1. NoSQL schema for the online shopping database

The main structure of the NoSQL schema for the online shopping database consists of 5 collections (customers, past_orders, products, inventory_levels and ratings). The following steps were followed to ensure that all requirements will be met:

1) Every "**customer**" entity should include information about name, sex, age, customer ID, and if this customer is a Loyalty Program Member, it will also include additional fields such as the date joined, points accumulated, and membership levels. This information is relatively stable, so the data are static and do not require frequent updates (more reads than writes). Therefore, the before mentioned fields will be embedded inside the customers collection.

```
customer {
"id": ObjectID,
"name": string,
"sex": string,
"LoyaltyMember": Boolean,
"date_joined": date,
"points_accumulated": int32,
"membership_level": string,
"customer_address": [ {
    "house_number": int32,
    "street": string,
    "city": string,
    "state": string,
    "province": string,
    "county": string,
    "post_code": string
    } ]
"current_order": [{
    "total_cost": double,
    "order_items": [ {
        "product_id": ref<product.id>,
        "quantity": int32 }]
    }]

"past_orders": [ ref<past_orders.id> ]
"ratings": [ ref<ratings.id> ]
"recommended" : [ ref<products.id> ]
}
```

Figure 1. Customer Collection

2) "**Customer Addresses**" will include information about one or more addresses of each customer. The data are mostly retrieved and not updated. Moreover, it is believed that each customer will have a few addresses, if not only one. Therefore,

this entity will not take much storage and it should be embedded as an array type inside “customers”.

3) “**Current orders**” entity will be embedded inside “customers” as an array-type. This is because the number of “current orders” is small (it will not take much storage), will often be retrieved (and we want to have fast access), and for some customers it may be rarely updated. The fields will be similar to past orders.

4) We want to quickly retrieve “**recommended**” products every time a user logs in. As a result, a relevant field should be inside “customers” so as we can achieve small read-time. However, we also want to update them frequently. Thus, this field inside “customers” will only reference to the id of a product and it won’t include all product details, which would slow down the updates.

5) “**Past orders**” will be a new collection. It is less often queried, but each customer may have many past orders, so embedding it inside “customers” collection will most likely increase the storage of “customers”. We will include an array-type field inside “customers” which will reference to “past orders” ids. “Past orders” collection will contain an order id, the total cost and the “order items”. “Order items” will be embedded inside “past orders” as an array-type subdocument (because they won’t be updated and won’t take much storage), with a quantity field and an id field pointing to the id of a product.

```
past_orders {
"id": ObjectID,
"total_cost": double,
"past_order_items": [ {
    "id": ref <product.id>,
    "quantity": int32
    } ]}
```

Figure 2. Past Orders Collection

6) “**Products**” will be a new collection. All products have some common information, but each product type has its own information as well. Thus, the collection will include fields for the common information and it will also include subdocuments, where each subdocument will represent a product type and will contain information which is

relevant for each type. As an example, for a book product there will be information about the common fields and the book fields, but it is irrelevant with CDs or home appliances and information in those fields will not exist.

```
products {
"id": ObjectID,
"name": string,
"description": string,
"dimensions": {
    "height": double,
    "width": double,
    "depth": double
    }
"weight": double,
"rating": double,
"price": double,
"supplier_cost": double,

"book": {
    "author_name": string,
    "publisher": string,
    "year_of_publication": int32,
    "page_count": int32
    }

"CD": {
    "artist_name": string,
    "producer_name": string,
    "no_tracks": int32,
    "playing_time_hours": double
    }

"home_appliances": {
    "colour": string,
    "voltage": int32,
    "style": string
    }

"inventory_levels": [ ref<inventory_levels.id> ]
}
```

Figure 3. Products Collection

7) Each product is linked with its "**daily inventory levels**". This includes stable information, not frequently used but the number of each products' "inventory levels" will be huge. Thus, "inventory levels" will be a new collection, and inside "products" collection we will only include an array-type field, which will have fields referencing to "inventory levels" collection. This won't take much storage, as it would if we embedded all "inventory levels" information inside "products".

