# Programming Assignment 2 Report

|  |  |
| --- | --- |
| Student: | Christoffer Eid – ce223af[@student.lnu.se](mailto:ilir.jusufi@lnu.se) |

## Project Idea

Here you should discuss your idea. For instance:

For this assignment we have designed and implemented a system to store data about the COVID vaccination. We have downloaded the data from this source: [www.somesource.com](http://www.somesource.com/) (alternatively you can generate your own data). Our tool enables users to track this data but also provide various statistics and visualizations of the data. ….

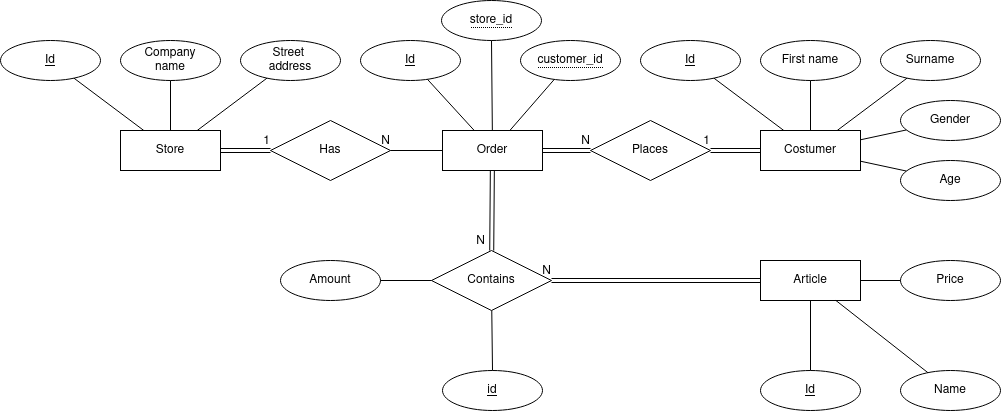
For this assignment I have designed and implemented a system to store customer, store, order and article data in one database.

I generated all the data using the tools supplied at [https://www.mockaroo.com](https://www.mockaroo.com/).

My program enable supervisor at a store chain to keep track of all its customers data and purchases, the chains supplied articles, all the chains stores different locations and more.

## Schema Design

Here you present your schema design. You can use softwares such as [https://app.diagrams.net](https://app.diagrams.net/) to draw your schema. Explain all the tables/relations and different connections they have.



All of my entities has an AUTO\_INCREMENTED id as primary key. This makes it easy to Query for the exact result the manager of the store wants to find. *Order* and *Article* has a many-to-many relationship. This makes it so that in the actual databse design, i had to create a relation table; *OrdersArticles* to keep track of what articles and how many of it there was in each order. The *OrdersArticles* table is represented by the relationship *Contains* inbetween the entities *Order* and *Article*.

Apart from this the other stuff is quite straight forward. There’s one customer per order, but a customer can have several orders, but atleast one. Because if a customer doesnt have an order, it isn’t a customer.

For each order there’s a store. And for each store, there can be any amount of orders. Here we even allow for a store to have 0 orders, since that will be the case any time we open a new store.

The *Order* entity keeps track of its connected store and customer by keeping their respctive primary keys as foreign keys.

## SQL Queries

V: **Combining *Orders*, *OrdersArticles* and *Articles* into one view.**

The following view is made up of a multirelation query and uses two *INNER JOINs.* We join table *Orders* on table *OrdersArticles* by matching the primary key *Articles.articleid* to the foreign key *OrdersArticles.article*. We then join that with table *Articles* by matching primary key *Articles.articleid* with foreign key *OrdersArticles.article.*

This view becomes really useful when querying from specific orders or shopping carts.

CREATE VIEW ordersandarticles AS

SELECT \*

FROM Orders

INNER JOIN OrdersArticles ON Orders.orderid=OrdersArticles.order

INNER JOIN Articles ON OrdersArticles.article=Articles.articleid;

Q: **List the average age of the customers in each store.**

The following query is a multirelation query and uses two *JOINs,* just like our view. We do as we did with the view except on different tables. What’s interesting in this query is that we get to use aggregation. Whenever we do aggregation in a query, it always has to be followed by a *GROUP BY*. In this query we wanted to know the average age of the customers in each store, hence we decide to *GROUP BY store.*

SELECT AVG(Customers.age), Stores.name

FROM Stores

JOIN Orders ON Orders.store=Stores.storeid

JOIN Customers ON Orders.customer=Customers.customerid

GROUP BY store

ORDER BY AVG(Customers.age);

Q: **List all of the chains Stores and their respective addresses.**

This query is an easy one but functional if the stores manager ever just need a list of all their sister stores and their respective addresses. I chose to sort the query by *name* just because I thought it would look nicer, this could be crucial if the chain were to grow.

SELECT name, address

FROM Stores

ORDER BY name;

Q: **List the amount of customers each store has.**

Does your store have more customers than your sister stores? No? Then get to work!

This query is important for friendly competition between stores. This is a multirelation query. We join tables Orders and Stores together, then we count the amount of customers *GROUP*ed *BY store* to get the amount of *customer*s per store.

SELECT COUNT(customer), Stores.name

FROM Orders

JOIN Stores ON Orders.store=Stores.storeid

GROUP BY store

ORDER BY COUNT(customer);

Q: **List the price and name for an article.**

This query might be the most useful one. It lists the *name* and *price* of an article based on its *articleid*. This will be the query that forexample cashiers use when they scan the barcodes or if the customers of manager just wants to check up on an article naming or pricing.

SELECT name, price

FROM Articles

WHERE articleid=?;

Q: **List the customer who placed an order.**

If we have to take a return from a customer. The customer brings the reciept. On the reciept there’s a *orderid.* Now we want to know if this is the customer who bought it. We check up on the *orderid* and in return we get the customer’s credentials, knowing for sure who the customer that bought the item was we can safely pay the customer back.

In this multirelation query we join *Customers* and *Orders* on the *customerid*, which is a foreign key in *Orders* and primary key in *Customers.*

SELECT Customers.customerid, Customers.firstname, Customers.lastname

FROM Orders

INNER JOIN Customers ON customer=Customers.customerid

WHERE orderid = ?;

Q: **List the shopping cart from an order.**

Using this query the store could make good analytical statistics to know what is bought in combination with what, or the query could be used as a customer service. Maybe the customer can log in from home to check all their previous shopping carts.

In this query we make full use of our view. We *SELECT* all the relevant information and show it in nice alpabetical order based on *name.* The search is made based on *orderid*.

SELECT name, price, amount

FROM ordersandarticles

WHERE orderid = ?

ORDER BY name;

## Discussion and Resources

The project uses libraries; mysql.connector, csv and tkinter.

Mysql.connector is used for connection to and querying from mysql database.

Csv is used for reading .csv-files into python.

Tkinter is used to create the interface in which the user uses the program and queries from the database. It was my first time ever creating a graphical user interface, so I knew I needed something that was fairly easy to use. I did some looking around and found these two sources that both used tkinter in a way that suited me perfectly:

<https://github.com/Ramoooona/Python-GUI-MySQL>

https://realpython.com/python-gui-tkinter/

I had no issues what so ever with my data thanks to the fact that I myself generated all the data that I needed through; [https://www.mockaroo.com](https://www.mockaroo.com/).

At first I was thinking about using already existing data from <https://www.dataportal.se/>. But I changed my mind after thinking about good database normalisation structures. It seemed like all the data that was up for grabs was very poorly organised. Basically just randomly thrown into a .csv-file.

Source code: <https://github.com/krukle/store-database>

Check readme.txt for installation instructions.

Video demonstration: