

Database Projekt

02327 Indledende databaser og databaseprogrammering

Projekttitel: Database Projekt

Gruppe nummer: 20

Deadline:

Version: 1.0

Denne rapport indeholder 38 sider, inklusiv denne side.

Studie nr., Efternavn, Fornavn

S165248, Gadegaard, Theis



S165208, Højsgaard, Tobias André



S165209, Jønsson, Danny



S165227, Petersen, Gustav Hammershøj



S145005, von Scholten, Frederik



Signature

Arbejdsfordeling

Vi har fået delt arbejdsfordelingen nogenlunde ligeligt hvor alle har fået lov til at rode med databasen og alle har fået skrevet noget på rapporten.

Navn	Analyse	Design	Impl.	Test	Doc.	Andet	Total antal timer
Gadegaard, Theis	2		3		4		9
Højsgaard, Tobias André			12	4	6		22
Jønsson, Danny			1		4,5	2	7,5
Petersen, Gustav Hammarshøi	4	4	6		5	1	20
von Scholten, Frederik	3,5	3,5	2			1,5	10,5
Total antal timer	9,5	7,5	24	4	19,5	4,5	69

Resumé/Abstract

We have made a database project with an already designed database. In this database project have we made a thorough analysis which explains what the database is capable of and what it consists of, what kind of views we have made to it (with perspective to a different assignment in a different course), what procedures we have made to the database and out JDBC connection to the database and more. For an easy overview have we set in some diagrams representing the database structure. Other than the main project part as was just described we were also assigned with some sql query tasks which solutions can be seen in our appendix.

Indholdsfortegnelse

Arbejdsfordeling	2
Resumé/Abstract	2
Indholdsfortegnelse	3
1 Introduktion	5
2 Analyse	6
2.1 Tabeller	6
2.2 Relationer	7
2.2.1 Nøgler	7
2.2.2 E/R	8
2.2.3 Skema diagram	9
2.3 Normalisering	9
Første normalform:	10
Anden normalform:	10
Tredje normalform:	11
Analyse af vægt database tabeller	12
3 Design og implementering	14
3.1 Views	14
3.2 Procedures	16
3.3 Java Database Forbindelse (JDBC)	18
3.3.1 Forbindelse (Connection)	18
3.3.2 Data overførsel	18
3.3.3 Data adgang	18
3.3.4 Diagram	19
3.4 Implementering	20
3.4.1 Connector	20
3.4.2 SQLMapper	21
3.4.3 Data Access Objekter	22
4 Test	24
5 Konklusion	25
Bilag 1 E/R Diagrammer	26
1.1	26
1.2	26
1.3	27
Bilag 2 SQL forespørgsler	27
2.1	28
2.1.1	28
2.1.2	28

2.1.3	29
2.1.4	30
2.1.5	31
2.1.6	32
2.1.7	33
2.1.8	34
2.1.9	34
2.2	36
2.2.1	36
2.2.2	36
2.2.3	37
2.2.4	38

1 Introduktion

Vi har fået udleveret en database som allerede er opsat og som allerede havde lagret data liggende i sig. Denne database har vi skulle skrive denne rapport omkring og vi skal også bruge den i en senere cdio opgave (som er et projekt i et andet kursus). I denne rapport kommer vi til at beskrive hvordan databasen fundamentalt er sat op og analysere den ud fra dette fundament. Lidt mere dybdegående analyse af databasen har vi også fået lavet som en normalforms analyse hvor vi viser hvilke tabeller i databasen der er på hvilken form (med begrundelse) og hvad dets følger er. Vi kommer også til at beskrive hvordan vi har udviklet på databasen som views, jdbc og procedures. For at få et overblik over databasen har vi fået lagt et ER diagram og et skema diagram ind i rapporten i deres egne afsnit.

Vi har også i dette projekt fået udleveret nogle sql query opgaver som vi har fået lavet og ligger i bilaget.

2 Analyse

2.1 Tabeller

Databasen i vores system er inddelt i en række forskellige tabeller der indeholde og gruppere dataen der skal bruges i systemet. For at få et overblik over hvordan tabellerne hænger sammen kan man med fordel se på vores skemadiagram. For at få en bedre forståelse af hver af skemaernes funktion, og sammenhængen mellem dem, vil vi gå lidt mere i dybden med dem her.

Recept

Skemaet “recept” er en oversigt over alle de recepter/opskrifter som virksomheden kan bruge når de skal producere en vare. Recept indeholder ikke noget information om hvad der indgår i hver opskrift, men er snarere en oversigt over hvilke produkter virksomheden kan fremstille. En tuppel i tabellen recept er angivet ved et identifikationsnummer *recept_id* og et navn *recept_navn*.

Receptkomponent

Skemaet “receptkomponent” er en mere detaljeret oversigt over hvilke råvare der skal bruges i hver recept, og hvor meget skal bruges af hver råvare. I receptkomponent bliver der i stedet for navne på recepter eller råvare, brugt *recept_id* og *raavare_id* til at angive disse. Hvis der er brug for det kan man nemt synliggøre dem ved at lave en naturlig join fra recept eller raavare. Udover *recept_id* og *raavare_id* indeholder receptkomponent også kolonnerne *nom_netto* og *tolerance*. *nom_netto* angiver den norminelle vægt af hver råvare i recepterne. *tolerance* angiver hvor stor en procentvis afvigelse i vægten der er acceptabel.

Produktbatch

Når der skal produceres en vare i virksomheden bliver det gjort i store batches. Når virksomheden skal producere et batch af en bestemt type vare, bliver det registreret i skemaet “produktbatch”. Her kan man se hvilket recept der bliver brugt (*recept_id*) og hvad status er på hvert batch: før produktion, under produktion, afsluttet produktion. Hvert produktbatch bliver numereret med et identifikationsnummer *pb_id*.

Produktbatchkomponent

Skemaet “produktbatchkomponent” er en oversigt over hvor meget af hver råvare der er blevet brugt i hvert batch. For at finde frem til en bestemt tuppel skal man bruge et produktbatch id *pb_id* og et råvarebatch id *rb_id*. I skemaet kan man se produktkomponentets nettovægt *netto*, vægten af indpakningen *tara* og id'en for den operatør der har afvejet produktkomponentet *opr_id*.

Råvare

Skemaet “raavare” indeholder en oversigt over alle de forskellige råvare der bliver brugt i virksomheden. Hver råvare bliver angivet ved et navn *raavare_navn* og varens leverandør *leverandoer*. Hver råvare bliver angivet med et identifikationsnummer *raavare_id*. Det er nødvendigt at have et *raavare_id*, da den samme slags vare godt kan bliver leveret af flere leverandører, og én leverandør godt kan levere flere slags råvare.

Råvarebatch

Skemaet "raavarebatch" repræsenterer de råvarer virksomheden har tilgængeligt på lageret. Hver tuppel er ét batch af råvarer. Råvarebatch bruger identifikationsnumrene fra råvare *raavare_id* til at vise hvilken råvare der er tale om, og har sin egen nummerering *rb_id* for at kunne skelne mellem hvert batch. Derudover har råvarebatch en kolonne *maegde* der angiver hvor meget af råvaren der er i batchet. Mængden bliver angivet i kilogram.

Operatører

For at have adgang til systemet og kunne afveje og registrerer råvare skal man være registreret som operatør. Information om hver operatør bliver opbevaret i skemaet *operatoer*. Hver operatør bliver registreret med et unikt operatør identifikationsnummer *opr_id* som bliver brugt i produktbatchkomponent til at se hvilken operatør der har afvejnet hver råvare. I operatør skemaet kan man se operatørens navn *opr_navn*, initialer *ini*, CPR-nummer *cpr* og kodeord *password*.

2.2 Relationer

2.2.1 Nøgler

Primærnøgler er en af de vigtigste ting i en god database. En primærnøgle er en tabel-kolonne der sikre kolonne niveau tilgængelighed.

Med en god primærnøgle kan man specificere en primærnøgle værdi som gør det muligt, at query hver tabel række individuelt og ændre i hver række uden at ændre noget i alle andre rækker i samme tabel. Værdierne i primærnøgle kolonnen er unik og ingen værdier er ens.

Hver tabel skal have en primær nøgle og kun én. En concatenated primærnøgle består af to eller flere kolonner. I en enkelt tabel kan der findes flere eller grupper af kolonner der kan fungere, er kandidater, som primærnøgler - de bliver kaldt kandidatnøgler. Kun én kandidatnøgle kan vælges til at være en primærnøgle for tabellen. De kandidatnøgler der ikke vælges til at være primærnøglen bliver kaldt sekundær nøgler.

