IMSE - Milestone 2
Online shop

Naderi Sahel, Mrekaj Matúš

20.06.2022

Running the project

To run the application navigate to the root directory of the project and then in the command line type `docker compose build`, this will build the frontend and backend images. After that, you can run the application with `docker compose up`. It takes some time to start the app, we observed that ~30-40 seconds should be enough to wait.

After the application is started successfully you can open it up on your browser by typing in one of these URLs: `http://localhost:80` `https://locahost:443`. The former uses plain HTTP and the latter uses HTTPS. However, no matter which one you choose you will always be redirected to use the HTTPS version. **We'd also like to note that the system is using a self-signed certificate thus, based on the browser you're using, you'll need to trust the self-signed certificate** (usually there should be a button that the browser displays for proceeding to the website).

Tech-stack

The requirements of the milestone were that we needed to implement the system in two versions. One for RDBMS, for which we decided to use the MySQL database. The other should be implemented with a NoSQL database for which we choose MongoDB.

The application itself is divided into the backend and frontend. The backend connects to these two databases and set's up the databases with the respective data. The backend itself is implemented using the Spring framework in Java, version 11 to be specific. The backend exposes a RESTful HTTP API that is then used by the frontend for communication. We decided on this design as it would simplify the logic significantly as both the RDBMS and MongoDB data would just need to be mapped to the agreed-upon responses of the backend API which the frontend would then consume. Thus, the frontend doesn't need to implement any changes to how the data is being displayed based on which database is used for the queries. A vital part of the backend is also populating the database with "fake" data. For that, we've decided to use the following library: https://github.com/DiUS/java-faker which simplified the process of data generation as we didn't need to spend much time on coming up with great names.

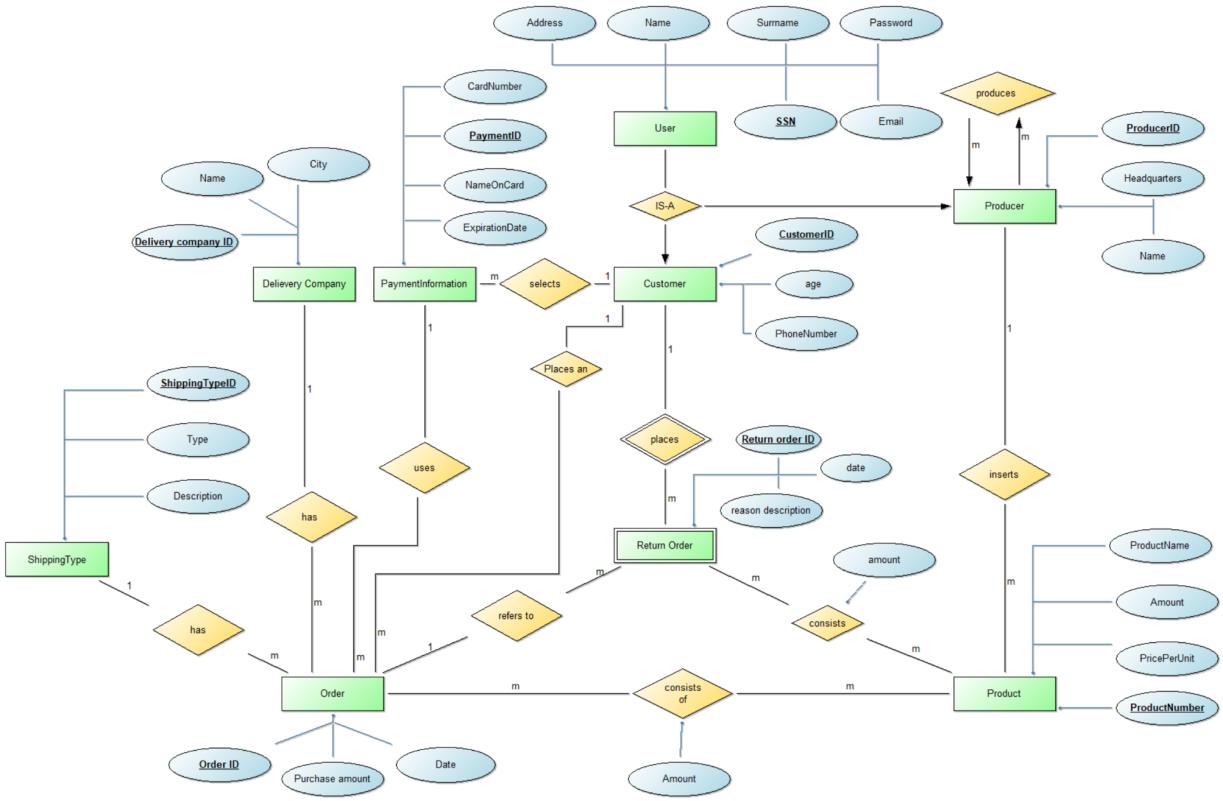
The frontend part is implemented with react, to handle rendering the components and all the logic for the system. For styling we decided to use TailwindCSS as it allows us to re-use predefined styling for buttons lists etc…, thus also saving us time on the frontend development. To deploy the frontend we've also decided to use NGINX to act as a

