Aalto University

Database project for the home delivery orders of the grocery chain

Saidnassimov Sanzhar, sanzhar.saidnassimov@aalto.fi

CS-A1150

2021

Introduction

The project describes database structure of the home delivery system of a particular grocery chain. The database has been designed in such way that it stores information about users of the service, the stores of the grocery chain, employees (collectors), products in the assortment of each store and their composition and orders. In addition to that database also contains additional structure for different recipes.

Description of the UML diagram

In the figure below, the UML diagram of the above mentioned architecture is presented (see fig.1).

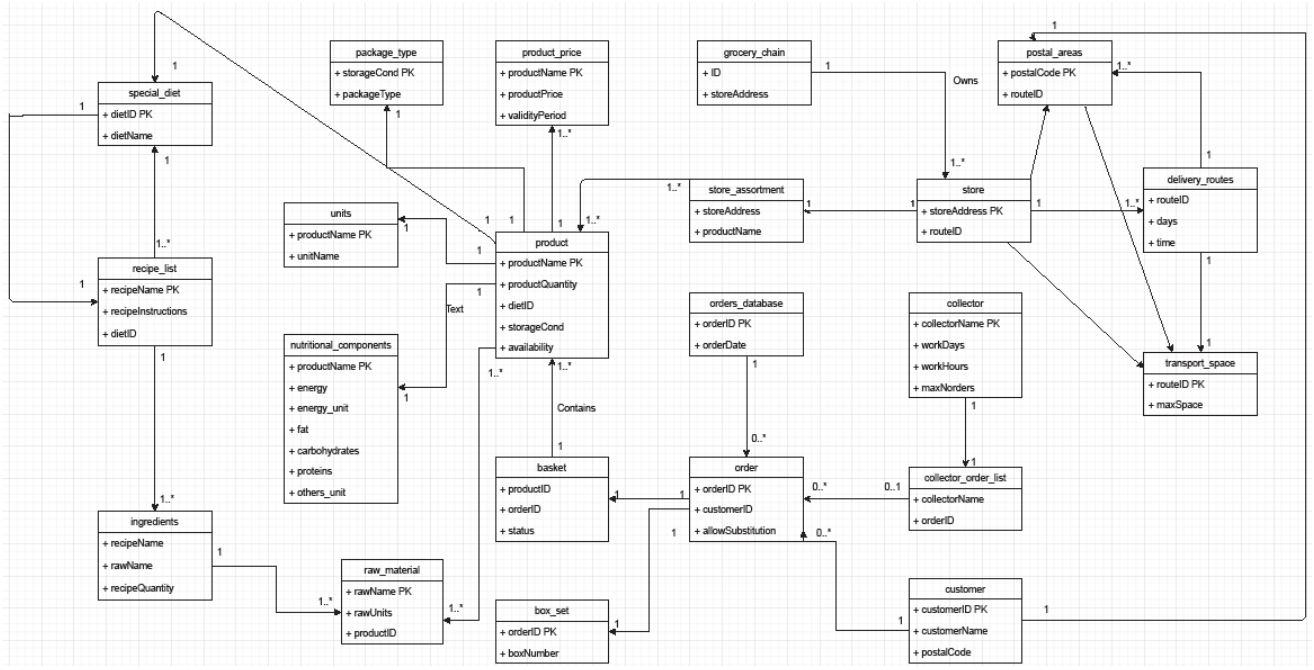

Figure 1: UML diagram of the delivery system

The architecture of the database begins with the table *grocery_chain,* which contains information on all stores of the chain. The attribute which differentiates stores from each other is address of the food mart itself. This address used as a foreign key to the table *store*, which consists of one more attribute – 'routeID'. In terms of the food delivery system, we have no necessity in any other data, but the delivery route of the particular store and identifier routeID, which in essence is just a number, gives us exactly that (see. Fig.2). Information from multiple tables (*postal_areas*,

delivery_routes, *transport_space*) can be accessed and extracted via foreign key routeID.

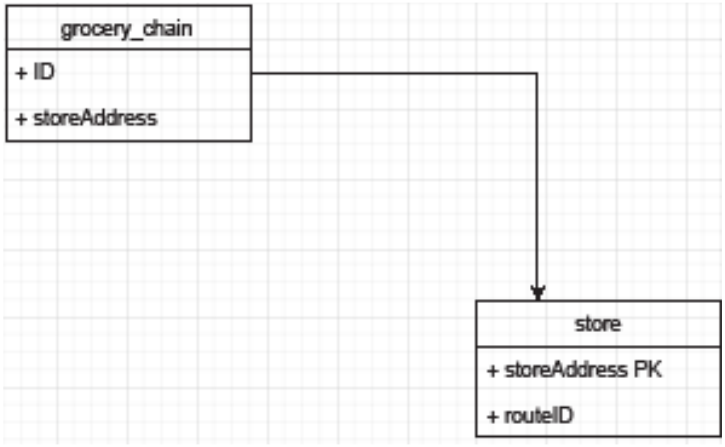

Figure 2: Grocery chain and stores

The *delivery_routes* provides data on which days of the week and what time the route is run. The table *postal_areas* lists all the postal codes included in a certain route. Lastly, *transport_space* gives us a maximum number of order boxes in the delivery van, which will be used when the customer will be making an order.

