Technical University of Denmark

02327 Indledende databaser og database programmering

Database Projekt

Gruppe 14

Peter Tønder Blendstrup
s175210

Mette L.B. Andersen
s172840

Joachim S. Mortensen
s175179

May 6, 2018

Abstract

During this project we're going to set up a database in conjunction with CDIO-Final in the three week period for use in another project that combines all our prior knowledge into a larger system.
This part of the project involves, as said, setting up the database, forming the relations, normalizing the database to reduce redundancy, and making sure the database otherwise follows most if not all of the industry standards.

Contents

1 Indledning **3**

2 Analyse **4**

2.1 Normalisering 4

2.2 Roller 5

2.3 JDBC 6

2.4 Views 6

3 Design **7**

3.1 Dannelse ad databasen 7

3.2 Udvideler 8

3.3 Normalisering af ny tabel 8

4 Konklusion **13**

5 Opgaver **15**

5.1 Opgave 1 15

5.2 Opgave 2 16

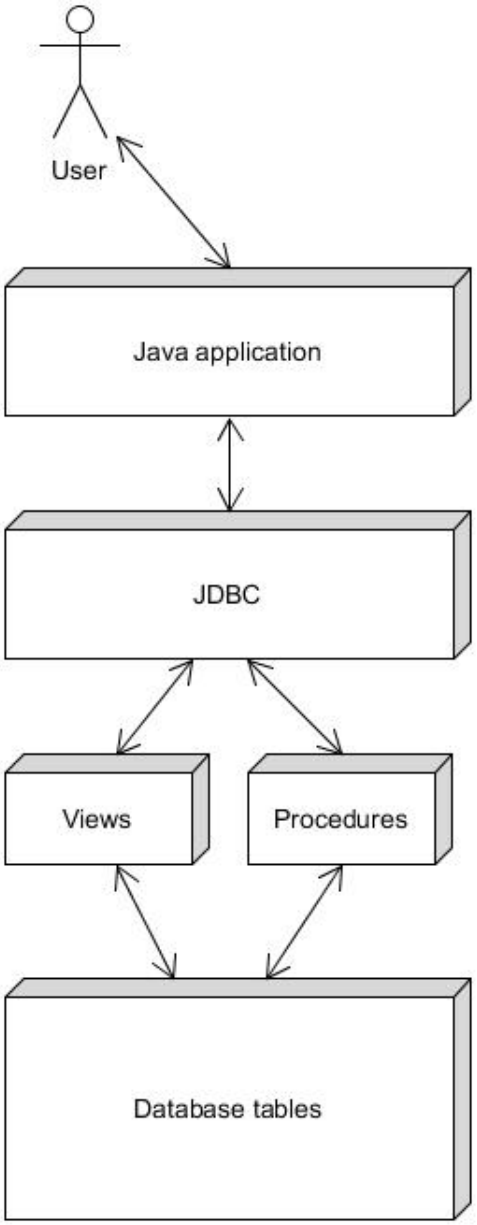
5.3 Opgave 3 16

5.4 Opgave 4 17

5.5 Opgave 5 17

1 Indledning

Denne rapport repræsenterer den proces vi gennemgår ved at producere et database-mangement system, som er en samling af sammenhørende data og et set af programmer for at kunne tilgå de data.
Vi skal udvikle en database som skal bruges i 3 uger projektet. databasen skal være et subelement til et afvejningsssytem.

