Database Projekt

'Implementering af et databaseinterface i Java'

Indledende databaser og database programmering 02327, Softwareteknologi

Gruppe Nr.: 13

Afleveringsfrist: 07/05-2017 23:59

Institut: DTU Compute

Vejleder: Ronnie Dalsgaard

Denne rapport er afleveret via Campusnet (der skrives ikke under)

Denne rapport indeholder 31 sider eks. denne sider.

Jesper Bang, s144211

Bertram Christian Henning, s153538

Jia Hao Johnny Chen, s165543

Jonathan Yngve Friis, s165213

Christopher David Carlson Chytræus, s165230

Thomas Kristian Lorentzen, s154424

Indholdsfortegnelse

Abstract

The project described in this report is a pre-made database for agricultural groceries. Our report is centered on certain theoretical terms about database design, and how we have been applying it to our specific database. The report also has a strong focus on how we have been using SQL to manipulate the database. In our appendix, we have shown several examples on how to use SQL statements to find specific data from tables, joining certain tables to get a broader view of our data, as well as using arithmetic expressions on our data. Our appendix also contains several examples of our created views and stored procedures. Another focus of our report is how we have been using a JDBC (Java Database Connector) to connect our SQL database to a Java program. The theoretical aspects we have been covering in the report are normalization and EER- and ER-Diagrams.

Overall, we can conclude that our report has successfully covered the important aspects we have been learning during our database course, and our SQL expressions as well as our JDBC connection have been successful.

Introduktion

Formålet med dette projekt er at designe et databasesystem hvor vi implementerer et database interface i Java. Projektet består af to opgaver. Den ene opgave handler om at lave SQL forespørgsler som skal have forklarende tekst. Den anden opgave handler om at implementere database interfaces samt lave test klasser, der tester implementationerne. I vores rapport har vi fokus på, hvordan vi har opbygget databasen, hvordan den er forbundet til java, samt modellering med forskellige diagrammer og teoretiske begreber.

Analyse og design

Normalisering

Ved normalisering forstås der, at man forsøger at fjerne så meget overflødigt data i databasens tabeller som muligt. Inden for normalisering er der 5 normalformer. I dette projekt har vi kun arbejdet os frem til at få vores database på 3. normal form. Grunden til, at vi i databaser benytter os af normalisering er, at vi gennem det får nemmere ved at opdatere vores databaser. Det vil sige, at hvis vi indsætter eller fjerner noget data fra en normaliseret database, ville det tage meget mindre tid end hvis det var en ikke normaliseret database.

Nedenunder er en beskrivelse og forklaring på, hvordan vi har fået vores data på de tre første normalformer.

1. Normalform:

Første normalform er opnået, når følgende kriterier er opfyldt:

- Enhver celle i en tabel indeholder en enkelt værdi.
- Alle data i en kolonne er fra samme domæne.
- Enhver række i et skema har en unik værdi som primærnøgle.

På 1. Normalform er det lige meget, hvilken rækkefølge vores rækker/kolonner er placeret i skemaet. I vores database som vi fik udleveret, har det hele allerede været på første normalform. Et eksempel på dette kan være raavare tabellen. I denne tabel er der ingen af rækkerne/kolonnerne, hvor værdierne ikke er atomare, da der kun er én værdi i hvert felt. Samtidigt kan vi se, at alle data i de forskellige kolonner er af samme type data. Alle ID'er i ID kolonnen er med tal, mens alle raavare_navn data er navne på råvarerne. Samtidigt er al data i leverandoer-kolonnen navne på leverandører.

Udover det kan vi ud fra raavare_id-kolonnen, som er primær nøglen i dette skema, også se, at enhver raavare har et unikt ID. Med disse observationer kan det konkluderes, at databasen er på 1NF. Udover dette har vi også valgt at dele opr_navn i operatoer op i opr_fornavn og opr_efternavn, så tabellen følger 1NF.

2. Normalform:

For at et data skema kan være på 2. normalform, skal følgende betingelser være opfyldt:

- 1. Normalform skal være opfyldt.
- Enhver attribut som ikke er primær-nøgle SKAL afhænge af primær-nøgle(rne). Hvis der er 2 eller flere kolonner som er primær nøgle, skal alle andre attributter afhænge af alle for, at skemaet er på 2. normalform.

I vores udleverede database har vores data skemaer også været på 2. normalform fra starten af. Et eksempel på dette kan være relationsskemaet produktbatchkomponent. Her har vi 2 primary keys, som er pb_id (produktbatch-ID) og rb_id (raavarebatch-ID). Disse er samtidigt foreign keys, da de også er primary keys i hhv. produktbatch- og raavarebatch-skemaet. Vi har de tre andre attributter: opr_id, tara og netto.

Netto og tara er to forskellige summer af vægt, og vægten hænger direkte sammen med pb_id og rb_id. Samme gælder også for opr_id attributten. En ting vi valgte ikke at røre ved er opr_id og cpr i operatoer tabellen. Begge disse kolonner kan entydigt bestemme en operatoer og det er derfor

ikke på 2NF. Vi har dog valgt at lade det forblive på denne måde da vi følte at en operatør både skulle have et ID og et cpr-nummer.

3. Normalform

For at ens database kan være på 3. normalform, skal følgende betingelser være opfyldt:

- Databasen skal være på 2. normalform (2NF)
- Attributterne er kun afhængige af primærnøglen, og ikke gennem nogle andre attributter eller kombinationer af attributter. Det vil sige, at ingen kolonner må afhænge transitivt af primærnøglen i 3NF

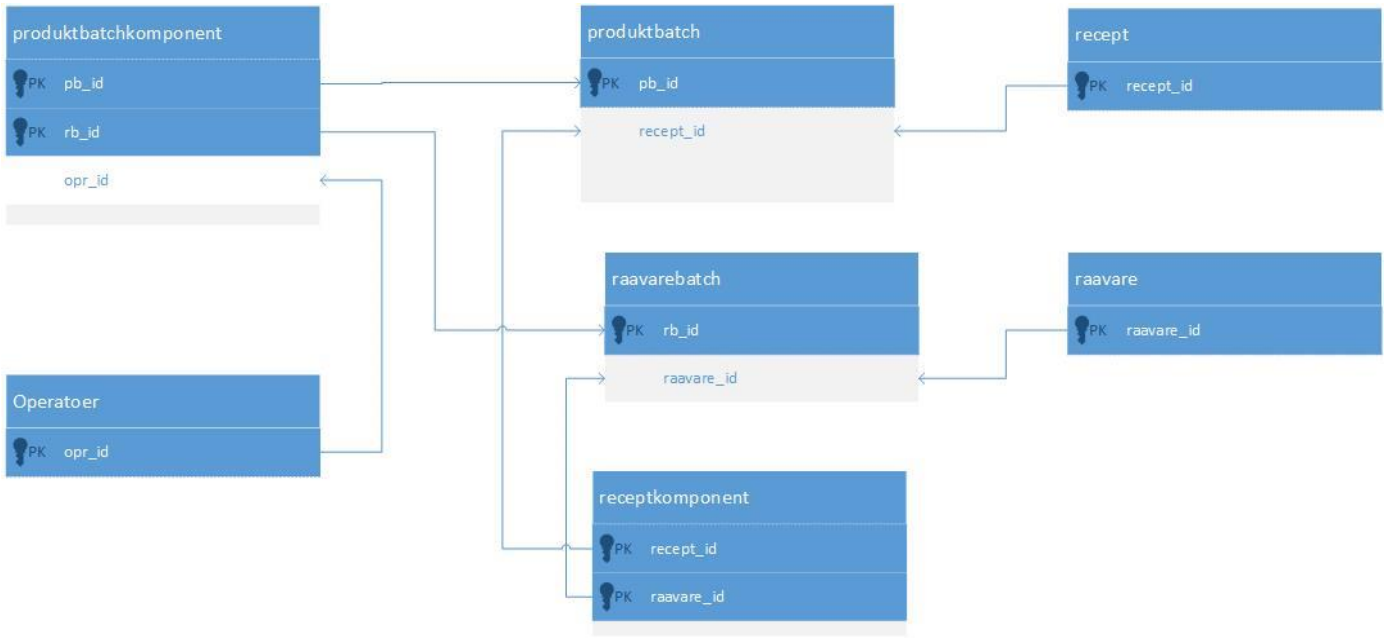
.

