The Technical University of Denmark

Database Systems, 02170

Database of an Online Shop

Authors:

s144968 - Mikkel Grønning

s146996 - Christian D. Glissov

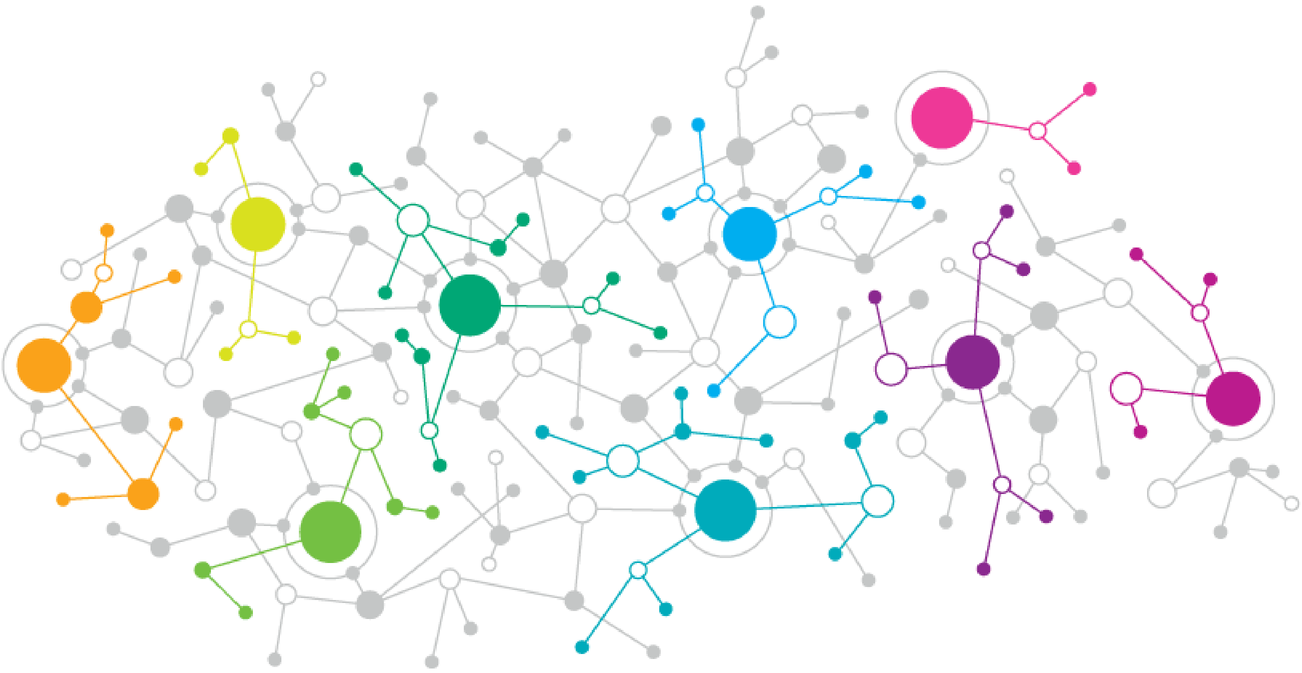
s164172 - Kamilla Skafte Bonde

s164207 - Melina Gharajehbarhagh

Supervisors:

Anne Elisabeth Haxthausen

Charlotte Friis Theisen



April 4, 2020, Kongens Lyngby

Group: Charlotte1

DTU Compute

Department of Applied Mathematics and Computer Science

DTU

Table of Contents

1 Statement of Requirements **1**

2 Conceptual Design **2**

2.1 Entities and relationships 2

2.2 Relationships, participation and cardinality 4

3 Logical Design **6**

4 Normalization **10**

4.1 Functional Dependencies (FDs) 10

4.2 First Normal Form (1NF) 11

4.3 Second Normal Form (2NF) 11

4.4 Third Normal Form (3NF) 12

4.5 Boyce-Codd Normal Form (BCNF) 12

5 Implementation **12**

5.1 Create Database 13

5.2 Create Table 13

5.3 Create Views 14

6 Database Instance **15**

6.1 Insertion of data 15

6.2 Display of instances of all tables and views 15

7 SQL Data Queries **17**

7.1 First select statement 17

7.2 Second select statement 18

7.3 Third select statement 18

8 SQL Table Modifications **19**

8.1 SQL UPDATE 19

8.2 SQL DELETE 21

9 SQL Programming **22**

9.1 Functions 22

9.2 Procedures 24

9.3 Transactions 24

9.4 Triggers 26

9.5 Events 27

Responsibilities

Everyone has contributed equally in the making of this report, however certain people are responsible for certain sections. At times there were several pieces of code made by different people, in this case there might be more than one responsible author. These are as mentioned:

Content	Responsible author
Title page	Christian
1	Christian
2	Christian
2.1	Christian & Mikkel
2.2	Christian
3	Kamilla & Melina
4	Mikkel
4.1	Melina
4.2	Kamilla
4.3	Melina
4.4	Melina
4.5	Melina
5	-
5.1	Mikkel
5.2	Mikkel
5.3	Christian
6	-
6.1	Mikkel
6.2	Mikkel
7	-
7.1	Christian
7.2	Christian
7.3	Christian
8	Mikkel
8.1	Christian & Kamilla
8.2	Melina & Christian & Mikkel
9	-
9.1	Christian & Melina
9.2	Mikkel
9.3	Kamilla
9.4	Kamilla
9.5	Kamilla

1 Statement of Requirements

This report contains the mandatory project for the course 02170 Database Systems. The project will illustrate the creation of a database system by modeling a fictive small scale online retailer. Let us name the online shop BSOS. The retailer specializes in selling clothing brands at an affordable price in Denmark. All products are sold online and stocked in a local facility. To efficiently run the shop, the client has ordered a database system to efficiently store the generated information from a growing customer base. The database will model the ordering and rating system of the online shop.

Based on a meeting with the client, the ordering system of BSOS is fairly simple. The `customer` register their personal information before making an order. The *address*, *zip code*, *city*, *name*, *phone number*, *e-mail* and whether or not they would like to subscribe to *newsletters*.

When an `order` is made it should be stored and also the information about the order, such as *date* of purchase, *shipping*, *payment information*, *approval* of the order. Approval enables the customer to see when the order has been registered by the employees of BSOS.

The customer can order several items in a single order, an order is, therefore, a basket of `ordered items`. The item has a *quantity* of the item ordered by the customer. It should be easy to view the items of an order.

The client has a *stock* of different `products` available for the customer to buy. To give an overview of the products and keep track of different attributes of the products, such as if it is in *stock*, *brand*, the *name* or description of the product and *type*. The client usually receives a new batch of clothes every month and likes to keep track of old batches. The base price should be adjustable by the client.

Sometimes, the client settles for a `discount` on certain products. The discount only lasts for a *period* of time.

Finally, the client is planning to implement a `rating` system of the products for the customers. The client wants to have two ratings, a rating based on the *fit* and *quality* of the products as well as a written review. It should be possible to query the entity enabling one to find the average rating of a product, but also to find the reviews of a customer.

2 Conceptual Design

The entity-relationship (ER) diagram, visible in Figure 1, displays a sketch of the database structure model of BSOS the online shop. In Figure 1 is the cardinality, participation and relationship between entities illustrated. This information is also summarized in Table 1. In the upcoming part each relationship will be described and arguments for the choices will be made.

ER Diagram	Cardinality constraint	Participation	Relationship	Relationship Attributes
Product-to-Order	many-to-many	partial, total	OrderItem	Quantity
Order-to-Customer	many-to-one	total, total	Buyer	None
Product-to-Customer	many-to-many	partial, partial	Rating	Review, Fit, Quality
Product-to-Deals	many-to-one	partial, total	Get Discount	None

Table 1: Table showing related information of the ER diagram.

2.1 Entities and relationships

- **Product** [Entity]: The product entity contains information about the stocked products of the business. *product_date* is the date of which the product is stored. It has a primary key **product_id**. *product_type* is the type of the product, as it is a database modeling a clothing business, types could range from "t-shirt" to "jeans". Other attributes are *product_name*, *product_price*, *product_stock* and *product_brand*. Brand indicates the clothing brand, such as, "Adidas" or "Nike". Stock tells the client whether the product is available or not and how many items of that specific product is in stock.