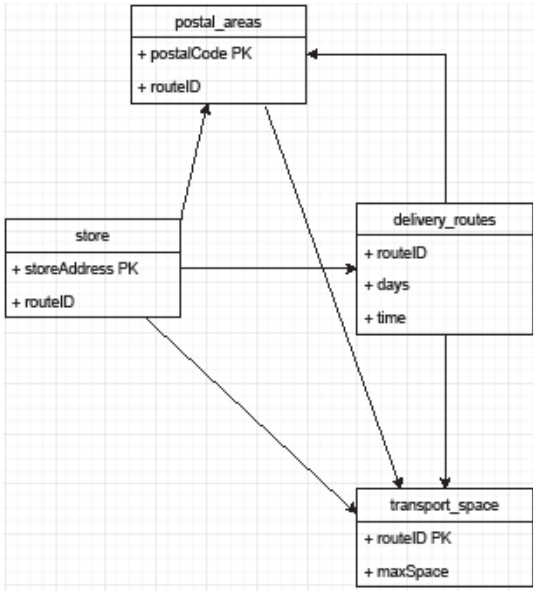

Figure 3: routeID as a foreign key in different tables

As has been explained in the instructions, each store has a separate assortment, which is handled inside of the *store_assortment* class. For each store (identified by the address of the store) the table contains every offered product. These products are recognized by their names - attribute productName, which serves as a foreign key in the *store_assortment* table but is the primary key of the table *product*.

Table *product* consists of several attributes – previously mentioned productName, productQuantity, dietID, storageCond, and availability. Using productName as a foreign key, data from four tables could be accessed: *nutritional_components*, *units*, and *product_price* and *raw_material*. Calorie energy, fat, carbohydrates, proteins are all stored in the attributes of the class, as well as their units. Improvement for the following structure would be to separate energyUnit and othersUnit from the *nutritional_components* into a distinct table. Raw materials, as well as their units (later to be used in *ingredients*) of the product can be extracted from the *raw_material*. The price of the product itself is kept in the *product_price*. Additional attribute validityPeriod is utilized to allow storage of old prices of the products. Finally, since for some products quantity can be indicated in different units, *units* table has been created.

When considering other attributes, at the first glance, productQuantity might seem excessive. However, the usefulness of the attribute becomes apparent when the customer desires to add raw materials (products in fact) from the recipe directly to the shopping basket. Each product has also a special diet linked to it, indicated by dietID, a foreign key to the table *special_diet* (more about it later). Furthermore, the database keeps track of information about storage conditions for a certain product. Since the packaging of the product is directly connected to the storage conditions, the database also maintains information on package type in a separate table *package_type*, data from which can be retrieved via storageCond foreign key.

