

Database Projekt 02327

DTU



DTU Compute

Institut for Matematik og Computer Science

Rapport af: Gruppe 19

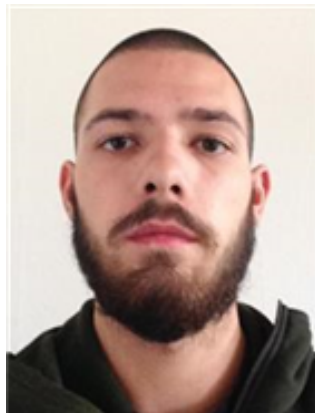
Bijan Negari
s144261

Jonathan Ask Larsen
s136381

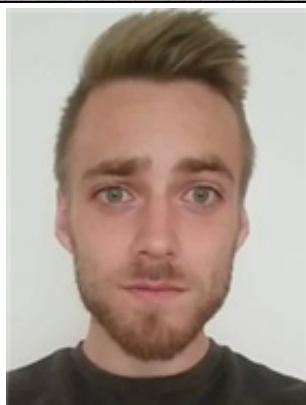
Stuart Benjamin McLean
s133018



Thomas Lien Christensen
s165242



Thomas Jakobsen
s136509



Indholdsfortegnelse

Indholdsfortegnelse	1
Introduktion	2
Opgavebeskrivelse	3
Database grundlag	4
Relationer	5
Normalisering	7
Mangler	8
Klassediagram	9
Design	11
Entity Relation Diagram	11
Enhanced Entity Relation Diagram	12
Implementation	13
JDBC	13
Roller	14
Views	15
Slet bruger	16
Transaktions logik - Afbrudt afvejning	17
Test	19
Konklusion	22
Bilag - Udsøgning i databasen	23
Bilag - ER diagram	31
Bilag - Views	32

Introduktion

Lavet af: Thomas J

Vi har fået til opgave at færdiggøre en udleveret database. Databasen er lavet i MySQL og vi skal lave implementeringen af et database interface i Java. Der er udleveret en del elementer i Java som mangler at få lavet interfaces, det er denne skabelon vores kode tager udgangspunkt i.

Det er et gruppeprojekt som en del af et uddannelsesforløb. Projektet er lavet som et selvstændigt projekt med intention om brug i et senere projekt.

Opgavebeskrivelse

Lavet af: Thomas J

Projektets database blev grundigt beskrevet med fremmednøgler og en forklaring til hver enkelt tabel. Hele databasens opsætning skulle forstås før næste opgave kunne løses. Projektbeskrivelsen kom herefter med en række SQL forespørgsler der skulle besvares. Vi skulle lave diverse udsøgninger og hermed også vise forståelse for opsætningen. Databasen skulle udvides så vi forsvarligt kunne slette brugere eller afbryde en afvejning, uden tab af data. De forskellige roller skulle have hver deres *view* for at almindelige brugere af programmet får begrænset deres brugerflade.

Efterfølgende skulle vi arbejde i udviklingsmiljøet Java. Alle rettelser i databasen skulle også opdateres i Java. derudover skulle vi også følge:

- 1. Implementér de resterende interfaces fra `dao/interfaces01917`. Implementationerne skal placeres i pakken `dao/impl01917`.*
- 2. Lav en test-klasse som tester din implementation. En meget simpel test-klasse som tester implementationen af `Operatoer` er allerede lavet i `Main`-klassen af `test01917`-pakken.”*

Vi skulle aflevere opgaven som en rapport, der forklarer alle de nødvendige oplysninger omkring projektet. Heriblandt skal der dokumenteres ved at benytte ER diagram, normalisering, transaktions logik og procedure.

Database grundlag

Lavet af: Thomas Lien Christensen

Til videre udvikling er der i starten af semesteret udleveret en fuldt funktionel database ved navn lab_database2. lab_database2 består af følgende tabeller:

- **operatoer**
 - Består af attributterne opr_id, opr_navn, ini, cpr og password. Dennes Primary key er opr_id.
- **raavare**
 - Består af attributterne raavare_id, raavare_navn og leverandoer. Primary key er raavare_id.
- **raavarebatch**
 - Består af attributterne rb_id, raavare_id og maengde. Primary Key er rb_id. Foreign key er raavare_id, der holder en reference til raavare-tabellens raavare_id.
- **recept**
 - Består af attributterne recept_id og recept_navn. Primary Key er recept_id.
- **receptkomponent**
 - Består af attributterne recept_id, raavare_id, nom_netto og tolerance. Primary Keys er recept_id og raavare_id. Disse primary keys er også foreign keys, og holder referencer til hhv. recept-tabellens recept_id og raavare-tabellens raavare_id.

- **produktbatch**

- Består af attributterne pb_id, status og receipt_id. Primary key er pb_id. Foreign key er receipt_id, der holder reference til receipt-tabellens receipt_id.

- **produktbatchkomponent**

- Består af attributterne pb_id, rb_id, tara REAL, netto REAL og opr_id. Primary keys er pb_id og rb_id. Foreign keys er pb_id, rb_id og opr_id, der holder referencer til hhv. produktbatch-tabellens pb_id, raavarebatch-tabellens rb_id og operatoer-tabellens opr_id.

Da en foreign key refererer til en anden tabels primary key, er brugeren begrænset i henhold til hvilke entiteter denne kan oprette. Vil brugeren eksempelvis oprette et raavarebatch med raavare_id = 4, er det kun muligt hvis der i forvejen findes en råvare i raavare-tabellen med raavare_id = 4. Dette er intuitivt klart. Du kan eksempelvis ikke have en bunke æbler hvis æblet ikke eksisterer. Vi ser nærmere på denne afhængighed når vi kigger på relationerne i databasen.

Relationer

Produktbatchkomponent | Operator

Relation: Én til mange

Flere operatoer kan afveje og gemme information til en produktbatchkomponent.

Raavarebatch | Raavare

Relation: Én til mange

Selvom det virker intuitivt korrekt, at et raavarebatch består af en mængde af den samme raavare og dermed giver en én til én relation, kan en råvare godt opdeles i flere raavarebatches.

Receptkomponent | Raavare

Relation: Én til mange

Som navnet hentyder, viser receptkomponent-tabellen hvilke komponenter (råvarer) der indgår i en recept (opskrift). Den egentlige "receptkomponent" er råvaren, af hvilke opskriften kan indeholde flere.

Produktbatchkomponent | Raavarebatch

Relation: Én til mange

Hvert produktbatchkomponent bruger en bestemt mængde råvarer fra de enkelte raavarebatches, der indgår i produktbatchkomponentet.

Recept | Receptkomponent

Relation: Én til mange

En recept kan have flere receptkomponenter.

Recept | Produktbatch

Relation: Én til mange

Der kan laves flere produktbatches på basis af den samme recept.

