Programming Assignment 2 Report

Student(s): Stefan Fransson – sf222vs@student.lnu.se

1. Project Idea

For this assignment I designed and implemented a system for a sales division to view company sales. The design enables users to track this data and also provide various statistics and filtering options. In this project random dummy data was used and generated with the tools on the site www.mockaroo.com.

2. Schema Design

The schema is designed around a transaction order. An *order* has a seller and a buyer. The seller is represented as *salesman* working at *office* in the company. And the buyer is represented as a *Client reference* for the buying company, the *client.* The *client* can have many *client references*/employees that places *orders* and the selling company can have many *salesmen* that sells, but only one *client reference* and only one *salesman* can make an *order.*

An *order* can be one or several *products* in various quantities This is represented by the "*Order Line Item"* entity because an *order* can have many *products* and a *product* can be sold in many *orders*.

1 or m Client references **represents** 1 Client
1 Client reference **places** 0 or m Orders
0 or m Salesman **works at** 1 Office
1 Salesman **sells** 0 or m Orders
1 Order **contains** 1 or m "Order Line Item"
0 or m "Order Line Item" **contain** 1 Product

In the end of this report there is a diagram of the entity relations.

3. SQL Queries

Q: Creates a view that is restricted to only focus on the salesman and the office sales activity.
This is a multi-relation query that uses JOIN to create a view that contains data that the users are allowed to access and by letting the users only query against the view other data is protected. In this project the client list. Another befit of using view is that it’s basically a global variable that holds a sub query that can be used in other queries.

The query first joins the data in the tables together on the sales table by matching the foreign keys to the prime keys. E.g. the foreign key sales.salesmanID is matched to the prime key salesmen.salesmanID. Then creates the view representing this joined tables.

In the ER diagram sales is named order.

```
CREATE VIEW sales_report AS
    SELECT
        sales.saleID,
        sales.salesDate,
        offices.city AS office,
        salesmen.name AS salesman,
        saleslineitem.quantity,
        products.productName,
        products.price AS productPrice
    FROM
        sales
            JOIN
        salesmen ON sales.salesmanID = salesmen.salesmanID
            JOIN
        offices ON salesmen.officeID = offices.officeID
            JOIN
        saleslineitem ON sales.saleID = saleslineitem.saleID
            JOIN
        products ON saleslineitem.productID = products.productID
```

Q: List the total price (total sale value) per month and office. Also add a row counter and format the total price for visual representation.
This query queries the view explained on previous page, so its indirectly a multi relation query that uses join. It also uses the aggregation and grouping. The sum of all (quantity * productPrice), initially taken from “Sales Line Item” a.k. “Order Line Item” in the ER diagram, and groups the sums of “Total price” per office and month.

```
SELECT row_number() over (order by salesDate) as 'rowID', monthname(salesDate) as
       'month', office, sum(quantity) as '# of products', format(sum(quantity *
       productPrice),0) as 'Total price'
FROM sf222vs_sales.sales_report
group by monthname(salesDate), sales_report.office
```

The query used the previous created view even if views are often slower, but views also give another security layer for databases.

This query gives an overview of how well different sales offices are doing compared to each other.

Q: **List the average quantity (number of sold products per sale) and average total price (total sale value per sale) per salesman and year. Also add a row counter and format the total price for visual representation.**

This query uses a sub query that queries the view to calculate the sum of quantity as tot_quantity, and the sum of (quantity * productPrice) as totalprice. grouped by salesID. After that the "head" query lists the average quantity and the average value per year based on the result from the sub query.

```
SELECT row_number() over (order by salesDate) as 'rowID', year(salesDate) as
'year', salesman,
format((avg(tot_quantity)),1) as avg_quantity, format((avg(totalprice)),0) as
avg_value
FROM (
SELECT saleID, salesDate, salesman, sum(quantity) as tot_quantity,
        sum(quantity * productPrice) as totalprice
        FROM sf222vs_sales.sales_report
        group by saleID
 ) as sub
group by year(salesDate)
```

The query can be used for comparing sales results between salesmen and be an indication for where extra support and education is needed.

Q: List products in descending order based on total selling value. Also includes total sold quantities of the listed product.

This multi relation query uses a sub query to narrow down the data processed by the query. In this case it doesn't really matter, but in very large data sets it will reduce the request time. The outer query combines all the sold quantities of a product and calculates the "Total price" base on product price and quantity.

```
select products.productName, sum(saleslineitem.quantity) as tot_quantity,
sum(products.price * quantity) as 'Total price'
from (
select sales.saleID, sales.salesDate
from sales
where sales.salesDate >='2020-01-01' and salesDate <='2020-12-31'
) as sub
join saleslineitem on sub.saleID = saleslineitem.saleID
join products on saleslineitem.productID = products.productID
group by products.productName
```

This query gives an overview of the how well the products are performing compared to each other.

A little side note about variable references with blank space inside like “Total price”. They seems to be a reason for headaches because mySQL doesn’t handle them very well and can give unforeseen consequences.

Q: List client companies and their contact references based on total selling value in descending order.