- **Deals** [Entity]: The deals entity models discounts of products and also tells when the discount starts and when it ends by the following attribute *discount* and composite attribute *date* containing *starts* and *expires* of the discount. It has a primary key *deal_id*.

- **Order** [Entity]: The order entity is used to store orders from the customers and to act as an identifier entity for OrderItems. The primary key is *order_id*. Furthermore, it contains *order_date*, when the order was made. *order_approval* telling if the order is approved, such as if it is in stock. *order_shipped* indicating when the order is shipped and finally the payment info of the customer *order_payment_info*, such as "mastercard" or "mobilepay".

- **Customer** [Entity]: The customer entity stores information about the individual customers. A customer is defined as an individual who has made at least one order. It stores two composite attributes, indicating the full name and the address of the customer. Furthermore phone number, e-mail and a Boolean based on whether the customer want to subscribe to newsletters or not.

- **Get Discount** [Relationships]: This relationship links the Product entity with a Deal entity. If a product has a deal the discount can be found through the relationship.

- **OrderItems** [Relationships]: The OrderItems relationships contains the products that a customer has selected for purchase stored in the order entity. The OrderItems relationships has Quantity as attributes, indicating quantity of the ordered item by the customer. This is added in order to keep track of number of the same product a customer add to their basket. It keeps redundancy low as multiple rows in the database is potentially just one row if it is the same product the customer picks.

- **Rating** [Relationships]: The rating relationships connects the product entity with reviews from the Customer entity. The Rating relationships then contain a *review*, *quality* and *fit* attribute allowing the customer to write a review for a product and grade the fit and quality on scale from 1-5.

2.2 Relationships, participation and cardinality

- **Product** to **Order**
  The Product-Order relationship describes the product that a customer has selected for purchase. The relationship between Product and Order is many-to-many of the entities. The reason is an order can contain many different products and in order for an order to make sense the order need to contain at least 1 product indicating a total partition for orders. Similarly, a single product might map to many orders, however a product might not have an order, giving a partial participation for products.

- **Order** to **Customer**
  In the database system it is assumed that in order to create a customer at least one order have to be made. Thus the first time a customer order a product, the customer is added to Customer table. Once the customer is added the same customer can create multiple orders, but an order must only map to one customer. This describes why the cardinality of Order and Customer is many-to-one. An order has to be associated with a customer thus the total participation of Order, the same goes for Customer, as a customer is assumed to have made at least one order. The relationship "buyer" is simply connecting the customer and the order by using customer ID for Order.

- **Customer** to **Product**
  The customers have the option to rate the products thus the relationship, `Rating`. The relationship is many to many with partial participation of Customer, it is optional to give a review, so some customers might not give a rating. The same customer can write many different reviews for many products, giving the many cardinality. The product only has partial participation as a product does not necessarily need to have a rating but a single product might also have mulitple ratings from different customers, once again giving the many cardinality.

- **Product** to **Deals**
  It has been decided that a product can have at most one deal, but a deal might map to many products, such as if there is a sales campaign being promoted by the company, giving a many-to-one cardinality for the relationship. A product might have no deals, but a deal must be set on a product if defined, leading to a partial participation of the product, but a total participation of the deal.

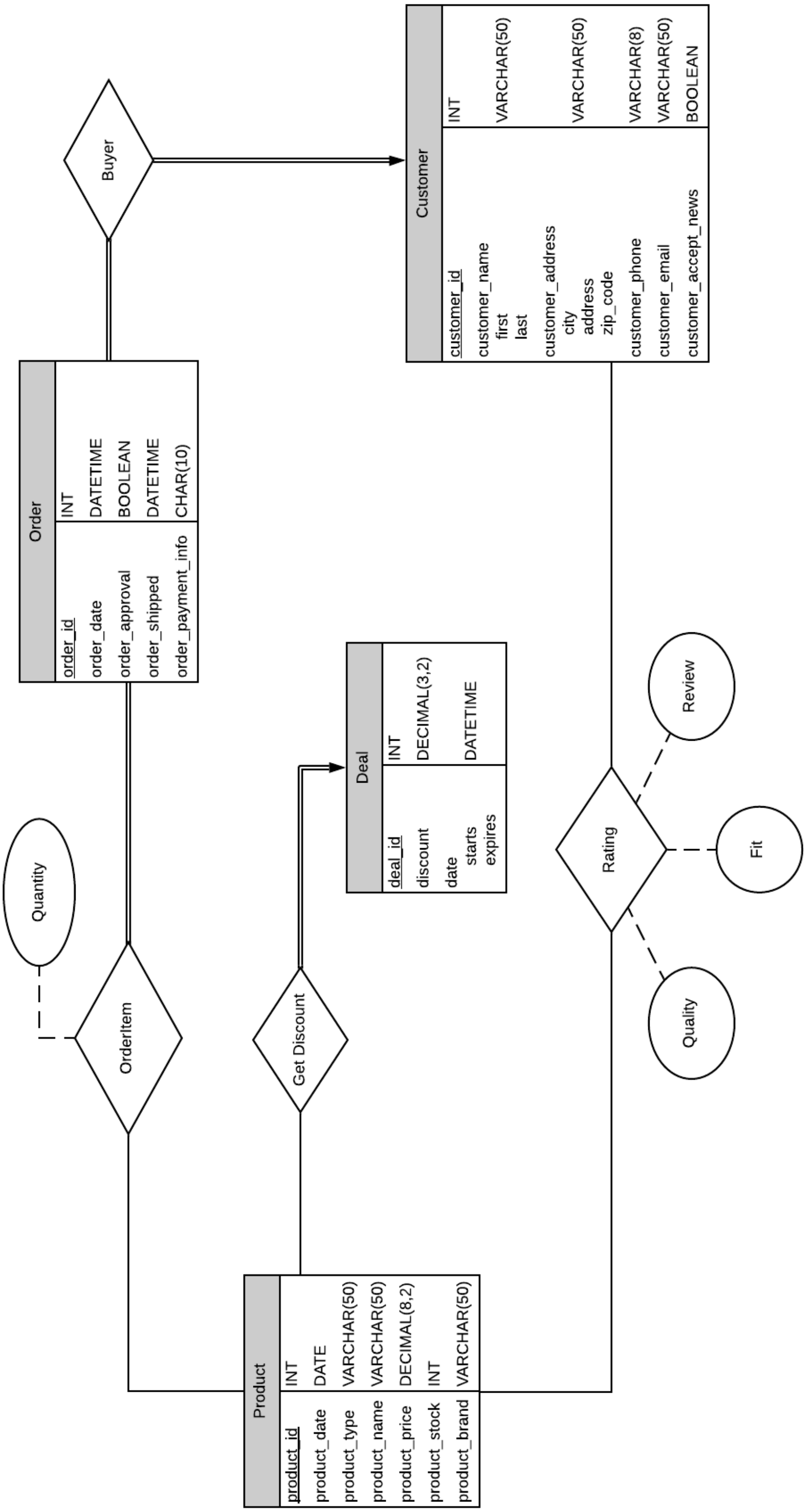


Figure 1: The figure displays the entity relationship diagram for the online shop BSOS.

3 Logical Design

This section describes the phase of constructing the logical model of the database based on entity set in Figure 1. This achieved by starting to convert the full entities set into relations schemes with a primary key and the same attributes. Subsequently, partial entities are transformed into relationship schemas with both primary and foreign keys. The many-to-many relationships are converted into relationship schemas with the primary key of entities that forms the relationships with the same attributes from the entity-relationship diagram, e.g. the `OrderItem` relationship turns into an entity on its own in the logical design. The one-to-one relationship turns into relationship schemas by adding the primary key from one entity as the foreign key of the other schema. Lastly, the many-to-one relationships are transformed by including the attributes from the relationship and the primary key from the one-side to the many side. I.e. The primary key from the one side become the foreign key on the many side. In Figure 2 can the relational database schema for BSOS web-shop be seen.
In the list below will each table from the logical design is described.

