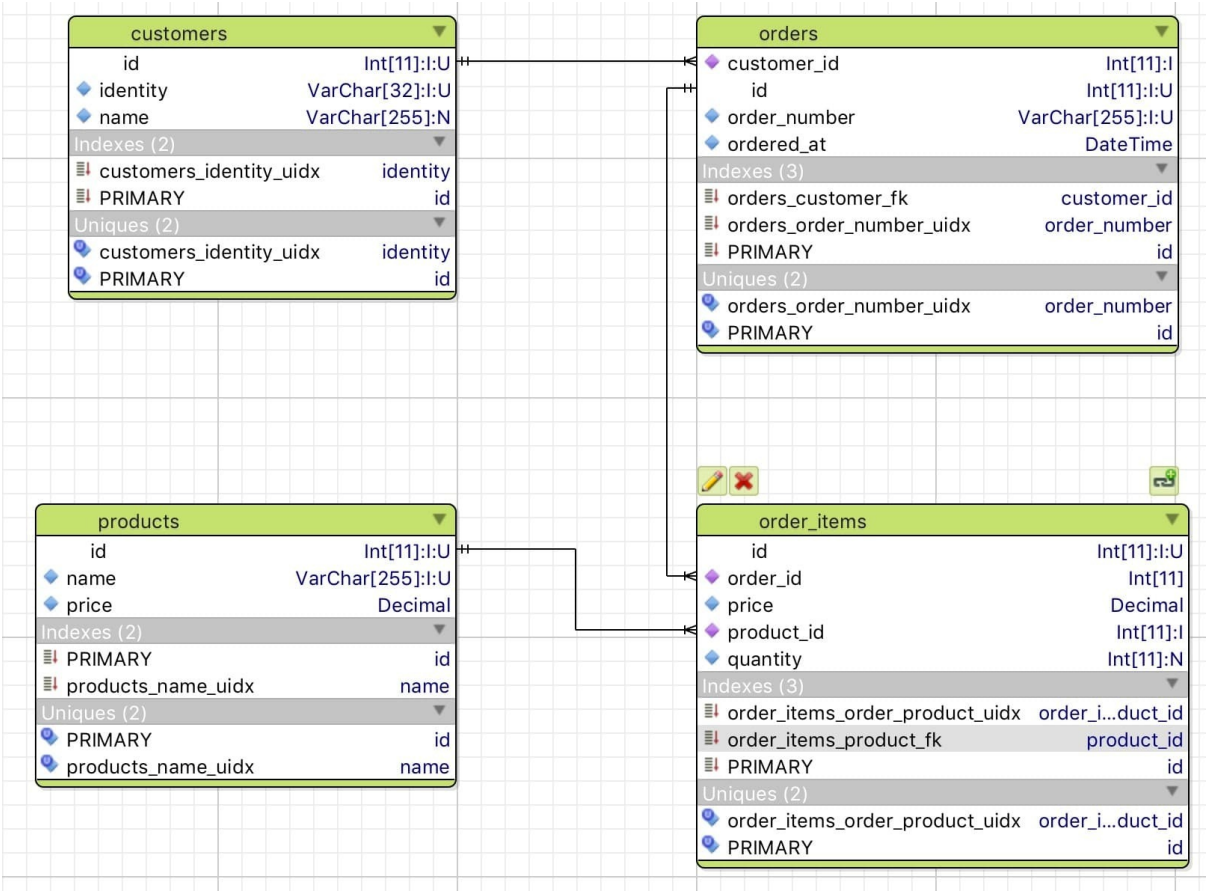
Let's switch to our database and list the tables that it contains:

```
mysql> use customers_db;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+------------------------+
| Tables_in_customers_db |
+------------------------+
| customers              |
| order_items            |
| orders                 |
| products               |
+------------------------+
4 rows in set (0.00 sec)
```

(the above code snippet online)

Review Database Schema

Let's review again the database schema of our `customers_db` database.



customers_db Database Schema

(1) `customers` table is linked to `orders` table. It is a 1-to-many relationship. The `customer_id` can be found inside the `orders` table. It is a foreign key to `customers id` column. Hence, 1 customer can have many orders.

(2) `orders` table has 1-to-many relationship to `order_items`. The `order_items` are the details of an order.

(3) `order_items` table holds the foreign key (column `order_id`) to `orders id` column. Hence, for every order item, we know which order it belongs to. Also, the order item holds the foreign key (`product_id`) to the `products id` column. In other words, an order item always references a specific product.

Retrieving Data

We are going to carry out lots of hands-on exercises on retrieving data stored inside our customers database. Doing that we will learn for `joins` and `sub-queries`.

Deleting Existing Data

Before we proceed, let's delete the existing data from our database. We will then insert data that will be useful for us to learn the new concepts.

The following commands will delete all the data except from the products. We will leave the products as they are:

```
mysql> delete from order_items;
Query OK, 2 rows affected (0.01 sec)

mysql> delete from orders;
Query OK, 2 rows affected (0.00 sec)

mysql> delete from customers;
Query OK, 2 rows affected (0.00 sec)
```

(the above code snippet online)

All the above are `delete` statements that they do not have any where clause. Hence, they delete all the data from the tables that they are called on.

We are only left with the products data:

```
mysql> select * from products;
+----+------------------------------+-------+
| id | name                         | price |
+----+------------------------------+-------+
|  1 | Game of Thrones - S01 - DVD | 50.00 |
|  2 | Of Mice and Men              | 19.80 |
|  3 | A Nice Story                 |  5.00 |
+----+------------------------------+-------+
3 rows in set (0.01 sec)
```

(the above code snippet online)

Inserting Necessary Data

We are now going to execute the following commands to create the necessary data to support our query exercises:

First the customers:

```
mysql> insert into customers (name, identity) values ('John Woo', 'JW001');
Query OK, 1 row affected (0.00 sec)

mysql> insert into customers (name, identity) values ('Maria Foo', 'MF001');
Query OK, 1 row affected (0.00 sec)

mysql> insert into customers (name, identity) values ('Jim Papas', 'JP001');
Query OK, 1 row affected (0.01 sec)

mysql> insert into customers (name, identity) values ('Jessy Romeo', 'JR001');
Query OK, 1 row affected (0.01 sec)

mysql> insert into customers (name, identity) values ('Arya Stark', 'AS001');
Query OK, 1 row affected (0.01 sec)
```

(the above code snippet online)

If we select the customers, we should see this:

```
mysql> select * from customers;
+----+-------------+----------+
| id | name        | identity |
+----+-------------+----------+
|  4 | John Woo    | JW001    |
|  5 | Maria Foo   | MF001    |
|  6 | Jim Papas   | JP001    |
|  7 | Jessy Romeo | JR001    |
|  8 | Arya Stark  | AS001    |
+----+-------------+----------+
5 rows in set (0.00 sec)
```

(the above code snippet online)

Then create orders and order items:

Insert 1 order, with 1 order item, for the 1st customer:

```
mysql> insert into orders (order_number, ordered_at, customer_id) values ('ABC001\
', current_timestamp, 4);
Query OK
mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  2 | ABC001       | 2016-09-12 21:36:56 |           4 |
+----+--------------+---------------------+-------------+
1 row in set (0.02 sec)
mysql> insert into order_items (order_id, product_id, price, quantity) values (2,\
```

```
 1, 50.00, 2);
Query OK
```