Produktbatch | Produktbatchkomponent

Relation: Én til mange

Et produktbatch kan have flere komponenter.

Normalisering

Lavet af: Thomas J

Da vi har fået vores database udleveret har vi måtte gennemgå den for at sikre os at normaliseringen var som ønsket. Vi laver blandt andet normaliseringen for at undgå redundant- og inkonsekvent data. For at undersøge hvor dybt vores normalisering går tjekker vi kronologisk om de er overholdt.

Vi startede derfor med at tjekke om normal form 1 var overholdt. For at NF1 skal være overholdt skal værdierne i alle tabellerne kunne beskrives som atomare. Derudover skal hver række have en unik primær nøgle og alle værdierne i hver kolonne skal komme fra det samme domæne. Vi gennemgik vores database og konkluderede at NF1 er overholdt.

```
mysql> select * from operatoer;
```

opr_id	opr_navn	ini	cpr	password	status
1	Angelo A	AA	070770-7007	lKje4fa	1
2	Antonella B	AB	080880-8008	atoJ21v	1
3	Luigi C	LC	090990-9009	jEfm5aQ	1

```
3 rows in set (0.00 sec)
```

Her ses opr_id som værende vores primærnøgle som overholder NF1 ved at være unik.

For at 2. normal form skal være overholdt skal 1. normal form også. Man kan aldrig have overholdt en normal form højere end den laveste ikke godkendte normal form. 2NF kræver at i tilfælde hvor primærnøglen består af 2 eller flere kolonner, skal alle ikke-nøgle kolonner være afhængige af hele primærnøglen og ikke delmængder af den.


```
mysql> select * from receptkomponent;
```

recept_id	raavare_id	nom_netto	tolerance
1	1	10	0.1
1	2	2	0.1
1	5	2	0.1
2	1	10	0.1
2	3	2	0.1
2	5	1.5	0.1
2	6	1.5	0.1
3	1	10	0.1
3	4	1.5	0.1
3	5	1.5	0.1

Her ses at hverken recept_id eller raavare_id i tabellen er unikke, men kombinationen mellem dem er unikke. Både nom_netto og tolerance er afhængige af kombinationen.

Den 3. normal form bestemmer at alle attributter kun må være afhængige af primære nøgler og ikke andre attributter eller en kombination af andre attributter. I alle vores tabeller har vi en primærnøgle eller en kombination af to fremmednøgler der sammenlagt danner vores primærnøgle. Da er ingen af vores tabeller, som har attributter, afhængige af ikke-primærnøgler. Den udleverede database overholder altså som udgangspunkt reglerne for 3NF.

Mangler

Skrevet af: Thomas Lien Christensen

Den udleverede database udgør et solidt fundament, der er opbygget efter de rigtige principper. Dog er der ting som den endnu ikke kan.

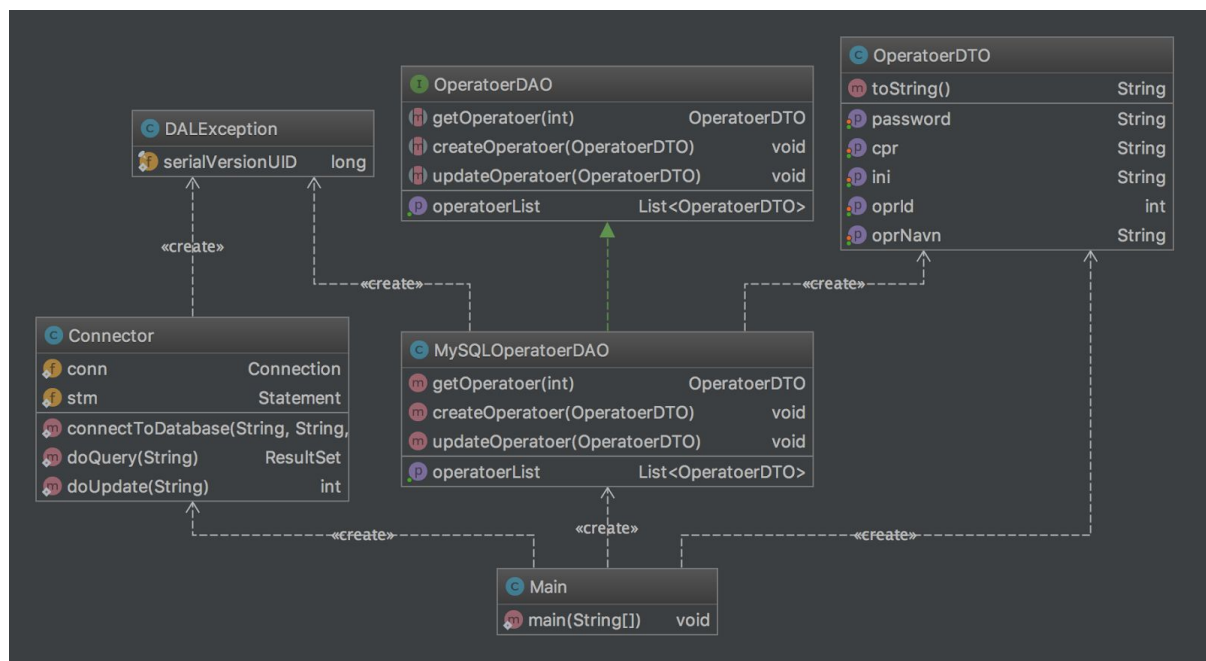
For at databasen fungerer efter hensigten, er vi nødt til at implementere yderligere tabeller, views og eventuelle metoder / procedurer. Derudover skal der implementeres transaktions logik og en optimal måde at slette operatører på, med minimal risiko for tab af data. Yderligere uddybning følger i afsnittet om implementation.