- **Customer**
  `Costumer` is a table where the details for each costumer is stored. The information is stored in the attributes *costumer_first_name*, *costumer_last_name*, *customer_address_street*, *customer_zipcode*, *customer_phone*, *customer_email* and *customer_accept_news* while the *costumer_id* is the primary key.
  The attribute *customer_accept_news* describes whether the costumer has accepted to receive news by email where 1 corresponds to new being accepted and 0 to news being declined. It should be noted that it is assumed the email is not necessarily unique for each costumer, so a family can have different profiles while using the same email.

  The foreign key *customer_zipcode* refers to the table `City`, where the corresponding city is stored. This ensures that the costumer has to enter a valid zip code.
  customer_address_street contains both street, street number and floor/apartment number in one string, instead of dividing the address into separate attributes. For a large database it may be preferable to divide it into separate attributes to ensure compatibility when the data exchange with delivery companies is needed.

- **Product**

  `Product` is a table consisting of the attributes *product_ id*, *product_ date*, *product_ type*, *product_ name*, *product_ price*, *product_ stock* and *product_ brand.* The primary key is the unique *product_ id.*

  The attribute *product_ date* is the date for which the company started selling the product. For simplicity there is no end date for expired products. These would just have a zero stock or eventually be removed from the table. *product_ type* denoted the type of apparel e.g. jacket, pants etc. *product_ name* and *product_ brand* is self-explanatory, as they contain the name and brand of the product. The price and stock of the product is stored in *product_ price* and *product_ stock* respectfully. The table is simplified such that if a the stock or the price changes, the respectful row is altered in the table. For future expansions one could expand the table to contain historic values for the stock and price where each row in the table would have a *valid_ from* and *valid_ to* date.

- **Order**

  `Order` is the table which stores information of any order made in the system. The table's primary key is *order_ id* and contains the attributes *order_ date*, *order_ approval*, *order_ shipped* and *order_ payment_ info.*

  The foreign key *costumer_ id* connects a given order to the corresponding costumer.

  The attributes *order_ date* and *order_ shipped* tells which date and time the order was placed and shipped respectively. The payment method used is stated in *order_ payment_ info.* In reality a transaction table would be included as there is a payment for each approved order, but for simplicity it is omitted from the database.

- **OrderItem**

  `OrderItem` consists of the composite primary key composed of *order_ id* and *product_ id.* Thus for each product in a given order, there will be a unique row in `OrderItem`, for which one can retrieve the quantity that has been bought of that specific product. This is store in *order_ item_ quantity. product_ id* refers to the Product table while *product_ id* refers to the `Order` table. Thus `OrderItem` connects the two tables.

- **Rating**
  Rating is a table where the composite primary key is composed of *product_ id* and *customer_ id*. *product_ id* is a foreign key referring to the `Product` table and *customer_ id* is foreign key to the `Customer` table. Alongside these, we have the attributes *rating_ quality*, *rating_ fit* and *rating_ review*. This means that the reviews are simplified such that each customer can only review one product one time. If a customer chooses to review the product again, the old review is overwritten.

- **Deal**
  `Deals` is a table including the primary key *deal_ id* and the attributes *discount*, *deal_ starts*, *deal_ expires*. The primary key *deal_ id* is referred to by the `Product` table, such that if a product has a *deal_ id*, one can find the *discount* in `Deals` and ensure its validity from the dates.

- **City**
  `City` is a table consisting of the primary key *customer_ zipcode* and the attribute *city_ name*. The purpose of this table is, that one can retrieve a city name given the zip code, which is why *customer_ zipcode* in the `Customer` table references the `City` table.

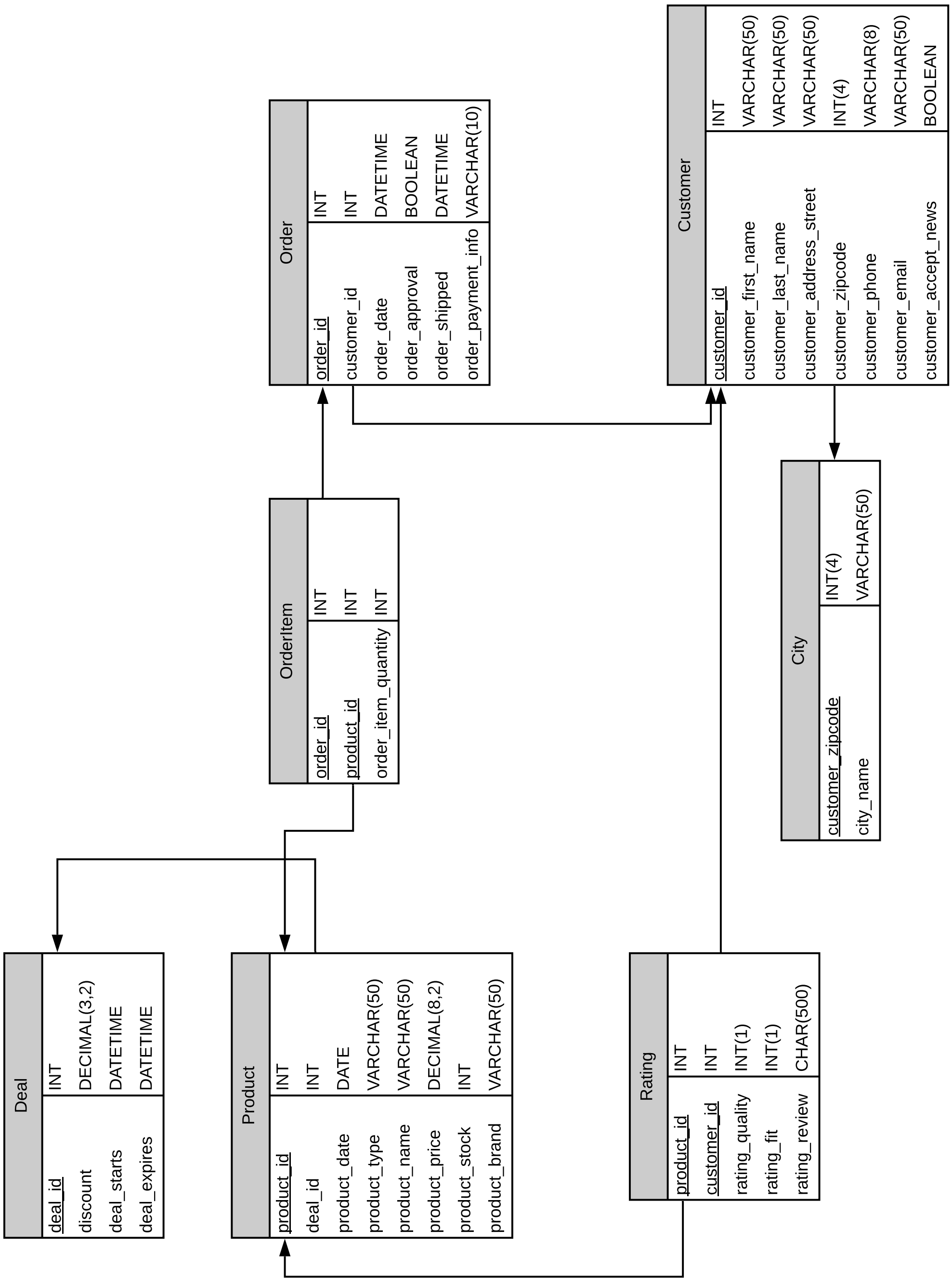


Figure 2: Database Schema Diagram

4 Normalization

Normalization is a technique used to minimize redundancy in the data usually by splitting tables up into smaller tables. When altering tables with redundant data, you may end up only altering some of the data which can lead to modification anomalies. This can also be eliminated by normalizing the database. In the following sections you will be introduced to the different degrees of normalization and its hierarchy. To do this, you first need to be introduced to *functional dependencies*.

4.1 Functional Dependencies (FDs)

If for a relation R, a set of attributes B can be determined by a set of attributes A, then B is functionally dependent on A.