reverse proxy and handle the incoming requests appropriately. NGINX is configured to use HTTPS by default even if the client connects with HTTP. The certificate was created by following the instructions here:

https://letsencrypt.org/docs/certificates-for-localhost/#making-and-trusting-your-own-certificates

RDBMS model

We've decided to implement no changes to the logical model that we presented in Milestone 1. Just to avoid any ambiguity we present the logical model from Milestone 1.



Following this logical design, we've mapped it 1:1 to MySQL as can be seen by the following MySQL schema. We'd like to note that the M:N relationships were broken up into 1:M relationships, for example, the M:N between the order and product is broken up into a separate table called `order_products` that is connected to the products and Order table via 1:M relationships.

```
CREATE TABLE `customers` (
  `age` int NOT NULL,
  `phone_number` varchar(255) DEFAULT NULL,
  `ssn` varchar(255) NOT NULL,
  PRIMARY KEY (`ssn`),
  CONSTRAINT `FKh0qdt9ua03b3xukvn09b43ee0` FOREIGN KEY (`ssn`) REFERENCES
`users` (`ssn`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `delivery_companies` (
  `delivery_company_id` bigint NOT NULL AUTO_INCREMENT,
  `city` varchar(255) NOT NULL,
  `name` varchar(255) NOT NULL,
  PRIMARY KEY (`delivery_company_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `order_products` (
  `uuid` bigint NOT NULL AUTO_INCREMENT,
  `amount` int NOT NULL,
  `order_id` bigint NOT NULL,
  `product_number` bigint NOT NULL,
  PRIMARY KEY (`uuid`),
  KEY `FKawxpt1ns1sr7al76nvjkv21of` (`order_id`),
  KEY `FKerkisau5crjk37b42pu4k0w29` (`product_number`),
  CONSTRAINT `FKawxpt1ns1sr7al76nvjkv21of` FOREIGN KEY (`order_id`)
REFERENCES `orders` (`order_id`),
  CONSTRAINT `FKerkisau5crjk37b42pu4k0w29` FOREIGN KEY (`product_number`)
REFERENCES `products` (`product_number`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `orders` (
  `order_id` bigint NOT NULL AUTO_INCREMENT,
  `date` date NOT NULL,
  `purchase_amount` double NOT NULL,
  `customer_ssn` varchar(255) NOT NULL,
  `delivery_company_id` bigint NOT NULL,
  `payment_id` bigint NOT NULL,
  `shipping_type_id` bigint NOT NULL,
  PRIMARY KEY (`order_id`),
```

```
  KEY `FKipdlmpxgprjxb5ib2nh8e6u24` (`customer_ssn`),
  KEY `FKfb7lbeigc1bbnw6kw9qb9boqv` (`delivery_company_id`),
  KEY `FKixfu58ap4ebuqkbg3rqme1mqj` (`payment_id`),
  KEY `FKsreawtnv69xlt9gi64hq5dsf` (`shipping_type_id`),
  CONSTRAINT `FKfb7lbeigc1bbnw6kw9qb9boqv` FOREIGN KEY
(`delivery_company_id`) REFERENCES `delivery_companies`
(`delivery_company_id`),
  CONSTRAINT `FKipdlmpxgprjxb5ib2nh8e6u24` FOREIGN KEY (`customer_ssn`)
REFERENCES `customers` (`ssn`),
  CONSTRAINT `FKixfu58ap4ebuqkbg3rqme1mqj` FOREIGN KEY (`payment_id`)
REFERENCES `payment_informations` (`payment_id`),
  CONSTRAINT `FKsreawtnv69xlt9gi64hq5dsf` FOREIGN KEY (`shipping_type_id`)