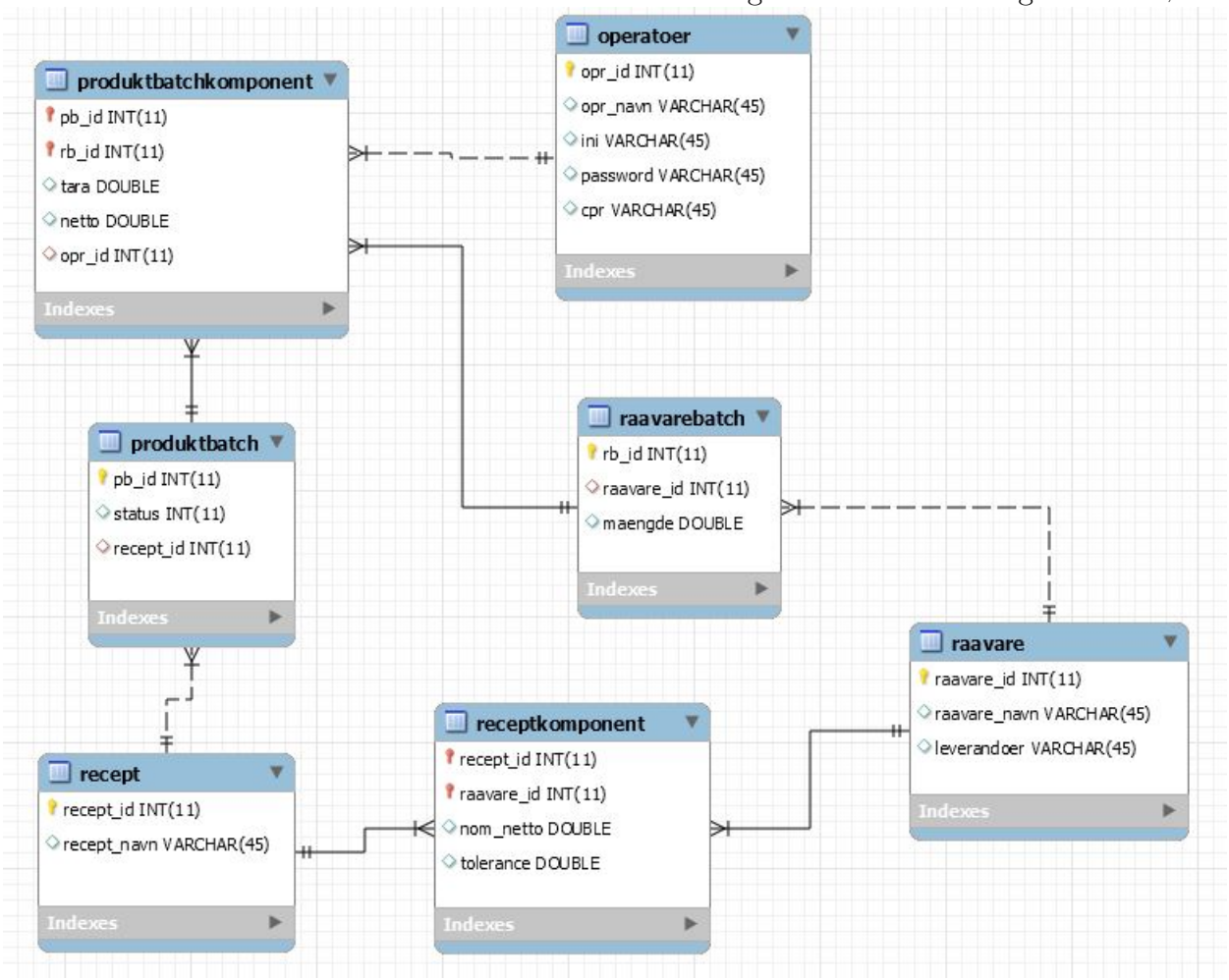


2 Analyse

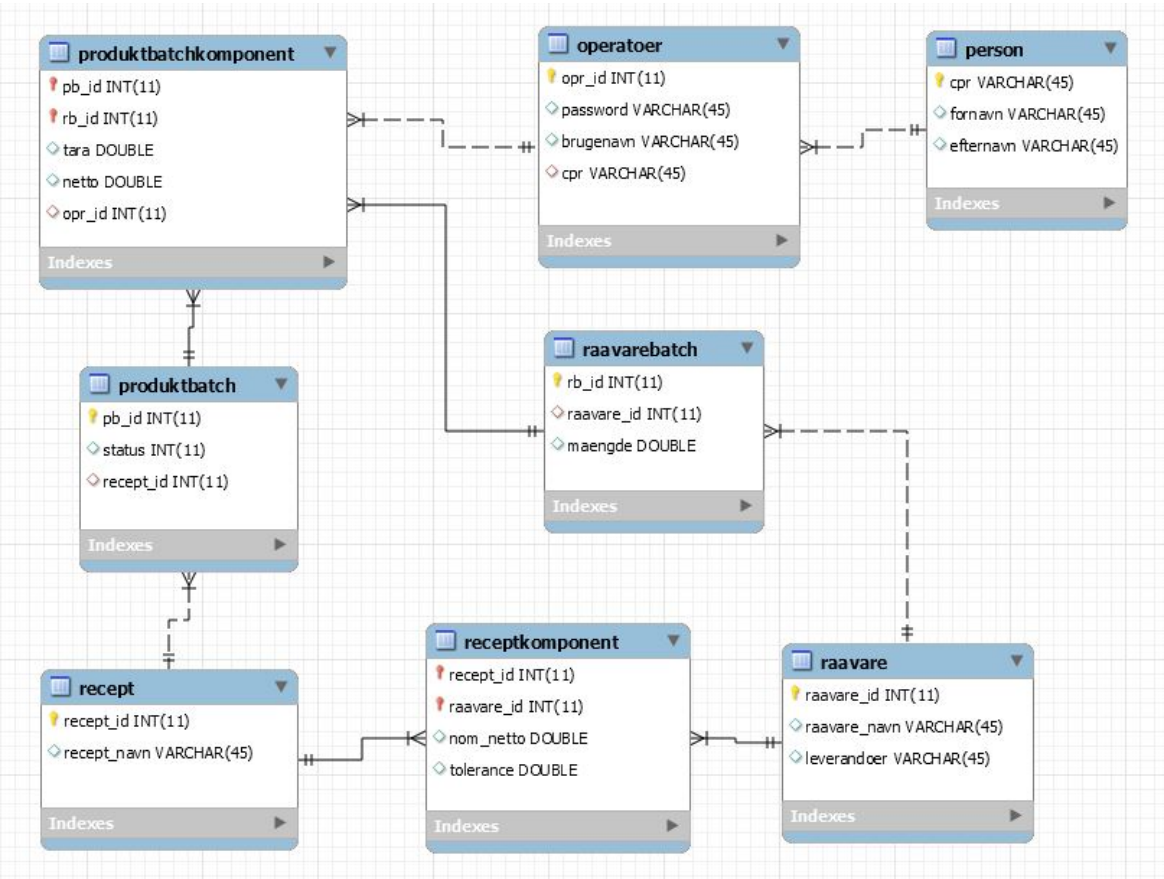
I følgene fase for vi kigget på hvilket krav der er til vores database. Vi starter med at betragte de data i de skemaer der er blevet givet. Og deres relationer.