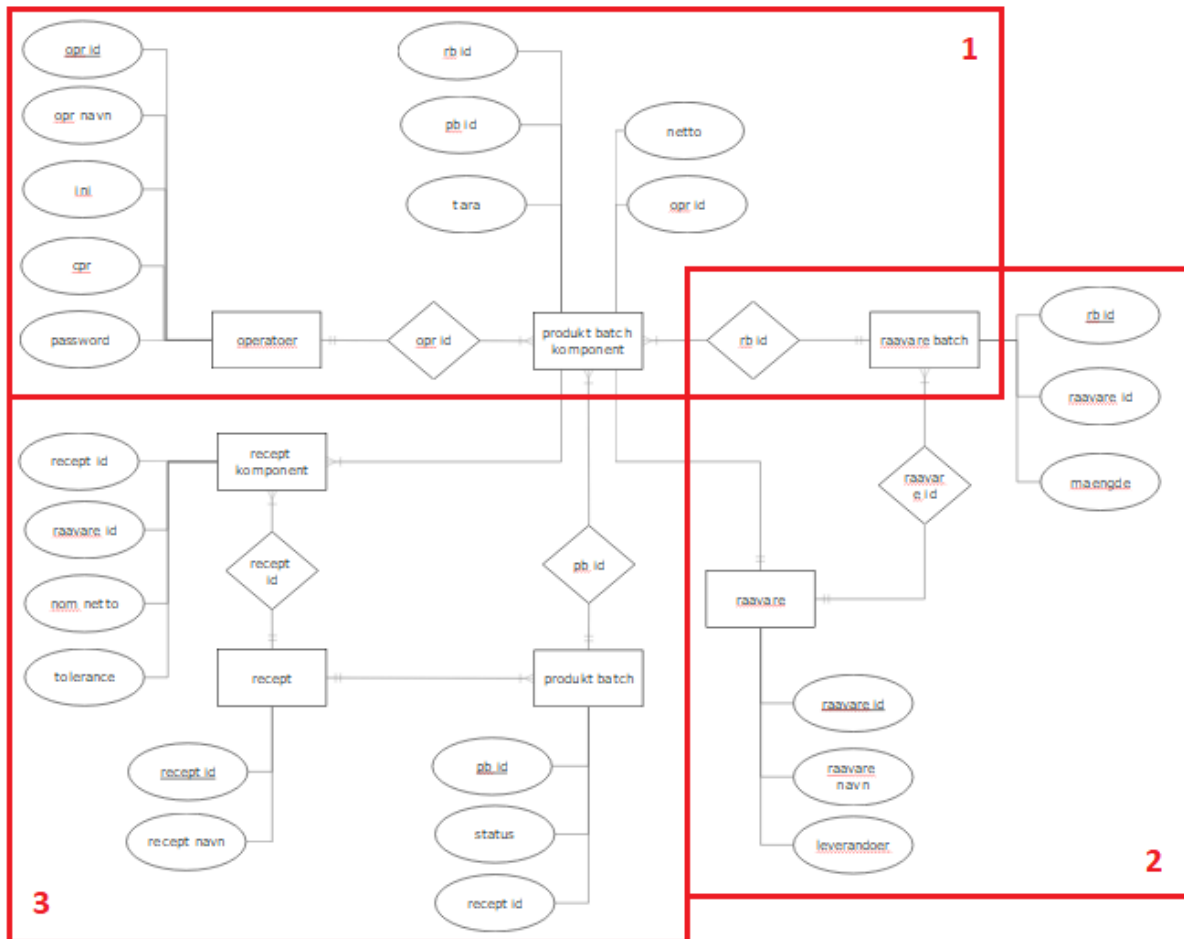
Primærnøgler må aldrig være NULL. Da null er en ukendt condition. Når en tabelværdi er null kan det betyde at værdien er ukendt, at værdien er irrelevant eller at man ikke ved om værdien er relevant.

Kandidatnøgler med specialkarakterer, blandet store og små bokstaver eller mellemrum er dårlige valg som primærnøgler. Desuden bør kandidatnøgler med blandet case værdier undgås. Da det er svært at arbejde med i queries og joint statements.

SQL server bearbejder nummer datatyper hurtigere end karakter datatyper. Derfor foretrækkes Integer datatyper af en fastsat længde. Datatyper med en fastsat længde er bedre end variable længde forde SQL servere skal dekomprimere variable længder for den kan bearbejde dataen. Når en primær nøgle værdi er valgt bør man aldrig ændre værdien. Da det vil betyde at den tilhørende fremmede nøgle også skal rettes.

Fremmede nøgler bruges til at linke to tabeller sammen. En fremmede nøgle kollision er en kombination af værdier ens med primærnøglen i den tabel som tabellen er linket til. Relationen er altså at primærnøglen i den ene tabel matcher med fremmede nøglen i den anden tabel.

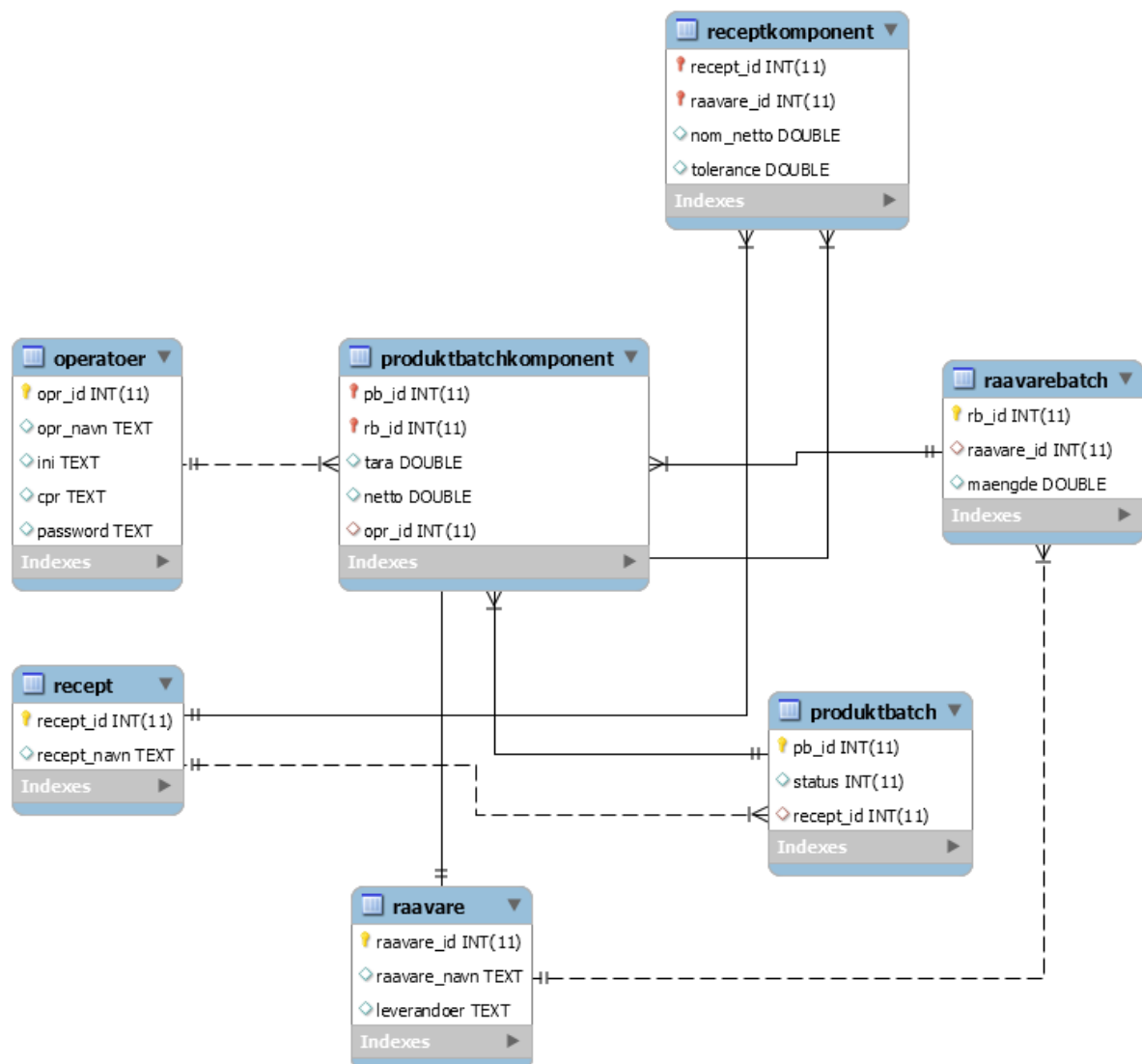
2.2.2 E/R



E/R diagram: Detaljerede udklip (hhv. område 1, 2 og 3) af diagrammet kan findes i bilag 1.1, 1.2 og 1.3. Her er det også nemmere at læse kardinalitet.

E/R diagrammet viser entity relationerne i databasen. Hver entity (tabel) symboliseret ved rektangler har nogle attributter (ellipse) hvor understreget attributter er primærnøgler. Relationerne ved frem nøgler mellem entities er repræsenteret i romberne. Kardinalitet er noteret ved crow-foot notation. f.eks. har én operatør (opr_id) en eller flere produkt batch altså en en-mange relation.

