**Idea**

Our general project idea was to develop a platform that allows customers to order coffee at registered coffee shops online. So we started creating our project with that goal in mind right from the beginning. So instead of designing a database and thinking about queries first, we had the plan for the application at the start and then designed a database with the necessary complexity around it. We also created the vast majority of our queries to serve a purpose within the application. So we basically used the approach you would use when planning to develop a real-world usable platform.

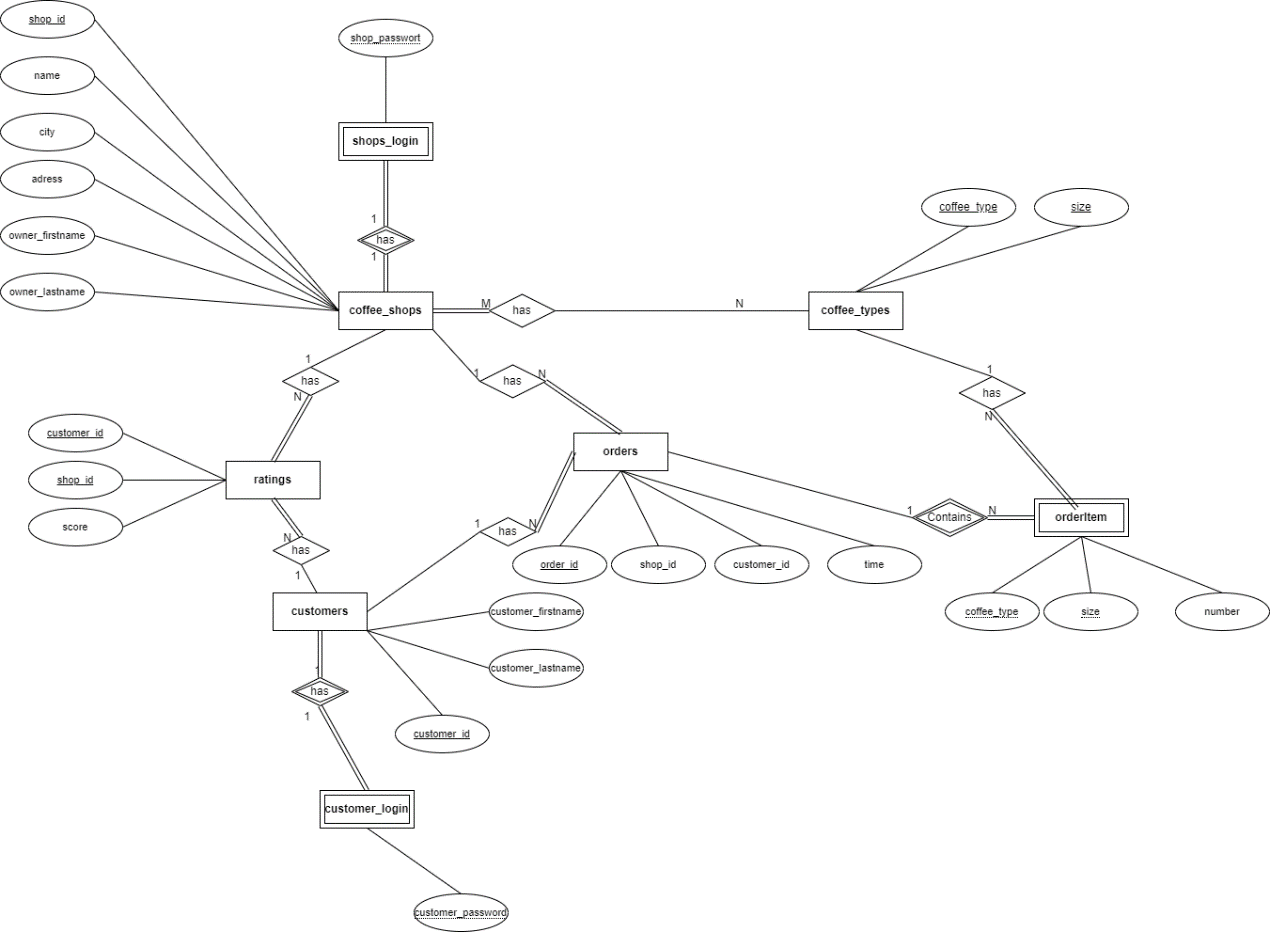
**Implementation**

Within our project we have implemented all queries into the backend of our Flask app. Some of the queries are behind the “admin page” of our application to give administrators an easy to use option to take a look at all the tables of the coffee database as well as offering some quick view options on predefined data.