2.1 Normalisering

For sjov har vi normaliseret databasen. Vi bruger dog ikke den normaliserede database efter dette underkapitel.
1. Normalform Alle elementerne var atomiske til at starte med, undtagen CPR-nummer som vi har tilladt. Hver kolonne har et unikt navn. Hver række har en unik primærnøgle. Rækkefølgen af både kolonner og rækker er ligegyldig.
2. Normalform
3. Normalform Operatoer splittet om CPR til ny tabel Person Initialer fjernet fra Operatoer og erstattet med brugernavn.
(Ikke normalisering?) Leverandør flyttet fra Raavare til Raavarebatch, da det andet ville skabe en masse redundans hvis man fik flere af samme slags råvare fra forskellige leverandører.



Vi kan kigge først på relationen "operatoer". Vi kan se at "opr_navn" ikke opfylder første normal form, da vi kan har flere forskellige operatører med det samme for- og efternavn, dvs. elementerne ikke er atomiske, da vi har lyst til at kunne sortere efter fornavn eller efternavn. En løsning til det er at lave en ny relation "personer" med "fornavn" og "efternavn" som almindelige attributter og "cpr" som en primær nøgle, som vist i grafen nedenfor. Da efter CDIO beskrivelsen "ini" er noget som brugeren selv bestemmer så giver det mere mening at lave navnet "ini" om til "brugernavn" i stedet. Vi beholder "cpr" i "operatoer" som fremmede nøgle fra "personer".



Da relationen "operatoer" og "personer" nu opfylder 3 normal form, kan vi kigge på de andre relationer.

2.2 Roller

Til CDIO Final har vi desuden brug for roller, som vi tilføjer som ny tabel.

Vi har 4 roller:

1. Administrator: Som skal kunne administrer bruger af systemet.
2. farmaceut : Står for råvarer og recepter. Farmaceut har også værkførerens rettigheder.

3. værksfører : Står for at adminstration af produktbacthes og råvarerbatches. Værksfører har også Laborant rettigheder. 4. Laborant / operatør : fortager vejningen.

2.3 JDBC

JDBC står for Java Database Connectivity hvis man kigger på model 1, er det et lag der ligger mellem applikationslaget og view laget.

Dette giver klient/bruger mulighed for at bruge databasen, dvs. gribe fat i vores database i en Java-baseret kontekst, og ud fra det kan vi så trække data ud af vores database og omstille dette til en objekt i et objektorienteret programscenarie. Dette gør det betydeligt lettere at arbejde med data fra databasen.

2.4 Views

Et view er et predefineret query som man så kan lave yderligere queries på. Dette lader dig eksempelvis udelade noget, tage noget fra flere forskellige relationer osv. og putte det i en tabel der så kan bruges. Et eksempel på et view der kunne være relevant, ville være, hvis man ønskede en liste af samtlige brugere, altså deres navne og cpr-numre. Da vi har valgt at dele operatoer og personer i to, ville vi være nødsaget til at sætte disse to sammen igen via et view. Samtidig ville det ikke være fornuftigt at lade password være tilgængelig i denne kontekst, selvom vi måske ikke viser den, ville et simpelt SQL-query til det givne view jo så give et kæmpe dump af alle kodeord. Derfor ville man så udelade password i sin view-creation query.

```
CREATE VIEW all_users AS
SELECT cpr, brugenavn, opr_id
FROM operatoer
INNER JOIN personer ON operatoer.cpr = personer.cpr
```

Herefter kan vi så lave queries som på enhver anden relation, eksempelvis:

```
SELECT * FROM all_users
```

Dette ville vise os en tabel hvori vores brugere og deres relevante info bliver vist, foruden kodeord, så derfor ville et query efter passwords til "all_users" viewet ikke resultere i en lækage af kodeord.

3 Design

Her beskrives hvorledes vi skaber vores database.