I vores database er stort set alle vores skemaer på 3NF på forhånd. Et eksempel på dette kan være skemaet for receptkomponent. I dette skema har vi 2 primary keys, som er recept_id og raavare_id.

I dette skema er der derudover attributterne nom_netto og tolerance. I dette skema kan tolerance og nem_netto bestemmes ud fra recept- og raavare_id. Dog kan tolerance ikke bestemmes fra de to primary keys indirekte gennem nom_netto, da de to attributter ikke kan forbindes transitivt til hverken recept_id eller raavare_id.

Schema Diagram

Schema Diagramet over databasen viser primarykey og foreignkey's afhængighed af hinanden. Dette kan afbildes med et schema diagram, som forneden, hvor vi har vores database. Hvert forhold vises som en boks med relationens navn som overskrift og dens attributter inde i boksen nedenunder.



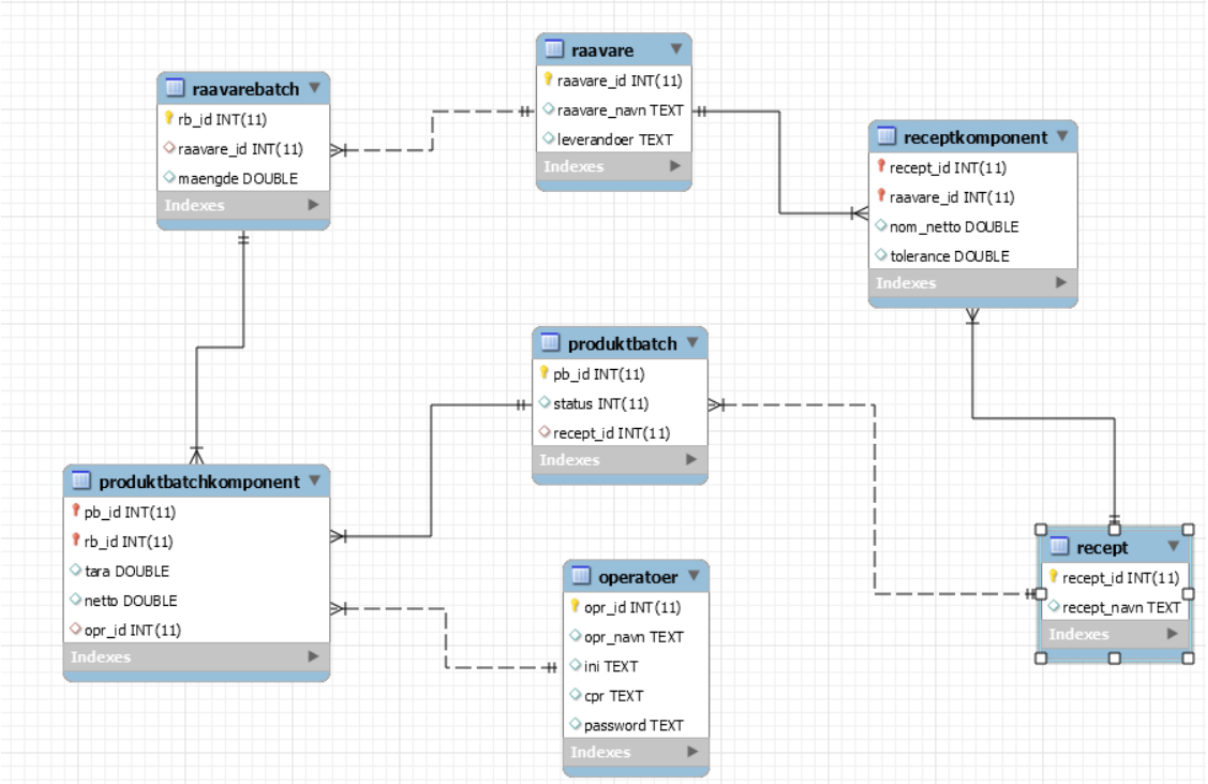
EER Diagram

Et EER (extended entity relations) diagram viser, hvordan de forskellige skemaer i vores database hænger sammen. Når en attribut har en gul nøgle betyder det, at det er en primary key. Hvis nøglen i stedet er rød, vil det sige, at det er en foreign key, som samtidigt også er en primary key. Når en attribut har en ikke-fyldt rhombe, vil det sige, at det er en attribut, som godt kan være NULL. Hvis rhomben uden for attributten derimod er fyldt, vil det sige, at attributten ikke kan have en NULL værdi.

Linjerne mellem de forskellige skemaer indikerer, hvorvidt deres primary keys kun er foreign keys, eller om de både er foreign og primary keys, i det andet skema. Nedenfor ses EER-diagram for vores database. Pilene viser, at der er one-to-many relationer mellem de forskellige skemaer. F.eks. kan vi se ud fra linjerne, at der kan indgå op til flere raavarebatches i en enkelt produktbatchkomponent. Vi kan ud fra samme linje også se, at rb_id, som er primary key i

raavarebatch, også er primary key, samtidigt også foreign key, i produktbatchkomponent skemaet. Dette er indikeret ved, at der er en solid linje som forbinder de to skemaer, samt at nøglen for pb_id er rød.

Derimod kan vi så også se, at raavare_id, som er primary key, ikke er primary key, men kun foreign key, i raavarebatch-skemaet. Ud fra vores raavarebatch skema kan vi samtidigt se, at vores maengde attribut godt kan være null, da det er en ikke-fyldt diamant/rhombe ved siden af.