2.2.3 Skema diagram



På det ovenstående diagram ses de forskellige datatyper og nøgler som udgør databasen, inklusiv fremmednøgler. Vi kan se at alle tabeller bortset fra receptkomponent og produktbatchkomponent har en primærnøgle bestående af et enkelt heltal. Disse to andre tabeller har primærnøgler bestående af fremmednøgler fra de øvrige tabeller. Cardinality er vist ved crow-foot notation.

2.3 Normalisering

Normalisering af databaser er nødvendigt for at gøre databasen lettere at arbejde med og for at forhindre at man bliver nødt til at omstrukturere datalaget hvis applikationslaget ændrer sig.

Normalisering finder sted i forskellige trin, kaldet normalformer, som hver har deres egne kriterier for hvordan databasen skal se ud. Normalformerne er hierarkiske, og kræver derfor at den tidligere normalform er blevet opfyldt før den næste normalforms kriterier kan blive mødt.

I de fleste tilfælde bestræber vi os efter at opnå 3. normalform, da højere normalformer ofte gør det tilpas besværligt at lave datalaget til at den fordel man får med hensyn til applikationslagets og funktionslagets design ikke længere opvejer den mængde arbejde man skal lægge i designet af datalaget.

Kriterierne for alle normalformerne op til tredje normalform er som følger.

Første normalform:

- Alle værdier skal være atomare
- Der må ikke være to ens rækker i tabellen

Hvad dette vil sige i praksis er at der ikke må være flere forskellige værdier i et felt. Man må for eksempel ikke have en kolonne der hedder "fag" og have de værdier der indsættes her være "dansk,engelsk,matematik". Dette skaber unødvendig kompleksitet på de øvrige lag af systemet. I stedet for kan man lave en ny tabel der hedder fag og have en række med selve faget og en anden værdi som repræsenterer den person som faget hører til.

En sådan værdi kunne være en primær nøgle. En primær nøgle er en kandidatnøgle som udvælges til at entydigt identificere en enkelt række i tabellen sådan at de alle er forskellige. Hvis man ikke har en primær nøgle risikerer man at to rækker i tabellen kan være fuldstændig ens, hvilket man ikke vil have i nogle systemer. Det kan for eksempel være at man laver et system der behandler finansielle transaktioner, og her kan man godt forestille sig at en enkelt person laver en transaktion for den samme genstand for samme beløb flere gange. I dette tilfælde har man brug for en entydig værdi til at bestemme hvilken transaktion vi har med at gøre. Dette kunne tage formen af et heltals felt kaldet transaktions-id.

Anden normalform:

- Tabellen skal være på første normalform
- Der må ikke være nogen attributter som kun afhænger af en del af en kandidatnøgle.

Hvad dette andet kriterium vil sige er at den nøgle vi bruger til entydigt at bestemme rækker i vores tabel enten kun er et enkelt felt (som for eksempel heltallet transaktions-id) eller det er en nøgle sammensat af flere forskellige attributter, men i dette tilfælde skal alle attributter som findes i nøglen bruges til at finde hvert felt som ikke er en del af nøglen.

Hvis vi bruger en universitets database som eksempel kunne vi forestille følgende relation:

kursus_nr	semester	antal_pladser	kursus_navn
1	2016-F	100	avanceret bagning
2	2016-F	150	kagekryptering

1	2016-E	135	avanceret bagning
2	2016-E	175	kagekryptering
1	2017-F	125	avanceret bagning
2	2017-F	180	kagekryptering

I dette tilfælde har vi en kandidatnøgle som består af kursus_nr, semester og antal_pladser som skal bruges til entydigt at bestemme en række. Problemet er at kursus_navn sagtens kan bestemmes ved hjælp af kursus_nr og slet ikke har noget at gøre med semesteret eller antallet af pladser. Altså afhænger denne attribut kun af en del af kandidatnøglen og opfylder derfor ikke 2NF.

Hvad man kunne gøre i stedet for ville være at lave en separat tabel med relationen mellem kursus_navn og kursus_nr.

kursus_nr	semester	antal_pladser
1	2016-F	100
2	2016-F	150
1	2016-E	135
2	2016-E	175
1	2017-F	125
2	2017-F	180

kursus_nr	kursus_navn
1	avanceret bagning
2	kagekryptering

Dette ville opfylde 2NF fordi den øverste tabel nu afhænger af hele kandidatnøglen og den nederste tabel ikke har en sammensat kandidatnøgle.

Tredje normalform:

- Tabellen skal være på anden normalform
- Enhver attribut i tabellen som ikke er en del af primærnøglen må ikke være transitivt afhængig af nogen af de andre nøgler i tabellen.

At attributterne ikke må være transitivt afhængige af nogen nøgler i tabellen vil sige at vi ikke må have en attribut i tabellen som har en anden nøgle i tabellen end primærnøglen som kan bruges til at bestemme den.

Lad os se på et eksempel:

kunstner	scene	by	pris
Hansi H.	Futterkisten	Hamsterdam	500
Bamse	Futterkisten	Hamsterdam	250
Bruno M.	U-Hallen	Hamsterdam	600

Den ovenstående tabel er anden normalform fordi den har en sammensat kandidatnøgle i form af kunstner, scene og by og den eneste attribut som ikke er en del af primærnøglen er prisen, som kun kan bestemmes ud fra hele den sammensatte kandidatnøgle.

Det som gør at den ovenstående tabel ikke opfylder tredje normalform er relationen mellem scene og by. Fordi en given scene altid er i samme by vil vi have en masse information som bliver gentaget hele tiden.

Scenen kan bruges til at bestemme hvilken by vi er i, og derfor har vi ikke brug for også at have information om byen i koncert tabellen ovenfor.

Løsningen ville være at lave en separat tabel som indeholder relationen mellem by og scene.

kunstner	scene	pris
Hansi H.	Futterkisten	500
Bamse	Futterkisten	250
Bruno M.	U-Hallen	600

scene	by
Futterkisten	Hamsterdam
U-Hallen	Hamsterdam

Analyse af vægt database tabeller

Nu hvor vi ved hvad kriterierne er for alle de forskellige normalformer op til 3. normalform kan vi begynde at analysere på vores database om de individuelle tabeller opfylder dem.

Til dette formål har vi lavet en tabel for at give et overblik over hvilke tabeller der opfylder kriterierne, og hvilke der ikke gør.

Tabel	1NF	2NF	3NF
operatoer	Y	Y	Y
produktbatch	Y	Y	Y
produktbatchkomponent	Y	Y	Y
recept	Y	Y	Y
receptkomponent	Y	Y	Y
raavare	Y	Y	Y
raavarebatch	Y	Y	Y

Alle vores tabeller er atomare. Der er ikke nogen felter hvor vi har flere forskellige informationer bundet sammen.

Vi har primærnøgler til alle tabeller. Nogle af disse primærnøgler er udvalgt fra sammensatte kandidatnøgler, som i produktbatchkomponent og receptkomponent, men i begge tilfælde er der ikke af nogen af de attributter i disse tabeller som kan findes med kun en del af den sammensatte kandidatnøgle. Man kan ikke entydigt bestemme en række hvis man kun har en af de to id'er som indgår i hver tabels primærnøgle, fordi de attributter som er i disse tabeller er relevante for begge de tabeller som de har fremmednøgler fra.

For eksempel er tara og netto ikke noget man entydigt kan bestemme ud fra pb_id i produktbatchkomponent tabellen, fordi vi sagtens kunne afveje samme netto af forskellige råvarer til samme produkt. Og på samme måde ville vi også kunne afveje den samme netto af samme råvare, men i forskellige produkt sammenhænge. Ergo siger hele den sammensatte kandidatnøgle noget om tabellens attributter, og man kan ikke bare tage en del af den og få en relation.

Alle tabeller opfylder altså uden tvivl 2. normalform.

Da der heller ikke er nogen åbenlyse transitive afhængigheder mellem attributterne i tabellerne vurderer vi også at de er på 3. normalform.

3 Design og implementering

3.1 Views

Views er midlertidige “tabeller”, som ikke indeholder nogen data, men som er meget nyttigt i tilfælde af, at nogen skal se noget i en database, eller bruge noget fra databasen, uden at de skal skrive ind en lang kommando, for at få det frem som brugeren har brug for.

Da views bare viser, hvordan dataen man har i sin database ligger (igen, views i sig selv indeholder ikke nogen data), behøves man ikke være bange for, at en bruger der ser gennem et view, ved et uheld kommer ind og roder med databasen, eller får set et uforståeligt dataset.

Man kan også bruge views til at opdele sine brugere, og begrænse hvad forskellige brugere kan se af databasen. Hvis man har en medarbejder fra et supermarkedet, vil man f.eks. ikke have, at medarbejderen skulle kunne se alle andre medarbejders brugerinformationer, men måske hellere at medarbejderen kan se, hvad hvilke vare koster i supermarkedet.

