Database for PizzaPretzel Digital Supports Store

*by Zoe Matrullo and Kilian Schulz*

Given the task to build a database, capable of handling the day to day business of a store offering digital support, we first had to define our **goals for the project**. For us, the main points consist in:

* intuitive division according to the third normal form, as much as it makes sense
* easy insertion of values, corresponding to general actions upon the database
* keeping track of different relationships between tables in an easy and efficient way
* having realistic tables and attributes, that might be used in a real digital support store

That in mind and a Red Bull in the stomach, we set off to achieve exactly that.

Some additional assumptions that we included in our Digital Supports Store, that extend the already given specifics:

The store is divided in different sectors, which correspond to the categories the respective products belong to. Each sector has a manager and a head technician, and is located in one of the store’s floors.

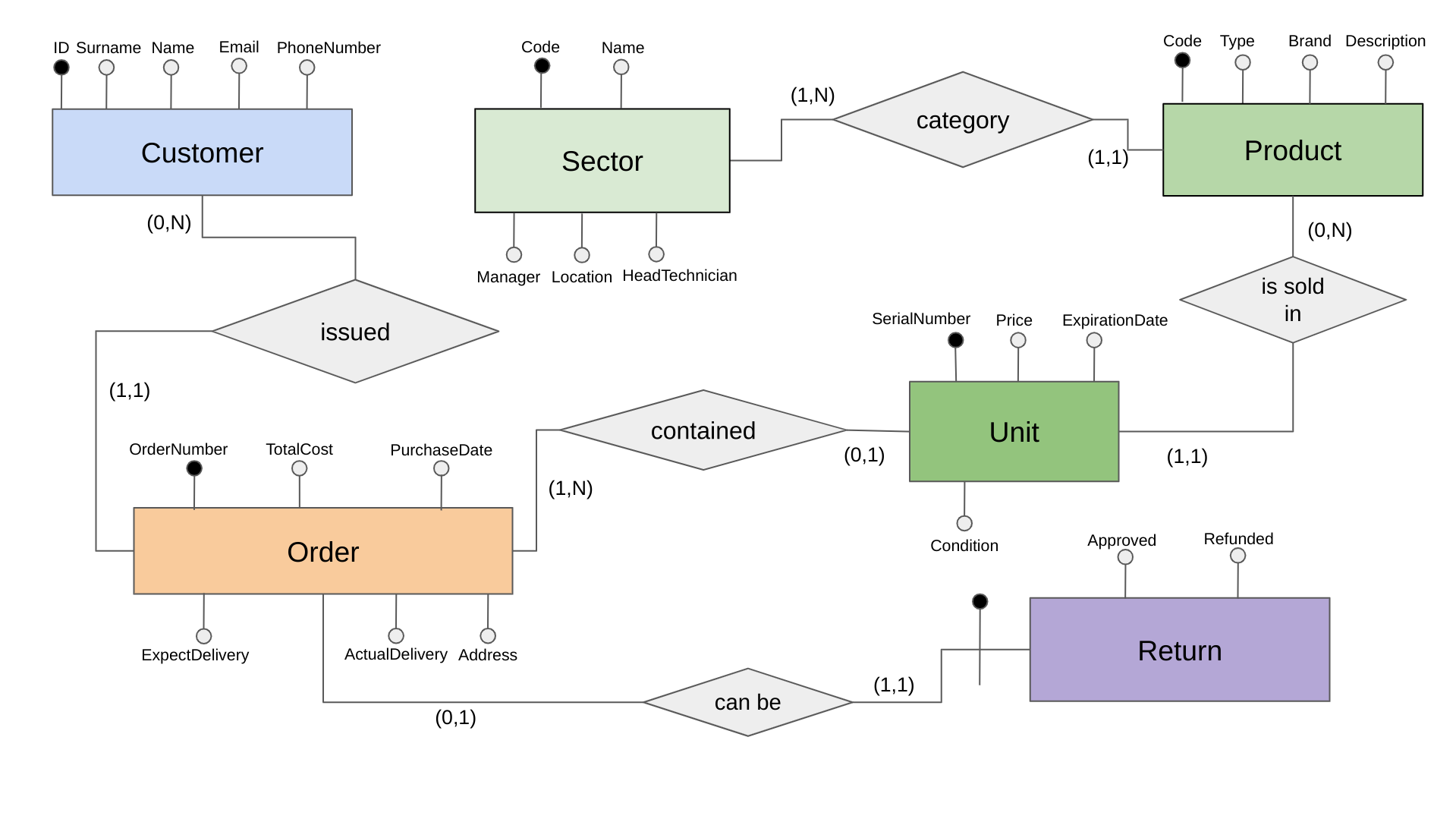
PizzaPretzel sells also used units for a discounted price.