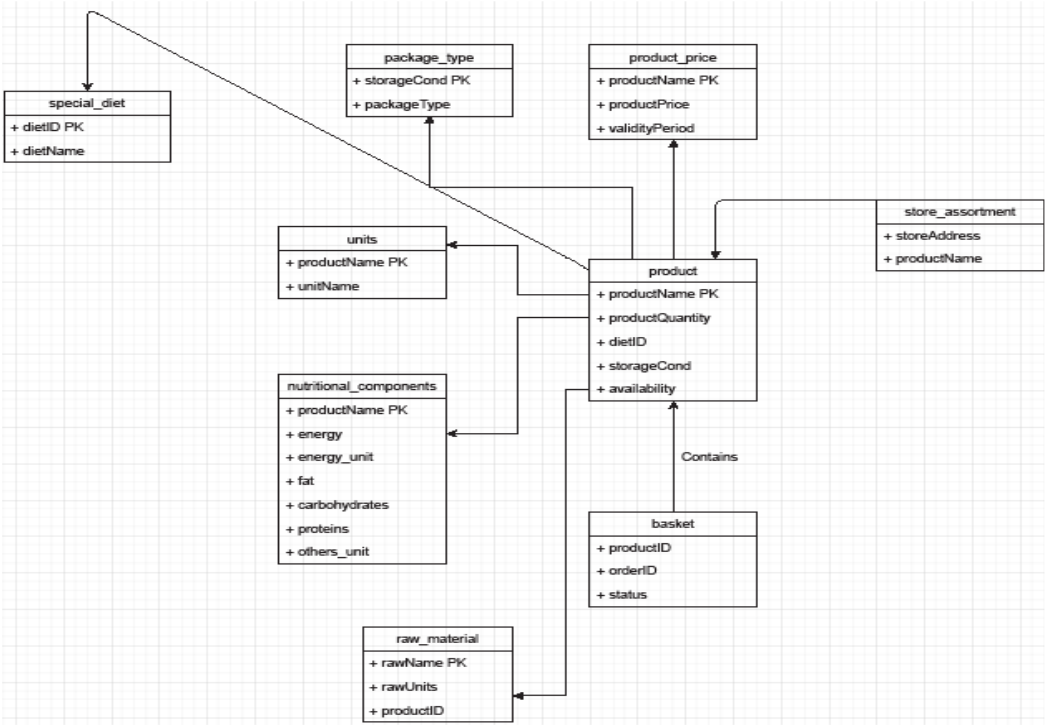

Figure 4: Product and related classes

Now, let us proceed to the principal part of the home delivery orders of the grocery chain – the orders themselves. Database stores all of the orders in the *orders_database* – each order has its own ID and date – keeping old orders to handle possible future complaints. Individual orders can be observed in the table *order*, which we transferred into using the orderID key. Order additionally has customerID and allowSubstitution attributes, to link order to the person and to allow product substitutions, respectively.

As can be seen from the UML diagram, several tables are in a relationship with the *order* class. Firstly, as mentioned earlier, customerID is a foreign key in the table order, which leads us to the *customer* table. Here we have the primary key of customerID (a string of numbers), the name of the customer for convenience when delivering, and the postal code (postalCode) of the client.

Moving on, we also have a *collector* and *collector_order_list* tables. The second one is a list of orders for a particular collector (indicated by his name in the table). The collector is structure maintaining information on collector: name, working days and time and the maximum number of orders which collector can handle. The last one is a valuable piece of information which comes useful when scheduling the orders.

Surely one of the essential parts of the database – *basket* class. The *basket* is a list of products selected by the customer. One attribute which might seem confusing is the 'status'; although it is quite straightforward: collected or not, delivered or not.

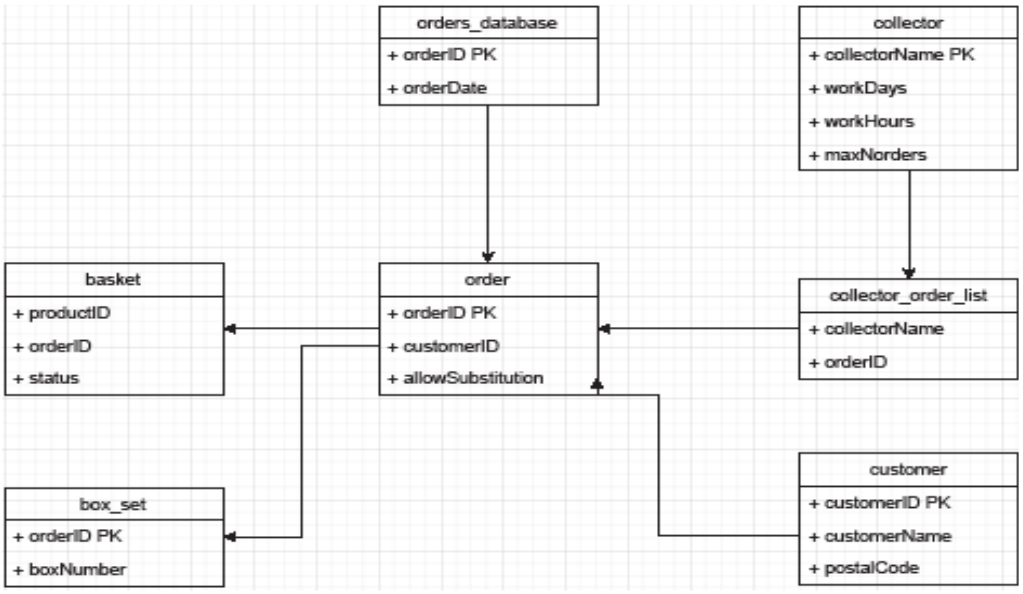

Figure 5: Orders and related classes

Orders kept in boxes while in the transport, and the quantity of the boxes linked to one order is provided in the *box_set* table. Again, the importance of this attribute is apparent when performing operations given in the project instructions.

One last essential section of the database is the recipes section, which introduces two additional tables – *recipe_list* and *ingredients*. Attributes of the *recipe_list* are the name of the recipe, instructions of the recipe and diet linked to this recipe. Using *recipe_name* we transfer to the *ingredients* table, to obtain all of the raw materials of the recipe and their respective quantities.

Solution of required operations

Solutions to some of the operations mentioned in the instructions are presented below.

1) *Search for recipes or products that are suitable for the chosen special diet.*

In order to do that, user needs to get dietID (knowing the dietName) and then easily access *recipe_list* and product table with it. After that extracting recipeName or productName is not a problem.

2) *Make a list for the collector with the products included in the order, their quantities and information on the storage temperature of each product.*

Knowing name of the collector (collectorName) we can obtain orderID (from the *collector_order_list*). After we extracted orderID we can safely proceed to the *basket* table, which has all the products (productID) selected by the customer. Using productID we can retrieve quantity of the product from the *product* table. storageCond is another attribute of this table, hence task is completed.