The vast majority of queries is deeply incorporated into the application though, so the user does not even get the notion of executing a query when navigating our application. Many queries are executed at the time a subpage is loaded for instance or with the click of icons or buttons.

So many of the queries described below include some variables that will be inserted dynamically by application logic.

**ER-Model**



**Normalization**

First normal form was quite easy to fullfill since we just designed our database with the principle of one value per cell in mind from the ground up.

The second normal form is fullfilled automatically with our database design for most tables since we mostly use single-column primary keys that are also the only candidate for primary key. With that it can not be the case that a non-primary key columns is functionally dependent on only a real subset of a primary key candidate. Also for our multi-column primary key tables "coffee types", "ratings" and "coffee\_shop\_coffee\_types" no column depends on only a real subset of the primary key, so there is also no violation of second normal form.

We had some concerns about meeting third normal form though. For most tables everything was fine, but for the “coffee\_shops\_coffee\_types” table something looked odd. In our first approach we used a column "type\_id" as primary key to uniquely identify each type-size-combination. But the columns "coffee\_type" and "size" actually are also a candidate key together. Basically “type\_id” could just be seen as a surrogate key and everything would be fine, however in our first approach we constructed “type\_id” based on information from the natural candidate key. Because the “type\_id” would always be a digit-letter-combination like “3L” for instance with the digit just enumerating different coffee types but the letter representing the size (“L” for “Large”, “S” for “Small”, “M” for “Medium”). So basically this introduced a functional dependency between the “size” column and the surrogate key which might not violate third normal form directly, but still introduced redundant and likely hard to maintain information. So we decided to simply remove the surrogate key.

**Queries overview**

**Login & Signup queries:**

Query behind the login form of the customer login:

"SELECT \* FROM customer\_login WHERE customer\_login.customer\_id='"+login\_info['username']+"' AND customer\_login.customer\_password='" + login\_info['password'] + "'"

Purpose: this query takes the user input from the customer login page and returns the row with the login data or nothing if no user is registered under these login credentials yet

(the user gets a response on success and failure from the UI based on the query return value)

Query behind the signup form of the customer signup:

“insert into customers (customer\_id, customer\_firstname, customer\_lastname) values (" + signup\_info['username'] + ",'" + signup\_info['firstname'] + "','" + signup\_info['lastname'] + "'); insert into customer\_login (customer\_id, customer\_password) values (" + signup\_info['username'] + ",'" + signup\_info['password'] + "');”

Purpose: this query takes all the user input from the input form on the signup page and inserts the information into the “customers” table as well as the defined id (serves as sort of username) and the password into the “customer\_login” table

Query behind the signup form of the coffee shop login:

"SELECT \* FROM shop\_login WHERE shop\_id='"+login\_info['username']+"' AND shop\_password='" + login\_info['password'] + "'"

Purpose: same as for customer login

Query behind the login form of the coffee shop login:

"insert into coffee\_shops (shop\_id, name, city, adress, owner\_firstname, owner\_lastname) values (" + signup\_info['shop\_id'] + ",'" + signup\_info['name'] + "','" + signup\_info['city'] + "','" + signup\_info['adress'] + "','"+ signup\_info['owner\_firstname'] + "','" + signup\_info['owner\_lastname'] + "'); insert into shop\_login (shop\_id, shop\_password) values (" + signup\_info['shop\_id'] + ",'" + signup\_info['password'] + "');"

Purpose: same as for customer signup

Login-validity-check query

"SELECT \* FROM customer\_login WHERE customer\_login.customer\_id='"+data['username']+"' AND customer\_login.customer\_password='" + data['password'] + "'"