I vores cdio projekt skal vi eventuelt bruge denne database, og muligvis tillade nogle brugere (som er opdelt i roller), at se forskellig information fra databasen. Dette er en oplagt mulighed for at bruge views, da de forskellige brugerroller ikke burde kunne se noget information, andet end den information den specifikke bruger skal bruge.

De forskellige brugerroller i projektet er: Admin, Pharmacist, Laborant, Produktionsleder. Som det står til i cdio-3, som er den aflevering vi er nået til, er der dog ikke givet nogen specifikke krav til, hvad brugerne skal kunne se, andet end at Admin skal kunne se alle brugeres informationer. Vi har dog fået lavet nogen views, som måske godt ville kunne bruges af nogle af rollerne. Da der ikke er nogen specifikke rolle funktionskrav, ud over et krav for admin rollen (angående at kunne se information, og redigere data), så har vi ikke kunne lave nogle rolle-dedikerede views, men har fået lavet 2 potentielt nyttige views, som evt. Admin rollen, eller nogle af rollerne som håndtere råvarene eller vægt informationerne fra databasen kunne bruge.

Kode:

```
CREATE VIEW mad
AS
SELECT rk.recept_id, rk.raavare_id, re.recept_navn, ra.raavare_navn,
ra.leverandoer
FROM receptkomponent rk
      NATURAL JOIN recept re
      NATURAL JOIN raavare ra;
```

Resultat eksempel:

recept_id	raavare_id	recept_navn	raavare_navn	leverandoer
1	1	margherita	dej	Wawelka
1	2	margherita	tomat	Knor
1	5	margherita	ost	Ost og Skinke A/S

Query for vores første view: mad samt resultat ved eksekvering

Det første view vi har lavet kaldet “mad”, er et view som giver den person, som har kaldet viewet, et overblik over de forskellige madvare, madvarens id nummer, receptens id nummer, hvilken ret råvarene skal bruges til (recept navn) og fra hvilken leverandør råvarene kommer fra. Dette view er godt til at få et overblik over, hvilke madvare der er til rådighed, hvad de skal bruges til, og hvor det er madvarerne kommer fra.

Kode:

```
CREATE VIEW vejning
AS
SELECT opr.opr_id, opr.opr_navn, pbk.tara, pbk.netto, ra.raavare_id,
ra.raavare_navn, rab.maengde
FROM operatoer opr
      NATURAL JOIN produktbatchkomponent pbk
      NATURAL JOIN raavare ra
      NATURAL JOIN raavarebatch rab;
```

Resultat eksempel:

opr_id	opr_navn	tara	netto	raavare_id	raavare_navn	maengde
1	Angelo A	0.5	10.05	1	dej	1000
1	Angelo A	0.5	10.07	1	dej	1000
1	Angelo A	0.5	2.03	2	tomat	300

Query for vores andet view: vejning, samt resultat ved eksekvering

Det andet view vi har lavet kaldet “vejning”, er et view som giver et overblik over alle de personer, der har brugt vægten, hvad det er de har vejet (råvarens navn), deres resultater af vejningen, mængden af det de har vejet, og mængden af hvad der er blevet tareret fra vejningen. På dette view kan man også se råvare id nummer, på det produkt der er blevet vejet. Hvis man føler man har brug for mere specifik information omkring varen, kan man

evt. bruge "mad" viewet til at finde denne information, da råvare id nummeret også kan ses på det view.

Man kan diskutere, om der burde laves et view til administratoren dedikeret til at vise brugerinformation, men da al brugerinformation er lagret i sin egen tabel ville det være lige så hurtigt bare at kalde alle elementer i tabellen, som det ville være at kalde et view af tabellen. Et argument for at lave et view ville være, at administratoren ikke ville komme til at pille ved databasen ved et uheld, men da det er en administrator kommando, vil det måske heller ikke give mening, at holde administratoren fra at kunne pille ved dataen i databasen.

Selve vores views er blevet implementeret i vores program, således at der er en metode pr. view som sætter viewet op i ens lokale database, der er en metode pr. view som dropper viewet og der er også en metode pr. view, hvor man kan hente en bestemt række fra viewet eller hente hele tabellen, som viewet er lavet til at lave.

3.2 Procedures

En procedure, også ofte kaldet en 'Stored Procedure' på engelsk, ligner på mange måder en metode/funktion som man kender det fra andre programmeringssprog, som har til formål at simplificere og effektivisere databasen. Den indeholder en række SQL statements der bliver gemt i en RDBMS (Relational Database Management System), således at den kan blive genbrugt og benyttet af andre programmer. Sådan en procedure har adgang til at hvide data ud mens den samtidig har mulighed for at modificere data i databasen.

Herudover fungerer en procedurer som et user interface mellem bruger og databasen, der giver en forstærket sikkerhed ved modificering af databasen ved kun at give tilladelse til at bruge proceduren og ikke tilgå data i tabellen. Ydermere giver det en kæmpe aflastning på netværkstrafikken, at man blot kalder proceduren, og ikke de adskillige linier kode som proceduren ellers indeholder.

En simpel procedure har strukturen som nedenstående:

```
CREATE PROCEDURE [procedure-navn] ([IN/OUT/INOUT] [Parameter]
[Type])
BEGIN
* Kode *
END
```

- Proceduren dannes og navngives efter eget valg.
- Herefter bestemmes det, om parameteren skal være af typen IN / OUT / INOUT
 - o IN: Man sender en parameter til proceduren
 - o OUT: Ved en OUT parameter tildeles værdien der modtages til den respektive variable.
 - o INOUT: Man sender og modtager værdier til proceduren med den samme variable.
- Parameter er navnet på variablen
- Type definerer typen af variablen, f.eks. INT, varchar, m.m.

Lad os antage at vi adskillige gange skal ind og finde alle recepter i databasen som ikke indeholder en specifik råvare (Der tages udgangspunkt i Bilag 2 SQL forespørgsler 2.1.4). At skulle skrive de mange linier SQL statements igen og igen øger risikoen for fejl og det er her derfor oplagt at lave en procedure der tager imod en parameter, som er den råvare man helst gerne vil undgå at se. Vi tager derfor de SQL statements og sætter dem ind i en procedure, tilføjer en parameter, som så er den råvare vi ikke ønsker at have med i recepten. i Cmd prompt vil det kunne skrives som nedenstående:

```
delimiter //
CREATE PROCEDURE ReceptUdenRaavare (IN raavare
varchar(20))
BEGIN
SELECT recept.recept_navn as "Recept Navn" FROM recept
WHERE recept.recept_id NOT IN
      (SELECT receptkomponent.recept_id FROM
receptkomponent
WHERE receptkomponent.raavare_id IN
      (SELECT raavare.raavare_id FROM raavare
WHERE raavare.raavare_navn = raavare)
);
END //

delimiter ;
```

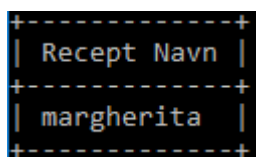
Bemærk de tre nedenstående linier:

- delimiter //
- END //
- delimiter ;

Delimiter benyttes til at forhindre SQL statements i at blive udført af SQL før den bliver sendt videre til serveren. Når den er sendt ændrer vi delimiteren til semicolon igen.

Nu hvor proceduren er oprettet kan vi kalde den vha. kommandoen CALL:

```
CALL ReceptUdenRaavare('skinke');
```



Recept Navn
margherita

Vi har hermed formindsket trafikken på netværket fra adskillige linier til blot et enkelt og sparet databasen fra at skulle compile de mange linier adskillige gange, da den allerede har compilet proceduren.

3.3 Java Database Forbindelse (JDBC)

3.3.1 Forbindelse (Connection)

Vi laver Java Database Connectivity ved at hive en mysql driver ind i vores projekt som en ekstern jar fil. Vi laver derefter en Connector klasse, som bruger `Class.forName()` metoden i sin constructor til at registrere en ny instans af denne mysql driver sådan at klassen kan bruge de statiske metoder fra driveren.

De relevante metoder er

- `connectToDatabase()`
- `createStatement()`
- `executeQuery()`
- `executeUpdate()`

`connectToDatase` lader os, som navnet siger, oprette en forbindelse til en database.

`createStatement()` bruges til at instantiere en `Statement` klasse, som kan indeholde en forespørgsel, levere denne til databasen og modtage resultatet og en given forbindelse.

`execute` kommandoerne udfører forespørgslen som er blevet lavet. Afhængigt af om det er en `SELECT` kommando eller en `INSERT/UPDATE` kommando ville man bruge henholdsvis `executeQuery` eller `executeUpdate`.

3.3.2 Data overførsel

Når vi skal arbejde med databasen er det bedst hvis vi sender et enkelt objekt til hver forespørgsel og at der er en mængde information som er så lille at der ikke vil være store gentagelser.

Derfor er den simpleste løsning at lave et dataoverførselsobjekt til hver eneste tabel og view i databasen, som vi gerne vil arbejde med, med variable til at gemme hver eneste attribut i en given række.