3) *Look for a replacement product in the store instead of the out-of-stock product and store the information that the product has been replaced by this other product in a certain order.*

Replacement in this database work via accessing the raw materials of the product. Finding in the store_assortment product which has matching raw materials and is not out of stock (and when replacements are allowed) will serve as a substitution for the product. All we need to do is to retrieve info of original and substituted products.

4) *Make a list showing which products have not been delivered at all because they were out of store and could not be replaced with any other product.*

For this specific task we have an attribute status in the table *basket*, helping us monitor which products have been or have not been delivered. In the *product* table we have attribute availability, to check if the product is out of stock or not. Depending on if substitutions were allowed or not, simple SQL query considering all of the previous factors, should be sufficient to retrieve necessary data.

Relational schema

Brief relational schema of the database in text:

grocery_chain (ID, storeAddress)

storeAddress is a primary and foreign key.

- store (storeAddress, routeID)

storeAddress is a primary and foreign key. routeID is a foreign key.

- postal_areas (postalCode, routeID)

postalCode is a primary and foreign key. routeID is a foreign key.

- delivery_routes (routeID, days, time)

routeID is primary and foreign key.

- transport_space (routeID, maxSpace)

routeID is primary and foreign key.

- store_assortment (storeAddress, productName)

storeAddress is a primary and foreign key. productName is the foreign key.

- orders_database (orderID, orderDate)

orderID is a primary and foreign key.

- order (orderID, customerID, allowSubstitutition)

orderID is a primary and foreign key. customerID is a foreign key.

- collector (collectorName, workDays, workHours, maxNorders)

collectorName is a primary key for collector class.

- box_set (orderID, boxNumber)

orderID is a primary and foreign key.

- customer (customerID, customerName, postalCode)

customerID is a primary/foreign key. postalCode is a foreign key.

- product (productName, productQuantity, dietID, storageCond, availability)

productName is a primary and foreign key. dietID, storageCond are foreign keys.

- raw_material (rawName, rawUnits, productID)

rawName is a primary/foreign key. productID is the foreign key

- nutritional_components (productName, energy, energy_unit, fat, carbohydrates, proteins, others_unit

productName is a primary and foreign key.

- units (productName, unitName)

productName is a primary and foreign key.

- package_type (storageCond, packageType)

storageCond is primary and foreign key.

- product_price (productName, productPrice, validityPeriod)

productName is a primary and foreign key.

- ingredients (recipeName. rawName, recipeQuantity)

recipeName is a primary and foreign key. rawName is a foreign key.

- recipe_list (recipeName, recipeInstructions, dietID)

recipeName is primary and foreign key. dietID is a foreign key.

- special_diet (dietID, dietName)

dietID primary and foreign key.

Analysis of the database

Relation R with schema:

R (product, product_price, basket, package_type, units, nutritional_components, raw_material, ingredients, recipe_list, special_diet, box_set, order, orders_database, collector_order_list, collector, customer, store_assortment, store, grocery_chain, delivery_routes, postal areas, transport_space)

Functional dependencies:

product -> special_diet, package_type, units, product_price, nutritional_components, raw_material

special_diet -> recipe_list

recipe_list -> special_diet, ingredients

ingredients -> raw_material

basket -> product

store_assortment -> product

order -> basket, box_set

order_database -> order

collector -> collector_order_list

collector_order_list -> order

customer -> order, postal_areas

store -> store_assortment, delivery_routes, postal areas, transport_space

grocery_chain -> store

postal_areas -> transport_space

delivery_routes -> postal areas, transport_space

$\{\}^+ = \{\}$

Closures:

$\{Product\}^+$ = {product, special_diet, package_type, units, product_price, nutritional_components, raw_material, recipe_list, ingredients, raw_material}

$\{special\_diet\}^+$ = {special_diet, recipe_list, ingredients, raw_material}

$\{recipe\_list\}^+$ = {recipe_list , special_diet, ingredients, raw_material}

$\{ingredients\}^+$ = {ingredients, raw_material}

$\{basket\}^+$ = {basket, product, special_diet, package_type, units, product_price, nutritional_components, raw_material, recipe_list, ingredients, raw_material}

$\{store\_assortment\}^+$ = {store_assortment, product, special_diet, package_type, units, product_price, nutritional_components, raw_material, recipe_list, ingredients, raw_material}

$\{order\}^+$ = {order, basket, box_set, product, special_diet, package_type, units, product_price, nutritional_components, raw_material, recipe_list, ingredients, raw_material}

{order_database}$^+$ = {order_database, order, basket, box_set, product, special_diet, package_type, units, product_price, nutritional_components, raw_material, recipe_list, ingredients, raw_material}

{collector}$^+$ = {collector, collector_order_list, order, basket, box_set, product, special_diet, package_type, units, product_price, nutritional_components, raw_material, recipe_list, ingredients, raw_material}

{collector_order_list}$^+$ = {order, basket, box_set, product, special_diet, package_type, units, product_price, nutritional_components, raw_material, recipe_list, ingredients, raw_material}