ER-diagram

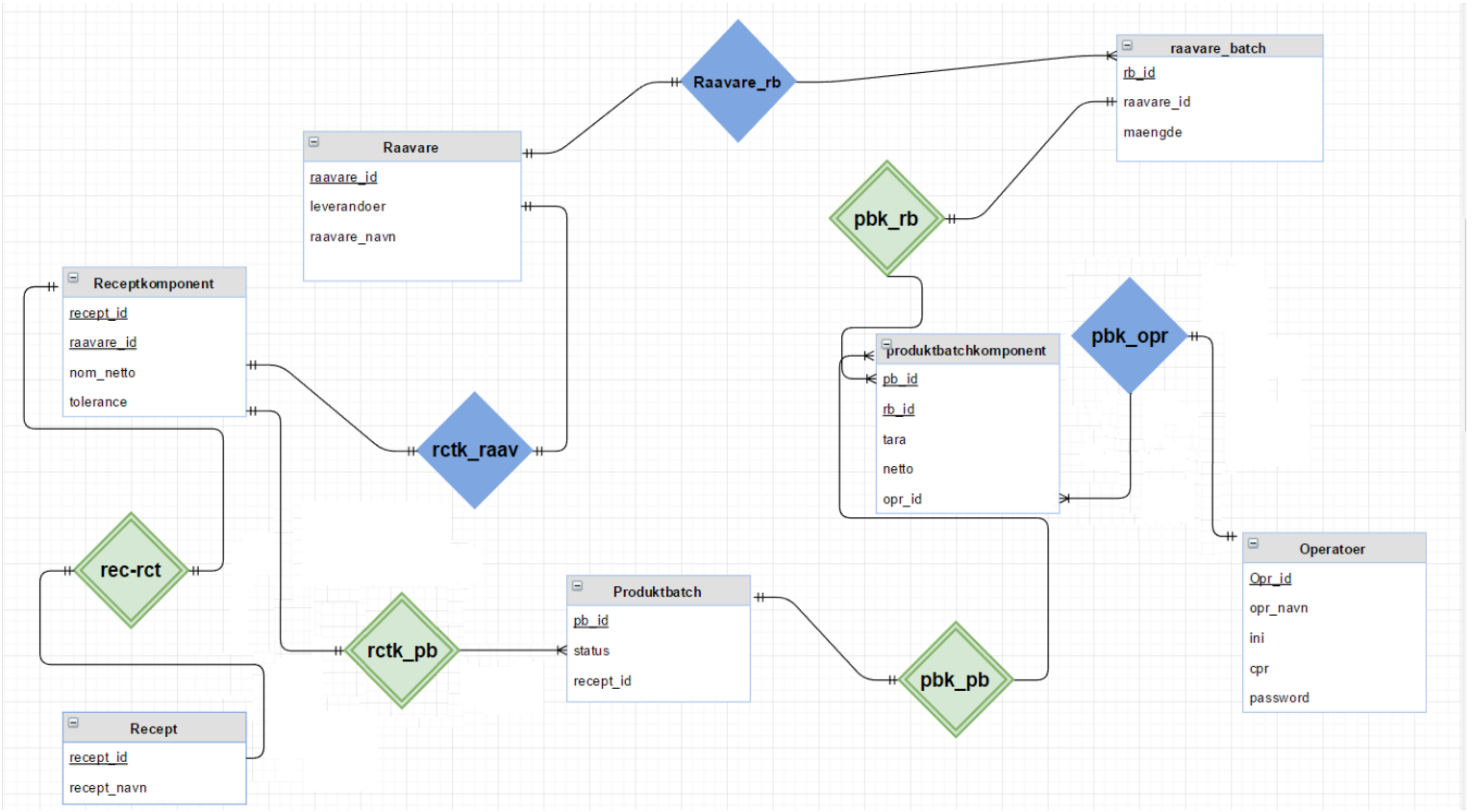
ER-diagrammer, som er en forkortelse for Entity Relations diagram, modellerer en database efter, hvordan de forskellige attributter hænger sammen gennem relationer. Til forskel fra EER modellen, hvor der mere er fokus på keys og hvilken mængde relation der er mellem de forskellige skemaer, har man i ER diagrammet mere fokus på, hvordan de forskellige skemaer hænger sammen med hinanden. I ER-diagrammer er entities et objekt eller begreb, som findes i den virkelige verden, mens at relationship er associeringen mellem de forskellige entities i databasen. Nedenunder ses ER diagram for vores database.

Relationer mellem entities i ER diagrammer markeres med en diamant, hvori der står navnet på relationen. Relationens navn vælger man selv. I dette diagram har vi kaldt relationerne som forkortelser for de forskellige entities som er bundet. Nogle af vores entities er markeret med en dobbelt diamant. Dette betyder, at der er tale om en svag relation. En svag relation, eller weak entity relation, karakteriseres ved, at primærnøgle(rne) i det ene skema er foreign keys fra et andet skema. Et eksempel på dette kan være i produktbatchkomponent, hvor begge primary keys er foreign keys fra andre skemaer. Derfor er relationen mellem den og både raavare_batch og raavare en svag entitets relation.

Hvis man derimod ser på f.eks. relationen mellem raavare_batch og raavare skemaet, kan det ses, at der er tale om en stærk entity relation. Dette skyldes, at modsat førnævnte produktbatckkomponent skemaet, hvor det ikke havde sine egne primary keys, så har raavare og raavarebatch sine egne primary keys.

Ved at se på linjerne som forbinder raavare og raabare_batch i deres relation, kan man se, at der er en-til-mange relation mellem de to skemaer. Det betyder, at en råvare godt kan indgå i flere batches af råvarer.

To andre relationer, som også bruges i ER diagrammer er en-til-en og mange-til-mange. En-til-mange relationen kan selvfølgelig også være omvendt.



Database

Vi har sat vores database op på et webhotel - freemysqlhosting.net. Hvor vi benytter os af phpmyadmin som administrationsmodul til at redigere og scripte serveren. Derudover har vi oprettet vores udgivne database, som har været lavet på forhånd, samt vores views, stored procedures og functions. Herfra har vi kunne forbinde vores JDBC lag sammen med databasen. SQL spørgsmålene er også afprøvet succesfuldt på databaseserveren, hvilket tyder på at vi har en god forbindelse til serveren, samt vi kan lave operationer på databasen. Dette har gjort det meget nemmere, at arbejde samtidig på samme database, uden nødvendigvis være på samme netværk og viser et mere realistisk brug af databasen.

Views

Når vi snakker om databaser er et view et resultset af gemte query's. Brugeren af databasen kan se 'udføre'/se viewet helt ligesom med en tabel i databasen. Man kan altså se et view som en virtuel tabel i databasen. Da viewsne er virtuelle indgår de ikke i nogle skemaer over den grundlæggende struktur af databasen, men hvis man skriver kommandoen "SHOW TABLES;" vil viewsne blive vist sammen med de reelle tabeller i databasen. Den grundlæggende form for et view ser således ud:

```
CREATE VIEW 'view_navn' as
SELECT 'række1', 'række2'...
FROM 'table1', ...
WHERE 'betingelse'
```

Grunden til at have views er at det sparer tid, så man ikke skal skrive en hel select-statement hver gang man vil se en eller flere tabeller, eller noget helt andet. For eksempel kunne man have en meget lang statement, som man gemmer som et view, og næste gang man skal bruge denne statement bruger man viewet i stedet for.

I vores opgave har vi brugt en del views og her er nogle af dem:

OperatoerControl-viewet er et view, som tager alt fra operatoer joined med produktbatch joined med produktbatchkomponent. Grunden til at vi har valgt at lave dette view er at en person med rollen produktionsleder, kan se hvilke operatøer har lavet hvilke produktbatches og hvilke komponenter disse batches består af. Vi har valgt at bruge natural join, da denne join samler

tabeller til en tabel, hvor kolonner med samme navn kun opstår en gang. Operatør har vi valgt at hente da vi skal bruge info om operatøren f. eks hans/hendes ID, navn og/eller cpr-nummer. Produktbatch har vi valgt da denne tabel har info om produktbatchet, nemlig ID, status og hvilke recepter. Vi har tilsidst valgt produktbatchkomponent, da vi via den kan se hvilke recepter/raavaer der i batchsne og hvilken operatør der har lavet disse batches.

```
CREATE VIEW OperatoerControl AS
SELECT *
FROM operatoer natural join produktbatch natural join produktbatchkomponent;
```

I java koden er det så meningen at produktionslederen f. eks kan indtaste ID for en operatør og dermed se hvad denne operatør har lavet.

Udover dette view har vi også:

```
CREATE VIEW getOperatoer AS
SELECT *
FROM operatoer;
```

getOperatoer viewet er et view der viser en liste over alle operatørene. Da man lige så godt kunne have lavet en SQL statement der ser sådan ud “SELECT * FROM operatoer;”, så virker dette view lidt ligegyldigt, men vi har valgt at have det med, fordi det øger sikkerheden da man jo ser et view i stedet for den reelle tabel operatør. I java koden kan en admin f. eks indtaste en operatør ID og dermed få al info omkring denne operatør. Siden at det kun er en admin der må se på en operatørs CPR-nummer og password, har vi også lavet et view ligesom dette, men til de lavere rangerede brugere:

```
CREATE VIEW getOperatoer2 AS
SELECT opr_id, opr_fornavn, opr_efternavn, ini
FROM operatoer;
```

Forskellen på getOperatoer2 og getOperatoer er at, når dette view bruges bliver CPR-nummeret og password ikke vist. Dette view er perfekt til en der gerne må se på operatørerne men ikke på personlige data.

