02327 Introductory Databases and Database Programming

Group Project Report (Cykelparadis)

Group 27:

Talha (s255543), Azad (s256184), Simon (s255632),

Kasper (s255422) & Magnus (s256735)

1st. semester: 16/11/2025

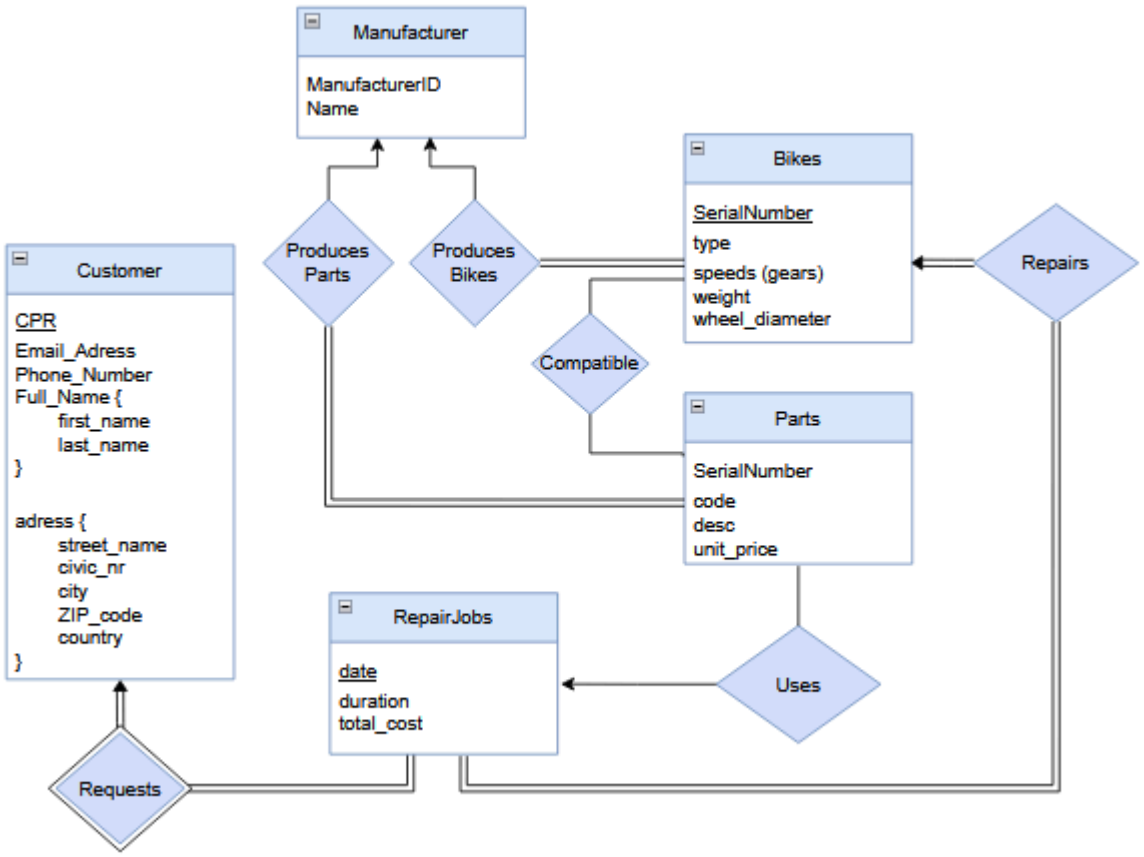
Table of Contents

0. Introduction

For this project, we had to develop a database for a bike repair shop called Cykelparadis. The purpose was to design a system that could keep track of the bikes, customers, and the parts used, and the different repair jobs. At first we spent time trying to figure out what the shop actually needed and how the things should be connected. We started out by creating a conceptual model to get an idea of the main entities and how they should relate to each other. Although we did discover some things we had misunderstood in the beginning which led to us having to change our design before moving on to the next stage. Once the conceptual model made sense we converted it into a logical model that we could later implement into SQL tables, and a full schema with example data to showcase different possible scenarios.

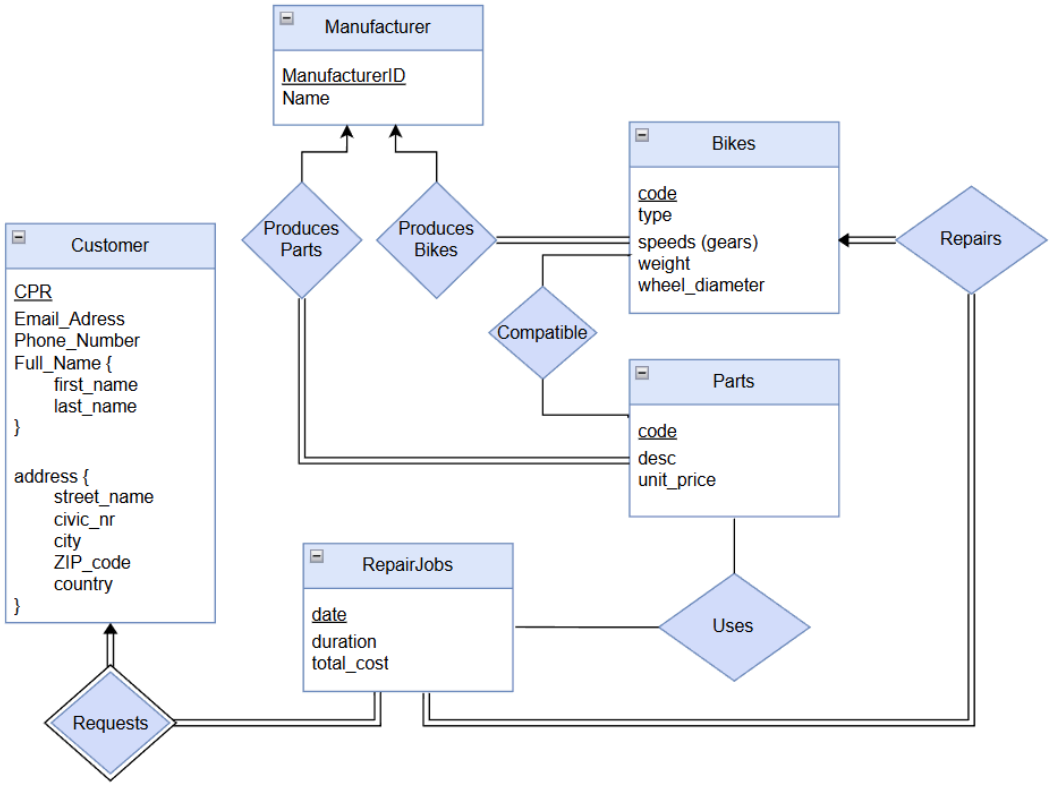
1. Conceptual Design

Example of conceptual model #1, during the development process:



When we designed our first conceptual model, we had slightly misinterpreted the assignment, we gave each part and bike their own unique serial number, with intent to track them individually. However we later realized that this was not necessary, and would result in bloat.

Finalised Conceptual Model #2:



Explanation of our conceptual design based on the ER model

In this version of the ER diagram we kept the overall structure the same, but we moved away from the idea of giving every bike and part their own unique serial number and instead we only store the model code, which makes things a lot simpler and makes more sense from a practical perspective.

Entities and their attributes

The entity *Customer* has attributes for personal information about that customer, including Email address, phone number and their first and last name. It also includes their address, which contains their street name, civic number, city ZIP code and country. Later on in our logical model, we placed customers' addresses in a separate table.

The entity *RepairJobs* holds the attributes for the date of repair, the duration and the total cost.

The entity *Bikes* holds the attributes code, type of bike, amount of gears/speeds, its weight and the wheel diameter.

The entity *Parts* has the attributes code, description and unit price.

The entity *Manufacturer* has the attributes manufacturer ID and the name of the manufacturer.

Relations:

A manufacturer can either produce parts, bikes, or both. So we have the relations *Produces Parts*, and *Produces Bikes*.

We have to be able to check whether a part is compatible with a bike so we have the relation *Compatible.*

To check which parts have been used in a repair job, we have the relation *Uses*.

To see which customer requests a job we have the relation *Requests*.

To see which bike gets repaired in a repair job we have the relation *Repairs*

Primary Key Assignments

Primary key in Customer

In customer we have our primary key as CPR number, which is unique by design among all Danish citizens. Of course this comes with the drawback that the repair shop can not do business with any customers who are not Danish citizens, but the assignment required us to use CPR.

Primary key in RepairJobs

In repair jobs we had set the primary key to the date of the repair, this would have led to problems where one customer would not be able to repair a bike twice in the same day. We later changed this in our logical model to use a repair_ID instead, explained later.

Primary key in Manufacturer

We implemented a manufacturer_ID as it is not too unlikely that two manufacturers could have the same name, for example, two manufacturers may be called ‘Bike Parts ApS’. With an ID we make sure that would not be an issue.

The reason we make our own ID, instead of using a pre-established ID like CVR-numbers, is that CVR numbers are not universal and are only for business in Denmark. If the repair shop buys their parts from other countries, which is to be expected, we must have our own ID’s.

Primary key in Parts and Bikes

In our conceptual model we only selected the bike and part codes as primary keys, however this could lead to an issue where two separate manufacturers have two parts or bikes of the same code. Later on in our logical model, we implemented Manufacturer_ID into Parts and Bikes so that this is a non-issue.