Person	Zipcode	City
A	8000	Aarhus
B	2300	Amager
C	2300	Amager
D	4874	Gedser
E	8000	Aarhus

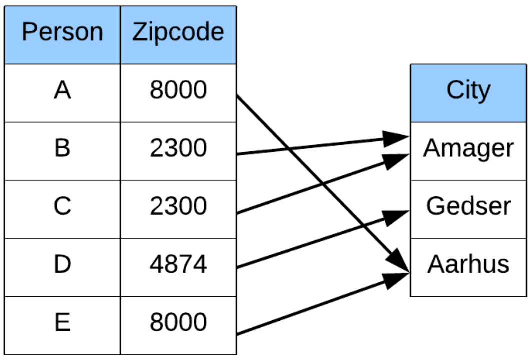


Figure 3: On the left a table with attributes $\{Person,\ Zipcode,\ City\}$ is shown. Since *City* can be determined by *Zipcode*, it is said that *City* functionally determines *Zipcode*. There is a *many-to-one* relation as can be seen to the right. Note that one could also say, that *Zipcode* determines *City* because there is a *one-to-one* relation between the two.

As can be seen in Figure 3 some information repeats itself, which creates redundancy. Therefore, by knowing the functional dependencies between attributes or sets of attributes in a relation, one can eliminate these.
Now, consider the table OrderItem. What are the functional dependencies of this table? The table consists of three attributes: *order_id*, *product_id* and *order_item_quantity* (denoted *quantity* for simplicity), which gives rise to $2^3 = 8$ subsets of attributes which can be combined in roughly $8^2 = 64$ ways to test for FDs, many of which are trivial. Some of them can be seen in Table 2.

No.	Determinant	Dependant	Validity	Remark
1	{order_id}	{order_id}	Legal	Trivial
2	{order_id}	{product_id}	Illegal	
3	{order_id}	{quantity}	Illegal	
4	{product_id}	{quantity}	Illegal	
5	{order_id, product_id}	{quantity}	Legal	
6	{order_id, quantity}	{product_id}	Illegal	
7	{product_id, quantity}	{order_id}	Illegal	

Table 2: Test of functional dependency (FD) for the table OrderItem. The only non-trivial legal FD is in No. 5, since one can uniquely determine the *quantity* given the *order_id* and *product_id*.

4.2 First Normal Form (1NF)

To ensure the entity table is in first normal form, all attributes must be atomic and depend on the key.
As an example the table `Costumer` is used. As described the functional dependencies are found, to find the possible candidate keys. Again, only one valid (non-trivial) key is present in the table i.e. the *costumer_id*. Hence, all attributes are dependent on the key and satisfies the one rule.
Now, only one attribute can be split, but though the *costumer_address_street* contain both street number and apartment floor it can still be considered atomic, since it is unique for the costumer. As both rules apply to all tables they are all 1NF.

4.3 Second Normal Form (2NF)

In order for a relation schema to be in Second Normal Form it must first be 1NF. Furthermore each non primary key attribute must depend on the entire primary key. Note that in the special case that the relation schema is 1NF and there is only one attribute in the primary key, then the relation schema is also 2NF. Therefore `Deals`, `Product`, `Orders` and Customer are already 2NF. The primary key in `Rating` is a composite of {*product_id*, *customer_id*}, but since each rating is unique per customer and item, it means that *rating_quality*, *rating_fit* and *rating_review* depend on the entire key, and thus it is 2NF. `OrderItem` is also 2NF by same reasoning, which can also be seen in Table 2 where *quantity* is dependent on the entire key {*order_id*, *product_id*}. Hence all the tables are 2NF as well.

4.4 Third Normal Form (3NF)

In order for a relation schema to be on the Third Normal Form, it must firstly be 2NF. Furthermore, it must hold, that each non primary key attribute must depend directly on the entire primary key. This means that no non primary attribute can depend on some other non primary attribute or alternatively on parts of the key.
Since all the tables are 2NF and there are no relation between the non primary attributes, they must simply also be 3NF. However, in the making of the `Customer` table, the table had both the attributes *customer_zip_code* and *customer_city_name*. Even though *customer_city_name* depends on the *customer_id*, it also depends to *customer_id* transitively via *customer_zipcode* (see Figure 4). Therefore the table was not 3NF. However, the table was normalized by separating *city_name* from the table as seen in Figure 5. In this way, all non primary key attributes depend on solely on the entire key. The table `City` references `Customer` such that one can find the *city_name* given the *customer_zipcode*.

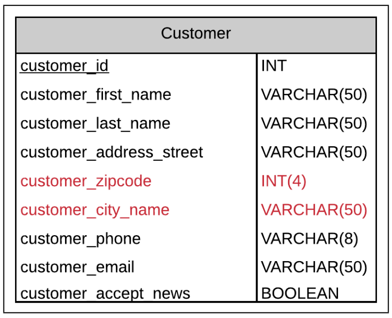


Figure 4: `Customer` table before normalization.

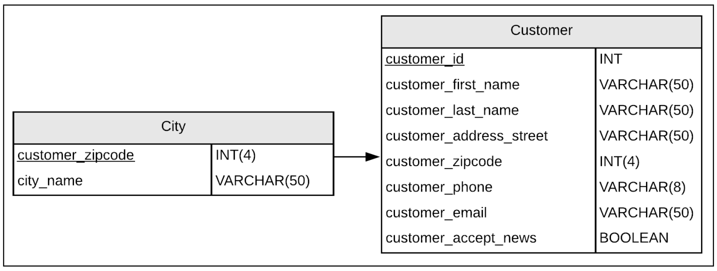


Figure 5: `Customer` table after normalization. Splitting the table into `Customer` and `City` makes the tables 3NF.

4.5 Boyce-Codd Normal Form (BCNF)

In order for a relation schema to be Boyce-Codd Normal Form it must be 3NF. Furthermore, it must also hold that for each non-trivial left irreducible FD, the determinant is a candidate key. Note that if there is only one candidate key, then 3NF and BCNF are the same.
Since all the tables are created in a way such that there is only one possible candidate key, all the tables are also BCNF.

5 Implementation

The last phase of creating a database system is the implementation itself. In this project MariaDB and MySQL Workbench will be utilized for setting up the database system. One example of creating a *database*, two examples of creating *tables*, and three examples of creating *view* will be covered in this section. The remaining SQL code can be found in the file `Charlotte1_02170DatabaseScript1_2020.sql`.

5.1 Create Database

In the SQL script listed below the creation of the database BSOS is initialised. This is the database where all the tables from logical design and normalization will be located. The code more or less speak for itself.

```
DROP DATABASE IF EXISTS BSOS;
CREATE DATABASE BSOS;
USE BSOS;
```

5.2 Create Table

In the below script is the two first tables, namely `City` and `Customer`, added to the database. Appropriate data types have been selected for minimizing the amount of space in the database e.g. danish zip codes only contain 4 digits thus `INT(4)` have been selected. Primary and foreign keys have been selected accordingly to the entity-relationship schema and the logical design. In the below example the zip code works as primary key for the `City` table and foreign key for `Customer` table. In the `Customer` table `ON DELETE SET NULL ON UPDATE CASCADE` have been selected ensuring customers will be replaced with NULL when specific *customer_ id* is deleted, but when a customer is updated, e.g. if they move from one city to another the city, will be updated as well through the foreign key zip code. Similar thought have been applied to the remaining tables from the logical scheme.

```
CREATE TABLE City(
    customer_zipcode INT(4) NOT NULL AUTO_INCREMENT,
    city_name VARCHAR(50),
    PRIMARY KEY(customer_zipcode)
    );

CREATE TABLE Customer(
    customer_id   INT NOT NULL AUTO_INCREMENT,
    customer_first_name VARCHAR(50),
    customer_last_name VARCHAR(50),
    customer_zipcode INT(4),
    customer_address VARCHAR(50),
    customer_phone VARCHAR(8),
    customer_email VARCHAR(50),
    customer_accept_news BOOLEAN,
```

```
    PRIMARY KEY(customer_id),
    FOREIGN KEY(customer_zipcode) REFERENCES City(customer_zipcode)
    ON DELETE SET NULL ON UPDATE CASCADE
```