(the above code snippet online)

Insert 1 order, with 2 order items, for the 2nd customer:

```
mysql> insert into orders (order_number, ordered_at, customer_id) values ('ABC002\
', current_timestamp, 5);
Query OK

mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  2 | ABC001       | 2016-09-12 21:36:56 |           4 |
|  3 | ABC002       | 2016-09-12 21:40:20 |           5 |
+----+--------------+---------------------+-------------+
2 rows in set (0.02 sec)

mysql> insert into order_items (order_id, product_id, price, quantity) values (3,\
 1, 50.00, 1);
Query OK, 1 row affected (0.00 sec)

mysql> insert into order_items (order_id, product_id, price, quantity) values (3,\
 2, 19.80, 1);
Query OK, 1 row affected (0.00 sec)
```

(the above code snippet online)

We then create 1 more order, for the 2nd customers. Only 1 order item:

```
mysql> insert into orders (order_number, ordered_at, customer_id) values ('ABC003\
', current_timestamp, 5);
Query OK

mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  2 | ABC001       | 2016-09-12 21:36:56 |           4 |
|  3 | ABC002       | 2016-09-12 21:40:20 |           5 |
|  4 | ABC003       | 2016-09-12 21:44:34 |           5 |
+----+--------------+---------------------+-------------+
3 rows in set (0.02 sec)

mysql> insert into order_items (order_id, product_id, price, quantity) values (4,\
```

```
 2, 19.80, 5);
Query OK, 1 row affected (0.00 sec)
```

(the above code snippet online)

Let's go to the 3rd customer, 3 orders with 1 order item each:

The 3rd customer has id 6

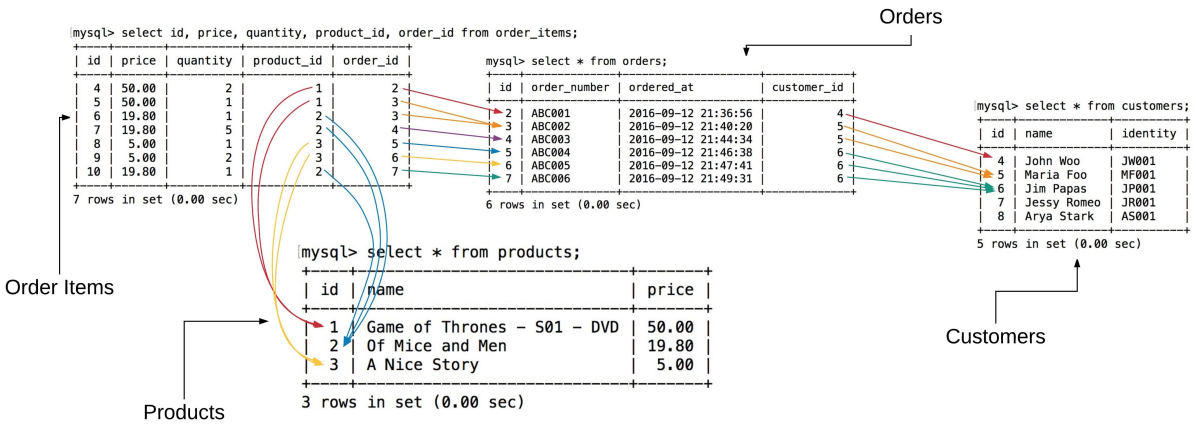
```
mysql> insert into orders (order_number, ordered_at, customer_id) values ('ABC004\
', current_timestamp, 6);
Query OK

mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  2 | ABC001       | 2016-09-12 21:36:56 |           4 |
|  3 | ABC002       | 2016-09-12 21:40:20 |           5 |
|  4 | ABC003       | 2016-09-12 21:44:34 |           5 |
|  5 | ABC004       | 2016-09-12 21:46:38 |           6 |
+----+--------------+---------------------+-------------+
4 rows in set (0.02 sec)

mysql> insert into order_items (order_id, product_id, price, quantity) values (5,\
 3, 5.00, 1);
Query OK

mysql> insert into orders (order_number, ordered_at, customer_id) values ('ABC005\
', current_timestamp, 6);
Query OK

mysql> select * from orders;
+----+--------------+---------------------+-------------+
| id | order_number | ordered_at          | customer_id |
+----+--------------+---------------------+-------------+
|  2 | ABC001       | 2016-09-12 21:36:56 |           4 |
|  3 | ABC002       | 2016-09-12 21:40:20 |           5 |
|  4 | ABC003       | 2016-09-12 21:44:34 |           5 |
|  5 | ABC004       | 2016-09-12 21:46:38 |           6 |
|  6 | ABC005       | 2016-09-12 21:47:41 |           6 |
+----+--------------+---------------------+-------------+
5 rows in set (0.02 sec)

mysql> insert into order_items (order_id, product_id, price, quantity) values (6,\
 3, 5.00, 2);
Query OK
```

```
39
40 mysql> insert into orders (order_number, ordered_at, customer_id) values ('ABC006\
41 ', current_timestamp, 6);
42 Query OK
43
44 mysql> select * from orders;
45 +----+--------------+---------------------+-------------+
46 | id | order_number | ordered_at          | customer_id |
47 +----+--------------+---------------------+-------------+
48 |  2 | ABC001       | 2016-09-12 21:36:56 |           4 |
49 |  3 | ABC002       | 2016-09-12 21:40:20 |           5 |
50 |  4 | ABC003       | 2016-09-12 21:44:34 |           5 |
51 |  5 | ABC004       | 2016-09-12 21:46:38 |           6 |
52 |  6 | ABC005       | 2016-09-12 21:47:41 |           6 |
53 |  7 | ABC006       | 2016-09-12 21:49:31 |           6 |
54 +----+--------------+---------------------+-------------+
55 6 rows in set (0.02 sec)
56
57 mysql> insert into order_items (order_id, product_id, price, quantity) values (7,\
58  2, 19.80, 1);
59 Query OK
```

(the above code snippet online)

We will not enter any more data.

In the following picture you can see the relationship of the data that we currently have in our database. We will use these data to carry out simple to complex queries:



customers_db Data

As you can quickly see from the picture:

1. Only 3 customers have placed any order.
2. The 1st customer has placed 1 order, the 2nd customer has placed 2 orders and the 3rd customer has placed 3 orders.
3. Order with `id` equal to 3 has 2 order items. All the other orders have 1 order item each.

Selecting All the Information - `join` or `inner join`

We are going to write a select that will display the information from all the tables combined. So, in one result set, we will be able to see the the customers, with their corresponding orders, with their corresponding order details and corresponding products.

The `select` command that we will use to achieve that is not complex, but it is long, only because we want to combine the information from many tables together. Combining information from two tables is done with the `join` technique, which relies on the existence of the foreign keys between the tables.