{customer}$^+$ = {customer, postal_areas, order, basket, box_set, product, special_diet, package_type, units, product_price, nutritional_components, raw_material, recipe_list, ingredients, raw_material}

{store}$^+$ = {store, store_assortment, delivery_routes, postal_areas, transport_space, product, special_diet, package_type, units, product_price, nutritional_components, raw_material, recipe_list, ingredients, raw_material}

{grocery_chain}$^+$ = {grocery_chain, store, store_assortment, delivery_routes, postal_areas, transport_space, product, special_diet, package_type, units, product_price, nutritional_components, raw_material, recipe_list, ingredients, raw_material}

{postal_areas}$^+$ = {postal_areas, transport_space}

{delivery_routes}$^+$ = {delivery_routes, postal areas, transport_space}

Conclusion

If R were in BCNF, all given closures should contain all attributes of R. Since none of them do, the left sides of the given dependencies are not superkeys of R and R is not in BCNF.

Modifications and supplements to the solution of the first part

Based on the feedback received after submitting the first part of the project, several amendments have been made, specifically to the structure of the home delivery system. In the figure below, an updated version of the UML diagram is presented (see fig.6).

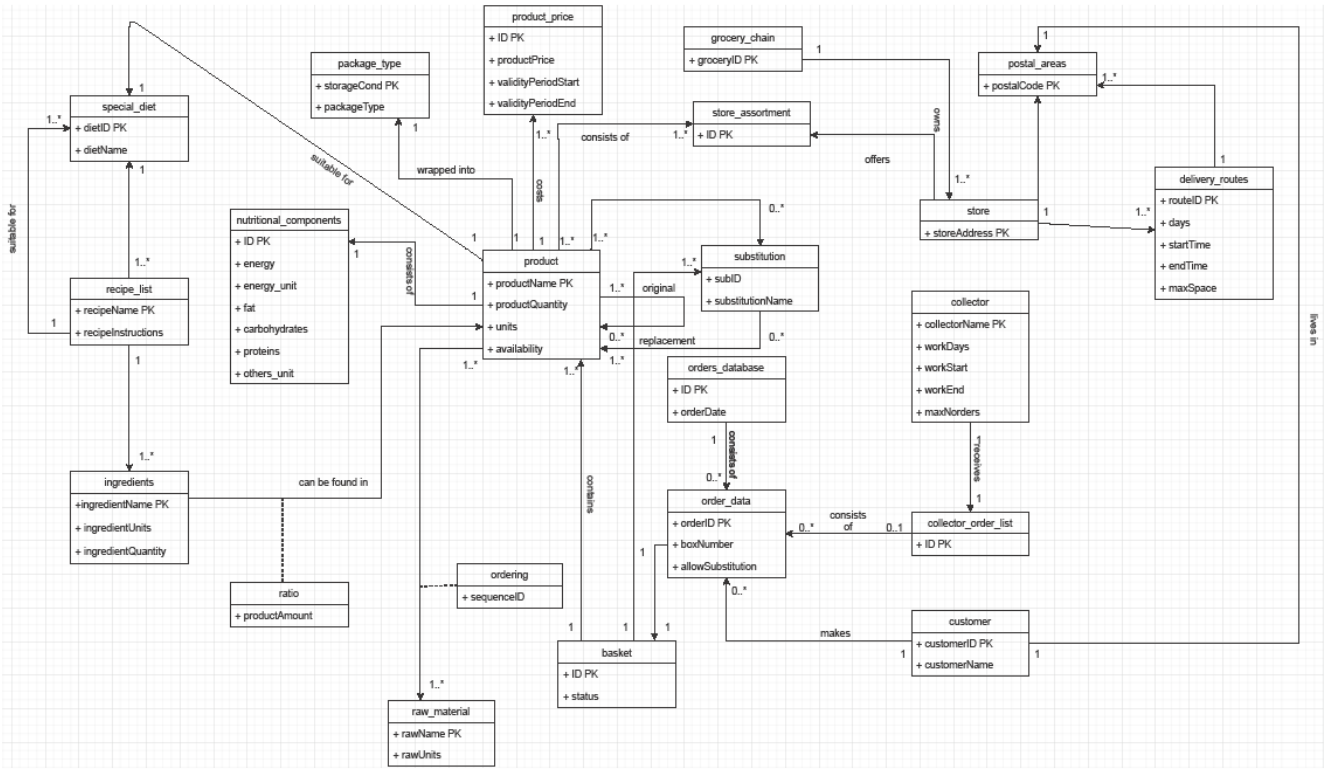

Figure 6: Updated UML of the delivery system

Diving into the alterations, one of the apparent changes of the diagram is the removal of the 'redundant' class attributes. As pointed out in the feedback, in UML notation, a class cannot have an attribute that happens to be the key of another class. Now, this information is indicated in the diagram by the means of associations.

Major changes have been made to the *product* class. Previously, replacing product information has been extracted using similarities in the raw materials of the products. New structure performs this operation utilizing self-association and additional *substitution* class. There are two associations between *substitution* and *product* class – one displaying the replacing product and the other standing for the product to be replaced. Logically, substitution has to be connected to the *order* class, which implemented via the association of *substitution* to the *basket* class.