5.3 Create Views

Three views have been created `ProductPrice`, `CustomerOrders` and `StockHealth`. The first one shows the total price of a product taking into account the discount. The second shows contact information of a customer and their orders. Finally, the last one gives an indicator of the stock of a product is low, medium or high. The first view of total price of each product is seen below, it calculates the price by using a `NATURAL LEFT OUTER JOIN` of product and deals. Product is joined by deals as we want to look at every product, not only the ones with a discount. The price is then calculated taking into account the discount, making sure that if the discount is NULL, the price should be simply multiplied by 1.

```
/*View the total price of a discounted product*/
CREATE VIEW ProductPrice AS SELECT product_price, deal_discount,
      product_price*IFNULL(1-deal_discount,1) AS total_price,
      product_price-product_price*IFNULL(1-deal_discount,1) AS
   savings
            FROM product NATURAL LEFT OUTER JOIN deals;
```

The second view of order of each customer, here we concatenate the names and look at the orders of a customer by joining orders with customer.

```
/*View the order of each customer and some contact information*/
CREATE VIEW CustomerOrders AS SELECT customer_id,
      CONCAT(customer_first_name,' ',customer_last_name) AS
   full_name,
            customer_phone, order_id
            FROM Orders NATURAL LEFT OUTER JOIN Customer;
```

The third view is the indicator of the stock. This is simply done by using a case statement. If the stock is below or equal 15, then it is low. If above 15 and less or equal to 100 then it is medium, and if above 100 then it is high.

```
/*View the stock of a product and gives an indicator if the stock
   is low*/
CREATE VIEW StockHealth AS SELECT product_id, product_stock, (CASE
            WHEN product_stock <= 15 THEN 'Warning: Low'
                        WHEN 15 < product_stock AND product_stock
   <= 100 THEN 'Medium'
                        WHEN 100 < product_stock THEN 'High'
                        END
            ) AS stock_level FROM product;
```

6 Database Instance

In this section it will be covered how values are inserted into the table. Only one example of an insert statement will be given in the report as it is more or less the same whether one insert into one table or the others as long as one take the appropriate data structures into consideration. The remaining SQL insert statements can be found in the `Charlotte1_02170DatabaseScript1_2020.sql` file. All instances of all tables and all views will be shown.

6.1 Insertion of data

In the script below an example can be seen of how the table `Customer` was populated with data. A total of seven persons have been added. As can be seen one row is added at the time and each column corresponds to the selected datatype. The same method have been applied to populate the other tables form Logical Design (see Figure 2).

```sql
INSERT Customer VALUES
(1,'Christian', 'Glissov', 2830, 'Gammel Haslevvej 32', '66778899',
  'glissovsen@gmail.com', TRUE),
(2,'Mikkel', 'Groenning', 2100, 'Oesterbro Alle 109', '11223344', '
  spam607@mail.com', FALSE),
(3,'Melina', 'Barhagh', 2300, 'Amager Road, 1 th', '11223344', '
  Barhagh@live.com', TRUE),
(4,'Kamilla', 'Bonde', 2200, 'Noerrebro Street 69 3 th', '44556677'
  , 'skafte@hotmail.com', TRUE),
(5,'Anne', 'Haxthausen',  9000,  'Aalborg gade 10, 1 th', '45257510
  ', 'aeha@dtu.dk',  FALSE),
(6,'Charlotte', 'Theisen', 9000,  'Aarhusvej 10', '00112112', '
  XXXXXX@student.dtu.dk', TRUE),
(7,'Asgarath', 'Boomkin',  9999,  'Orgrimmar Road 14', '12345678',
  'nerd@worldofwarcraft.com', TRUE);
```

6.2 Display of instances of all tables and views

In Figure 6 can all entities of all tables in the database BSOS be seen. The views can be seen in Figure 7 an Figure 7.

```
SELECT * FROM City;
```

customer_zipcode	city_name
2100	København Ø
2300	København S
2750	Ballerup
2830	Virum
2850	Nærum
4000	Roskilde
4900	Nakskov
9000	Aalborg
9999	Orgrimmar

```
SELECT * FROM Product;
```

product_id	product_date	product_type	product_name	product_pri...	product_sto...	product_brand	deal_id
1	2020-02-03	tops	sports top, just do it	300.00	20	nike	NULL
2	2019-11-06	jacket	comfy winter jacket	1200.00	30	zara	NULL
3	2019-01-16	tshirt	bsos basic tshirt darkblue	89.00	189	bsosdesign	2
4	2019-01-16	tshirt	bsos basic tshirt black	89.00	200	bsosdesign	2
5	2019-05-06	unspecified	denim underpants	100.00	2000	bsosdesign	NULL
6	2020-01-02	dress	boohoo colourful dress	280.00	40	boohoo	NULL
7	2019-12-09	shirt	casual shirt	600.00	20	obey	NULL
8	2019-07-09	pants	stretchy sports tights	449.00	50	nike	3
9	2018-12-09	pants	bsos slim fit jeans	700.00	150	bsosdesign	NULL
10	2018-12-09	sweater	bsos christmas sweater	300.00	150	bsosdesign	NULL
11	2019-10-09	sweater	bsos wow theme recked...	600.00	10	bsosdesign	NULL
12	2018-01-01	pants	very good pants	1200.00	0	nike	1

```
SELECT * FROM Customer;
```

customer_id	customer_first_name	customer_last_name	customer_zipcode	customer_address	customer_phone	customer_email	customer_accept_news
1	Christian	Glissov	2830	Gammel Haslevvej 32	66778899	glissovsen@gmail.com	1
2	Mikkel	Grønning	2100	Østerbro Alle 109	11223344	spam607@mail.com	0
3	Melina	Barhagh	2300	Amager Road, 1 th	11223344	Barhagh@live.com	1
4	Kamilla	Bonde	4900	Nakeskov Bouldevard 69 3 th	44556677	skafte@hotmail.com	1
5	Anne	Haxthausen	9000	Aalborg gade 10, 1 th	45257510	aeha@dtu.dk	0
6	Charlotte	Theisen	9000	Aarhusvej 10	00112112	XXXXXX@student.dtu.dk	1
7	Asgarath	Boomkin	9999	Orgrimmar Road 14	12345678	nerd@worldofwarcraft.com	1

```
SELECT * FROM Orders;
```

order_id	customer_id	order_date	order_approval	order_shipped	order_payment_info
1	1	2019-01-01 10:34:00	1	2019-01-01 09:30:01	mobilepay
2	1	2019-02-10 04:58:00	1	2019-02-15 11:42:01	paypal
3	2	2019-03-07 14:35:00	1	2019-03-15 11:07:01	mobilepay
4	3	2019-03-09 17:25:00	1	2019-03-09 13:00:01	transaction
5	2	2019-04-18 03:14:00	1	2019-04-23 15:58:01	visa
6	1	2019-04-24 09:58:00	1	2019-04-29 14:39:01	visa
7	4	2019-05-03 16:57:00	1	2019-05-06 14:38:01	mastercard
8	5	2019-05-27 18:12:00	1	2019-06-04 12:57:01	mobilepay
9	3	2019-05-29 19:47:00	1	2019-06-04 17:52:01	transaction
10	6	2019-06-02 13:36:00	1	2019-06-04 12:35:01	mobilepay
11	4	2019-06-07 12:52:00	1	2019-06-15 14:39:01	mastercard
12	7	2019-07-31 17:41:00	1	2019-08-04 14:09:01	mobilepay

```
SELECT * FROM OrderItem;
```

order_id	product_id	order_item_quantity
4	1	3
6	1	2
1	2	2
4	2	5
7	2	1
11	2	4
6	3	1
7	3	1
12	3	2
2	4	1
1	6	1
3	6	1
5	6	1
7	6	1
8	9	2
9	9	1
10	9	1
1	10	3
4	10	1

```
SELECT * FROM Deals;
```

deal_id	deal_discount	deal_starts	deal_expires
1	0.50	2018-01-01 07:00:00	2018-01-05 17:00:00
2	0.10	2019-05-01 07:00:00	2018-05-10 17:00:00
3	0.20	2020-02-10 07:00:00	2018-02-20 17:00:00

```
SELECT * FROM Rating;
```

product_id	customer_id	rating_quality	rating_fit	rating_review
2	1	4	4	nice
2	7	1	5	bad quality, but awesome fit
3	2	1	1	I DID NOT LIKE THIS PRODUCT! I DEMAND REFUND, NOW
4	2	5	5	!!! OMG, it was just the best piece of clothe i have ever had hahhah
6	6	4	4	jeg kan lide det
9	6	3	3	average