Joining `customers` with `orders`

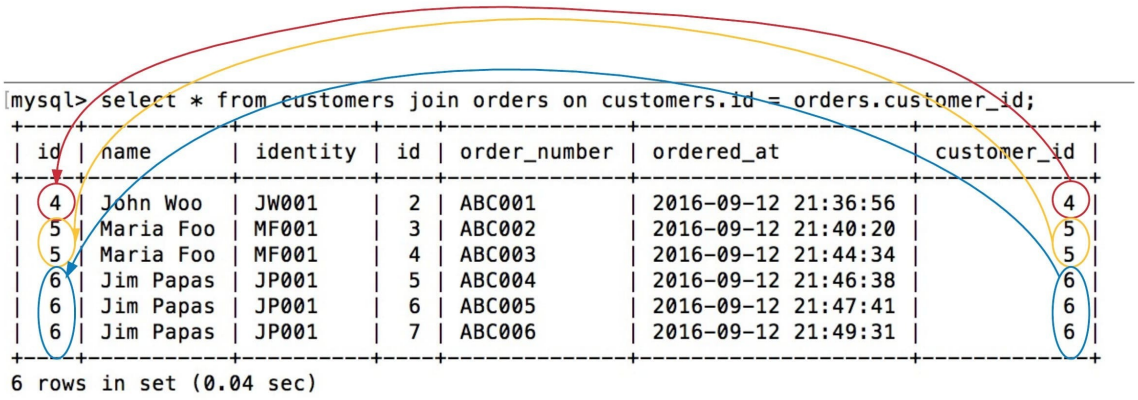
If we want to combine the information between `customers` and `orders` we start with `select * from customers join orders`. Telling MySQL that we want to combine the information from `customers` table with the information from `orders` table. But this is not enough. We need to tell MySQL which columns we want to use as common / referencing information. Note that we can use the `join` construct to combine information from two tables without using the foreign key relationship that they may have. But this is uncommon. We usually specify that the foreign key relationship is the one that should be used for the joining. Hence, `select * from customers join orders on customers.id = orders.customer_id` is the correct statement to join the information from `customers` table to `orders` table.

Let's execute this statement:

```
mysql> select * from customers join orders on customers.id = orders.customer_id;
+----+-----------+----------+----+--------------+---------------------+----------\
---+
| id | name      | identity | id | order_number | ordered_at          | customer_\
id |
+----+-----------+----------+----+--------------+---------------------+----------\
---+
|  4 | John Woo  | JW001    |  2 | ABC001       | 2016-09-12 21:36:56 |          \
 4 |
|  5 | Maria Foo | MF001    |  3 | ABC002       | 2016-09-12 21:40:20 |          \
 5 |
|  5 | Maria Foo | MF001    |  4 | ABC003       | 2016-09-12 21:44:34 |          \
 5 |
|  6 | Jim Papas | JP001    |  5 | ABC004       | 2016-09-12 21:46:38 |          \
 6 |
|  6 | Jim Papas | JP001    |  6 | ABC005       | 2016-09-12 21:47:41 |          \
 6 |
|  6 | Jim Papas | JP001    |  7 | ABC006       | 2016-09-12 21:49:31 |          \
 6 |
+----+-----------+----------+----+--------------+---------------------+----------\
---+
6 rows in set (0.04 sec)
```

(the above code snippet online)

As you can see, we now have the `customers` and the `orders` table rows combined. Each row from the `customers` table is being matched to a row from `orders` table using the combination logic `customers.id = orders.customer_id`. In other words, a given `customers` row is matched to the `orders` row only if the column `customer_id` on the `orders` table has the same value as the `id` column of the given row in `customers` table.



```
mysql> select * from customers join orders on customers.id = orders.customer_id;
+----+----------+----------+----+--------------+---------------------+-------------+
| id | name     | identity | id | order_number | ordered_at          | customer_id |
+----+----------+----------+----+--------------+---------------------+-------------+
|  4 | John Woo | JW001    |  2 | ABC001       | 2016-09-12 21:36:56 |           4 |
|  5 | Maria Foo | MF001    |  3 | ABC002       | 2016-09-12 21:40:20 |           5 |
|  5 | Maria Foo | MF001    |  4 | ABC003       | 2016-09-12 21:44:34 |           5 |
|  6 | Jim Papas | JP001    |  5 | ABC004       | 2016-09-12 21:46:38 |           6 |
|  6 | Jim Papas | JP001    |  6 | ABC005       | 2016-09-12 21:47:41 |           6 |
|  6 | Jim Papas | JP001    |  7 | ABC006       | 2016-09-12 21:49:31 |           6 |
+----+----------+----------+----+--------------+---------------------+-------------+
6 rows in set (0.04 sec)
```

Customers Combined With Orders

The above picture shows how the values of the `id` column of the `customers` rows have been combined with the values of the `customer_id` column of the `orders` rows.

The output that we got from the above command repeats every customer information for each one of the orders that the customer has. That's why you see the `Maria Foo` details appearing twice and the `Jim Papas` 3 times. This is because `Maria Foo` has 2 orders and `Jim Papas` has 3 orders.

Also, you do not see the customers that they have not placed any order yet. `Jessy Romeo` and `Arya Stark` have not placed any orders yet, so, they do not appear in this result set. This is expected, because their ids, `7` and `8` respectively cannot be found in any `orders` row in the column `customer_id`.

Then join `order_items`

We can follow the same `join <join_with_table_name> on <join_from_table_name.column_name> = <join_with_table_name.column_name>` pattern to continue the previous `select` and bring more information in the same result set. The new table we will join with is `order_items`.

Let's try the following command:

```
mysql> select * from customers join orders on customers.id = orders.customer_id j\
oin order_items on orders.id = order_items.order_id;
+----+-----------+----------+----+--------------+---------------------+----------\
---+----+----------+------------+-------+----------+
| id | name      | identity | id | order_number | ordered_at          | customer_\
id | id | order_id | product_id | price | quantity |
+----+-----------+----------+----+--------------+---------------------+----------\
---+----+----------+------------+-------+----------+
|  4 | John Woo  | JW001    |  2 | ABC001       | 2016-09-12 21:36:56 |          \
 4 |  4 |        2 |          1 | 50.00 |        2 |
|  5 | Maria Foo | MF001    |  3 | ABC002       | 2016-09-12 21:40:20 |          \
 5 |  5 |        3 |          1 | 50.00 |        1 |
|  5 | Maria Foo | MF001    |  3 | ABC002       | 2016-09-12 21:40:20 |          \
 5 |  6 |        3 |          2 | 19.80 |        1 |
|  5 | Maria Foo | MF001    |  4 | ABC003       | 2016-09-12 21:44:34 |          \
 5 |  7 |        4 |          2 | 19.80 |        5 |
|  6 | Jim Papas | JP001    |  5 | ABC004       | 2016-09-12 21:46:38 |          \
 6 |  8 |        5 |          3 |  5.00 |        1 |
|  6 | Jim Papas | JP001    |  6 | ABC005       | 2016-09-12 21:47:41 |          \
 6 |  9 |        6 |          3 |  5.00 |        2 |
|  6 | Jim Papas | JP001    |  7 | ABC006       | 2016-09-12 21:49:31 |          \
 6 | 10 |        7 |          2 | 19.80 |        1 |
+----+-----------+----------+----+--------------+---------------------+----------\
---+----+----------+------------+-------+----------+
7 rows in set (0.01 sec)
```

(the above code snippet online)

The `join order_items on orders.id = order_items.order_id` is the new part that we have added at the end of the previous SQL statement.