Disse data overførselsobjekter skal have get- og set-metoder til hver enkel variabel og variablene selv skal være private.

3.3.3 Data adgang

Når vi skal lave de klasser som laver forespørgsler til databasen er vi igen nødt til at lave data access klasser til hver enkel tabel og view i vores database som vi har brug for at arbejde med.

Vi bruger indirection pattern, hvilket vil sige at vi indsætter en interface for hver data adgangs klasse sådan at vores data adgangs lag er adskilt fra vores logiklag og præsentationslag.

En anden fordel ved interfaces er at vi kan starte med at hardkode vores SQL sætninger for at have en løsning der fungerer og senere udskifte dem med en løsning som indlæser SQL sætninger fra en fil relativt nemt, fordi det eneste vi er nødt til at gøre er at udskifte navnet af den data access klasse vi bruger når vi instantierer klassen ud fra interfacet.

Alle interfaces skal have metoder der

- Finder en unik række i tabellen og returnerer det data transfer object der hører til tabellen. Denne funktion tager primærnøglen som parameter.
- Returnerer samtlige rækker i tabellen.
- Opdaterer tabellen med et givent DTO til den pågældende tabel
- Indsætter en række i tabellen med et givent, tilhørende, DTO.

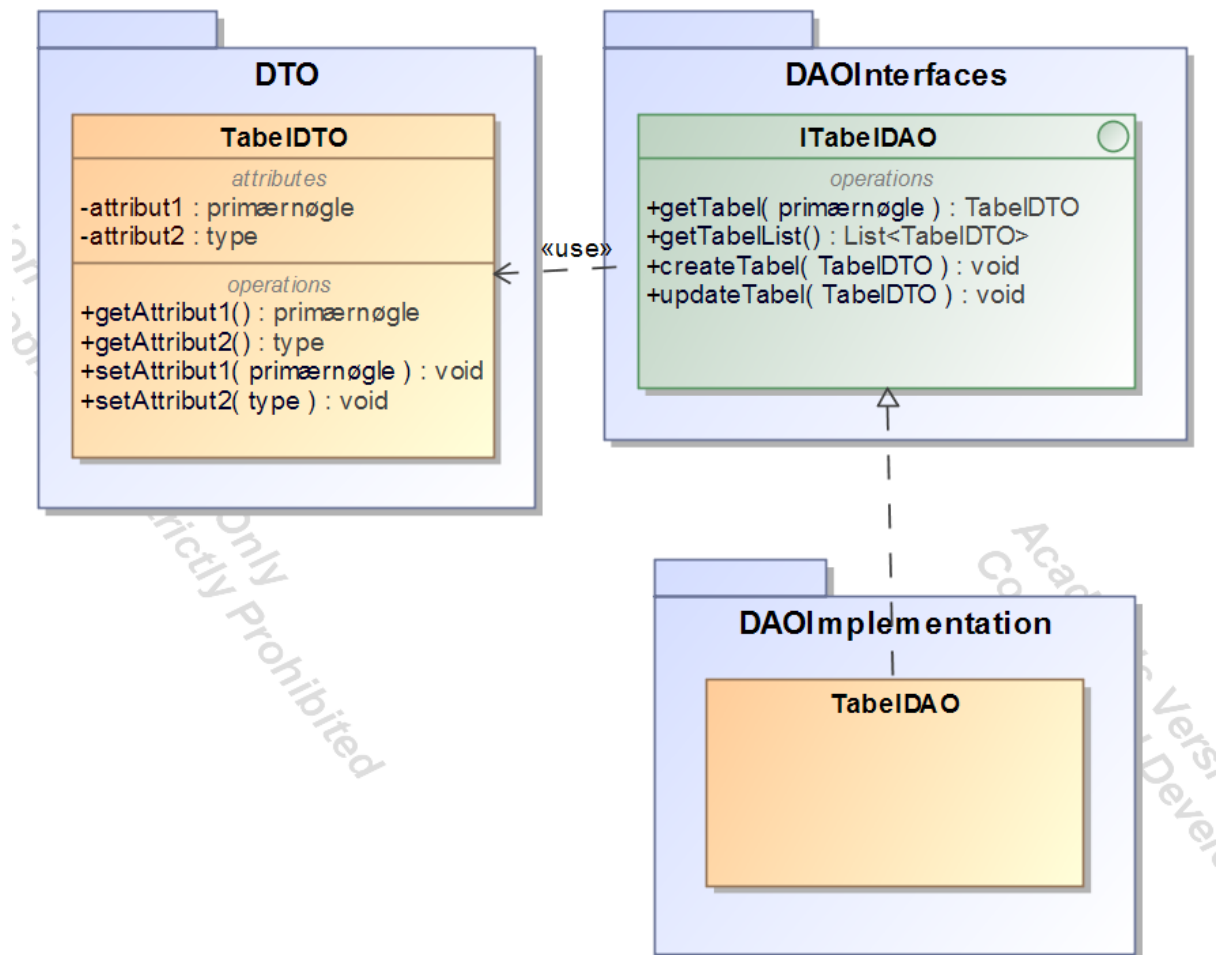
3.3.4 Diagram

Vi har lavet et generaliseret diagram der viser princippet i vores klasser, da de alle minder ret meget om hinanden.

Altså har vi vores DTO klasser i en pakke som bruges til dataoverførsel mellem vores data access lag og vores øvrige lag i systemet. De er navngivet efter deres tabel.

Vi har en interface for hver tabel, som benytter sig af det DTO der passer til den pågældende tabel. Disse interfaces har metoder som lader dem indsætte og opdatere tabellen og hente enten alle tabellens rækker eller en enkelt specifik række. Sidstnævnte metode tager primærnøglen som parameter. Hvis primærnøglen er sammensat tager metoden bare alle de forskellige attributter som indgår i primærnøglen som parametre.

Til sidst har vi så selve implementationen af hver enkelt interface. Man kan gøre det ved at hardkode SQL sætningerne ind i programmet, men dette er ikke idéelt da opdateringer af SQL sætningerne bliver meget sværere. Det er bedre at implementere dem sådan at SQL sætningerne bliver læst ind fra en fil, men dette kan være problematisk at lave fordi man skal have de værdier som DTO'ernes variable indeholder ind i SQL sætningen, så man bliver nødt til at dele SQL sætningen op mens man læser den ind, hvilket også er udfordrende.



3.4 Implementering

Vi har lavet klasser som for hver eneste af vores tabel-interfaces implementerer de metoder som er defineret i interfacen. Vi har benyttet os af det projekt skelet som var blevet gjort tilgængeligt ved projektets begyndelse. Dette inkluderer Connector klassen og dennes metoder, som forbinder til databasen på den måde som vi beskrev i afsnit "3.3.1 Forbindelse".

3.4.1 Connector

Connector bruger som sagt metoden `Class.forName()`, sådan at vi opretter en ny instans af den MySQL driver der ellers ligger som et eksternt bibliotek. På den måde kan vi bruge de statiske metoder fra MySQL driveren i Connector.

Når vi har disse statiske metoder kan vi oprette en forbindelse til vores database

```

public Connector(String server, int port, String database,
    String username, String password)
    throws InstantiationException, IllegalAccessException,
    ClassNotFoundException, SQLException
{
    conn    = connectToDatabase("jdbc:mysql://" + server + ":" + port + "/" + database,
        username, password);
    stm     = conn.createStatement();
}

```

Som det ovenstående eksempel viser kan vi lave forbindelsen til databasen ved hjælp af connectToDatabase metoden fra MySQL driveren. Parametrene er databasens adresse hvilket svarer til en sti hen til den rette port på serveren samt databasens navn. Derefter har vi så et brugernavn og password. Alle disse parametre er defineret i en anden klasse kaldet Constant. Vi kører den ovenstående metode i vores constructor for Connector og indsætter i kaldet værdierne i Constant så vi får den rigtige sti.

Det er VIGTIGT at ændre Constant så alle værdierne passer til det setup man har, ellers vil forbindelsen ikke blive oprettet.

Når først vi har en forbindelse kan vi sende forespørgsler til databasen. Dette gøres i metoderne doQuery(cmd) og doUpdate(cmd).

```

public static ResultSet doQuery(String cmd) throws DAException
{
    try { return stm.executeQuery(cmd); }
    catch (SQLException e) { throw new DAException(e); }
}

```

Dette eksempel er for doQuery(), men doUpdate() er næsten præcis det samme, bare med executeUpdate i stedet for executeQuery.

3.4.2 SQLMapper

Vi vil gerne undgå at hardkode vores SQL sætninger ind i vores data access objekter, da dette kan lede til vanskeligheder når vi senere skal opdatere systemet. Hvad vi gør er at lave en klasse, som kan hente SQL sætninger ind i en fil og også stå for at indsætte vores data transfer objekters værdier ind i disse sætninger.

