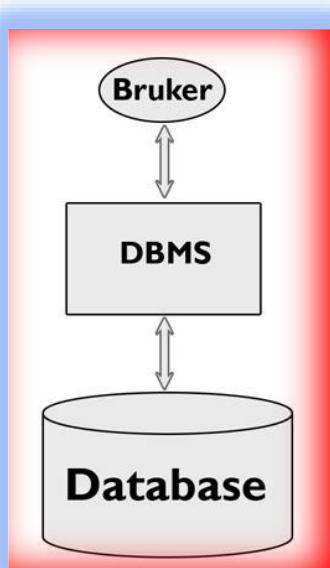


# Databaser – praksis og teori.

Dette kompendiet er en oversikt over en del sentrale temaer innen databaser. Opplegget er ment som **støtte til forelesningene og lærebok**. Mer stoff finnes i lærebøker, manualer og ikke minst på nettet.



Jeg tar gjerne imot feilretting og forslag til forbedringer.

Edgar Bostrøm

Vennligst **ikke viderekopier!**

<sup>1</sup> <https://www.digi.no/artikler/sa-mange-leier-photoshop/288398>

© Edgar Bostrøm



# Forord

Dette kompendiet har vært utviklet i flere versjoner gjennom mange års forelesninger i databaser ved Høgskolen i Sør-Øst-Norge, HSN på Ringerike, Universitetet for Miljø- og Biovitenskap, NMBU<sup>2</sup> i Ås, men først og fremst ved Høgskolen i Østfold, HiØ. Teksten er skrevet med så store fonter at den egner seg til **bruk direkte i forelesningene**, slik at studentene kjenner igjen stoffet når de leser det igjen senere. Dermed kan man også skrive ut kompendiet på A5-format, evt. som et hefte.

Målet har vært å gi en grundig innføring i databaser generelt og relasjonsdatabaser spesielt (engelsk: Relational Data Base Management Systems, RDBMS), som har vært og er den dominerende database-typen. I praksis er det snakk om systemer som bygger på standarden SQL (Structured Query Language).

Andre databaseformer, som objekt-orienterte systemer (OODB), not-only SQL (noSQL) m.fl., er mye i bruk i spesialanvendelser, ikke minst i forbindelse med svært store datamengder («Big Data») og en del på web-baserte systemer. På tross av 40 års historie, er likevel SQL-baserte systemer standard i de fleste sammenhenger. Og: kunnskap i RDBMS er nødvendig også om man jobber med andre databasetyper.

Kompendiet kan grovt sett deles i 6 deler, som kan tenkes på som en logisk oppbygging fra det generelle om databaser, via praksis i SQL til videregående emner.

- Del A: Generelt om databaser, som er grunnlag for relasjonsdatabaser, uavhengig av SQL.
- Del B: SQL-språket, hvordan tabeller defineres, hvordan data settes inn, endres, slettes og plukkes ut.
- Del C: «Bak kulissene» i relasjonsdatabaser (indeks, systemtabeller, brukere m.m.)
- Del D: Datamodellering og normalisering (hvordan planlegge, lage en «arkitekttegning» for databaser).
- Del E: Transaksjoner og samtidighet (problemer kan oppstå når flere bruker samme database samtidig).
- Del F: Diverse temaer.

Selv om denne rekkefølgen etter min mening er en logisk oppbygging av et slikt kompendium, mener jeg det er gode grunner til å gjennomføre et kurs i databaser i en litt annerledes rekkefølge. Jeg mener at man får best forståelse av databaser ved først å bruke allerede eksisterende data, og gjøre spørrenger på disse. Jeg har brukt databasen [www.sqlzoo.net](http://www.sqlzoo.net) (som ligger på en datamaskin et eller annet sted i Skottland) for å lage enkle spørrenger som en første innføring i SQL. Den inneholder informasjon om land, areal, befolkning, etc. for alle land i verden. Det er også mer motiverende enn å lage en tabell med 5-6

Det er bevisst at jeg bruker **litt** matematikk (se kap. 23, TILLEGG: Om og og eller, eller eller og og, og om alle, noen og ingen, om sannhet og usannhet, om tilhørighet, Om alting og ingenting (!).<sup>3</sup>) og annen teoretisk bakgrunn for å begrunne de ulike praktiske delene. Dermed ser man også at også at relasjonsdatabaser bygger på et solid teoretisk grunnlag. Relasjonsbegrepet er så knyttet til databaser at det tas fortløpende.

Min rekkefølge i undervisningen av den første delen av kurset har dermed vært:

Bakgrunn	Tema	Kommentar
Litt logikk	→ Enkle spørrenger med bruk av logikk.	Bruk av allerede registrerte data, spesielt <a href="http://www.sqlzoo.net">www.sqlzoo.net</a>
Litt om mengder	→ Spørrenger med bruk av mengder	
All- og eksistenskvantorer	→ Spørrenger med EXISTS, ALL og ANY	
Litt generelt om databaser (spesielt integritetsregler)	→ Laging / endring av databasestruktur.	Bruk av eget databasesystem, f.eks. MySQL.

<sup>2</sup> HSN var tidligere SLHK, HiBu, HVE. NMBU var tidligere UMB, NLH.

<sup>3</sup> Ja, jeg kommer faktisk til å ta opp alle disse begrepene, men det er enklere enn du tror!

I de senere delene av kurset er det mer naturlig å følge den logiske rekkefølgen ut fra kompendiet. Det finnes mye om databaser og SQL på nettet. Spesielt anbefalt: <https://www.w3schools.com/sql>, og det er lurt å bruke dette og andre for detaljer, andre eksempler osv. Det erstatter likevel ikke systematiske oversikter, som et kompendium eller lærebok.

Et viktig aspekt er at databaser må læres gjennom praksis, spesielt når det gjelder SQL og datamodellering/normalisering. Hvilket databasesystem du bruker spiller liten rolle. Tidligere har jeg brukt Oracle og MS SQL Server i undervisningen, men de siste årene har MySQL vært brukt.

Andre opplegg innenfor databaser:

- distribuerte databaser
- detaljer om 4. og 5. normalform
- triggerer og lagrede prosedyrer
- relasjonsalgebra og relasjonskalkyle
- datavarehus
- objektorienterte databaser
- datamodellering: Egen bok:  
"Datamodellering – praksis og teori", ISBN 82-91915-21-0

Lykke til – det er moro å få til databaser!

**Halden, 08.01.2018.**

# Innholdsfortegnelse

## Innhold

1.	<i>Kort innledning om databaser</i> .....	10
1.1.	Databaser - generelt .....	10
1.2.	Databaser på datamaskiner.....	11
1.3.	Databasesystemer - for å håndtere data i en database.....	12
1.4.	Oppbygging av en database ut fra en datamodell. ....	13
2.	<i>Relasjonsdatabaser – litt begreper og teori</i> .....	14
2.1.	Hva skal være strukturen i en database? .....	14
2.2.	Hva er en relasjon?.....	18
2.3.	Teoretiske begreper - relasjonsdatabaser .....	20
2.4.	En alternativ definisjon av relasjoner .....	21
2.5.	Relasjoner som matematiske objekter.....	22
3.	<i>Integritetsregler for databaser</i> .....	23
3.1.	Entitetsintegritet.....	23
3.2.	Referanseintegritet .....	23
4.	<i>Koblingsformer</i> .....	25
5.	<i>Relasjonsdatabasesystemer - noen produkter</i> .....	26
6.	<i>Relasjonsoperatorer</i> .....	27
6.1.	Utplukk fra relasjoner – kort om 3 relasjonsoperatorer. ....	27
6.2.	Utplukk fra relasjoner – kort om mengdeoperasjoner. ....	28
6.3.	Enda kortere om andre relasjonsoperatorer .....	29
6.4.	Generelt om relasjonsoperatorer. ....	29
7.	<i>SQL-språket - oversikt</i> .....	30
7.1.	Generelt .....	30
7.2.	DDL – datadefinisjonsspråket – kort oversikt .....	31
7.3.	DML - data-manipulasjons-språket - kort oversikt .....	32
8.	<i>Datadefinisjon</i> .....	33
8.1.	Create table-setninger, kun til bruk når det ikke er sammensatte nøkler. ....	33
8.2.	Create table-setninger, generelt. ....	34
8.3.	Handling ved innsetting / endring / sletting av fremmednøkler .....	38
8.4.	Navngivning av nøkler. ....	40
8.5.	SQL: generelt om skranker i tabeller .....	41
8.6.	Datatyper.....	42
8.7.	Endring / sletting av tabeller .....	42
9.	<i>Datomanipulasjon</i> .....	43
9.1.	Enkle spørninger med en tabell .....	44
9.2.	Spørninger med logiske operatorer.....	47
9.3.	Spørninger bygd koblinger, m/ kobling i where-setningen. ....	50

9.4.	Spørninger bygd koblinger m/ kobling i from-setningen .....	52
9.5.	IN- operatoren, inkl. delspørninger med IN .....	53
9.6.	Exists / not exists.....	55
9.7.	Gruppering (aggregering).....	56
9.8.	ALL / ANY (SOME).....	59
9.9.	Union, snitt, differanse etc. ....	61
9.10.	Beregningsuttrykk, kommentarer etc. i SQL-setningene .....	62
9.11.	Alias .....	62
9.12.	Kartesisk produkt.....	65
9.13.	Noen funksjoner som det kan være behov for .....	66
9.14.	Innsetting av nye data. ....	67
9.15.	Endring og sletting av data.....	69
9.16.	I praksis: Data legges inn, endres og søkes etter via et brukergrensesnitt. ....	70
10.	<i>Bruk av SQL for å kommunisere mellom ulike systemer</i> .....	72
10.1.	Verktøy og lagring av data .....	72
10.2.	ODBC, JDBC m.fl. ....	72
11.	<i>Utsnitt / visning / view</i> .....	75
11.1.	Utsnitt / visning / view i SQL .....	75
11.2.	Utsnitt: typer og bruksområder. ....	76
11.3.	Hvordan utføres en SQL-select-setning med utsnitt? .....	77
11.4.	Oppdaterbarhet via et utsnitt: .....	78
12.	<i>Indekser, fysisk datadefinisjon og aksessmekanismer</i> .....	79
12.1.	Indekser – hva og hvorfor .....	79
12.2.	Indeksdefinisjon - SQL. ....	79
12.3.	Indekser - fysisk databasedefinsjon. ....	80
12.4.	Fullstendige/ufullstendige indekser.....	80
12.5.	Indekser - i praksis. ....	82
12.6.	Noen variasjoner mellom ulike DBMS. ....	83
12.7.	Hva bør indekseres? .....	84
12.8.	Eksempel på B-tre .....	85
13.	<i>Adgangskontroll til en database</i> .....	86
13.1.	Hensikten med adgangskontroll.....	86
13.2.	SQL-syntaks for adgangskontroll .....	87
14.	<i>Systemtabeller</i> .....	88
14.1.	Hva er en systemtabell? .....	88
14.2.	Eksempel: systemtabeller fra et reelt system .....	89
15.	<i>SQL-språket – sammenfatning</i> .....	90
15.1.	Parallelitet mellom metadata og data .....	90
15.2.	Noen bruksområder for SQL .....	91
15.3.	Typisk arkitektur for et SQL-basert system.....	92
16.	<i>Datamodellering</i> .....	93
16.1.	Datamodellering – prinsipielt.....	93

16.2.	Grunnleggende notasjon, som kråkefot hhv. UML-basert.....	98
16.3.	Forholdet mellom ting og navn på ting.....	100
16.4.	Ulike notasjoner/«dialekter» (orienteringsstoff).....	101
16.5.	Generell metodikk (kort) .....	108
16.6.	Vanlige strukturer .....	111
16.7.	Plassering av entitetstyper i modellen. ....	113
16.8.	Notasjon og variasjoner i 3 av dialektene: Chen, kråkefot og nedskalert UML.....	114
17.	<i>Normalisering - t.o.m. BCNF</i> .....	116
17.1.	Hva er normalisering? .....	116
17.2.	Funksjonelle avhengigheter / determineringer.....	117
17.3.	Normaliseringsregler – oversikt .....	118
17.4.	Eksempel på normalisering .....	119
17.5.	Spiller det noen rolle hvorledes man splitter? .....	122
17.6.	Oppsummering / sammenstilling: .....	123
17.7.	Testskjema – normalisering.....	125
17.8.	Litt om 4. normalform-problemer, eksempel.....	126
17.9.	Ulike utgangspunkt for å lage en database: .....	127
17.10.	Normalisering i praksis.....	128
18.	<i>Ulike modeller - prinsipiell oppbygging av en database</i> .....	129
19.	<i>Dataavhengighet. ANSI/SPARC-arkitekturen</i> .....	131
19.1.	Dataavhengighet.....	131
19.2.	ANSI-SPARC-arktitekturen .....	132
19.3.	Sammenligning ANSI/SPARC v.s SQL .....	135
20.	<i>Transaksjoner &amp; samtidighet</i> .....	136
20.1.	Eksempler på samtidighetsproblemer: NB! Hvis vi ikke har transaksjoner. ....	137
20.2.	Serialiserbarhet. ....	138
20.3.	Låsing. ....	139
20.4.	2-fase-låsing.....	140
20.5.	Vranglås (deadlock). ....	141
20.6.	Tidsstempeling av transaksjoner. ....	143
20.7.	Pessimistiske vs. optimistiske teknikker. ....	145
20.8.	Gjenopprettelse av databaser. ....	147
20.9.	Databaser og sårbarhet.....	150
21.	<i>Ulike temaer - databaser</i> . ....	151
21.1.	Utviklingstrender.....	151
21.2.	Datavarehus .....	151
21.3.	XML og JSON .....	152
21.4.	OO og OR-databasesystemer. ....	154
21.5.	noSQL-systemer .....	154
21.6.	Big Data.....	155
22.	<i>Noen bøker innen databaser</i> .....	157

23. <i>TILLEGG: Om og og eller, eller eller og og, og om alle, noen og ingen, om sannhet og usannhet, om tilhørighet, om allting og ingenting (!).</i> .....	158
23.1. Logikk, lukkede og åpne utsagn.....	159
23.2. Åpne utsagn og kvantorer.....	160
Mengder .....	162
23.3. Relasjoner og funksjoner .....	163
23.4. Nettverk.....	164
23.5. Trær.....	167
23.6. Logikk, med matematiske symboler (alternativ til kap. 23.1). .....	168

# 1. Kort innledning om databaser

## 1.1. *Databaser - generelt*

En database er en samling med data, strukturert på en slik måte at det er lett å registrere og søke etter data.

Dersom det er svært enkle data med en enkel struktur, kan man lagre en database på f.eks.

- arkivkort
- ringperm
- et tekstbehandlingsdokument



Ulemper med slike databaser, bl.a.:

- tvinger ikke en helt fast struktur
- svært mye data
- gjerne sortert bare på ett kriterium (fysisk rekkefølge)
- vanskelig å kombinere data på ulike kriterier
- vanskelig med samtidig bruk
- dårligere sikkerhet (brann, innbrudd)
- tar mye plass (hvis det ikke blir lagret digitalt)
- etc. etc.

Dersom vi har en mer komplisert struktur eller ønsker ulike søkemuligheter etc. lagres gjerne databasen på en/flere datamaskiner. Men: vær klar over at lovgivning om personvern, inkl. felles personvernforordning i EU fra mai 2018 (The General Data Protection Regulation, GDPR) for en stor del også gjelder for manuell lagring av data.

## 1.2. Databaser på datamaskiner.

### Hvordan strukturere kompliserte data på en datamaskin?

I praksis er såkalte relasjonsdatabaser (= tabell.databaser) dominerende<sup>4</sup>. Dataene lagres i tabeller - og bare tabeller. Sammenhenger mellom data i flere tabeller vises ved data i en tabell "matcher" med data i en (annen) tabell.

Eksempel:

AVDELING

avdkode	avdnavn	etasjenr	areal
1	Ost	4	100
14	Hvite potteplanter	5	1000
3	Sko	4	
2	Kjøkken	57	900
5	Røde potteplanter	1	50

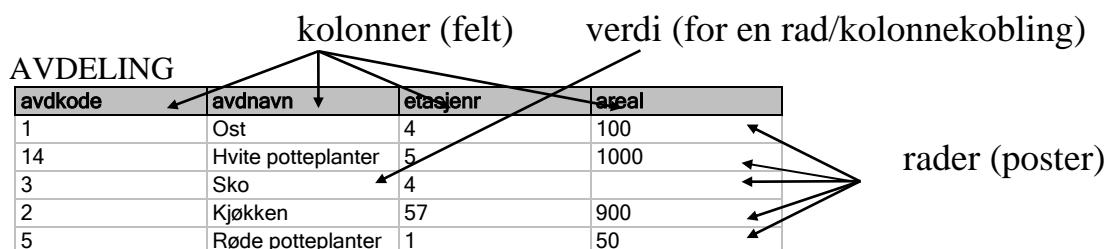
ANSATT

ansattnr	fnavn	enavn	adresse	telefonnr	avdkode	lønn
1000	Anders	Andersen	Aveien 1	1234567	3	450000
1001	Bernt	Bertsen	Bveien 1	2345678		
1002	Cesar	Cesarsen	Cveien 1	3456789	1	500000
1003	David	Davidsen	Dveien 1	4567890	1	600000
9834	Edgar	Edgarsen	E	5678901	14	750000

- Hvordan finne fram til avdelingsnavn og etasje som Cesar jobber i?
- Hvilke fordeler er det med å dele disse data på to tabeller?

Altså: vi bruker en enkel metodikk for å koble sammen data, men det kan gjøres mange ganger, slik at vi kan modellere en komplisert struktur.

### Grunnbegreper - tabeller:



Merk: ingen bestemt rekkefølge radene imellom (usortert)

<sup>4</sup> Vi snakker om før-relasjonelle databasestrukturer (f.eks. hierarkiske og nettverksdatabaser). Det finnes også produkter som er rent objektorienterte eller som kombinerer relasjonell og objektorientert tenkning. Det finnes også systemer som brukes i tilfeller hvor det er så store datamengder at relasjonsdatabaser ikke er raske nok («Big Data»). I tillegg har vi enklere lagringsformer som f.eks. JSON som er mye brukt bl.a. i enkle nettbaserte systemer. Relasjonstenkning er imidlertid fremdeles det dominerende pr. i dag, og vil antagelig være det også framover.

### **1.3. Databasesystemer - for å håndtere data i en database.**

Oppgavene rundt håndtering av en database er like, uavhengig av hvilke data som lagres ==> det er laget generelle databasehåndteringssystemer (Data Base Management System, DBMS), oftest kalt bare databasesystemer.

**I praksis er de aller fleste relasjonsdatabasesystemer (RDBMS)**

#### **DBMS har tre hovedoppgaver:**

##### **Datadefinisjon, dvs. definere (lage/endre) datastrukturen.**

- Tabeller og kolonner (og om det er heltall, tekst, bilde etc.)
- Skranker (dvs. regler/begrensninger i databasen).

Fire viktige skranker i rene relasjonsdatabaser:

- Atomisitet, at en kolonne bare inneholder en ”udelelig” verdi.
- Primærnøkkel, må velges som en kolonne/kolonnekombinasjon som har entydige verdier, og som identifiserer / representerer denne tabellen.
- Fremmednøkkel, at en verdi i en tabell skal ”matche med andre” verdier, gjerne i en annen tabell
- Verdinødvendighet, om en kolonne alltid må ha en verdi (være fylt ut).
- Indekser og andre mekanismer for å få en rask database, m.m.

##### **Datomanipulasjon, dvs. legge inn/endre selve dataene.**

- innsetting (registrering), endring, sletting
- selektering (utplukk), via kobling, spørring m/ sortering etc, summering etc.

##### **Annen adminstrasjon,**

- brukere/brukerrettigheter
- samtidighetskontroll
- sikkerhetskopiering, m.m.

#### **Kommunikasjon med databasen.**

De fleste systemer har flere muligheter for kommunikasjon direkte med databasen og f.eks. grensesnitt på skjerm:

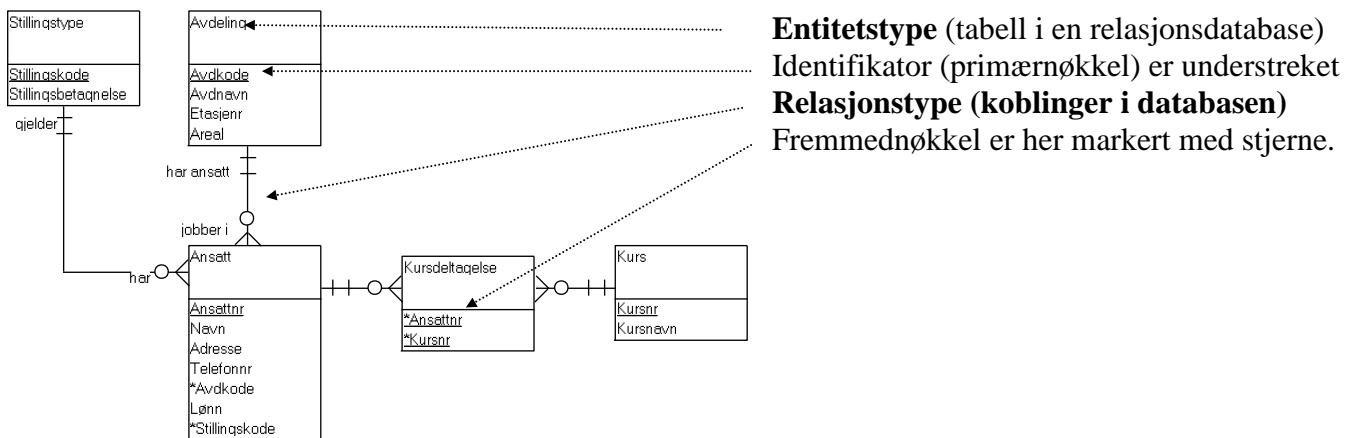
- Et grafisk grensesnitt, spesielt for det aktuelle systemet.
- Et tekstlig standardisert språk, SQL (select .., insert osv.)

Den mest vanlige er imidlertid at dette bygges inn i en ferdig applikasjon. Databaser er sentralt i de aller fleste IT-systemer.

## 1.4. Oppbygging av en database ut fra en datamodell.

Et godt råd er å lage en grafisk modell (datamodell) som viser hvorledes databasen skal se ut. Dette blir som en arkitekttegning for systemet.

Eksempel på en enkel modell:



Bruk mye tid på å lage en god modell, bl.a. pass på:

- å tenke på den logiske oppbygningen / sammenhengen mellom data, ikke på hvorledes skjermbilder etc. vil se ut.
- å tenke på “tingene” først - Ansatte, Avdelinger etc., og deretter sammenhenger mellom disse.
- at “tingene” (entitetstypene) er klart definert - slik at ikke ulike personer mener ulike ting om det samme.
- at hver entitetstype bare kan inneholde informasjon om en ting, og ikke en ting som gjentas mange ganger.
- at all informasjon du trenger i systemet enten finnes som attributter/kolonner, eller kan beregnes ut fra disse (summer, multiplikasjoner etc.)
- at data ikke gjentas, bortsett fra for å danne koblinger (fremmednøkler)
- du kan godt lage mange-til-mange-forhold, men disse må splittes i 2 en-til-mange før du lager databasen i databasesystemet.
- at de som skal bruke systemet blir spurtt om hva som viktige “ting og sammenhenger” i virksomheten.

NB: Det finnes flere ulike standarder for hvorledes slike modeller tegnes, se kap. 16.4.

## 2. Relasjonsdatabaser – litt begreper og teori

### 2.1. Hva skal være strukturen i en database?

Det er en rekke spørsmål man må ta stilling til når man skal strukturere data. Det gjelder bl.a.

- regler som er en del av en ”databasefilosofi”, f.eks. relasjonsdatabaser
- struktur i selve dataene, og som (ofte) kan defineres i databasesystemet

For enkelhets skyld bruker vi begreper fra relasjonsdatabaser i forklaringen.

### Skal vi tillate mange verdier i en kolonne? Nei - Atomærkravet.

En kolonne bør være atomisk, dvs. at vi ikke bør tillate sammensatte strukturer, som for eksempel

avdkode	avdnavn	etasje_og_areal
1	Ost	3 / 100

- FY!, sammensatt, flere opplysninger i en kolonne

avdkode	avdnavn	etasjenr	areal
17	Administrasjon	3, 5	600

- FY!, repeterende, en avd. kan være i flere etasjer

avdkode	avdnavn	etasje_og_areal
17	Administrasjon	3/400, 5/200

- FY!, både sammensatt og repeterende

Noen databasesystemer tillater dette<sup>5</sup>, men rene relasjonsdatabaser krever at hver kolonne bare har **en** informasjon, vi sier at kolonnen må være **atomisk**<sup>6</sup>.

### Er det nødvendig å fylle ut alle verdier? – Verdinødvendighet.

avdkode	avdnavn	etasjenr	areal
3	Sko	4	
2	Kjøkken	57	90

Finnes ikke noen verdi.