REFERENCES `shipping_types` (`shipping_type_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `payment_informations` (
  `payment_id` bigint NOT NULL AUTO_INCREMENT,
  `card_number` varchar(255) NOT NULL,
  `expiration_date` date DEFAULT NULL,
  `name_on_card` varchar(255) NOT NULL,
  `customer_ssn` varchar(255) NOT NULL,
  PRIMARY KEY (`payment_id`),
  KEY `FKbwxyvhpc3e7c8p0rnb2w3s9r9` (`customer_ssn`),
  CONSTRAINT `FKbwxyvhpc3e7c8p0rnb2w3s9r9` FOREIGN KEY (`customer_ssn`)
REFERENCES `customers` (`ssn`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `producers` (
  `headquarters` varchar(255) NOT NULL,
  `producer_name` varchar(255) NOT NULL,
  `ssn` varchar(255) NOT NULL,
  PRIMARY KEY (`ssn`),
  CONSTRAINT `FK2d48yv1d2bhapcafwlkqc6oma` FOREIGN KEY (`ssn`) REFERENCES
`users` (`ssn`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `producers_of_producer` (
  `parent_ssn` varchar(255) NOT NULL,
  `child_ssn` varchar(255) NOT NULL,
  PRIMARY KEY (`parent_ssn`,`child_ssn`),
  KEY `FKchvewiw77w272buvboalq7cma` (`child_ssn`),
  CONSTRAINT `FKchvewiw77w272buvboalq7cma` FOREIGN KEY (`child_ssn`)
REFERENCES `producers` (`ssn`),
  CONSTRAINT `FKdtch2p1h6rapsj2exifmcdeot` FOREIGN KEY (`parent_ssn`)
```

```
REFERENCES `producers` (`ssn`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `products` (
  `product_number` bigint NOT NULL AUTO_INCREMENT,
  `amount` int NOT NULL,
  `price_per_unit` double NOT NULL,
  `product_name` varchar(255) NOT NULL,
  `producer_ssn` varchar(255) NOT NULL,
  PRIMARY KEY (`product_number`),
  KEY `FKhd7xwc7jjoi0kw874v4lie7lr` (`producer_ssn`),
  CONSTRAINT `FKhd7xwc7jjoi0kw874v4lie7lr` FOREIGN KEY (`producer_ssn`)
REFERENCES `producers` (`ssn`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `return_order_products` (
  `uuid` bigint NOT NULL AUTO_INCREMENT,
  `amount` int NOT NULL,
  `product_number` bigint NOT NULL,
  `customer_ssn` varchar(255) NOT NULL,
  `return_order_id` bigint NOT NULL,
  PRIMARY KEY (`uuid`),
  KEY `FKb5oiqdgvoshl1kr8xgcy7u2yx` (`product_number`),
  KEY `FKd7m611kcxwdfbf20vgmveqicg` (`customer_ssn`,`return_order_id`),
  CONSTRAINT `FKb5oiqdgvoshl1kr8xgcy7u2yx` FOREIGN KEY (`product_number`)
REFERENCES `products` (`product_number`),
  CONSTRAINT `FKd7m611kcxwdfbf20vgmveqicg` FOREIGN KEY (`customer_ssn`,
`return_order_id`) REFERENCES `return_orders` (`customer_ssn`,
`return_order_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `return_orders` (
  `customer_ssn` varchar(255) NOT NULL,
  `return_order_id` bigint NOT NULL,
  `date` date NOT NULL,
  `reason_description` varchar(255) NOT NULL,
  `order_id` bigint NOT NULL,
  PRIMARY KEY (`customer_ssn`,`return_order_id`),
  KEY `FK7lptnhci1qqu8rbe82tqcjpcv` (`order_id`),
  CONSTRAINT `FK7lptnhci1qqu8rbe82tqcjpcv` FOREIGN KEY (`order_id`)
REFERENCES `orders` (`order_id`),
  CONSTRAINT `FKkt8p4rox9uqai1dcjjgdstnrr` FOREIGN KEY (`customer_ssn`)
REFERENCES `customers` (`ssn`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