Each customer can order one or more units of the same product and return its order if he is not satisfied.

This translates into:

**ER MODEL**

We used a top-down approach as learned in class, in order to create the following Entity Relationship Model:



As you can see, we wanted to focus on useful relations in order to paint a true picture of the day to day operations in a digital support store.

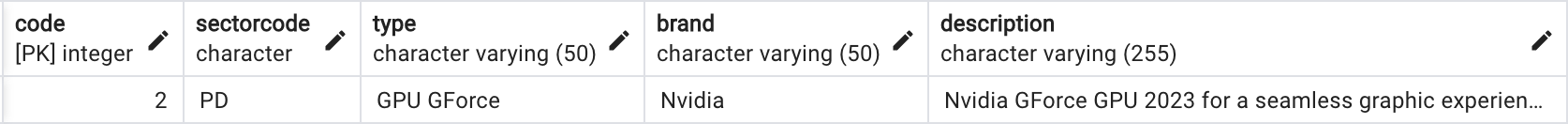
The database for our PizzaPretzel Digital Supports Store has to store data about its (many) CUSTOMERS, namely a unique ID, name, surname, email address and optionally a phone number.

Our store sells items belonging to different SECTOR:

* Input and Output devices,
* Storage Devices,
* Processing Devices,
* Telecommunication supports,
* PC and Laptops
* Others

Each of them has a unique code, sector name (mentioned above), name and surname of its manager, location in the store (ground floor or first floor) and a head technician who is an expert on the products in its sector.

The store sells different products belonging to each one of the mentioned sectors (such as Hard disks in Storage Devices, GPUs in Processing Devices …) of which we record a unique code, the sector/category it belongs to, the type of item (such as CPU i7, GPU GForce …), the brand and a short description.



PIzzaPretzel sells multiple units of the same product, so it’s necessary to have a table that stores information about each UNIT.

In this table we record the unique serial number of the single item, its price, its expiration date and its condition (new or used). If a unit is sold, we specify in which order it is included, so that we can search for the customer that purchased the item.

Each customer can issue one or more orders. For each order we store a unique key, the total cost and purchase date. If the order is sent to the customer’s house, we also record the expected delivery date, the actual delivery date and the address. If the customer picks up the order in the store, these attributes are null.

In the rare event that a customer is not satisfied, the order can be returned. For each RETURN, we record the id of the returned order, if the return has been approved and if so, if the customer has been refunded with his money.

All that is mentioned above translates into the following

**RELATIONAL SCHEMA**

CUSTOMER(ID, Surname, Name, Email, PhoneNumber)

ORDER(OrderNumber, IDCustomer, TotalCost, PurchaseDate, ExpectedDelivery, ActualDelivery, Address)

RETURN(IDOrder, Approved, Refunded)

SECTOR(Code, Name, Manager, Location, HeadTechnician)

PRODUCT(Code, Type, SectorCode, Brand, Description)

UNIT(SerialNumber, Product, Price, ExpirationDate, Condition, IDOrder)

ORDER(IDCustomer) ⊆ CUSTOMER(ID)

PRODUCT(SectorCode)⊆SECTOR(Code)

UNIT(IDOrder)⊆ORDER(OrderNumber)

UNIT(Product)⊆PRODUCT(Code)

RETURN(IDOrder)⊆ORDER(OrderNumber)

We made the conscious decision to implement relations more as foreign keys, than as separate tables, since we do not have any N to N relations and thus don’t need to unnecessarily complicate things.

We chose to define a **TRIGGER**, which, upon inserting a Unit that references an Order, updates the TotalCost attribute of said Order by the Product it references.

CREATE OR REPLACE FUNCTION update\_order\_total\_cost()

RETURNS TRIGGER AS

$$

BEGIN

IF NEW.idorder IS NOT NULL THEN

UPDATE "ORDER"

SET TotalCost = TotalCost + (SELECT Price FROM UNIT WHERE idorder = NEW.idorder)

WHERE OrderNumber = NEW.idorder;

END IF;

RETURN NEW;

END;

$$

LANGUAGE plpgsql;