Cardinality

Relationship	Cardinality
RepairJobs → Parts	Many to Many
Customer → RepairJobs (Weak)	One to Many
Bikes → RepairJobs	One to Many
Bikes → Parts	Many to Many
Manufacturer → Bikes	One to Many
Manufacturer → Parts	One to Many

The cardinality of the conceptual model has mostly one to many relationships, but there are two exceptions with bikes to parts and repair jobs to parts, which refers to the fact that many parts can be compatible with many bikes and vice versa. The repair to parts is many to many as one part can be used for many repair jobs, and one repair job can use many parts.Customer to repair jobs has to be one to many as there can only be one customer per job. Bikes to repair jobs have to be one to many since only one bike can be assigned to a job. And with manufacturers there can only be a single one per part or bike.

Participation

Relationship	Participation
RepairJobs → Parts	Both Partial participation
Customer → RepairJobs (Weak)	Both Total participation
Bikes → RepairJobs	Both Total participation
Bikes → Parts	Both Partial participation
Manufacturer → Bikes	Both Total participation
Manufacturer → Parts	Partial to Total participation

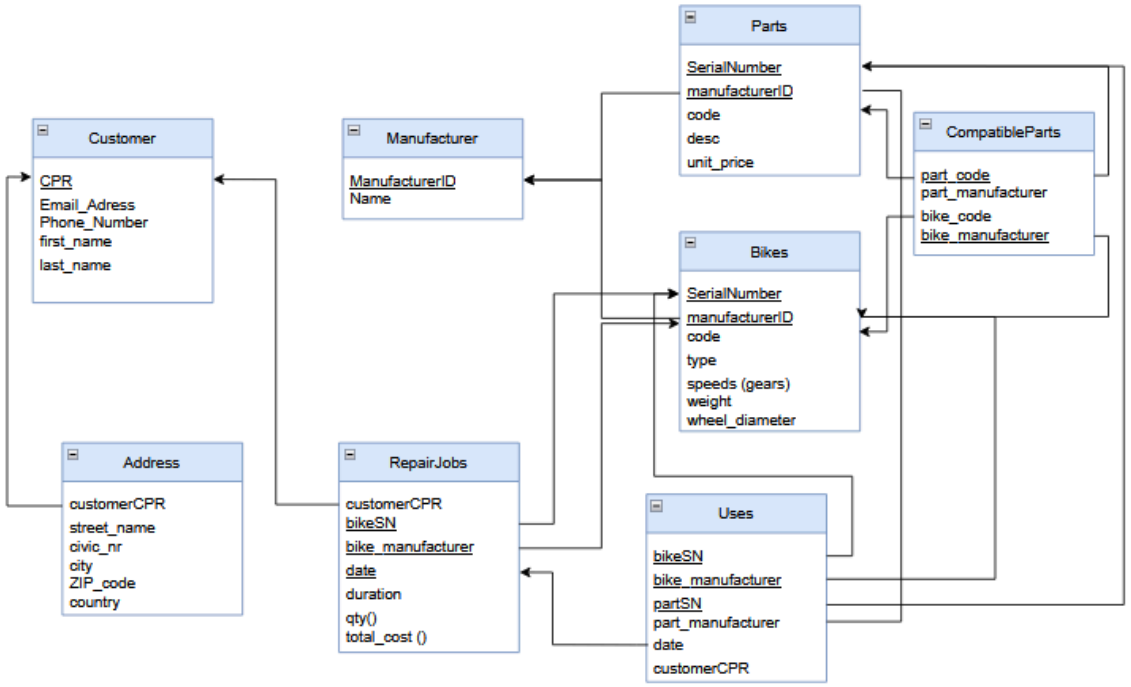
The participation of Bikes to Repair Jobs may not seem immediately obvious, because it requires an assumption. It makes sense that all repair jobs must have a bike that is repaired. But the question of whether all bikes in the database must have a repair job depends on whether the bike shop only has bikes that have been or are being repaired. We have decided that this assumption is reasonable, especially considering that it is specified that the shop is a

repair shop, and not a vendor for selling bikes. For the rest it's fairly simple. Parts can exist without being used in a repair job, and vice versa. Customers have to have a repair job to exist in the database, and each repair job must have a customer. Bikes and parts don't need to be compatible per default. For manufacturers, all bikes and parts need a manufacturer and all manufacturers need to produce either parts, bikes or both as specified in the requirements. Thus its a partial participation from manufacturers to either parts or bikes.

2. Logical Design

In this phase, we begin with the logical design, where we basically translate our conceptual model into a structure that could actually be implemented in SQL. During this process we went through several versions of the logical model, each improving on problems we identified as we got deeper and deeper into the assignment requirements. Down below we explain in detail the evolution of our models.

Retired model #1:



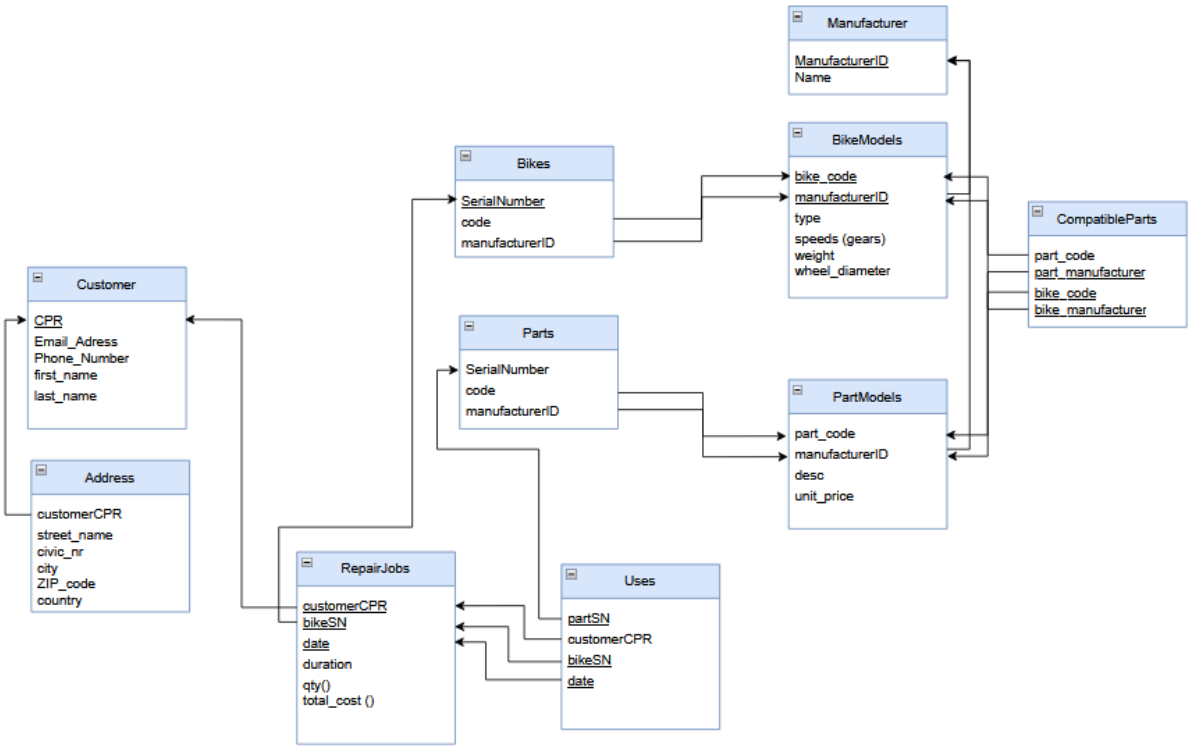
This was our first logical model, in this first attempt we introduced SerialNumber as the primary key for both bikes and parts. At this point we misunderstood the assignment and treated bikes and parts as individual items instead of models or types, which caused multiple issues:

- Every single bike and part would need its own serial number, leading to unnecessary repetition.

- CompatibleParts would have contained thousands of rows, because each serial number part would of course need compatibility entries.
- Uses and RepairJobs would end up highlighting individual items, not necessarily models making queries messy and a bit unrealistic if it makes sense.
- We duplicated many attributes unnecessarily fx. (part_code + serial number), failing to follow normalization rules.

The model technically worked, but it was overly complicated and did not reflect how a bike shop actually operates. That's why this model was retired.

Retired model #2:



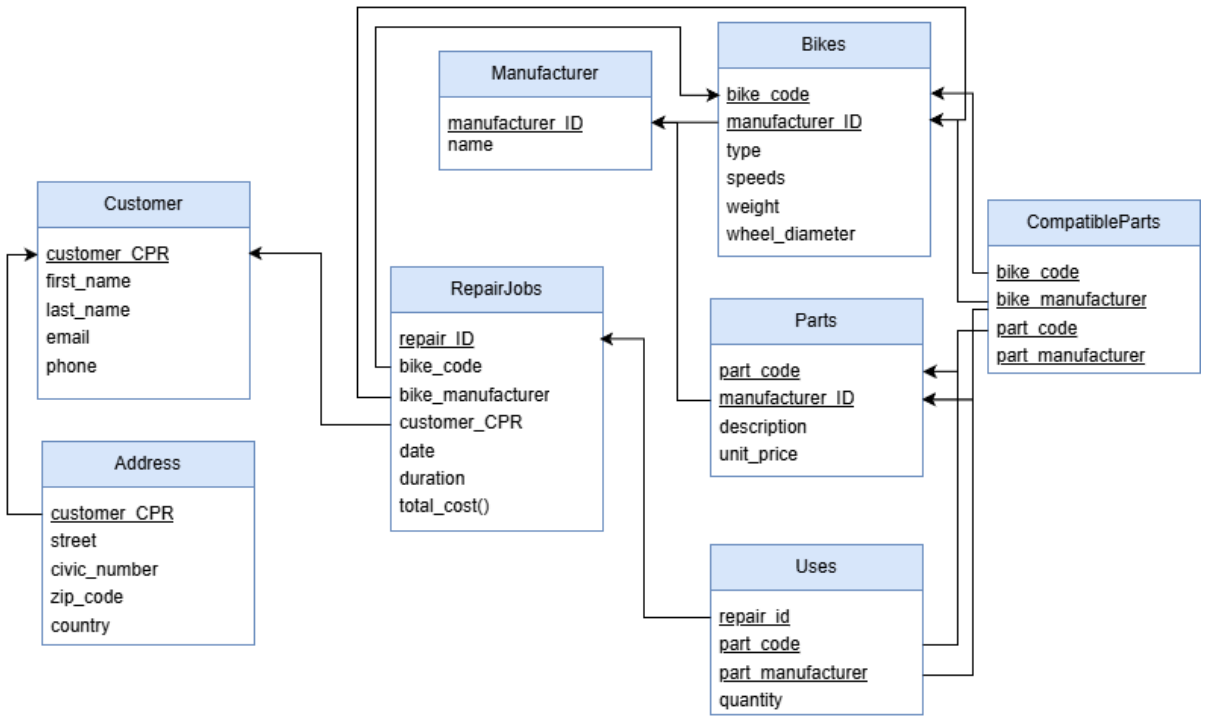
This was our second logical model, midway through converting the logical design to a database schema, we realized that compatible parts as we had it would result in a wildly complicated table with many rows, since each individual part needed to be stored with all its attributes even though a lot of it would be shared amongst parts of the same type. Bikes were the same way, which led to this re-design. Shortly this solved some issues but unfortunately introduced new ones:

- We had two tables per entity, duplicating most attributes, fx. (code, manufacturer ID or description).
- It created unnecessary joins and over normalizations.
- The uses table became a bit confusing, we got stuck on if a repair should reference the serial number or the model.

- CompatibleParts still needed to reference model-level data, making the instance tables more overlapping.

Therefore we realized we were still treating bikes like inventory management items, even though the assignment never asked us to track physical stock, and therefore this version was retired as well.

Up to date model #3:



This is our final logical model. At this point we had discovered and fixed the serial number mistake, as we realized that tracking individual bikes and parts was not required in the assignment. Instead, both bikes and parts are stored as models, identified by:

(bike_code, manufacturer_ID) for Bikes

(part_code, manufacturer_ID) for Parts

Meaning that in this case, a bike shop repairs models and not individual units. A part code combined with its manufacturer uniquely identifies a part type, and there is no need to track physical units or stock. Which gives us a clean normalized schema.

Key improvements in final model #3.

1. Composite Primary Keys for bikes and Parts:

Using (`code, manufacturer_ID`) ensures uniqueness without even needing artificial serial numbers. Meaning that two manufacturers can reuse the same bike or part code. But the same manufacturer cannot reuse the code, which is perfect for real businesses.

2. RepairJobs now uses a surrogate key `repair_ID`:

Well, so originally we considered using (`customer`, `date` and `bike`) as keys, but that breaks down when, a customer repairs two identical bikes on the same day or the shop wants to create a draft repair before completing the details, so that is why we used a surrogate repair_ID that avoids all these issues.

3. CompatibleParts uses full primary keys:

Using both bike and part codes together with their manufacturer IDs ensures 0 ambiguous compatibility entries and 0 risk of mixing manufacturers.

4. Uses table is now clean and minimal:

It only stores (`repair_ID`), (`part_code + part_manufaturer`) and (`quantity`). This makes it easy to compute total cost and keeps relationships simple.

5. Address separated from Customer:

We kept the Address in a separate table because, once customer has exactly one address (1:1). ADdress attributes do not belong in the core Customer entity. This keeps Customer smaller and more focused.

6. Full normalization (third normal form 3NF):

So last but not least, all the attributes depend directly on the primary key of their table, there is no overlapping and no repeating groups.

3. Implementation

In this part of the report we aim to explain all the implementation decisions we made during the development of the Bike Shop database. We describe the structure of the Database Schema and how we converted it from the logical model into the Data Definition Language that creates the database.

When we first started implementing the database in MariaDB the goal was to follow our logical design as accurately as possible, since we wanted all the technical decisions to make sense from a practical perspective while still matching the requirements.

Database Schema Definition in DDL using SQL:

Before we define the DB tables we first must create the database:

```
SET SQL_SAFE_UPDATES = 0;
DROP DATABASE IF EXISTS BikeRepairShop;
CREATE DATABASE BikeRepairShop;
USE BikeRepairShop;
```

- Disables safe updates so we can use Delete and other functions later

- DROPs the database if it already exists
- CREATEs empty DB named “BikeRepairShop”
- USEs that DB.

3.1 Customer & Address

```
CREATE TABLE Customer (
    customerCPR     CHAR(10),
    first_name      VARCHAR(20),
    last_name       VARCHAR(20),
    email_address   VARCHAR(73),
    phone_number    INT NOT NULL,

    PRIMARY KEY(customerCPR)
);
```

We convert from the logical model and define 5 attributes within the Customer table. CHAR with length 10 is chosen as all Danish CPR numbers have the length of 10 characters e.g {“0101251234”}. These are of course all numbers but since they can begin with 0 we choose CHAR to avoid issues with leading zeros. We also assign the NOT NULL constraint to the phone nr. attribute as it would be unwise to have a customer with no contact information.

```
CREATE TABLE Address (
    customerCPR     CHAR(10),
    street_name     VARCHAR(64),
    civic_number    VARCHAR(6),
    city            VARCHAR(64),
    ZIP_code        CHAR(4),

    PRIMARY KEY(customerCPR),
    FOREIGN KEY(customerCPR)
        REFERENCES Customer(customerCPR)
        ON DELETE CASCADE
);
```

As described in the logical conversion from the conceptual ER model diagram we split the address into a separate table. Within this table we must have the customers CPR number as a foreign key that references the Customer table. If a customer is deleted we cascade all addresses related to that customer, since it would no longer be relevant to store that information. The customer CPR is also the primary key of this table.

The rest of the attributes is the actual address information. ZIP code is another place where we chose CHAR instead of INT even though zip codes are always numbers in Denmark. But one example of a ZIP code with a leading zero is DR Byen : “0999.”

3.2 Manufacturer

```
CREATE TABLE Manufacturer (
    manufacturer_ID     INT,
    manufacturer_name   VARCHAR(64),

    PRIMARY KEY(manufacturer_ID)
);
```

The manufacturer table is defined by two attributes. A surrogate key “manufacturer_ID” and the name of the manufacturer. As mentioned in the logical conversion there can exist multiple manufacturers with the same name, and thus an ID is warranted. We chose not to use AUTO INCREMENT for the ID as this was heavily discouraged in the assignment we were given. However it also makes sense to let the bikeshop define their own terms for the ID and generate the ID upon insertion rather than letting the DBMS handle the generation, as it can become troublesome to manage as manufacturers are added/removed over time.

3.3 Bikes

```
CREATE TABLE Bikes (
    bike_code       VARCHAR(10),
    manufacturer_ID INT,
    type            VARCHAR(20),
    speeds          INT,
    weight          DECIMAL(5,2),
    wheel_diameter  DECIMAL(4,1),

    PRIMARY KEY (bike_code, manufacturer_ID),

    FOREIGN KEY (manufacturer_ID)
        REFERENCES Manufacturer(manufacturer_ID)
        ON DELETE CASCADE
);
```

The bikes table contains all relevant information about the bike models that the shop repairs. The attribute bike_code is assigned the data type VARCHAR(10) to account for both numerical codes as well as codes with characters, symbols and varying lengths. We combine bike_code and manufacturer_ID into a composite primary key as those two are enough to differentiate two bikes. Just using bike_code is insufficient because we cannot reasonably assume that two manufacturers will never assign two products to the same bike_code, but the same manufacturer can be expected to have unique codes.

We use cascade on delete to ensure that the bikes that are offered by a manufacturer are also removed when that manufacturer is removed. The remaining attributes describe the specifications of the bike and we assign the domain constraints such as DECIMAL for weight and wheel diameter to support fractional values.

3.4 Parts

```
CREATE TABLE Parts (
    part_code       VARCHAR(10),
    manufacturer_ID INT,
    description     VARCHAR(256),
    unit_price      DECIMAL(8,2),

    PRIMARY KEY (part_code, manufacturer_ID),

    FOREIGN KEY (manufacturer_ID)
        REFERENCES Manufacturer(manufacturer_ID)
        ON DELETE CASCADE
);
```

The parts table represents all bike parts that are purchased from their manufacturers. As with bikes, the part_code is a VARCHAR since no fixed format is specified. We use DECIMAL for unit_price to account for the standardised 2 decimal places with pricing. The description using a VARCHAR of max length 256 should permit a reasonably long description while avoiding unnecessary over-allocation.

By the same principle as with bikes we also used a composite key as the primary key and used a foreign key manufacturer_ID that cascades on deletion of the referenced manufacturer.

3.5 Repair Jobs

```
CREATE TABLE RepairJobs (

    repair_ID           INT,
    bike_code           VARCHAR(10) NOT NULL,
    bike_manufacturer   INT NOT NULL,
    customerCPR         CHAR(10) NOT NULL,
    repair_date         DATE,
    duration            INT,

    total_cost DECIMAL(8,2),

    PRIMARY KEY (repair_ID),

    FOREIGN KEY (customerCPR)
        REFERENCES Customer(customerCPR),

    FOREIGN KEY (bike_code, bike_manufacturer)
        REFERENCES Bikes(bike_code, manufacturer_ID)
);
```

The RepairJobs table captures each individual repair performed by the shop. The key attribute is repair_id which is another surrogate key. This key is created because no natural combination of attributes (such as bike, date, and customer) can guarantee uniqueness. Once again we dont use Auto Incrementation as per the assignment policy. We assume the shop manages the ID generation.

From our conceptual and logical model we can see that repair jobs participate in several relationships:

- Weakly dependent on Customer (Total participation), aka every repair job has exactly one customer and since it's a one to many relationship it's only ever one customer per repair job. Thus to enforce the Total participation from customers, the CPR domain must be constrained to not null as it's also not the primary key and otherwise could be.
- It references a bike from the bikes table with a Total to partial participation, where its total participation on the side of the repair jobs. This also needs to have the not null constraint for the same reason as customer CPR.

We also have our derived attribute total_cost. This is set to null in our schema as we do not yet have the data required to compute it. This will be done using a function that for any tuples/rows in the table its total_cost can be computed using the uses table that relates to repair jobs and stores the parts that are used for each job. You could also use a trigger for this if you want to automatically update them when you update the uses table (add/remove parts from a job would affect the total_cost).

3.6 Uses

```
CREATE TABLE Uses (
    repair_ID           INT,
    part_code           VARCHAR(10),
    part_manufacturer   INT,
    quantity            INT,

    PRIMARY KEY (repair_ID, part_code,part_manufacturer),

    FOREIGN KEY (repair_ID)
        REFERENCES RepairJobs(repair_ID)
        ON DELETE CASCADE,

    FOREIGN KEY (part_code, part_manufacturer)
        REFERENCES Parts(part_code,manufacturer_ID)
);
```

The Uses table represents the many-to-many relationship between RepairJobs and Parts. A single repair job may use several parts and a single part may be used in many different repairs. The composite primary key ensures that the same part-manufacturer combination cannot be recorded twice for the same repair job. Both relationships between RepairJobs and Parts are partial participation, meaning a repair job may have zero parts, and a part does not have to be used in any job. We include ON DELETE CASCADE in the repair job foreign key as if a job is deleted it makes sense to also delete the used tuples as they are no longer relevant. We do not allow this for parts since a job may still exist that requires that information.

3.7 Compatible Parts

```
CREATE TABLE CompatibleParts (
    bike_code   VARCHAR(10),
    part_code   VARCHAR(10),
    bike_manufacturer INT,
    part_manufacturer INT,


    PRIMARY KEY (bike_code, part_code,bike_manufacturer,part_manufacturer),

    FOREIGN KEY (bike_code,bike_manufacturer)
        REFERENCES Bikes(bike_code,manufacturer_ID)
        ON DELETE CASCADE,

    FOREIGN KEY (part_code,part_manufacturer)
        REFERENCES Parts(part_code,manufacturer_ID)
        ON DELETE CASCADE
);
```

This table is a many-to-many relation between bikes and parts since e.g a bike may have many parts it can use, and a part may be compatible with many bikes. In our conceptual model, we showed this relation as a partial to partial participation as no part has to be compatible with a bike, and vice versa.

The primary key for this table is a composite key consisting of the primary key from bikes and from parts. For the references we again use cascade on delete since the compatibility is void if either the part or the bike in that relationship no longer exists in the database.

4. Database Instance

4.1 Introduction

In this part we explain our data insertions as well as why we have written the data the way we have.

4.2 Database insertions

When writing our data we wanted to make sure that it would represent realistic data for a repair shop. We also cover a lot of edge cases with our data, we do this to show that the database is flexible when it comes to things like longer than average names and such.

4.2.1 Customer

In Customer we represent our data as follows:

```
customerCPR     CHAR(10),
first_name      VARCHAR(20),
last_name       VARCHAR(20),
email_address   VARCHAR(73),
phone_number    INT NOT NULL,
```

For customerCPR we have chosen to use the data type CHAR with a set length of 10, we chose this due to CPR-numbers always being 10 digits.

For first_name we have chosen a VARCHAR of max length 20, we have chosen a max length that covers 99% of first names.

For last_name we have chosen a VARCHAR of max length 20, we do this for the same reason as first_name.

For email_address we chose a VARCHAR of max length 73 to really make sure we have covered any realistic email length.

For phone_number we have chosen an INT, with the only limitation being that it cannot be null, as this is a Danish business and because of CPR-numbers being stored, we reasonably assume that a customer also uses a Danish phone number.

SQL data insertion:

```
INSERT INTO Customer VALUES
('1308981817','John','Name','John@gmail.com',22334455),
('1212824762','Jane','Doe','JaneTheDoeWithLongEmailAddress@gmail.com',12312312),
('3108789976','Joergen','Von Einer','abc@gmail.com',11111123),
('0101014321','Mads','Andersen','MAndersen@outlook.com',12345678),
('1212988825','Emma','Sky','emmasky@gmail.com',87654321),
('2307127139','Jan','Mikkelson','Unreasonablylongemailthatiswaytohardtorememberinownhead@gmail.com',88884312),
('1203112231','Mia','Mikkelson','mia23153@outlook.com',88776655),
('1606047842','Karl Emil','Kraghe','altdetgode@gmail.com',23234490),
('1104671111','Maria','Nielsen','Nielsen.m@outlook.com',11992288),
('0102037755','Ella','Larsen','elllar@gmail.com',12323556),
('1309092385','Agnes','Andersen','Hemmeligperson@gmail.com',27346892),
('1010828850','Valdemar','Pedersen','Valde.ped27823@outlook.com',49027643),
('0710299836','Matheo','Soerensen','2324234MS@gmail.com',38127374),
('1111111111','Esther','Jensen','Estherjensen@outlook.com',22736475),
('2212978765','Hannah','Larsen','hannnnnnnnnnnnnnnnnnnah83457@gmail.com',18239102);
```

In our data insertion we have chosen to include a very long email as an edge case.

SELECT * FROM query result:

customerCPR	first_name	last_name	email_address	phone_number
0101014321	Mads	Andersen	MAndersen@outlook.com	12345678
0102037755	Ella	Larsen	elllar@gmail.com	12323556
0710299836	Matheo	Soerensen	2324234MS@gmail.com	38127374
1010828850	Valdemar	Pedersen	Valde.ped27823@outlook.com	49027643
1104671111	Maria	Nielsen	Nielsen.m@outlook.com	11992288
1111111111	Esther	Jensen	Estherjensen@outlook.com	22736475
1203112231	Mia	Mikkelson	mia23153@outlook.com	88776655
1212824762	Jane	Doe	JaneTheDoeWithLongEmailAddress@gmail.com	12312312
1212988825	Emma	Sky	emmasky@gmail.com	87654321
1308981817	John	Name	John@gmail.com	22334455
1309092385	Agnes	Andersen	Hemmeligperson@gmail.com	27346892
1606047842	Karl Emil	Kraghe	altdetgode@gmail.com	23234490
2212978765	Hannah	Larsen	hannnnnnnnnnnnnnnnnnnah83457@gmail.com	18239102
2307127139	Jan	Mikkelson	Unreasonablylongemailthatiswaytohardtorem...	88884312
3108789976	Joergen	Von Einer	abc@gmail.com	11111123

4.2.2 Address

In Address we represent data as follows:

```
customerCPR   CHAR(10),
street_name   VARCHAR(64),
civic_number  VARCHAR(6),
city          VARCHAR(64),
ZIP_code      CHAR(4),
```

customerCPR is represented the same way as in Customer

street_name is represented by a VARCHAR of max length 64, this is almost double the longest street name in Denmark and will therefore cover every possible danish address.

civic_number is represented by a VARCHAR of max length 6, we choose to not use an INT since some civic numbers are represented with letters e.g ‘43A’, we have also chosen the length 6 as some civic numbers can be 4 numbers, the 2 extra characters are for rare cases of longer numbers.

city is represented by a VARCHAR of max length 64, the length may be a bit overkill as city names are often not very long but we would in this case rather be safe.

ZIP_code is represented by a CHAR of length 4, every danish ZIP code is exactly 4 numbers. We chose to use a CHAR instead of an INT sinde some ZIP codes start with the number 0, which an INT cant.

```
INSERT INTO Address VALUES
('1308981817','Roskildevej','12','Haderslev','6070'),
('1212824762','Hovedvej','4A','Roskilde','4000'),
('3108789976','Faglaertvej','67','Herlev','2730'),
('0101014321','Landevej','419','Slagelse','4270'),
('1212988825','Uhyggeligvej','7B','Herning','6933'),
('2307127139','Funktionelletbanevej','9980C','Aarhus','8000'),
('1203112231','Funktionelletbanevej','9980C','Aarhus','8000'), -- shared address
('1606047842','Havvej','21','Skagen','9990'),
('1104671111','Nyvej','420','Ishoej','2625'),
('0102037755','Strandvej','152','Rungsted','2960'),
('1309092385','Bagsværdvej','99','Bagsvaerd','2800'),
('1010828850','Ballerupvej','31D','Ballerup','2740'),
('0710299836','Kongensvej','1','Kokkedal','2970'),
('1111111111','En rigtig vej med utrolig langt navn som en test af grænse på 64','12345','Helsinge','0999'),
('2212978765','Denforkertevej','54321','Koebenhavn S','2300');
```

We have chosen to have two different people living at the same address, this is to show that the database can identify unique persons. We have also made up a very long street name as an edge case.