```
CREATE TABLE `shipping_types` (
  `shipping_type_id` bigint NOT NULL AUTO_INCREMENT,
  `description` varchar(255) NOT NULL,
  `type` int NOT NULL,
  PRIMARY KEY (`shipping_type_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `users` (
  `ssn` varchar(255) NOT NULL,
  `address` varchar(255) NOT NULL,
  `email` varchar(255) NOT NULL,
  `name` varchar(255) NOT NULL,
  `password` tinyblob NOT NULL,
  `surname` varchar(255) NOT NULL,
  PRIMARY KEY (`ssn`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

The schema above was obtained by the `mysqldump` command.

Data import

The database filling has been made available on the frontend and can be found in the top left corner after opening the application in the browser. Since it's implemented as an endpoint on the backend theoretically it can be pressed an unlimited number of times

It will only fill out the SQL database to be able to perform the main use-cases and reports. Enough data is being generated to be able to work with the use-cases and reports

Implementation of a Web system

Each member of the team implemented his **main use-case** and the **report**.

Mrekaj Matus:

`Main use-case` - Return order

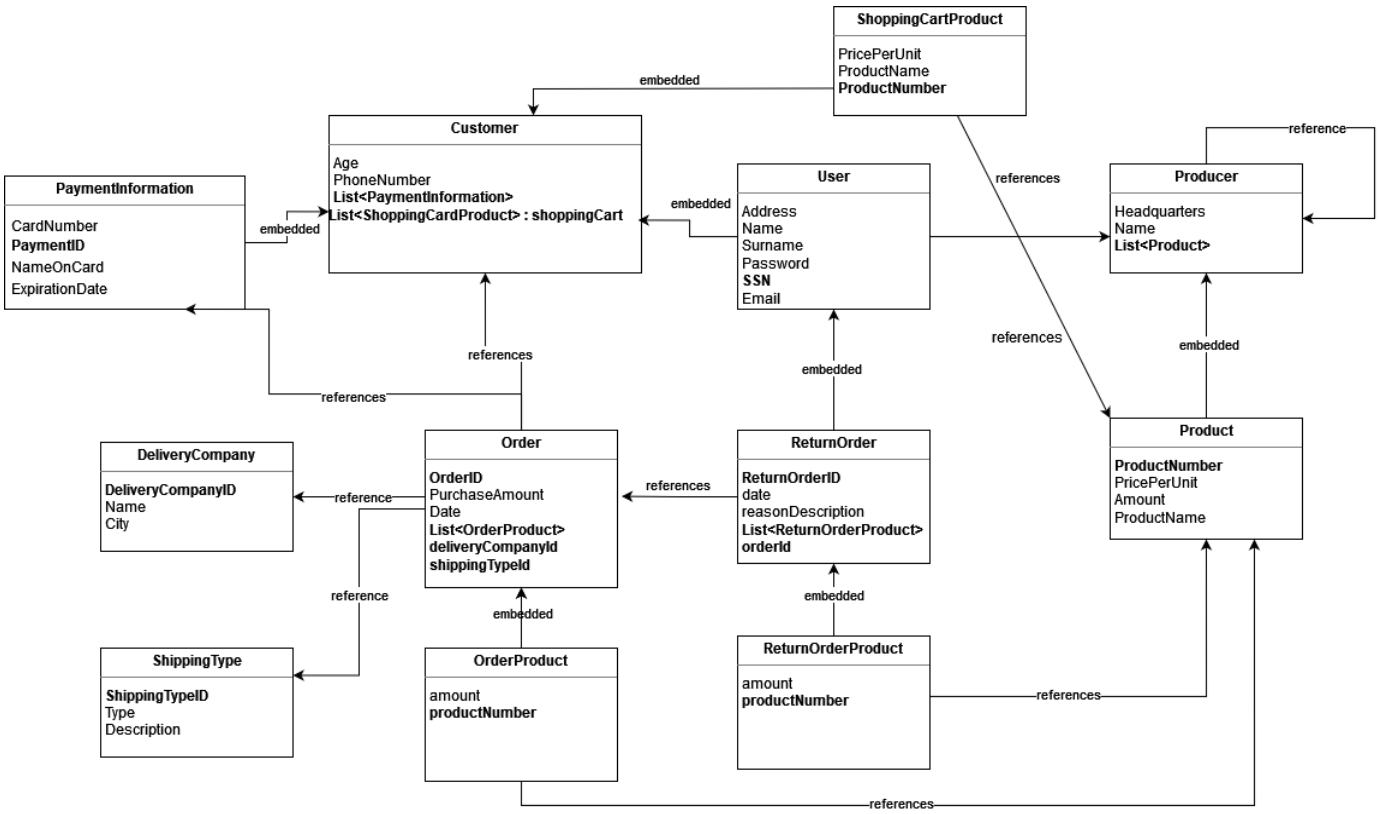
`Report` - top 5 returned products last year per user