Klassediagram

Lavet af: Thomas J og Jonathan

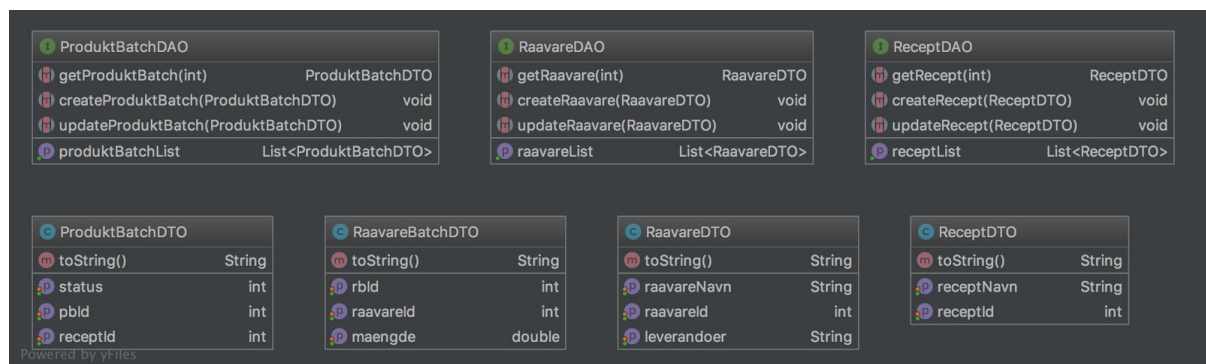
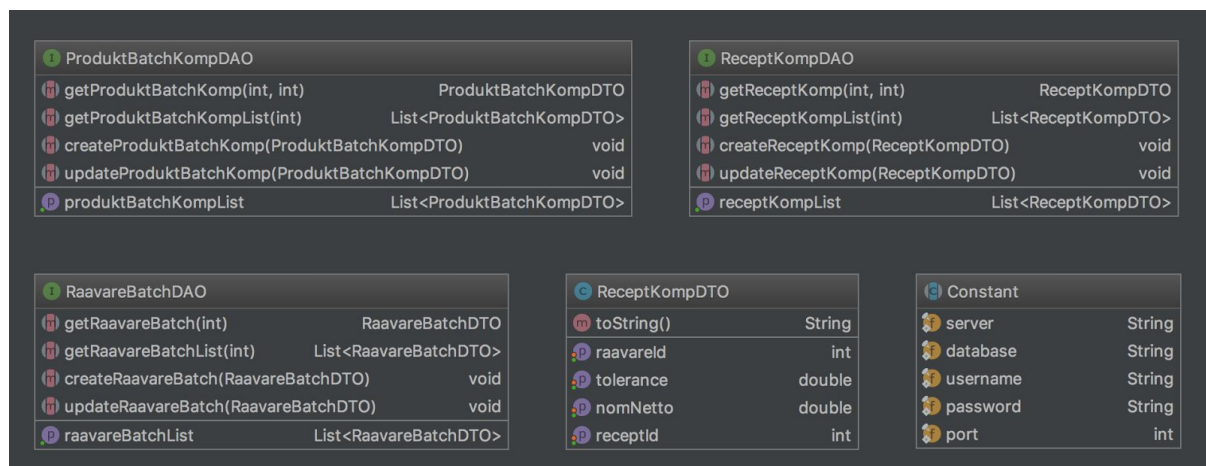
Vi havde fået en stor del prædefineret kode udleveret og deraf har vi kunne samle vores analyse klassediagram som vist herunder. Klassernes relationer fremgår tydeligt og overholder de regler for GRASP som vi mener gør sig gældende.

Klassediagrammet viser hvordan vi starter programmet i vores Main og opretter instanserne MySQLOperatoerDAO, Connector og OperatoerDTO. Vores Connector skal herefter starte en forbindelse op til vores MySQL database.



Klassediagrammet med dets relationer som udleveret.

I den udleverede Java source code er der en masse ekstra klasser, hvoraf mange er prædefinerede og en del af dem mangler at blive lavet færdige. Disse klasser indgår ikke i vores analyse klassediagram da vi ikke har fastlagt relationerne mellem dem. Til gengæld er de byggesten når vi skriver koden og får samlet relationerne mellem dem. Der er desuden lavet nogle interfaces der skal færdiggøres og opretholdes.



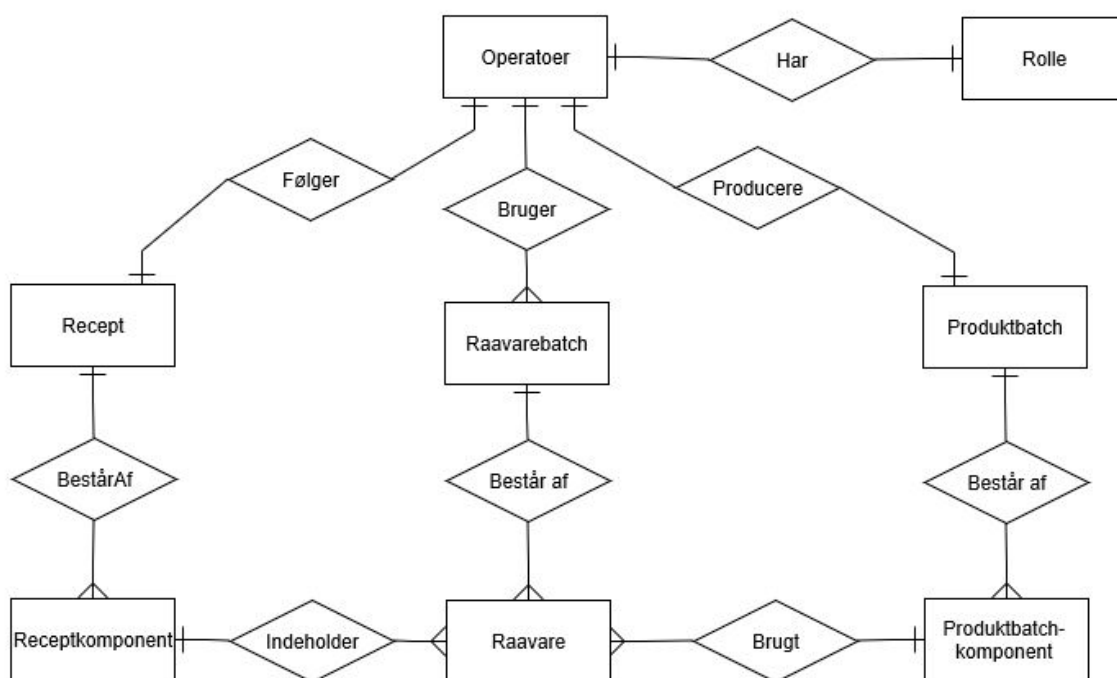
Yderligere billeder af klassediagrammer kan findes i bilag.

Design

Entity Relation Diagram

Lavet af: Thomas J

Vi har som del af opgavebeskrivelsen skulle lave et entity-relationsdiagram. Det består af vores entiteter (Rektangler) som beskriver vores tabeller fra databasen. Hver entitet bør være relateret til mindst en anden entitet. Relationen bliver beskrevet kort i vores rombe. Alle relationer skal være en af de fire følgende: *one to one*, *one to many*, *many to one* el. *many to many*. Det kan bedst beskrives med vores ene *Operatoer* har én rolle, altså en *one to one* relation, hvorimod vores ene *recept* består af mange *receptkomponenter*.