SELECT * FROM query result:

customerCPR	street_name	civic_number	city	ZIP_code
0101014321	Landevej	419	Slagelse	4270
0102037755	Strandvej	152	Rungsted	2960
0710299836	Kongensvej	1	Kokkedal	2970
1010828850	Ballerupvej	31D	Ballerup	2740
1104671111	Nyvej	420	Ishoej	2625
1111111111	En rigtig vej med utrolig langt navn som en test ...	12345	Helsinge	0999
1203112231	Funktionelletbanevej	9980C	Aarhus	8000
1212824762	Hovedvej	4A	Roskilde	4000
1212988825	Uhyggeligvej	7B	Herning	6933
1308981817	Roskildevej	12	Haderslev	6070
1309092385	Bagsværdvej	99	Bagsvaerd	2800
1606047842	Havvej	21	Skagen	9990
2212978765	Denforkertevej	54321	Koebenh...	2300
2307127139	Funktionelletbanevej	9980C	Aarhus	8000
3108789976	Faglaertvej	67	Herlev	2730

4.2.3 Manufacturer

In Manufacturer we represent data as follows:

```
manufacturer_ID     INT,
manufacturer_name   VARCHAR(64),
```

manufacturer_ID is represented by a INT, we have chosen to give IDs of 5 digits, an INT represents that with no problem. We chose to give IDs with a length of 5 digits, this gives 6 million possibilities, which should be more than enough.

manufacturer_name is represented by a VARCHAR of max length 64, this covers all realistic names a manufacturer may have.

SQL data insertion:

```
INSERT INTO Manufacturer VALUES
(11111, 'Bike Parts'),
(22222, 'Bike Parts'),
(33333, 'Only Bikes'),
(43126, 'All Things Bikes'),
(12372, 'Quality Bikes'),
(55555, 'Part Performance'),
(98765, 'Nordic Speed');
```

We did not include any extremes here as it shows the same kind of edge case from Customer and Address.

SELECT * FROM query result:

manufacturer_ID	manufacturer_name
11111	Bike Parts
12372	Quality Bikes
22222	Bike Parts
33333	Only Bikes
43126	All Things Bikes
55555	Part Performance
98765	Nordic Speed

4.2.4 Bikes

In Bikes we represent data as follows:

```
bike_code       VARCHAR(10),
manufacturer_ID INT,
type            VARCHAR(20),
speeds          INT,
weight          DECIMAL(5,2),
wheel_diameter  DECIMAL(4,1),
```

bike_code is represented by a VARCHAR of max length 10, we use VARCHAR because our bike codes include both numbers and letters.

manufacturer_ID is represented the same way as in Manufacturer.

type is represented as a VARCHAR of max length 20, this length is enough to cover all types of bikes that exist.

speeds is represented by an INT, we have chosen INT because speeds is measured by a number.

weight is represented by a DECIMAL with the specifics (5,2), we have chosen these limits as a reasonable bike shouldn't weight more that 5 digits worth, the decimals are to show precise weights. Weight is measured in KGs.

wheel_diameter is represented by a DECIMAL with the specifics (4,1), we have chosen these limits as they make sense for a reasonable bike. Diameter is measured in inches.

```
INSERT INTO Bikes VALUES
('BK-A100',11111,'Mountain Bike',18,14.50,27.5),
('BK-B200',22222,'Mountain Bike',18,14.55,27.5), -- nearly identical bike from different manufacturer
('BK-C300',43126,'Road Bike',22,9.50,28.0),
('BK-D400',33333,'Hybrid',21,12.10,27.0),
('BK-E500',12372,'Electric',8,24.00,29.0),
('BK-F600',98765,'City',7,11.20,26.0);
```

We have chosen to have two identical bikes from two different manufacturers, this is to show an edge case where these aspects may match.

SELECT * FROM query result:

bike_code	manufacturer_ID	type	speeds	weight	wheel_diameter
BK-A100	11111	Mountain Bike	18	14.50	27.5
BK-B200	22222	Mountain Bike	18	14.55	27.5
BK-C300	43126	Road Bike	22	9.50	28.0
BK-D400	33333	Hybrid	21	12.10	27.0
BK-E500	12372	Electric	8	24.00	29.0
BK-F600	98765	City	7	11.20	26.0

4.2.5 Parts

In Parts we represent data as follows:

```
part_code       VARCHAR(10),
manufacturer_ID INT,
description     VARCHAR(256),
unit_price      DECIMAL(8,2),
```

part_code is represented by a VARCHAR of max length 10, we use VARCHAR because our part codes include both numbers and letters.

manufacturer_ID is represented the same way as in Manufacturer.

description is represented by a VARCHAR of max length 256, we chose a long length as it is supposed to be a short description of the part.

unit_price is represented by a DECIMAL with the specifics (8,2), we chose 8 as the max primary length as no bike part should be more expensive than that, the decimals are for value less than 1 crown.

```
INSERT INTO Parts VALUES
('P-100',11111,'Bike chain 9-speed - stainless steel, rust resistant, designed for mountain bikes',120.50),
('P-101',22222,'Bike chain 9-speed - identical name but different manufacturer',119.99),
('P-200',11111,'Front wheel 27.5 inch, reinforced rim, compatible with BK-A100',340.00),
('P-300',55555,'Hydraulic brake lever with ergonomic grip and anti-slip design',290.00),
('P-400',55555,'Bike seat ergonomic with double padding and weatherproof leather cover',150.00),
('P-500',43126,'Pedal set with reflectors, alloy build',180.00),
('P-600',98765,'Universal inner tube, 26-29 inch range',90.00),
('P-123',11111,'Front wheel 20 inch, standard rim, universal',100.00);
```

We only have standard data for this table.

SELECT * FROM query result:

part_code	manufacturer_ID	description	unit_price
P-100	11111	Bike chain 9-speed - stainless steel, rust resista...	120.50
P-101	22222	Bike chain 9-speed - identical name but differen...	119.99
P-123	11111	Front wheel 20 inch, standard rim, universal	100.00
P-200	11111	Front wheel 27.5 inch, reinforced rim, compatibl...	340.00
P-300	55555	Hydraulic brake lever with ergonomic grip and a...	290.00
P-400	55555	Bike seat ergonomic with double padding and w...	150.00
P-500	43126	Pedal set with reflectors, alloy build	180.00
P-600	98765	Universal inner tube, 26-29 inch range	90.00

4.3.6 RepairJobs

In RepairJobs we represent data as follows:

```
repair_ID           INT,
bike_code           VARCHAR(10) NOT NULL,
bike_manufacturer   INT NOT NULL,
customerCPR         CHAR(10) NOT NULL,
repair_date         DATE,
duration            INT,
total_cost DECIMAL(8,2),
```

repair_ID is represented by an INT, repair IDs are manually assigned and are unique to a repair job.

bike_code is represented the same way as in Bikes, with the addition of it being NOT NULL, we do this since there needs to be a bike for it to be repaired.

bike_manufacturer is represented as a VARCHAR of max length 10 that also is NOT NULL, it is the manufacturer ID connected to the bike.

customerCPR is represented the same way as in Customer, with the addition of it being NOT NULL.

repair_date is represented by a DATE, marking the date the repair job started and duration is represented by an INT, takes duration in the context of minutes.

total_cost is represented by a DECIMAL with the specifics (8,2), we chose the lengths for the same reasons as unit_cost in Parts, total_cost is initialised as null and is updated later with a function.

```
INSERT INTO RepairJobs VALUES
(1,'BK-F600',98765,'1308981817','2023-12-19',90,NULL),
(2,'BK-B200',22222,'1212824762','2024-03-15',60,NULL),
(3,'BK-C300',43126,'3108789976','2024-04-10',120,NULL),
(4,'BK-D400',33333,'0101014321','2025-04-12',45,NULL),
(5,'BK-E500',12372,'1212988825','2025-05-01',180,NULL),
(6,'BK-A100',11111,'1203112231','2025-05-01',100,NULL),
(7,'BK-A100',11111,'1203112231','2025-05-15',25,NULL);
```

Standard data, total_cost is null for all RepairJobs as total_cost function has not been run yet.

SELECT * FROM query result:

repair_ID	bike_code	bike_manufacturer	customerCPR	repair_date	duration	total_cost
1	BK-F600	98765	1308981817	2023-12-19	90	NULL
2	BK-B200	22222	1212824762	2024-03-15	60	NULL
3	BK-C300	43126	3108789976	2024-04-10	120	NULL
4	BK-D400	33333	0101014321	2025-04-12	45	NULL
5	BK-E500	12372	1212988825	2025-05-01	180	NULL
6	BK-A100	11111	1203112231	2025-05-01	100	NULL
7	BK-A100	11111	1203112231	2025-05-15	25	NULL

4.3.7 Uses

In Uses we represent data as follows:

```
repair_ID   INT,
part_code   VARCHAR(10),
part_manufacturer INT,
quantity    INT,
```

repair_ID is represented the same way as in RepairJobs

part_code is represented the same way as in Parts

part_manufacturer is represented by an INT and is connected to parts the same way bike_manufacturer is connected to bikes.

quantity is represented by an INT, we chose this since quantities are counted in whole numbers.

```
INSERT INTO Uses VALUES
(1,'P-300', 55555, 1),
(1,'P-600', 98765, 2),
(2,'P-101', 22222 ,1),
(3,'P-400', 55555, 1),
(4,'P-500', 43126, 1),
(5,'P-600', 98765, 1),
(6,'P-100', 11111, 3),
(6,'P-200', 11111, 1);
```

Standard data, no edge cases.