Naderi Sahel:

`Main use-case` - Submit order

`Report` - top 10 ordered products in the last year

NoSQL Design



Overall there are many changes between the RDBMS and NoSQL version of the design. We decided to Embedd a lot of the data into documents to avoid unnecessary Joining of documents. It also made sense to embed the data into those documents.

Users Collection - the `users` collection, producers, customers, and the users table in RDBMS, were merged into a single document. To distinguish between the two types of users a field named `_class` is added which either contains `CustomerNoSQL` or `ProducerNoSQL` values. The common fields which were originally in the `users` table in RDBMS are now embedded in the `users` collection for both customers and producers. This saves us a lot of joining since we have all the information in one place and make it overall easier to access.

Furthermore, for `producers` we have also changed their design, The `products` are now embedded in the `producers` as a dynamic growing array. We don't expect the array to grow significantly, a producer has a number of products but we don't expect the number to go above 3000-4000. And we found it to be more convenient for the products to be embedded in the producers as we often needed the producer with the product and thus avoided additional joining. For customers, we've had a similar change with the `payment information`. It is very common in our application when dealing with orders or returns orders to also include the `payment information` with it, thus embedding the data inside the customers made perfect sense, a customer doesn't have that much of credit cards, and it avoids unnecessary joining. Another embedded document in the customer is `returned orders`. In RDBMS it was modeled as a weak entity, meaning it cannot exist if the customer doesn't exist. Additionally, customers don't return that many products over a period of 1 year, thus we decided to embed the returned orders in customers in the user collection. We also don't expect the returned orders array to grow significantly.

In RDBMS we've split up the M:N relationships into tables with 1:M relationships. These tables contained the ordered products and their amount or the returned products and their amount. In the NoSQL design, we've looked at it as overhead designing it as a separate collection, thus they are now embedded in the `returned orders`, part of the `users` collection, and in the `orders` collection saving unnecessary joins when dealing with orders or returned orders.

The other tables such as `delivery companies`, `shipping types`, and `orders` were designed as separate collections as embedding any of the information would be counterproductive. To give an example embedding the shipping type or delivery companies within the orders means that if we wanted to change the name of the delivery company or something related to the shipping type we would need to update all of the orders whereas instead if we use a reference for into this collection we just update a single entry.

For the same reason, as was just stated above, we also decided not to include the `product name`, for example, the ordered products or returned products as if the name of the product would change we would need to update all of the names of the products in all of the collections instead we use just a reference to the product that is embedded inside the producer.