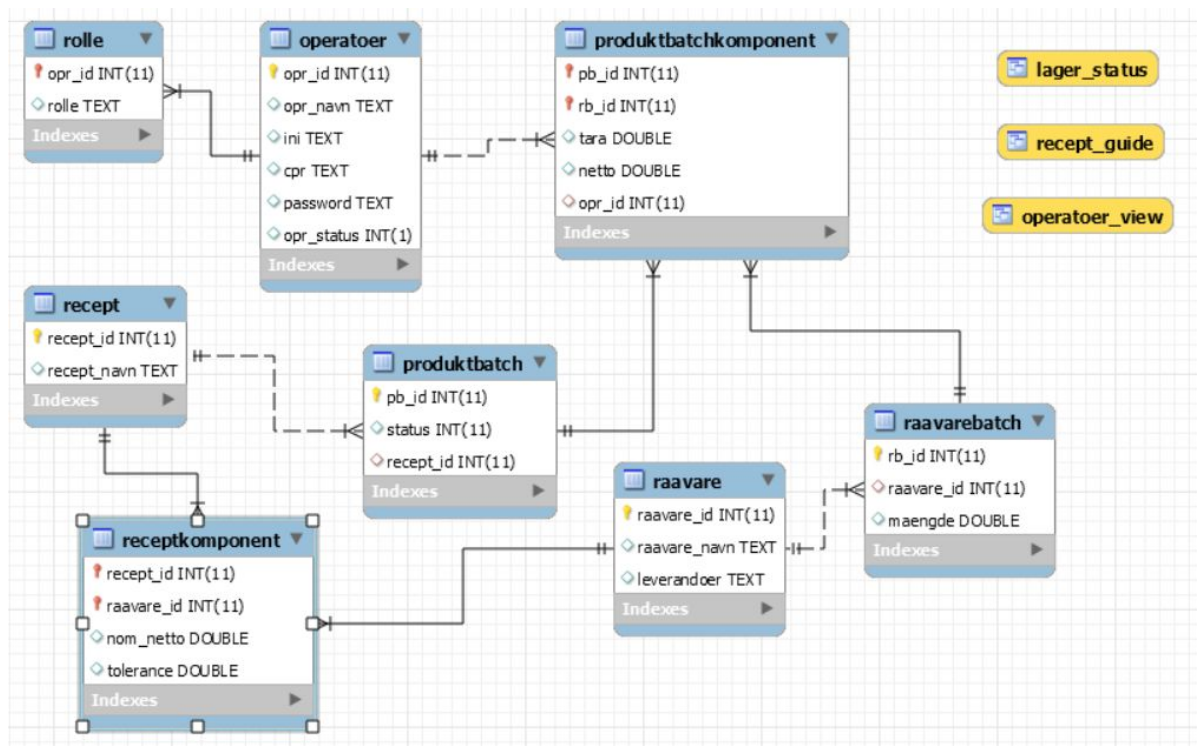


Der er et fuldt billede af vores ER diagram vedlagt som bilag med primær nøgler.

Enhanced Entity Relation Diagram

Lavet af: Thomas J

Vi har sammensat et EER diagram for databasen for at vise relationerne mellem tabellerne. I modsætning til det almindelige ER diagram, viser det her relationerne i henhold til databasen hvor ER diagrammet forklare til den mere gængse bruger hvordan man bruger diagrammet. Skemaet er autogenereret af MySQL Workbench og viser primær nøgler med gult og fremmednøgler med rødt. De gule felter er vores views.



Diagrammet er desuden med til at vise hvordan vores 3. normal form bliver overholdt. Nærmere beskrivelse findes under normalisering i afsnittet om databasens grundlag.

Implementation

JDBC

Lavet af: Jonathan

For at skabe forbindelse mellem JAVA og MySQL skal et Java Database Connectivity API (JDBC) inkorporeres i vores projekt. Denne gør det muligt at skrive SQL kommandoer i JAVA og sende dem ned til databasen. Vi kan nu benytte os af diverse opdaterings kommandoer som SQL's CREATE, UPDATE, DELETE mm.

I vores connector01917 klasse implementeres JDBC'en og giver hermed andre klasser mulighed for at sende kommandoer ned i serveren, ved at importere connectoren. Her peges også på vores database ved hjælp af jdbc:mysql som benytter HOST, PORT og databasens navn til at definere stien til databasen. JDBC benyttes også til at håndtere transaktioner og giver bl.a. muligheden for at kontrollere hvornår og hvordan commits bliver lavet til databasen hvis man ønsker at placere transaktionerne i java delen.

Roller

Lavet af Stuart McLean:

Rollerne Admin, Foreman, Operator, Pharmacist kan blive tildelt inde i en tabel kaldt "rolle". Alt dette kan blive gjort via de forskellige lag, grænseflade, funktionalitet og data. I interfacet når man laver en "operator" får man mulighed for at tildele en rolle af de 4 typer. Rollernes navne og id bliver sat i OperatoerDTO, hvorefter referencen bliver brugt af MySQLOperatoerDAO. I klassen MySQLOperatoerDAO bliver rollen tildelt en bruger i databasen, når "createOperatoer" metoden bliver kaldt. Rollerne har man mulighed for at slette eller ændre i metoden "updateRolle". Rollerne har ikke nogle administrative rettigheder tildelt, men dette kan gøres muligt i videreudviklingen af programmet.

Views

Lavet af: Thomas J

Views er et værktøj vi bruger i vores projekt for at den almene bruger kan bruge vores database. Dette giver os muligheden for at tildele en række rettigheder til de enkelte brugere i form af *select* statements. Vores views giver ikke brugerne yderligere rettigheder til database. Det er desuden en sikkerhed for at vores forskellige brugertyper ikke har adgang til data uden for deres eget felt.

I vores projekt har vi oprettet en række views for at brugerne af vores database har nemmere adgang til den data der er dem relevant. Vi har lavet et view med henblik på operatører eller foreman kan slå sine kollegaer op i en tabel. Dette er vores *operatoer_view* som viser alle bruger af systemet som ikke er slettet. Vi har derudover også lavet et view med henblik på vores pharmacists. Dette view bliver kaldt med *recept_guide* som viser recepterne med navne, mængde i stedet for tal, for at opretholde en brugervenlig forståelse.

Vores views er ikke implementeret som en del af vores Java kode. Vi har dog vist teorien for at lave forskellige views i sql.

Operatoer_view

- CREATE VIEW lab_database2.operatoer_view AS SELECT opr_id, opr_navn, ini, password FROM operatoer WHERE opr_status=1;

Viser databasens aktive brugere og operatører.

Recept_guide

- CREATE VIEW lab_database2.recept_guide AS SELECT recept_id, recept_navn, raavare_navn, nom_netto FROM recept NATURAL JOIN receptkomponent NATURAL JOIN raavare WHERE recept_id LIKE '%%' ORDER BY recept_id;

Viser recepterne med navne og mængde på de råvare der skal bruges.

Lager_status

- CREATE VIEW lab_database2.lager_status AS SELECT raavare_id, raavare_navn, leverandoer, maengde FROM raavare NATURAL JOIN raavarebatch ORDER BY raavare_id;

Viser hvor meget af hver råvare der er tilbage på lageret.