3.1 Dannelse ad databasen

Før vi fortsætter kan vi arbejde lidt på at anvende vores nye design på databasen.
Vi begynder med at lave en kopi af databasen og arbejde på det i stedet for at arbejde på originalen.
Vi skaber en ny database.

```
create database lab_database2_copy;
```

Vi laver en kopi fra den gamle til den nye database. Det skal gøres i konsollen.

```
mysqldump -u root -p lab_database2 | mysql -u root -p lab_database2_copy
```

3.2 Udvideler

Her demonteres hvoledes person tabellen bliver oprettet.

Vi skaber "personer" tabelen.

```
create table personer
   (cpr varchar(45),
     fornavn varchar(45),
    efternavn varchar(45),
    primary key (cpr)
   );
```

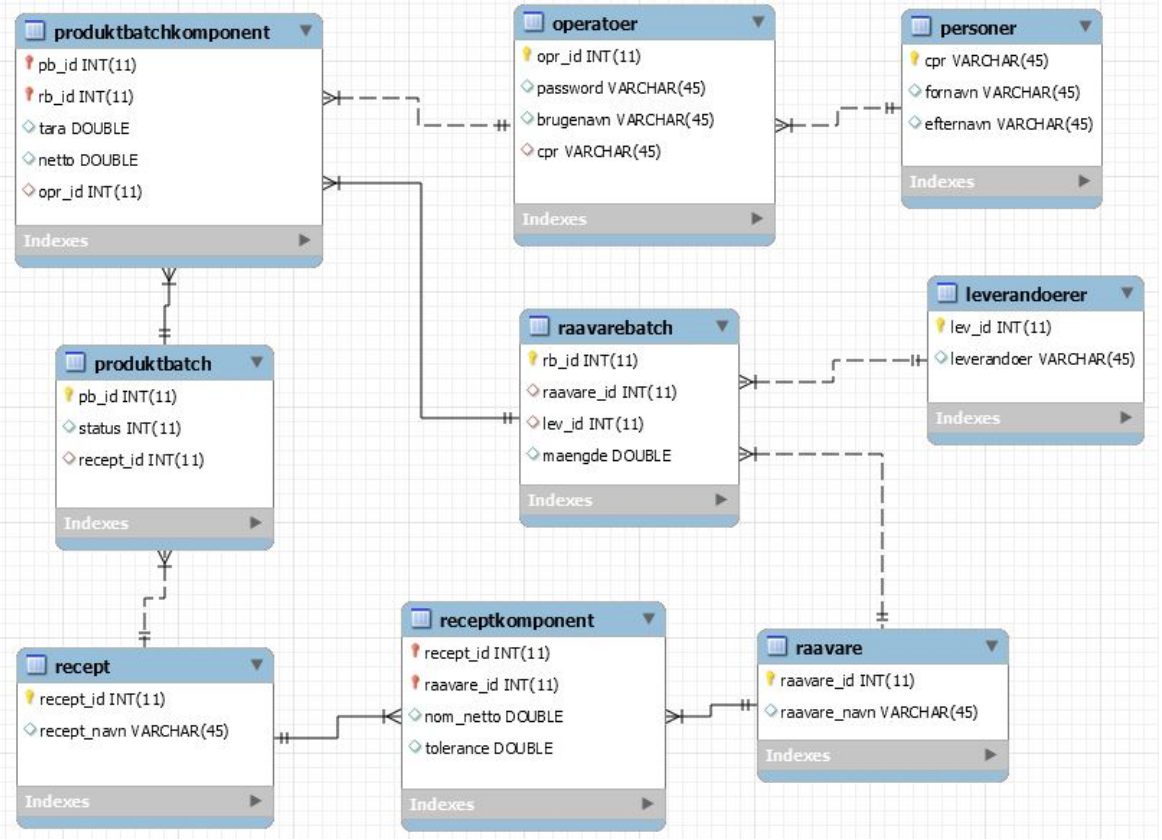
Vi kopierer informationen fra "operatoer" til "personer".

```
insert into personer (cpr, fornavn, efternavn)
select cpr,
      substring_index(`opr_navn`, ' ', 1),
      substring_index(`opr_navn`, ' ', -1)
from operatoer;
ALTER TABLE operatoer ADD CONSTRAINT cpr FOREIGN KEY (cpr) REFERENCES
   personer(cpr);
ALTER TABLE operatoer DROP COLUMN opr_navn, CHANGE ini brugernavn varchar(45);
```

3.3 Normalisering af ny tabel

Vi kigger nu på næste relation "produktbatchkomponent" og kan se at den også er i 3 normal form uden at vi behøver lave om på noget i relationen.

Hvis vi kigger på relationen "raavare" kan vi se at hvis vi har flere forskellige "leverandoer" for det samme "raavare_id" så opfylder vi ikke 1 normal form, grunden af at i dette situation vil der være flere rækker med det samme primær nøgle. Løsning på det problem er at flytte "leverandoer" til en ny relation "leverandoerer"(fleretal) med en primær nøgle "lev_id" og "leverandoer" og så vil "raavarebatch" bruge "lev_id" som fremmede nøgle.