In the RDBMS model, we did not include a seperate table for the shopping cart, but instead saved every added Shopping Cart product for a customer in a HashMap on the server. By contrast, in the NoSql model we embedded the shopping Cart products into the customerNoSql due to the reason, that the shopping Cart products will not grow that extremely. One side effect is, that if one product name for example changes, we need to update all specified shoppingCartproducts in the customer documents. But this will not be a huge problem, due to the previous reason stated, that the dynamically growing array of shoppingCartproducts will not likely have more than 10 products inside. We also included productName and pricePerUnit to the ShoppingCartNoSql, because of the reason that the GET ShoppingCartProducts endpoint gets called really often, thus we get all details needed with one single request.

NoSQL Indexing

ReturnOrder (part of Customers in the users collection):

`ReturnOrderId` - Joining on the returned order is also common whether in the report or just working on returned orders in general.

`Date` - The report about returning the top 5 returned products filters the data by last year thus we added an Index on the date of the returned order.

Product (part of Producers in the users collection):

`ProductNumber` - Whether listing orders or returned orders they always include the product name. Thus the index on the productNumber made sense.

PaymentInformation (part of Customers in the users collection):

`PaymentID` - Orders on the frontend are displayed with the last 4 digits of the credit card used. Thus when listing orders you always need to look up the payment option from the customers.

Producer:

`ProducerName`- Included as an index because of the report - top 10 ordered products need the productName from Products which is embedded in the Producer document.

Data Migration

This part is similar to the data import. The migration is exposed as a button on the frontend, so whenever the migrate button is pressed the current state of the MySQL database will be migrated to NoSQL and the frontend will also update the URLs to use the NoSQL paths so further interaction with the frontend would be using the NoSQL database.

Comparison of use-cases and reports

Main use-case - Return Order (`Mrekaj Matus`)

MySQL

We actually need to insert into two tables namely **`return_orders`** and **`return_order_products`** (this table was created by splitting up the M:N relationship). The first insert just inserts the information regarding the returned order, the second insert actually inserts the returned product and the amount of the ordered product from the order. At last, we also need to update the product amount by the returned amount.

```
INSERT INTO return_orders(customer_ssn, return_order_id, date,
reason_description, order_id)
VALUES('0aa08c69-92ec-45e3-b9a0-66430468f32a', 485, CURRENT_DATE(),
'description', 257);

INSERT INTO return_order_products(
uuid, amount, product_number, customer_ssn, return_order_id)
VALUES(1000, 1, 15, '0aa08c69-92ec-45e3-b9a0-66430468f32a', 485);

UPDATE products SET amount = amount + 1 WHERE product_number = 15;
```

NoSQL

For the NoSQL version the processes are simplified. Since the returned orders are embedded in the customer we only need to execute 2 queries. Inserting a new return order and updating the amount in the products.

```
db.users.updateOne({
"_class": "com.imse.onlineshop.nosql.entities.CustomerNoSQL",
"_id": "0aa08c69-92ec-45e3-b9a0-66430468f32a"
}, {
   $push: {
       "returnedOrders": {
           "returnOrderId": 485,
           "reasonDescription": "description",
           "date": new Date("2022-07-30T22:00:00.000Z"),
           "orderId": 257,
           "products": [{"product": 15, "amount": 1}]
       }
   }
})

db.users.updateOne({
"_class": "com.imse.onlineshop.nosql.entities.ProducerNoSQL",
"products.productNumber": 15
}, {
   $inc: {
       "products.$.amount": 1
   }
})
```

Report - top 5 most returned products from last year for each customer sorted by the returned amount. (Mrekaj Matus)

From Milestone 1 the report states that:

Entities	Customer, Returned Order, Product
Filtered by	Date (last year),
Sorted by	amount.