The query uses a sub query to first get the information of the total quantity and total selling price per client reference. After that the outer query connects the client references to the client company and combine their quantities and total sell prices into total values for the company.

```
select clients.name as Company, group_concat(clientreferences.referenceName) as
Contacts, sum(sub.quantity) as tot_quantity, sum(sub.totalprice) as tot_price
from (
select sales.referenceID, sum(saleslineitem.quantity) as quantity
,sum(saleslineitem.quantity * products.price) as totalprice
from sales
join saleslineitem on sales.saleID = saleslineitem.saleID
join products on saleslineitem.productID = products.productID
group by sales.referenceID
) as sub
join clientreferences on sub.referenceID = clientreferences.referenceID
join clients on clientreferences.clientID = clients.clientID
group by clients.name
order by tot_price desc
```

This query shows what companies and their contacts are most important to nourish.

4. Discussion and Resources

The application is made by using the tkinter library, it’s a standard library so just importing it should work. The app is also using tkcalendar. Just do a: pip install tkcalendar

The python code is divided into three packages model, controller and view.

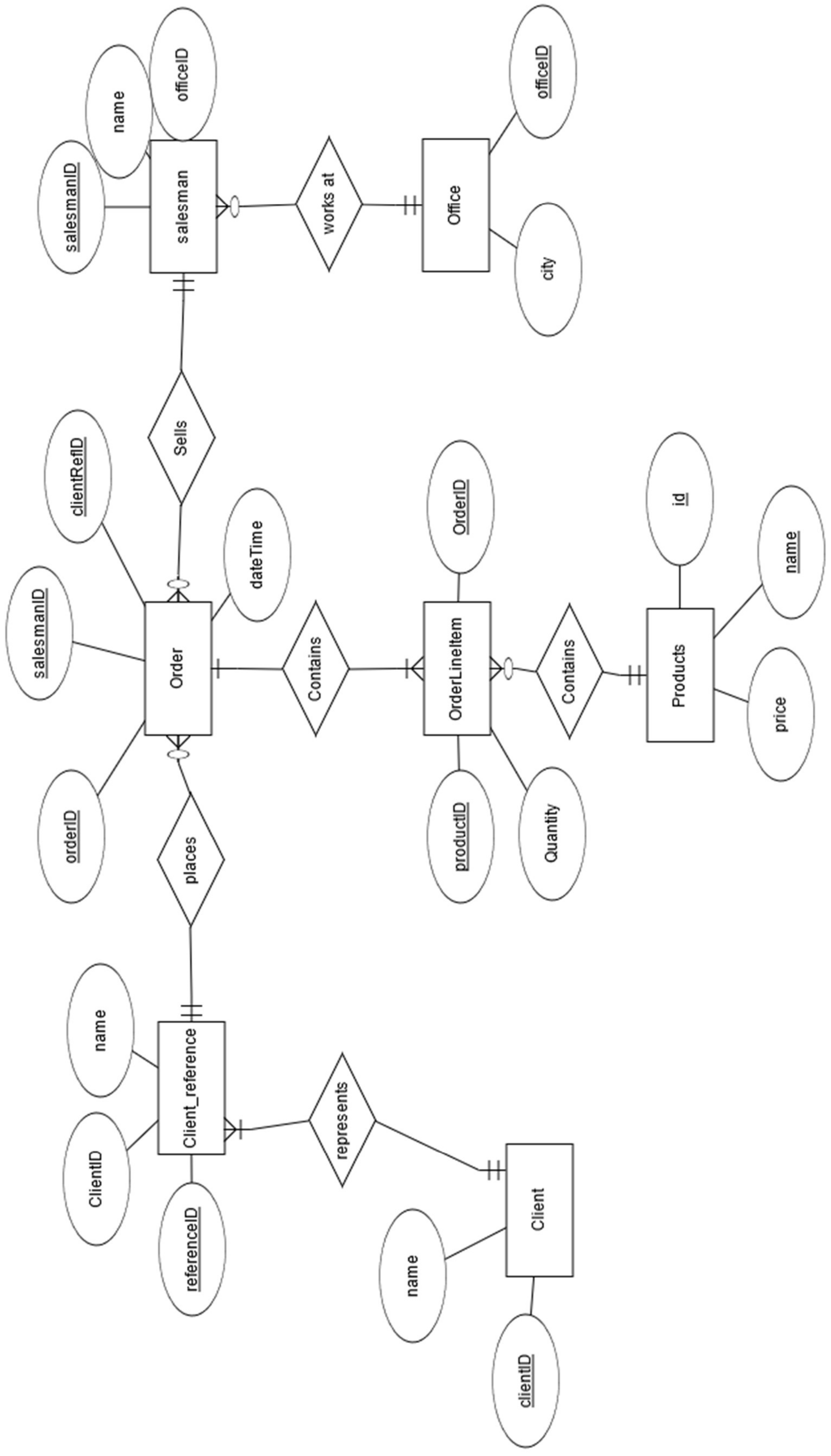
In model there is a sub directory Fake_data containg all the csv files and also the file core_queries.py that contains all the queries the application use for creating the database, tables and inserting data.
In controller the file controller_queries.py contains all the queries that the app uses for creating the view and sending user request to the database.

Start the app by running the controller.py

Link to video:
https://www.youtube.com/watch?v=9-HYCdOyhOU

Sorry for the watermark!

5. Appendix


salesman
salesmanID
name
officeID
works at
Office
officeID
city
Sells
Order
clientRefID
salesmanID
orderID
dateTime
Contains
OrderLineItem
OrderID
productID
Quantity
Contains
Products
id
name
price
places
Client_reference
name
ClientID
referenceID
represents
Client
name
clientID