Denne rolle bliver udfyldt af SQLMapper. Klassen har to metoder. getStatement() og insertValuesIntoString().

Den første metode ses her:

```

public static String getStatement(int i){

    Properties props = new Properties();
    try {
        File file = new File("SQL.txt");
        FileInputStream in = new FileInputStream(file);
        props.load(in);
        String res = props.getProperty(Integer.toString(i));
        in.close();
        return res;
    } catch (IOException e) {
        throw new IllegalStateException("Unable to load properties");
    }
}

```

Vi benytter os af properties. Hvad properties gør er at lade os definere en værdi i en tekstfil, og så kan vi tilgå den linje tekst baseret på den værdi vi har defineret. For eksempel kan vi skrive at opr_SELECT er lige med en forespørgsel der vælger alle kolonner fra operatoer og kun tager den række som har en bestemt opr_id.

```
opr_SELECT = SELECT * FROM operatoer WHERE opr_id = ?;
```

Vi kan så sige getStatement("opr_SELECT") og dette vil returnere forespørgslen.

Vores anden metode insertValuesIntoString tager en string og et string array som parametre. For hver enkelt forekomst af et spørgsmålstegn tager metoden det næste element i listen og indsætter dette i string'en. På den måde kan vi indsætte vores variable værdier i DTO'erne i vores forespørgsel.

```

public static String insertValuesIntoString(String statement, String[] values){
    for(int i=0; i<values.length; i++){
        statement = statement.replaceFirst("[?]", values[i]);
    }
    return statement;
}

```

Det er vigtigt at værdierne i string arrayet er i samme rækkefølge som man forventer de skal indsættes i, ellers får man indsat værdierne forkert hvilket kan give fejl i SQL forespørgslen. Men det er kun rækkefølgen som man behøver at bekymre sig om når man opdaterer i stedet for at man skal sætte nogen til at lave hele SQL forespørgslen.

3.4.3 Data Access Objekter

Data Access Objekterne bruger altså SQLMapperen til at lave forespørgslen og bruger derefter de metoder der er i Connector til at udføre denne forespørgsel.

```

@Override
public OperatoerDTO getOperatoer(int oprId) throws DALException {

    String statement = SQLMapper.getStatement("opr_SELECT");
    String[] values = new String[]{Integer.toString(oprId)};
    statement = SQLMapper.insertValuesIntoString(statement, values);
    System.out.println("Query: "+statement);
    ResultSet rs = Connector.doQuery(statement);

    try {
        if (!rs.first()) throw new DALException("Operatoeren " + oprId + " findes ikke");
        return new OperatoerDTO (rs.getInt("opr_id"), rs.getString("opr_navn"),
            rs.getString("ini"), rs.getString("cpr"), rs.getString("password"));
    }
    catch (SQLException e) {throw new DALException(e); }

}

```

Ovenover ses et eksempel på en af de metoder der er blevet implementeret. Princippet i de andre metoder er det samme, også for de andre data access objekter. Vi henter en forespørgsel fra filen SQL.txt ved hjælp af SQLMapper klassen, indsætter de værdier der skal indsættes i denne forespørgsel ved at køre insertValuesIntoString og bruger til sidst Connectorens doQuery metode til at sende forespørgslen til databasen og modtage svaret i form af et Resultset.

Vi kan derefter bruge .first() til at se om der er nogen rækker i Resultsettet. Når vi forventer at få flere værdier bruger vi i stedet .next() sådan at vi bliver ved med at bearbejde værdier så længe der er værdier at tage af fra Resultsettet.

I den ovenstående metode indsætter vi værdierne fra alle de forskellige kolonner ind i et data transfer objekt for operatoer tabellen og kan så returnere denne ene operatør.

4 Test

I vores projekt har vi en package test01917 med en mainklasse, som vi bruger til at teste vores implementation af data access object. Vi har implementeret en DAO-klasse til hver af tabellerne i databasen, og hver af klasserne bliver testet i en metode i mainklassen. Da klasserne hovedsageligt består af getter/setter metoder og hver DAO-klasse bliver testet næsten ens, vil vi kun gennemgå en enkelt af dem her. Hvis man har lyst til at køre en af de resterende test kan man gå ind i projektet i Java Resources/ src/ test01917 og afprøve dem enkeltvis i main klassen.

Vi har lavet en testmetode testRec() der skal teste om metoderne i MySQLReceptDAO virker efter hensigten.

Test case	testRec()
Beskrivelse	Tester om metoderne getRecept(), getReceptList(), createRecept() og updateRecept() i MySQLReceptDAO virker efter hensigten
Forudsætninger	<ol style="list-style-type: none">1. En mySQL database med det indhold vi er blevet givet skal køre2. Navnet på databasen skal matche navnet der står i klassen "Constant"
Testprocedure	<ol style="list-style-type: none">1. Vi udskriver de returnerede værdier af getRecept() og getReceptList() i konsollen2. Vi indsætter en ny recept ind på en ubrugt række i tabellen med createRecept()3. Vi ændre navnet på den ny recept ved brug af updateRecept4. Vi udskriver igen med getRecept() for at se for at se om det nye recept er blevet oprettet og ændret
Test data	<ol style="list-style-type: none">1. getRecept() finder recept_12. createRecept indsætter et nyt recept med recept_id = 4 og receptnavn = hawaii3. updateRecept() ændre recept_navn for recept_id 4 til at være ananas_pizza
Forventet resultat	<ol style="list-style-type: none">1. getRecept() returnere 1 margherita2. getReceptList() returnere 1 margherita, 2 prosciutto, 3 capricciosa3. createRecept() sætter en ny række ind med værdierne 4, hawaii4. updateRecept() ændre navnet for recept_id 4 til at være ananas pizza5. en ny getRecept(4) returnere 4 ananas_pizza
Result	De reelle resultater er de samme som de forventede
Status	Bestået
Testmiljø	Eclipse Neon 4.6.1

5 Konklusion

Alle vores tabeller er atomare. Der er ikke nogen felter hvor vi har flere forskellige informationer bundet sammen. Alle tabeller opfylder altså uden tvivl 2. normalform. Desuden vurderer vi også at alle tabellerne er på 3. normalform, da der heller ikke er nogen åbenlyse transitive afhængigheder mellem attributterne.

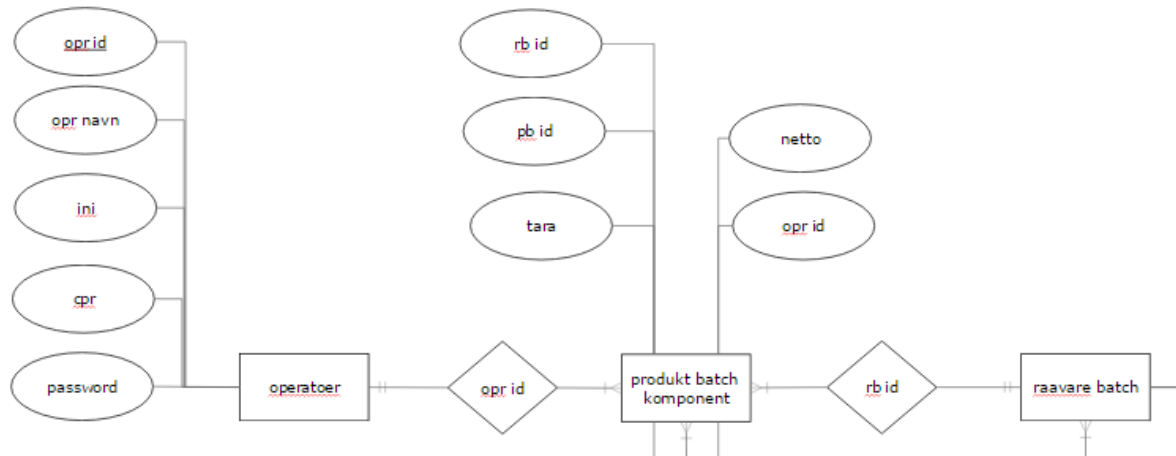
Endvidere har vi lavet klasser som for hver eneste af vores tabel-interfaces implementerer de metoder som er defineret i interfacen. Klasserne kan testes og virker efter hensigten. Til sidst har vi lavet følgende analyser og design-modeller af tabeller, relationer (E/R og skema-diagram) samt normalform.

Det er lykkedes at løse alle opgaver i opgavens del 1 - som findes i bilag.

Bilag 1 E/R Diagrammer

1.1

Detaljeret E/R diagram område 1.