Now the results brings back 7 rows. This is because we have 6 orders but the 2nd order, with id 3 has 2 order items. And this is the reason you now see 3 rows in the result set with `Maria Foo`.

The join takes place between the foreign key `order_items.order_id` and the primary key of the referenced table `orders.id`.

The `as` helper to rename columns

The result set has lots of columns and there is a small caveat, that there are columns with same name. Look at the result again, above. It has 3 columns `id`. But of course, these are not the same columns. The first `id` belongs to the first table referenced in the SQL statement, i.e. the `customers`. The 2nd `id` column belongs to `orders` table and the 3rd `id` column belongs to the `order_items`.

If you want, you can set a label for a column using the `as` reserved word. In that case, the column header will not be the column name itself, but it is going to be the label that you set after the `as` word. The disadvantage here is that you explicitly need to specify the columns that you want the result set to include. Which is not, of course, something bad, because usually, none wants all the columns present.

Let's try this:

```
mysql> select customers.id as customer_id, name, identity, orders.id as order_id,\
 order_number, ordered_at,
    > order_items.id as order_item_id, product_id, price, quantity from customer\
s join orders on customers.id = orders.customer_id join order_items on orders.id \
= order_items.order_id;
+-------------+-----------+----------+----------+--------------+----------------\
----+---------------+------------+-------+----------+
| customer_id | name      | identity | order_id | order_number | ordered_at      \
    | order_item_id | product_id | price | quantity |
+-------------+-----------+----------+----------+--------------+----------------\
----+---------------+------------+-------+----------+
|           4 | John Woo  | JW001    |        2 | ABC001       | 2016-09-12 21:36\
:56 |             4 |          1 | 50.00 |        2 |
|           5 | Maria Foo | MF001    |        3 | ABC002       | 2016-09-12 21:40\
:20 |             5 |          1 | 50.00 |        1 |
|           5 | Maria Foo | MF001    |        3 | ABC002       | 2016-09-12 21:40\
:20 |             6 |          2 | 19.80 |        1 |
|           5 | Maria Foo | MF001    |        4 | ABC003       | 2016-09-12 21:44\
:34 |             7 |          2 | 19.80 |        5 |
|           6 | Jim Papas | JP001    |        5 | ABC004       | 2016-09-12 21:46\
:38 |             8 |          3 |  5.00 |        1 |
|           6 | Jim Papas | JP001    |        6 | ABC005       | 2016-09-12 21:47\
:41 |             9 |          3 |  5.00 |        2 |
|           6 | Jim Papas | JP001    |        7 | ABC006       | 2016-09-12 21:49\
:31 |            10 |          2 | 19.80 |        1 |
+-------------+-----------+----------+----------+--------------+----------------\
----+---------------+------------+-------+----------+
7 rows in set (0.01 sec)
```

(the above code snippet online)

As you can see above, the select statement now explicitly mentions the columns that we want the result set to contain. And for the `id` columns we have prefixed each `id` reference with the table name, e.g. `customers.id` and `orders.id`. Otherwise, select wouldn't know which `id` to select for the result set. Also, we have used the `as` keyword to set labels to the columns to make sure we understand which table each `id` refers to.

This is a nice technique to get easier to read results. But, let's continue by bringing information from products too, inside the same result set.

Then join `products`

Let's join `products`. This is going to be done via the foreign key `order_items.product_id` and the primary key on products, the `products.id`.

Try the following command:

```
mysql> select * from customers
    ->   join orders on customers.id = orders.customer_id
    ->   join order_items on orders.id = order_items.order_id
    ->   join products on products.id = order_items.product_id;
+----+-----------+----------+----+--------------+---------------------+----------\
---+----+----------+-----------+-------+----------+----+----------------------\
-----+-------+
| id | name      | identity | id | order_number | ordered_at          | customer_\
id | id | order_id | product_id | price | quantity | id | name                 \
     | price |
+----+-----------+----------+----+--------------+---------------------+----------\
---+----+----------+-----------+-------+----------+----+----------------------\
-----+-------+
|  4 | John Woo  | JW001    |  2 | ABC001       | 2016-09-12 21:36:56 |          \
 4 |  4 |        2 |          1 | 50.00 |        2 |  1 | Game of Thrones - S01 -\
 DVD | 50.00 |
|  5 | Maria Foo | MF001    |  3 | ABC002       | 2016-09-12 21:40:20 |          \
 5 |  5 |        3 |          1 | 50.00 |        1 |  1 | Game of Thrones - S01 -\
 DVD | 50.00 |
|  5 | Maria Foo | MF001    |  3 | ABC002       | 2016-09-12 21:40:20 |          \
 5 |  6 |        3 |          2 | 19.80 |        1 |  2 | Of Mice and Men       \
     | 19.80 |
|  5 | Maria Foo | MF001    |  4 | ABC003       | 2016-09-12 21:44:34 |          \
 5 |  7 |        4 |          2 | 19.80 |        5 |  2 | Of Mice and Men       \
     | 19.80 |
|  6 | Jim Papas | JP001    |  5 | ABC004       | 2016-09-12 21:46:38 |          \
 6 |  8 |        5 |          3 |  5.00 |        1 |  3 | A Nice Story          \
     |  5.00 |
|  6 | Jim Papas | JP001    |  6 | ABC005       | 2016-09-12 21:47:41 |          \
 6 |  9 |        6 |          3 |  5.00 |        2 |  3 | A Nice Story          \
     |  5.00 |
|  6 | Jim Papas | JP001    |  7 | ABC006       | 2016-09-12 21:49:31 |          \
 6 | 10 |        7 |          2 | 19.80 |        1 |  2 | Of Mice and Men       \
     | 19.80 |
+----+-----------+----------+----+--------------+---------------------+----------\
---+----+----------+-----------+-------+----------+----+----------------------\
-----+-------+
7 rows in set (0.01 sec)
```

(the above code snippet online)

> *Hint:* When you type in a long sql command, you can hit on <kbd>Enter</kbd> to break it in multiple lines in order for it to be easier to type and read. The command is not sent to the server unless you type the last ; and hit <kbd>Enter</kbd> exactly after that.

Nice, we now have all the information displayed in one table-result set. The join with the `product`

table didn't increase the number of rows in the result set, because each order items can reference only 1 product.

The above result set still has the problem that it displays all the columns. But they are grouped in the order they are referenced and joined in the select statement.

More Complex Queries

Let's start answering some more complex queries.

How many customers do I have with orders?

Suppose that we had a lot of customers with lots of orders. But some of the customers are not really customers, they are leads, just because they have never placed any order. So, we have customers that they have orders and customers that they don't have any order.

How can we count the customers that they have orders?