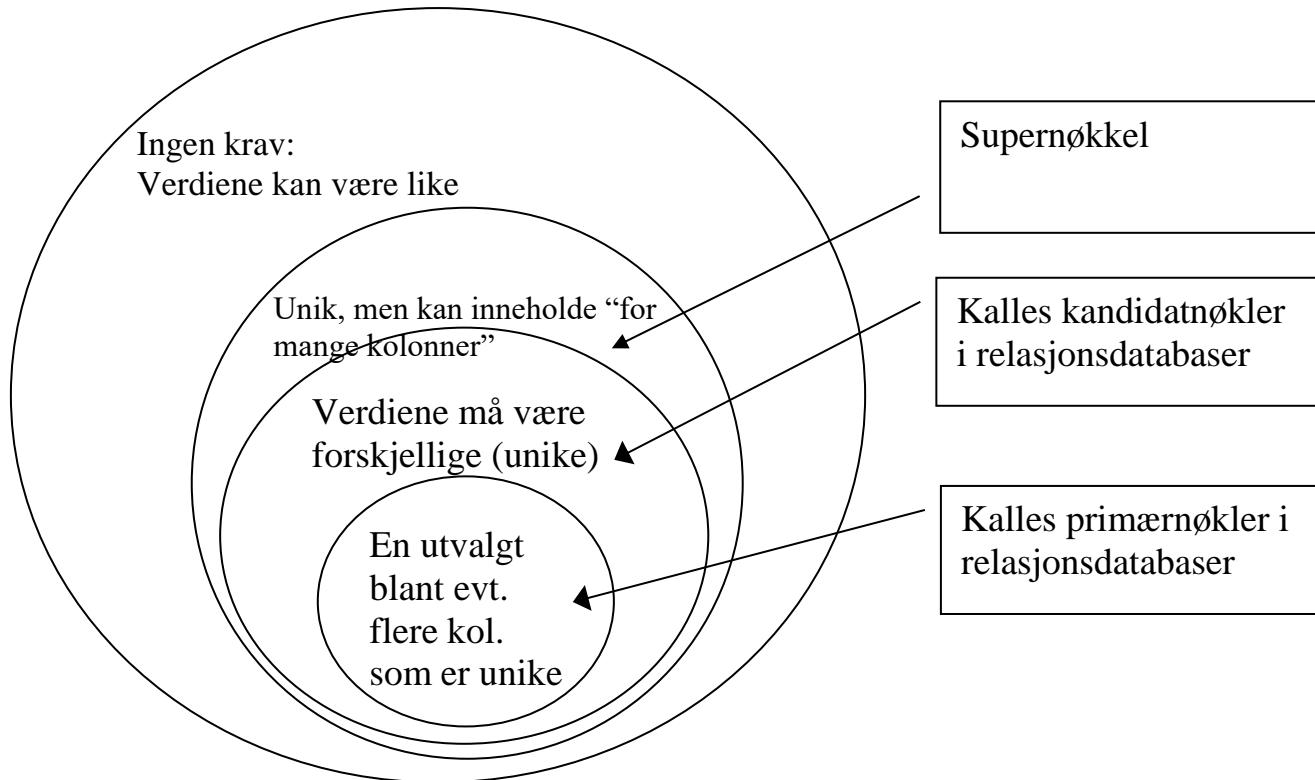
Her finnes ikke noe areal for avdkode 3. Kanskje skyldes det at avdelingen er helt ny, kanskje vet man ikke dette, eller det er uinteressant. Derimot bør man kunne kreve at avdkode alltid er nødvendig å fylle ut, muligens gjelder det også avdnavn. Det bør derfor være slik at vi må kunne bestemme om vi bør ha verdinødvendighet eller ikke for den enkelte kolonnen. I SQL-språket defineres verdinødvendighet som NOT NULL.

<sup>5</sup> Bl.a. objekt-relasjonelle databasesystemer.

<sup>6</sup> Selvsagt vet vi at det man en gang trodde var atomisk, dvs. u-delelig, i virkeligheten består av mindre deler (protoner, nøytroner, elektroner og enda mindre elementer), men begrepet atom brukes likevel i betydningen udelelig.

## Skal vi tillate like verdier? Unikhet. Kandidat- og primærnøkler.

Vi tenker f.eks. på verdiene i kolonner (evt. kolonnekombinasjoner) som avkode, avdelingsnavn, etasje, areal.



- Kravene kan godt gjelde kombinasjoner. Eksempel: selv om det finnes flere like avdelingsnavn, er det neppe like avdelingsnavn i samme etasje, slik at kombinasjonen (avdelingsnavn, etasje) er unik.
- Det er vanlig å stille krav om **"minimalitet"**/**"irredusibilitet"** til noe som er unikt. F.eks. er kombinasjonen (avkode, areal) unik. Her er imidlertid avkode unik i seg selv. En unik, men ikke nødvendigvis irredusibel struktur kalles en supernøkkel.
- Primærnøkkelen er også en kandidatnøkkel, hvis det er flere mulige kandidatnøkler velges en av disse som primærnøkkelen. Vi krever også verdinødvendighet (NOT NULL) for primærnøkkelen.
- Det må skilles mellom tilfeldige egenskaper ("ekstensjoner") og strukturelle egenskaper ("intensjoner"). Selv om det tilfeldigvis ikke finnes to like arealer i avdelingstabellen, er det ikke noen grunn til at det skal være en strukturell, fast egenskap.

### Oppsummert, via mengdesymboler:

$$\text{Ingen krav} \supseteq \text{Supernøkkel} \supseteq \text{Kandidatnøkkelen} \supseteq \text{Primærnøkkelen}.$$

## Må data være sorterte? (Svar: Nei)

Fra en side sett er det viktig at data er sorterte når vi skal bruke dem, men

- det ulikt hva vi ønsker dataene sortert på
- dataene er de samme enten de er sortert eller ikke, som her:

er det samme som

avdkode	avdnavn	etasjenr	areal
1	Ost	4	100
14	Hvite potteplanter	5	1000

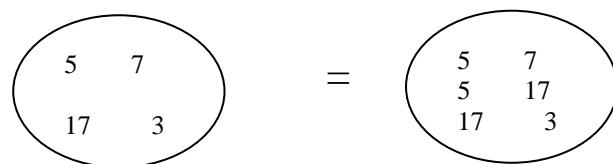
avdkode	avdnavn	etasjenr	areal
14	Hvite potteplanter	5	1000
1	Ost	4	100

I relasjonsdatabaser har man valgt en enklest mulig struktur: **dataene er usortert**.

Men: vi kan selvsagt sortere dem på et eller annet kriterium når de skal presenteres eller behandles.

Dersom rekkefølge er viktig er det nødvendig å legge på en ekstra kolonne som viser en rekkefølge. Eksempel: dersom vi skal ha lagre data om en rekkefølge av innlegg (f.eks. i en diskusjon på et møte eller på nettet) i en relasjonsdatabase, må du også ha med en kolonne som viser rekkefølgen.

Teorien for relasjonsdatabaser bygger mye på mengder (engelsk: set) fra matematikken. Disse har heller ingen rekkefølge, spørsmålet er bare om elementene er med i mengden eller ikke.



## Skal det kunne brukes pekere? (Svar: Nei)

Innen databehandling brukes det ofte pekere, f.eks. et tall som viser plasseringen av data i indre lager mens et program kjører. Dette kan være hensiktsmessig i mange sammenhenger, men gir en mer komplisert struktur. I relasjonsdatabaser finnes det ingen pekere el.l. Andre databaseformer (bl.a. nettverksdatabaser og objektorienterte databaser) bruker pekere.

## Hvordan ”koble” data? Fremmednøkler.

I stedet for å peke direkte på posisjonen der dataene ligger, må vi ha en annen måte å koble data i ulike tabeller på. For å koble data som ligger i (ulike) tabeller, bruker vi like verdier i data.

## AVDELING

avdkode	avdnavn	etasjenr	areal
1	Ost	4	100
14	Hvite potteplanter	5	1000
3	Sko	4	
2	Kjøkken	57	900
5	Røde potteplanter	1	50

## ANSATT

ansattnr	fnavn	enavn	adresse	telefornr	avdkode	lønn
1000	Anders	Andersen	Aveien 1	12345678	3	450000
1001	Bernt	Bertsen	Bveien 1	234567890		
1002	Cesar	Cesarsen	Cveien 1	345678901	1	500000
1003	David	Davidsen	Dveien 1	456789012	1	600000
9834	Edgar	Edgarsen	Eveien 8	567890123	14	750000

Legg merke til at koblingen skjer på likhet i verdier, ikke på rekkefølge av radene, jf. at datene i tabellene over er usorterte.

Det viktigste med fremmednøkler er imidlertid at man sikrer integritet mellom dataene i to tabeller. Det er et brudd på såkalt referanseintegritet dersom vi hadde en avdkode=8 i ANSATT uten at det finnes en tilsvarende i AVDELING. Se senere, kap. 3.2.

## Oppsummering – strukturen i relasjonsdatabaser

### Faste egenskaper:

- Relasjonsdatabaser består bare av tabeller (ingen pekere el.l)
- Radene er usorterte, ingen krav til rekkefølge
- Atomærprinsippet: en kolonne skal kun inneholde en verdi.
- I en et relasjonsdatabasesystem bør vi i kunne definere bl.a. følgende strukturer:

### Egenskaper som vi selv bestemmer:

Strukturelement	gjelder	Kommentar
Verdi-nødvendighet	<i>Kolonner</i>	en kolonne alltid MÅ inneholde en verdi
Primærnøkkel	<i>Tabeller</i>	en valgt kolonne (eller –kombinasjon) som alltid har ulike verdier. Brukes til å identifisere radene i tabellen. Krav til en primærnøkkel: <ul style="list-style-type: none"> <li>○ unikhet</li> <li>○ minimalitet/ikke-reduksibilitet</li> <li>○ verdi-nødvendighet</li> </ul>
Andre unike (andre kandidatnøkler)	<i>Tabeller</i>	andre kolonne (eller –kombinasjon) som også alltid har ulike verdier. Krav til kandidatnøkler: <ul style="list-style-type: none"> <li>○ unikhet</li> <li>○ minimalitet/ikke-reduksibilitet</li> </ul>
Fremmednøkkel	<i>Mellom tabeller</i>	en kolonne (eller – kombinasjon) hvor verdiene alltid skal matche verdier i en (annen) <sup>7</sup> tabell.

<sup>7</sup> I noen situasjoner definerer vi opp fremmednøkler mellom kolonner i samme tabell.

## 2.2. Hva er en relasjon?

### Egentlig betydning. Litt teori om relasjoner.

Ordet relasjon kommer fra matematikken, og er et generelt begrep på linje med f.eks. funksjonsbegrepet eller mengdebegrepet.

Gitt en relasjon R og to variable, x og y. De kombinasjonene av x og y hvor "x er relatert til y" er sant, danner verdiene for relasjonen R.

#### Eksempel 1:

R = Person x har studert fag y.

x	y
Ole	Informatikk
Ole	Norsk
Jens	Informatikk
osv.	osv.

Dette er altså en opplisting av alle kombinasjoner som gjør at R er sann, dvs. at x er relatert til y er sant. Dette skrives kort  $xRy$ , evt.  $R(x,y)$ .

Angela Merkel	Norsk
---------------	-------

er ikke med i relasjonen, fordi Angela Merkel ikke har studert norsk (så vidt jeg vet).

#### Eksempel 2:

R = "Tallet x < tallet y".

x	y
3	4
-1	3457,7767623
$\pi$	5.78
osv.	osv.

8	7
---	---

er derimot ikke med i relasjonen, siden  $8 < 7$  er usant.

Her er altså  $R = <$ , slik at  $xRy$  er det samme som  $x < y$ .

#### Eksempel 3:

Man kan studere generelle egenskaper ved relasjoner, som f.eks.

- a) "Hvis  $xRy$ , så er  $yRx$ " (kalles symmetri)
- b) "Hvis  $xRy$  og  $yRz$ , så er  $xRz$ " (kalles transitivitet)
- c)  $x R x$  (refleksivt)

Sjekk dette for relasjonene:

- $R = \text{"er gift med"}$
- $R = \text{"er barn til"}$
- $R = \text{"er yngre enn"}$
- $R = \text{"er søskjen til"}$

a)	b) (sant/usant)

## Konklusjon - egentlig betydning:

En relasjon R mellom x, y, z, .... er alle kombinasjoner av x,y,z,... som gjør R "sann". Hvis det ikke er uendelig mange, kan de settes opp i en tabell.

Eksempel: På en arbeidsplass ønsket man å sette opp en relasjon mellom kjennetegn på en bil som en av de ansatte eier, personnummer, samt biltype.

x = Kjennetegn,                y = Personnummer,                z = Biltype, slik:

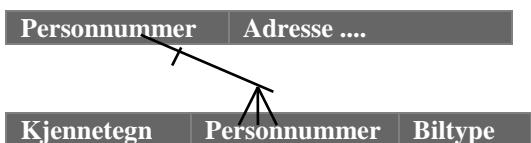
Kjennetegn	Personnummer	Biltype
BS 461812	12128012121	Skoda Fabia
AS 12345	21118112121	Ford Fiesta
XR 56321	13128899119	.....
....	....	....

(dvs. det er disse kombinasjonene som er sanne).

Altså: en relasjon er det samme som det vi mer folkelig kaller en tabell. Det er derfor en relasjonsdatabase også dersom det bare er en tabell.

## Alternativ betydning (feil, men vanlig misforståelse):

Siden relasjonsdatabaser har sin styrke i muligheten for å relatere mellom relasjoner/tabeller, er det mange som bruker begrepet relasjonsdatabase i betydningen "en database hvor tabeller er relatert". Eventuelt kan man bruke ordet sammenheng om dette.



## For språk-interesserte:

På engelsk er det lettere å skille disse begrepene:

**Relation = relasjon i den egentlige betydningen av ordet**  
**Relationship = sammenheng**

Men: Også engelskmennene er sløve med begrepsbruken her!

## 2.3. Teoretiske begreper - relasjonsdatabaser

**Domene:** Mengde av tillatte verdier

- D<sub>1</sub> f.eks. = lovlig ansattnr
- D<sub>2</sub> f.eks. = lovlig kursnr
- D<sub>3</sub> f.eks. = {1,2,3,4,5,6}
- D<sub>4</sub> f.eks. = {lovlige datoer > 1.1.1900}
- D<sub>5</sub> f.eks. = {lovlige navn}

....

D<sub>n</sub>

D<sub>1</sub> ... D<sub>n</sub> behøver ikke nødvendigvis være forskjellige

**Relasjon:**

Vi lager en funksjon ( $x \in D_1, y \in D_2, z \in D_3, \dots$ )  $\rightarrow \{\text{sant}, \text{usant}\}$ . Elementene ( $x, y, z, \dots$ ) som gjør funksjonen sann kaller vi en relasjon,  $R(x, y, z, \dots)$ . Hvis dette danner en endelig mengde, kan dette skrives på tabellform (som en matrise, jf. kap. 2.2).

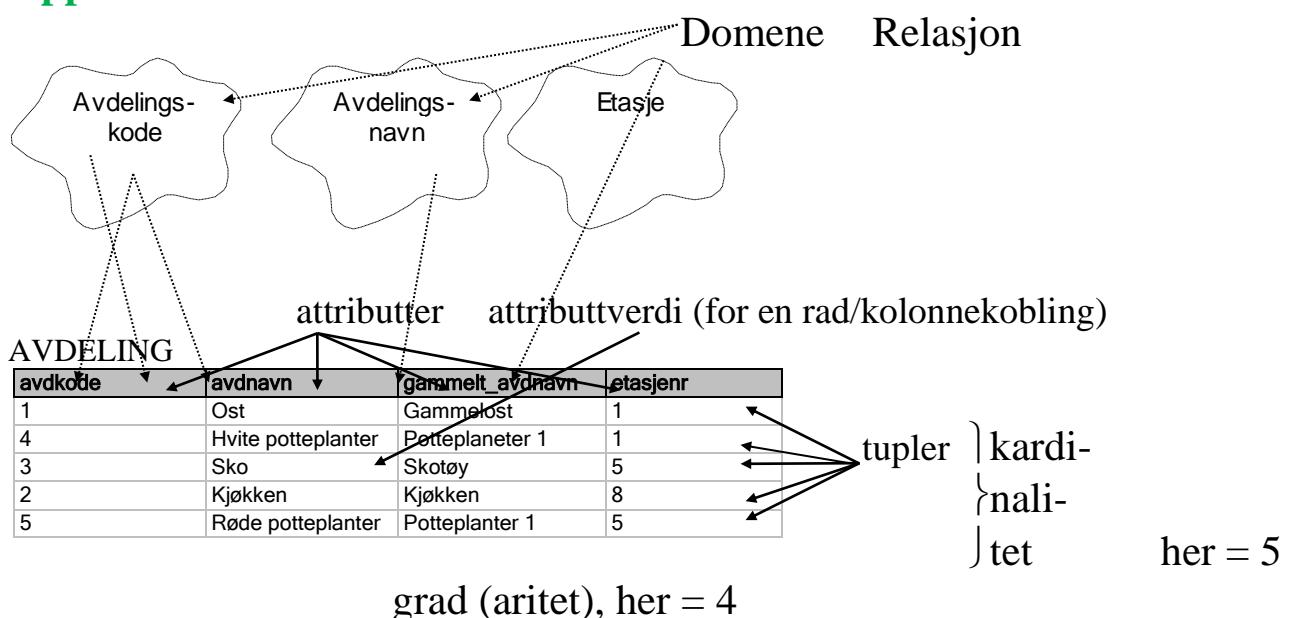
**Attributt:**

En kolonne i en relasjon.

**Tuppel:**

Rad i en relasjon

**Oppsummert:**



## 2.4. En alternativ definisjon av relasjoner

### Det kartesiske produkt av domener.

Gitt domenene  $D_1, D_2, D_3 \dots D_j$ . Det kartesiske produkt  $D_1 \times D_2 \times D_3 \dots \times D_j$  er alle kombinasjoner av mulige verdier for domenene.

### Relasjon - alternativ men ekvivalent definisjon:

En relasjon er en delmengde av det kartesiske produkt av et aktuelt sett med domener, dvs.  $R \subseteq D_1 \times D_2 \times D_3 \dots \times D_j$ .

Hvis f.eks. domenene er

$$\begin{array}{ll} D_1 & \text{f.eks. = lovlige ansattnr} = \{1, 2, 3, \dots, 10\} \\ D_2 & \text{f.eks. = lovlige kursnr} \{1001, 1002, 1003\} \end{array}$$

vil det kartesiske produkt bestå av

$D_1 \times D_2$	
<b>a<sub>1</sub>:D<sub>1</sub></b>	<b>a<sub>2</sub>:D<sub>2</sub></b>
1	1001
1	1002
1	1003
2	1001
2	1002
2	1003
4	1001
..	..
10	1003



Kardinalitet: 30.

Sortering er ikke nødvendig.

Men: det er neppe slik at alle har tatt alle kurs. I så tilfelle vil relasjonen nettopp vise den delmengden av  $D_1 \times D_2$  som har tatt et gitt kurs, f.eks. slik:

1	1002
1	1003
3	1001
3	1002
4	1002
5	1001

Her er det i praksis en delmengde av det kartesiske produkt, dvs.  
 $R \subseteq D_1 \times D_2$ .

Det samme prinsippet kan brukes for alle relasjoner: En relasjon av grad j viser de kombinasjonene av  $D_1 \times D_2 \times D_3 \dots \times D_j$  som er "sanne".

## 2.5. Relasjoner som matematiske objekter

### Hvorfor stresse “teoretiske” definisjoner og begreper?

Først og fremst:

Ved en klar definisjon (vi har langt fra gjort det presist nok!) kan man bruke hele den matematiske viten om relasjoner og mengder. Siden en relasjon essensielt sett er en mengde, kan man f.eks. bruke mengdeoperasjoner som snitt, union, differens etc.

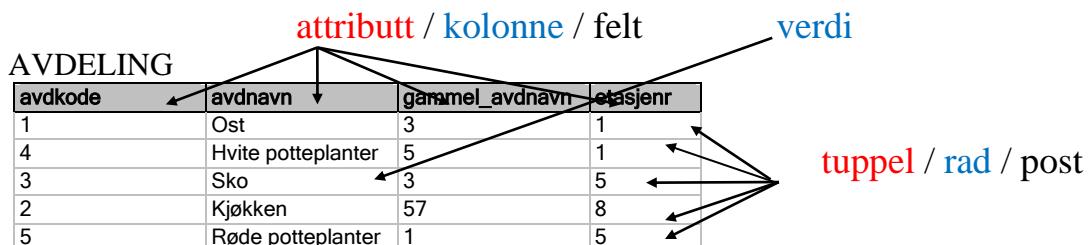
### Andre betegnelser for relasjoner:

Det finnes (minst) tre sett med begreper som brukes om hverandre innen databaser:

- teoretiske begrep fra relasjonsdatabaseteorien: relasjoner, attributter
- begrep som er brukt i vanlige relasjonsdatabasesystemer: tabeller, kolonner
- gamle, tradisjonelle begrep som har overlevd fra filbehandling: fil, felt

At selve relasjonsbegrepet er flertydig, gjør ikke saken bedre:

Altså: ulike begrep for relasjoner / tabeller / filer:



### 3. Integritetsregler for databaser

Det kan tenkes mange integritetsregler for databaser, f.eks. at lovlige kundenr er mellom 1000 og 8000 eller at summen av prosenter for xyz skal være  $\leq 100$ . Disse er imidlertid valgt spesielt for en enkelt database (semantiske integritetsregler).

Derimot er det to generelle integritetsregler som gjelder alle databaser:

- entitetsintegritet
- referanseintegritet

#### 3.1. Entitetsintegritet

Enhver tabell skal ha en primærnøkkel (som er "not null").

Formålet er at alle forekomster (entiteter = "ting") skal kunne identifiseres og ha mening. Hvis ansattnr brukes for å identifisere, vil vi ikke tillate like ansattnr, heller ikke at ansattnr er "ukjent".

**Kravet er altså:**

- alle verdier av det valgte kolonne/kombinasjon må være forskjellige.
- ingen del av en kolonne/kombinasjonen kan være NULL.

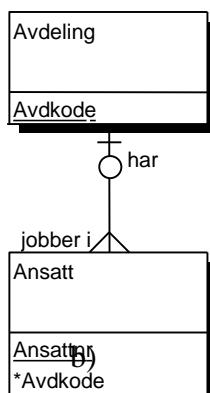
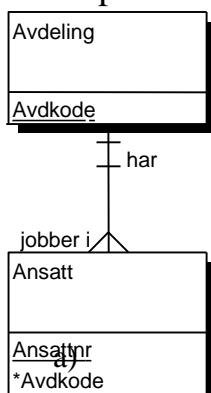
#### 3.2. Referanseintegritet

Hvis to tabeller er sammensatt i et Primærnøkkel - Fremmednøkkel-forhold, må alle verdier av fremmednøkkelen

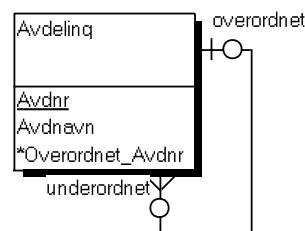
enten: "matche med" en verdi i primærnøkkelen

eller: helt ut være NULL (udefinert, finnes ikke)

Eksempel:



- Merk at hvis primærnøkkelen er sammensatt, må fremmednøkkelen være sammensatt tilsvarende.
- En referanseintegritet kan også gjelde internt i en tabell (som her →): Dette danner et hierarki av avdelinger (avdelinger, underavdelinger, under-under-avdelinger, under-under-under-avdelinger osv.).



## Referanseintegritet: strategi ved innsetting i en fremmednøkkel.

Sjekk at fremmednøkkelen matcher med dens "eier", evt. er NULL.

- Ikke tillatt å sette inn et Avdnr i ansatt som ikke matcher med et avdnr i Avdeling.

- For b): Tillatt å sette inn Ansatt uten avdelingsnr.

Ved endring av en eksisterende fremmednøkkel gjelder det samme for den nye verdien

## Referanseintegritet: strategier ved endring / sletting:

### - Betinget / ingen handling.

Primærnøkkelen kan bare endres/slettes hvis den ikke har noen tilhørende fremmednøkler (bare avdelinger uten ansatte kan slettes).

### - Kaskade:

Ved endring/sletting av primærnøkkelen: Endre/slette alle tilhørende fremmednøkler (hvis en avdeling slettes: slett tilhørende ansatte).

### - Nullstill:

Ved endring/sletting av primærnøkkelen: Sett alle tilhørende fremmednøkler til NULL (hvis en avdeling slettes: alle dens ansatte har ukjent avdnr, dvs. de blir satt til NULL )

### - Egendefinert sjekk/prosedyre:

Kjør en sjekk/prosedyre som gjør visse handlinger på før endring/sletting. (Eks. på mulig prosedyre: Ved sletting av en avdeling spør systemet først hvor de ansatte skal fordeles. Det er da programmererens jobb å sjekke at referanseintegritet blir opprettholdt.<sup>8)</sup>)

- I noen systemer kan man definere at primærnøkkelen ikke kan endres / slettes.

Hva blir konsekvensene med ulike strategier for:

### AVDELING

avdkode	avdnavn	etasjenr	areal
1	Ost	3	100
14	Hvite potteplanter	5	900
3	Sko	3	500
2	Kjøkken	57	1000
999			

### ANSATT

ansattnr	fnavn	enavn	adresse	telefornr	avdkode	lønn
1000	Anders	Andersen	Aveien 1	12345678	3	250000
1001	Bernt	Bertsen	Bveien 1	234567890		
1002	Cesar	Cesarsen	Cveien 1	345678901	1	300000
1003	David	Davidsen	Dveien 1	456789012	1	400000
9834	Edgar	Edgarsen	E	567890123	14	250000

De tre øverste er standardisert i de fleste SQL-baserte systemer, og kalles NO ACTION (evt. RESTRICT), CASCADE og SET NULL. Vi skal senere forklare hvorledes dette brukes i SQL (se kap. 8).