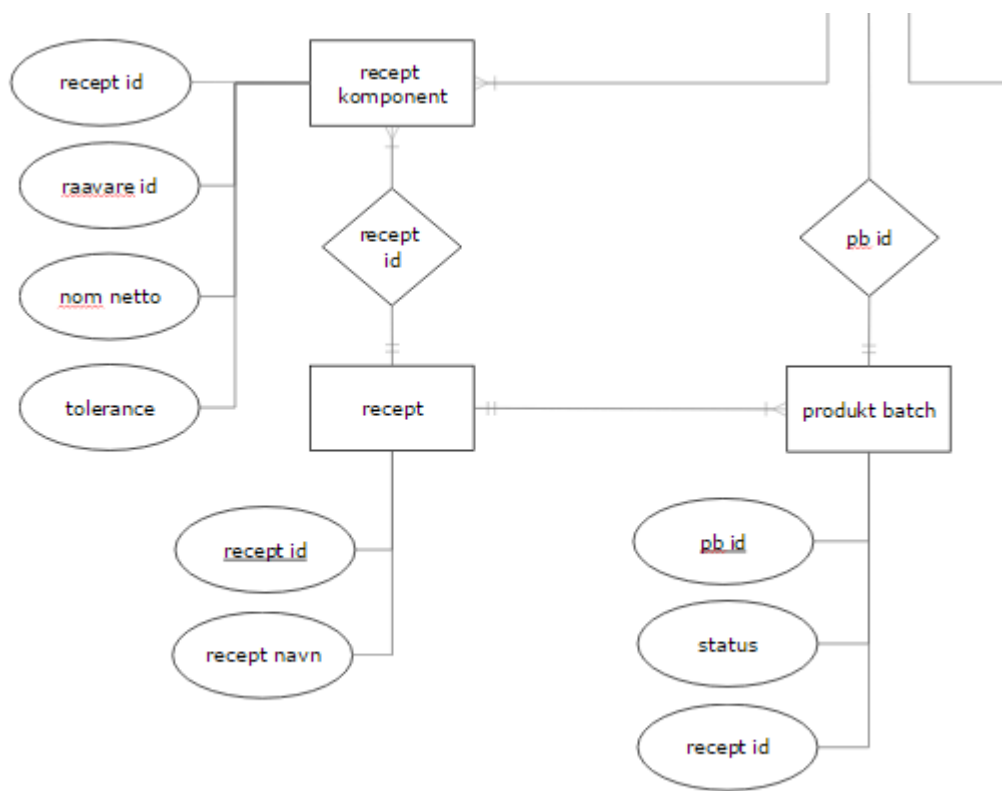
1.2

Detaljeret E/R diagram område 2.



1.3

Detaljeret E/R diagram område 3.



Bilag 2 SQL forespørgsler

2.1

2.1.1

Vi skal bestemme navnene på de råvarer som indgår i mindst to forskellige råvarebatches.

```
SELECT
raavare.raavare_navn,
raavarebatch.raavare_id
FROM raavare
JOIN
(
SELECT raavare_id, count(*)
FROM raavarebatch
GROUP BY raavare_id
HAVING count(*) >= 2
)
raavarebatch
ON raavare.raavare_id = raavarebatch.raavare_id
;
```

raavare_navn	raavare_id
ost	5

2.1.2

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

```
SELECT recept.recept_id,
recept.recept_navn,
raavare.raavare_navn
FROM receptkomponent
LEFT JOIN recept
    ON receptkomponent.recept_id = recept.recept_id
LEFT JOIN raavare
    ON receptkomponent.raavare_id = raavare.raavare_id;
```

recept_id	recept_navn	raavare_navn
1 [->]	margherita	dej
1 [->]	margherita	tomat
1 [->]	margherita	ost
2 [->]	prosciutto	dej
2 [->]	prosciutto	tomat
2 [->]	prosciutto	ost
2 [->]	prosciutto	skinke
3 [->]	capricciosa	dej
3 [->]	capricciosa	tomat
3 [->]	capricciosa	ost
3 [->]	capricciosa	skinke
3 [->]	capricciosa	champignon

2.1.3

Vi prøver at finde de recepter som indeholder mindst én af råvarerne skinke eller champignon. Vi skal også finde de recepter som indeholder begge.

```
SELECT recept.recept_navn as "Recept Navn" FROM recept
WHERE recept.recept_id IN
    (SELECT receptkomponent.recept_id FROM receptkomponent
    WHERE receptkomponent.raavare_id IN
        (SELECT raavare.raavare_id FROM raavare
        WHERE raavare.raavare_navn = 'champignon'))
OR receptkomponent.raavare_id IN
    (SELECT raavare.raavare_id FROM raavare
    WHERE raavare.raavare_navn = 'skinke'));
```

Recept Navn
prosciutto
capricciosa

2.1.4

Vi skal finde navnene på de recepter som ikke indeholder ingrediensen champignon. Problemet med denne forespørgsel er at vi er nødt til at få data fra 3 forskellige tabeller for at kunne bestemme om champignon er en ingrediens i en given recept.

```
SELECT raavare.raavare_id FROM raavare
WHERE raavare.raavare_navn = 'champignon'
```

Vi starter bagfra og laver en selection hvor den eneste ting der bliver valgt er råvare id fra alle råvarer med champignon som deres navn. Derefter laver et select uden om denne, hvor vi vælger alle recept id'er som har råvare id'er inden for det sæt vi har valgt. På den måde finder vi alle de recept id'er som har champignon i sig. Dette er nemmere end bare at ekskludere champignon fra sin første selection da recepterne også har andre ingredienser i sig, hvilket kan lede til at en recept som har champignon i sig stadigvæk ville blive taget med.

```
SELECT receptkomponent.recept_id FROM receptkomponent
WHERE receptkomponent.raavare_id IN
    (SELECT raavare.raavare_id FROM raavare
     WHERE raavare.raavare_navn = 'champignon')
```

Nu hvor vi kender alle de recepter der har champignon i sig kan vi lave et select som siger at vi kun vil have recept navnene fra da recepter som ikke er i sættet for recepter som har champignon i sig.

```
SELECT recept.recept_navn as "Recept Navn" FROM recept
WHERE recept.recept_id NOT IN
    (SELECT receptkomponent.recept_id FROM receptkomponent
     WHERE receptkomponent.raavare_id IN
        (SELECT raavare.raavare_id FROM raavare
         WHERE raavare.raavare_navn = 'champignon'))
);
```

Det ovenstående er den færdige forespørgsel. Nedenfor ses resultatet. Vi har 2 recepter som ikke har champignon i sig.

Recept Navn
margherita
prosciutto

2.1.5

Vi ønsker at finde navnene på den (eller de) recepter som indeholder den største nominelle vægt af ingrediensen tomat. Vi bruger en metode som ligner den vi brugte i afsnit 2.1.4.

Den inderste parentes finder alle de råvare id'er som har navnet tomat.

```
SELECT raavare.raavare_id FROM raavare
WHERE raavare.raavare_navn = 'tomat'
```

Derefter finder vi i receptkomponent den maximale nominelle vægt af de rækker som indeholder råvare id'er som vi fandt i den inderste parentes i forespørgslen og sætter denne ind i en WHERE sætning hvor den nominelle vægt skal være lige med den maximale nominelle vægt og hvor råvare id'en stadig også er inden for den mængde af råvarer der har navnet tomat.

```
SELECT receptkomponent.recept_id FROM receptkomponent
WHERE receptkomponent.raavare_id IN
    (SELECT raavare.raavare_id FROM raavare
     WHERE raavare.raavare_navn = 'tomat')
AND receptkomponent.nom_netto =
    (SELECT MAX(receptkomponent.nom_netto) FROM receptkomponent
     WHERE receptkomponent.raavare_id IN
        (SELECT raavare.raavare_id FROM raavare
         WHERE raavare.raavare_navn = 'tomat'))
```

På nuværende tidspunkt har vi alle de recept id'er som har den maximale nominelle vægt af tomat, så vi finder nu bare de navne der hører til.

```
SELECT recept.recept_navn FROM recept
WHERE recept.recept_id IN
    (SELECT receptkomponent.recept_id FROM receptkomponent
     WHERE receptkomponent.raavare_id IN
        (SELECT raavare.raavare_id FROM raavare
         WHERE raavare.raavare_navn = 'tomat')
     AND receptkomponent.nom_netto =
        (SELECT MAX(receptkomponent.nom_netto) FROM receptkomponent
         WHERE receptkomponent.raavare_id IN
            (SELECT raavare.raavare_id FROM raavare
             WHERE raavare.raavare_navn = 'tomat')));
```

Resultatet ses her. Vi har to recepter som indeholder den største nominelle vægt af tomat. Det var kun navnene der blev ønsket, men man kunne sagtens også returnere recept id'erne ved at ændre den øverste forespørgsel til `SELECT * FROM recept`:

recept_navn	
margherita	
prosciutto	

2.1.6

Vi ønsker at bestemme relationen, som for hver produktbatchkomponent indeholder tuplen (i, j, k) bestående af produktbatchets identifikationsnummer **i**, råvarens navn **j** og råvarens nettovægt **k**.

```
SELECT
produktbatchkomponent.pb_id AS 'Produktbatch id',
raavare.raavare_navn AS 'Råvare navn',
produktbatchkomponent.netto AS 'Netto'
FROM produktbatchkomponent
LEFT JOIN raavarebatch
    ON produktbatchkomponent.rb_id = raavarebatch.rb_id
LEFT JOIN raavare
    ON raavarebatch.raavare_id = raavare.raavare_id;
```