Vi skaber den nye relation "leverandoerer".

```
create table leverandoerer
   (lev_id int AUTO_INCREMENT,
    leverandoer varchar(45),
    primary key (lev_id)
   );
```

Vi kopierer "leverandoer" fra "raavare" til "leverandoerer".

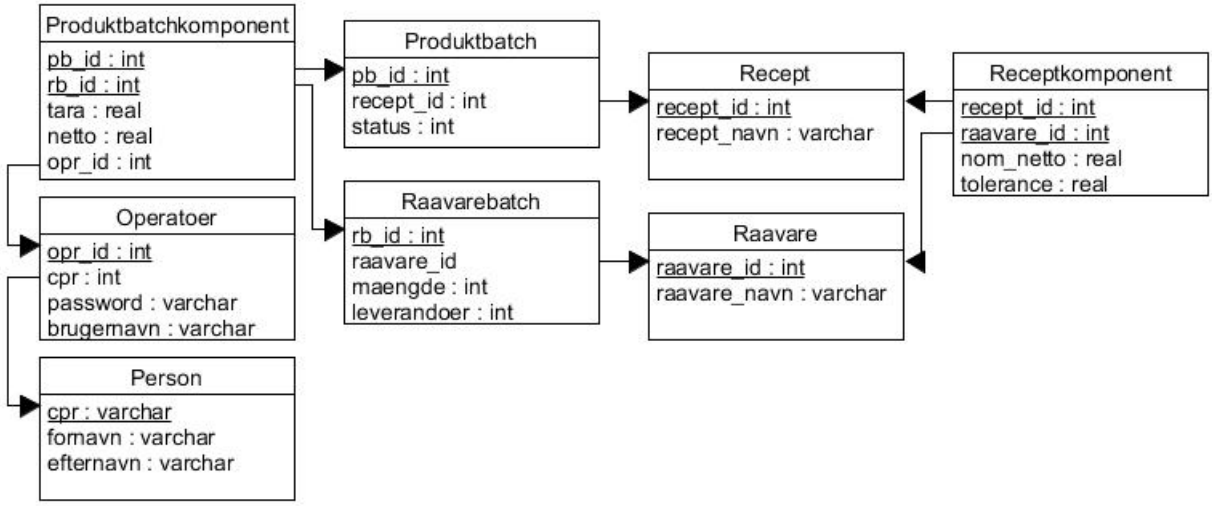
```
insert into leverandoerer (leverandoer)
select distinct leverandoer
from raavare;
```

Vi lave en ny kolonne "leverandoer" i "raavarebatch" som fremmednøgle fra "leverandoerer".

```
ALTER TABLE raavarebatch
ADD COLUMN lev_id INT,
ADD FOREIGN KEY (lev_id) REFERENCES leverandoerer(lev_id);
```

Vi kopierer "lev_id" til "raavarebatch" på baggrund af de råvarer de består af.

```
UPDATE raavarebatch, raavare, leverandoerer
SET raavarebatch.lev_id = leverandoerer.lev_id
WHERE (raavarebatch.raavare_id = raavare.raavare_id)
      AND (leverandoerer.leverandoer = raavare.leverandoer);
```



Før vi fortsætter kan vi arbejde lidt på at anvende vores nye design på databasen.
Vi begynder med at lave en kopi af databasen og arbejde på det i stedet for at arbejde på originalen.
Vi skaber en ny database.

```
create database lab_database2_copy;
```

Vi laver en kopi fra den gamle til den nye database. Det skal gøres i konsollen.

```
mysqldump -u root -p lab_database2 | mysql -u root -p lab_database2_copy
```

Vi skaber "personer" tabelen.