We have tried the following query earlier:

```
mysql> select * from customers join orders on orders.customer_id = customers.id;
+----+-----------+----------+----+--------------+---------------------+----------\
---+
| id | name      | identity | id | order_number | ordered_at          | customer_\
id |
+----+-----------+----------+----+--------------+---------------------+----------\
---+
|  4 | John Woo  | JW001    |  2 | ABC001       | 2016-09-12 21:36:56 |          \
 4 |
|  5 | Maria Foo | MF001    |  3 | ABC002       | 2016-09-12 21:40:20 |          \
 5 |
|  5 | Maria Foo | MF001    |  4 | ABC003       | 2016-09-12 21:44:34 |          \
 5 |
|  6 | Jim Papas | JP001    |  5 | ABC004       | 2016-09-12 21:46:38 |          \
 6 |
|  6 | Jim Papas | JP001    |  6 | ABC005       | 2016-09-12 21:47:41 |          \
 6 |
|  6 | Jim Papas | JP001    |  7 | ABC006       | 2016-09-12 21:49:31 |          \
 6 |
+----+-----------+----------+----+--------------+---------------------+----------\
---+
6 rows in set (0.01 sec)
```

(the above code snippet online)

This will join the customers with the orders and will not include any customer that does not have an order. This is a start to answer our question, but it is not the answer. It brings 6 rows. Hence, if we do:

```
mysql> select count(*) from customers join orders on orders.customer_id = custome\
rs.id;
+----------+
| count(*) |
+----------+
|        6 |
+----------+
1 row in set (0.01 sec)
```

(the above code snippet online)

we will get `6` which is not the correct number of customers that they have placed at least `1` order. The correct number should be `3`. The above query repeats the same customer for every order that the customer has. That's why we do not get the correct result.

One correct SQL statement to count the customers that have placed an order is the following:

```
mysql> select count(distinct customers.id) from customers join orders on orders.c\
ustomer_id = customers.id;
+------------------------------+
| count(distinct customers.id) |
+------------------------------+
|                            3 |
+------------------------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

So, instead of `count(*)` we tell MySQL that we want to count the distinct `customers.id` values. The `customers.id` values are `4, 5, 5, 6, 6, 6` and the distinct, unique set is `4, 5, 6`, which means that the count would return `3`, which is the correct, expected result.

> *Hint:* Remember that you can alias a column header with a label on your own to make the header of the column read easier: `sql mysql> select count(distinct customers.id) as number_of_customers_with_orders from customers join orders on orders.customer_id = customers.id; +---------------------------------+ | number_of_customers_with_orders | +---------------------------------+ | 3 | +---------------------------------+ 1 row in set (0.00 sec)`

Another, easier way to count the number of customers that they have placed orders, is to use only the `orders` customers and count the distinct `orders.customer_id` values instead:

```
mysql> select count(distinct orders.customer_id) from orders;
+-----------------------------------+
| count(distinct orders.customer_id) |
+-----------------------------------+
|                                 3 |
+-----------------------------------+
1 row in set (0.00 sec)
```

(the above code snippet online)

Isn't it correct? It is. Since every customer that has placed an order appears inside the `orders` table, there is no reason to use the `customers` table. The SQL statement above consumes less resources and it is faster than the count that uses both `customers` and `orders` table, since it does not have to join any information from two different tables. Joins are costly. If you can avoid them, you have to do it.

How many leads do I have? - `left join`

Now, let's go to the opposite question. How many customers do we have that they have not placed any order? How many leads do we have?

This is more tricky and cannot be carried out only by the use of `orders` table. This is because in `orders` table we have only the customers that they have placed an order. There are some other customers inside the `customers` table that they have not placed an order, and hence, their `customers.id` does not exist as a value inside the `orders.customer_id` column.

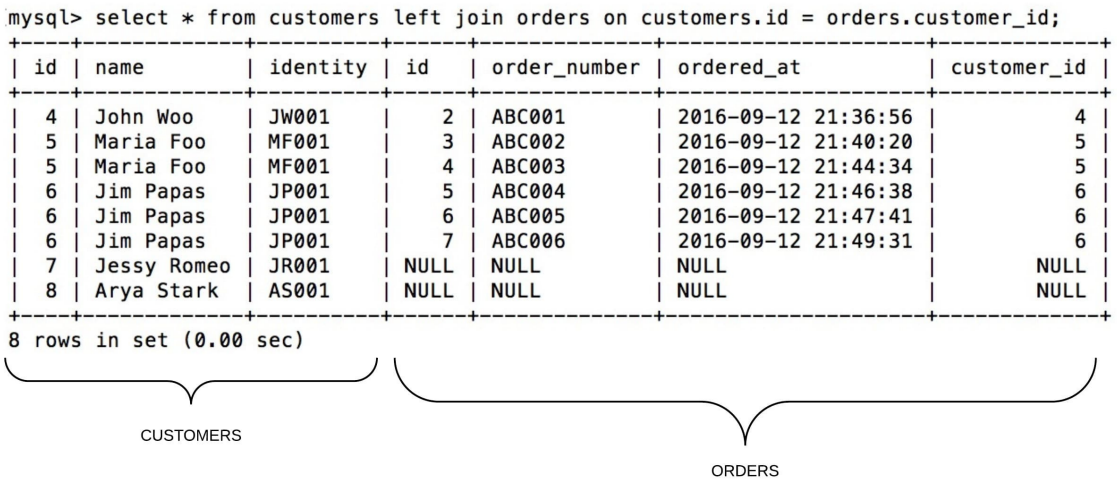
Before we actually give the query that does what we want, let's try the following query:

```
mysql> select * from customers left join orders on customers.id = orders.customer\
_id;
+----+-------------+----------+------+-------------+---------------------+------\
-------+
| id | name        | identity | id   | order_number | ordered_at          | custo\
mer_id |
+----+-------------+----------+------+-------------+---------------------+------\
-------+
|  4 | John Woo    | JW001    |    2 | ABC001       | 2016-09-12 21:36:56 |      \
      4 |
|  5 | Maria Foo   | MF001    |    3 | ABC002       | 2016-09-12 21:40:20 |      \
      5 |
|  5 | Maria Foo   | MF001    |    4 | ABC003       | 2016-09-12 21:44:34 |      \
      5 |
|  6 | Jim Papas   | JP001    |    5 | ABC004       | 2016-09-12 21:46:38 |      \
      6 |
|  6 | Jim Papas   | JP001    |    6 | ABC005       | 2016-09-12 21:47:41 |      \
      6 |
|  6 | Jim Papas   | JP001    |    7 | ABC006       | 2016-09-12 21:49:31 |      \
      6 |
```

```
|  7 | Jessy Romeo | JR001     | NULL | NULL          | NULL                |       \
  NULL |
|  8 | Arya Stark  | AS001     | NULL | NULL          | NULL                |       \
  NULL |
+----+-------------+-----------+------+---------------+---------------------+------\
-------+
8 rows in set (0.00 sec)
```

(the above code snippet online)

This is almost the same query like the one we issued earlier to get the customers with orders. The only difference is that the `join` with `customers` is not simple `join`, but, instead, it is a `left join`. The `left join` actually tells MySQL to bring all the `customers` even if they do not have corresponding entries inside the `orders` table. That's why you see the extra 2 rows, for the customers `Jessy Romeo` and `Arya Stark`. You see one row for each one of these customers, which only has data for the `customers` part of the result set, whereas it has `NULL` values for all the columns that belong to the `orders` part of the result set.



Customers With No Orders

Having this information, we can now limit the result set to the customers that they do not have orders:

```
mysql> select * from customers left join orders on customers.id = orders.customer\
_id where orders.id is NULL;
+----+-------------+----------+------+--------------+------------+-------------+
| id | name        | identity | id   | order_number | ordered_at | customer_id |
+----+-------------+----------+------+--------------+------------+-------------+
|  7 | Jessy Romeo | JR001    | NULL | NULL         | NULL       |        NULL |
|  8 | Arya Stark  | AS001    | NULL | NULL         | NULL       |        NULL |
+----+-------------+----------+------+--------------+------------+-------------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

This query uses a `where` clause. You have already learnt about the `where` clauses. Here, we limit the results to those that they have `orders.id` being `NULL`. Note that compare with `NULL` is done with the operator `is` (or `is not` for negation) and not with the operator = (or `<>` for negation). This is done because `NULL` is not a value. It only means the `absence of a value`.