Produktbatch id	Råvare navn	Netto
1 [->]	dej	10.05
1 [->]	tomat	2.03
1 [->]	ost	1.98
2 [->]	dej	10.01
2 [->]	tomat	1.99
2 [->]	ost	1.47
3 [->]	dej	10.07
3 [->]	tomat	2.06
3 [->]	ost	1.55
3 [->]	skinke	1.53
4 [->]	dej	10.02
4 [->]	ost	1.57
4 [->]	skinke	1.03
4 [->]	champignon	0.99
5 [->]	tomat	420.69

2.1.7

Vi vil gerne finde identifikationsnumrene af den eller de produktbatches, som indeholder den største nettovægt af ingrediensen tomat.

```

SELECT produktbatchkomponent.pb_id FROM produktbatchkomponent
WHERE produktbatchkomponent.rb_id IN
    (SELECT raavarebatch.rb_id FROM raavarebatch
    WHERE raavarebatch.raavare_id IN
        (SELECT raavare.raavare_id FROM raavare
        WHERE raavare.raavare_navn = 'tomat'))
AND produktbatchkomponent.netto =
    (SELECT MAX(produktbatchkomponent.netto) FROM
produktbatchkomponent
    WHERE produktbatchkomponent.rb_id IN
        (SELECT raavarebatch.rb_id FROM raavarebatch
        WHERE raavarebatch.raavare_id IN

```

```
(SELECT raavare.raavare_id FROM raavare
WHERE raavare.raavare_navn = 'tomat')));
```

pb_id	
5 [->]	

2.1.8

Vi skal finde navnene på alle de operatører som har været involveret i at producere partier af varen margherita.

```
SELECT operatoer.opr_navn AS "Operatoer Navn"
FROM operatoer
WHERE operatoer.opr_id IN
(
SELECT produktbatchkomponent.opr_id
FROM produktbatchkomponent
WHERE produktbatchkomponent.pb_id IN
(
SELECT produktbatch.pb_id
FROM produktbatch
WHERE produktbatch.recept_id IN
(
SELECT recept.recept_id
FROM recept
WHERE recept.recept_navn = 'margherita'
)
)
)
;
```

Operatoer Navn	
Angelo A	
Antonella B	

2.1.9

Vi prøver at finde den relation som for hver produktbatchkomponent indeholder tuplen (i, j, k, l, m, n) som består af produktbatchens id **i**, produktbatchens status **j**, råvarens navn **k**, råvarens nettovægt **l**, navnet på den recept der hører til **m** og operatørens navn **n**.

```
SELECT
```

```

produktbatchkomponent.pb_id AS 'Produktbatch id',
produktbatch.status AS 'Status',
raavare.raavare_navn AS 'Råvare navn',
produktbatchkomponent.netto AS 'Netto',
recept.recept_navn AS 'Recept navn',
operator.opr_navn AS 'Operator navn'
FROM produktbatchkomponent
LEFT JOIN produktbatch
    ON produktbatchkomponent.pb_id = produktbatch.pb_id
LEFT JOIN recept
    ON produktbatch.recept_id = recept.recept_id
LEFT JOIN operator
    ON produktbatchkomponent.opr_id = operator.opr_id
LEFT JOIN raavarebatch
    ON produktbatchkomponent.rb_id = raavarebatch.rb_id
LEFT JOIN raavare
    ON raavarebatch.raavare_id = raavare.raavare_id;

```

Produktbatch id	Status	Råvare navn	Netto	Recept navn	Operator navn
1 [->]	2	dej	10.05	margherita	Angelo A
1 [->]	2	tomat	2.03	margherita	Angelo A
1 [->]	2	ost	1.98	margherita	Angelo A
2 [->]	2	dej	10.01	margherita	Antonella B
2 [->]	2	tomat	1.99	margherita	Antonella B
2 [->]	2	ost	1.47	margherita	Antonella B
3 [->]	2	dej	10.07	prosciutto	Angelo A
3 [->]	2	tomat	2.06	prosciutto	Antonella B
3 [->]	2	ost	1.55	prosciutto	Angelo A
3 [->]	2	skinke	1.53	prosciutto	Antonella B
4 [->]	1	dej	10.02	capricciosa	Luigi C
4 [->]	1	ost	1.57	capricciosa	Luigi C
4 [->]	1	skinke	1.03	capricciosa	Luigi C
4 [->]	1	champignon	0.99	capricciosa	Luigi C
5 [->]	0	tomat	420.69	capricciosa	Antonella B

2.2

2.2.1

Vi ønsker at finde antallet af produktbatchkomponenter med en nettovægt på mere end 10. Til dette bruger vi funktionen COUNT.

COUNT bruges uden om en kolonne i en SELECT forespørgsel. Den tæller antallet af rækker som normalt ville være i denne kolonne i den normale forespørgsel. Vi laver også en WHERE kommando for at fortælle at vi kun ønsker at få de rækker tilbage som har en netto på over 10.

```
SELECT COUNT(produktbatchkomponent.pb_id) AS 'Antal' FROM
produktbatchkomponent WHERE produktbatchkomponent.netto > 10;
```

Resultatet af den ovenstående forespørgsel er:

Antal	
4 [->]	

Vi kan se hvis vi bare kigger på produktbatchkomponent at der fra starten af er 4 rækker som har en nominel vægt på over 10, så forespørgslen har virket korrekt.

2.2.2

Vi ønsker nu at finde den samlede mængde tomat som er på lageret, hvilket vil sige den samlede mængde tomat som optræder i raavarebatch tabellen.

SUM virker ligesom count, bortset fra at i stedet for at tælle mængden af rækker, så ligger den de værdier rækkerne har sammen. Vi kører sum på maengde kolonnen og bruger en WHERE kommando hvor råvare id'en skal være inden for en anden SELECT forespørgsels returnerede værdier. Denne inderste forespørgsel finder bare de råvare id'er som hører til råvarer med navnet tomat.

Den samlede forespørgsel ses nedenfor.

```
SELECT SUM(raavarebatch.maengde) AS 'Samlet maengde' FROM
raavarebatch
WHERE raavarebatch.raavare_id IN
      (SELECT raavare.raavare_id FROM raavare
       WHERE raavare.raavare_navn = 'tomat');
```

Resultatet er

Samlet mængde
600

Baseret på den inderste forespørgsel ved vi at de råvare id'er som hører til tomat er 2, 3 og 4. Vi kigger derefter i raavarebatch tabellen og ser at der er to tækker med råvare id'er inden for denne mængde. De har rb_id 2 og 3 og har hver en mængde på 300. Vores forespørgsel har altså fungeret korrekt.

2.2.3

Vi vil nu gerne finde den samlede mængde for alle ingredienser på lageret. Vi kan derfor ikke længere bruge WHERE IN kommandoerne til at finde ud af en mængde af råvare id'er som hører til en enkelt ingrediens.

Vi kan benytte os af GROUP BY til at gruppere den samlede mængde af hver vare med en råvare id, men i sidste ende er det råvarens navn vi gerne vil have, og dette ligger i en anden tabel.

Det vi gør er at joine raavare med raavarebatch. Vi bruger inner join så det kun er mængder som også har en ingrediens der hører til sig, som bliver taget med i vores join. Når vi har lavet dette join ved vi at vi har en tabel hvor vi har råvare navn, råvare id og mængder der hører sammen. Vi vælger råvarenavn fra jointet mellem de to tabeller, samt summen af mængden i raavarebatch og grupperer vores resultater efter råvarens navn sådan at SUM aggregatfunktionen lægger summerne til en given ingrediens sammen.

Den komplette forespørgsel ses nedenfor.

```
SELECT raavare.raavare_navn AS 'Ingrediens',
SUM(raavarebatch.maengde) AS 'Samlet mængde'
FROM raavarebatch
INNER JOIN raavare ON raavarebatch.raavare_id = raavare.raavare_id
GROUP BY raavare.raavare_navn
ORDER BY raavare.raavare_navn;
```

Resultatet af forespørgslen er:

Ingrediens	Samlet mængde
champignon	100
dej	1000
ost	200
skinke	100
tomat	600

Vi ved allerede at den samlede mængde af tomat er 600, så denne række er i hvert fald korrekt. Vi kan med rimelig stor sandsynlighed sige at denne forespørgsel fungerer efter hensigten.

2.2.4

Vi vil finde alle de råvarer som indgår i mindst 3 recepter.

```
SELECT raavare.raavare_navn AS 'Råvare navn',
COUNT(receptkomponent.recept_id) AS 'Antal recepter' FROM
receptkomponent
LEFT JOIN raavare
ON receptkomponent.raavare_id = raavare.raavare_id
GROUP BY raavare.raavare_navn
HAVING COUNT(receptkomponent.recept_id) >= 3;
```