Purpose: on each new subpage this query is executed first to make sure that only a loged in user can access the page (& to identify which user is acting and thereby also reading from or writing to the database)

(application logic: with the login button on the login page the id and password are saved in a cookie locally on the users machines and these values are then used in the query against the login tables to make sure a valid customer / coffee shop is interacting with the app)

**Customer landing page queries:**

Coffee shop overview query (behind the customer landing page)

select c.shop\_id, c.name, c.city, r.score, round(average\_rating\_mat.average\_score, 1) from (coffee\_shops c left join ratings r on c.shop\_id = r.shop\_id) left join average\_rating\_mat on average\_rating\_mat.shop\_id = c.shop\_id where r.customer\_id = {data['username']} order by r.score desc;

Purpose: Uses two left joins to first combine the coffee shops table with the ratings table and left joins the result of the first join with the materialized view about the average ratings of the coffee shops.

This retrieves a list of all the coffee shops in any case since we automatically insert a rating with value null for each coffee shop from each newly registered customer. Without that actually not all shops would be displayed since the join condition with the rating table would not be met when a customer has not rated a shop yet.

Ratings overview query over ratings a user has given (most critical ones)

"select c.name, r.score from coffee\_shops c join ratings r on c.shop\_id = r.shop\_id where r.customer\_id = {data['username']} order by r.score limit 5;"

Purpose: Uses a join between the coffee\_shops and the ratings table and filters by the customer\_id according to the value read from the cookie in the application in order to retrieve the 5 worst ratings a specific customer has given

Recent orders overview query

"select o.order\_id, o.order\_date, cof.name from (customers c join orders o on c.customer\_id = o.customer\_id) join coffee\_shops cof on o.shop\_id = cof.shop\_id where o.customer\_id = {data['username']} order by o.order\_date desc limit 5;"

Purpose: giving an user an overview over the last 5 orders he has done (sorted by timestamp from newest to oldest), for that a join of the orders with the customers table is done and results are cut down to the orders of one specific user using a where condition with his id (which is read from the cookie)

Rating update query

"INSERT INTO ratings (customer\_id, shop\_id, score) VALUES ({data['customer\_id']}, {data['shop\_id']}, {data['score']}) ON CONFLICT (customer\_id, shop\_id) DO UPDATE SET score = {data['score']};"

Purpose: Inserts a new rating given by a customer into the "ratings" table or updates an existing rating (note: with the concept that there is always a null value rating created at registration of a new customer it's now basically always update)

**Coffee shop landing page queries:**

Coffee types offered query:

"SELECT ct.coffee\_type, ct.size, false AS is\_not\_null FROM coffee\_types ct WHERE NOT EXISTS (SELECT 1 FROM coffee\_shops\_coffee\_types rel WHERE ct.coffee\_type = rel.coffee\_type AND ct.size = rel.size AND rel.shop\_id = {data['username']}) UNION (SELECT rel.coffee\_type, rel.size, true AS is\_not\_null FROM coffee\_shops\_coffee\_types rel WHERE rel.shop\_id = {data['username']}) ORDER BY coffee\_type asc, size desc;"

Purpose: The goal behind this rather complex query is to display all the coffee types that there are and in addition the information whether the specific coffee shop is offering them or not. Since we specify which coffee shop offer which coffee types with our M-to-N relation through the table coffee\_shops\_coffee\_types it would be not so hard to retrieve all the offered coffee types of a shop. However since we wanted to include the option to see all possible offerings and make the UI in such a way that it is possible to change the offering with a simple click we had to combine the offered types with the possible but not currently offered ones. Within this process the additional column containing "True" or "False" according to whether a coffee type is offered is added.

This is achieved by this query, which basically consist of two parts that are combined via a union statement and then sorted at the end. In the first half we retrieve all of the coffee types that a coffee shop does not offer using a "where not exists" to filter out those which are offered from all possible coffee types. In the second half we add all those coffee types that are offered by checking whether they are present in the coffee\_shops\_coffee\_types table.