Slet bruger

Lavet af: Thomas J

For at slette en bruger fra systemet har vi måtte tilføje nogle ændringer i databasen. Vi har givet alle operatørerne en *opr_status* der blot indikere om brugeren er aktiv eller ej. Er brugeren aktiv agere databasen som hidtil. Er brugeren derimod sat til inaktiv så vil man ikke kunne logge ind i systemet og foretage ændringer. Det er kun administratorer der kan se brugere, der er inaktive.

Vi har valgt denne her fremgangsmåde for at undgå datatab inde i tabellen i tilfælde af en bruger har lavet et produkt parti og så efterfølgende bliver slettet. Alternativet var at oprette en ny tabel og flytte slettede brugere dertil - igen for at undgå et eventuelt datatab. Arbejdsprocessen virkede dog omstændig og unødvendig.

Transaktions logik - Afbrudt afvejning

Lavet af: Thomas & Jonathan

Vi har lavet afvejningssystemet således at hvis det bliver afbrudt af hvilken som helst årsag, så bliver intet gennemført. Vi har med andre ord lagt vores afvejningsproces ind i en transaktions blok. Det sikrer os fra ikke at have ufuldkomne oplysninger i databasen, og at der ikke pludselig er taget et parti fra råvarebatchen uden, at der er blevet produceret noget eller sat en produktion i gang. Vi indkapsler vores queries og transaktion i en "procedure". Dette gør at vi kan sende data fra java'en og ned til proceduren som derved kan køre en transaktion.

Transaktions logikken bygger på et "alt eller intet" koncept. Database logikken bliver stillet op og en række kommandoer bliver rammet ind i denne transaktion blok. Man laver sin afvejning men i tilfælde af at man bliver afbrudt, vil transaktions logikken lave et *rollback* uden at gemme de indtastede oplysninger. Vi har valgt at lave et eksempel som består af `createRecept` i klassen `MySQLReceptDAO`. Her sendes dataen til vores procedure "createNewRecept" sætter vi nogle exceptions til at håndtere eventuelle fejl når data bliver overført og skal gemmes via vores transaktion. Herefter startes vores transaktion som implementere dataen eller laver et rollback hvis vores exceptions bliver kørt. Herved undgår vi ufærdig data i vores database. Transaktioner kan også sættes til at kontrollere om værdier er blevet overført korrekt og fejlfinde på queries. hvis en transaktion finder en fejl her vil den også lave et rollback

```
public void createRecept(ReceptDTO recept) throws DALException {
    try
    {
        String query =
            "Call " +
            "(" + recept.getReceptId() + ", '" + recept.getReceptNavn() + "'"");

        PreparedStatement pstmt = this.connector.getConnection().prepareCall( sql: "{call CreateNewRecepts(?,?,?)}");

        pstmt.setInt( parameterIndex: 1, recept.getReceptId());
        pstmt.setString( parameterIndex: 2, recept.getReceptNavn());

        //pstmt.registerOutParameter(3, java.sql.Types.VARCHAR);
        pstmt.executeUpdate();

    } catch ( SQLIntegrityConstraintViolationException e) {
        e.printStackTrace();
        throw new DALException("Duplicate entry");
    } catch (SQLException e){
        e.printStackTrace();
    }
}
```

```

CREATE PROCEDURE `CreateNewRecept`(in_id IN recept.recept_id%TYPE, in_navn IN recept.recept_navn%TYPE)
BEGIN

DECLARE exit handler for sqlexception
BEGIN
--ERROR
ROLLBACK;
END;

DECLARE exit handler for sqlwarning
BEGIN
--WARNING
ROLLBACK;
END;

START TRANSACTION;
INSERT INTO recept(raavare_id, recept_navn)
VALUES (in_id, in_navn)
COMMIT;
END
$$

```

vores ovenstående procedure skal have foretaget visse ændringer for at virke med den nuværende java del.

Test

Lavet af: Thomas J & Thomas Lien Christensen

Projektet vi fik udleveret har en test klasse som indeholdte vores Main. Den hentede en eksisterende bruger, oprettede en ny, opdaterede en operatør og viste en liste med alle operatørerne. Den udleverede test viste os at det virkede, men det testede det ikke igennem, og dokumentationen var mangelfuld.

Vi fjernede den gamle test klasse og erstattede den med JUnit test hvor vi løber alle databasen tabeller igennem for *INSERT* og *DELETE*. Vi tager databasens svar og laver til en tekststreng med toString() og sammenligner input med databasens indhold.

```
//Test opret og get række i raavarebatch
public String opretRaavareBatch() throws Exception {
    RaavareBatchDTO rb = new RaavareBatchDTO(8, 8, 500);
    MySQLRaavareBatchDAO rDAO = new MySQLRaavareBatchDAO(connector);
    RaavareDTO rv = new MySQLRaavareDAO(connector).getRaavare(8);
    rDAO.createRaavareBatch(rb);
    System.out.println("Raavarebatch oprettet. Primary key: rb_id = 8");
    System.out.println(rDAO.getRaavareBatchList(rv.getRaavareId()).toString() + "\n");
    return rDAO.getRaavareBatchList(rv.getRaavareId()).toString();
}
```

