Oppgave 1 - SQL

A)

```
SELECT *
FROM hundeeier;
```

B)

```
SELECT COUNT(hunde_id) AS antall_hunder
FROM hund;
```

C)

Nord-Amerika er en enum datatype, vi kan derfor være sikker på at stringen vi søker opp består av nøyaktig 'Nord-Amerika', derfor bruker jeg = tegn og ikke LIKE.

```
SELECT navn
FROM hunderase
WHERE verdensdel = 'Nord-Amerika'
ORDER BY navn ASC;
```

D)

Her bruker vi en subquery og kobler sammen rase_id-ene for å finne hvilke rase_id-er det er som er fra verdensdelen Asia.

```
SELECT navn, fødselsdato
FROM hund
WHERE rase_id = (SELECT rase_id
          FROM hunderase
          WHERE verdensdel = 'Asia')
```

E)

Her må vi bruke en LEFT JOIN slik at vi får opp *alle* hundene og deres hunderase. Deretter setter vi at rase_ID ikke kan være NULL(dvs. tom celle).

```
SELECT hund.navn, hund.fødselsdato, hunderase.navn
FROM hund
LEFT JOIN hunderase ON hund.rase_id = hunderase.rase_id
WHERE hund.rase_id IS NOT NULL;
```

F)

For å finne hvor mange hunder det er som har samme navn, grouper jeg denne spørringen etter navn(slik at det ikke blir duplikater av denne). For deretter å bruke HAVING på den opptellingen

```
SELECT navn, COUNT(*) AS antall_hunder
FROM hund
GROUP BY navn
HAVING antall_hunder > 1;
```

G)

```
/*Lager et view*/

CREATE OR REPLACE VIEW fellesnavn
AS SELECT navn, COUNT(*) AS antall_hunder
FROM hund
GROUP BY navn
HAVING antall_hunder > 1;

/*Kjører en spørring mot viewet*/

SELECT *
FROM fellesnavn;
```

H)

```
DELETE FROM hund
WHERE eier_id = 8;
```

I) WHERE clausen i denne updaten spesifiserer at det kun er eier_id 3 sin epostadresse vi ønsker å endre.

```
UPDATE hundeeier
SET epostadresse = ola.olsen@mymail.com
WHERE eier_id = 3;
```

J)

Vi legger først inn hundeeier og hunderase fordi tabellen hund har to fremmednøkler som refererer til disse to, ergo vil det være best at de tabellene fremmednøkkelene referer til allerede har den infoen som trengs. I tillegg er det oppgitt at en hund må registreres med én eier, dermed *må* vi legge til den tilhørende eieren før hunden. Det er også oppgitt at en hund ikke trenger å registreres med en rase, så det hadde strengt talt ikke vært nødvendig å legge denne infoen inn før hunden. Men, det er jo lettere å legge til hunden med all info den trenger, fremfor å først legge inn hunden og deretter UPDATE tabellen.

```
/*Første statement legger inn info i hunderase*/

INSERT INTO hunderase
VALUES (NULL, 'Schipperke', 'Europa');

/*Andre statement legger inn info i hundeeier*/

INSERT INTO hundeeier
VALUES (12345, 'Per', 'Petterssen', '1993-01-22', 'per.pettersen@somemail.no');

/*Tredje statement legger inn info i hund*/

INSERT INTO hund
VALUES (NULL, 'Passop', '2015-12-12', 35, 12345);
```