Figure 6: The figure shows all entities in all tables in the database BSOS.

customer_id	full_name	customer_phone	order_id
1	Christian Glissov	66778899	1
1	Christian Glissov	66778899	2
1	Christian Glissov	66778899	6
2	Mikkel Grønning	11223344	3
2	Mikkel Grønning	11223344	5
3	Melina Barhagh	11223344	4
3	Melina Barhagh	11223344	9
4	Kamilla Bonde	44556677	7
4	Kamilla Bonde	44556677	11
5	Anne Haxthausen	45257510	8
6	Charlotte Theisen	00112112	10
7	Asgarath Boomkin	12345678	12

product_price	deal_discount	total_price	savings
306.00	NULL	306.0000	0.0000
1200.00	NULL	1200.0000	0.0000
89.00	0.10	80.1000	8.9000
89.00	0.10	80.1000	8.9000
100.00	NULL	100.0000	0.0000
280.00	NULL	280.0000	0.0000
600.00	NULL	600.0000	0.0000
449.00	0.20	359.2000	89.8000
700.00	NULL	700.0000	0.0000
300.00	NULL	300.0000	0.0000
600.00	NULL	600.0000	0.0000
1200.00	0.50	600.0000	600.0000

Figure 7: View of `customerorders` and `productprice`

product_id	product_stock	stock_level
1	20	Medium
2	30	Medium
3	189	High
4	200	High
5	2000	High
6	40	Medium
7	20	Medium
8	50	Medium
9	150	High
10	150	High
11	10	Warning: Low
12	0	Warning: Low

Figure 8: View of `stockprice`

7 SQL Data Queries

In this section four examples of typical select SQL statements with `ORDER BY`, `GROUP BY` and `JOIN` will be explained.

7.1 First select statement

BSOS is interested in providing its customers with a quick overview of the ratings of the products. To do this the average rating of *rating_fit*, *rating_quality* and the combination of the two is shown. The `SELECT` also shows the corresponding product ID. Figure 9 confirms that the script works as intended. The script works by `NATURAL LEFT OUTER JOIN` the rating by the product, this gives all ratings for each product, after this a `GROUP BY` each of the products is utilised, making sure the average of the ratings of each product is taken, this is done by utilising the function `AVG`. Finally, the products are sorted by the best to worst average rating, using the `ORDER BY` keyword.

```sql
/*Calculates the average rating of a product*/
SELECT product_id, AVG(rating_quality) as avg_quality_rating,
AVG(rating_fit) as avg_fit_rating,
(AVG(rating_quality)+AVG(rating_fit))/2
FROM rating NATURAL LEFT OUTER JOIN product
GROUP BY product_id
ORDER BY (AVG(rating_quality)+AVG(rating_fit))/2 DESC;
```

product_id	avg_quality_rating	avg_fit_rating	avg_rating
4	5.0000	5.0000	5.00000000
6	4.0000	4.0000	4.00000000
2	2.5000	4.5000	3.50000000
9	3.0000	3.0000	3.00000000
3	1.0000	1.0000	1.00000000

Figure 9: The table shows the output from calling first select SQL statement

7.2 Second select statement

BSOS wants to easily see who has reviewed what product and what they have been writing about the product in the review. By "who" BSOS means the full name i.e. first and last name. BSOS wants to know the reviews of each product type and brand, to analyze the what people write about a certain type and brand. The script that achieves this is seen in below and Figure 10 confirms the desired output. The full name is achieved concatenating the first and last name using `CONCAT` function. Getting the customer who made what review is achieved by joining customer IDs of rating and customer, similarly getting the products that have been reviewed is achieved by the join of product IDs for rating and product.

```sql
/*Shows the rating reviews of a product and the persons who wrote it*/
SELECT CONCAT(customer_first_name,' ',customer_last_name) as full_name,
rating_review, product_type, product_name, product_brand FROM rating
JOIN customer ON customer.customer_id = rating.customer_id
JOIN product ON product.product_id = rating.product_id;
```

full_name	rating_review	product_type	product_name	product_brand
Christian Glissov	nice	jacket	comfy winter jacket	zara
Asgarath Boomkin	bad quality, but awesome fit	jacket	comfy winter jacket	zara
Mikkel Grønning	I DID NOT LIKE THIS PRODUCT! I DEMAND REF...	tshirt	bsos basic tshirt darkblue	bsosdesign
Mikkel Grønning	!!! OMG, it was just the best piece of clothe i ha...	tshirt	bsos basic tshirt black	bsosdesign
Charlotte Theisen	jeg kan lide det	dress	boohoo colourful dress	boohoo
Charlotte Theisen	average	pants	bsos slim fit jeans	bsosdesign

Figure 10: The figure show the output from calling second select SQL statement

7.3 Third select statement

Lastly BSOS is interested in finding the total spending that each customer has spent in the online shop. I.e. the sum of each product price multiplied by the discount in every order. To do this it is required to first use the `JOIN` keyword. First `Customer` is joined by `Orders` using the *Customer_id* using an `INNER JOIN` keyword, this merges `Orders` and `Customer` by the IDs, to `SELECT` the customer last name. This is joined by `orderitem` and then `product` again using the *order_id* first and then the *product_id*, first to get the items of each order and then to get the price of each product. As we want to look at each customer a `GROUP BY` statement is used on the *customer_id* and then finally these are ordered to by the total spending given by the sum of all the prices of each individual order. Below the SQL code can be observed. Figure 11 confirms the desired output.

```
/*Shows the amount a customer have spent on products*/
SELECT orders.customer_id, customer_last_name,
SUM(order_item_quantity*product_price*IFNULL(1-deal_discount,1))
AS total_spending FROM
customer
INNER JOIN orders ON
orders.customer_id = customer.customer_id
INNER JOIN orderitem ON
orders.order_id = orderitem.order_id
INNER JOIN product ON
product.product_id = orderitem.product_id
NATURAL LEFT OUTER JOIN deals
GROUP BY customer.customer_id
ORDER BY total_spending DESC;
```

customer_id	customer_zipcode	customer_last_name	total_spending
3	2300	Barhagh	7900.0000
4	4900	Bonde	6360.1000
1	2830	Glissov	4340.2000
5	9000	Haxthausen	1400.0000
6	9000	Theisen	700.0000
2	2100	Grønning	560.0000
7	9999	Boomkin	160.2000

Figure 11: The figure show the output from calling thirds select SQL statement

8 SQL Table Modifications

In this section some examples of frequent SQL table update and delete will given and explained why the make sense in terms om BSOS webshop database system. In total two examples of the `UPDATE` statements will be given as well as two examples of `DELETE` statements. Furthermore an example of procedure which makes it easy to UPDATE and DELETE will be given

8.1 SQL UPDATE

In order to promote BSOS own design collection the owner of BSOS come up with the idea of a 25 % discount only on BSOS own design. The following SQL query displays first how the deal is created and then how products with the brand *bsosdesign* get the discount by using the `UPDATE` statement.

```
-- Create new campaign
INSERT Deals VALUES(4, 0.25, '2020-04-01 00:00:00', '2018-04-07
    23:59:59');

-- Update products with bsosdesign brand to have the new deal
```

```
UPDATE product set deal_id = 4 WHERE product_brand = 'bsosdesign';
```

Occasionally, a costumer will make an error when typing their email or simply change it to another email provider. This can be changed in the system by using the following code where customer with *customer_id* = 3 would like her email changed.

```
UPDATE Customer
SET customer_email = 'barhagh@hotmail.com', customer_accept_news =
    0
WHERE customer_id = 3;
```

Before the change in the database is made the customer information is shown in Figure 12.

customer_id	customer_first_name	customer_last_name	customer_email	customer_accept_news
3	Melina	Barhagh	Barhagh@live.com	1

Figure 12: Customer with *customer_id* = 3 before update of *customer_email* and *customer_accept_news*

Since the customer email is not referenced in other tables, there are no dependencies that should be accounted for, which leads to the updated customer information in Figure 13.

customer_id	customer_first_name	customer_last_name	customer_email	customer_accept_news
3	Melina	Barhagh	barhagh@hotmail.com	0

Figure 13: Customer with *customer_id* = 3 after *customer_email* is updated. *customer_accept_news* is set to 0, since we are not ensured permission for the new e-mail.