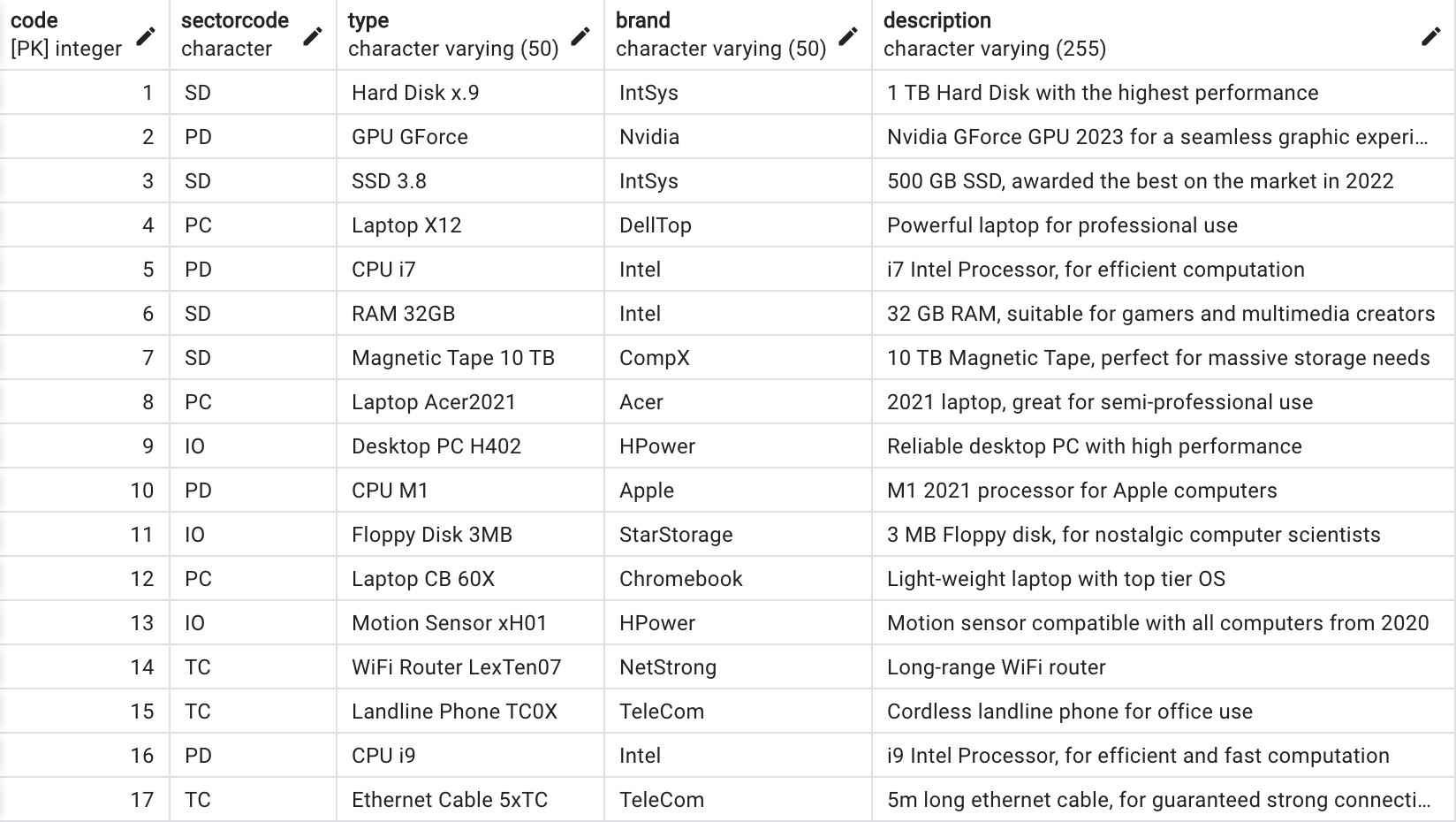
CREATE TRIGGER unit\_idorder\_update\_trigger

AFTER UPDATE OF idorder ON UNIT

FOR EACH ROW

EXECUTE FUNCTION update\_order\_total\_cost();

*Some examples*

1. Here is an example of the PRODUCT table, filled with possible values for illustration purposes:

1. Creating a new order

**INSERT INTO "ORDER" (IDCustomer) VALUES ((select "id" from CUSTOMER where "name" = 'Pam' and surname = 'Beesly')) RETURNING OrderNumber;**

This is going to create an order with the date of today and return the order number that then can be referenced in the units the user wants to order.

Now just go into the UNITS table, and change the IDOrder from NULL to the previously returned OrderNumber.

1. Adding a new customer

**INSERT INTO CUSTOMER(Surname, Name, Email, PhoneNumber) VALUES (‘Schulz’,’Kilian’,’**[**schulz.kilian@outlook.de**](mailto:schulz.kilian@outlook.de)**’,’+39 3509485772’)**

1. Adding a new unit

Also adding a new unit is fairly straightforward. We just need to figure out the Product the unit belongs to, the condition and depending on these two factors, the price.

**INSERT INTO UNIT(Product, Price, Condition) VALUES**

**((SELECT Code FROM PRODUCT WHERE "type" = 'CPU i9' AND Brand = 'Intel'), 350.99, 'New');**