Furthermore, an association between *ingredients* and *raw_material* tables has been removed. Instead, we append association between *product* and *ingredients*. In terms of use case, a connection is utilized when searching for the necessary ingredient in the list of the products. Since ingredientQuantity and productQuantity attributes might indicate different values, *ratio* association class is introduced (to determine the amount of product needed for the recipe).

One functionality that has not been implemented in the first part of the project is the order of the raw material in the product information. This feature is now implemented through association class *ordering*, which contains sequenceID attribute indicating the order of specific raw material in the product (see Fig.7).

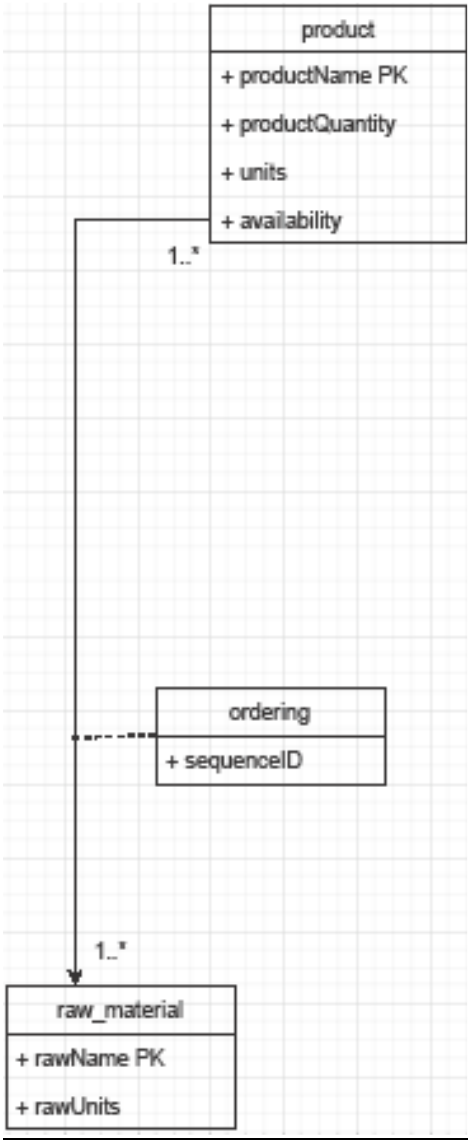

Figure 7: Classes raw*_material*, *product,* and association class *ordering*

Considering that, below is the updated variant of the relational schema of the home delivery system.

Relational schema:

- grocery_chain (groceryID, storeAddress)

storeAddress – Foreign Key (FK)

- delivery_routes (routeID, days, startTime, endTime, maxSpace)
- store (storeAddress, routeID)

routeID - FK

- postal_areas (postalCode, routeID)

routeID - FK

- customer (customerID, customerName, postalCode)

postalCode - FK

- order_data (orderID, boxNumber, allowSubstitution, customerID)

customerID - FK

- collector (collectorName, workDays workStart, workEnd, maxNorders)
- collector_order_list (ID, collectorName, orderID)

collectorName, orderID - FKs

- orders_database (ID, orderID, orderDate)

orderID - FK

- basket (ID, orderID, productName, status)

orderID, productName - FKs

- product (productName, productQuantity, units, availability, storageCond, dietID)

storageCond, dietID, substitutionName - FKs

- substitution (subID, substitutionName, productName)

productName – FK, subID and substitutionName – composite key

- product_price (ID, productPrice, validityPeriodStart, validityPeriodEnd, productName)

productName -FK

- package_type (storageCond, packageType)
- store_assortment (ID, storeAddress, productName)

storeAddress, productName - FKs

- nutritional_components (ID, energy, energy_unit, fat, carbohydrates, proteins, others_unit, productName)

productName - FK

- recipe_list (recipeName, recipeInstuctions, dietID)

dietID - FK

- special_diet (dietID, dietName)
- ingredients (ingredientName, ingredientUnits, ingredientQuantity, recipeName)

recipeName - FK

- ratio (productAmount, productName)

productName - FK

- ordering (sequenceID, rawName)

rawName - FK

- raw_material (rawName, rawUnits, productName)

productName – FK

Creating SQL Database

In this part of the project, the SQL version of the planned database is created. When creating the database, primary and foreign keys have been indicated, as well as other constraints including NOT NULL statements. In addition to that, some attributes required AUTOINCREMENT parameter. The principal section of the database creation is selecting proper data types for the table attributes. Below we display examples and corresponding explanations from the final solution.

To facilitate the usage of the 'time' attribute in the *delivery_routes*, it was split into two new attributes: startTime and endTime with the data type TIME. The attribute maxSpace, which previously was a part of the *transport_space* class is now one of the columns in the *delivery_routes* (unnecessary class *transport_space* has been removed).

Attribute *postal_code* is of data type VARCHAR (5). Character number has been limited to 5 ('00370', '02150') to resemble Finish postal format. A similar data type

has been chosen for the orderID – VARCHAR (8), an example of such an attribute being 'OID69686'.