BSOS webshop can easily adjust prices of a product by using the procedure, `priceScaler`, it takes the *product_id* and a scale as input, then it uses the `UPDATE` statement to update the price of the product.

```
DELIMITER //
CREATE PROCEDURE priceScaler(IN scale DECIMAL (3,2), vproduct
    INT)
BEGIN
  UPDATE product SET product_price=product_price*scale WHERE
    product_id = vproduct;
 END; //
 DELIMITER ;

```

In Figure 14 an example can be seen.

product_id	product_date	product_type	product_name	product_price	product_stock	product_brand	deal_id
1	2020-02-03	tops	sports top, just do it	300.00	20	nike	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

product_id	product_date	product_type	product_name	product_price	product_stock	product_brand	deal_id
1	2020-02-03	tops	sports top, just do it	306.00	20	nike	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

Figure 14: Before and after a price increase of 2 percent.

8.2 SQL DELETE

It is crucial to keep good reviews of the products within the company. This will give a potential for higher profits. Some might find this method disturbing, but an easy way to achieve this is to simply delete bad ratings. This is done by the SQL `DELETE` statement. The way it works is, that all rating below or equal to two is deleted in the rating entity.

```sql
DELETE FROM Rating WHERE rating_quality <= 2;
```

Delete statements can be very useful for clean-ups as well. In Denmark it is only statutory to save billing information for 5 years. Hence one could delete all invoices that places at least 5 years ago. Now, since `Orders` do not have any orders of that age, we simply do the same procedure for orders, that are at least 1 years old instead.

```sql
DELETE FROM Orders
WHERE order_date < DATE_SUB(CURRENT_DATE(), INTERVAL 1 YEAR);
```

The `Order` table before the deletion can be seen in Figure 16, while the table after deletion can be seen in Figure 16.

order_id	customer_id	order_date	order_approval	order_shipped	order_payment_info
1	1	2019-01-01 10:34:00	1	2019-01-01 09:30:01	mobilepay
2	1	2019-02-10 04:58:00	1	2019-02-15 11:42:01	paypal
3	2	2019-03-07 14:35:00	1	2019-03-15 11:07:01	mobilepay
4	3	2019-03-09 17:25:00	1	2019-03-09 13:00:01	transaction
5	2	2019-04-18 03:14:00	1	2019-04-23 15:58:01	visa
6	1	2019-04-24 09:58:00	1	2019-04-29 14:39:01	visa
7	4	2019-05-03 16:57:00	1	2019-05-06 14:38:01	mastercard
8	5	2019-05-27 18:12:00	1	2019-06-04 12:57:01	mobilepay
9	3	2019-05-29 19:47:00	1	2019-06-04 17:52:01	transaction
10	6	2019-06-02 13:36:00	1	2019-06-04 12:35:01	mobilepay
11	4	2019-06-07 12:52:00	1	2019-06-15 14:39:01	mastercard
12	7	2019-07-31 17:41:00	1	2019-08-04 14:09:01	mobilepay

Figure 15: A snapshot of the `Order` table.

order_id	customer_id	order_date	order_approval	order_shipped	order_payment_info
5	2	2019-04-18 03:14:00	1	2019-04-23 15:58:01	visa
6	1	2019-04-24 09:58:00	1	2019-04-29 14:39:01	visa
7	4	2019-05-03 16:57:00	1	2019-05-06 14:38:01	mastercard
8	5	2019-05-27 18:12:00	1	2019-06-04 12:57:01	mobilepay
9	3	2019-05-29 19:47:00	1	2019-06-04 17:52:01	transaction
10	6	2019-06-02 13:36:00	1	2019-06-04 12:35:01	mobilepay
11	4	2019-06-07 12:52:00	1	2019-06-15 14:39:01	mastercard
12	7	2019-07-31 17:41:00	1	2019-08-04 14:09:01	mobilepay

Figure 16: A snapshot of the `Order` table after all orders placed at least 1 year ago are deleted.

If a customer cancels an item in a product, then it is easy to delete by using the procedure using a `DELETE` statement. This is done in the procedure `delete_orderitem`.

```
/* Delete an ordered item */
DELIMITER //
CREATE PROCEDURE delete_orderitem(IN vorder_id INT, vproduct_id
    INT)
BEGIN
  DELETE FROM orderitem WHERE order_id = vorder_id AND product_id =
    vproduct_id ;
END; //
DELIMITER ;
```

As a test case the input of product ID equal to 1 and an order ID of 4.

order_id	product_id	order_item_quantity
4	1	3
6	1	2
1	2	2
4	2	5
7	2	1

order_id	product_id	order_item_quantity
6	1	2
1	2	2
4	2	5
7	2	1

Figure 17: A snapshot of the `delete_orderitem` procedure.

9 SQL Programming

In this section of the report examples of SQL programming will be explained. Two `FUNCTION`, one `PROCEDURE`, one `TRIGGER`, and one `EVENT` will be shown.

9.1 Functions

It is essential to see how much a customer is spending for a business, the function will take an input of a *customer_id* and return the total spending of that customer.

It does this by joining several tables, such as `Orders`, `Product` and `OrderItem` by their respective IDs.

```
DELIMITER //
CREATE FUNCTION CustomerPrice (vCustomer_id INT)
RETURNS FLOAT
BEGIN
  DECLARE vPrice FLOAT;
  SELECT SUM(order_item_quantity*product_price*IFNULL(1-
   deal_discount,1)) INTO vPrice
            FROM customer
            INNER JOIN orders ON
            orders.customer_id = customer.customer_id
            INNER JOIN orderitem ON
            orders.order_id = orderitem.order_id
            INNER JOIN product ON
            product.product_id = orderitem.product_id
            NATURAL LEFT OUTER JOIN deals
            WHERE  customer.customer_id = vCustomer_id;
  RETURN vPrice;
END; //
DELIMITER ;
```

Using the `SELECT` statement the spending of all customers can also be seen

```
SELECT customer_id, CustomerPrice(customer_id) AS total_spending
   FROM customer ORDER BY total_spending DESC;
```

Given an input of just a single number gives the spending of a single customer, for customer 3, the output is 7900, this is the base price, with no inserts or deletes prior to the execution of the function. Customer 3 has order 4, with 5 items of product 2, 1 item of product 10, 3 items of product 1 and order 9 with 1 items of product 9. This gives

$$\text{CustomerPrice}(3) = 5 \cdot 1200 + 300 + 3 \cdot 300 + 700 = 7900$$

Another useful function is one, that tests whether all items in an order are on stock. Therefore a function is created, such that given an *order_id*, the stock is checked for each item in that order. If $order\_item\_quantity \leq product\_stock$ for all items in the given order, the function returns the Boolean value 1, if not it returns 0. Thus this function can be used to check, whether an order can be approved or not. This will be shown in subsection 9.4.

```
DROP FUNCTION IF EXISTS in_stock;
DELIMITER //
CREATE FUNCTION in_stock(orderid INT) RETURNS BOOLEAN
BEGIN
```

```
DECLARE approved BOOLEAN;
  SELECT sum(order_item_quantity <= product_stock) = count(
 distinct product_id)
  INTO approved
FROM OrderItem
  LEFT JOIN Product using(product_id)
  WHERE order_id = orderid;
  RETURN approved;
END; //
DELIMITER ;
```

Say one wanted to check whether the order with $order\_id = 7$ could be approved. Then one could write the following command:

```
SELECT in_stock(7);
```

Given the current state of the database, this function call would return 1.