SELECT * FROM query result:

repair_ID	part_code	part_manufacturer	quantity
1	P-300	55555	1
1	P-600	98765	2
2	P-101	22222	1
3	P-400	55555	1
4	P-500	43126	1
5	P-600	98765	1
6	P-100	11111	3
6	P-200	11111	1

4.3.8 CompatibleParts

In CompatibleParts we represent our data as follows:

```
bike_code   VARCHAR(10),
part_code   VARCHAR(10),
bike_manufacturer INT,
part_manufacturer INT,
```

bike_code is represented the same way as in Bikes.

part_code is represented the same way as in Parts

bike_manufacturer is represented the same way as in Repairjobs

part_manufacturer is represented the same way as in Uses.

```
INSERT INTO CompatibleParts VALUES
('BK-A100','P-100',11111, 11111),
('BK-A100','P-200',11111, 11111),
('BK-B200','P-101',22222, 22222),
('BK-C300','P-400',43126, 55555),
('BK-D400','P-500',33333, 43126),
('BK-E500','P-400',12372, 55555),
('BK-E500','P-600',12372, 98765),
('BK-F600','P-600',98765, 98765),
('BK-F600','P-300',98765, 55555);
```

Standard data.

SELECT * FROM query result:

bike_code	part_code	bike_manufacturer	part_manufacturer
BK-A100	P-100	11111	11111
BK-A100	P-200	11111	11111
BK-B200	P-101	22222	22222
BK-C300	P-400	43126	55555
BK-D400	P-500	33333	43126
BK-E500	P-400	12372	55555
BK-E500	P-600	12372	98765
BK-F600	P-300	98765	55555
BK-F600	P-600	98765	98765

5. SQL Table Modification

5.1 Introduction

This part shows how we modified some data in our BikeRepairShop database, through the following SQL Data Manipulation Language commands:

INSERT, UPDATE and DELETE.

In our case we made a fictive use case scenario, with the bare purpose of illustrating how our database handles realistic day to day business operations in the context of "Cykelparadis", such as registering new parts, updating customer information, and deleting old or unnecessary records. And we made it with two examples of each DML command.

5.2 Use case scenario

A returning customer, named "John Name" with CPR "1308981817", visits the shop with his bike (BK-F600) for repair.

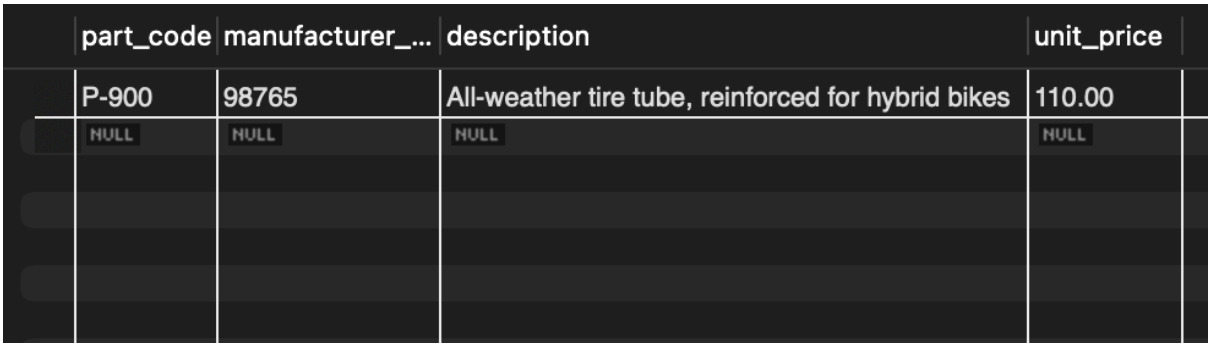
During this process, several operations happen in the system.
The workshop receives a new type of tire tube from Nordic Speed (98765) and it is added to the database, the part is then linked to John's bike in the compatibility table, and then the shop or mechanic updates John's phone number in the customer register, and the total cost of the repair job. Finally, all old irrelevant records such as a test repair and discounted parts are removed to keep the system clean and up to date. In the following sub-chapter (5.3) you can follow the process.

5.3 Queries

– 1st. COMMAND: INSERT STATEMENT –

Example 1 + output:

```
-- The shop receives a new type of tire tube from manufacturer 98765
("Nordic Speed")

-- and registers it in the Parts table so it can be used in future
repairs.

INSERT INTO Parts (part_code, manufacturer_ID, description, unit_price)

VALUES ('P-900', 98765, 'All-weather tire tube, reinforced for hybrid
bikes', 110.00);

-- so we verify the insertion

SELECT * FROM Parts

WHERE part_code = 'P-900' AND manufacturer_ID = 98765;
```



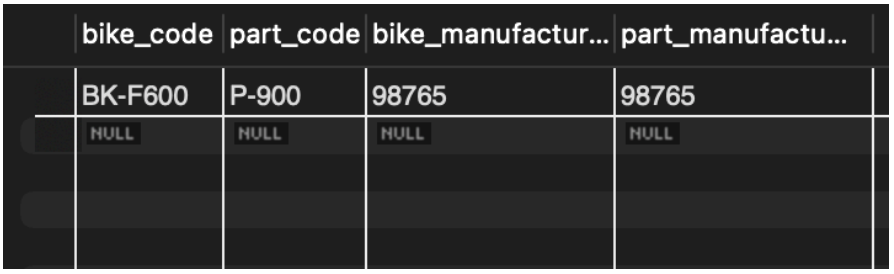
part_code	manufacturer_...	description	unit_price
P-900	98765	All-weather tire tube, reinforced for hybrid bikes	110.00
NULL	NULL	NULL	NULL

This query simulates a situation where the bike shop receives a new part from one of its manufacturers, in this case as shown it is “Nordic Speed” with the `ID 98765`.

The new part `P-900` is registered in the database with the description and price. So we verify with `SELECT` and confirm that the new record has been added correctly to the Parts table.

Example 2 + output:

```
-- The new part is also marked as compatible with John's bike (BK-F600)

INSERT INTO CompatibleParts (bike_code, part_code, bike_manufacturer,
part_manufacturer)

VALUES ('BK-F600', 'P-900', 98765, 98765);

-- verify the new compatibility record

SELECT * FROM CompatibleParts

WHERE bike_code = 'BK-F600' AND part_code = 'P-900';
```



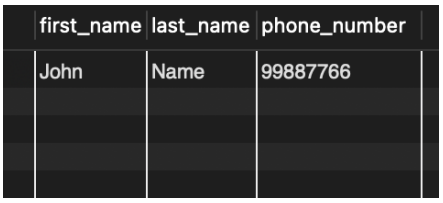
bike_code	part_code	bike_manufactur...	part_manufactu...
BK-F600	P-900	98765	98765
NULL	NULL	NULL	NULL

Once the new part has been added, it must be, like associated with the type of bike it fits with. The query inserts a record into the CompatibleParts table, linking the part P-900 with the bike BK-F600, that is also produced by the same manufacturer named previously, and finally again we use `SELECT` to output that the relationship has been successfully inserted.

– 2nd. COMMAND: UPDATE STATEMENT –

Example 3 + output:

```
-- John changes his phone number; the shop updates his contact info.

UPDATE Customer

SET phone_number = '99887766'

WHERE customerCPR = '1308981817';

-- Verify that the phone number was updated

SELECT first_name, last_name, phone_number

FROM Customer

WHERE customerCPR = '1308981817';
```

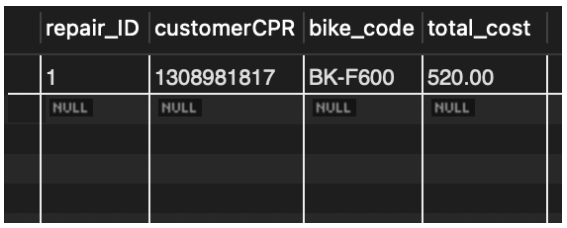


first_name	last_name	phone_number
John	Name	99887766

So this example 3, handles updates of customer information in the Customer table. In our case “John Name” has changed his phone number and of course in this case scenario the database must reflect this update to ensure accurate communication with the client. Then the verification query displays the updated phone number, confirming that the modification was successful.