Udover disse views har vi også flere views som minder meget om getOperatoer og derfor gennemgår vi dem ikke her, men de kan ses i bilaget.

Functions/stored procedures

Functions:

Functions i SQL er ligesom funktioner i matematik. Man giver funktionen et input og så returnerer den et output. I SQL modtager functions typisk atomariske værdi og returnerer også typisk atomariske værdier. Med andre ord kan man kun indtaste en værdi fra en række ind i en funktion og den vil så også kun returnerer en enkelt værdi som svar. Typisk bruges functions til små søge funktioner, som f. eks ved indtastning af en operatør ID så gives navnet på operatøren og det også præcis sådan vi har brugt dem i vores database.

Formularen for en SQL function ser således ud:

```
delimiter //
CREATE FUNCTION "navn"(inputnavn inputtype)
RETURNS returntype
begin
declare svar svartype;
/*
SQL statement/funktionens funktion
*/
return svar;
end; //
delimiter ;
```

Når vi skriver delimiter // ændrer vi execute tegnet fra ; til // og til sidst ændres det tilbage.

I vores opgave har vi lavet følgende functions:

```
delimiter //
create function operatoer_cpr(opr_id int(11)) returns text
begin
declare opr_cpr text;
select cpr into opr_cpr from operatoer where operatoer.opr_id = opr_id;
return opr_cpr;
end; //
delimiter ;
```

Funktionen operatoer_cpr modtager et operatør ID fra brugeren og returnerer derefter CPR-nummeret for den operatør med operatør ID'et.

```
delimiter //
create function pb_status(pb_id int(11)) returns integer
begin
declare status_result integer;
select status into status_result from produktbatch where produktbatch.pb_id = pb_id;
return status_result;
end; //
delimiter ;
```

Funktionen pb_status modtager et produktbatch ID fra brugeren og returnerer så statussen for dette produktbatch.

Vi har lavet lignende funktioner til de andre tabeller og de kan ses i bilaget. Der er dog nogle tabeller vi ikke har lavet funktioner til da disse tabeller har to eller flere primære nøgler. Det kan derfor ikke lade sig gøre at lave en funktion til dem, da man ikke kan finde den eksakte tuple uden at bruge begge primære nøgler og funktioner tager kun et input. En vej uden om dette kunne være at gøre en af primær nøglerne konstante i funktionen, men så ville funktionen kun virke for en værdi.

Stored procedures:

Stored procedures minder meget om functions. Den eneste reelle forskel er at stored procedures kan tage flere inputs og derfor også kan returnere flere outputs eller ændre en hel tuple i tabellen. Vi har hovedsageligt brugt det til at oprette, fjerne eller updatere tubles i de forskellige tabeller, f. eks har vi brugt det at oprette nye operatører.

Stored procedures følger følgende formular:

```
delimiter //
create procedure procedure_navn
(in input_navn1 inputtype, in input_navn2 inputtype,...)
begin
/*
SQL statement/procedurens funktionalitet
*/
end; //
delimiter ;
```

Her bruger vi igen delimiter //, så proceduren kan eksekvere statements inde i sig.

Vi har brugt følgende stored procedures i vores database:

```
delimiter //
create procedure add_operatoer
(in opr_id_input int(11), in opr_fornavn_input text, in opr_efternavn_input text,
 in ini_input text, in cpr_input text, in password_input text)
begin
insert into operatoer(opr_id, opr_fornavn, opr_efternavn, ini, cpr, password)
values(opr_id_input, opr_fornavn_input, opr_efternavn_input, ini_input, cpr_input, password_input);
end; //
delimiter ;
```

Add_operatoer proceduren tilføjer en ny operatør til operatør tabellen. Siden at vi har valgt at det kun er adminen der skal kunne oprette nye operatører skal der også indtastes CPR-nummer og password. Da dette er en procedure der står for at tilføje tupler har vi en insert statement stående inden i den og det er denne der eksekveres når man kalder proceduren. Hvis man gerne vil bruge denne procedure skriver man bare:

```
call add_operatoer('5', 'lars', 'hansen', 'LH', '122334-3455', 'password123');
```

En ny operatør med ID = 5 og navn lars hansen bliver nu oprettet i databasen og man kan nu tilføje ham til andre tabeller. Nedenfor ses en til stored procedure som her opretter en ny tuple i produktbatchkomponent tabellen. Den fungerer helt ligesom add_operatoer proceduren ovenfor.

```
delimiter //
create procedure add_produktbatchkomponent
(in pb_id_input int(11), in rb_id_input int(11),
 in tara_input double, in netto_input double, in opr_id_input int(11))
begin
insert into produktbatchkomponent(pb_id, rb_id, tara, netto, opr_id)
values(pb_id_input, rb_id_input, tara_input, netto_input, opr_id_input);
end; //
delimiter ;
```

Udover procedures der tilføjer ting til databasen har vi også procedures der kan fjerne ting fra databasen:

```
delimiter //
create procedure remove_operatoer(in opr_id_input int(11))
begin
delete from operatoer where opr_id = opr_id_input;
end; //
delimiter ;
```

Remove_operatoer proceduren gør ligesom navnet antydet, nemlig sletter en række fra en tabel. Man indtaster bare et operatør ID og så sletter denne procedure operatøren fra tabellen. Denne procedure bruges på følgende måde:

```
call remove_operatoer('1');
```

Efter dette er operatør med ID = 1 fjernet fra tabellen.

Vi har også lavet procedures der kan opdatere ting fra databasen, de ser sålede ud:

```
delimiter //
create procedure update_operatoer1(in opr_id_input int(11), in opr_fornavn_input text,
in opr_efternavn_input text, in ini_input text, in cpr_input text, in password_input text)
begin
update operatoer set opr_fornavn = opr_fornavn_input where opr_id = opr_id_input;
update operatoer set opr_efternavn = opr_efternavn_input where opr_id = opr_id_input;
update operatoer set ini = ini_input where opr_id = opr_id_input;
update operatoer set cpr = cpr_input where opr_id = opr_id_input;
update operatoer set password = password_input where opr_id = opr_id_input;
end; //
delimiter ;
```

update_operatoer1 opdaterer en operatør. Her indtastes et operatør ID og alle andre attributter i tuplen og så køres der en enkelt update for hver af attributterne. Dog kan operatør ID'et ikke opdateres da det er en primær nøgle og vi ikke vil have at primær nøglerne ændres, da de bruges i andre tabeller og vi så også skulle til at opdatere dem.

Resten af vores procedure kan ses i bilaget.

Transactions

I SQL er en transaction en handling som udføres på databasen og dets data. Handlingerne består typisk af SQL-kommandoerne; Select, Insert, Update og Delete, hvis funktion enten er at skrive eller læse til en database. Centralt til transactions er en række egenskaber, som sørger for at holde en database intakt og fri for fejl. Disse egenskaber betegnes for ACID: **A**tomiticy, **C**onsistency, **I**solation og **D**urability. For det første udføres en transaktion aldrig delvist. Det vil sige, at når en transaktion er færdig er det vigtigt, at vi ikke blot har ændret værdierne for f.eks. konto A og konto B, men at summen af de to kontoer er den samme efter en transaktion. Hvis dette krav er opfyldt