9.2 Procedures

BSOS online shop thinks it is complicated to create deals. They want an easy implementation to add a new discount starting the very moment it put into the database and last a week. The SQL script below displays a procedure where an new deal can easily be added by just adding the discount e.g. 0.33 (33 %) and it is put in to the database right away starting from the moment procedure is called lasting exactly 7 days.

```
DROP PROCEDURE IF EXISTS standardDeal;

DELIMITER //
CREATE PROCEDURE standardDeal (IN discount DECIMAL(3,2))
BEGIN
  INSERT Deals(deal_discount, deal_starts, deal_expires)
  VALUES (discount, NOW(), NOW() + INTERVAL 7 DAY);
END; //

DELIMITER ;
```

9.3 Transactions

Since the database has been simplified and does not include returns and payment transactions, the transaction has been used to update the stock of a product, though it does not remove the quantity from the `OrderItem` table.

This can be used in case a synchronization error has occurred and the stock is not updated automatically when the order is approved. It can be done by using the procedure

```sql
DROP PROCEDURE IF EXISTS Stock_update;
DELIMITER //
CREATE PROCEDURE Stock_update(
IN vproduct INT, vquantity INT, approval BOOLEAN, OUT vStatus VARCHAR(45))
BEGIN
DECLARE Oldstock, Newstock INT DEFAULT 0; START TRANSACTION;
SET Oldstock = (SELECT product_stock FROM Product WHERE product_id = vproduct); SET Newstock = Oldstock - vquantity;
UPDATE Product SET product_stock = Newstock WHERE product_id = vproduct;
IF (approval)
THEN SET vStatus = 'Transaction Transfer committed!'; COMMIT;
ELSE SET vStatus = 'Transaction Transfer rollback'; ROLLBACK; END IF;
END; //
DELIMITER ;
```

The procedure takes 3 inputs; the *product_id* and *order_item_quantity* from the table `OrderItem` and the *order_approval* from the `Order` table. It then removes the purchased quantity from the product table, when the *order_approval* is `TRUE`, i.e. when the order has been approved.

To perform the update manually the transaction procedure is called as below, where a customer has placed an order containing two *sports top, just do it*.

```sql
CALL Stock_update(1, 2, 1, @Status);
```

In Figure 18 the product and current stock is shown while Figure 19 shows the updated stock after running the transaction procedure.

product_id	product_name	product_stock
1	sports top, just do it	20

Figure 18: Product before stock is updated according to the placed order

product_id	product_name	product_stock
1	sports top, just do it	18

Figure 19: Product before stock is updated according to the placed order

Should the procedure be run by mistake for a product which is part of a rejected order, a `rollback` would be performed and the stock would remain unchanged. Since the update is successful the status in Figure 20 shows that the transaction was saved in the database.

@Status
Transaction Transfer committed!

Figure 20: Transaction status when the procedure has been successfully completed and stock is updated.

9.4 Triggers

Every time an order is placed a new row is inserted in the `Order` table, which triggers an automatic response in the database, to verify that the products are in stock. This is done with a trigger that uses the previously described function `in_stock()` which determines the value of the attribute *order_ approval* for all new orders.
When the function returns `TRUE` the order is accepted and else rejected as the items are not in stock.

```
DROP TRIGGER IF EXISTS approve_order;
DELIMITER //
CREATE TRIGGER approve_order
AFTER INSERT ON OrderItem FOR EACH ROW
BEGIN
  UPDATE Orders
    SET Orders.order_approval = in_stock(NEW.order_id) WHERE NEW.
   order_id = Orders.order_id;
END; //
DELIMITER ;
```

To demonstrate how the trigger is used two new orders are placed. Order with *order_ id* = 13 contains two items of product 1 and one item of product 12. The costumer then receives a notice that the order can not be accepted since product 12 is out of stock. Therefore the costumer places an order with *order_ id* = 14 which still contains 2 items of product 1 and now one item of product 11 instead. The result by using the trigger is shown in Figure 21. As seen order 13 is rejected while order 14 is approved when the trigger is applied.

order_id	customer_id	order_date	order_approval	order_shipped	order_payment_info
1	1	2019-01-01 10:34:00	1	2019-01-01 09:30:01	mobilepay
2	1	2019-02-10 04:58:00	1	2019-02-15 11:42:01	paypal
3	2	2019-03-07 14:35:00	1	2019-03-15 11:07:01	mobilepay
4	3	2019-03-09 17:25:00	1	2019-03-09 13:00:01	transaction
5	2	2019-04-18 03:14:00	1	2019-04-23 15:58:01	visa
6	1	2019-04-24 09:58:00	1	2019-04-29 14:39:01	visa
7	4	2019-05-03 16:57:00	1	2019-05-06 14:38:01	mastercard
8	5	2019-05-27 18:12:00	1	2019-06-04 12:57:01	mobilepay
9	3	2019-05-29 19:47:00	1	2019-06-04 17:52:01	transaction
10	6	2019-06-02 13:36:00	1	2019-06-04 12:35:01	mobilepay
11	4	2019-06-07 12:52:00	1	2019-06-15 14:39:01	mastercard
12	7	2019-07-31 17:41:00	1	2019-08-04 14:09:01	mobilepay
13	1	2020-01-01 08:13:00	0	NULL	mobilepay
14	1	2020-01-01 08:15:00	1	NULL	mobilepay

Figure 21: The figure show the registered orders after where the trigger approve_order have rejected and accepted the orders with *order_id* 13 and 14 respectively.

9.5 Events

To ensure a fast delivery BSOS has decided that all orders made before 3 PM are shipped the same day. The **Orders** table is updated every day to register the day the order was shipped. As they aim to deliver outstanding service all orders made during weekends and bank holidays are also shipped the same day.
Hence, the attribute *order_shipped* is updated for the accepted orders once each day starting at 3 PM by using the event Shipped as shown below

```sql
CREATE EVENT Shipped
ON SCHEDULE EVERY 1 DAY
STARTS '2020-04-04 15:00:00'
DO UPDATE Orders
  SET order_shipped = CASE WHEN order_approval = 1
                            AND order_shipped IS NULL
                               THEN CURRENT_TIMESTAMP
                           ELSE order_shipped
                      END;
```

As seen in Figure 21 none of the items have been shipped. To demonstrate how the event Shipped works, it is scheduled to update every minute to ship the accepted order immediately. After the item is shipped, the timestamp has been updated as seen in Figure 22.

order_id	customer_id	order_date	order_approval	order_shipped	order_payment_info
1	1	2019-01-01 10:34:00	1	2019-01-01 09:30:01	mobilepay
2	1	2019-02-10 04:58:00	1	2019-02-15 11:42:01	paypal
3	2	2019-03-07 14:35:00	1	2019-03-15 11:07:01	mobilepay
4	3	2019-03-09 17:25:00	1	2019-03-09 13:00:01	transaction
5	2	2019-04-18 03:14:00	1	2019-04-23 15:58:01	visa
6	1	2019-04-24 09:58:00	1	2019-04-29 14:39:01	visa
7	4	2019-05-03 16:57:00	1	2019-05-06 14:38:01	mastercard
8	5	2019-05-27 18:12:00	1	2019-06-04 12:57:01	mobilepay
9	3	2019-05-29 19:47:00	1	2019-06-04 17:52:01	transaction
10	6	2019-06-02 13:36:00	1	2019-06-04 12:35:01	mobilepay
11	4	2019-06-07 12:52:00	1	2019-06-15 14:39:01	mastercard
12	7	2019-07-31 17:41:00	1	2019-08-04 14:09:01	mobilepay
13	1	2020-01-01 08:13:00	0	NULL	mobilepay
14	1	2020-01-01 08:15:00	1	2020-04-03 21:51:25	mobilepay

Figure 22: The figure show the output when the event Shipped has registered that the order with *order_id* 14 will be shipped on the current day.