Top 5 most critical ratings query:

"select c.customer\_firstname || '' || c.customer\_lastname as customer\_name, r.score from customers c join ratings r on c.customer\_id = r.customer\_id where r.shop\_id = {data['username']} order by r.score limit 5;"

Purpose: Same functionality as on customer page query, but now of course the customers table is involved in the join the where condition uses the shop\_id. The query gives a shop owner an overview over the 5 most critical (worst) ratings. Also the name of the customer will be displayed. (Real world purpose: owner can get a good understanding of the experience from customers and can maybe derive specific action items for improvements from critical ratings).

Orders query:

"select o.order\_id, o.order\_date, c.customer\_firstname || '' || c.customer\_lastname as customer\_name from customers c join orders o on c.customer\_id = o.customer\_id where o.shop\_id = {data['username']} order by o.order\_date desc;"

Purpose: Retrieves all the orders a certain coffee shop has received, the join is used to output the full name of a customer along with the order "metadata" (so id & time, the actual items of the order are not retrieved here though -> this is done on the "Ordering details page" with a simple query)

Add new coffee type to offering query:

"INSERT INTO coffee\_shops\_coffee\_types (shop\_id, coffee\_type, size) VALUES ({data['shop\_id']}, '{data['coffee\_type']}', '{data['size']}');"

Purpose: This query is run in the backend if a specific coffee shop adds a coffee type from their offerings list

Remove a coffee type from offering query:

"delete from coffee\_shops\_coffee\_types where shop\_id = {data['shop\_id']} and coffee\_type = '{data['coffee\_type']}' and size = '{data['size']}';"

Purpose: This query is run in the backend if a specific coffee shop removes a coffee type from their offerings list

**Ordering details page:**

Returning details of an order:

"select coffee\_type, size, number from orderitem where order\_id = {data['order\_id']}"

Purpose:

Displays a coffee shop owner the details of an order so that they can begin processing it and actually preparing the coffee in a real world scenario

**Ordering page:**

Returning coffee types and sizes that can be ordered:

"select coffee\_type, size from coffee\_shops\_coffee\_types where shop\_id = {data['shop\_id']}"

Purpose: Returns coffee types and sizes that a customer can order at a specific coffee shop

Retrieve last order\_id value query:

"select order\_id from orders order by order\_id desc limit 1;"

Purpose: Retrieves the highest order\_id in order to increment that number by 1 and thereby set a new order\_id for the order that will be created

Order insertion query:

"insert into orders (order\_id, shop\_id, customer\_id, order\_date) values ({last\_id\_value+1}, {data['shop\_id']}, {data['customer\_id']}, {current\_date});

insert into orderitem (coffee\_type, size, number, order\_id) values {insert\_items\_tuples};"

Purpose: Inserts all the order details needed to perform an order by a customer into the orders and the orderitem table

(the value for customer\_id is read from the cookie, the value for shop\_id is read from the url of the page where it has been dynamically set previously, last\_id\_value is calculated using the previously defined query, the insert values for the orderitem table are sent by the frontend of the application and have to be parsed to fit the insert statement syntax of SQL)

(Important: For one order multiple lines have to inserted into orderitem table, which made parsing not completely straightforward)

**Admin Page**

**(Note: Admin Page is not secured with login credentials right now, which of course makes no sense, but this could easily be changed in the future. For development purposes thw admin page has been used to quickly view tables etc., so a login would have slowed this process down.)**

Freestyle query can be any query and query for tables is just "select \* from {table}"

Purpose: Gives an easy to use option to get an overview over the whole database, allows an administrator to do corrections

Quick-view queries:

General purpose: The quick view queries make use of the view & materialized view we definied and use a cross join (these queries do not necessarily serve a purpose with respect to the application, but are rather there to have some more complex queries for the project. We implemented them into the application in this way anyways since we wanted to have all queries integrated into the application and thereby executable via UI.)

Materialized view query:

"SELECT \* FROM public.average\_rating\_mat;"

View query:

SELECT \* FROM public.worst\_shop\_ratings;"

Cross-join query:

"SELECT \* FROM coffee\_shops CROSS JOIN coffee\_types;"