Example 4 + output:

```
-- The ongoing repair job (repair_ID = 1) is updated to include total
cost.

-- This value is derived from used parts but can also be updated manually.

UPDATE RepairJobs

SET total_cost = 520.00

WHERE repair_ID = 1;

-- Verify that the total cost was updated

SELECT repair_ID, customerCPR, bike_code, total_cost

FROM RepairJobs

WHERE repair_ID = 1;
```



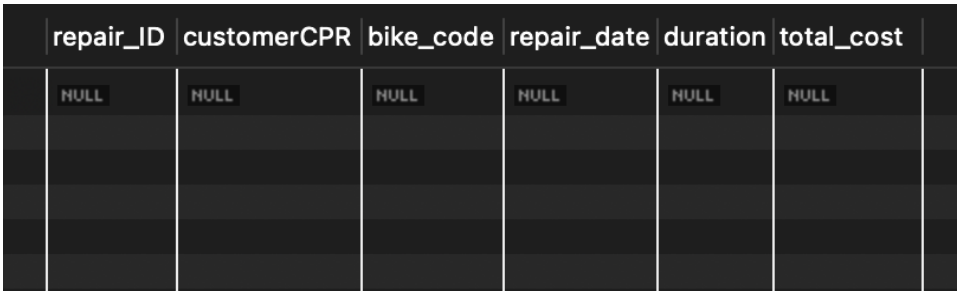
repair_ID	customerCPR	bike_code	total_cost
1	1308981817	BK-F600	520.00
NULL	NULL	NULL	NULL

As shown, this statement updates the derived attribute `total_cost` for a specific repair job. In this case, after all parts have been registered and work or labor is been calculated, the shop records the total cost of the repair. Although total_cost can later be automatically maintained by SQL functions or triggers, in this case our example shows a manual update, and outputs that the new total has been stored correctly for repair `ID 1`.

– 3rd. COMMAND: DELETE STATEMENT –

Example 5 + output:

```
-- An old test repair (repair_ID = 7) was mistakenly entered and had no
parts used.

-- The shop removes it to keep the system clean.

DELETE FROM RepairJobs

WHERE repair_ID = 7;

-- Verify that the repair was deleted

SELECT * FROM RepairJobs

WHERE repair_ID = 7;  -- Here it should return an empty result set
```



repair_ID	customerCPR	bike_code	repair_date	duration	total_cost
NULL	NULL	NULL	NULL	NULL	NULL

So during the database testing, a repair job `ID 7`, was created for validation purposes but is no longer relevant. The `DELETE` command removes it from the `RepairJobs` table, and then the verification `SELECT` confirms that no record with `ID 7` remains. By doing this query it is clear that the system can safely remove the data without affecting dependent entities, since the job contained, no associated parts or dependencies.

Example 6 + output:

```
-- Similarly a discontinued part from manufacturer 55555 ("P-300") is
removed.
-- We do it stepwise:
-- Step 1: is to remove the repair usages of that part first (to avoid FK
violation)
DELETE FROM Uses
WHERE part_code = 'P-300' AND part_manufacturer = 55555;

-- Step 2: and then delete the part itself
DELETE FROM Parts
WHERE part_code = 'P-300' AND manufacturer_ID = 55555;

-- Verify that both the part and its uses are gone
SELECT * FROM Parts
WHERE part_code = 'P-300' AND manufacturer_ID = 55555;

SELECT * FROM Uses
WHERE part_code = 'P-300' AND part_manufacturer = 55555;
```

part_code	manufacturer_ID	description	unit_price
NULL	NULL	NULL	NULL

repair_ID	part_code	part_manufacturer	quantity
NULL	NULL	NULL	NULL

This example shows how some referential integrity between two tables prevents accidental deletes. The part `P-300` from manufacturer “Part Performance” with `ID 55555`, was previously used in repair jobs. So to delete it safely, the dependent rows in the Uses table are removed first. After that, the part itself can be deleted from Parts without triggering foreign key errors. And last but not least the verification queries confirm that both the part and its related uses are gone, ensuring the database is up to date and consistent without any errors.

6. SQL Data Queries

6.1 Introduction

This part demonstrates typical queries that might be written when using the BikeRepairShop database. We have situations that might be common that we have constructed queries for.

6.2 Queries

We have 5 situations we have to solve with the use of queries, they are as follows:

- Show the CPR number of all customers who got one of their bikes repaired more than once.
- Show the code and manufacturer of all parts that were never used for any repair.
- For each part, show the code, manufacturer, and total quantity being used for all repair jobs in 2024.
- For each bike type, show the type, code and manufacturer of the most repaired bike.
- Show the code and manufacturer of bikes that can use only parts from the same manufacturer.

6.2.1 Query 1

Show the CPR number of all customers who got one of their bikes repaired more than once.

```
SELECT customerCPR FROM RepairJobs
GROUP BY customerCPR
HAVING COUNT(customerCPR)>1;
```

For this query we look in the RepairJobs table, we use a HAVING clause with a COUNT of CPR-numbers present to find CPR-numbers that are registered to more than one repair job.

The result of running the query:

	customerCPR
▶	1203112231

The query returns this CPR-number as it is the only one that shows up twice in RepairJobs.

6.2.2 Query 2

Show the code and manufacturer of all parts that were never used for any repair.

```
SELECT part_code, manufacturer_ID
FROM Parts
WHERE (part_code, manufacturer_ID) NOT IN
      (SELECT part_code, part_manufacturer FROM Uses);
```

For this query we first get a list of all the parts that exist in circulation, then we compare them to the Uses table, which lists all repair jobs and what parts each repair job has used. When comparing these two tables we choose to only take parts from the Parts table that do not appear in the Uses table, this results only unused parts being selected.

Result of running the query:

	part_code	manufacturer_ID
▶	P-123	11111

The query returns this part as it is the only part not used in any repairs.

6.2.3 Query 3

For each part, show the code, manufacturer, and total quantity being used for all repair jobs in 2024.

```
SELECT p.part_code, p.manufacturer_ID, SUM(u.quantity) AS
total_quantity
FROM Uses u, Parts p, RepairJobs r
WHERE p.part_code = u.part_code
  AND p.manufacturer_ID = u.part_manufacturer
  AND r.repair_ID = u.repair_ID
  AND r.repair_date LIKE '2024%'
GROUP BY p.part_code, p.manufacturer_ID;
```

For this query we look in the tables Uses, Parts and RepairJobs, we compare the part codes to make sure they are the correct ones, then we use the SUM function to add together the quantities of the individual parts to get a total of each unique one. We also have an AND clause that checks if the date is within the year 2024, as we only want repair jobs from that year. The resulting selection is all the unique parts, and how many were used from the year 2024.

Result of running the query:

part_code	manufacturer_ID	total_quantity
P-101	22222	1
P-400	55555	1

The query returns the part codes with manufacturer id and the quantity of each part used for the year 2024.

6.2.4 Query 4

For each bike type, show the type, code and manufacturer of the most repaired bike.

```
DROP VIEW IF EXISTS BikeRepairs;
CREATE VIEW BikeRepairs AS
SELECT bike_code, bike_manufacturer, COUNT(*) AS repairs_count
FROM RepairJobs
GROUP BY bike_code, bike_manufacturer;

SELECT bike_code, bike_manufacturer, repairs_count
FROM BikeRepairs
WHERE repairs_count = (SELECT MAX(repairs_count) FROM
BikeRepairs);
```

Result:

bike_code	bike_manufacturer	repairs_count
BK-A100	11111	2

This query first creates a view that consists of the set of repairs carried out for each individual bike. Since bikes are identified by their code and manufacturer, we naturally need to use the entire primary key. Our view “BikeRepairs” looks like this:

bike_code	bike_manufacturer	repairs_count
BK-A100	11111	2
BK-B200	22222	1
BK-C300	43126	1
BK-D400	33333	1
BK-E500	12372	1
BK-F600	98765	1

Then we can execute our select statement where we select the tuple of the view with the maximum value of repairs_count, and thus we get the bike that has been repaired most.

bike_code	bike_manufacturer	repairs_count
BK-A100	11111	2

6.2.5 Query 5

Show the code and manufacturer of bikes that can use only parts from the same manufacturer.

```
SELECT bike_code, bike_manufacturer
FROM CompatibleParts
GROUP BY bike_code, bike_manufacturer
HAVING MIN(part_manufacturer) = bike_manufacturer
   AND MAX(part_manufacturer) = bike_manufacturer;
```

Result:

	bike_code	bike_manufacturer
▶	BK-A100	11111
	BK-B200	22222

For this query we just need a select statement. We select the bike_code and bike_manufacturer from the compatible parts table and group by those two attributes. We then specify the having condition consisting of two criteria. Firstly the minimum part_manufacturer for that bike must equal the bike's manufacturer, and secondly the maximum part_manufacturer must also equal the bike's manufacturer.

The reason we use both MIN and MAX instead of simply checking part_manufacturer = bike_manufacturer is that the query is grouped by bike.

Within each group there may be multiple rows, each representing a different part. If we only wrote a simple condition like WHERE part_manufacturer = bike_manufacturer, we would be checking it per row, not per bike, and we would not detect whether all parts match the manufacturer, only that some do. By comparing both the minimum and maximum values in the group, we ensure that if any part belongs to a different manufacturer, then either MIN or MAX would differ from the bike's manufacturer

7. SQL Programming

7.1 Introduction

In this part we show the SQL features we used to handle some of the automatic tasks in the BikeRepairShop database. We worked with the following three constructors:

1. Function that calculates the total cost of each repair job dynamically.
2. Procedure inserts parts into repair jobs only when they are compatible with the bike being repaired.
3. Trigger validates repair jobs before insertion to prevent unrealistic or invalid records.