Ovenstående er et udklip af koden fra JUnitDBTest klassen, hvor CREATE for samtlige tabeller forekommer i metoder magen til. Kode-udklippet til højre viser hvordan rDAO.getRaavareBatchList((xx).toString()) bliver sammenlignet med en hard-coded "forventet" (se String expected) tekststreng.

```
@Test
public void test() {
    JUnitDBTest test = new JUnitDBTest();
    String expected = "[8, 8, 500.0]";
    System.out.println("Expected: " + expected);
    try {
        String actual = test.opretRaavareBatch();
        System.out.println("Actual: " + actual);
        assertNotNull(actual);
        assertNotNull(expected);
        assertEquals(expected, actual);
    }
    catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

Samtlige tests bliver kørt i den rigtige rækkefølge, og brugeren kan sideløbende følge med i databasen og tjekke at alt stemmer overens.

Der er ikke testet for *UPDATE* med automatisk test. *UPDATE* virker dog på lige fod med *INSERT*, der bliver gennemgået som en del af JUnit testen på normal vis. For at undgå dobbeltarbejde, er det gjort muligt at køre testen om og om igen. Denne

funktion sletter alle de oprettede og altererede elementer og dermed giver en mulighed for at køre testen igen uden at få dubletter i databasen. Der skal dog tages forbehold for at du ikke bare kan slette tabeller i databasen, da vi har referencer mellem *parent* og *child*.

For at undgå *key restraint* skal tabellernes rækker droppes i den rigtige rækkefølge. I stedet for at hard-code test-gendannelsen, er der i stedet lavet en procedure kaldet `resetTest`, der kan benyttes ved brug af MySQLs `CALL` funktion.

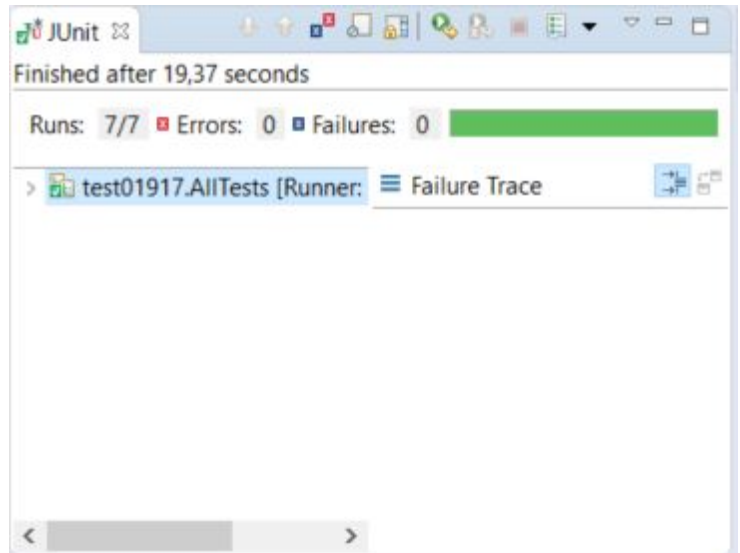
```
mysql> CREATE PROCEDURE resetTest ()
-> BEGIN
-> DELETE FROM produktbatchkomponent WHERE pb_id = 6 AND rb_id = 8;
-> DELETE FROM operatoer WHERE opr_id = 8;
-> DELETE FROM raavarebatch WHERE rb_id = 8;
-> DELETE FROM raavare WHERE raavare_id = 8;
-> DELETE FROM produktbatch WHERE pb_id = 6;
-> DELETE FROM receptkomponent WHERE recept_id = 4;
-> DELETE FROM recept WHERE recept_id = 4;
-> END
-> //
Query OK, 0 rows affected (0.00 sec)
```

```
public void resetTestResults() {
    System.out.println("\nTryk på en knap for at gendanne testen.");
    scan.nextLine();
    try {
        //CALL resetTest kører den gemte procedure resetTest i databasen
        connector.executeUpdate("CALL resetTest");
    }
    catch (SQLException ex) {
        ex.printStackTrace();
    }
}
```

Alle tests bliver kørt på samme tid. På figurerne nedenunder kan det ses, at JUnit returner 0 fejl på de 7 test cases.

```
@RunWith(Suite.class)
@SuiteClasses({
    OperatorTest.class,
    RaavareTest.class,
    RaavareBatchTest.class,
    ProduktBatchTest.class,
    ProduktBatchKompTest.class,
    ReceptTest.class,
    ReceptKompTest.class
})

public class AllTests {
}
```



Vi har testes vores views manuelt i databasen og vedlagt skærmpoint i bilag.

Konklusion

Skrevet af: Jonathan

Vi har i dette projekt lært hvorledes en delvist opbygget database med tilhørende java del skal opsættes og færdiggøres. Vi har opsat en analysedel for at overskueliggøre arbejdsopgaverne. En af de primære færdigheder vi kan tage med os er hvorledes en databasestruktur skal opstilles for at skabe en stabil database uden redundant data. Vi har opbygget vores tabeller således at de overholder normalformerne og opsat relevante views for at overskueliggøre gængs data. Ydermere har vi lært om hvilke sikkerhedshensyn der bør foretages under databehandling i form af transaktioner i enten JDBC eller SQL. Vi har fået en forståelse for at sende queries ned i databasen gennem java ved brug af en JDBC connector og udført test af vores java og SQL del i projektet.

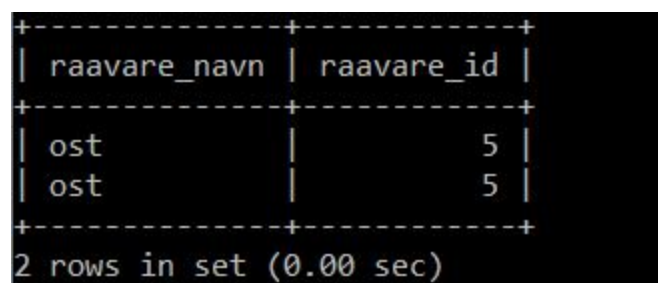
Bilag - Udsøgning i databasen

Lavet af: Stuart, Bijan, Thomas C, Thomas J og Jonathan.

Som en del af opgaven skal vi kunne søge i vores database på forskellig vis og kunne fremstille vores resultater på en forståelig og kortfattet måde. Vi er blevet stillet en række spørgsmål som vi kort har beskrevet, besvaret og forklaret heriblandt også skrevet den kommando der er blevet brugt i kommandoprompten.

1: Vi laver her en tabel med de råvare som bliver brugt i mindst to forskellige råvare batches. Tabellen viser de råvare som går igen, inklusiv sig selv. Det vil i dette tilfælde sige at den viser de *raavare_id* nummer som går igen. Havde der været 3 eller flere af samme identifikationsnumre var de også blevet vist.

```
SELECT raavare_navn, raavare_id FROM `lab_database2`.`raavare` natural join
`lab_database2`.`raavarebatch` WHERE raavare_id IN (SELECT raavare_id FROM
(SELECT raavare_id FROM raavarebatch GROUP BY raavare_id HAVING
COUNT(*) > 1) x);
```



raavare_navn	raavare_id
ost	5
ost	5

2 rows in set (0.00 sec)

2: For at vise en tabel af *receptkomponent* med læsbare navne i stedet for identifikationsnumre på opskriften og råvare skal vi udspecificere at vi vil lave en tabel med *recept_id*, *recept_navn* og *raavare_navn*. Vi laver *natural join* funktionen mellem de tabeller vi henter informationer fra for at undgå at gengive data flere gange. Til sidst i vores query siger vi at den skal gøre det for alle *recept_id* og sortere her efter.

```
SELECT recept_id, recept_navn, raavare_navn FROM `lab_database2`.`recept`
natural join `lab_database2`.`receptkomponent` natural join
`lab_database2`.`raavare` WHERE recept_id LIKE '%%' ORDER BY recept_id;
```

recept_id	recept_navn	raavare_navn
1	margherita	dej
1	margherita	tomat
1	margherita	ost
2	prosciutto	dej

3: Vi starter med at finde de opskrifter som indeholder enten skinke eller champignon med følgende SQL. Vi laver en query der kigger opskrifterne igennem og hvis en af dem indeholder "sKinke" eller "champignon" så viser vi opskriftens navn.