Finally, since, we now know how to get the rows that include only the leads, the customers that they do not have orders, then we only have to count them with `count(*)` to find out how many they are:

```
mysql> select count(*) from customers left join orders on customers.id = orders.c\
ustomer_id where orders.id is NULL;
+----------+
| count(*) |
+----------+
|        2 |
+----------+
1 row in set (0.00 sec)
```

(the above code snippet online)

How can we get the leads without displaying order columns?

But, how can we get the leads/customers that they do not have orders, without actually listing useless order columns?

In other words, can we get a result like this?

```
+----+-------------+----------+
| id | name        | identity |
+----+-------------+----------+
|  7 | Jessy Romeo | JR001    |
|  8 | Arya Stark  | AS001    |
+----+-------------+----------+
2 rows in set (0.00 sec)
```

(the above code snippet online)

We can do that by limiting the result set to the columns of the `customers` table.

```
mysql> select customers.* from customers left join orders on customers.id = orders\
.customer_id where orders.id is null;
+----+-------------+----------+
| id | name        | identity |
+----+-------------+----------+
|  7 | Jessy Romeo | JR001    |
|  8 | Arya Stark  | AS001    |
+----+-------------+----------+
2 rows in set (0.01 sec)
```

(the above code snippet online)

How can we get the total amount of each order?

This is going to be a little bit more difficult. Let's try to answer this question taking small steps.

If we list the `orders_items`, we will see that we have the `price` and the `quantity` columns.

```
mysql> select * from order_items;
+----+----------+------------+-------+----------+
| id | order_id | product_id | price | quantity |
+----+----------+------------+-------+----------+
|  4 |        2 |          1 | 50.00 |        2 |
|  5 |        3 |          1 | 50.00 |        1 |
|  6 |        3 |          2 | 19.80 |        1 |
|  7 |        4 |          2 | 19.80 |        5 |
|  8 |        5 |          3 |  5.00 |        1 |
|  9 |        6 |          3 |  5.00 |        2 |
| 10 |        7 |          2 | 19.80 |        1 |
+----+----------+------------+-------+----------+
7 rows in set (0.00 sec)
```

(the above code snippet online)

One can say that the total amount of the first order with id 2 is 2 x 50.00 = 100.00. Whereas the total amount of the order with id 3 is 1 x 50.00 + 1 x 19.80 = 69.80. And the total amount of the order with id 4 is 5 x 19.80 = 99.00.

This means that the `order_items` table has all the necessary information to build a result set like the following:

```
mysql>
+----------+------------------+
| order_id | amount_per_order |
+----------+------------------+
|        2 |           100.00 |
|        3 |            69.80 |
|        4 |            99.00 |
|        5 |             5.00 |
|        6 |            10.00 |
|        7 |            19.80 |
+----------+------------------+
6 rows in set (0.01 sec)
```

(the above code snippet online)

How can we get there. Let's try the following first:

```
mysql> select order_items.order_id, order_items.price * order_items.quantity as a\
mount_per_order_item from order_items;
+----------+-----------------------+
| order_id | amount_per_order_item |
+----------+-----------------------+
|        2 |                100.00 |
|        3 |                 50.00 |
|        3 |                 19.80 |
|        4 |                 99.00 |
|        5 |                  5.00 |
|        6 |                 10.00 |
|        7 |                 19.80 |
+----------+-----------------------+
7 rows in set (0.00 sec)
```

(the above code snippet online)

The above result set does not differ much from the following:

```
mysql> select * from order_items;
+----+----------+------------+-------+----------+
| id | order_id | product_id | price | quantity |
+----+----------+------------+-------+----------+
|  4 |        2 |          1 | 50.00 |        2 |
|  5 |        3 |          1 | 50.00 |        1 |
|  6 |        3 |          2 | 19.80 |        1 |
|  7 |        4 |          2 | 19.80 |        5 |
|  8 |        5 |          3 |  5.00 |        1 |
|  9 |        6 |          3 |  5.00 |        2 |
| 10 |        7 |          2 | 19.80 |        1 |
```

```
+----+----------+------------+-------+----------+
7 rows in set (0.00 sec)
```

(the above code snippet online)

except from the facts that

1. it does not display the column `id`.
2. it does not display the column `product_id`.
3. the two columns `price` and `quantity` have been multiplied, and they now give the `amount_per_order_item`.

As you can see, this one here: `...order_items.price * order_items.quantity as amount_per_order_item...` multiplies the values of the two columns and sets a label to the result. Hence, the new, derived-value column, is displayed with the label `amount_per_order_item`.

Cool! Now, let's try another concept:

```
mysql> select sum(price), sum(quantity) from order_items;
+------------+---------------+
| sum(price) | sum(quantity) |
+------------+---------------+
|     169.40 |            13 |
+------------+---------------+
1 row in set (0.01 sec)
```

(the above code snippet online)

As we can see the `sum()` function can be used on a column. Then SQL will sum the values of that column for all the rows that match the result set criteria. `sum()` is an aggregate SQL function. We will learn more later on.

And what about this?

```
mysql> select order_items.order_id, sum(price), sum(quantity) from order_items gr\
oup by order_id;
+----------+------------+---------------+
| order_id | sum(price) | sum(quantity) |
+----------+------------+---------------+
|        2 |      50.00 |             2 |
|        3 |      69.80 |             2 |
|        4 |      19.80 |             5 |
|        5 |       5.00 |             1 |
|        6 |       5.00 |             2 |
|        7 |      19.80 |             1 |
+----------+------------+---------------+
6 rows in set (0.00 sec)
```

(the above code snippet online)

This is even better, because it applied the aggregate functions (the 2 `sum()` function calls) on all rows that match the criteria but, repeatedly and separately, for each order_id. Double check the amounts. Do the order_id 2 prices sum up to `50.00`? Do the order_id 3 prices sum up to `69.80`? What about the sums on `quantity` columns? Do they match?

It seems that we are getting closer and closer to our target. We are only left to combine the information from the 2 columns price and quantity, before summing up. The combination needs to be a multiplication.

```
mysql> select order_items.order_id, sum(price * quantity) as amount_per_order fro\
m order_items group by order_id;
+----------+------------------+
| order_id | amount_per_order |
+----------+------------------+
|        2 |           100.00 |
|        3 |            69.80 |
|        4 |            99.00 |
|        5 |             5.00 |
|        6 |            10.00 |
|        7 |            19.80 |
+----------+------------------+
6 rows in set (0.00 sec)
```

(the above code snippet online)

Perfect! In other words, we display the sums of prices multiplied by quantities, but limiting the summation / aggregation to the matching order_ids. So, 1 summation/aggregation for each order_id.