<sup>8</sup> Ofte brukes såkalte triggere for dette. Dette er et program som legges i databasen og «avfyres» dersom det skjer endringer i databasen. Vi tar ikke dette opp videre her, det hører til avansert bruk av databaser.

## 4. Koblingsformer.

Når vi snakker om å koble tabeller, gjøres det gjerne med kriteriet om at data i en tabell må være lik data i en annen tabell. Det finnes likevel varianter av dette. Vi lager en oversikt:

### "Inner join" (indrekobling):

Dette er «vanlig kobling», med krav om at data skal være like mellom primærnøkkelen og fremmednøkkelen («matchende verdier»). I eksempelet over får vi:

Ost	Cesar Cesarsen
Ost	David Davidsen
Hvite potteplanter	Edgar Edgarsen
Sko	Anders Andersen

### "Outer join" (ytrekobling):

Ved utlisting: Alle avdelinger kommer med, sammen med evt. ansatte for hver avdeling.

Ost	Cesar Cesarsen
Ost	David Davidsen
Kjøkken	Navn blir her NULL
Hvite potteplanter	Edgar Edgarsen
Sko	Anders Andersen

Ofte defineres left outer join (alt fra venstre tabell og eventuelle koblende) og right outer join (alt fra høyre tabell og eventuelle koblende).

### "Full join" (full kobling):

Alle på "venstre side", alle på "høyre side" og de som finnes i begge.

### Generelt:

- De fleste moderne relasjonsdatabasesystemer har mulighet for inner og outer join.
- Inner join er gjerne standard.
- Full join finnes ikke i de fleste systemer.

## 5. Relasjonsdatabasesystemer - noen produkter

### Databaseverktøy

- Oracle
  - Microsoft SQL Server (fra Microsoft)
  - MySQL (eies nå av Oracle)
  - MariaSQL (selvstendig videreutvikling av MySQL)
  - Access (fra Microsoft)
  - Sybase
  - DB2 (fra IBM)
  - PostgreSQL (open source)
  - SQLite
  - Informix (kjøpt opp av IBM)
  - InterBase (fra Embarcadero Technologies)
  - Progress
  - Ingres
- Mange flere finnes, se
- [http://en.wikipedia.org/wiki/List\\_of\\_relational\\_database\\_management\\_systems](http://en.wikipedia.org/wiki/List_of_relational_database_management_systems)
- 
- ulike mht. datamengder  
og miljø de er ideelle for

Hvilke operativsystem går de på?

- Microsofts verktøy går kun på Windows
- De andre stort sett på mange plattformer (typisk Windows, Mac og Linux)

### Integrt, eller som tilleggsverktøy:

- Planleggingsverktøy, f.eks. for datamodellering (evt. del av et større verktøy, CASE-verktøy)
- Utviklings/realiseringssverktøy, for å lage applikasjonen
- Mellomvare, f.eks. mellom database og applikasjonsutviklingsverktøy
- Administrasjonsverktøy (brukere, rettigheter)

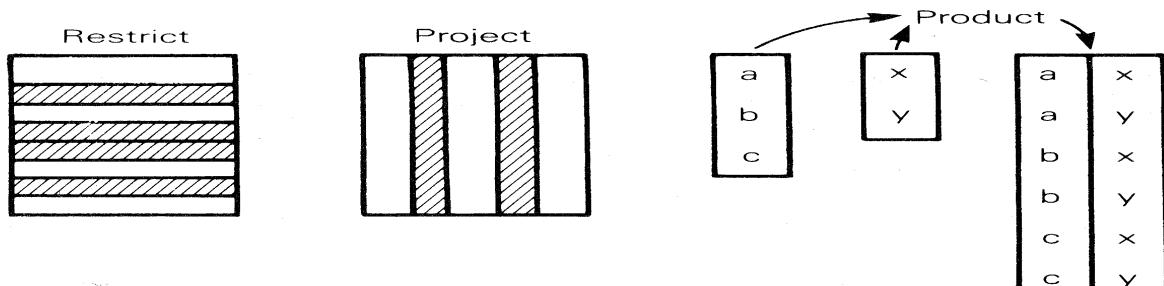
NB! Mange av disse verktøyene er laget av uavhengige leverandører, og fungerer ofte sammen med mange RDBMS-er.

# 6. Relasjonsoperatorer

## 6.1. Utplukk fra relasjoner – kort om 3 relasjonsoperatorer.

SQL (se mer senere) er et ”standardspråk” for spøringer mot en database. Det finnes imidlertid også andre språk som kan brukes eller som er nyttige for en forståelse av prosessen. Et av disse kalles relasjonsalgebra.

Relasjonsalgebra består av noen få (vanligvis 8) operatorer. Vi skal se på noen av dem som en innledning til SQL.



### Restriksjon – horisontalt utvalg:

Eks: Plukk ut ansatte med avdnr. 33 fra relasjonen Ansatt:

$\text{Restrict}_{\text{avdnr} = 33}(\text{ANSATT})$       kortere:       $\sigma_{\text{avdnr} = 33}(\text{ANSATT})$

### Prosjeksjon – vertikalt utvalg:

Eks: Plukk ut ansnr, ansnavn, avdnr fra relasjonen Ansatt:

$\text{Project}_{\text{ansnr}, \text{ansnavn}, \text{avdnr}}(\text{ANSATT})$       kortere:       $\pi_{\text{ansnr}, \text{ansnavn}, \text{avdnr}}(\text{ANSATT})$

### (Kartesisk) produkt - alle i en tabell koblet med alle i en annen

Eks: Kobl alle fra relasjonen Avdeling med alle i relasjonen Ansatt:

$\text{Avdeling product Ansatt}$       kortere:       $\text{Avdeling} \times \text{Ansatt}$

NB! Gir ”alle mot alle”

### Kombinasjon av disse:

Eks: Plukk ut avdnr, avdnavn og ansnavn på alle i avd. som er i etasjenr 3.

$\text{Project}_{\text{avdnr}, \text{avdnavn}, \text{ansnavn}}(\text{Restrict}_{\text{Avdeling.Avdnr} = \text{Ansatt.Avdnr} \text{ and } \text{etasjenr} = 3}(\text{Avdeling product Ansatt}))$

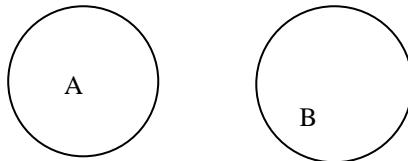
Dermed: project av restrict av produkt. (Dette kan gjøres enklere, men droppes her)

alternativt:  $\pi_{\text{avdnr}, \text{avdnavn}, \text{ansnavn}}(\sigma_{\text{Avdeling.Avdnr} = \text{Ansatt.Avdnr} \text{ and } \text{etasjenr} = 3}(\text{Avdeling} \times \text{Ansatt}))$

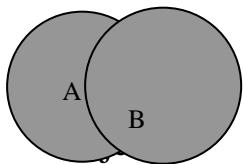
SQL bygger både på relasjonsalgebra og såkalt relasjonskalkyle. En detaljert gjennomgang av dette hører til videregående stoff innen databaser. Kompendium om dette kan fås på forespørsel – dessuten finnes mye på nettet.

## 6.2. Utplukk fra relasjoner – kort om mengdeoperasjoner.

Vi tenker oss at to firmaer, A og B slår seg sammen. De har hver sine relasjoner (tabeller) over kunder, og strukturen i disse tabellene er like. Vi tenker på hver av disse som mengder.

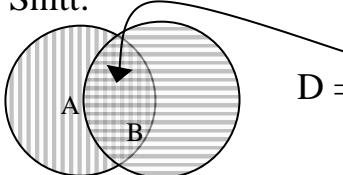


Union:



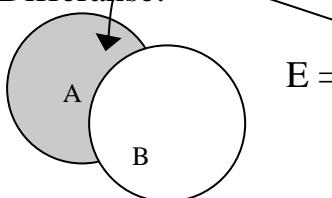
$C = A \cup B$ : de kunder som er med i ett eller begge. Duplikater

Snitt:



$D = A \cap B$ : de kunder som er med i begge.

Differanse:



$E = A \setminus B$ : de kunder som er med i A, men ikke i B.

Tilsvarende kan man definere

- $B \setminus A$ , de kundene som er med i B, men ikke i A, samt
- $A \Delta B$  (symmetrisk differanse), de som ikke er med i begge,  $(A \cup B) \setminus (A \cap B)$

Poenget er altså:

Vi kan bruke mengdeoperasjoner til å plukke ut data vi måtte ønske. Det er også mulig å bruke mengdeoperasjoner til å legge til eller slette tupler.

### **6.3. Enda kortere om andre relasjonsoperatorer**

Vi har ofte brukt for ulike kombinasjoner og spesialiseringer av produkt, projeksjon og restriksjon. Blant disse er

- outer join ("ytterforening")
- $\theta$ -join, eqvi-join, natural join ("naturlig forening"). Den siste er (litt forenklet) en kobling mellom primær- og fremmednøkler, slik at enten fremmednøkkelen(e) tas bort etter koblingen.  
Det er dette som foretas i vanlige koblinger mellom to tabeller.
- semijoin, m.m.
- divisjon av en relasjon/tabell med en annen.

Dette tas ikke opp her (se større lærebøker, nettet eller et eget av Edgars kompendier).

### **6.4. Generelt om relasjonsoperatorer.**

- Relasjonsoperatorene er alltid på mengder (sett) av data

Det betyr at man ikke behøver å behandle et og et tuppel / rad om gangen, det er samme tenkning uansett hvor mange rader som behandles.

Det er nettopp dette som også er typisk for relasjonsdatabaser, og som er en av dets styrker, samtidig som det også har svakheter.

- Det er få operasjoner

Vi arbeider med hele mengden om gangen, og trenger derfor ikke språkelementer som løkker, finn første, neste, siste, kriterium for avslutning osv.

- Operasjoner på relasjoner / tabeller gir relasjoner / tabeller tilbake (operasjonene er "lukket").

# 7. SQL-språket - oversikt

## 7.1. Generelt

### SQL: Structured Query Language.

**Prinsipp:** Du beskriver hva du vil ha som resultat, programmet finner selv hvordan det skal gjøres - mest mulig effektivt.

### Språket består av

- et **datadefinisjonsspråk** for definisjon av struktur (DDL).
- et **datamanipulasjonsspråk** for endring / søking på data (DML).
- et **administrasjon av databasen** (lagring av brukere, tillatelser m.m.)

### Språket er standardisert og videreutviklet i mange versjoner

- SQL1 (SQL89) – vedtatt 1989.
- SQL2 (SQL92) – vedtatt 1992, spesielt standardisering av skranker (bl.a. primærnøkler, fremmednøkler, unike).
- SQL3 (SQL:1999) – vedtatt 1999, nye elementer spesielt standardisering av triggere, objektorienterte utvidelser m.m.
- SQL4 (SQL:2003) – vedtatt i 2003, hvor standarden er delt i en ”kjerne-del” som alle leverandører må ha for å tilfredsstille minimum, og ulike ”pakker”, f.eks. for objektorientering og datavarehus. Litt kobling mot XML.
- SQL:2006 – integrering av XML
- SQL:2008, 2011, 2016.

Det er gitt ut egne standardiseringsdokumenter for språket, utgitt av ANSI, American National Standards Institute, se [www.ansi.org](http://www.ansi.org).

Språket er ”nesten felles” for alle moderne relasjonsdatabaser. Likevel: ulike leverandører har noen

- spesialiteter (f.eks. forskjell på syntaks)
- begrensninger i forhold til standarden(e)
- utvidelser, nye ”features”, men som senere kan bli standardisert i tråd med eller i konflikt med et bestemt produkts utvidelser.

### Vårt fokus:

Vi vil så og si utelukkende konsentrere oss om SQL1 og SQL2, fordi dette er det basale grunnlaget. De nyere utgavene beskriver avanserte utvidelser som bygger på SQL2. Mange av de nye utvidelsene (f.eks. objektorienterte utvidelser) er også til dels komplikst, lite brukt i praksis og/eller dårlig implementert i dagens systemer.

## 7.2. DDL – datadefinisjonsspråket – kort oversikt

### Oppretting av tabeller:

```
CREATE TABLE avdeling  
(avdnr integer NOT NULL,  
avdnavn varchar(20),  
add constraint ansatt_pk PRIMARY KEY (avdnr));
```

navnet her må ofte være entydig  
noen systemer krever navngivning av skranker

### Endring av tabeller: alter table avdeling

```
add column avdelings_sted varchar(20);
```

```
alter table ansatt  
add constraint ansatt_fk  
foreign key (avdnr)  
references avdeling (avdnr);
```

ordet column er ikke med i alle systemer

```
alter table ansatt drop  
column avdelings_sted;  
- finnes ikke i alle systemer.
```

### Sletting av tabeller:

```
drop table ansatt;
```

I noen systemer må tabeller som refererer til tabellen fjernes først.

### Laging/sletting av utsnitt (views):

```
create view oslo_avdeling as  
select * from avdeling  
where avdelings_sted = 'Oslo';
```

```
drop view oslo_avdeling;
```

### 7.3. DML - data-manipulasjons-språket - kort oversikt

Eksempel på spørring med to tabeller:

```
SELECT avdeling.avdnr, avdnavn,  
ansattnr, fnavn, enavn  
FROM avdeling, ansatt  
WHERE avdeling.avdnr = ansatt.avdnr
```

Resultat-eksempel:

Avdnr	Avdnavn	Ansattnr	Fnavn	Enavn
2	Sko	431	Hans	Bø
1	Ost	345	Ole	Hansen

osv.

Eksempel på spørring med gruppering:

```
SELECT Avdnavn, Count(*)  
FROM avdeling, ansatt  
WHERE avdeling.avdnr = ansatt.avdnr  
GROUP BY avdeling.avdnavn;
```

Resultat-eksempel:

Avdnavn	count(*)
Sko	3

osv.

Innsetting: `INSERT INTO avdeling VALUES (3,'Klær');`

Sletting: `DELETE FROM ansatt WHERE ansattnr = 431;`

Endring: `UPDATE avdeling  
SET avdnavn = 'Sko & lærvarer'  
WHERE avdnr = 2;`

## 8. Datadefinisjon

Merk: Les først det generelle om primær- og fremmednøkler og om andre unike/kandidatnøkler (kap. 2.1 og 3).

Når vi bruker begrepet nøkler her, er det som samlebetegnelse på disse tre strukturene.

Temaet her er hvordan vi lager tabeller i SQL, for dermed å lage en tom struktur som vi kan fylle med data. De 3 setningene som brukes er:

- Lage nye tabeller:  
CREATE TABLE <tabellnavn> ( <kolonne- og strukturdefinisjon>.....)
- Endre eksisterende tabeller:  
ALTER TABLE ADD / DROP <kolonner eller struktur>
- Slette eksisterende tabeller:  
DROP TABLE <tabellnavn>  
Merk: her slettes både tabellene og innholdet (dataene i disse)

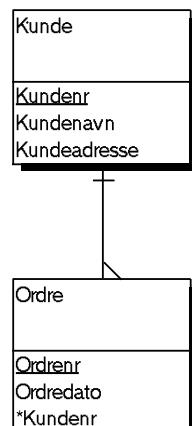
Vi setter nøkkelordene med store bokstaver, selv om det normalt ikke spiller noen rolle i SQL.

### 8.1. *Create table-setninger, kun til bruk når det ikke er sammensatte nøkler.*

Merk: det anbefales å bruke den mer generelle syntaksen (kap. 8.2 og 8.3). Vi tenker oss et system med kunder og ordrer. Slike systemer er naturligvis veldig mye mer kompliserte enn det vi viser her!

#### Tabell med enkel primærnøkkel.

```
create table kunde (kundenr integer NOT NULL PRIMARY KEY,  
                    kundenavn varchar(20) NOT NULL,  
                    kundeadresse varchar(50));9
```



<sup>9</sup> I de fleste systemer er not null unødvendig, når noe defineres som PRIMARY KEY. Systemet setter som regel dette inn uansett.

## Tabell med enkel primær- og fremmednøkkel.

Vi forutsetter at vi allerede har laget tabellen kunde (se over). I tillegg til å lage Ordretabellen, vil vi ha en fremmednøkkel som refererer til («henter verdier fra»)<sup>10</sup> Kunde sitt kundenr.

```
create table ordre (ordrenr integer NOT NULL PRIMARY KEY,  
                    ordredato date,  
                    kundenr integer NOT NULL REFERENCES kunde(kundenr) );
```

## Tabell med unik.

Anta at vi vil ha et krav om at kundenavn også er unikt, dvs. at man ikke kan registrere to like kundenavn.

Vi utvider da til

```
create table kunde (kundenr integer not null PRIMARY KEY,  
                    kundenavn varchar(20) NOT NULL UNIQUE,  
                    kundeadresse varchar(50) );
```

### 8.2. *Create table-setninger, generelt.*

Det anbefales å bruke denne skriveformen som vi viser her, eller formen i kap. 8.4, fordi den er anvendelig både når vi har enkle og sammensatte nøkler. Dermed holder det å ha en type syntaks å lære.

Merk:

- En enkel (ikke-sammensatt) nøkkel består bare av en kolonne.
- En sammensatt nøkkel (primær-, fremmed eller unik) består av to eller flere kolonner som til sammen utgjør nøkkelen.

En sammensatt nøkkel er altså ikke knyttet til én bestemt kolonne, og vi må derfor sette den for seg selv.

---

<sup>10</sup> Det er fristende å si «peker på», men saken er at det ikke er en peker, slik vi vanligvis bruker dette begrepet i programmeringsspråk. Det er derimot data som kan ha like verdier, og som man dermed kan koble på dette vilkåret.

## Tabell med sammensatt primærnøkkel

Tenk deg at vi har kunder i ulike land, og kundenr starter på 1 for hvert land. Da må vi ha kombinasjonen (landkode,kundenr) som primærnøkkel. Definisjonen kan da skrives:

```
create table kunde (    landkode varchar(2) NOT NULL,  
                        kundenr integer NOT NULL,  
                        kundenavn varchar(20) NOT NULL,  
                        kundeadresse varchar(50),  
                        PRIMARY KEY (landkode,kundenr) );
```



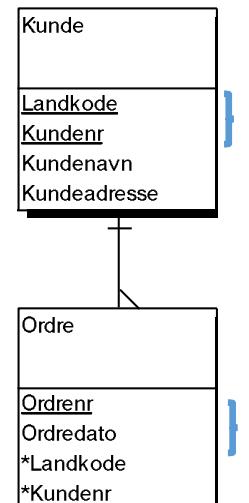
Merk forskjellen: Hver tabell kan ha bare en primærnøkkel, men denne kan være sammensatt av flere kolonner. Vi klargjør dette med følgende oppsett:

Feil, <u>det snakkes om at en tabell har to primærnøkler</u>	Riktig, en tabell kan bare ha en primærnøkkel, men det kan være <u>en sammensatt primærnøkkel</u>
Landkode varchar(2) not null primary key, Kundenr integer not null primary key,  (her er det jo to primærnøkler, det er ikke lov)	Landkode varchar(2) not null, Kundenr integer not null, ..... primary key (landkode, kundenr),

## Tabell med sammensatt fremmednøkkel.

Tenk en utvidet Kundetabell hvor vi har (landkode,kundenr) som primærnøkkel. Da vil ordretabellen referere til den sammensatte primærnøkkelen (landkode,kundenr).

```
create table ordre (ordrenr integer NOT NULL,  
                    ordredato date,  
                    landkode varchar(2) NOT NULL,  
                    kundenr integer NOT NULL,  
                    PRIMARY KEY (ordrenr),  
                    FOREIGN KEY (landkode,kundenr) REFERENCES kunde (landkode,kundenr) );
```



NB: Kolonnene på fremmednøkkelsiden må stemme fullt overens med de på primærnøkkelsiden, både når det gjelder rekkefølge og hvilken datatype de har.

## Fremmednøkler og andre endringer i egne setninger.

Det er lurt å legge fremmednøkler som egen setning ved bruk av ALTER TABLE-setningen, f.eks. slik:

```
create table ordre ( ordrenr integer not null,
                     ordredato date,
                     landkode varchar(2),
                     kundenr integer not null,
                     kontaktperson varchar(30),
                     PRIMARY KEY (ordrenr));
```

```
ALTER TABLE ordre ADD FOREIGN KEY (landkode, kundenr) REFERENCES
                      kunde (landkode, kundenr);
```

Merk altså at vi først definerer landkode og kundenr som ”vanlige” kolonner, deretter setter vi en betingelse / skranke / begrensning på disse, nemlig at kombinasjonen av disse skal være en fremmednøkkel som refererer opp til primærnøkkelen.

Vi anbefaler at man først lager alle tabellene (ink. primærnøkler og eventuelle unike), deretter lager man alle alter-table-setningene. Altså:

```
CREATE TABLE ....          (med primærnøkler og evt. andre unike)
CREATE TABLE ....
CREATE TABLE ....
```

```
ALTER TABLE ..... ADD FOREIGN KEY ....
ALTER TABLE ..... ADD FOREIGN KEY ....
```

Grunnen at vi anbefaler å ha alle CREATE TABLE-setningene i starten, er at en tabell, f.eks. kunde må være laget før du kan referere til den i form av en fremmednøkkel (f.eks. fra Ordre til Kunde). Dette vil altså ikke fungere:

- 1) CREATE TABLE Ordre (..... , Kundenr references Kunde(kundenr))
- 2) CREATE TABLE Kunde (Kundenr .....)

Det er også lurer å splitte slike setninger i to av en annen grunn, nemlig for å få to enkle setninger i stedet for en mer komplisert. Dermed er det lettere avsløre en eventuell feil i setningen/e.

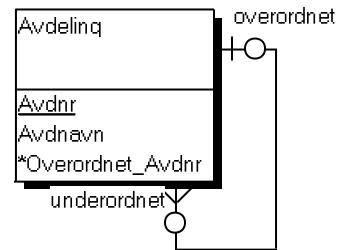
Vi kan også gjøre andre endringer i tabellen via ALTER TABLE, f.eks. hvis vi ønsker å legge til en ny kolonne, slik:

```
ALTER TABLE kunde ADD COLUMN omsetning integer;
```

I en slik setning kan man også legge til UNIQUE, NOT NULL o.l.

## Spesialtilfelle, hvor man MÅ ha fremmednøkkel i en egen setning.

I et spesielt tilfelle MÅ vi ha fremmednøkkelen i en egen setning, nemlig når vi har en «intern fremmednøkkel», f.eks. med avdelinger med under-avdelinger, under-under-avdelinger, osv. i vilkårlig mange nivåer.



Create table

```
avdeling (avdnr integer, avdnavn varchar(20), overordnet_avdnr integer,
```

```
PRIMARY KEY (avdnr),
```

```
FOREIGN KEY (over_avdnr) REFERENCES avdeling (avdnr));
```

fungerer ikke, fordi den refererer til seg selv, men den er jo ikke laget enda!

Vi må altså ha rekkefølgen

1) CREATE TABLE

```
avdeling (avdnr integer, avdnavn varchar(20), overordnet_avdnr integer,
```

```
PRIMARY KEY (avdnr));
```

2) ALTER TABLE ADD

```
FOREIGN KEY (overordnet_avdnr) REFERENCES avdeling (avdnr));
```

## Tabell med sammensett unik.

Vi antar først at kundenavnene fremdeles må være helt unike, da får vi som før:

```
create table kunde ( landkode varchar(2) NOT NULL,
                      kundenr integer NOT NULL,
                      kundenavn varchar(20) NOT NULL UNIQUE,
                      kundeadresse varchar(50),
PRIMARY KEY (landkode,kundenr) );
```

Men, **hva hvis kundenavnene bare skal være unike for hvert land**, f.eks. at vi kan ha firmaet Borregaard bare en gang i Norge, men at det også må kunne finnes en gang f.eks. i Danmark og i andre land. Da må kombinasjonen (landkode, kundenavn) være unik. Dermed må vi skrive:

```
create table kunde ( landkode varchar(2) NOT NULL,
                      kundenr integer NOT NULL,
                      kundenavn varchar(20) NOT NULL,
                      kundeadresse varchar(50),
PRIMARY KEY (landkode,kundenr),
UNIQUE (landkode, kundenavn) );
```

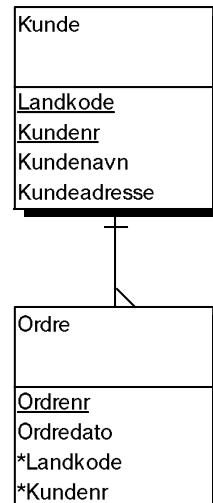
### 8.3. Handling ved innsetting / endring / sletting av fremmednøkler

I kap. 3.2 tok vi opp referanseintegritet, for å sørge for at det ikke blir uoverensstemmelser / inkonsistenser vi endrer databasen. Det bør f.eks. ikke være mulig å slette data om en kunde som har en eller flere ordrer i databasen – evt. bør alle dens ordrer også slettes. Det bør heller ikke være lovlig å sette inn en ordre på en kunde som ikke finnes fra før av.