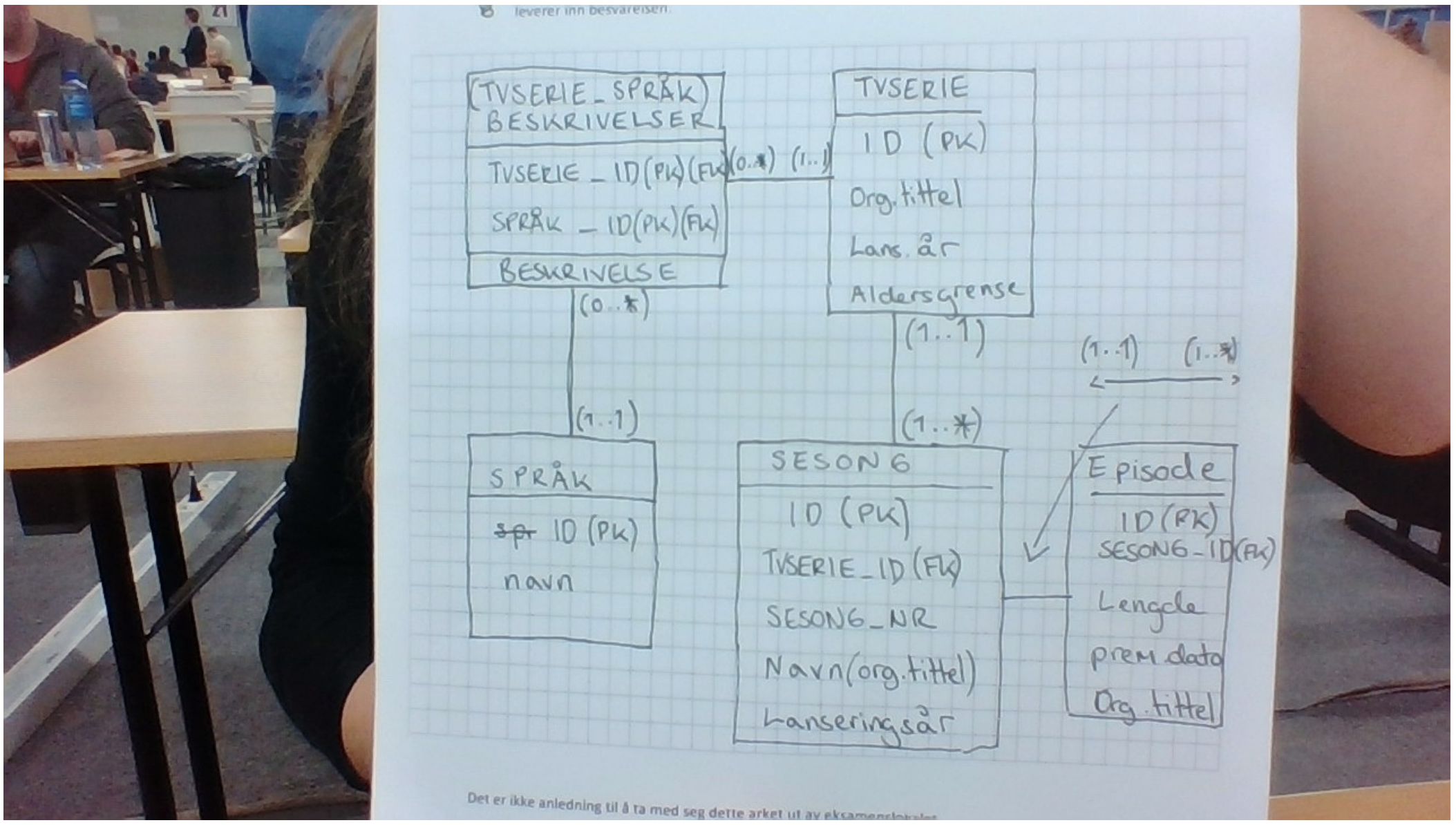
Oppgave 2: Modellering

Etter oppgaveteksten ser vi at *Tvserie*, *Sesong, Språk* og *Episode,* passer fint som hver sine entiteter. I tillegg til de attributtene

oppgaveteksten etterspør har jeg også lagt til et unikt løpenr i form av en id for hver entitet(man kan forestille seg at dette blir en auto_inkrementert id når man implementerer databasen). For sesong og episode vil sesong_id og episode_id ikke være like sesong_nr og episode_nr, da numrene deres refererer til episoden *innenfor* sesongen, og sesongen *innenfor* Tv-serien. ID-ene deres er uavhengig av dette, for å unngå å måtte ha sammensatte primærnøkler.

Vi "kobler" deretter disse entitetene sammen etter primær- og fremmednøklene deres og ser at det oppstår en mange til mange relasjon mellom *Tvserie* og *Språk*. Vi må altså opprette en koblingsenitet mellom disse, og det er i denne koblingsentiten det vil være hensiktsmessig å legge til beskrivelser slik at vi kan finne frem til beskrivelsene for *hver* tvserie på *hvert* språk.

Når det kommer til deltagelse har jeg antatt at det må finnes minst én sesong for hver tv-serie, og minst én episode for hver sesong.