```
SELECT distinct recept_navn FROM `lab_database2`.`raavarebatch` natural join
`lab_database2`.`raavare` natural join `lab_database2`.`receptkomponent` natural
join `lab_database2`.`recept` WHERE raavare_navn LIKE '%skinke%' OR
raavare_navn LIKE '%champignon%';
```

recept_navn
prosciutto
capricciosa

2 rows in set (0.02 sec)

3 fortsat: Som opfølgning på ovenstående, skal vi også finde de opskrifter som indeholder både “skinke” og “champignon”.

```
SELECT distinct recept_navn FROM `lab_database2`.`raavarebatch` natural join  
`lab_database2`.`raavare` natural join `lab_database2`.`receptkomponent`  
natural join `lab_database2`.`recept` WHERE raavare_navn LIKE '%skinke%' OR  
raavare_navn LIKE '%champignon%';
```

The screenshot shows a database query tool interface. At the top, there's a 'Result Grid' tab with a table containing three rows: 'recept_navn', 'prosciutto', and 'capricciosa'. Below this is a 'Result 4' tab. The 'Output' section shows a dropdown menu set to 'Action Output'. Below that, a log table displays the execution of the query. The log table has columns for '#', 'Time', 'Action', and 'Message'. The first row shows a successful execution at 23:24:10, returning 2 rows.

#	Time	Action	Message
1	23:24:10	SELECT distinct recept_navn FROM `lab_database2`.`raavarebatch` natural join `lab_...	2 row(s) returned

4: For at finde navnene på de recepter som ikke indeholder champignon gør vi følgende. Den returnere *recept_navn* og alle dets råvare

```
SELECT distinct recept_navn, raavare_navn FROM receptkomponent rk1 natural
join raavare natural join raavarebatch natural join recept WHERE NOT exists
(SELECT * FROM receptkomponent rk2 natural join raavare natural join
raavarebatch natural join recept WHERE rk1.recept_id = rk2.recept_id AND
raavare_navn LIKE '%champignon%');
```

recept_navn	raavare_navn
margherita	dej
prosciutto	dej
margherita	tomat
prosciutto	tomat
margherita	ost
prosciutto	ost
prosciutto	skinke

7 rows in set (0.02 sec)

5: For at finde navnene på de recepter den indeholder den største mængde nominelle vægt af tomat, skal vi blot se efter den højeste værdi der er i tabellen *maengde* af navnet "tomat".

```
SELECT recept_id, recept_navn, raavare_navn, max(maengde) FROM
`lab_database2`.`raavarebatch` natural join `lab_database2`.`raavare` natural join
`lab_database2`.`receptkomponent` natural join `lab_database2`.`recept` WHERE
raavare_navn LIKE '%tomat%';
```

recept_id	recept_navn	raavare_navn	max(maengde)
1	margherita	tomat	300

1 row in set (0.00 sec)

6: For at bestemme relationen for hver produktbatchkomponent som indeholder tuplen bestående af produktbatch identifikationsnummeret, råvarens navn og råvarens netto vægt, har vi lavet en tabel hvor det klart fremstår hvor meget der er der er brugt af produktbatchkomponenten ad gangen.

```
SELECT rb_id, raavare_navn, netto FROM `lab_database2`.`raavare` natural join  
`lab_database2`.`raavarebatch` natural join  
`lab_database2`.`produktbatchkomponent` natural join  
`lab_database2`.`produktbatch` WHERE rb_id LIKE '%%' ORDER BY rb_id;
```

rb_id	raavare_navn	netto
1	dej	10.05
1	dej	10.01
1	dej	10.07
1	dej	10.02
2	tomat	2.03
2	tomat	1.99
3	tomat	2.06
4	ost	1.98
4	ost	1.55
5	ost	1.47
5	ost	1.57
6	skinke	1.53
6	skinke	1.03
7	champignon	0.99

14 rows in set (0.01 sec)

Vi ser blandt andet at der er taget af råvare batch med identifikationsnummer 1, 4 gange og der fremstår til højre i hver tupel hvor meget der er taget ad gangen.

7: For at finde de productbatches med størst indhold af tomat gør vi igen brug af SQL's indbyggede max funktion til at finde den maksimale *nom_netto* hvor vores råvare er tomat.

```
SELECT recept_id, pb_id, raavare_navn, max(nom_netto) FROM  
`lab_database2`.`raavarebatch` natural join `lab_database2`.`raavare` natural join  
`lab_database2`.`produktbatch` natural join `lab_database2`.`recept` natural join  
`lab_database2`.`receptkomponent` WHERE raavare_navn LIKE '%tomat%';
```

```
+-----+-----+-----+-----+  
| recept_id | pb_id | raavare_navn | max(nom_netto) |  
+-----+-----+-----+-----+  
|          1 |      1 | tomat        |                2 |  
+-----+-----+-----+-----+  
1 row in set (0.00 sec)
```

8: For at finde alle operatørene der har været involveret i at lavet margherita skal vi blot se hvem der har haft lavet en margherita og udskrive vores resultat.

```
SELECT distinct opr_navn FROM `lab_database2`.`operatoer` natural join  
`lab_database2`.`produktbatchkomponent` natural join  
`lab_database2`.`produktbatch` natural join `lab_database2`.`recept` WHERE  
recept_navn LIKE '%margherita%';
```

```
+-----+  
| opr_navn |  
+-----+  
| Angelo A |  
| Antonella B |  
+-----+  
2 rows in set (0.01 sec)
```

9: Vi skal bestemme relationerne for de tupler der indeholder råvare batches, status, råvare navn, netto, opskrift og operatør. Vi laver et natural join på vores tabeller og sortere det efter vores identifikationsnumre på vores råvare batch.

```
SELECT rb_id, status, raavare_navn, netto, recept_navn, opr_navn FROM
`lab_database2`.`raavare` natural join `lab_database2`.`raavarebatch` natural join
`lab_database2`.`produktbatchkomponent` natural join
`lab_database2`.`produktbatch` natural join `lab_database2`.`recept` natural join
`lab_database2`.`operatoer` WHERE rb_id LIKE '%%' ORDER BY rb_id;
```

rb_id	status	raavare_navn	netto	recept_navn	opr_navn
1	2	dej	10.05	margherita	Angelo A
1	2	dej	10.01	margherita	Antonella B
1	2	dej	10.07	prosciutto	Angelo A
1	1	dej	10.02	capricciosa	Luigi C
2	2	tomat	2.03	margherita	Angelo A
2	2	tomat	1.99	margherita	Antonella B
3	2	tomat	2.06	prosciutto	Antonella B
4	2	ost	1.98	margherita	Angelo A
4	2	ost	1.55	prosciutto	Angelo A
5	2	ost	1.47	margherita	Antonella B
5	1	ost	1.57	capricciosa	Luigi C
6	2	skinke	1.53	prosciutto	Antonella B
6	1	skinke	1.03	capricciosa	Luigi C
7	1	champignon	0.99	capricciosa	Luigi C

14 rows in set (0.02 sec)

Den ovenstående tabel viser os hvilken råvare der er brugt, hvor meget der er brugt, til hvad det er brugt og hvem der har brugt det. Statussen fortæller os desuden hvem der er i gang med at bruge råvare batchen og hvad til den bliver brugt til.

Vi har desuden blevet stillet nogle forespørgsler i SQL som skal laves og testes i MySQL.

Q1: Angiv antallet af produktbatchkomponenter med en nettovægt på mere end 10.