Noen data som gjør at det er lettere å tenke seg hva som skjer hvis du f.eks. ønsker å slette eller endre en rad i Kunde:

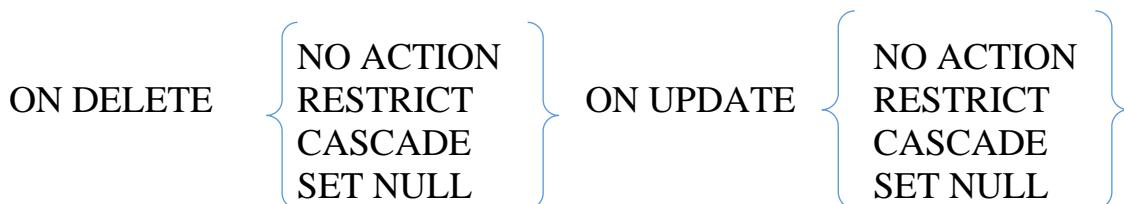
KUNDE		
Landkode	Kundenr	....
N	1	
N	2	
DK	1	

ORDRE		
Ordrenr	Landkode	Kundenr
1001	N	1
1002	DK	1
1003	N	1



Vi tok opp  
NO ACTION / RESTRICT<sup>11</sup>  
CASCADE  
SET NULL  
og egendefinerte prosedyrer.

De fleste databasesystemer har mulighet for å definere opp dette, i større eller mindre grad. Dette gjøres ved å legge til en av disse i fremmednøkkelsenhetningen med hensyn til sletting og oppdatering:



I SQL ser dette f.eks. slik ut:

ALTER TABLE ordre ADD FOREIGN KEY (landkode, kundenr) REFERENCES kunde (landkode,kundenr) ON DELETE NO ACTION ON UPDATE CASCADE;

Dette er antagelig det vanligste valget, og betyr:

- **For Kunde:**

Sletting av en rad: bare tillatt hvis det ikke finnes noen ordre for denne kunden.  
Endring av (Landkode, Kundenr): tillatt, og fører til tilsvarende endringer i Ordre.

<sup>11</sup> Det er en hårfin forskjell mellom disse to: Ved NO ACTION sjekkes referanseintegriteten når endringen er ferdig, mens RESTRICT sjekker på forhånd om dette vil gå bra under hele endringsprosessen. NO ACTION er enklest og raskest.

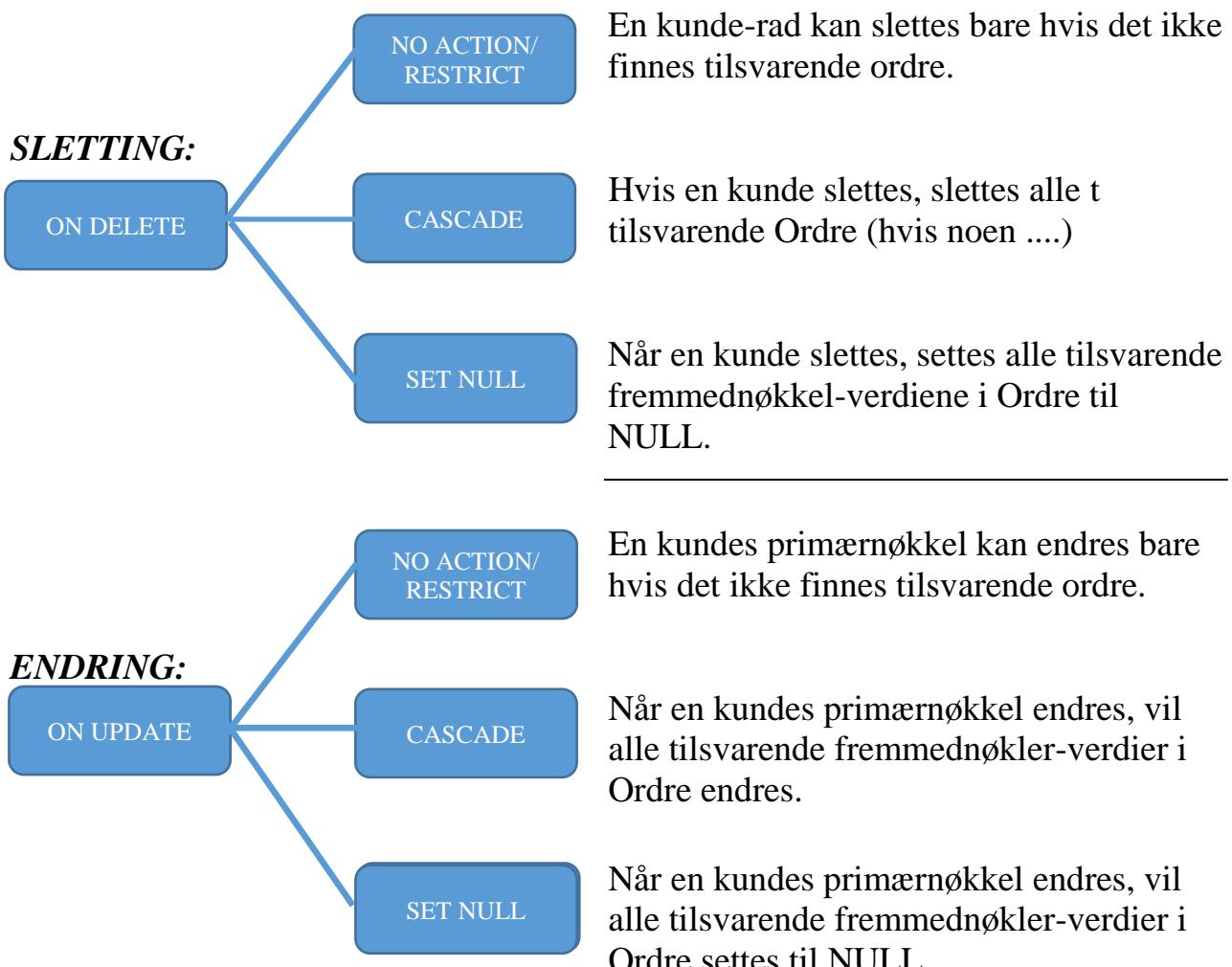
- **For Ordre:**

Innsetting: Må sette inn en (Landkode,Kundenr) som allerede finnes i Kunde (evt. må begge deler være NULL).

Endring av (Landkode,Kundenr): Må endre til en (Landkode,Kundenr) som allerede finnes i Kunde (evt. må begge deler være NULL).

**En fullstendig liste over primær/fremmednøkkelhandlinger kommer her:**

**FOR ALLE ENDRINGS/  
SLETTINGSMÅTER:**



Hvis denne setningsdelen utelates, blir det NOT ACTION (evt. RESTRICT) for sletting og endring. Det er jo det som er det mest restriktive, og dermed mest "safe".

## 8.4. Navngivning av nøkler.

Opplysning om en nøkkel blir jo lagret i databasen, og det kan derfor være lurt å kunne gi den et eget navn, f.eks. i tilfelle den skal endres eller slettes. Slike opplysninger er eksempler på en begrensende struktur eller en skranke. På engelsk kalles en skranke for en CONSTRAINT.

Vi utvider syntaksen med å legge et fritt valgt navn for hver slik skranke.

Her har vi valgt å skrive PK, FK og UN henholdsvis til slutt i skrankenavnene, dermed er de lettere å finne igjen, men det er helt opp til dere.

Vi lager en fullstendig beskrivelse med for både Kunde og Ordre-tabellen:

```
create table kunde (      landkode varchar(2) NOT NULL,  
                          kundenr integer NOT NULL,  
                          kundenavn varchar(20) NOT NULL,  
                          kundeadresse varchar(50),  
CONSTRAINT KundePK PRIMARY KEY (landkode,kundenr),  
CONSTRAINT KodeOgNavnUN UNIQUE (landkode, kundenavn));
```

```
create table ordre ( ordrenr integer not null,  
                     ordredato date,  
                     landkode varchar(2),  
                     kundenr integer not null,  
                     kontaktperson varchar(30),  
CONSTRAINT OrdrePK PRIMARY KEY (ordrenr));
```

```
ALTER TABLE ordre ADD CONSTRAINT OrdreKundeFK  
FOREIGN KEY (landkode, kundenr) REFERENCES kunde (landkode, kundenr);
```

Dette er altså den mest omfattende måten å lage primær- fremmednøkkel og unik-skranke på, og også den som anbefales, fordi den kan brukes i alle tilfeller.

Noen systemer krever at man har en slik CONSTRAINT-navngivning, men for de fleste systemene er det frivillig. Hvis vi ikke lager egne skrankenavn, vil database-systemet selv lage slike, og i noen tilfelle er det bare snakk om en teller eller et annet uforståelig navn.

I et større system finnes det selvsagt mange slike skranker. En av fordelene med å navngi en skranke, er at dersom man f.eks. skal slette en skranke, så man vite hvilken. Vi kan da henvise direkte til navnet, f.eks.

i noen systemer: ALTER TABLE ordre DROP FOREIGN KEY OrdreKundeFK;  
i andre systemer: ALTER TABLE ordre DROP CONSTRAINT OrdreKundeFK;

## 8.5. SQL: generelt om skranker i tabeller

Det finnes også andre typer skranker / begrensninger / regler som kan være nyttig å legge i databasen. NB: Dette er ofte likt ulikt fra et databasesystem til et annet. Eksempelvis er det mulig å skrive check-setninger i MySQL, men systemet tar ikke hensyn til det. ASSERTION-setninger finnes ikke, heller ikke i flere av de største databasesystemene. Dette fast derfor bare med som eksempler på ting som kan finnes i noen systemer, men ikke i alle. Mye av dette som mangler her, kan implementeres via såkalte triggere.

Strengt tatt er f.eks. en NOT NULL-beskrivelse en skranke, du MÅ ha en verdi i denne kolonnen.

### Ulike sjekker

Syntaks: CHECK ( <betingelse>); Hvis betingelsen gjelder bare en kolonne, kan man sette den direkte på denne kolonnen, ellers gjør man det til slutt i setningen eller som en egen ALTER TABLE ADD CHECK .....

Noen eksempler:

kjonn CHECK (kjonn in ['m', 'k', 'M', 'K'])

CHECK (antall >= 0)

CHECK (timelonn is null or maanedslonn is null)

### Antagelser / generelle skranker

Dette brukes for felles betingelser som det ikke er naturlig å knytte til noen bestemt tabell.

CREATE ASSERTION <navn>

CHECK(<betingelse>)

### Standard-verdier – er ikke egentlig skranke, men praktisk!

Dersom samme verdi brukes i de fleste tilfelle, kan man sette denne som en standard / default-verdi, slik at det blir denne verdien som blir satt inn hvis man ikke gjør noe annet.

Syntaks: DEFAULT verdi, f.eks.

```
create table reise (reisenr integer,
                    antall_personer integer default 2,
                    bestilt_via varchar(20) default 'Internett',
                    primary key (reisenr))
```

## 8.6. Datatyper

Datatyper, f.eks. heltall, dato gir begrensninger på verdier som kolonnen kan ha, og er dermed også en form for skranke.

**Vanlige datatyper (sjekk aktuelt databasesystem for detaljer):**

varchar(n)	tekst, lengde – dvs. n må være kontret tall	Max lengde n tegn Ofte må n ≤ 256 tegn Noen systemer har text(n) eller andre varianter, evt. lengre tekster, notat el.l.
integer, long	heltall	4 bytes (vanligvis), kalles i noen systemer for int.
smallint, short	heltall	2 bytes
byte	heltall	1 byte. Kalles i noen systemer for tinyint
boolean, logical, binary		Ja/nei, sant/usant, etc. 1bit (men bruker ofte 1 byte på disk)
numeric(n,m)	desimaltall	totalt , etter komma
real, single	reelt tall	4 bytes (vanligvis)
double precision, double	reelt tall	8 bytes
date, datetime		inneholder ofte også tidspunkt
oleobject, image, raw	generelt objekt hvor det kan lagres ”alt mulig”	Kalles ofte for en BLOB (Binary Large Object). Må tolkes av et annet program, f.eks. tekstbehandler, bildeprogram e.l.

## Skranker via domenedefinisjon

I noen systemer kan man lage egne domener, som «utvidede datatyper»; f.eks. CREATE TYPE kundenr as integer

CHECK (value > 0);

CREATE DOMAIN kjonn as CHAR

CHECK (VALUE IN ['m', 'k', 'M', 'K']);  
eller alternativ syntaks.

## 8.7. Endring / sletting av tabeller

Vi har tidligere sett på ALTER TABLE ADD ... i forbindelse med fremmednøkler. Dette kan også gjøre med kolonner. Man kan også ta bort kolonner eller f.eks. fremmednøkler. I tillegg finnes en farlig kommando:

DROP TABLE <tabellnavn>;

En enkelt kommando kan slette millioner og rader i tillegg til å slette selve strukturen!

Tilsvarende finnes ALTER TABLE DROP for få fjerne kolonner, skranker m.m.

# 9. Datamanipulasjon

Denne delen gir oversikt over de fleste språkelementer i SQL, basert på en rekke eksempler.

## Oversikt

De mest brukte språkelementene er

SELECT <kommaseparert kolonneliste, evt. med beregninger>

FROM <kommasep. tabelliste> -- evt. også med faste koblinger

WHERE <betingelser med not/og/eller> -- evt. med koblinger

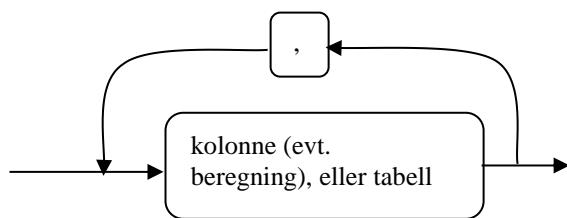
ORDER BY <kommaseparert kol.liste evt. med beregninger , evt med asc eller desc>

- De eneste som MÅ være med er select
  - Hvert av disse språkelementene skal være med bare **en gang i hver setning**.
- 

En kommaseparert liste består av en eller flere elementer, med komma mellom, f.eks.

- select kundenavn
  - select kundenr, kundenavn
  - from kunde, ordre
  - order by etternavn, fornavn
- 

Annerledes sagt:



## 9.1. Enkle spørninger med en tabell

### Noen grunnelementer

Select \*  
from kunde

\* betyr at alle kolonner skal med

```
select kundenr, kundenavn, postnr  
from kunde  
where kundenr > 1000;
```

Tilsvarende (<, ≤ osv.):  
For ≠ kan man skrive:

kundenr < 1000, kundenr <= 1000, kundenr = 1000  
kundenr <> 1000 eller kundenr != 1000

select kundenr, kundenavn, adresse, omsetning  
from kunde  
where omsetning > 100000  
order by postnr, kundenavn;

Kommaseparert  
kolonneliste, evt.  
med beregninger

select kundenr, kundenavn, adresse, round(omsetning / 1000,1)<sup>12</sup>  
from kunde  
order by omsetning desc;

Skriv ut omsetningen i antall 1000,  
avrundet med en desimal.  
Det kan et mer komplisert uttrykk, og  
med flere kolonner, f.eks. pris \* antall.

SELECT Kundenr, Kundenavn, Adresse as Adr,  
Round(omsetning/1000,1), ' i NOK' AS NOK  
FROM kunde  
ORDER BY kunde.Omsetning DESC;

asc – stigende  
desc – minkende  
asc er default

Navngivning av en  
kolonne

Strengkonstanter som del av en  
spørring. NB! Vanlige fnutter,  
ikke fra tekstbehandleren.

<sup>12</sup> Funksjonen round (<tall>, <desimaler>) avrunder <tall> til <desimaler> desimaler. F.eks. er round(3.14159,2)= 3.14. I noen systemer vil man få heltallsdivisjon hvis både dividend og divisor er heltall. Er det én tallet en konstant, kan man gjøre det om til desimaltall ved å skrive f.eks. 100.0 i stedet for 100. Er begge variable, kan man skrive f.eks. 1.0\*omsetning for å omforme det til et desimaltall (et eksempel på det som kalles typecasting).

## DISTINCT: fjern duplikater

```
select distinct postnr  
from kunde;
```

List opp de postnr som er i bruk i tabellen, men bare en gang pr. postnr.

## NULL: ingenting / finnes ikke

At noe er NULL betyr at noe mangler eller ikke finnes. En verdi som ikke finnes er NULL, en verdi som finnes er ikke NULL.

```
select *  
from kunde  
where telefonnr is null;
```

List ut de som ikke har oppgitt telefonnr.

NB! Det er ingenting som = NULL.  
==> telefonnr = NULL er feil.  
==> telefonnr <> NULL er også feil  
==> NULL = NULL er også feil  
==> NULL <> NULL er også feil

```
select *  
from kunde  
where telefonnr is not null;
```

List ut de som har oppgitt telefonnr.

( Sidespor: skjermdump av feilmelding

The screenshot shows a browser window with the URL <https://telenormobil.no/minesider/efaktura.do?invno=5507853166036&Signature=%3CMSG%2B6%2BSigData>. A red box highlights an error message: "Mangler signatur. Sesjonen på innloggingen har bare en viss varighet. Forsök å oppdatere siden (F5) for å utvide sesjonen, eller log inn på nytt." Below the message is a table of session variables:

ApplicationName	null
MerchantName	null
Accountid	null
QueryString	path=subscription.jsp channel=2008 userId= periode=35
RemoteAddr	46.9.33.15
efaktura-web	

Manglende verdier. Dette burde imidlertid ikke vært vist i brukergrensenettet! )

## Jokernotasjon, bl.a. delstrenger

```
select *  
from kunde  
where kundenavn = 'Biltema'
```

List ut den / de som har kundenavn nøyaktig lik Biltema.  
De fleste systemer er case-insensitive.

```
select *  
from kunde  
where kundenavn like '%Bil%'13
```

List ut kundenavn som begynner på Bil.  
Jokertegnet kan også brukes foran, f.eks.  
vil betingelsen  
like '%Bil%'  
gi alle som inneholder teksten bil.

\_ -tegnet brukes for ett virkårlig tegn.

NB: noen systemer (bl.a. Access) bruker \* (0 eller flere) og ? (1 tegn) som jokertegn i stedet

## Between

Dersom vi skal uttrykke at f.eks. kundenr skal ligge mellom 1 og 1000, kan det skrives:

```
select *  
from kunde  
where kundenr between 1 and 100;
```

Tilsvarende

.... where postnr between '1500' and '1599'

.... where kundenavn between 'M' and 'P'

NB! Paraplybutikken kommer ikke med i resultatet!

I stedet for between kan man bruke den logiske operatoren AND, f.eks.

```
select *  
from kunde  
where kundenr >= 1 and kundenr <= 100;
```

To kommentarer:

<sup>13</sup> Hva hvis vi ønsker å søke etter en streng som inneholder tegnet %? Vi bruker da #-tegnet for å si at neste tegn skal tolkes bokstavelig, ikke som et jokertegn. Eks: leter etter en streng som ender på %. Dette kan uttrykkes som like '%#%'.

- Det er viktig å skille mellom to ulike bruk av AND
  - den logiske operatoren AND, med <logisk uttrykk> AND <logisk uttrykk> .... – neste underkap.
  - BETWEEN <startverdi> AND <sluttverdi>
- Between behandles noe ulikt mellom de ulike databasesystemer. Forskjellene går på
  - om startverdien tas med eller ikke, og om sluttverdien tas med eller ikke.
  - hvilke datatyper man kan bruke. (Tall: alltid OK. Strenger, datoer m.m.: fungerer ikke i alle systemer).

## 9.2. Spørninger med logiske operatorer.

Enkle spørninger kan settes sammen til komplekse spørninger med mange deler med not/and/or/xor m.fl. mellom. Dette er helt likt matematisk logikk:

- Operatorprioritet: not, and, or
- Parenteser brukes for å danne uttrykk som skal behandles som en enhet.

## Et par sidesprang

Søk

---

[Logisk søk \(boolsk\)](#)

[Tidsskrift](#)

[Ledige stillinger](#)

[Kurskatalogen](#)

**Logisk søk (boolsk)**

Boolsk, eller logisk søk betyr at brukeren kan skrive et komplisert uttrykk med ord han ønsker skal være med eller ikke være med i artikkelen han vil ha tak i, adskilt av logiske (boolske) operatorer og parenteser.

---

De tillatte operatorene her er: **AND, OR og NOT**.

Ved logisk søk kan man lage kompleks kombinasjoner av dette. Noen eksempler:

- "lege AND sykehus OR blindtarm" vil gi treff på alle artikler som inneholder ordene "lege" og "sykehus", men også på nettsider som inneholder ordet "blindtarm".
- "lege AND ( sykehus OR blindtarm )" vil gi treff på alle sider som inneholder ordet "lege" og i tillegg "sykehus" eller "blindtarm".
- "lege AND sykehus AND NOT blindtarm" vil gi treff på alle sider som inneholder ordene "lege" og "sykehus" men ikke de som samtidig inneholder "blindtarm".
- "( lege OR søker ) AND NOT ( sykehus OR blindtarm )" vil gi treff på alle artikler som inneholder ordet "lege" eller "søker" men som samtidig ikke inneholder ordene "sykehus" eller "blindtarm".

Det er også regler for hvilken rekkefølge operatorene blir utført i. Rekkefølgen er: NOT, AND, OR. Parenteser kan brukes til å endre denne rekkefølgen ved at det som står inne i en parentes utføres først. Se forskjellen på de to første eksemplene. Det første eksempelet kunne også vært skrevet som: "( lege AND sykehus ) OR blindtarm", men dette er unødvendig fordi AND utføres før OR.

Utrykkene kan lages mye mer komplisert enn dette, også med flere parentesnivåer, men i praksis vil 3-4 ord med operatører og parenteser være det mest man bruker. Søketiden vil også øke hvis man lager for kompliserte uttrykk.

AND, OR og NOT kan også skrives med små bokstaver.

Når brukeren starter søket vil det bli sjekket om det som brukeren har skrevet inn er et lovlig logisk uttrykk, og det vil bli gitt feilmelding hvis det ikke er lovlig.

Også leger trenger logiske operatorer! Fra <http://www.legeforeningen.no/id/10351> (febr. 2011). (Artikkelen er kun tatt med for å vise at og/eller m.m. kommer igjen i mange sammenhenger. )

Se tilsvarende mht. søker f.eks. etter biler eller eiendommer hos [www.finn.no](http://www.finn.no).

Tilsvarende finner vi f.eks. i CQL-språket (Contextual Query Language):

The screenshot shows the BIBSYS Ask search interface. At the top, there's a green header bar with the BIBSYS logo and the word "Ask". Below the header, a dark blue navigation bar has tabs for "Søkekilder", "Søk", "Treffliste", "Dokument", "Samlekurv", "Søkehistorikk", and "Min side". The "Søk" tab is active. A user profile "Edgar Boström (n00)" is shown on the right. The main area has three tabs: "Enkelt søker" (selected), "Avansert søker", and "Spesialsøk". Below these tabs, a message says "Du søker i: Bibliotekbasen". To the right of the message is a dropdown menu containing the following items:

- Agrovoc
- Antall instrumenter
- Antall roller, kvinnelige
- Antall roller, mannlige
- Antall roller, totalt
- Autoritetsid
- Avdeling
- Avdelingsamling
- Besetning
- Bibkode

Below the dropdown is a note: "Det kreves at du skriver en gyldig CQL-streng for å få utført søkeret. Søkefeltene hentes fra listen til høyre. F.eks.: forfatter = "Ibsen, Henrik" AND tittel = "Peer Gynt" besetning ALL "orkester Hardingfele" (besetning = orkester AND besetning = Hardingfele) dewey ANY "004? 005? 006?" (dewey = "004?" OR dewey = "005?" OR dewey = "006?")". At the bottom are two buttons: "Søk" and "Nullstill".

## Tilbake til SQL igjen:

I SQL må hver del av en betingelse være et logisk utsagn. Eksemplene i artikkelen må dermed omformes for å fungere, f.eks. til yrke = "Lege" and sykdom = "Blindtarm".

### Noen eksempler:

select kundenr, kundenavn, adresse  
from kunde

where (kundenr > 1000 and (postnr < '2000' or postnr > '3000')) or postnr > '9000';

Kan ha vilkårlig komplekse boolske uttrykk her (dvs. som resulterer i sant eller ikke sant)

Bruk fnutter ( ' ) dersom kolonnen er definert som en streng. Noen systemer bruker dobbelfnutter ( " ) i stedet, eller tillater begge.  
NB!  
Pass på riktig fnuttype, " er f.eks. ikke tillatt.

*Skriv ut omsetning for kundene 'Biltema' og 'Plantasjen' og kunder som har omsetning mer enn 100000.*

```
select kundenr, kundenavn, adresse  
from kunde  
where kundenavn = 'Biltema' or kundenavn = 'Plantasjen' or omsetning > 1000000;
```

*Skriv ut omsetning for kundene 'Biltema' og 'Plantasjen', hvis de har omsetning mer enn 1000000.*

```
select kundenr, kundenavn, adresse  
from kunde  
where (kundenavn = 'Biltema' or kundenavn = 'Plantasjen') and omsetning > 1000000
```

where-delen kan omformes til:

where kundenavn = 'Biltema' and omsetning > 1000000  
or  
kundenavn = 'Plantasjen' and omsetning > 1000000;      (*distributiv lov, se kap.23.1*).