8) We want to intensively retrieve the "**ratings**" that each customer gave to a product, so as we can quickly retrieve information for recommender system. We will not

include this entity inside “products”, as almost 1% of customers have voted each product and thus, it will require much storage. We will also not include it inside customers, since they will be retrieved slower in this case. It will be a new collection and a field referencing to “ratings” id will be included inside customers. The “ratings” collection will reference to a customer and a product, and it will also include a score field.

```
inventory_levels {
"id": ObjectID,
"date": date,
"quantity": int32,
"depot": string,
"remark": string
}

ratings {
    "id": ObjectID,
    "CustomerID": ref<customer.id>,
    "product_id": ref<product.id>,
    "rate": double
}
```

Figure 4. Inventory Levels and Ratings Collection

2. Recommender system: User-based Collaborative Filtering

Based on the above schema, we will now create a recommender system, which will recommend products to customers. In this method, we predict the rating of products that a customer hasn’t rated yet, and the prediction is based on users who are considered as similar to the target user. In the end, we recommend the target user the products that have the highest predicted rating.

A recommender system needs to intensively retrieve the ratings’ information, which was already taken into account by storing ratings in a new collection. To implement this system, we first created three main functions, which are given in the attached *py file*. A short description of each function is given below:

1) **pearson(ratings, u)**: This function takes as input a “ratings” dataframe, and a string “u” which corresponds to the id of the target user. “Ratings” contains the rating that each user gave to every product (with possible missing values if no rating is given, while the index consists of the id (string) of each user and the column names are the id (string) of every product. This function calculates the

pearson correlation coefficients and finds the similarity between target user and every other user. In the end, it returns a dataframe where the index is the id of the target user and the column names are the id of all the other users.

2) **neighbours(sim, item, k = 10)**: This function takes as input a similarities' dataframe (which is returned by pearson function), the string id of an item and a threshold parameter k. In the end, it returns a list with the ids of the k most similar users who have rated the input item. Of course, if there are less than k users who have rated this item, then it returns all the users who rated it, sorted from the most similar to the least one.

3) **predict(ratings, u)**: This function uses the two previously mentioned functions. It takes as input the "ratings" dataframe and the id of the target uses (as a string). It finds all the items of "ratings" dataframe that the target user has not rated yet, and it predicts the rating of these items based on the k similar users. In the end, it returns a sorted dataframe with the predicted ratings, where the indexes are the ids of the items and the column name is the id of target user.

We will now test these functions on our tiny database. In order to connect python with mongodb, we will import **MongoClient** from **pymongo** python package. The following code is used to perform the connection (the *connection_string* variable should be changed to reflect the correct mongodb connection string, and the *UD* should be changed to reflect the database's name):

```
from pymongo import MongoClient
# connect to MongoDB
# change the connection_string to
# reflect your own connection string
connection_string = '''input connection string'''
client = MongoClient(connection_string)
# UD is the name of the database. It should be changed
# to reflect the correct database name
db=client.UD
```

Next, we retrieve all documents from "ratings" collection and we insert them in a list. Then, we create the "ratings" dataframe, which has the same structure as previously mentioned:

```
# Get all documents from mongodb
ratings_collection = db['ratings']
cursor = ratings_collection.find({})
ratings_storage = []
for document in cursor:
    ratings_storage.append(document)
# Keep only the ratings' info we need
ratings = pd.DataFrame()
for rating in ratings_storage:
    col = str(rating['Product_ID'])
    ind = str(rating['Customer_ID'])
    score = rating['rate']
    ratings.loc[ind, col] = score
```

We will now use *predict* function to predict the ratings of a customer named “Kina” (with id '5c0d31079f9bf10be414828e'). Kina has rated 5 out of 11 products, so we will predict her ratings on the rest 6 products and we will recommend her the product with the highest predicted ratings. After getting the results from *predict* function, we store the id of the product into the “recommended” field, inside the customer named “Kina”.

```
from bson import ObjectId
# Predict the ratings
recommended = predict(ratings, '5c0d31079f9bf10be414828e')
# Get customers collection
customers_col = client.UD["customers"]
# Insert recommended product into target user
product_id = recommended.index[0]
find_customer = {"_id": ObjectId('5c0d31079f9bf10be414828e')}
newvalues = { "$push": { "recommended": ObjectId(product_id) } }
customers_col.update(find_customer, newvalues)
```

The product that we recommended her is a CD named “50 Years-Don't stop”, with a predicted rating of 8.58. The *recommended* dataframe which was returned from *predict* function can be seen below:

```
In [22]: recommended
Out[22]:
                          5c0d31079f9bf10be414828e
5c0e857f2ab7e21390451331                      8.58
5c0e87722ab7e21390451333                      6.98
5c0e88be2ab7e21390451334                      5.58
5c0e862c2ab7e21390451332                      2.58
5c15865b7da1860be092011f                      0.58
5c0e84a32ab7e21390451330                     -0.42
```