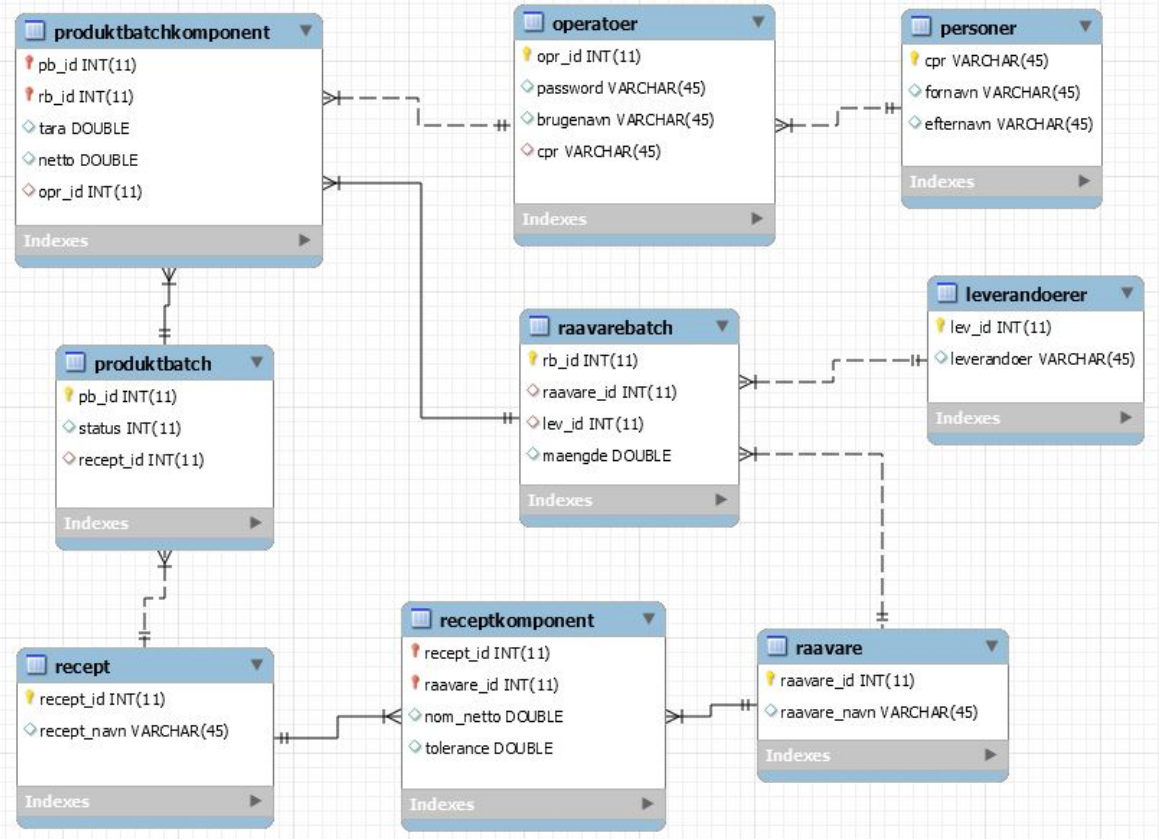
```
create table personer
   (cpr varchar(45),
     fornavn varchar(45),
    efternavn varchar(45),
    primary key (cpr)
   );
```

Vi kopierer informationen fra "operatoer" til "personer".

```
insert into personer (cpr, fornavn, efternavn)
select cpr,
      substring_index(`opr_navn`, ' ', 1),
      substring_index(`opr_navn`, ' ', -1)
from operatoer;
ALTER TABLE operatoer ADD CONSTRAINT cpr FOREIGN KEY (cpr) REFERENCES
    personer(cpr);
ALTER TABLE operatoer DROP COLUMN opr_navn, CHANGE ini brugernavn varchar(45);
```

Vi kigger nu på næste relation "produktbatchkomponent" og kan se at den også er i 3 normal form uden at vi behøver lave om på noget i relationen.

Hvis vi kigger på relationen "raavare" kan vi se at hvis vi har flere forskellige "leverandoer" for det samme "raavare_id" så opfylder vi ikke 1 normal form, grunden af at i dette situation vil der være flere rækker med det samme primær nøgle. Løsning på det problem er at flytte "leverandoer" til en ny relation "leverandoerer"(fleretal) med en primær nøgle "lev_id" og "leverandoer" og så vil "raavarebatch" bruge "lev_id" som fremmede nøgle.



Vi skaber den nye relation "leverandoerer".

```
create table leverandoerer
   (lev_id int AUTO_INCREMENT,
    leverandoer varchar(45),
    primary key (lev_id)
   );
```

Vi kopierer "leverandoer" fra "raavare" til "leverandoerer".

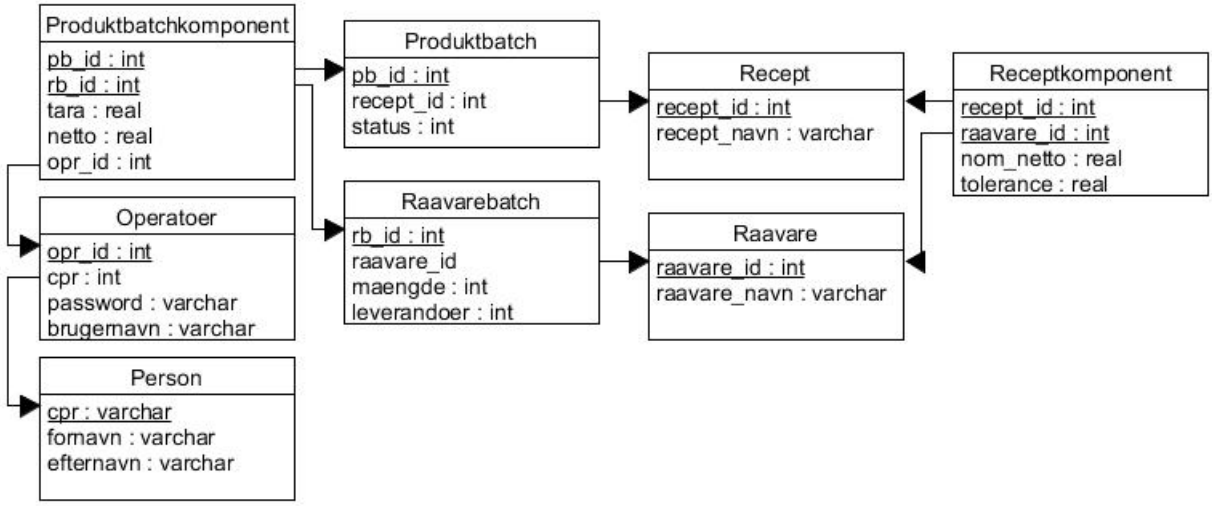
```
insert into leverandoerer (leverandoer)
select distinct leverandoer
from raavare;
```