*Hva blir*

```
select kundenr, kundenavn, adresse  
from kunde  
where kundenavn = 'Biltema' or kundenavn = 'Plantasjen' and omsetning > 100000;
```

NB:

- Pass på presis definisjon av og / eller.
- Pass på prioritet av operatorer, bruk parenteser hvis nødvendig.
- Pass på at hver del blir et fullt logisk utsagn

## **Omformulering av logiske uttrykk.**

Av og til kan vi omforme logiske uttrykk:

Et eksempel på en slik omforming er deMorgans lover (kap. 23.1**Feil! Fant ikke referansekilden.**):

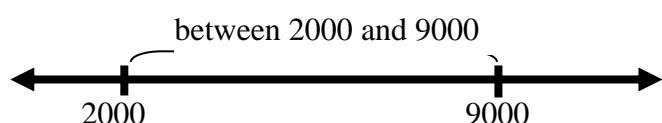
ikke (p eller q) = ikke p og ikke q,   og ikke (p og q) = ikke p eller ikke q

postnr < 2000 or postnr > 9000 =

not (not (postnr < 2000) and not(postnr > 9000)) =

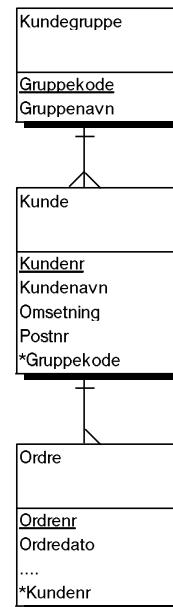
not (postnr >= 2000 and postnr <= 9000) =

not (postnr between 2000 and 9000).



### 9.3. Spørninger bygd koblinger, m/ kobling i where-setningen.

Kobling mellom ( $\geq$ ) 2 eller flere tabeller gjøres (nesten) alltid ved at vi krever at verdien i primærnøkkelen i en tabell er lik fremmednøkkelen i en (som regel) annen tabell.



Kundenr og -navn, ordrenr og -dato  
på alle kunde med postnr < 1300

```

select ordre.kundenr, kundenavn, ordrenr, ordredato
from kunde, ordre
where kunde.kundenr = ordre.kundenr
and postnr < 1300;
    
```

Kundenr , ordrenr og -dato på  
alle kunder med kundegruppe-  
navn "Offentlig"

```

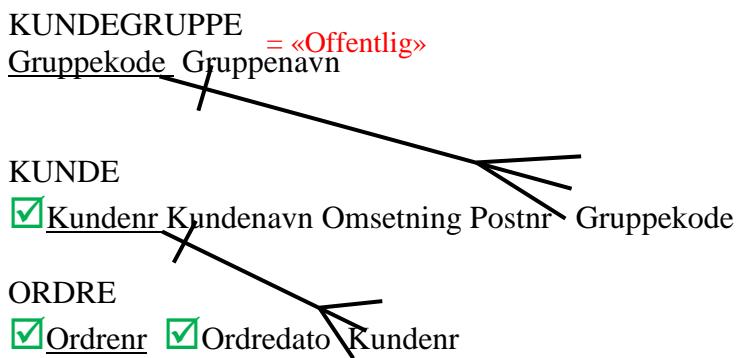
select ordre.kundenr, ordrenr, ordredato
from kundegruppe, kunde, ordre
where kunde.kundenr = ordre.kundenr
and kundegruppe.gruppekode = kunde.gruppekode
and gruppenavn = 'Offentlig';
    
```

3 kommentarer:

- Fast mønster her er at f.eks. 1:m-relasjonstyper resulterer i  
where ..... and <1-erside>.<primærnøkkel> = <m-side>.<fremmednøkkel>
- Tankegangen når denne setningen utføres er at alle kundegrupper, kunder og ordrer kobles sammen. Deretter begrenses de data som skal skrives ut med betingelsene i where-setningen.
- Dersom en kolonne finnes i flere tabeller, må kan skrive <tabellnavn> . før kolonnenavnet. Hvis det er entydig, holder det med kolonnenavnet, selv om tabellnavnet godt kan brukes i dette tilfelle også.

## Tips: visualiser spørringene!

Når det er spørninger med flere tabeller, kan det være lurt å lage en visualisering av tabellstrukturen, for dermed å kunne se lettere hva denne skal gjøre. Vi tar oppgaven over som eksempel.  viser det som skal skrives ut, mens det røde er betingelser.



Vi ser altså tydelig:

- Vi trenger alle de tre tabellene Kundegruppe, Kunde og Ordre.
- **Kobling (kråkefötter):**  
Kundegruppe.Gruppekode=Kunde.Gruppekode AND Kunde.Kundennr= Ordre.Kundennr  
→ *where-setningen (evt. som del av FROM-setningen, se neste delkapittel).*
- **Betingelse: Gruppenavn = «Offentlig» → where setningen.**
- **Utskrift:** Kunde.Kundennr, Ordrenr, Ordredato (evt. kunne vi brukt Ordre.Kundennr i stedet)  
→ *select-setningen .*

Dermed er setningen enkel å lage:

```
select ordre.kundenr, ordrenr, ordredato
from kundegruppe, kunde, ordre
where kunde.kundenr = ordre.kundenr and kundegruppe.gruppekode = kunde.gruppekode
      and gruppenavn = 'Offentlig';
```

Poenget her er ikke hvilke symboler eller farger man bruker, men bare det at spørringen vises grafisk gjør den mye enklere å forstå. Dette vises enda mer klart når det er vanskeligere spørninger.

## 9.4. Spørninger bygd koblinger m/ kobling i from-setningen.

De fleste systemer har mulighet for en mer moderne syntaks (fra SQL92), hvor koblingen uttrykkes som en inner eller outer join i from-setningen (på norsk kan vi snakke om innerkobling og ytterkobling, evt. indre og ytre sammenføyning).

**NB!** Kan hoppes over i første omgang, kom tilbake til det senere.

- **INNER:**

Kundenr og -navn, ordrenr og -dato  
på alle kunde med postnr < 1300,  
som tidligere

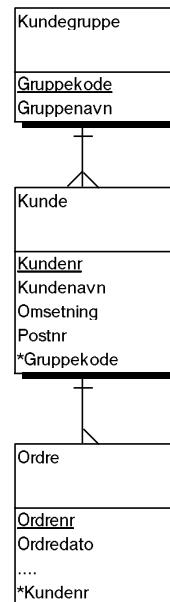
Den øverste setningen fra forrige eksempel kan uttrykkes som  
`select ordre.kundenr, kundenavn, ordrenr, ordredato  
from kunde inner join ordre on kunde.kundenr = ordre.kundenr  
where postnr < 1300;`
  - **LEFT OUTER:**

Alle kunder, og evt. ordre blir da en outer join.  
`select ordre.kundenr, kundenavn, ordrenr, ordredato  
from kunde left outer join ordre on kunde.kundenr =  
ordre.kundenr where postnr < 1300;  
(ikkematchende får NULL på ordre)`
  - **RIGHT OUTER:**

Alle kunder, med eventuell gruppenr og gruppenavn (det er ikke sikkert alle kunder er koblet til en kundegruppe).  
`select kundenr, kundenavn, kunde.gruppekode, gruppenavn  
from kundegruppe right join kunde on kunde.gruppekode = kundegruppe.gruppekode;`
  - **FULL JOIN**

finnes også i noen systemer, er "outer" fra begge sider.
  - **NESTET KOBLING ( $\geq 3$  TABELLER) MED JOIN I FROM-SETNINGEN:**

Skriv ut alle kundegrupper med evt. kunder og evt. ordrer:  
`select kundegruppe.gruppenr, gruppenavn, kunde.kundenr, kundenavn, ordrenr,  
ordredato  
from kundegruppe left outer join (kunde left outer join ordre on  
kunde.kundenr = ordre.kundenr) on kundegruppe.gruppenr = kunde.gruppenr;  
--- evt. med where-del ...`
- NB! Alle deler av en slik setning må ha samme join-type.



### KOMMENTARER:

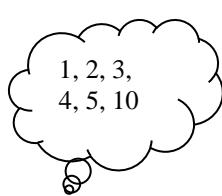
- ordet outer er frivillig.
- Når vi har betingelsen i FROM, så får vi et skille mellom "faste betingelser" pga. 1:mange eller 1:1 og betingelser som gjelder den spesifikke spørringen.
- En slik kobling danner en "virtuell tabell".

## 9.5. IN- operatoren, inkl. delspørninger med IN

Enkelt eksempel med in-operatoren er

```
Select kundenavn, kundeadresse  
from kunde  
where kundenr in (1,2,3,4,5,10)
```

kundenr ∈



de som er med i  
denne mengden

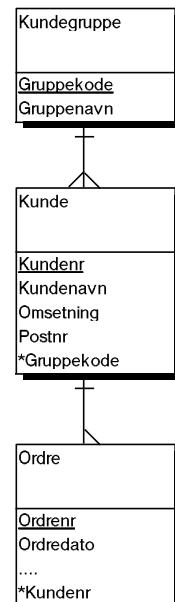
```
Denne kan også uttrykkes som  
Select kundenavn, kundeadresse  
from kunde  
where kundenr between 1 and 5 or kundenr = 10
```

## Delspørninger med IN

I stedet for at f.eks. kundenr er en fast/konstant mengde, kan denne mengden komme som resultat fra en spørring.

```
select kundenr, ordrenr, ordredato  
from ordre  
where kundenr in  
(select kundenr  
from kunde);
```

Kundenr, ordrenr og -dato på alle  
kunder som har minst en ordre



Vi utvider til å hente mengder i flere nivåer:

```
select kundenr, ordrenr, ordredato  
from ordre  
where kundenr in  
(select kundenr  
from kunde  
where gruppekode in  
(select gruppekode  
from kundegruppe  
where gruppenavn = 'Offentlig'));
```

Kundenr, ordrenr og -dato på alle  
kunder med gruppenavn 'Offentlig'

Merk: slike setninger kan leses «utenfra og inn» eller «innenfra og ut»:

Utenfra: Finn ordrer som er slik at den har kunder som er slik at den har kundegruppe/r som er slik at gruppenavnet er lik 'Offentlig'.

Innenfra: Først finner vi gruppekode/r hvor gruppenavnet = 'Offentlig'. Disse gruppekodeene tar vi med oss til kunde og finner tilsvarende kundenr. Så tar vi med disse til ordre og skriver ut de som matcher.

Spørninger med IN kan alternativt skrives som en kobling / inner join med «matchende» data. Men: hva hvis vi ønsker å finne data som ikke matcher? Teknikken er å bruke NOT IN i stedet<sup>14</sup>. Altså: delspøringer med NOT IN.

## Delspøringer med NOT IN

```
select kundenr, kundenavn  
from kunde  
where kundenr not in
```

(select kundenr from ordre);

Skriv ut kunder som aldri har bestilt noe.  
Det er det samme som å si at et gitt kundenr finnes i kunde, men ikke i ordre.

Dette blir en tabell over kundenr for de kundene som har en eller flere order. Vi skal skrive ut de som ikke er med i denne tabellen / mengden.

Det er også mulig at hoved- og delspørringen bruker samme tabell (men ulike instanser):

```
select *  
from kunde  
where postnr in (select postnr from kunde where kundenr = 5)
```

Skriv ut alt om kundene som har samme postnr som kunde nr. 5

**NB!** Resultatet av delspøringen er en mengde, derfor der det **feil** med f.eks. where kundenr = (select kundenr ....)

Normalt vil jo select-setningen returere med flere verdier også .. (Noen systemer godtar dette dersom det er garantert at vi får bare en verdi tilbake).

## Visualisering - igjen!

På samme måte som tidligere kan det være lurt å visualisere en spørring. Vi tar oppgaven over som eksempel:

KUNDE

Kundenr  Kundenavn Omsetning .....

**IKKE SÅNNE kundenr! (NOT IN)**

ORDRE

Ordrenr Ordredato Kundenr

<sup>14</sup> Vi kan også bruke f.eks. NOT EXISTS eller LEFT OUTER JOIN.

I SQL blir dette

```
SELECT Kundenr, Kundenavn FROM kunde  
WHERE kundenr NOT IN  
(SELECT kundenr FROM ordre);
```

Igjen ser vi at det er mye enklere å forstå spørringen hvis den vises grafisk.

## 9.6. *Exists / not exists*

Samme spørring som over kan formuleres som:

```
Select kundenr, kundenavn  
from kunde  
where not exists (select kundenr from ordre where kunde.kundenr = ordre.kundenr);
```

**NB:** Kunde defineres i den ytterste spørringen, og er da tilgjengelig også i den indre spørringen. Kunde.kundenr i den innerste spørringen skal referere til kunden i den ytterste, du må derfor ikke ha en ny from kunde i den innerste spørringen).

Når spørringen kjøres, skal man for en gitt kunde (ytterste spørringen) finne ut om de aktuelle kundedata skal skrives ut eller ikke. Det er da to muligheter:

- underspørringen inneholder ikke data. Det er da sant at det ikke eksisterer korresponderende ordre, og kundenr, kundenavn skrives ut.
- underspørringen inneholder data. WHERE-delen blir da totalt sett usann, og kundenr, kundenavn skal ikke skrives ut.

Derfor blir begge setningene under feil.

### Feil - 1:

```
Select kundenr, kundenavn  
from kunde  
where not exists  
(select kundenr from ordre);
```

### Feil - 2:

```
Select kundenr, kundenavn  
from kunde  
where not exists  
(select kundenr  
from ordre, kunde  
where kunde.kundenr = ordre.kundenr);
```

### Siste kommentar om not exists:

Not exists i 2 nivåer (dobbelt negering) kan brukes for å få ut alle som har er koblet til alle av noe annet. Eks: kunder som har bestilt alle varer = kunder hvor det ikke eksisterer varer som kunden ikke har kjøpt.

## 9.7. Gruppering (aggregering)

SQL kan også brukes til å telle opp (**count**), summere (**sum**), finne maksimum og minimum (**max**, **min**), gjennomsnitt (**avg**) m.m. Det er snakk om

- en oppstilling av alle i en tabell, evt. i koblingen av flere tabeller
- en oppstilling som gjelder innenfor en gruppe, f.eks. antall ordrer pr. kunde, ordresum pr. kunde, hva som er høyest omsetning pr. kundegruppe osv.

Dermed introduserer vi to nye språkelementer.

**GROUP BY** angir grupperingskolonnen/ene.

**HAVING** angir evt. betingelser som gjelder gruppen.

### Aggregering over alle, dvs. alle er i samme gruppe.

Vi har brukt tegnet \* tidligere for å markere hele raden. Dette kan også brukes for å telle opp antall rader.

```
select count(*)  
from kunde;
```

list ut kunder som har mer enn dobbelt så mye som gjennomsnittskunden i omsetning.  
Funker ikke i alle systemer

```
select kundenr, kundenavn  
from kunde  
where omsetning > 2 * (select avg(omsetning) from kunde);
```

```
select max(omsetning)  
from kunde;
```

```
select max(omsetning), min (omsetning)  
from kunde
```

Skriv ut kundenr og navn på kunden(e) med høyest omsetning.  
NB! Det kan være flere med høyest omsetning.  
NB! Her er = tillatt, fordi du sammenligninger en verdi med en annen verdi.

```
select kundenr, kundenavn  
from kunde where omsetning =  
(select max(omsetning) from kunde);
```

Et lite brukt språkelement, men som vi tar med for fullstendighetens skyld, er select count (distinct <kolonne>) ....

Vi kunne f.eks. tenke oss at vi skulle telle opp antall ulike kunder som har bestilt noe i 2018, men som sagt er det ikke så mye i bruk. Denne kan omskrives ved hjelp av andre språkelementer.

NB! Aggregering kan ikke gjøres direkte som en del av where-setningen. Dette er **feil**:

**Feil 1:** ~~select kundenr, kundenavn~~

~~from kunde where omsetning = max(omsetning);~~ ↪ etter = må vi ha en hel SQL-setning.

**Feil 2:** Denne er også feil, fordi ~~max~~ sammen med andre kolonner trenger en group by (se under):

~~select kundenr, kundenavn, max(omsetning)~~  
from kunde;

Kjører vi denne, får vi riktig maks. omsetning, men vi har ikke sagt noe om at kundenr og kundenavn skal gjelde den raden (evt. de radene) som har maksimal omsetning. Som regel skriver systemet ut kundenr og kundenavn for den siste raden i tabellen sammen med den riktige maksimale omsetningen.

Dermed må slike setninger må omskrives til f.eks.

select kundenr, kundenavn  
from kunde where **omsetning = (select max(omsetning) from kunde);**

Forklaring: Tenk dere at vi har data som vist i tabellen under.

**1) Resultatet av** select max(omsetning) from kunde blir 1500000

**2) For hver rad** spør vi: er det sant at omsetning=(select max(omsetning) from kunde)?

Kundenr	Kundenavn	Omsetning
371	.....	1000000
111	.....	1500000
207	.....	1200000

Svar: **Usant**, det er ikke slik at  $100000 = 1500000$ .  
Svar: **Sant**, det er slik at  $100000 = 1500000$ . **Skriv ut!**  
Svar: **Usant**, det er ikke slik at  $100000 = 1500000$ .

## Aggregering over en gruppe

Vi fokuserer altså på en og en verdi i en kolonne (kombinasjon), og gjør noe felles for hver slik verdi, f.eks. summerer, teller opp, finner gjennomsnitt eller den største eller siste.

Eksempel 1:

select gruppekode, sum(omsetning)  
from kunde  
group by gruppekode;

Finn samlet  
omsetning pr  
kundegruppe

## Eksempel 2:

```
select kunde.kundenr, kundenavn, count(ordenr)
from kunde, ordre
where kunde.kundenr = ordre.kundenr
group by kunde.kundenr, kundenavn;
```

Tell opp antall  
ordrer pr. kunde.

Merk at det som skal skrives ut (i select-setningen) må være

- **enten** være kolonner som finnes i GROUP BY
- **eller** aggregeringer (SUM, COUNT, MAX m.m.)

## Gruppering med betingelse på gruppen, HAVING

HAVING er en betingelse som gjelder gruppen, altså som må være sant for gruppen, ikke for hvert enkelt element. Hvis vi har f.eks. gruppekode = 'P' f.eks. har omsetning på 250000, 100000 og 250000, så er det ikke slik at omsetning > 300000 for noen av de, derimot er sum(omsetning) > 300000.

Altså:

```
SELECT <kolonner/beregninger>
FROM <tabeller/spørninger som definerer en virtuell tabell/er>
WHERE <betingelse på enkeltrader>
GROUP BY <kolonner/beregninger>
HAVING <betingelse på gruppen>
```

### Eksempel 1:

```
select gruppekode, sum(omsetning) - som over, men utvidet.
from kunde
group by gruppekode
having sum(omsetning) > 300000;
```

Finn samlet omsetning pr  
gruppe, men bare de  
kundegruppene som har samlet  
omsetning > 300 000

### Eksempel 2:

```
select kunde.kundenr, kundenavn, count(ordenr) - som over, men utvidet.
from kunde, ordre
where kunde.kundenr = ordre.kundenr
group by kunde.kundenr, kundenavn
having count(*) >= 10;
```

tell opp antall ordrer  
pr. kunde, men ta  
bare med de som har  
minst 10 ordre

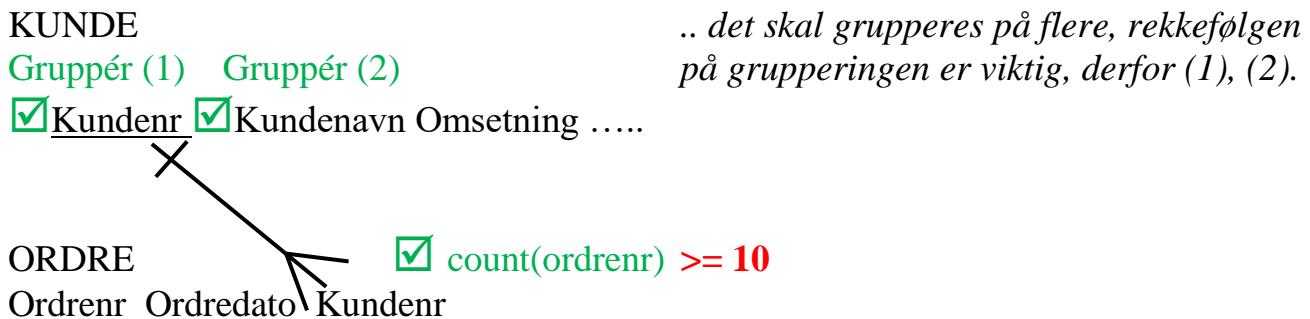
### Eksempel 3:

```
select kundenavn
from kunde
group by kundenavn
having count(*) > 1;
```

list opp  
kundenavn som  
er duplike

## Visualisering

Vi bruker eksempel 2 for å visualisere.



### 9.8. ALL / ANY (SOME)

ALL og ANY brukes på resultatet av en delspørring.

- ALL er sann hvis alle i delspørringen oppfyller kriteriet. Usant hvis delspørringen er tom.
- ANY er sann hvis noen (en eller flere) oppfyller kravet. Sant hvis delspørringen er tom. SOME er ekvivalent med ANY.

#### Eks. 1

```
Select *  
from kunde  
where kundenr >= ALL (select kundenr from kunde);
```

er det samme som

```
Select *  
from kunde  
where kundenr >= (select max(kundenr) from kunde); --- = går greit
```

#### Eks. 2

```
Select *  
from kunde  
where kundenr <= ALL (select kundenr from kunde);
```

er det samme som

```
Select *  
from kunde  
where kundenr <= (select min(kundenr) from kunde); --- = går også greit
```

### **Eks. 3**

```
Select *  
from kunde  
where kundenr > ANY (select kundenr from kunde);
```

er det samme som

```
Select *  
from kunde  
where kundenr > (select min(kundenr) from kunde);
```

### **Eks. 4 (vanskelig !)**

Er det bare en kundegruppe som har kunder med omsetning over 1 mill.?

M.a.o.: Er det slik at alle kunder som har omsetning over 1 mill hører til samme gruppe?

```
select *  
from kundegruppe  
where gruppekode = ALL (select gruppekode from kunde where omsetning > 1000000)
```

### **Eks. 5 (vanskelig !)**

Den virkelige nytten av ALL etc. har vi når man vi skal gjøre operasjoner på allerede aggregerte data. Et eksempel: Vi skal skrive ut den kundegruppen som har høyeste gjennomsnittsomsetning, altså en ”dobbelt aggregering”. Det er ikke lovlig å skrive noe slikt som where max(select avg( ...)).

Derimot:

```
Select gruppekode  
from kunde  
group by gruppekode  
having avg(omsetning) >= ALL (select avg(omsetning) from kunde group by  
gruppekode);
```

- altså: skriv ut den gruppekoden som har en gjennomsnittsomsetning som er større eller lik den største av alle de andre gjennomsnittsomsetningene. Dette fungerer også dersom det skulle være flere som har den samme største gjennomsnittsomsetningen.

Skal man ha med gruppenavn etc. blir utvidelsen slik:

```
Select kundegruppe.gruppekode, gruppenavn  
from kunde, kundegruppe  
where kunde.gruppekode = kundegruppe.gruppekode  
group by kundegruppe.gruppekode, gruppenavn  
having avg(omsetning) >= ALL (select avg(omsetning) from kunde  
group by gruppekode);
```

## 9.9. **Union, snitt, differanse etc.**

Siden resultatet av SQL-spørninger er mengder, kan man i prinsippet bruke mengdeoperatorer på resultatet av disse.

Union er implementert i de fleste systemer, snitt og differanse er sjeldnere.

### Union – eksempel

```
select * from kundebase1  
union  
select * from kundebase2;
```

NB! Forutsetter at tabellene  
er helt kompatible

Slå sammen kundebasene. Forutsetter  
at det ikke er overlappende  
primærnøkkelverdier.

### Snitt - eksempel

```
select * from kundebase1  
intersect  
select * from kundebase2;
```

Hvilke kunder finnes i begge?

### Mengdedifferanse - eksempel

```
select * from kundebase1  
minus  
select * from kundebase2;
```

Hvilke kunder finnes i Kundebase1,  
men ikke i Kundebase2?

I noen systemer heter denne operatoren EXCEPT, i andre kan man bruke begge om MINUS og EXCEPT om hverandre. I andre systemer igjen finnes ikke denne operatoren.

### Utfordring

Hvordan kan man bruke andre språkelementer til å uttrykke det samme som UNION, INTERSECT og MINUS/EXCEPT?

## 9.10. Beregningsuttrykk, kommentarer etc. i SQL-setningene

Du kan foreta beregninger, skrive konstante tekster etc. i SQL-setningene, f.eks.

```
select kundenr,'Omsetning',omsetning,'Moms', omsetning*0.25 as moms  
from kunde;
```

as ..... finnes ikke i alle  
systemer



Noen systemer kan ha en ny select i select-setningen.

To eksempler:

```
select (select sum(omsetning) from kunde) / count(*)  
from kunde;
```

```
select ordrenr,kundenr, (select kundenavn from kunde where kunde.kundenr  
= ordre.kundenr)  
from ordre;
```

## 9.11. Alias

Alias vil si at man lager et annet navn på noe. Vi kan f.eks. skrive setningen

```
select ordre.kundenr, ordrenr, ordredato  
from kundegruppe, kunde, ordre  
where kunde.kundenr = ordre.kundenr  
    and kundegruppe.gruppekode = kunde.gruppekode  
and gruppenavn = 'Offentlig';
```

som

```
select o.kundenr, ordrenr, ordredato  
from kundegruppe kg, kunde k, ordre o  
where k.kundenr = o.kundenr  
    and kg.gruppekode = k.gruppekode  
and gruppenavn = 'Offentlig';
```

I tillegg til at dette kan være en latmannsvariant når det er lange tabellnavn, er det av og til nødvendig å bruke alias. Dette gjelder når man trenger 2 eller flere instanser av samme tabell samtidig.