7.2 SQL Function - `totalCost`

The function `totalCost()` computes the derived attribute `total_cost` in the `RepairJobs` table by summing the cost of all parts used during a repair:

```
-- Lets check repair jobs first. Should be all nulls in cost
SELECT * FROM RepairJobs;
DROP FUNCTION IF EXISTS totalCost;
DELIMITER //
CREATE FUNCTION totalCost (vrepair_ID INT) RETURNS DECIMAL(10,2)
BEGIN
   DECLARE TotalUnitPrice DECIMAL(10,2);

   SELECT SUM(Uses.quantity * Parts.unit_price)
   INTO TotalUnitPrice
   FROM Uses, Parts
   WHERE Uses.repair_ID = vrepair_ID
     AND Uses.part_code = Parts.part_code
     AND Uses.part_manufacturer = Parts.manufacturer_ID;

   RETURN TotalUnitPrice;
END//
DELIMITER ;

-- now test (note that total_cost for job 7 should remain at null
since no parts exists for that job)
UPDATE RepairJobs
SET total_cost = totalCost(repair_ID);
SELECT * FROM RepairJobs;
```

Result:

repair_ID	bike_code	bike_manufacturer	customerCPR	repair_date	duration	total_cost
1	BK-F600	98765	1308981817	2023-12-19	90	470.00
2	BK-B200	22222	1212824762	2024-03-15	60	119.99
3	BK-C300	43126	3108789976	2024-04-10	120	150.00
4	BK-D400	33333	0101014321	2025-04-12	45	180.00
5	BK-E500	12372	1212988825	2025-05-01	180	90.00
6	BK-A100	11111	1203112231	2025-05-01	100	701.50
7	BK-A100	11111	1203112231	2025-05-15	25	NULL

So as shown above, for every repair, the function multiplies each parts unit price by its quantity and sums the results. Jobs with no parts return 0 due `IFNULL`. So basically this shows how derived attributes can be automatically computed instead of stored manually.

7.3 SQL Procedure - `InsertParts`

The procedure `InsertParts()` adds parts to a repair job only if they are compatible with the bike being repaired.

```
-- Look in uses first
SELECT * FROM uses WHERE part_code = 'P-400' OR part_code = 'P-600';
DROP PROCEDURE IF EXISTS InsertParts;
DELIMITER //
CREATE PROCEDURE InsertParts (
   IN vPart_Code VARCHAR(10),
   IN vManufacturer_ID INT,
   IN vQuantity INT,
   IN vRepair_ID INT
)
BEGIN
   DECLARE vExists INT;
   DECLARE vCompatible INT;
   DECLARE vBikeCode VARCHAR(10);
   DECLARE vBikeManu INT;
   -- Find the bike being repaired
   SELECT bike_code, bike_manufacturer
   INTO vBikeCode, vBikeManu
   FROM RepairJobs
   WHERE repair_ID = vRepair_ID;
   -- Check if the part is compatible with that bike
   SELECT COUNT(*)
   INTO vCompatible
   FROM CompatibleParts
   WHERE bike_code = vBikeCode
     AND bike_manufacturer = vBikeManu
     AND part_code = vPart_Code
     AND part_manufacturer = vManufacturer_ID;

   -- If incompatible: stop the procedure with error of course
```

```
    IF vCompatible = 0 THEN
        SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Part is NOT compatible with this bike.';
    END IF;

    -- Check if the part already exists in Uses
    SELECT COUNT(*)
    INTO vExists
    FROM Uses
    WHERE part_code = vPart_Code
      AND part_manufacturer = vManufacturer_ID
      AND repair_ID = vRepair_ID;

    -- Update existing part qty
    IF vExists > 0 THEN
        UPDATE Uses
        SET quantity = quantity + vQuantity
        WHERE part_code = vPart_Code
          AND part_manufacturer = vManufacturer_ID
          AND repair_ID = vRepair_ID;

    -- Insert new since there wasnt one already
    ELSE
        INSERT INTO Uses (part_code, part_manufacturer, repair_ID, quantity)
        VALUES (vPart_Code, vManufacturer_ID, vRepair_ID, vQuantity);
    END IF;
END //
DELIMITER ;
--  Test the procedure
-- The first one adds 3 P-400 parts that come from manufacturer[55555] into the
repair-job with id [5].
-- This information stored in the uses table
CALL InsertParts('P-400', 55555, 3, 5); -- No parts already in uses for that
repair job, so it inserts
CALL InsertParts('P-600', 98765, 5, 5); -- Those parts already exist for that
repair job, so it updates

-- Look in uses again
SELECT * FROM uses WHERE part_code = 'P-400' OR part_code = 'P-600';
```

So this procedure recovers the bike from the repair, checks the `CompatibleParts` table, and throws an error if the part cannot be used.

So if compatible, the part is inserted into `Uses` or its quantity is increased.

The use of `SIGNAL SQLSTATE '4500'` provides feedback to users in case of some invalid operations.

7.4 SQL Trigger - `tooMuch`

In this part, the trigger `tooMuch` makes sure that no unrealistic repair jobs are inserted into the system.

```
DROP TRIGGER IF EXISTS tooMuch;
DELIMITER //
CREATE TRIGGER tooMuch
BEFORE INSERT ON repairJobs
FOR EACH ROW
BEGIN
   IF NEW.total_cost > 100000 OR NEW.duration > 4320 THEN
       SIGNAL SQLSTATE '45000' -- Error 45000 means generic user error
           SET MYSQL_ERRNO = 1525,
               MESSAGE_TEXT = 'Too expensive or too long';
   END IF;
END //
DELIMITER ;
-- Test if it throws an error here. It should do of course "Too
expensive or too long"
INSERT INTO RepairJobs
(repair_ID, bike_code, bike_manufacturer, customerCPR, repair_date,
duration, total_cost)
VALUES
(100, 'BK-A100', 11111, '1308981817', '2025-06-01', 120, 200000);
```

The trigger executes before insertion into `RepairJobs`. Meaning that if a repair exceeds 100.000 DKK or 72 hours ($4 * 24 * 60\,min = 4320$), so the insertion is rejected. This enforces realistic business limits and prevents user errors/invalid data empty.

Discussion & Conclusion

Throughout the development of this project we designed and implemented a complete relational database for the fictional bike repair shop “Cykelparadis”, our goal was to design a system that not only satisfied the functional requirements from the case description but also adhered to the theoretical principles of database design. To be specific, normalization, referential integrity and of course data consistency.

The process of designing, implementing and testing the database has highlighted both the challenges and advantages of structured database programming.

From the conceptual design part, we learned the importance of correctly interpreting the “problem domain” before diving deep into the technical implementation. As mentioned our early models contained design flaws, such as unnecessary serial numbers for every part and bike, which overcomplicated the structure and made the schema less scalable and without re-design would have resulted in bloat. However through constant iteration and discussion, we refined the model into one that correctly reflects real entities and relationships focusing on model codes, manufacturers, customers and repair jobs, while keeping redundancy to a minimum.

In the logical design, we emphasized clear key definitions and normalization. The decision to use composite primary keys in both Bikes and Parts enforced that the same manufacturer could not reuse the same codes while still allowing identical codes across different manufacturers. We also introduced the `manufacturer_ID` as a surrogate key to handle naming collisions, and the `repair_ID` as a unique identifier to support scenarios where customers could repair the same bike type multiple times on the same date. These choices that we made, resulted in a database schema where each relation could be uniquely identified without depending on mutable or nonunique attributes such as names or dates.

During the implementation phase, we focused on enforcing integrity through constraints rather than through application logic. The large use of foreign keys using the `ON DELETE CASCADE` constraint ensures that deletions propagate logically and safely across related tables. For instance, removing a customer automatically removes their address, whereas deleting a manufacturer removes its associated parts and bikes but not repair history, maintaining business relevant data. So, these design patterns show how a real bike shop would want to preserve essential records of customers while maintaining data clean.

The database instance we inserted was designed as a demonstration dataset and also as a validation tool. We intentionally included edge cases, such as two manufacturers sharing the same name, customers with the same address, and unused parts, all this to verify that our schema behaves as predicted under realistic conditions. But overall this dataset allowed us to test in depth, cardinality and participation constraints across every relation in the schema.

In section 5, the SQL modification examples showcased how the system can handle one of many daily operational tasks in a bike shop. Using `INSERT`, `UPDATE` and `DELETE` statements, we simulated business operations such as adding new parts, updating customer details and

removing unwanted records. By verifying every modification with the `SELECT` statement, we showed that the schema correctly enforces foreign key constraints and maintains referential integrity.

In section 6, the data queries further proved the schema's flexibility and efficiency, through a series of `SELECT` statements, we were able to extract nontrivial insights from the dataset, such as identifying customers with repeated repairs, unused parts, and the most frequently serviced bikes, and all the examples correctly illustrated the required results.

Finally, section 7 introduced procedural SQL programming that automated parts of the system logic. Together these constructs illustrated how procedures can enhance the reliability and maintainability of a relational database system, as well as implement the functionality behind derived attributes.

The function `totalCost` demonstrated how derived attributes can be developed to avoid manual updates. The procedure `InsertParts` added controlled logic that secured only compatible parts that can be used for a repair, strengthening data integrity beyond basic constraints. The trigger `tooMuch` ensured real business rules by preventing like invalid repair records from being inserted, such as unrealistically long or expensive repiars.

Overall, this project has provided us with valuable and deep learning insight into the complete process of a database design, from abstract modeling to a more concrete implementation. We have experienced how each design phase builds upon the previous one, and how theoretical concepts such as entity-relationships, functional dependencies, and integrity took us directly into what we call practical database engineering.

In conclusion the final BikeRepairShop database we designed successfully demonstrates the principles we learned in introductory relational database design and showcases how SQL’s declarative statements and procedural features can be combined to form a consistent, maintainable and realistic database solution for a realistic use case scenario.

Appendix

Github Repository: https://github.com/KaspeDKK/DatabaseProject