Vi lave en ny kolonne "leverandoer" i "raavarebatch" som fremmednøgle fra "leverandoerer".

```
ALTER TABLE raavarebatch
ADD COLUMN lev_id INT,
ADD FOREIGN KEY (lev_id) REFERENCES leverandoerer(lev_id);
```

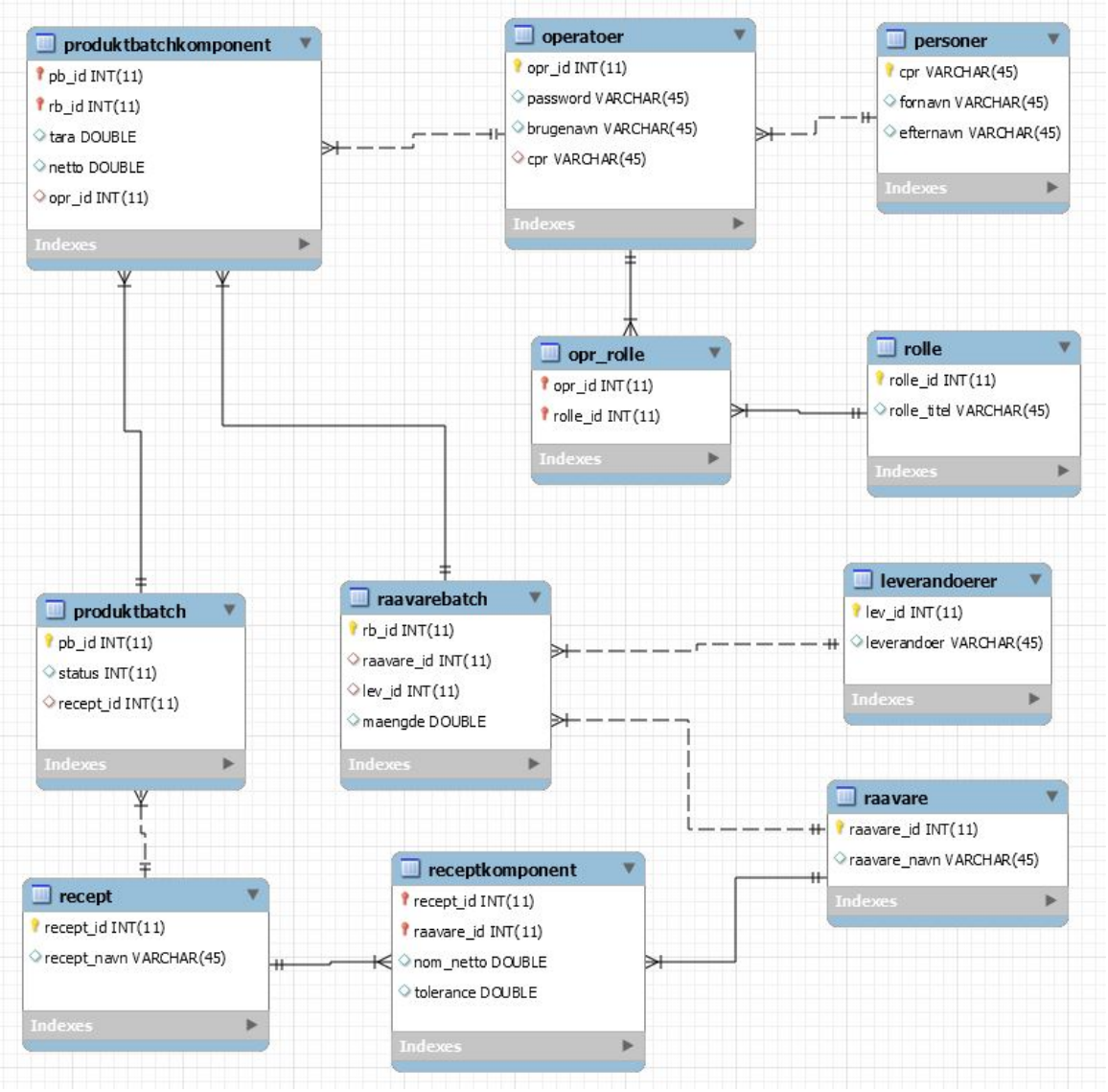
Vi kopierer "lev_id" til "raavarebatch" på baggrund af de råvarer de består af.

```
UPDATE raavarebatch, raavare, leverandoerer
SET raavarebatch.lev_id = leverandoerer.lev_id
WHERE (raavarebatch.raavare_id = raavare.raavare_id)
      AND (leverandoerer.leverandoer = raavare.leverandoer);
```



4 Konklusion

Her ses relationskema for vores database.