### Eks. 1:

```
Select k1.kundenavn, k1.kundenr
from kunde k1, kunde k2
where k1.kundenavn = k2.kundenavn
    and k1.kundenr > k2.kundenr;
```

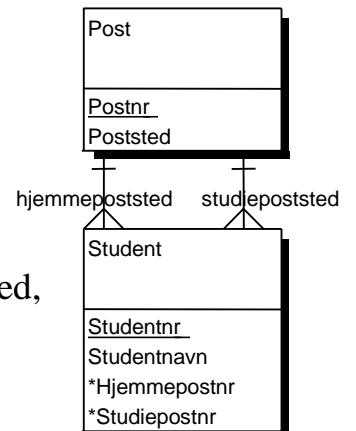
list opp kundenr/navn  
for de med samme  
kundenavn

lager to instanser  
av kundetabellen

### Eks. 2:

En student har **et** postnr/sted for hjemmeadresse,  
**et (annet)** for sin studieadresse.

```
select studentnr, studentnavn, hjemmepostnr, hjemmepoststed.poststed,
       studiepostnr, studiepoststed.poststed
from student, post hjemmepoststed, post studiepoststed
  where hjemmepostnr = hjemmepoststed.postnr and
        studiepostnr = studiepoststed.postnr;
```



### Eks. 3:

```
select stedformaxikunde.kundenr, stedformaxikunde.kundenavn
from kunde maxikunde, kunde stedformaksikunde
where maxikunde.omsetning = (select max(omsetning) from kunde)
and maxikunde.postnr = stedformaxikunde.postnr;
```

Skriv ut alle kunder som  
er på samme sted(er)/  
postnr som den/de  
kunder som har størst  
omsetning.

### Eks. 4:

Forberedelse: Skriv «Alt betalt» dersom alle kundene har betalt alle ordrene.

```
select 'Alt betalt' -- hvis ikke skrives det ikke ut noe.
From ordre
Where 'Be' = ALL ( select status from ordre);
```

Her kommer eksempelet: Skriv ut kundenr for de kundene hvor alt er betalt.

Her vil vi skrive ut et kundenr, men bare hvis 'Be' er status for alle med dette kundenr.  
Vi bruker alias for å koble sammen kundenr for ytre og indre del av spørringen.

select kundenr

From ordre

Where 'Be' = ALL ( select status from ordre as ordre2 where ordre.kundenr = ordre2.kundenr);

### Eks. 5: - en spørring kan også gis et alias-navn

select kunde.kundenr, o.antordre as antall

from kunde inner join (select kundenr, count(\*) as antordre from ordre group by kundenr) as o on kunde.kundenr = o.kundenr;

I forhold til dette vil vi imidlertid heller anbefale å navngi det som utsnitt (view, se kap. 11).

### Eks. 6 - aliasnavn i en vanskelig spørring – kun for spesielt interesserte (!)

Vi har tidligere sett på å skrive ut den kundegruppen som har høyeste gjennomsnittsomsetning. En løsning med bruk av ALL fungerte. Det finnes imidlertid også andre metoder – men denne er ganske vanskelig.

Vi må først finne gjennomsnittsomsetningen, deretter hvilken kundegruppe som har dette.

#### 1) Finn gjennomsnittsomsetningen

```
select avg(omsetning) as snitt      -- Vi trenger dette aliaset til senere
from kunde
group by gruppekode;    Vi skal bruke spørringen i videre beregninger, og lager dermed et alias også
på spørringen.
```

#### 2) Finn maks av gjennomsnittsomsetninger

```
select max(snitt)
from
(select avg(omsetning) as snitt
from kunde
group by gruppekode) as snitt_oms;
```

#### 3) Skal man så finne den kundegruppen som dette gjelder, må man ta med

```
select gruppekode
From kunde
Group by gruppekode
having avg(omsetning) =
(select max(snitt)
from
(select avg(omsetning) as snitt
from kunde
group by gruppekode) as snitt_oms);
```

4) og til slutt for å få med både gruppekode og gruppenavn

```

select * from kundegruppe
where gruppekode in (
    select gruppekode
    From kunde
    Group by gruppekode
    having avg(omsetning) =
        (select max(snitt)
        from
            (select avg(omsetning) as snitt
            from kunde
            group by gruppekode) as snitt_oms));

```

Pust ut!

## 9.12. Kartesisk produkt

Hva blir

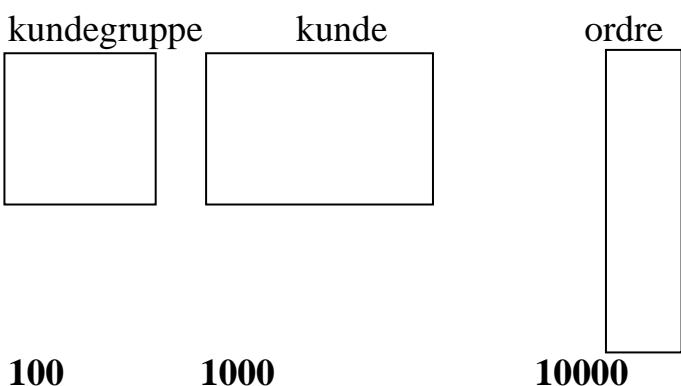
```

select ordre.kundenr, ordrenr, ordredato
from kundegruppe, kunde, ordre
-      altså uten where / betingelser.

```

Det blir alle kundegrupper, kunder og ordrer koblet med hverandre, selv om de ikke har like verdier for primær- og fremmednøkler.

Resultat:



Resultatet blir en kobling med alle-mot-alle-mot alle. Hvis vi har f.eks. 100, 1000 og 10.000 rader, blir resultatet 1.000.000.000 rader (!!)

Dette er sjeldent lurt, men kan brukes i spesielle sammenhenger, f.eks. hvor man skal lage mange rader automatisk (lag 1 million testdata raskt!).

## 9.13. Noen funksjoner som det kan være behov for

Under følger noen funksjoner som det kan være behov for når man arbeider med SQL. Noen av disse er bra standardiserte, andre varierer mellom de ulike systemene.

Beskrivelse	Matematisk notasjon	Notasjon i SQL
Absoluttverdien	$ x $ . NB! absoluttverdien kan brukes til å beskrive tall i nærheten av et annet. F.eks er $ x-100  < 10$ de tall som ligger $\leq 10$ unna tallet 100.	ABS(x)
Heltallsdivisjon	$\lfloor x / y \rfloor$ - divisjon og deretter <u>ned</u> til nærmeste heltall. I andre tilfelle ønskelig å uttrykke rundes opp til nærmeste heltall. Dette beskrives som $\lceil x / y \rceil$	<ul style="list-style-type: none"> <li>I noen systemer: x DIV y.</li> <li>I noen systemer blir x / y automatisk heltall dersom både x og y er heltall.</li> <li>CEILING (x) runder nærmeste heltall <math>\geq x</math>.</li> <li>FLOOR runder til nærmeste heltall <math>\leq x</math>.</li> </ul>
Avrunding	til nærmeste heltall: Gå en halv opp, ta deretter bort desimalene: $\lfloor x + 0.5 \rfloor$	ROUND(x,desimaler) etter vanlige avrundingsregler.
Rest etter divisjon, modulus	x mod y, mod(x,y). F.eks. er 17 mod 3 = 2	Ulik, men gjerne x MOD y eller x % y
startverdi ligger mellom x og y	$x \leq \text{startverdi} \leq y$	startverdi BETWEEN x AND y.  Kan også formuleres som $x \leq \text{startverdi} \text{ and } \text{startverdi} \leq y$
lengden av en streng		LEN(s), LENGTH(s) el.l., hvor s er den aktuelle strengen
delstreng		heter f.eks. SUBSTRING(s,i,j), hvor i er start og j er antall tegn,
finning av en streng i en streng		forskjellig fra system til et annet. Behovet kan f.eks. være å finne første , (komma) i en streng, f.eks. hvis et navn er gitt som "Hansen, Ole" og vi skal splitte fornavn og etternavn
fjerning av blanke foran eller etter		<ul style="list-style-type: none"> <li>LTRIM(s): left trim</li> <li>RTRIM(s):right trim</li> <li>TRIM(s): begge deler.</li> </ul> Eksempelvis kan man skrive SELECT RTRIM(navn) FROM ....
datofunksjoner, f.eks. uttrekk av år for en dato, sammenligning etc.		her er ting delvis ustandardisert, delvis er det et problem med lokale variasjoner, f.eks. når det gjelder datoformater
Mange andre funksjoner finnes, men er sjeldnere i bruk		

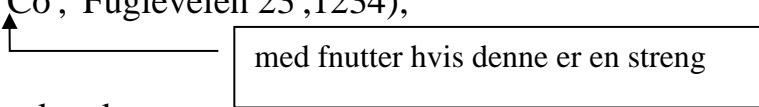
## 9.14. Innsetting av nye data.

Det finnes 3 måter å sette inn data på via SQL.

### Enten: innsetting av enkeltrader.

insert into <tabell>(<kommaseparert kolonneliste>) values (<kommaseparert verdiliste>);

f.eks. insert into kunde(kundenr,kundenavn,adresse,postnr) values (234, 'Hansen & Co', 'Fugleveien 23',1234);



Fire merknader:

- På denne måten kan man sette inn noen, men ikke alle kolonner i en tabell.
- Hvis en kolonne utelates, må dette enten ha at NULL er tillatt, eller ha en default-verdi.
- Dersom man setter inn **verdier i alle kolonner**, kan kolonnelisten utelates, f.eks.  
insert into kunde values (234, 'Hansen & Co', 'Fugleveien 23',1234);
- Det er mulig å legge inn ett eller flere sett med ,(<verdier>) for å legge inn flere rader på en gang.

### Eller: innsetting av data som er resultat av en SQL-setning.

Eks.1:

```
insert into kunde2
select * from kunde
where kundenr <= 1000;
```

Legg alle kundenr, kundenavn og landsnavn for  
alle utenlandskunder over i en annen tabell.

Eks. 2:

```
insert into utenlandskunder (kundenr, kundenavn, landsnavn)
select kundenr, kundenavn, landnavn
from kunde, land
where kunde.landkode = land.landkode
and land.landkode <> 'N';
```

LAND	
landkode	landsnavn
DK	Danmark
N	Norge
S	Sverige
D	Tyskland

KUNDE			
kunde nr	kunde navn	...	landkode
832	...		N
712			S
615	..		S
713	..		DK
....	....	...	.....

Eks. 3:

**Kolonnenavnene behøver ikke å være like. Det kan også være på grunnlag av en beregning, f.eks.**

```
insert into ant_pr_gruppe (gkode, gruppenavn, gruppeantall)
select kunde.gruppekode, gruppenavn, count(*)
from kundegruppe, kunde
where kundegruppe.gruppekode = kunde.gruppekode
group by kunde.gruppekode;
```

## **Eller: oppretting av en tabell og direkte innsetting av data.**

**NB! Denne muligheten finnes ikke i alle systemer (eller har noe annerledes syntaks). Det gjelder bl.a. mySQL**

Eks. 1 – opprett en tabell og ta en ren kopi:

```
insert into kunde2
select * from kunde;
```

Eks. 2 – laging av tabeller, med utvalg og navngiving.

```
select prosjektnr, beskrivelse as prosjektnavn
into prosjekt2
from prosjekt
where prosjektnr =1;
```

## **Andre måter å legge inn data på.**

Mange systemer har dessuten andre måter å legge inn store datamengder på, eksempelvis

- innlesing via ulike formater. En enkel standard er såkalte kommaseparerte filer.
- direkte ”innliming av data”, f.eks. kopiering av data fra et regneark som limes inn i databasen via grafisk brukergrensesnitt for det aktuelle databasesystemet.
- hvis data ligger i en annen database, er det ofte mulig å koble seg på denne databasen og så enten importere dataene inn eller ”linke dem inn” for så å kjøre en insert ... select-setning som gjør selve overføringen.

## **9.15. Endring og sletting av data.**

Tilsvarende kan man endre en rekke poster eller slette en rekke poster ut fra where-delen i en SQL-spørring:

### **Endring av data.**

Eks. 1:

```
update kunde  
set kundenr = kundenr + 1000  
where landkode <> 'N';
```

```
update kunde  
set kundenavn = "Olsen"  
where kundenr = 1743
```

Eks. 2:

Flere tabeller kan være involvert, flere kolonner kan oppdateres på en gang. Funger ikke i alle systemer!

```
update kunde, kundegruppe  
set kundegruppe.gruppekode = 'FC', gruppenavn = 'First Class Customer'  
where kundegruppe.gruppekode = 'ST' and kundegruppe.gruppekode =  
kunde.gruppekode;
```

Eks. 3:

```
update kunde  
set postnr = null; - farlig !!
```

### **Sletting av data.**

Eks. 1:

```
delete from kunde  
where landkode <> 'N';
```

Eks. 2:

```
delete from kunde  
where gruppekode in (select gruppekode from kundegruppe where gruppenavn =  
'Offentlig');
```

#### **Advarsel:**

Dette er kraftfulle kommandoer, og noen systemer har begrensninger i tillatelse i forhold til disse.

### **9.16. I praksis: Data legges inn, endres og søkes etter via et brukergrensesnitt.**

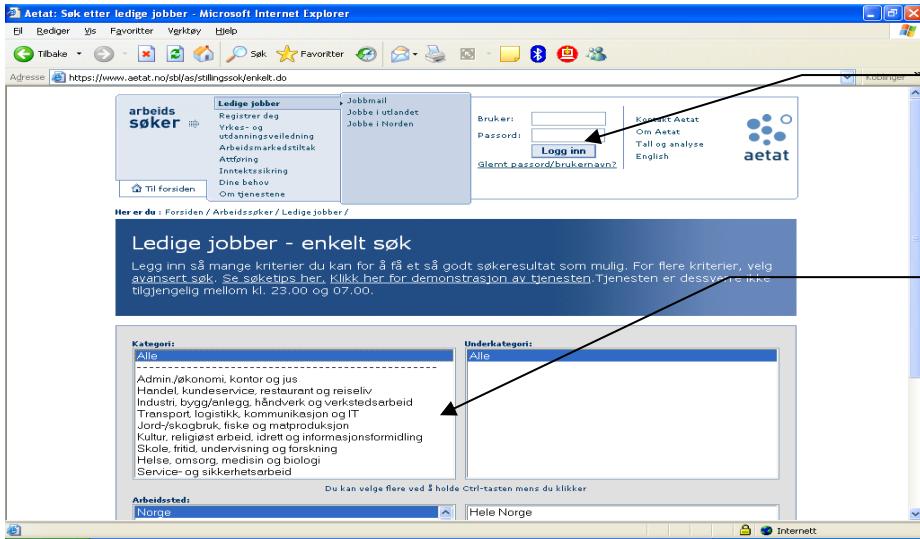
I praksis lages det et brukergrensesnitt som kjøres web-basert eller som en applikasjon i operativsystemet.

- Innsetting / endring / sletting av enkeltrader eller flere rader gjøres i praksis via dette grensesnittet.
- Ditto med søking etter data – select-setninger.
- SQL brukes for å definere hva de ulike delene av en slik applikasjon skal gjøre med dataene.
- Der hvor vi har brukt konkrete tall, strenglitteraler etc. over, vil da SQL-setningene i stedet inneholde en variabel som har verdien som finnes f.eks. i en kontroll (tekstboks, listebox, kombobox osv.) i brukergrensesnittet.

Vi snakker om slike systemer som databaseapplikasjoner, eller databasebaserte applikasjoner, men det vil like ofte være systemer hvor databasen er helt skjult, f.eks. i regnskapssystemer, bestillingssystemer på nettet, pasientsystemer til leger osv.

Eksempel på oppbygningen av et slikt system:

- brukergrensesnitt, ofte i to ulike modus
  - ett for endring av data, ofte passordbeskyttet
  - ett for kun søking etter data, åpent



Modus for endring av  
data, passordbeskyttet

Søking, åpent

- Program for styring av skjermbilder (ofte i flere nivåer av programvare) og kommunikasjon med databasen

SQL-setning                    resultattabell

- Databasesystem
- Selve den fysiske databasen



Ved feil hender det jo t.o.m. at databasefeilmeldinger kommer på skjermen:

```

DBD::Oracle::db do failed: ORA-01653: unable to extend table STEPHEN.WHOIS_LOG by 2560 in
tablespace STEPHEN (DBD ERROR: OCIStmtExecute) at /usr/local/sbin/in.whoisd line 322, line 1.
Couldn't insert: ORA-01653: unable to extend table STEPHEN.WHOIS_LOG by 2560 in tablespace
STEPHEN (DBD ERROR: OCIStmtExecute) at /usr/local/sbin/in.whoisd line 322, line 1.
Issuing rollback() for database handle being DESTROY'd without explicit disconnect(), line 1.

```

## 10. Bruk av SQL for å kommunisere mellom ulike systemer.

### 10.1. Verktøy og lagring av data

**Tidligere:**

Database og utviklingsverktøy er ett felles verktøy. Verktøyet har sitt "eget format" for filstruktur, indeks etc., og leser & skriver kun i sitt eget format ("lukket system").

**Nå:**

Standarder som gjør at ulike databasesystemer og utviklingsverktøy kan kommunisere.  
4 muligheter:

- felles format for utveksling via filer, f.eks.
  - XML (eXtendable Markup Language), JSON<sup>15</sup>
  - CSV (kommaseparerte verdier), gjerne via et regneark
  - .DBF (gammelt format fra såkalte dBase-filer) .
- Utviklingsverktøy kan ofte lese / skrive i dette formatet.
- direkte koblinger mellom et gitt databasesystem og utviklingsverktøyet (f.eks. direkte kobling fra et utviklingsverktøy til Oracle).
- generelle utvekslingsformater, uavhengig av verktøy, slik at disse formatene fungerer som et ”mellomnivå” mellom database og utviklingsverktøy.
- Programvare som ligger mellom databasen og utviklingsverktøyet (”mellomvare”). Typisk: applikasjonsservere.

### 10.2. ODBC, JDBC m.fl.

Det finnes en rekke standarder for å sende spørrsager og få tilbake data på et eller annet format. Formålet er å kunne lese og skrive data uavhengig av fysisk lagringsformat, databasesystem etc., ==>

- Kan lage applikasjoner for flere databasesystemer.
- Lettere å bytte databasesystem.
- Lett å koble f.eks. et rapporteringsverktøy mot stor database

Mest kjent er

- ODBC, Open Data Base Connectivity
- JDBC – Java Data Base Connectivity .

Disse støttes av de aller fleste databaseleverandører, og er «ikke-proprietære».

---

<sup>15</sup> Se kap. 21.3.



Her sier man fra og skriver/leser til/fra datakildenavn (f.eks. "Kundedata"), men vet ikke hvor data fysisk lagres.

SQL-setninger og postutvalg

Her defineres datakildenavn + hvor data finnes fysisk etc. (f.eks. "Kundedata" er på Oracle-format, ligger på f:\.....\...|kun..)

Drivere pr. DBMS

SQL-setninger og postutvalg

## Det hender ting krasjer .....

Vi tar med noen godbiter:



[www.dagbladet.no](http://www.dagbladet.no)

**f-b.no - Microsoft Internet Explorer**

File Edit View Favorites Tools Help

Back Forward Stop Home Search Favorites Print Links

Address: <http://www.f-b.no/apps/pbcs.dll/nabolaget?show=freetext&searchreset=0&fromrow=1&pagesize=10&Category=STEP3&CatId=&Cat=&> Go Links >

The screenshot shows a Microsoft Internet Explorer window displaying the f-b.no website. The main content area shows a search form with fields for 'Jeg er på jakt etter:' and 'Sted:', both containing placeholder text. Below the form, an error message is displayed: 'Error code: -170 [Microsoft][SQLServer 2000 Driver for JDBC][SQLServer]Line 1: Incorrect syntax near ')'. The page also includes a sidebar with links like 'Til forsiden', 'Hvordan søke', 'Annonser', etc., and a news section with a 'LYTT' banner.

**Squirrelmail - Windows Internet Explorer**

File Rediger Vis Favoritter Verktøy Hjelp

Favoritter Squirrelmail Sikkerhet Verktøy

Søk: Forrige Neste Alternativer

The screenshot shows a Windows Internet Explorer window displaying the Squirrelmail web interface. On the left, there's a sidebar with 'Mapper' (Folders) and 'Innboks (90)' (Inbox). The main content area has a red 'FEIL:' (Error) box containing the text 'FEIL: Tilkobling avsluttet av IMAP-tjeneren.' and 'Query: SELECT "mail/Sent"'. The browser toolbar at the top includes icons for Back, Forward, Stop, Home, Search, and Favorites.

**500 - Internal server error.**

There is a problem with the resource you are looking for, and it cannot be displayed.

The web site you are accessing has experienced an unexpected error.  
Please contact the website administrator.

The following information is meant for the website developer for debugging purposes.

Error Occurred While Processing Request

Element SOKFELT is undefined in FORM.

The error occurred in E:\www\www.mf.no\sok.cfm: line 16

```
14 :     AND (from_date < #createodbcdatetime(now())# OR from_date IS NULL)
15 :     AND (to_date > #createodbcdatetime(now())# OR to_date IS NULL)
16 :     AND (nyhetnavn LIKE '%#FORM.sokfelt#%' OR ingress LIKE '%#FORM.sokfelt#%' OR brodtekst LIKE '%#FORM.sokfelt#%')
17 :     </cfquery>
18 :
```

Resources:

- Check the [ColdFusion documentation](#) to verify that you are using the correct syntax.
- Search the [Knowledge Base](#) to find a solution to your problem.

Browser Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Trident/4.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; InfoPath.3; Tablet PC 2.0; .NET4.0C)

# 11. Utsnitt / visning / view.

## 11.1. Utsnitt / visning / view i SQL.

Et utsnitt/visning (engelsk: view) er en “kunstig” (virtuell) tabell, avledet av andre tabeller eller utsnitt i databasen.

- Det lagres aldri data i et view, kun evt. i basistabell(er).
- Definisjonen lagres og samkjøres med basistabeller v/ spørring etc.

### Eksempel 1:

```
CREATE VIEW Oslo_kunde
```

```
AS
```

```
SELECT * FROM kunde
```

```
WHERE postnr < 1299;
```

Utsnitt basert på  
annet utsnitt

### Eksempel 2:

```
CREATE VIEW Oslo_sumomsetning (gruppekode,gruppenavn, sum_oms)
```

```
AS
```

```
SELECT kunde.gruppekode, gruppenavn, sum (omsetning)
```

```
FROM oslo_kunde, kundegruppe
```

```
WHERE kunde.gruppekode = kundegruppe.gruppekode
```

```
GROUP BY kunde.gruppekode,gruppenavn;
```

forutsetter at ingen utsnitt  
baserer seg på denne  
igjen. Her: Drop blir ikke  
utført.

### Eksempel 3:

```
DROP VIEW Oslo_kunde;
```

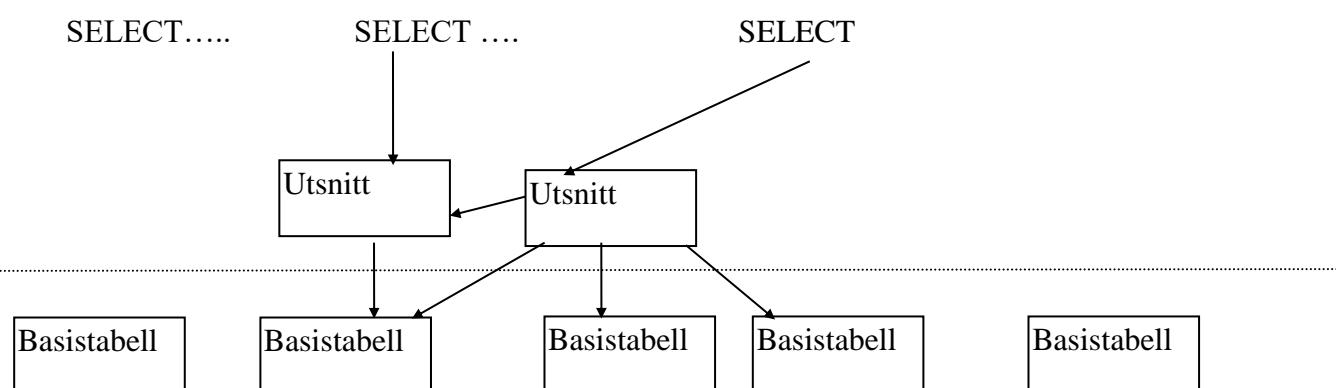
evt. med  
nøkkelord  
RESTRICT

```
eller
```

```
DROP VIEW Oslo_kunde CASCADE;
```

utsnitt som baserer seg på  
denne slettes også. NB!  
Finnes ikke i alle dialekter.

SQL-setninger kan bruke bare basistabeller, bare utsnitt eller en kombinasjon.



## 11.2. Utsnitt: typer og bruksområder.

### Vanlige typer:

- Horisontalt utplukk (utvalgte rader), restriksjoner
- Vertikalt utplukk (utvalgte kolonner), projeksjoner
- Koblinger av flere tabeller (f.eks. for forenkling) eller “folkelig gjøring”
- Summering
- Blanding av disse

### Bruksområder:

- Skjuling av rader eller kolonner, detaljerte data i tabellen for noen brukere.  
Brukere ser / kan endre bare “sin del” av databasen, eller det de trenger av den.  
Eks.: En bruker får kun se et “summeringsview” av en tabell med sensitive data.
- Skjuling av uaktuelle data i en bestemte situasjoner (filtrering):  
Eks: En varetabell har mange varer utgåtte varer. Disse må beholdes i systemet, men bør ikke vises ved registrering av nye ordrer.