Kommentarer til tegningen:

Det skal ikke være strek mellom attributet beskrivelse og resten av koblingsentiteten, det er rett og slett en tegnefeil(dvs. beskrivelse skal være en helt normal attributt i entiteten beskrivelser).

Jeg fikk ikke plass til å skrive opp deltagelse mellom entitetene *sesong* og *episode*, pilen med tilhørende strek og notasjon refererer altså til denne koblingen.

Oppgave 3: Normalisering

a) I databasesammenheng beskriver redundans forekomsten av det *samme* dataelementet mange ganger i en relasjon. Med samme menes altså at den beskriver den samme informasjonen dvs. ikke data som tilfeldigvis er lik: Et navn kan f.eks. repeteres i en tabell uten at dette nødvendigvis er et eksempel på redundans, dersom det beskriver *ulike* personers navn. Man ønsker som oftest å ha minst mulig redundans i en database, da dette vil gjøre det vanskelig å oppdatere og slette innhold i en tabell uten å gjøre feil. Dersom du, for å f.eks. slette en kunde må slette denne 10 ganger i én eller flere tabeller, fremfor én gang i en egen kundetabell kan man jo forestille seg at det kan oppstå komplikasjoner.

I den første tabellen oppgitt i oppgave 3 ser vi flere eksempler på redundans, blant annet kan vi se at kundenavn med samme kundenr repeteres flere ganger i tabellen. Vi vet at det er den samme fordi kundenummer identifiserer kunden. Grunnen til at jeg legger vekt på at det er samme *navn* som repeteres er fordi dette egt. er informasjon som kun tilhører kundenummer, ergo forekommer samme data flere ganger.

Et annet eksempel på redundans er at filmtitler med *samme* film_id repeterer seg flere ganger. Igjen vet vi at det er samme film fordi film_id er identifiserende for filmen. Tittelen er også informasjon som kun tilhører film_id, og fordi titlene forekommer gjentatte ganger er dette også et godt eksempel på redundans.

b)

For at en database skal være på 2NF må den først oppfylle kravene for 1NF, i tillegg til å ikke ha noen *delvise avhengigheter til primærnøkkelen*. Vi tar først for oss 1NF og ser at tabellen inneholder attributten kundenavn som egt. består av to ulike data: for- og etternavn. Vi splitter dermed kundenavn inn i to nye kolonner for henholdsvis for- og etternavn. Tabellen vår er nå på 1NF.

Dersom vi hadde hatt en primærnøkkel bestående kun av én kolonne, kunne man vært trygg på at tabellen var på 2NF fordi det ikke hadde vært rom for noen delvise avhenigheter. Denne tabellen derimot har en sammensatt primærnøkkel av kolonnene kundenummer og film_id. Som jeg nevnte i oppgave 3a) er kolonnen filmtittel *kun* funksjonellt avhengig av film_id(hverken kundenr, eller kundenavnene vil determinere hva filmen de har leid heter). Det samme gjelder kunde for- og etternavn(som vi nå har delt kundenavn inn i), disse to kolonnene er *kun* funksjonellt avhengig av kundenummer. Vi kan derfor konkludere med at tabellen *ikke* er på 2NF.

For å normalisere databasen deler vi dermed for-, etternavn og kundenummer inn i sin egen tabell(kundenr blir primærnøkkel). Så deler vi film_id og filmtittel inn i sin egen tabell(film_id blir primærnøkkel). Men vi vil jo fortsatt vite hvilke filmer hver kunde har leid, dette kan man jo på en måte se på som en koblingsentitet der vi har et mange til mange forhold mellom kunde og film, og dette vil derfor være en egen tabell med en sammensatt primærnøkkel av to fremmednøkler(hhv. kundenr og film_id).

Til slutt vil vi derfor stå igjen med tabellene slik som dette:

Kunde		
kundenummer (PK)	fornavn	etternavn
8	Morten	Hanssen
9	Lene	Jenssen
11	Hans	Hanssen
12	Andre	Jenssen

Film	
film_ID(PK)	filmtittel
15	Pretty Woman
24	Terminator 2
37	Tatt av vinden

Leide filmer	(evt. kunde_film)
kundenummer(PK)(FK)	film_id(PK)(FK)
8	37
9	15
11	24
12	15
12	24
12	37

C)