The RDBMS version of the report actually joins 4 tables in total, namely `Customers`, `ReturnOrders`, `Products`, and the `ReturnOrderProducts` table that was created from the M:N relationship.

```
SELECT ranking, name, surname, product_name, year, total_returned
FROM (
         SELECT ROW_NUMBER() over (PARTITION BY name, surname ORDER BY sum(rop.amount) DESC) as ranking,
                u.name,
                u.surname,
                p.product_name,
                year(ro.date) as year,
                sum(rop.amount) as total_returned
         FROM users as u
                  INNER JOIN customers as c ON c.ssn = u.ssn
                  INNER JOIN return_orders as ro ON c.ssn = ro.customer_ssn
                  INNER JOIN return_order_products as rop ON rop.return_order_id = ro.return_order_id
                  INNER JOIN products as p ON p.product_number = rop.product_number
         WHERE ro.date
                   BETWEEN (MAKEDATE(year(current_date) - 1, 1)) AND (MAKEDATE(year(current_date) - 1,
365))
         GROUP BY u.name,
                  u.surname,
                  p.product_name,
                  year(ro.date)
     ) as result
where result.ranking <= 5;
```

Compared to the RDBMS version, due to the embedding of documents and organization of documents in the NoSQL database, the NoSQL version only performs a single join on the embedded `products` in the `user` collection. Other than that the difference between the two queries is minimal, `ROW_NUMBER` was just replaced by `group` operation and the filter by date and the second group by is equivalent in the NoSQL version. Thus, the NoSQL version of the report should be the better performing one.

```
db.users.aggregate([
    {"$match": {"_class": {"$eq": "com.imse.onlineshop.nosql.entities.CustomerNoSQL"}}},
    {"$unwind": "$returnedOrders"},
    {"$match": {"$expr": {"$eq": [{"$subtract": [{"$year": "$$NOW"},{"$year": "$returnedOrders.date"}]},
1]}}},
    {"$addFields": {"returnedOrders.date": {"$year": "$returnedOrders.date"}}},
    {"$unwind": "$returnedOrders.products"},
    {
        "$lookup": {
            "from": "users",
            "let": {"product": "$returnedOrders.products.product"},
            "pipeline": [
                {"$unwind": "$products"},
                {"$match": {"$expr": {"$eq": ["$products.productNumber","$$product"]}}},
                {"$project": {"_id": 0,"products.productNumber": 1,"products.productName": 1}}
            ],
            "as": "returnedProductProducer"
        }
    },
    {"$unwind": "$returnedProductProducer"},
    {
        "$group": {
            "_id": {
                "name": "$name",
                "surname": "$surname",
                "productName": "$returnedProductProducer.products.productName",
                "year": "$returnedOrders.date"
            },
            "totalReturned": {"$sum": "$returnedOrders.products.amount"}
        }
    },
    {"$sort": {"totalReturned": -1}},
    {
        "$group": {
            "_id": {
                "name": "$_id.name",
                "surname": "$_id.surname",
                "year": "$_id.year"
            },
            "products": {
                "$push": {
                    "productName": "$_id.productName",
                    "totalReturned": "$totalReturned"
                }
            }
        }
    },
    {
        "$project": {
            "_id": "0",
            "name": "$_id.name",
            "surname": "$_id.surname",
            "year": "$_id.year",
            "products": {
                "$slice": ["$products",0,5]
            }
        }
    },
    {"$sort": {"name": 1,"surname": 1}}
])
```

Main use-case - Submit Order (Sahel Naderi)

MySQL

```
INSERT INTO payment_informations (card_number, customer_ssn,
expiration_date, name_on_card)
VALUES('513041312311231', '184cb449-6547-4d8b-b876-bc075cc9bb4e',
DATE '2022-12-12', 'sahel naderi');

INSERT INTO orders (customer_ssn, date, delivery_company_id, payment_id,
purchase_amount, shipping_type_id)
VALUES('0aa08c69-92ec-45e3-b9a0-66430468f32a', current_date(),
1, 1, 12.3, 1 );

INSERT INTO orders (customer_ssn, date, delivery_company_id,
payment_id, purchase_amount, shipping_type_id)
VALUES('184cb449-6547-4d8b-b876-bc075cc9bb4e', current_date(),
1, 266, 12.3, 1 );

INSERT INTO order_products(order_id, product_number, amount )
VALUES(10, 20, 1);

UPDATE products SET amount = amount - 1 WHERE product_number =
20;
```