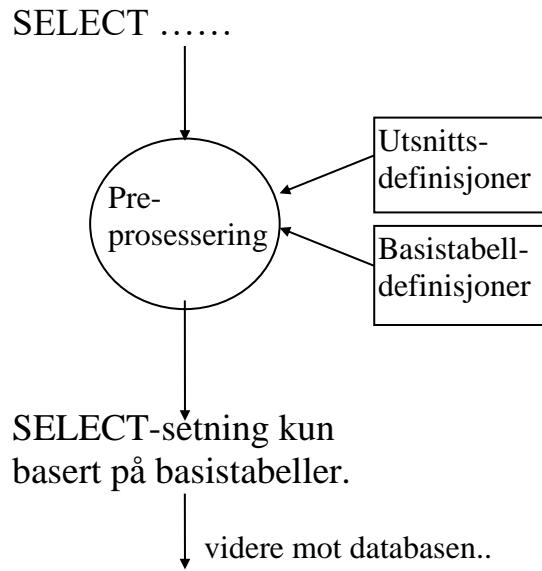
Forslag:

```
CREATE VIEW aktuelle_varer (varenr, varenavn) AS  
SELECT varenr, varenavn FROM varer WHERE status <> 'Utgått';
```

- Oppdeling av en komplisert setning i flere deler som bygger på hverandre. ==>  
Setningene blir enklere å lage og forstå.
- Hvis spørringene baseres på utsnitt: tabeller kan endres, splittes, omorganiseres etc. uten at spørringene endres. (**Viktig prinsipp: Datauavhengighet.**)
- Forenkling av tabelldefinisjoner, slik at lettere å spørre mot disse. Kan også fungere som en “lagret spørring”.
- Få flere tabeller med samme struktur til å se ut som om det er en tabell (f.eks. med UNION mellom tabellene).
- Endring av tabell- eller kolonnenavn, f.eks. norske navn på engelske tabellnavn.

Noen utviklere lager utsnitt på alle tabeller, lar all videre behandling skje mot disse.

### 11.3. Hvordan utføres en SQL-select-setning med utsnitt?



====> ekstra “overhead” ved kjøring av SQL-setninger.

I praksis spiller det gjerne liten rolle, fordi denne omformingen er svært rask.

## 11.4. Oppdaterbarhet via et utsnitt:

NB! Oppdateringen gjelder i basistabellene. Derfor er det riktig å si oppdaterbarhet via et utsnitt, ikke oppdaterbarhet av et utsnitt. Dessverre er begrepet view updateability et så innarbeidet begrep at det er vanskelig å bli kvitt det.

### Prinsipp:

- Dersom data som skal innsettes / endres / slettes kan entydig identifiseres i en/flere basistabeller, er utsnittet oppdaterbart.
- Dessuten: Nødvendig (NOT NULL)-kolonner må få en verdi.

### Noen regler for oppdaterbarhet:

- Utsnitt basert på én tabell, og som inneholder primærnøkkelen og alle Nødvendig-kolonner, er oppdaterbare.
- Også utsnitt basert på flere kan være oppdaterbart hvis koblingen har PK / FK-koblinger. (Mange syst. har begrensninger her!)
- Aggregeringsverdier er ikke oppdaterbare.

## Verdier om faller ut av / inn i et utsnitt:

Hva hvis vi har

abbnr	avsnavn		postnr
130	Morgenavisen	..	1234
131	Dagsavisen	..	0171
132	Morgenavisen	..	0612 ▾
133	Nattavisen	..	0721

og sier

```
UPDATE Oslo_kunde  
SET postnr = 7000  
WHERE kundenr = 132;
```

evt. endrer direkte på skjermen

eller

```
INSERT INTO Oslo_kunde VALUES (143, 'Frokostavisen', ..., 7000);
```

Tilsvarende:

```
UPDATE KUNDE  
SET postnr = 0107  
WHERE kundnr = 130;
```

I noen systemer:

Kan hindre slik inn / utfall av verdier ved å spesifisere  
WITH CHECK OPTION CASCADING eller  
WITH CHECK OPTION LOCAL  
==> får konsistenssjekk på verdier.

endringer som gir utfall i  
denne eller  
underliggende tillates

endringer som gir utfall tillates bare  
hvis det fører til sletting av  
tilhørende rad i basistabellen

## 12. Indekser, fysisk datadefinisjon og aksessmekanismer

### 12.1. Indekser – hva og hvorfor ...

- Indekser kan tenkes på som hjelpetabeller som kan brukes til raskt oppslag i en basistabell.
- Kan bestå av en kolonne eller en ordnet rekkefølge av kolonner (sammensatt indeks).
- Kan være hver økende (ASC - default) eller synkende (DESC). For en sammensatt indeks: kan blandes.
- Kan være unik eller ikke-unik.
- Kan være fullstendig (dense) eller ufullstendig (non-dense, kun hvis underliggende fil/nivå er sortert)
- Noen spørninger kan besvares bare ut fra indekser, uten å se i selve basistabellen.

Navneindeks

Verdi	Pos
Andersen	3
Hansen	6
Hansen	1
Jensen	4
Karlsen	5
Nilsen	2

Basistabell (usortert)

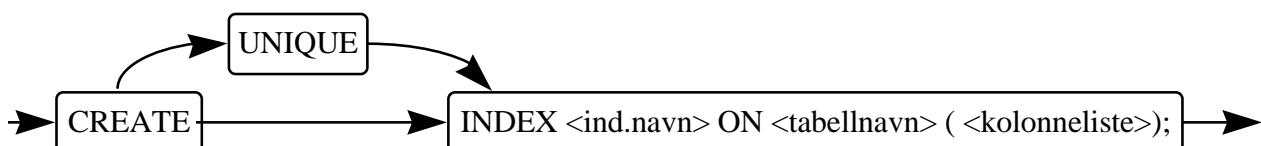
Ansnr	Navn	Adr	Postnr
56	Hansen		9000
21	Nilsen		9100
59	Andersen		9030
89	Jensen		9000
51	Karlsen		9000
72	Hansen		1234

Postnrindeks

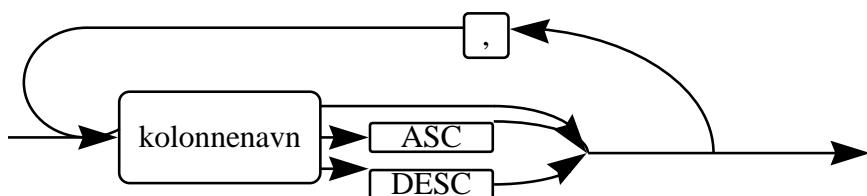
Verdi	Pos
1234	6
9000	1
9000	4
9000	5
9030	3
9100	2

### 12.2. Indeksdefinisjon - SQL.

Vi kan definere en indeks ved å se på følgende oversikt (kalles gjerne et syntaksdiagram)



<kolonnliste>:



DROP INDEX <indeksnavn>;

## 12.3. Indekser - fysisk databasedefinsjon.

Konseptuell  
definisjon

Uavhengig av databasetype bl.a.

- datamodell
- unikhet, nødvendighet

Logisk definisjon

- Logisk oppbygging, kun avh. av databasetype, bl.a.
- primærnøkler, fremmednøkler

Fysisk definisjon

- Fysisk utforming av databasen, bl.a.
- indeks
- størrelse på I/O-buffer
- fordeling av tabeller på disker

NB! Mange (de fleste?) slår sammen konseptuell og logisk.

- Disken deles opp i sider / blokker (pages / blocks) som vanligvis er den mengden data som blir overført pr. I/O-operasjon. Typiske verdier: 1024, 2048, 4096 etc.
- Blokkene ligger ikke nødvendigvis sekvensielt, men med pekere mellom seg, ser dermed sekvensielt ut.
- Dermed: henting av en rad kan innebære
  - en filaksess
  - ingen filaksess
  - mange filaksesser
- Prinsipp for relasjonsdatabaser: Enkelt sett fra brukerens side, kan være svært komplisert "bak kulissene".

## 12.4. Fullstendige/ufullstendige indekser

En indeks er **fullstendig/tett/dense**, hvis det er en indeks for hver verdi av nivået under. Dette må brukes hvis nivået under er usortert

## Fullstendig indeks

123	215	311	325	600
-----	-----	-----	-----	-----

## Usorterte basisdata

325	311	215	600	123
Karlsen	Bø	Thorsen	Hansen	Eide
Karl	Bente	Thor	Hanne	Else

Hvis vi f.eks. hadde 1 000 rader og ville finne ansatnr 421:

- Uten indeks må vi søke gjennom alle 1000 i basisdata for å være sikker på å finne den (eller at den ikke finnes).
- Med indeks kan man begynne på midten av indeksen, finne hvilken halvpart hvor 421 finnes, deretter finne halvparten av halvparten osv. til vi (evt.) finner riktig rad – dermed holder med noen få (max. 11 ganger). Ved å indekser indeksen igjen kan det gå enda raskere.

En indeks er **ufullstendig/non-dense/sparse**, hvis det er en indeks for hver x-te (f.eks. hver 100.) verdi nivået under. Forutsetter at nivået under er sortert.

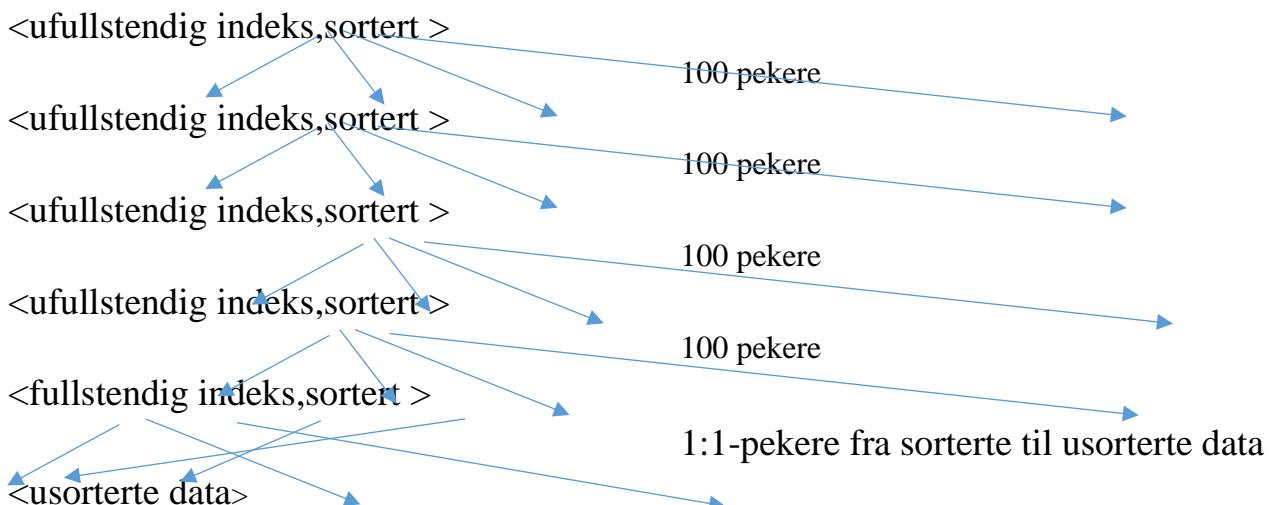
## Ufullstendig indeks (basisdata er sortert)

12	280	1398																
----	-----	------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

## Sorterte basisdata

12	45	78	111	268	299	305	326	277	280	299	321	345	378	432	541	733	882	869	950
...	...	...	...	...															

**I praksis:** hvis det er store datamengder, bør man ha en ikke-fullstendig indeks (evt. i flere nivåer) på en indeks igjen. Med 4 nivåer av ufullstendige pekere og 100 pekere pr. nivå kan vi finne  $(10^2)^4 = 10^8 = 100$  millioner rader!



## 12.5. Indekser - i praksis.

- Indekser er et av de viktigste elementene i "tuning" av systemer.
- Skal kun påvirke ytelsen på databasen, ikke om spørringen er mulig eller ikke.
- Hvilke indekser man bør ha er avhengig av bruken av databasen,  
==> del av fysisk design, ikke logisk design  
==> kan legges på / endres etterhvert som man ser bruken, flaskehals osv.
- Ved store oppdateringsjobber kan det lønne seg å fjerne (noen) indekser først, kjøre jobben, deretter reindeksere.
- Databasesystemet bestemmer om / hvilke indekser som skal brukes ved en spørring.
- det idelle: DBMS-et burde selv finne ut hvilke indekser som trengs ut fra statistikk på bruken, fjerne de indeksene som ikke trengs osv., m.a.o. "selvtunende" systemer.
- Såkalte B-trær (evt. B+-trær) er nesten blitt enerådende som indekseringsteknikk. Disse er flernivås, og stort sett selvutvidende og -skrumpende. Beskrevet av Bauer (1972).
- Rask aksess til svært mye data pga. hvis flere indeksnivåer.
- Kan likevel lønne seg å omorganisere indeksen ved å fjerne indeksen + reindeksere fra tid til annen. Noen systemer har REINDEX <indeksnavn>;

### NB! Indekser vil ofte være helt nødvendig i praksis

- spørring som ellers ville ta timer går på minutter
- spørringer som ellers ville ta minutter gjøres ”øyeblikkelig”

## 12.6. Noen variasjoner mellom ulike DBMS.

- Finnes analyseverktøy for å se hvorledes indekser brukes.
- Systemene er ulike m.h.t. når og hvordan indekser brukes, dvs. ulike måter å optimalisere f.eks. en spørring på.
- Noen systemer har klyngeindekser (CLUSTERED), data sorteres fysisk på indeksen i tillegg (eller i alle fall mest mulig sortert, f.eks. sortert innen hver "page", dvs. I/O-overføringsenhet).
  - ==> Kun en CLUSTERED indeks pr. tabell.
  - Svært rask mulig søk, men tregere ved endring av data.
  - Noen systemer: også CLUSTER som omfatter flere tabeller.
- Noen systemer har også andre tilleggsmekanismer for "tuning". NB! Dette er på fysisk nivå, ikke på logisk.  
eksempler:
  - direkte organiserte filer (radnr = primærnøkkelen)
  - fysisk resortering av filen f.eks. på primærnøkkelen.
  - hash-filer
  - pekere
  - spesialindekser til bruk for filer med mange like verdier
- I noen systemer kan man si fra USING .. <indeksnavn>, men dette er ikke obligatorisk (med noen få unntak i noen systemer).
- I noen systemer kan du angi

$$\text{fyllingsfaktor} = \frac{\text{antall rader}}{\text{avsatte plasser for indeks}}, \text{f.eks. } \frac{600.000}{1.000.000} = 0,6$$

Tips:

- høy fyllingsfaktor på tabeller med relativt stabile data
- lav fyllingsfaktor på " " data som endres raskt (flyktige data)

## 12.7. Hva bør indekseres?

### Alltid:

- Primærnøkkelen
  - Kolonner med unik-skranke
- begge deler gjøres automatisk i de fleste DBMSer.
- For unikhet: bedre å beskrive UNIQUE... (dvs. på logisk nivå) enn å lage en unik indeks.

### Nesten alltid:

- Fremmednøkler
  - gjøres autom. i noen DBMS.
  - dersom sammensatt PK/FK: Indeksér kolonnene i den rekkefølge de finnes i primærnøkkelen.
- Kolonner som brukes mye i spørninger, spes. ved = .....
- Kolonner som brukes mye i sorteringer eller som grupperingskolonne.

### Indeksere alt? Nei !

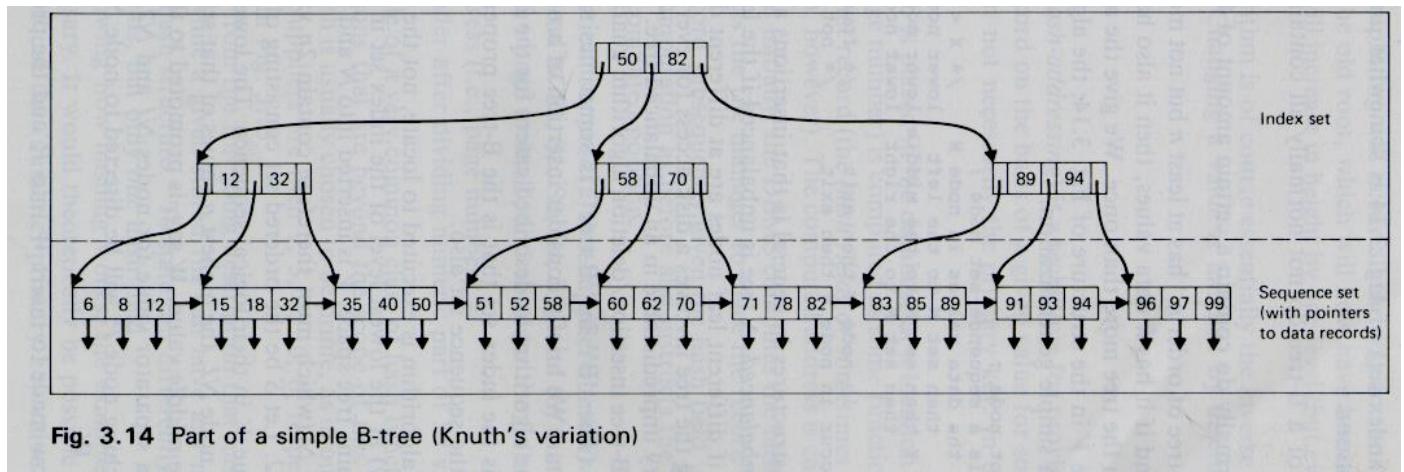
- indekser tar plass
- tar tid å oppdatere ved endringer av tabellen, etc.
- data med liten variasjonsgrad (mange like verdier) bør ikke indekseres. Indekser ikke Ja/Nei-kolonner !
- erfaring, testing etc. må til for å finne ut hva som bør indekseres. Bruk gjerne simuleringer med mye data for testformål før systemet leveres.

Det finnes systemer (ikke-relasjonelle) som baserer seg på alt er indeksert ==> trenger ikke selve basistabellen.

Kalles inverterte databaser ("vrenchte databaser").

## 12.8. Eksempel på B-tre

Her kommer et eksempel på et B-tre, hentet fra Chris Dates bok: Introduction to Database Systems. I praksis har hver blokk f.eks. 100 pekere, ikke bare 3 som her.



- Søking: Søk alltid i pekeren til venstre for verdien som er  $\geq$  den du leter etter.
- Dersom en blokk blir full, splittes den i to, evt. i flere nivåer (rekursivt).
- Dersom flere blokker blir mindre enn halvfulle, slås de sammen.

NB! Dette foregår ”bak kulissene” helt utenfor vår kontroll. B-trær er også vanlig i bruk i andre sammenhenger, f.eks. i filsystemet i et operativsystem, hvor det jo er viktig å kunne finne filer (i praksis fil-deler fordelt rundt på harddisken / SSD-en).

## 13. Adgangskontroll til en database.

### 13.1. Hensikten med adgangskontroll.

#### Hensikt: begrense adgang til tabeller etc.

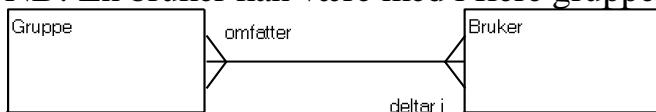
- for å forenkle en brukers syn på databasen
- for å hindre misbruk av data / innsyn til data
- å hindre utilsiktet ødeleggelse av data og struktur

#### Hvem ?

Det kan defineres adgangskontroll til tabeller, kolonner etc. for

- enkeltbrukere
- grupper

NB! En bruker kan være med i flere grupper og omvendt:



- grupper kan være forhåndsdefinerte (PUBLIC, DBA etc)

#### Aktuelle rettighetstyper:

SELECT  
DELETE  
INSERT  
UPDATE

} evt. på enkeltkolonner

REFERENCES  
USAGE

tillatelse til bruk av domener etc.

-----  
ALL

#### Rettighetene kan gjelde:

- tabeller, utsnitt (mest vanlig)
- indeks
- domener
- lagrede prosedyrer, triggere
- etc. - forskjellig fra system til et system til et annet

Vi kan definere hvem som skal ha adgang via SQL-språket. Vi kan også hente data f.eks. fra en brukerdatabase og lage mange samtidige endringer av rettigheter.

## 13.2. SQL-syntaks for adgangskontroll

### Gi rettigheter:

GRANT <privilegier> ON <tabell el.l.> TO <bruker/gruppeliste>

[WITH GRANT OPTION];

rettighet til å gi samme  
rettighet videre

Eks. på <privilegier>:

SELECT, INSERT, UPDATE (kundenavn), UPDATE(kundeadresse)

### Frata rettigheter:

REVOKE <privilegier>

ON <tabell el.l.> FROM <bruker/gruppeliste>

[RESTRICT eller CASCADE];

Ta bort rettigheter  
bare hvis det ikke får  
følger for andre. Er  
det et...

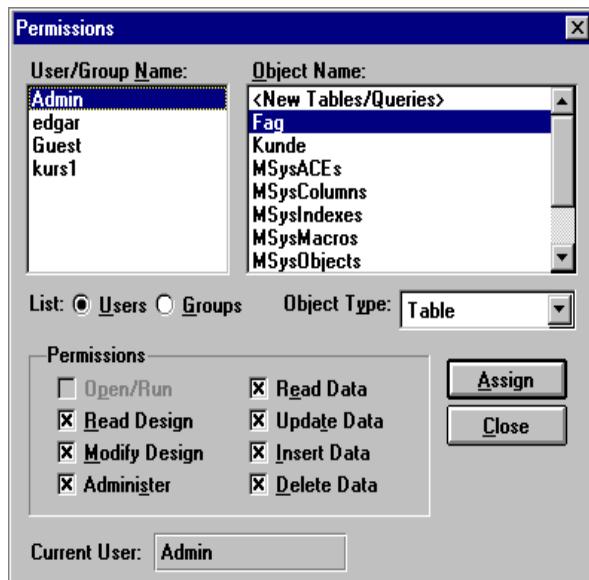
Ta bort rettigheter  
som andre har fått  
“via” denne  
bruker/gruppe.

Ta bort  
“videregivningsretten”

Med CASCADE kan altså også brukere som indirekte har fått sine rettigheter fra denne bruker/gruppe, miste sine rettigheter.

### Noen sluttkommentarer:

- Datasystemene er noe forskjellig på dette.
- I noen systemer kan brukere, grupper etc. samordnes med brukere og grupper i OS-et.
- I de fleste systemer kan man definere opp rettigheter via et grafisk grensesnitt, evt. bare ha det grafiske grensesnittet. Eks. fra Access: .....



## 14. Systemtabeller.

= systemkatalogen, datakatalogen, data dictionary.

### 14.1. Hva er en systemtabell?

Databasesystemene må lagre opplysninger om hvilke tabeller og filer som finnes etc. Disse lagres i egne tabeller, kalt systemtabeller. Typisk oppsett:

Systemtabeller  
(= metadata,  
2.ordens data)

SYSTABLES

Tabnr	Tabnavn	Kommentar	Type
4563	Kunde	Gjelder kun reg. ..	Table
4570	Ordre		
4571	...		View

SYSOLUMNS

Tabnr	Kolnavn	Type	P	NN	Len	Des
4563	Kundenr	3	Y	Y	4	0
4563	Kundenavn	1		Y	20	0
4563	Kundeadresse	1			40	0
4563	Omsetning	4			6	2
....						
4570	Ordrenr	3	Y	Y	..	
4570	.....					

SYSCONSTRAINTS ....

SYSPermissions..

etc.