How can we see which customer has placed which order alongside the order totals?

We need to come back with a result such as the following:

```
+-------------+-----------+----------+--------------+----------+-----------------\
-+
| customer_id | name      | identity | order_number | order_id | amount_per_order\
 |
+-------------+-----------+----------+--------------+----------+-----------------\
-+
|           4 | John Woo  | JW001    | ABC001       |        2 |           100.00\
 |
|           5 | Maria Foo | MF001    | ABC002       |        3 |            69.80\
 |
|           5 | Maria Foo | MF001    | ABC003       |        4 |            99.00\
 |
```

```
|           6 | Jim Papas | JP001    | ABC004       |        5 |             5.00\
 |
|           6 | Jim Papas | JP001    | ABC005       |        6 |            10.00\
 |
|           6 | Jim Papas | JP001    | ABC006       |        7 |            19.80\
 |
+-------------+-----------+----------+--------------+----------+----------------\
-+
6 rows in set (0.00 sec)
```

(the above code snippet online)

Let's take little steps at a time. We initially see that it contains information from the `customers` table:

```
mysql> select customers.id as customer_id, customers.name, customers.identity fro\
m customers;
+-------------+-------------+----------+
| customer_id | name        | identity |
+-------------+-------------+----------+
|           4 | John Woo    | JW001    |
|           5 | Maria Foo   | MF001    |
|           6 | Jim Papas   | JP001    |
|           7 | Jessy Romeo | JR001    |
|           8 | Arya Stark  | AS001    |
+-------------+-------------+----------+
5 rows in set (0.00 sec)
```

(the above code snippet online)

We have used the verbose way to enlist the properties of the `customers` because we had to alias the `customers.id` to `customer_id`.

Then, desired result set references the order number and the order id. Cool. We can get them from `orders` table of course, by joining.

```
mysql> select customers.id as customer_id, customers.name, customers.identity, or\
ders.order_number, orders.id as order_id from customers join orders on orders.cus\
tomer_id = customers.id;
+-------------+-----------+----------+--------------+----------+
| customer_id | name      | identity | order_number | order_id |
+-------------+-----------+----------+--------------+----------+
|           4 | John Woo  | JW001    | ABC001       |        2 |
|           5 | Maria Foo | MF001    | ABC002       |        3 |
|           5 | Maria Foo | MF001    | ABC003       |        4 |
|           6 | Jim Papas | JP001    | ABC004       |        5 |
|           6 | Jim Papas | JP001    | ABC005       |        6 |
|           6 | Jim Papas | JP001    | ABC006       |        7 |
```

```
+-------------+-----------+----------+--------------+----------+
6 rows in set (0.00 sec)
```

(the above code snippet online)

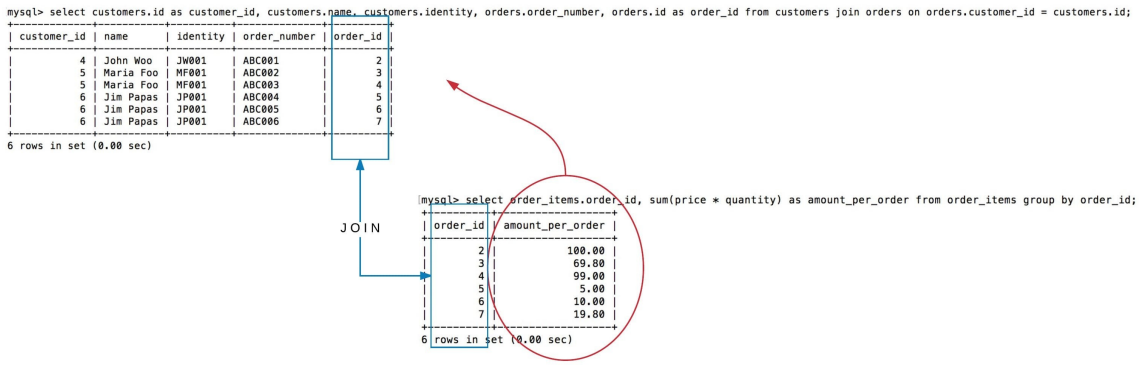
This is pretty much known to you. We joined with `orders` using the relationship of `customers` and `orders` and then we picked up the columns of `orders` that we wanted, aliasing them to label names, `order_number` and `order_id`.

Hence, we are now missing the column `amount_per_order`. You only need to remember that we got this column, earlier, using the following query:

```
mysql> select order_items.order_id, sum(price * quantity) as amount_per_order fro\
m order_items group by order_id;
+----------+------------------+
| order_id | amount_per_order |
+----------+------------------+
|        2 |           100.00 |
|        3 |            69.80 |
|        4 |            99.00 |
|        5 |             5.00 |
|        6 |            10.00 |
|        7 |            19.80 |
+----------+------------------+
6 rows in set (0.00 sec)
```

(the above code snippet online)

As you can see in the picture, our objective is to basically **join** information from 2 different queries:

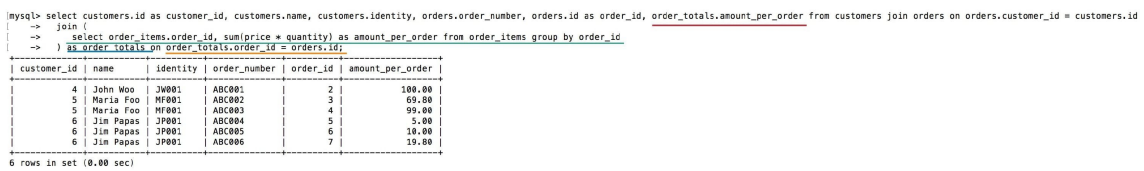


Joining Information From 2 Queries

We have the column that will join these 2 queries. It is the `order_id`. Hence, we only have to use a `join` statement. With the gotcha that we need to name the query we join on.

```
mysql> select customers.id as customer_id, customers.name, customers.identity, or\
ders.order_number, orders.id as order_id, order_totals.amount_per_order from cust\
omers join orders on orders.customer_id = customers.id
    ->   join (
    ->     select order_items.order_id, sum(price * quantity) as amount_per_orde\
r from order_items group by order_id
    ->   ) as order_totals on order_totals.order_id = orders.id;
+-------------+-----------+----------+--------------+----------+-----------------\
-+
| customer_id | name      | identity | order_number | order_id | amount_per_order\
 |
+-------------+-----------+----------+--------------+----------+-----------------\
-+
|           4 | John Woo  | JW001    | ABC001       |        2 |           100.00\
 |
|           5 | Maria Foo | MF001    | ABC002       |        3 |            69.80\
 |
|           5 | Maria Foo | MF001    | ABC003       |        4 |            99.00\
 |
|           6 | Jim Papas | JP001    | ABC004       |        5 |             5.00\
 |
|           6 | Jim Papas | JP001    | ABC005       |        6 |            10.00\
 |
|           6 | Jim Papas | JP001    | ABC006       |        7 |            19.80\
 |
+-------------+-----------+----------+--------------+----------+-----------------\
-+
6 rows in set (0.00 sec)
```

(the above code snippet online)



How Above Query Is Constructed

It is easy to combine the results of one query to another. You wrap the second one into parentheses and you give it a name. `order_totals` in our case. Then you can use that name to reference to the columns of this query. Hence we use the `order_totals.order_id` to join the query result set with the table `orders` on the common column. And also, we use the `order_totals.amount_per_order` in the main select statement to make sure that the amount per order is displayed in the final result set (see: `..., order_totals.amount_per_order from customers...`.