Attributes of some tables, such as allowSubstitution in *order_data*, or availability in *product*, utilize Boolean data type to check if customer allowed replacements in his order, and if a product is available in the store, respectively.

Such tables as *orders_database* and *collector_order_list*, and plenty of other classes as well, have auto-incremented ID as their primary key, which is logical since both of the tables serve as a list for specific purposes.

Table *product*, one of the most important classes of the database is in association with multiple different classes, which is the reason it is being indicated as a foreign key in the attributes of those classes. Table *substitution*, one of such classes, is having productName as its foreign key and subID and substitutionName as the composite key.

Two association classes are also implemented in the database: *ordering* and *ratio*. The purpose of those tables has been explained before. Apart from that, SQL code of the database is simply collection of CREATE TABLE commands with specific primary and foreign keys. In order to proceed with the solution to the rest of the tasks (indexing, views, use cases) several INSERT INTO commands have been made, to fill the database with data (see .sql file)

Views and indexes

Several indexes have been created in order to quickly locate data, which might be frequently accessed – either by developers or clients. Thereby we have indexes for the productName column, to search necessary products from the stock. The picture below is the SQL query of this index (see fig.8).

```
CREATE UNIQUE INDEX product_index ON product (
    productName
);
```

Figure 8: productName index

Another example of the index is the productPrice index, which is one of the most important columns, at least, from the client's perspective (see fig.9).

```
CREATE UNIQUE INDEX product_price_index ON product_price (
    productPrice
);
```

Figure 9: productPrice index

Employers might need a full list of their collectors; therefore, it would be useful to have easy access to this piece of data (see fig.10).

```
CREATE UNIQUE INDEX collector_index ON collector (
    collectorName
);
```

Figure 10: collectorName index

Finally, postal_codes_index is introduced (see fig.11). The information about most ordering postal areas might be retrieved via counting distinct postal codes in the column, which might be beneficial to the store owners.

```
CREATE UNIQUE INDEX postal_codes_index ON customer (
    postalCode
);
```

Figure 11: postal_codes_index

Next, the views are presented. One of the views is the combination of the previously mentioned columns – productName and productPrice. Use case would be to observe prices of the offered products. Second view allows clients to find recipes for a specific diet (dietName and recipeName).

```
SELECT product.productName, product_price.productPrice
FROM product
JOIN product_price
ON product.productName == product_price.productName;

SELECT recipe_list.recipeName, special_diet.dietName FROM recipe_list
JOIN special_diet
ON recipe_list.dietID == special_diet.dietID;
```

Figure 12: Views

Use cases

Several use cases have been created on the basis of our home delivery system in order to showcase workability of the associations and created tables. Most of the use cases have been formulated on the basis of the possible operations from instructions to the first part of the project.

1) Search for recipes or products that are suitable for the chosen special diet (see fig.13)

```
SELECT productName
FROM product
JOIN special_diet
ON special_diet.dietID = product.dietID
WHERE special_diet.dietName = 'Low-Carb';
```

	productName
1	Egg
2	Bread

Figure 13: Products for a special diet

Since a view for the case of recipes has been implemented earlier, in this situation we create query which chooses products for a specific diet ('Low-Carb').

2) Retrieve information on old orders already shipped.

```
SELECT orderID
FROM order_data
JOIN orders_database
ON order_data.orderID == orders_database.orderID
WHERE order_data.customerID = '786667'
AND orders_database.orderDate LIKE '2018-08-30%';
```

	orderID
1	OID76921

Figure 14: Acquiring ID of an old order

Using ID of the customer and knowing date of the order, it is possible to obtain the ID of the order (see fig.14). Later, if necessary, this data can be used to retrieve product list, prices, etc.

3) Make a list for the collector with the products included in the order, their quantities and information on the storage temperature of each product.

Each collector has a list of specific orders assigned to him and each of those orders consists of different products with respective quantities and storage conditions (for transportation). The query was designed to handle this situation (see .sql file)

```
SELECT productName, productQuantity, product.units, storageCond
FROM product
JOIN basket, order_data, orders_database
ON product.productName == basket.productName
AND basket.orderID = order_data.orderID
AND order_data.orderID = orders_database.orderID
WHERE order_data.customerID= '786667'
AND orders_database.orderDate LIKE '2018-08-30%';
```

Figure 15: Query generating the product list for collector

Below are the results of the query. The list for collector consists only of one product, but all other required data is perfectly displayed.

	productName	productQuantity	units	storageCond
1	Chicken	5	kg	Fridge

Figure 16: Results of the previous query

4) Investigate which products in the order have already been collected and what still needs to be collected.

One of the simpler queries. For a specific order we create a list of products with their status displayed - 'collected' or 'not collected'. Figure below is the screenshot of this query (see fig.17).

```
SELECT productName, status
FROM basket
WHERE basket.orderID = 'OID69686'
AND basket.status = 'Collected' OR basket.status = 'Not collected' ;
```

Figure 17: Products already collected and not

5) Calculate the total price of the order, taking into account possible product replacements, discounts, and products not delivered at all.