NoSQL

For the NoSQL part, we only need to insert into the orders table and reference the Customer, the PaymentInformation and the ShippingType. The ordered products are embedded in the Order Table, so we can directly add the ordered Items to the Order document

```
db.orders.insert({
"orderId": 400,
"purchaseAmount": 23.2,
"Date": new Date("2021-06-12T20:00:00.000Z"),
"deliveryCompanyId": 1,
"shippingTypeId": 1,
"paymentId": 1,
"customerSSN": "0aa08c69-92ec-45e3-b9a0-66430468f32a",
"products": [{ "product": "20", "amount": 1}]})


db.orders.updateOne({
```

```
        {_id: 400},
        { $push: { "products": { "product": "20", "amount": "1" }}}
    )

    db.users.updateOne({
    "_class": "com.imse.onlineshop.nosql.entities.ProducerNoSQL",
    "products.productNumber": 20
    }, {
       $inc: {
           "products.$.amount": -1
       }
    })
```

Report - top 10 ordered products (Sahel Naderi)

Originally I planned to do a report on top 5 orders with the biggest revenue in the last year, sorted by Customer's name. The problem is that I do not see any business value in this report, because it does not show the progress the business is making and the only scenario in which the management would use this kind of report is to give discounts for the customers with the biggest orders in terms of purchase amount. That is why I changed my report to the top 5 ordered products, so that the management can buy more of these products because of the high demand.

```
SELECT ranking, total_amount, product_name, producer_name, year
        FROM (
                SELECT ROW_NUMBER() over (ORDER BY sum(op.amount) DESC) as ranking, SUM(op.amount) as
total_amount, p.product_name, pr.producer_name, year(o.date) as year
                from orders as o
                        inner join order_products as op on o.order_id = op.order_id
                        inner join products as p on op.product_number = p.product_number
                        inner join producers as pr on p.producer_ssn = pr.ssn
                                WHERE o.date BETWEEN (MAKEDATE(year(current_date) - 1, 1)) AND
(MAKEDATE(year(current_date) - 1, 365))
                                GROUP BY p.product_number, pr.producer_name, year(o.date)
                                ) as result
                WHERE result.ranking <= 10
```

Entities	Producer, Order, Product
Filtered by	Date (last year),
Sorted by	amount.

It joins the Orders table with the Products table and subsequently with the Producer table and filters only the orders that occured in the last year.

```
db.orders.aggregate([

{ "$match" : { "$expr" : { "eq" : [{ "$subtract" : [{ "$year" : "$$NOW"},

{ "$year" : "$date"}]}, 1]}}},

{ "$addFields" : { "date" : { "$year" : "$date"}}},

{ "$unwind" : "$products"},

{ "$lookup" : { "from" : "users", "let" : { "product" : "$products.product"}, "pipeline" : [{ "$unwind" :
"$products"}, { "$match" : { "$expr" : { "$eq" : ["$products.productNumber", "$$product"]}}}, ], "as" :
"orderedProduct"}},

{ "$unwind" : "$orderedProduct"},

{ "$group" : { "_id" : { "productName" : "$orderedProduct.products.productName", "producerName" :
"$orderedProduct.producerName", "year" : "$date"}, "totalOrdered" : { "$sum" : "$products.amount"}}},

{ "$sort" : { "totalOrdered" : -1}},

{ "$group" : { "_id" : { "year" : "$_id.year"}, "products" : { "$push" : { "productName" :
"$_id.productName", "totalOrdered" : "$totalOrdered", "producerName" : "$_id.producerName"}}}},

{ "$project" : { "_id" : 0, "year" : "$_id.year", "products" : { "$slice" : ["$products", 0, 10]}}}

])
```

In comparison to the RDBMS report, this query is more efficient, because only one single join is necessary instead of 3 joins. Due to the fact that Products is embedded in Producer and order_products is embedded in Order, we only join the Orders table with the Producers table. So when I join the order table with the producer table in NoSQL, then I can also get the products table, so I can get the producer name and the product name. In the last phase I sort by the totalOrdered amount and push all products in a products array in the last $group command. Then I only need to slice the products array to get the top 10 ordered products.