kan transaktionen gemmes og transaktionen anses vellykket. Er der derimod gået noget galt under transaktionen, som efterlader databasen i en forkert tilstand skal hele operationen afbrydes og en tilbagerulning sørger for, at databasen igen er som før den påbegyndte transaktion. Man siger, at transaktionen skal være atomar: Enten gennemføres alle funktioner i en procedure, eller ingen af dem. Isolation sikrer, at en transaktion foregår som en enkeltstående transaktion. Det vil sige, at hvis vi f.eks. forsøger at lave en række transaktioner fra konto A til konto B på samme tid må vi lade den ene transaktion eksekvere først. Fejl af planlægningen af de enkelte trin i transaktionen kan medføre en inkonsistent database. Derudover skal en database være holdbar altså, hvis der er sket en transaktion fra konto A til konto B skal transaktionen og den opdaterede data være intakt på f.eks. en backup-disk. Denne sikkerhed medfører bl.a. at man kan gå tilbage til et bestemt tidspunkt for at se, hvad der evt. er gået galt i en transaktion.

```
delimiter //
create procedure BatchMaengdeExchange1(in transfer int(11), in rb1 int(11), in rb2 int(11))
begin start transaction;
set@OldMaengde = (select maengde from raavarebatch where rb_id = rb1);
set@NewMaengde = @OldMaengde - transfer;
update raavarebatch set maengde = @NewMaengde where rb_id = rb1;
set@OldMaengde1 = (select maengde from raavarebatch where rb_id = rb2);
set@NewMaengde2 = @OldMaengde1 + transfer;
update raavarebatch set maengde = @NewMaengde2 where rb_id = rb2;
if(((@OldMaengde + @OldMaengde1) = ((select maengde from raavarebatch where rb_id = rb1) + (select maengde from raavarebatch where rb_id = rb2)))
and ((select raavare_id from raavarebatch where rb_id = rb1) = (select raavare_id from raavarebatch where rb_id = rb2)) and @oldMaengde >= transfer)
then commit;
else rollback;
end if;
end; //
delimiter ;
```

Metoden BatchMaengdeExchange står for transaktionen mellem to råvarebatches. Først startes transaktionen, som sørger for at gemme ændringer i en midlertidig buffer, hvis transaktionen ikke fuldføres og man så må foretage et rollback til før transaktionen begyndte. OldMaengde sættes til den mængde som råvarebatch 'rb1' havde før den påbegyndte transaktion og trækker efterfølgende den ønskede 'transfer'-mængde fra for at lave NewMaengde. Derefter opdateres mængden for 'rb1'. De tre steps udføres på samme vis for rb2, bare hvor 'transfer'-mængden lægges til den nye værdi og opdateres så. If-sætningen sikrer at databasen er intakt. Altså at der ikke lige pludselig er en højere eller lavere sum af de to råvarebatch's samlede mængde. Hvis if-sætningen er opfyldt commit'es ændringerne til databasen. Hvis ikke foretages et rollback.

JDBC

Connector

I vores java project bruger vi Connector klassen til at forbinde koden med databasen og sende SQL-forespørgsler til den. Det kan vi gøre på tre måder: "doQuery()" sender en hel SQL-forespørgsel og returnerer svaret som et resultSet hvor den data man har bedt om fra databasen er gemt. Vi bruger f.eks. "doQuery()" når vi gerne vil hente en hel tabel fra vores database. "doUpdate()" sender en hel SQL-forespørgsel men den returnerer ikke noget, så den er beregnet til at ligge noget forudbestemt ind i databasen. Vi bruger ikke "doUpdate()" i vores projekt, men vi kunne f.eks. bruge det hvis vi havde brug for en funktion der nulstillede vores brugere.

```
public ResultSet doQuery(String query) throws SQLException{
    Statement stmt = connection.createStatement();
    ResultSet res = stmt.executeQuery(query);
    return res;
}

public void doUpdate(String query) throws SQLException{
    Statement stmt = connection.createStatement();
    stmt.executeUpdate(query);
}

public Connection getConnection(){
    return connection;
}
```

Metoden "getConnection()" returnerer et connection objekt som vi bruger i næsten alle vores metoder, det betyder at vi selv kan oprette et statement objekt ud fra vores connection, i statement objektet kan vi så indsætte en skabelon til en SQL-forespørgsel, og udskifte spørgsmålstegnene i skabelonen med variabler fra vores java program. Det bruger vi f.eks. når vi vil hente en bestemt operatør, så vi kan lave en SQL-forespørgsel skabelon og have en variabel i stedet for at lave en hel forespørgsel til hver operatør.

createOperatoer

For at oprette en ny operatør og ligge den ind i databasen bruger vi metoden "createOperatoer()", den bruger data fra et OperatoerDTO objekt og en tekstfil til at lave en SQL-forespørgsel, og sende den til databasen.

```
public void createOperatoer(OperatoerDTO opr) throws DALException {
        try {
            PreparedStatement stmt = connector.getConnection().prepareStatement(Files.readAllLines(Paths.get("createCommands.txt")).get(0));
            stmt.setInt(1, opr.getOprId());
            stmt.setString(2, opr.getOprFornavn());
            stmt.setString(3, opr.getOprEfternavn());
            stmt.setString(4, opr.getIni());
            stmt.setString(5, opr.getCpr());
            stmt.setString(6, opr.getPassword());
            stmt.executeQuery();
        } catch (Exception e) {
            throw new DALException(e.getMessage());
        }
}
```

Først laver den et statement med en linje tekst hentet fra den første linje i filen "createCommands.txt" hvor der står: "call add_operatoer(?, ?, ?, ?, ?, ?);". Derefter udskifter den alle spørgsmålstegnene med id, navn, initialer, cpr og password som den henter fra OperatoerDTO objektet. Når alle spørgsmålstegnene er fyldt ud med den rigtige information er Statement objektet klar og "executeQuery()" metoden bliver brugt til at sende den færdige SQL-forespørgsel til vores database.

getOperatoer

For at hente information om en specifik operatør fra databasen bruger vi metoden "getOperatoer()", for at finde operatøren i databasen burges deres ID nummer da det altid er unikt til den operatør.

```
public OperatoerDTO getOperatoer(int oprId) throws DALException {
    ResultSet rs = null;
    try {
        PreparedStatement stmt = connector.getConnection().prepareStatement(Files.readAllLines(Paths.get("getCommands.txt")).get(0));
        stmt.setInt(1, oprId);
        rs = stmt.executeQuery();
    } catch (Exception e) {
        throw new DALException(e.getMessage());
    }
    try {
        if (!rs.first()) throw new DALException("Operatoeren " + oprId + " findes ikke");
        return new OperatoerDTO (rs.getInt("opr_id"), rs.getString("opr_fornavn"), rs.getString("opr_efternavn"), rs.getString("ini"),
                rs.getString("cpr"), rs.getString("password"));
    }
    catch (SQLException e) {
        throw new DALException(e.getMessage());
    }

}
```

Først laver den et statement med en linje tekst hentet fra den første linje i filen "getCommands.txt" hvor der står: "SELECT * FROM operatoer WHERE opr_id = ?;". Spørgsmålstegnet bliver så udskiftet med tallet i oprId og SQL-forespørgslen bliver sendt til databasen. Den forespørgsel returnere så et resultSet fra databasen der indeholder den data der høre sammen med den ID der blev brugt, det bliver så brugt til at oprette et nyt OperatoerDTO objekt der bliver brugt til at returnere dataen til der hvor der blev "getOperatoer()" blev kaldt.

SQL-forespørgsler - Forklarende tekst(se bilag for SQL statements)

SQL-forespørgslerne kan findes under bilag afsnittet ved bilag 4 og 5, hvortil der er fulde eksempler på hvordan SQL-udtrykkene ser ud for samtlige forespørgsler.

1. For at bestemme navnene på de råvarer som indgår i mindst to forskellige råvarebatches benytter vi "SELECT * FROM raavare" som vælger alle råvarer fra råvare tabellen. Herefter benyttes "WHERE raavare_id IN" som kigger på alle råvare ID i tabellen og i den følgende IN clause benyttes der "SELECT raavare_id FROM raavarebatch GROUP BY raavare_id HAVING COUNT(raavare_id) > 1)", som kigger på alle råvare ID fra råvarebatch og gruppere de råvare ID der indgår i råvarebatch mere end 1 gang.