How can we get the customers alongside their total order amounts?

We now want to see how much money each customer has spend with us. Something like the following:

```
+-------------+-----------+----------+----------------+
| customer_id | name      | identity | customer_total |
+-------------+-----------+----------+----------------+
|           4 | John Woo  | JW001    |         100.00 |
|           5 | Maria Foo | MF001    |         168.80 |
|           6 | Jim Papas | JP001    |          34.80 |
+-------------+-----------+----------+----------------+
3 rows in set (0.00 sec)
```

(the above code snippet online)

The above result set contains the total amount purchased by each customer, next to the customer details.

How are we going to get into that result?

First, we need to think about the fact that amounts are inside `order_items`. But, `order_items` do not link to `customers` directly. It is the `orders` table that bears the `customers` information, i.e. the `customer_id`. The `orders.customer_id` could be used to link the amounts from order_items to customers.

Hence, we need to start from `order_items`, that hold the amounts and then join with `orders` that hold the `customers` `customer_id` information. Let's do that:

```
mysql> select * from order_items join orders on orders.id = order_items.order_id;
+----+----------+------------+-------+----------+----+--------------+------------\
---------+-------------+
| id | order_id | product_id | price | quantity | id | order_number | ordered_at \
         | customer_id |
+----+----------+------------+-------+----------+----+--------------+------------\
---------+-------------+
|  4 |        2 |          1 | 50.00 |        2 |  2 | ABC001       | 2016-09-12 \
21:36:56 |           4 |
|  5 |        3 |          1 | 50.00 |        1 |  3 | ABC002       | 2016-09-12 \
21:40:20 |           5 |
|  6 |        3 |          2 | 19.80 |        1 |  3 | ABC002       | 2016-09-12 \
21:40:20 |           5 |
|  7 |        4 |          2 | 19.80 |        5 |  4 | ABC003       | 2016-09-12 \
21:44:34 |           5 |
|  8 |        5 |          3 |  5.00 |        1 |  5 | ABC004       | 2016-09-12 \
21:46:38 |           6 |
|  9 |        6 |          3 |  5.00 |        2 |  6 | ABC005       | 2016-09-12 \
21:47:41 |           6 |
```

```
| 10 |        7 |          2 | 19.80 |        1 |  7 | ABC006       | 2016-09-12 \
21:49:31 |           6 |
+----+----------+------------+-------+----------+----+--------------+------------\
---------+-------------+
7 rows in set (0.00 sec)
```

(the above code snippet online)

Now, we have both `customer_id` and the attributes that contribute to the amounts, i.e. the `price` and the `quantity`, inside the same result set.

We can first use the `sum()` aggregate function like we used it earlier. We are going to get the sum of the multiplication of the `price` to `quantity`, but grouped by `customer_id`.

```
mysql> select customer_id, sum(price * quantity) as total_amount from order_items\
 join orders on orders.id = order_items.order_id group by customer_id;
+-------------+--------------+
| customer_id | total_amount |
+-------------+--------------+
|           4 |       100.00 |
|           5 |       168.80 |
|           6 |        34.80 |
+-------------+--------------+
3 rows in set (0.00 sec)
```

(the above code snippet online)

Perfect, this result set holds the information that we want. The only caveat is that we see the customer ids and not the details of the customers. But, now, we know how we can join information from the `customers` table:

```
mysql> select customers.id as customer_id, name, identity, customer_totals.total_\
amount from customers
    -> join (
    ->   select customer_id, sum(price * quantity) as total_amount from order_ite\
ms join orders on orders.id = order_items.order_id group by customer_id
    -> ) as customer_totals on customer_totals.customer_id = customers.id;
+-------------+-----------+----------+--------------+
| customer_id | name      | identity | total_amount |
+-------------+-----------+----------+--------------+
|           4 | John Woo  | JW001    |       100.00 |
|           5 | Maria Foo | MF001    |       168.80 |
|           6 | Jim Papas | JP001    |        34.80 |
+-------------+-----------+----------+--------------+
3 rows in set (0.00 sec)
```

(the above code snippet online) This is the technique that we used in the previous query. We enclose the query `select customer_id, sum(price * quantity) as total_amount from`

`order_items join orders on orders.id = order_items.order_id group by customer_id` into parentheses and we give it a name: `customer_totals`. Then we use it as the destination of a join from the `customers` table. Since both `customers` table and the query have columns that reference the same information, the customer id, we can use these columns (happen to have the same name `customer_id`) for the join.

How can we get the best customer?

Now that we have a way to get the customers alongside their total amount purchased from us, we can get the best customer. We only have to order the previous result set and limit to just 1 row. Careful, the ordering needs to be in reverse order, so that the customer with the maximum total_amount to be at the top, and then selected by the limit.

```
mysql> select customers.id as customer_id, name, identity, customer_totals.total_\
amount from customers
    -> join (
    ->   select customer_id, sum(price * quantity) as total_amount from order_ite\
ms join orders on orders.id = order_items.order_id group by customer_id
    -> ) as customer_totals on customer_totals.customer_id = customers.id
    -> order by customer_totals.total_amount desc limit 1;
+-------------+-----------+----------+--------------+
| customer_id | name      | identity | total_amount |
+-------------+-----------+----------+--------------+
|           5 | Maria Foo | MF001    |       168.80 |
+-------------+-----------+----------+--------------+
1 row in set (0.02 sec)
```

(the above code snippet online)

As you can see, the best customer is `Maria Foo`, which has total amount `168.80`.

There is another version of the above query that would bring the same result:

```
mysql> select customers.id as customer_id, name, identity, customer_totals.total_\
amount from customers
    -> join (
    ->   select customer_id, sum(price * quantity) as total_amount from order_ite\
ms join orders on orders.id = order_items.order_id group by customer_id
    -> order by total_amount desc limit 1
    -> ) as customer_totals on customer_totals.customer_id = customers.id;
```

(the above code snippet online)

This is an improved version of the previous one, because it first limits the internal query results, the results of the query that sums the order items per customer, and then joins with the customers.

Closing Note

We would like to close this chapter by giving you a link to a video / screencast with the chapter content. You may want to watch it as an alternative source of learning. However, please, note that there might be some differences to the queries executed during this video. Nevertheless, the concepts are the same.

Chapter Video / Screencast - Left Joins and SubQueries

Tasks and Quizzes

> **Before you continue**, **you may want to know that**: You can sign up to Tech Career Booster and have a mentor evaluate your tasks, your quizzes and, generally, your progress in becoming a Web Developer. Or you can sign up and get access to Tech Career Booster Slack channel. In that channel, there are a lot of people that can answer your questions and give you valuable feedback.

Quiz

The quiz for this chapter can be found here