For å oppfylle kravene om 3NF må databasen først være på 2NF i tillegg til å ikke inneholde *transitive avhengigheter*. En transitiv avhengighet kan også beskrives som A->B->C = A->C (A determinerer B som determinerer C, dermed determinerer A også C). Jeg har tidligere gått gjennom kravene for 1NF og 2NF i deloppgave B) så her nevner jeg bare at siden det er oppgitt at ansattnummer er den

eneste primærnøkkelen vil tabellen være på 2NF. Vi kan dermed se på hvilke transitive avhengigheter det eksisterer i tabellen. Den enkleste måten å gjøre dette på er å se på hvilke kolonner det er som determinerer hva. Vi ser att ansattnummer vil determinere fornavn, etternavn og avdelingsnummer, men det er igjen avdelingsnummer som determinerer avdelingssted. Vi kan dermed konkludere at denne tabellen *ikke* er på 3NF.

For å oppnå 3NF skiller vi dermed ut avdelingsnummer og avdelingssted inn i sin egen tabell og gir denne f.eks. navnet "avdeling". avdelingsnummer. Avdelingsnummer vil da bli primærnøkkel i den nye tabellen vår, samt eksistere som en fremmednøkkel i ansatt-tabellen.

Tabellene blir altså seende slik ut på 3NF.

Ansatt			
ansattnummer(PK)	fornavn	etternavn	avdelingnummer(FK)
3	Per	Persson	3
5	Ole	Olsen	4
8	Liv	Hanssen	3
9	Beate	Jensen	2

Avdeling	
avdelingsnummer(PK)	avdelingssted
2	Fredrikstad
3	Hovseter
4	Drammen

D) For å oppfylle kravene for BCNF, må databasen først være på 3NF i tillegg til at alle *determinanter må være kandidatnøkler*. Med determinante menes det altså alle kolonner som bestemmer noe for en annen kolonne(slik som f.eks. avdelingsnummer i tabellen i oppgave C). En determinant er altså en kolonne som gjør at dersom vi vet innholdet i den determinante kolonnen, vil dette "gi" oss info i en annen kolonne.

Boyce-Codd NormalForm beskriver altså at alle slike determinanter i en tabell må være del av en *kandidatnøkkel.* Definisjonen av en kandidatnøkkel er *en supernøkkel som ikke kan reduseres til færre kolonner uten å slutte å være unik*. En supernøkkel, er rett og slett en sammensetting av kolonner som gir oss noe informasjon(den kan derfor teknisk sett faktisk bestå av samtlige kolonner i en tabell). En kandidatnøkkel er derfor altså bare en sammensetting av en eller flere ulike kolonner som vil være identifiserende for resten av tabellen(dvs. unik).

Oppgave 4 - Transaksjoner og ACID

a) En transaksjon er en sammensettning av SQL-statements som kjøres "sammen". Med sammen menes altså at enten utføres samtlige statements innenfor denne transaksjons-bolken, ellers kjører ingen av dem.

BEGIN TRANSACTION vil "starte" utførelsen av en transaksjon, men statementsene innenfor den vil ikke være gyldige med mindre dette etterfølges av en COMMIT. COMMIT beskriver altså at statementsene i transaksjonen skal være gyldige. Dersom begin transaction derimot etterfølges av en ROLLBACK, vil ingen av statementsene innenfor transaksjonen gjennomføres, og det vil ikke være noen endringer i databasen fra denne transaksjonen.

b)

ACID-egenskapene, er en forkortelse for fire ulike egenskaper man ønsker at en transaksjon skal ha.

Atomicity

Transaksjonen skal være "atomisk" det vil si, den skal oppføre seg som én egen entitet(entitet menes ikke her likt som i modelleringssammenheng), og ingen av delene innenfor transaksjonen skal kunne påvirke noe utenforliggende.

Consistency

Transaksjonen skal gjennomføres slik at databasen både starter og ender opp i en gyldig tilstand. Dvs. at en transaksjon skal ikke kunne føre til at databasen ender opp i en ufullstendig tilstand.

Isolation

Transaksjonene skal kunne gjennomføres samtidig og uavhengig av hverandre, det skal altså være mulig å kjøre flere transaksjoner "*samtidig"* på samme database uten at dette fører til feil.

Durability

En transaksjon, når gjennomført skal være "permanent". Permanent i den forstand at dersom endringen transaksjonen gjør skal eksistere helt frem til man evt. endrer samme data igjen.