As opposed to the previous query, this situation requires complex filtering. First of all, a list of product prices for particular order has to be extracted. After that, availability is examined. In case the product is available in stock, the price of the product will be counted in the total price of the order. In the situation where the product is not available, it has to be substituted (if a customer has allowed replacements in the order). After that, all available replacements for the product are extracted and the first one is chosen from the list. The validity period of the price is checked: in case date of the order is earlier than the end of the validity period, then the current price is calculated. The procedure is repeated for all of the products in the basket of the order and the total price is summed up.

```
SELECT SUM(productPrice)
FROM product_price
JOIN product, basket, orders_database
ON product.productName == product_price.productName
AND product.productName == basket.productName
AND orders_database.orderID == basket.orderID
WHERE
(product.availability IS TRUE
OR
product.productName IN
(SELECT productName
FROM product
WHERE productName IN
(
SELECT substitutionName
FROM substitution
JOIN basket, order_data
ON basket.productName = substitution.productName
WHERE basket.orderID = 'OID56604'
AND order_data.allowSubstitution = 1
)
)
)
AND basket.status = 'Delivered'
AND basket.orderID = 'OID56604'
AND orders_database.orderDate <= product_price.validityPeriodEnd;
```

Figure 18: Calculating total price of the order

```
INSERT INTO basket (orderID, productName, status)
VALUES  ('OID69686', 'Milk', 'Collected'),
        ('OID56604', 'Bread', 'Delivered'),
        ('OID76921', 'Chicken', 'Not delivered'),
        ('OID69686', 'Bread', 'Not collected');

INSERT INTO basket (orderID, productName, status)
VALUES  ('OID56604', 'Chicken', 'Delivered'),
        ('OID56604', 'Milk', 'Delivered'),
        ('OID56604', 'Milk', 'Not delivered');
```

```
INSERT INTO product_price (productPrice, validityPeriodStart, validityPeriodEnd, productName)
VALUES  (7.5, '2019-07-09', '2020-12-30', 'Chicken'),
        (2.4, '2019-10-01', '2020-12-30', 'Milk');
```

Figure 19: New data inserted in the database

6) Storage conditions for a specific product from particular order

Below is a simple query for deriving storage conditions for specific order. The information might be found useful to collector who needs to know how to store products while transporting them.

```
SELECT basket.orderID, product.productName, product.storageCond FROM basket
JOIN
product ON basket.productName == product.productName;
```

Figure 20: Storage conditions query

7) Acquiring product information for every product offered

Clients might be interested in the product information while ordering them

```
SELECT energy, energy_unit, fat, carbohydrates, proteins, others_unit, product.productName FROM nutritional_components
JOIN
product ON product.productName == nutritional_components.productName;
```

Figure 21: Product information query

8) Obtain list of ingredients for a certain recipe

Since our database is designed so that there would be variety of recipes offered in it, clients might be interested in more detailed information about recipes of interest. Therefore, we query list of ingredients for a recipe 'Zoodles'.

```
SELECT ingredientName
FROM ingredients
WHERE recipeName = 'Zoodles';
```

Figure 22: Ingredients list for 'Zoodles'

9) Make list showing which products in the order have been replaced and with which product it has been replaced

```
SELECT order_data.orderID, basket.productName AS 'Replaced',
substitution.substitutionName AS 'Replaced with'
FROM basket
JOIN substitution
ON basket.productName == substitution.productName
JOIN order_data
ON basket.orderID == order_data.orderID
WHERE order_data.allowSubstitution == 1
GROUP BY basket.productName;
```

Figure 23: Replacements of the product

10) Calculate number of customers during specific period

Being owner of the grocery chain, you might want to know performance of each store in the chain. In case there is a specific store of interest, the query below will output number of customers in certain period of time.

```
SELECT COUNT(DISTINCT customerID)
FROM order_data
JOIN orders_database
ON orders_database.orderID = order_data.orderID
WHERE orderDate > '2019-08-30'
AND orderDate < '2020-12-29';
```

Figure 24: Amount of customers

11) Order delivery information

This query provides list of collectors for all the orders in the system.

```
SELECT order_data.orderID, collector_order_list.collectorName,
postal_areas.postalCode, postal_areas.routeID
FROM collector_order_list
JOIN order_data ON collector_order_list.orderID == order_data.orderID
JOIN customer ON order_data.customerID == customer.customerID
JOIN postal_areas ON customer.postalCode == postal_areas.postalCode
JOIN delivery_routes ON delivery_routes.routeID == postal_areas.routeID
```

Figure 25: Order, respective collector, postal code, and route

12) Extract orderID and number of boxes for specific collector

```
SELECT orderID, boxNumber
FROM order_data
JOIN collector_order_list
ON collector_order_list.orderID = order_data.orderID
WHERE collectorName = 'Dias';
```

Figure 26: Order and number of boxes included in the order