2. Her skal man bestemme relationen som for hver receptkomponent indeholder tuplen (i. j, k) bestående af receptens ID, receptens navn og råvarens navn. Der benyttes "SELECT" til at vælge de tre værdier i tuplen. For at kunne finde de tre værdier benyttes der NATURAL JOIN, hvor man slår tabeller sammen. Dette benyttes på recept, raavare og receptkomponent tabellerne.

3. Denne forespørgsel er delt op i to spørgsmål:

3.1. Her skal man finde de recepter der enten indeholder skinke champignon eller begge dele. Der benyttes "SELECT DISTINCT" som vælger recept navn - DISTINCT betyder at den kun vil vise forskellige værdier. Herefter benyttes "FROM" som vælger tabellerne receptkomponent, raavare og recept og slår dem sammen med "NATURAL JOIN" og kalder tabellen rk1. I tabellen rk1 benyttes "WHERE" for at finde ud af om recepten indeholder råvare navnet "champignon". "OR

EXISTS” bruges til at kunne komme med et andet statement i clausen der kommer lige efter og finde ud af om det eksistere. Statementet i clausen efter “OR EXISTS” har den samme SELECT statement, hvor den kalder tabellen rk2. Forskellen her er at den sammenligner ved “WHERE” og “AND” som finder ud af om recept ID i rk1 har den samme som rk2 og ser om den indeholder råvare navnet “skinke”.

3.2. Her skal man finde de recepter som både indeholder skinke og champignon. Forespørgslen her er næsten nøjagtig den samme som 3.1. Forskellen her er at i stedet for “OR EXISTS” bruges “AND EXISTS”.

4. For at finde de recepter som ikke indeholder ingrediensen champignon benytter man “NOT EXISTS”. Først starter med med at vælge recept navn med “SELECT DISTINCT” for at undgå dubletter, og herfra vælges det fra receptkomponent hvor man slår tabellen sammen med råvare og recept tabellerne gennem “NATURAL JOIN” og kalder den samlet tabel for rk1. Herefter benyttes “WHERE NOT EXISTS”, hvor der efterfølgende i clausen har den samme “SELECT” statement, hvor den kalder den sammenslået tabel for rk2. Men her sammenligner den ved “WHERE” og “AND” som finder ud af om recept ID i rk1 har den samme som rk2 og ser om den indeholder råvare navnet “champignon”. Hvis den indeholder det, vil den ikke returnere den.

5. Her skal man finde de recepter der indeholder den største nominelle vægt af ingrediensen tomat. Man benytter “SELECT DISTINCT” der vælger recept navn uden at returnere dupletter. Dette vælger man med “FROM” fra tabellerne receptkomponent, recept og råvare som man slår sammen til én tabel med “NATURAL JOIN”. Derefter benyttes “WHERE”, “LIKE” og “AND” som kigger på råvare navnet tomat og dens nominelle vægt. Efterfølgende i clausen efter “AND” er der en “SELECT” statement som vælger tabellerne receptkomponent og råvare og slår dem sammen med “NATURAL JOIN”. Herefter ser den på den maksimale nominelle vægt af tomat med “MAX(nom_netto)” og “ LIKE”.

6. Her skal man bestemme relation for hvert produktbatchkomponent der indeholder tuplen (i, j, k) bestående af produktbatch ID, råvarens navn og råvarens netto vægt. For at gøre dette benyttes “SELECT DISTINCT” som vælger recept navn, uden at den returnere dubletter. “FROM” benyttes til at vælge recept navn fra den samlede tabel produktbatch, råvare og produktbacthkomponent,hvor benytter “NATURAL JOIN” til at slå tabellerne sammen.

7. Her skal der findes produktbatch ID’er for dem der indeholder en størst nettovægt af ingrediensen tomat. Dette forespørgsel minder om nr. 5. “SELECT DISTINCT” vælger pb_id fra produktbatchkomponent og råvare med “FROM” og slår de to tabeller sammen med “NATURAL JOIN”. Herefter kigger man på råvare navnet “tomat” og dens nettovægt med “WHERE” og “AND”. Efterfølgende kommer det en clause hvor der er en statement i som finder den maksimale nettovægt af tomat. Dette gøres med “SELECT MAX(netto)” fra råvare og produktbatchkomponent tabellen der bliver slået samme med “NATURAL JOIN”. Derefter benyttes “WHERE” til at finde råvare navnet tomat.

8. For at finde navnene på alle operatører som har været involveret i at producere partier af varen “margherita” benyttes “SELECT DISTINCT”, som vælger operatørere, og “FROM”, som vælger tabeller det skal findes fra - produktbatchkomponent, recept og operatoer. Tabellerne bliver slået sammen med “NATURAL JOIN”. Herefter benyttes “WHERE” til at finde dem som har været involveret i at producere “margherita”.

9. For at bestemme relationen for hvert produktbatchkomponent der indeholder tuplen (i, j, k, l, m, n) bestående af pb_id, status, råvare navn, nettovægt, recept navn og operatør navn skal alle tabeller slås sammen med “NATURAL JOIN” for at hente diverse informationer. “SELECT DISTINCT” benyttes til at vælge alle værdierne i tuplen hvor man herefter “NATURAL

JOIN" tabellerne produktbatch, produktbacthkomponent, raavare, recept, operatoer, receptkomponent og raavarebatch.

Q1. I denne forespørgsel skal man angive antallet af produktbatchkomponenter med en nettovægt på mere end 10 med aggregat funktionen "COUNT()". Måden hvorpå man bruger COUNT() er, at man bruger "SELECT" udtrykket før "COUNT", og i parentesen for "COUNT" skriver man navnet på en attribut ind. Resultatet er, at SQL programmet udskriver antallet af værdier for den attribut man har brugt i "COUNT" funktionen. For så specifikt at finde de værdier af attributten i kolonnen, hvor nettovægten er mere end 10, tilføjer man i samme udtryk "WHERE", og derefter specifikt hvad det er.

Q2. I SQL forespørgsel har vi skulle finde den samlede mængde af "tomat" som findes på lageret, dvs. den samlede mængde af tomat som optræder i raavarebatch-tabellen. For at finde den samlede mængde har vi benyttet os af aggregat funktionen "SUM()". Måden man bruger "SUM()" på er den samme som med "COUNT()". For at kunne vise mængden med "SUM()", har vi skulle foretage et "NATURAL JOIN" mellem raavare og raavarebatch, for at få alle de nødvendige attributter med. "NATURAL JOIN" udtrykket er i forespørgslen blevet skrevet efter aggregat funktionen, men før vores "WHERE". For at specificere at det er mængden af tomater vi vil have vist, har vi benyttet "WHERE raavare_navn = 'tomat';"

Q3. I denne SQL forespørgsel har vi for hver råvare skulle finde for hver ingrediens den samlede mængde af det bestemte ingrediens som findes på lageret. Måden dette er blevet gjort på var at benytte "SUM" funktionen igen, men fordi vi har skulle gøre det med alle ingredienser, har vi her også benyttet os af GROUP BY funktionen. GROUP BY funktionen gør, at man kan opdele de data man gerne specifikt vil have fat i efter forskellige kriterier. Her gruppere vi efter råvare navn og det skrives som: "GROUP BY raavare_navn;". I denne tabel har vi ligesom i foregående brugt "NATURAL JOIN" mellem raavare og raavare_batch.

Q4. I denne SQL forespørgsel, hvor vi leder efter råvarer, som indgår i mindst tre forskellige recepter, har vi benyttet os af et forgrenet SQL statement. I det første statement selecter vi alle råvarenavne fra raavareskemaet, hvor vi derefter starter et indre SQL udtryk hvor vi selecter raavare_id fra receptkomponent og bruger GROUP BY funktionen til at finde raavare_id. Herefter bruger vi funktionen HAVING, som er et alternativ til WHERE funtkionen når man skal bruge aggregat til at filtrere data. Efter having skriver vi "COUNT(raavare_id) >= 3);"

Brugertyper

Nedenfor ses de fire forskellige typer af brugere som kan logge ind på vores database. I access matrix er der angivet, hvilke tabeller de forskellige rolletyper har adgang til, og hvorvidt de kan skrive (W) eller kun læse (R) de data som står på de tables de har adgang til.