Final diagram

5 Opgaver

5.1 Opgave 1

Til at starte med kan man se på relationen "raavarebatch".

Status | Result1

```
select *
from raavarebatch

Elapsed Time:  0 hr, 0 min, 0 sec, 7 ms.
```

Status | Result1

	rb_id	raavare_id	maengde
1	1	1	1000.0
2	2	2	300.0
3	3	3	300.0
4	4	5	100.0
5	5	5	100.0
6	6	6	100.0
7	7	7	100.0

Man kan se fra billederne at der kun er råvarer med "raavare_id" lige med 5 der indgår i mere end en "raavarebatch" tuples. Da "rb_id" attributtet i "raavarebatch" er unik for hver "raavarebatch" tuple, og da for hver "rb_id" kun kan være en type råvare, dvs. kun en "raavare_id" per "raavarebatch". Så kan man sige at hvis man taller "raavare_id" i "raavarebatch" relationen og lave en ny relation kun med de "raavare_id"-er der findes mere end en gang så kan man bruge den ny relation til at finde navne og tilsvarende id-er fra relationen "raavare".

Det kan man se fra det følgende query:

Status | Result1

```
select raavare_id, raavare_navn
from raavare
where raavare_id = ( select raavare_id
                     from raavarebatch
                     group by raavare_id
                     having count(raavare_id) > 1)

Elapsed Time:  0 hr, 0 min, 0 sec, 23 ms.
```

Status | Result1

	raavare_id	raavare_navn
1	5	ost

Status | Result1

```
select *
from raavare

Elapsed Time:  0 hr, 0 min, 0 sec, 12 ms.
```

Status | Result1

	raavare_id	raavare_navn
1	1	dej
2	2	tomat
3	3	tomat
4	4	tomat
5	5	ost
6	6	skinke
7	7	champignon

Vi kan se at "tomat" har "raavare_id" 2, 3 og 4 dvs. det samme "raavare_navn" har 3 forskelige "raavare_id"-er som er et problem for os, da vi kun får "ost" som en råvare der indgår i mindst to forskellige råvarebatches når der i virkligheden skal være to, både "ost" og "tomat".

5.2 Opgave 2

Vi laver natural join mellem receptkomponent, recept og raavare, da recept-tabellen indeholder receptens ID og navn, og råvarens navn ligger i raavare-tabellen.

```
select recept_id, recept_navn, raavare_navn
from receptkomponent
    natural join recept
    natural join raavare;
```

5.3 Opgave 3

Tabellen receptkomponent fortæller hvilke opskrifter indeholder hvilke ingredienser, via deres ID'er. For at omregne ID til og fra navn har vi brug for både recept og raavare tabellerne. Vi joiner alle tre tabeller om deres fællespunkter (raavare_id og recept_id), isolerer dem som indeholder enten skinke eller champignon, og skriver kun unikke recepters navne ud.

```
select distinct recept_navn
from receptkomponent
    natural join raavare
    natural join recept
where raavare_navn = "skinke"
    or raavare_navn = "champignon";
```

For at finde de recepter som indeholder både skinke og champignon, joiner vi de tre tabeller som før og isolerer de receptkomponenter for enten skinke eller champignon. Derefter grupperer vi dem omkring hvilken recept de tilhører, og fjerner de grupper der ikke indeholder begge to.

```
select recept_navn
from receptkomponent
    natural join raavare
    natural join recept
where raavare_navn = "skinke"
    or raavare_navn = "champignon"
group by recept_id
having count(*) >= 2;
```

5.4 Opgave 4

Vi finder først de recepter som indeholder ingrediensen champignon. Derefter finder vi alle recepter undtagen de ovennævnte.

```
select recept_navn
from recept
where recept_id not in ( select distinct recept_id
    from recept natural join receptkomponent natural join raavare
    where raavare_navn = "champignon" );
```

5.5 Opgave 5

Vi finder største nominelle vægt for tomat, og finder derefter recepter hvis receptkomponent for tomat har nominel vægt lig med største nominelle vægt.

```
select recept_navn
from recept natural join receptkomponent natural join raavare
where raavare_navn = "tomat"
    and nom_netto = ( select max(nom_netto)
        from receptkomponent natural join raavare
        where raavare_navn = "tomat" );
```