```
SELECT count(*) AS Above10 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 *raavare_batch*-tabellen.

```
SELECT sum(maengde) AS Total FROM raavarebatch natural join `lab_database2`.`raavare` WHERE raavare_navn = 'tomat';
```

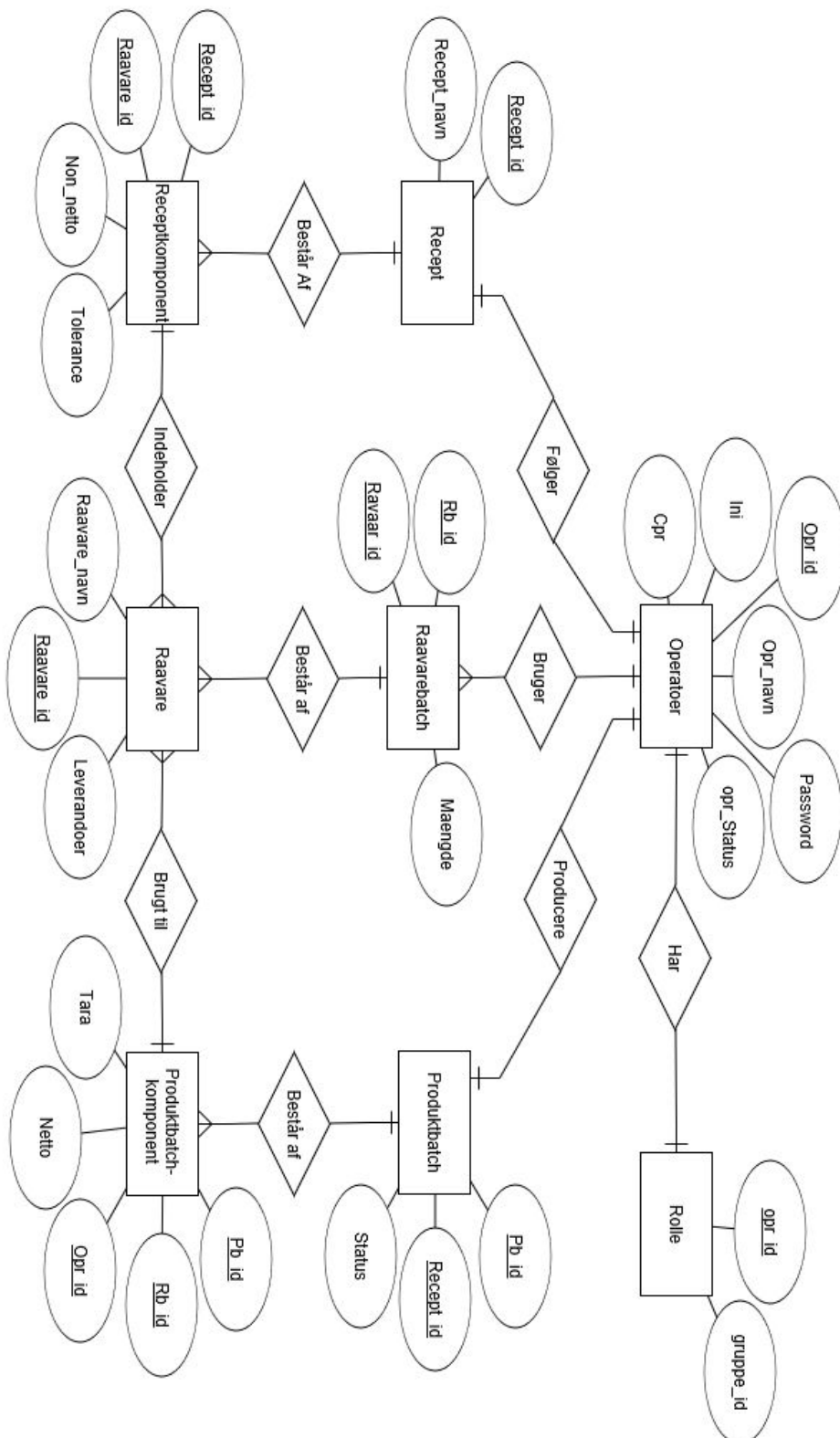
Q3: Find for hver ingrediens (råvare) den samlede mængde af denne ingrediens som findes på lageret.

```
SELECT raavare_navn, sum(maengde) AS Total 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, count(*) quantity FROM recept natural join receptkomponent natural join raavare GROUP BY raavare_navn HAVING quantity > 2;
```

Bilag - ER diagram



Bilag - Views

```
mysql> select * from operatoer_view;
+-----+-----+-----+-----+
| opr_id | opr_navn | ini | password |
+-----+-----+-----+-----+
|      1 | Angelo A | AA | lKje4fa |
|      2 | Antonella B | AB | atoJ21v |
|      3 | Luigi C | LC | jEfm5aQ |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

Operatoer_view

```
mysql> select * from recept_guide;
+-----+-----+-----+-----+
| recept_id | recept_navn | raavare_navn | nom_netto |
+-----+-----+-----+-----+
|      1 | margherita | dej | 10 |
|      1 | margherita | tomat | 2 |
|      1 | margherita | ost | 2 |
|      2 | prosciutto | dej | 10 |
|      2 | prosciutto | tomat | 2 |
|      2 | prosciutto | ost | 1.5 |
|      2 | prosciutto | skinke | 1.5 |
|      3 | capricciosa | dej | 10 |
|      3 | capricciosa | tomat | 1.5 |
|      3 | capricciosa | ost | 1.5 |
|      3 | capricciosa | skinke | 1 |
|      3 | capricciosa | champignon | 1 |
+-----+-----+-----+-----+
12 rows in set (0.00 sec)
```

Recept_guide

```
mysql> select * from lager_status;
```

raavare_id	raavare_navn	leverandoer	maengde
1	dej	Wawelka	1000
2	tomat	Knoor	300
3	tomat	Veaubais	300
5	ost	Ost og Skinke A/S	100
5	ost	Ost og Skinke A/S	100
6	skinke	Ost og Skinke A/S	100
7	champignon	Igloo Frostvarer	100

```
7 rows in set (0.00 sec)
```

Lager_status