- **Admin**
- **Pharmaceut**
- **Produktionsleder**
- **Laborant**

The Access Matrix:

Ved at se på vores access matrix, kan man se, at administratoren er den eneste brugertype, som har fuld adgang til både at læse og redigere i alle skemaer i databasen. Ellers fordeler dataadgangen sig for de tre andre brugertyper efter relevans mht. deres arbejde inden for råvarerne.

	Operator	**Produktbatch**	**Produktnbacth-komponent**	**Raavare**	**Raavarebatch**	**Recept**	**Recept-komponent**
Admin	RW	RW	RW	RW	RW	RW	RW
Pharmaceut				R	R	RW	RW
Produktionsleder	R	RW	RW	RW	RW	R	R
Laborant		RW	RW	R	R	R	R

Konklusion

Ud fra vores rapport, og de dokumentationer der er skrevet ned i den, kan vi samlet set konkludere, at vi har opnået målene for denne rapport. Vores SQL forespørgsler er blevet udført og afprøvet med succes. Det samme gør sig gældende for vores JDBC, som succesfuldt forbinder vores SQL database til vores Java program. Vi har med den uddelte database formået at analysere den med flere typer diagrammer og normalisering. Vores opgave har dog ikke været lige så fokuseret på det med at oprette en database og java forbindelse som den har været fokuseret på den teoretiske forståelse af database konceptet. Derfor har vores rapport ikke gået i dybere detaljer om hvordan man opretter databasen fra bunden af.

Bilag

Bilag 1 - Views:

Her ses vores views:

```
CREATE VIEW getOperatoer AS
SELECT *
FROM operatoer;


CREATE VIEW getOperatoer2 AS
SELECT opr_id, opr_fornavn, opr_efternavn, ini
FROM operatoer;


CREATE VIEW OperatoerControl AS
SELECT *
FROM operatoer natural join produktbatch natural join produktbatchkomponent;



CREATE VIEW getProduktbatch AS
SELECT *
FROM produktbatch;


CREATE VIEW getproduktbatchkomponent AS
SELECT *
FROM produktbatchkomponent;


CREATE VIEW getraavare AS
SELECT *
FROM raavare;


CREATE VIEW getraavarebatch AS
SELECT *
FROM raavarebatch;


CREATE VIEW getrecept AS
SELECT *
FROM recept;

CREATE VIEW getreceptkomponent AS
SELECT *
FROM receptkomponent;
```

Bilag 2 - Functions:

Her vores functions.

```
delimiter //
create function operatoer_cpr(opr_id int(11)) returns text
begin
declare opr_cpr text;
select cpr into opr_cpr from operatoer where operatoer.opr_id = opr_id;
return opr_cpr;
end; //
delimiter ;

delimiter //
create function pb_status(pb_id int(11)) returns integer
begin
declare status_result integer;
select status into status_result from produktbatch where produktbatch.pb_id = pb_id;
return status_result;
end; //
delimiter ;

delimiter //
create function getRaavare_navn(raavare_id int(11)) returns text
begin
declare navn text;
select raavare_navn into navn from raavare where raavare.raavare_id = raavare_id;
return navn;
end; //
delimiter ;

delimiter //
create function getMaengde(rb_id int(11)) returns double
begin
declare maengde_rb double;
select maengde into maengde_rb from raavarebatch where raavarebatch.rb_id = rb_id;
return maengde_rb;
end; //
delimiter ;
```

Bilag 3 -Stored procedures:

add_procedure:

```
delimiter //
create procedure add_operatoer
(in opr_id_input int(11), in opr_fornavn_input text, in opr_efternavn_input text,
 in ini_input text, in cpr_input text, in password_input text)
begin
insert into operatoer(opr_id, opr_fornavn, opr_efternavn, ini, cpr, password)
values(opr_id_input, opr_fornavn_input, opr_efternavn_input, ini_input, cpr_input, password_input);
end; //
delimiter ;

delimiter //
create procedure add_produktbatch
(in pb_id_input int(11), in status_input int(11), in recept_id_input int(11))
begin
insert into produktbatch(pb_id, status, recept_id)
values(pb_id_input, status_input, recept_id_input);
end; //
delimiter ;

delimiter //
create procedure add_produktbatchkomponent
(in pb_id_input int(11), in rb_id_input int(11),
 in tara_input double, in netto_input double, in opr_id_input int(11))
begin
insert into produktbatchkomponent(pb_id, rb_id, tara, netto, opr_id)
values(pb_id_input, rb_id_input, tara_input, netto_input, opr_id_input);
end; //
delimiter ;

delimiter //
create procedure add_raavare
(in raavare_id_input int(11), in raavare_navn_input text, in leverandoer_input text)
begin
insert into raavare(raavare_id, raavare_navn, leverandoer)
values(raavare_id_input, raavare_navn_input, leverandoer_input);
end; //
delimiter ;



delimiter //
create procedure add_raavarebatch
(in rb_id_input int(11), in raavare_id_input int(11), in maengde_input double)
begin
insert into raavarebatch(rb_id, raavare_id, maengde)
values(rb_id_input, raavare_id_input, maengde_input);
end; //
delimiter ;

delimiter //
create procedure add_recept
(in recept_id_input int(11), in recept_navn_input text)
begin
insert into recept(recept_id, recept_navn)
values(recept_id_input, recept_navn_input);
end; //
delimiter ;

delimiter //
create procedure add_receptkomponent
(in recept_id_input int(11), in raavare_id_input int(11), in nom_netto_input double, in tolerance_input double)
begin
insert into receptkomponent(recept_id, raavare_id, nom_netto, tolerance)
values(recept_id_input, raavare_id_input, nom_netto_input, tolerance_input);
end; //
delimiter ;
```

remove_procedures:

```
delimiter //
create procedure remove_operatoer(in opr_id_input int(11))
begin
delete from operatoer where opr_id = opr_id_input;
end; //
delimiter ;


delimiter //
create procedure remove_produktbatch(in pb_id_input int(11))
begin
delete from produktbatch where pb_id = pb_id_input;
end; //
delimiter ;

delimiter //
create procedure remove_produktbatchkomponent(in pb_id_input int(11), in rb_id_input int(11))
begin
delete from produktbatchkomponent where pb_id = pb_id_input and rb_id = rb_id_input;
end; //
delimiter ;

delimiter //
create procedure remove_raavare(in raavare_id_input int(11))
begin
delete from raavare where raavare_id = raavare_id_input;
end; //
delimiter ;

delimiter //
create procedure remove_raavarebatch(in rb_id_input int(11))
begin
delete from raavarebatch where rb_id = rb_id_input;
end; //
delimiter ;

delimiter //
create procedure remove_receptkomponent(in recept_id_input int(11), in raavare_id_input int(11))
begin
delete from receptkomponent where recept_id = recept_id_input and raavare_id = raavare_id_input;
end; //
delimiter ;
```

update_procedures:

```
delimiter //
create procedure update_raavare(in raavare_id_input int(11), in raavare_navn_input text, in leverandoer_input text)
begin
update raavare set raavare_navn = raavare_navn_input where raavare_id = raavare_id_input;
update raavare set leverandoer = leverandoer_input where raavare_id = raavare_id_input;
end; //
delimiter ;

delimiter //
create procedure update_raavarebatch(in rb_id_input int(11), in raavare_id_input int(11), in maengde_input double)
begin
update raavarebatch set maengde = maengde_input where rb_id = rb_id_input and raavare_id = raavare_id_input;
end; //
delimiter ;

delimiter //
create procedure update_recept(in recept_id_input int(11), in recept_navn_input text)
begin
update recept set recept_navn = recept_navn_input where recept_id = recept_id_input;
end; //
delimiter ;

delimiter //
create procedure update_receptkomponent(in recept_id_input int(11), in raavare_id_input int(11), in nom_netto_input double, in tolerance_input double)
begin
update receptkomponent set nom_netto = nom_netto_input where recept_id = recept_id_input and raavare_id = raavare_id_input;
update receptkomponent set tolerance = tolerance_input where recept_id = recept_id_input and raavare_id = raavare_id_input;
end; //
delimiter ;

delimiter //
create procedure update_operatoer1(in opr_id_input int(11), in opr_fornavn_input text,
in opr_efternavn_input text, in ini_input text, in cpr_input text, in password_input text)
begin
update operatoer set opr_fornavn = opr_fornavn_input where opr_id = opr_id_input;
update operatoer set opr_efternavn = opr_efternavn_input where opr_id = opr_id_input;
update operatoer set ini = ini_input where opr_id = opr_id_input;
update operatoer set cpr = cpr_input where opr_id = opr_id_input;
update operatoer set password = password_input where opr_id = opr_id_input;
end; //
delimiter ;

delimiter //
create procedure update_produktbatch(in pb_id_input int(11), in status_input int(11))
begin
update produktbatch set status = status_input where pb_id = pb_id_input;
end; //
delimiter ;

delimiter //
create procedure update_produktbatchkomponent(in pb_id_input int(11), in rb_id_input int(11), in tara_input double, in netto_input double)
begin
update produktbatchkomponent set tara = tara_input where pb_id = pb_id_input and rb_id = rb_id_input;
update produktbatchkomponent set netto = netto_input where pb_id = pb_id_input and rb_id = rb_id_input;
end; //
delimiter ;

delimiter //
create procedure update_raavare(in raavare_id_input int(11), in raavare_navn_input text, in leverandoer_input text)
begin
update raavare set raavare_navn = raavare_navn_input where raavare_id = raavare_id_input;
update raavare set leverandoer = leverandoer_input where raavare_id = raavare_id_input;
end; //
delimiter ;
```

Bilag 4 - SQL forespørgsler del 1:

1. Bestem navnene på de råvarer som indgår i mindst to forskellige råvarer batches.

- SELECT * FROM raavare WHERE raavare_id IN (SELECT raavare_id FROM raavarebatch GROUP BY raavare_id HAVING COUNT(raavare_id) > 1);

2. Bestem relationen som for hver receptkomponent indeholder tuplen (i, j, k) bestående af receptens identifikationsnummer i, receptens navn j og råvarens navn k.

- SELECT recept_id, recept_navn, raavare_navn FROM recept NATURAL JOIN raavare NATURAL JOIN receptkomponent;

3. Find navnene på de recepter som indeholder mindst én af følgende to ingredienser (råvarer): skinke eller champignon. Find navnene på de recepter som indeholder både ingrediensen skinke og ingrediensen champignon.

- Finder recept, hvor der enten indeholder skinke, champignon eller begge ingredienser: SELECT DISTINCT recept_navn FROM receptkomponent rk1 NATURAL JOIN raavare NATURAL JOIN recept WHERE raavare_navn = 'champignon' OR EXISTS (SELECT DISTINCT recept_navn FROM receptkomponent rk2 NATURAL JOIN raavare NATURAL JOIN recept WHERE rk1.recept_id = rk2.recept_id AND raavare_navn = 'skinke');

- Finder recept, der både indeholder skinke og champignon: SELECT DISTINCT recept_navn FROM receptkomponent rk1 NATURAL JOIN raavare NATURAL JOIN recept WHERE raavare_navn = 'champignon' AND EXISTS (SELECT DISTINCT recept_navn FROM receptkomponent rk2 NATURAL JOIN raavare NATURAL JOIN recept WHERE rk1.recept_id = rk2.recept_id AND raavare_navn ='skinke');

4. Find navnene på de recepter som ikke indeholder ingrediensen champignon.

- SELECT DISTINCT recept_navn FROM receptkomponent rk1 NATURAL JOIN recept NATURAL JOIN raavare WHERE NOT EXISTS(SELECT DISTINCT recept_navn FROM receptkomponent rk2 NATURAL JOIN raavare NATURAL JOIN recept WHERE rk1.recept_id = rk2.recept_id AND raavare_navn = 'champignon');

5. Find navnene på den eller de recepter som indeholder den største nominelle vægt af ingrediensen tomat.

- SELECT DISTINCT recept_navn FROM receptkomponent NATURAL JOIN recept NATURAL JOIN raavare WHERE raavare_navn LIKE 'tomat' AND nom_netto = (SELECT

MAX(nom_netto) FROM receptkomponent NATURAL JOIN raavare WHERE raavare_navn LIKE 'tomat');

6. Bestem relationen som for hver produktbatchkomponent indeholder tuplen (i, j, k) bestående af produktbatchets identifikationsnummer i, råvarens navn j og råvarens nettovægt k.

- SELECT DISTINCT pb_id, raavare_navn, netto FROM produktbatch NATURAL JOIN raavare NATURAL JOIN produktbatchkomponent;

7. Find identifikationsnumrene af den eller de produktbatches som indeholder en størst nettovægt af ingrediensen tomat.

- SELECT DISTINCT pb_id FROM produktbatchkomponent NATURAL JOIN raavare WHERE raavare_navn = 'tomat' AND netto = (SELECT MAX(netto) FROM raavare NATURAL JOIN produktbatchkomponent WHERE raavare_navn = 'tomat');

8. Find navnene på alle operatører som har været involveret i at producere partier af varen margherita.

- SELECT DISTINCT opr_fornavn, opr_efternavn FROM produktbatchkomponent NATURAL JOIN recept NATURAL JOIN operatoer NATURAL JOIN produktbatch WHERE recept_navn = 'margherita';

9. Bestem relationen som for hver produktbatchkomponent indeholder tuplen (i, j, k,l, m, n) bestående af produktbatchets identifikationsnummer i, produktbatchets status j, råvarens navn k, råvarens nettovægt l, den tilhørende recepts navn m og operatørens navn n.

- SELECT DISTINCT pb_id, status, raavare_navn, netto, recept_navn, opr_fornavn, opr_efternavn FROM produktbatch NATURAL JOIN produktbatchkomponent NATURAL

JOIN raavare NATURAL JOIN recept NATURAL JOIN operatoer NATURAL JOIN receptkomponent NATURAL JOIN raavarebatch;

Bilag 5 - SQL forespørgsler del 2:

Q1. Angiv antallet af produktbatchkomponenter med en nettovægt på mere end 10. Hint: Brug aggregatfunktionen COUNT.

- SELECT COUNT(pb_id) FROM produktbatchkomponent WHERE netto > 10;

Q2. Find den samlede mængde af tomat som findes på lageret, dvs. den samlede mængde af tomat som optræder i raavarebatch-tabellen. Hint: Brug aggregatfunktionen SUM.

- SELECT SUM(maengde) FROM raavarebatch NATURAL JOIN raavare WHERE raavare_navn = 'tomat';

Q3. Find for hver ingrediens (råvare) den samlede mængde af denne ingrediens som findes på lageret. Hint: Modificér forespørgslen ovenfor. Brug GROUP BY.

- SELECT SUM(maengde), raavare_navn FROM raavarebatch NATURAL JOIN raavare GROUP BY raavare_navn;

Q4. Find de ingredienser (navne på råvarer) som indgår i mindst tre forskellige recepter.

- SELECT raavare_navn FROM raavare WHERE raavare_id IN (SELECT raavare_id FROM receptkomponent GROUP BY raavare_id HAVING COUNT(raavare_id) >= 3);