Selv databellene  
(=rådata,  
1.ordens data)

KUNDE

1	Olsen	..	100
2	Hansen	..	500
3	Jensen	..	300
4	Nilsen	..	120
5	Asen	..	210

ORDRE

1000	12.09.03	..	1
1001	13.09.03	..	4
1002	13.09.03	..	4

Kommentarer:

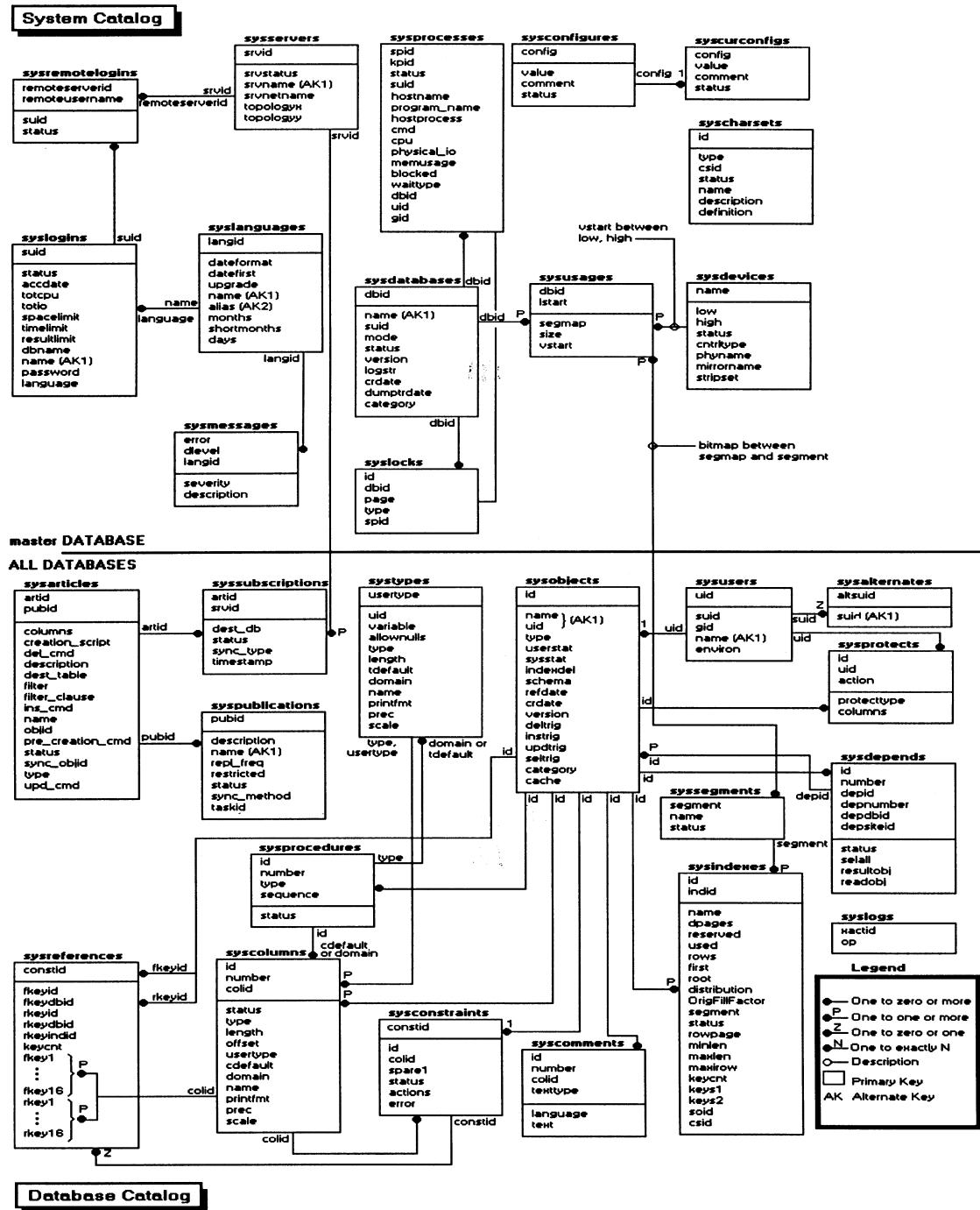
- Navn, hvilke tabeller som finnes etc. kan variere fra system til system.
- Hvorvidt disse opplysningene virkelig lagres fysisk som tabeller, kan vi ikke vite, men logisk framstår de for oss som tabeller.
- Kan selektere fra disse (SELECT \* FROM syscolumns, .. WHERE.. )
- Systemtabellene opprettes og vedlikeholdes av databasesystemet, vi kan ikke endre disse (bortsett fra via CREATE/DROP/ALTER ...).
- Vær obs. på at systemtabellenes egne definisjoner også finnes i systemtabellene. (Systables inneholder definsjonen av systables etc.)
- I noen databasesystemer er (noen av) systemtabellene ikke synlige for vanlige brukere.

## 14.2. Eksempel: systemtabeller fra et reelt system

Systemtabellene i MS SQL Server (hentet fra hjelpefilen)

## System Tables Diagram

The following illustration shows the relationships among the system catalog and database catalog tables.



## 15. SQL-språket – sammenfatning

### 15.1. Parallelitet mellom metadata og data

SQL inneholder<sup>16</sup> (ikke helt fullstendig oversikt):

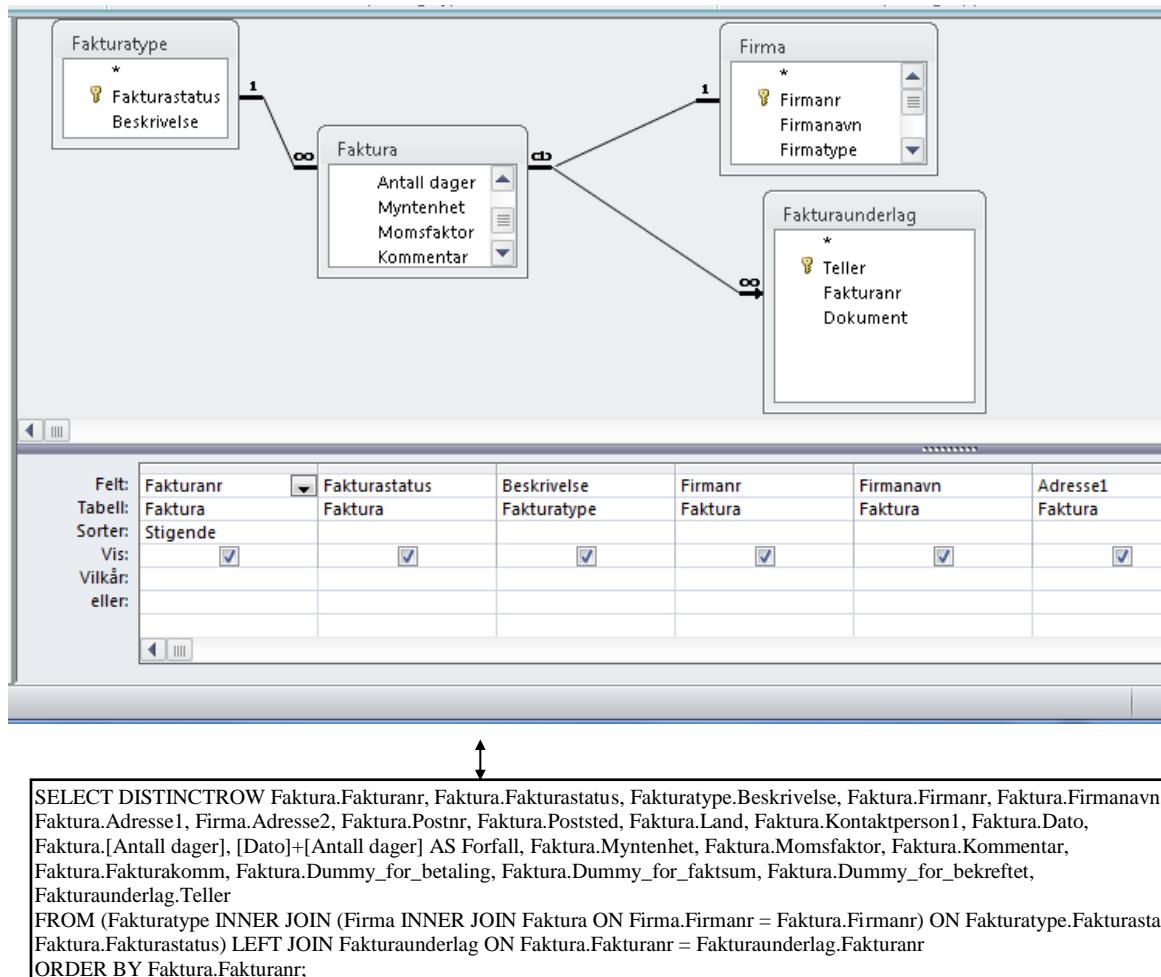
	Datadefinisjon (DDL, metadata)	Datamanipulasjon (DML)
Lage/slett database	CREATE DATABASE .. DROP DATABASE..	
Domener	CREATE DOMAIN .. DROP DOMAIN	
Opprette / legge til	Lag ny tabell CREATE TABLE ..., med primærnøkler, fremmednøkler, not null etc.	Legg inn ny rad INSERT INTO <tab.navn> VALUES (.....)
Endre	ALTER TABLE ADD ... <sup>17</sup>	UPDATE ... SET .... = ..... , ... = ...
Slette	DROP TABLE ...	DELETE FROM ...
Spørre / hente ut	Kan bruke DML for å spørre på systemtabeller, evt. bruke systemspesifikke kommandoer.	SELECT <kolonne(r), evt. med operatorer> FROM <tabeller> WHERE <betingelser> GROUP BY <kolonne(er)> HAVING <gr.beting.> ORDER BY <kolonne(er)>
Utsnitt	CREATE/DROP VIEW	
Indekser	CREATE/DROP INDEX	
Brukere og rettigheter	CREATE/DROP GROUP / USER ... GRANT/REVOKE ... TO/FROM ...	
Annet	SYNONYM, SNAPSHOT etc.	COMMIT,ROLLBACK

<sup>16</sup> Mye parallelitet mellom DDL og DML: samme operasjoner, hhv. på metadata (“struktur”) og data.

<sup>17</sup> Endel systemer har begrensinger på hva som kan endres. Access o.l. er relativt fleksibel, mens større systemer ofte ikke tillater sletting av kolonner o.l. Må i tilfelle flytte data over på en annen tabell midlertidig.

## 15.2. Noen bruksområder for SQL

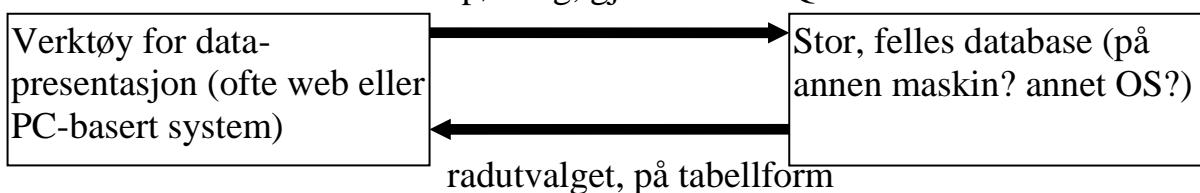
- definsjons- og manipulasjonsspråk (innsetting, spørring..) for brukere/utviklere, ofte interaktiv
- som utvidelse av et vanlig programmeringsspråk ("embedded SQL")
- tekstlig beskrivelse av grafisk spørring el.l. Eksempel fra Access:



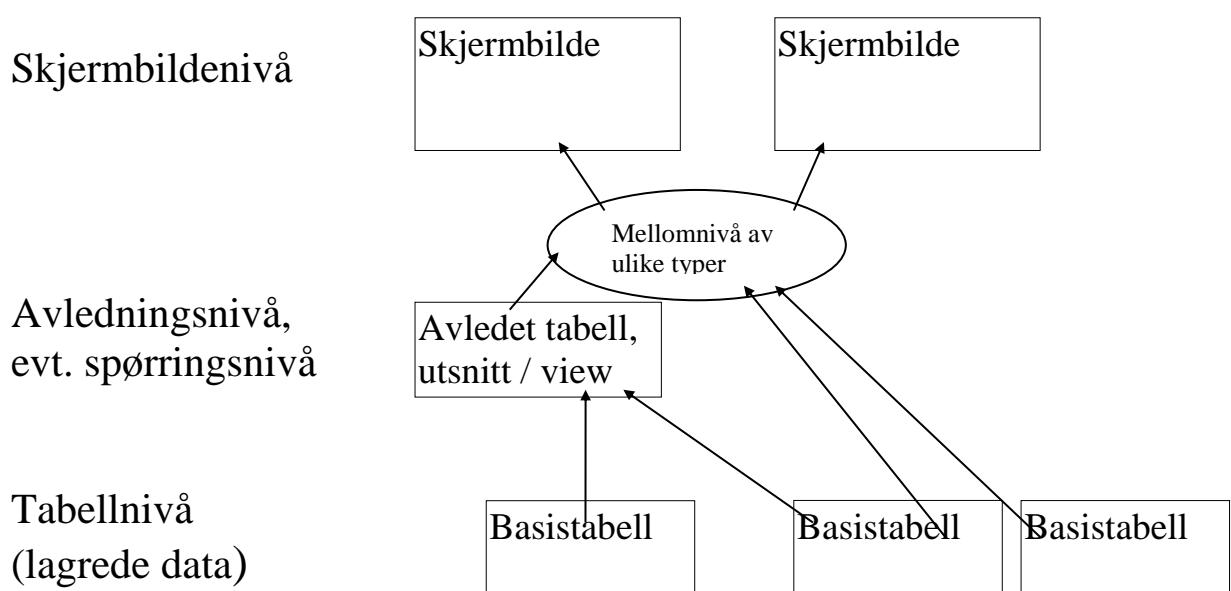
- felles "kommunikasjonsspråk" mellom ulike systemer, f.eks. i et klient-tjener-system.

Eksempel:

spørring, gjort om til SQL



### 15.3. Typisk arkitektur for et SQL-basert system



- Avledeede tabeller / views fungerer stort sett som vanlige tabeller, bortsett fra evt. begrensninger i oppdaterbarhet.

## 16. Datamodellering

Når vi snakker om datamodeller i forbindelse med databaser, er det gjerne oversikter «ett nivå over» selve databasene. Som regel bruker vi en grafisk notasjon om gir en god oversikt over en planlagt eller eksisterende database. Når man skal lage nye systemer eller endre eksisterende systemer, er det å lage en god datamodell svært viktig for å få en oversikt over de logiske sammenhengene som finnes.

Disse notatene er kun en oversikt over en del prinsipielt stoff innen datamodellering.

Disse må kompletteres med

- mer om aktuell(e) notasjon(er) som brukes (her finnes bare en grov oversikt).
- mer om vanlige struktur (noen eksempler, uten forklaring finnes i de siste sidene)

Etter min mening er dette stoff som best foreleses muntlig, ikke bare skrives.

Det er viktig å understreke at selv om selve notasjonen som brukes her (såkalt kråkefot-notasjon) er svært enkel å lære, så kreves det mye jobb for å bli god til å datamodellere i praksis. En naturlig parallel: det er lett å lære seg de enkle grepene på en gitar, men det tar tid å bli en god gitarist!

Det finnes en rekke systemer som hjelper oss til å lage datamodeller, og gjerne også lage databaser automatisk ut fra denne, evt. gjør en «reverse engineering» fra database til datamodell. Dersom man bruker MySQL, kan det være naturlig å bruke MySQL sin Data Modeling Tool.

Undertegnede har også laget et slikt verktøy, kalt Modelator.

### 16.1. Datamodellering – prinsipielt

#### Hva er en datamodell?

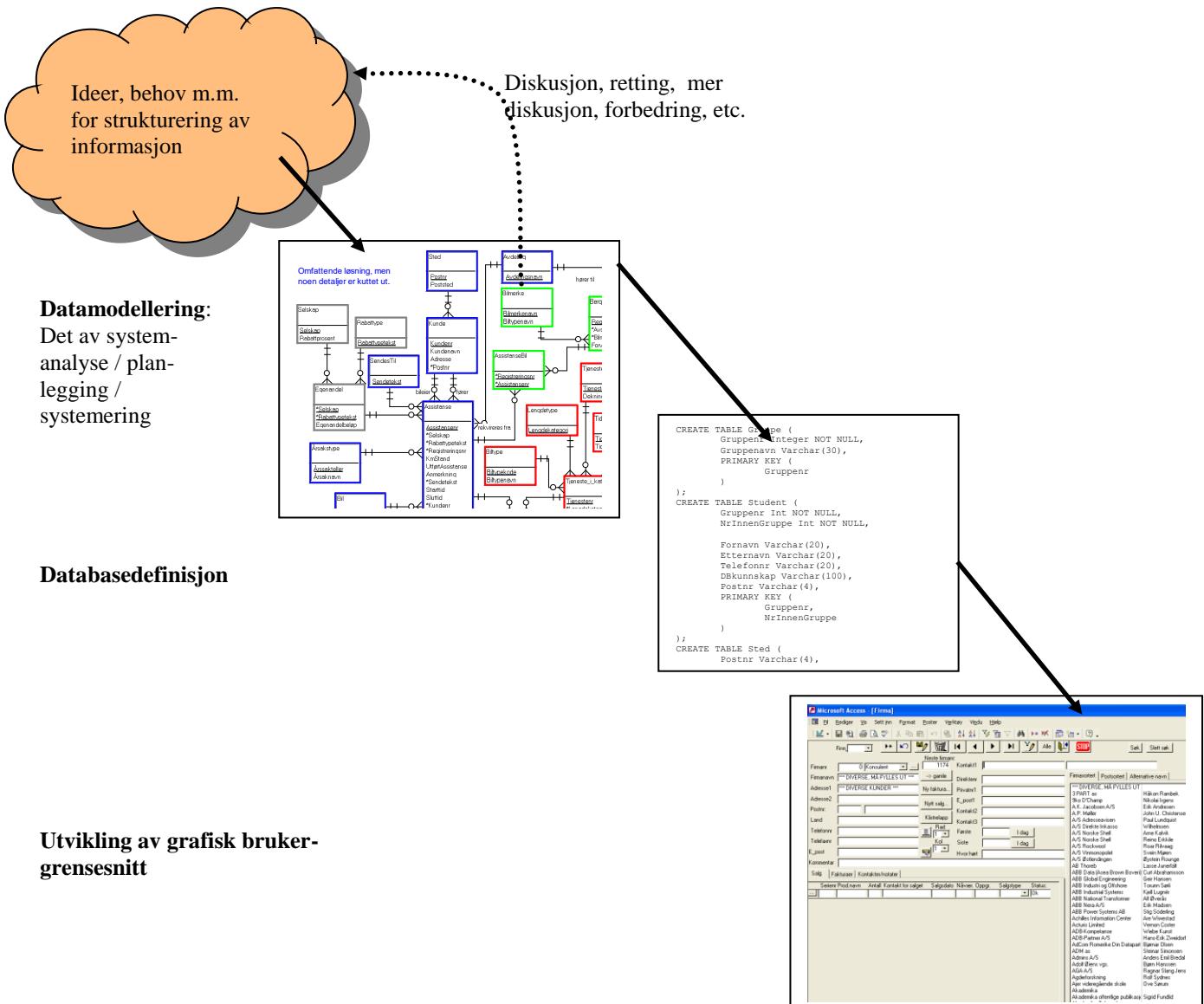
Noen fokuserer på

Andre på

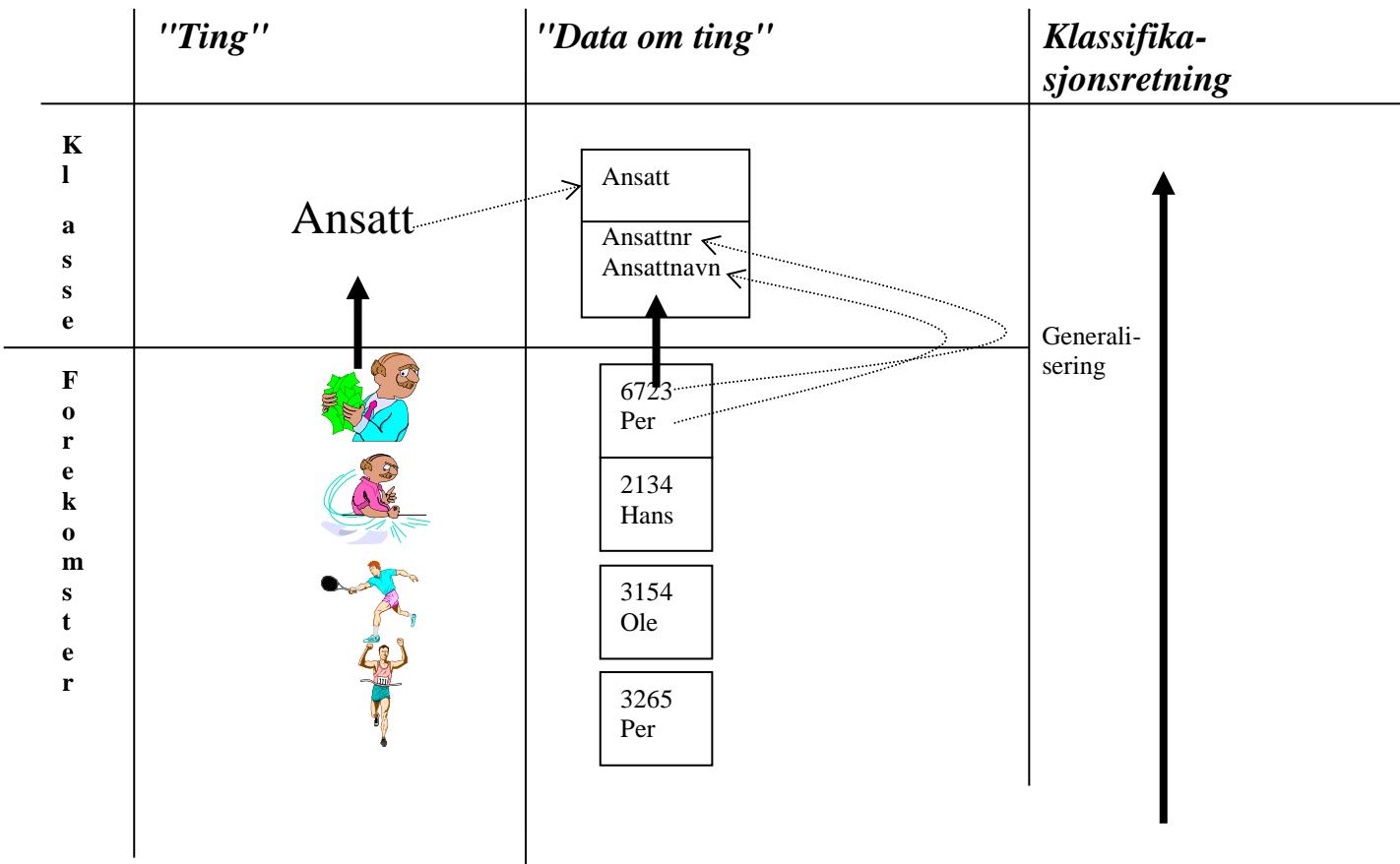
Beskrivelse av en  
informasjonsstruktur

Tegning av en  
databasestruktur

# Mest vanlig bruk i praksis:

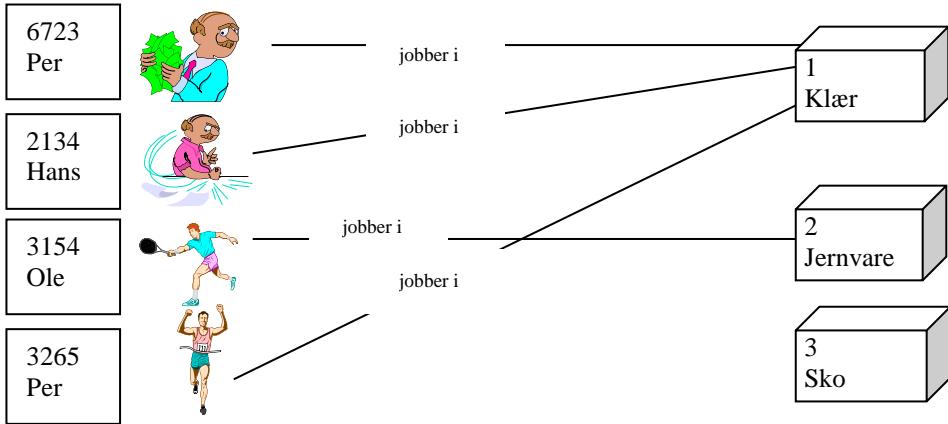


## Fra enkeltforekomster til typer / klasser av "ting"



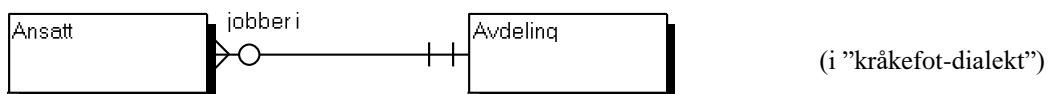
Beskrivelse	Symbol	Formelt begrep
Type/klasse av tingen (hele boksen)		entitetstype
Navn på denne klassifikasjonen		entitetstypenavn
Interessante data for denne typen/	<pre> classDiagram     class Ansatt {         string Ansattnr         string Ansattnavn     }   </pre>	attributter

## Fra enkeltsammenhenger til typer / klasser av sammenhenger.



Hver av strekene er en relasjon mellom to "ting". Siden disse er mellom de samme ting av samme type, er det naturlig å gjøre en tilsvarende klassifikasjon som vi gjorde for entitetstyper.

Hvis vi forutsetter at hver ansatt jobber i en og bare en avdeling, og at avdelinger i alle fall av og til kan være uten ansatte, ser vi at denne relasjonstypen blir min. 0 max. mange på venstre side, min. 1, max. 1 på høyre side, altså:



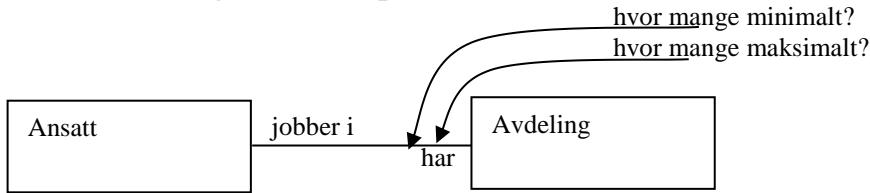
## Litt om maksima og minima

Vi ønsker altså å kartlegge hvor mange av "noe" som er knyttet til "noe annet":

- er det noen begrensninger i **hvor mange** av "noe" som kan knyttes til "noe annet" - dvs. maksimalt antall. Vanligvis er det snakk om
  - **maksimum 1**
  - **maksimum mange**, dvs. ingen begrensninger på hvor mange.
- er det noen begrensninger i **hvor få** av "noe" som kan knyttes til "noe annet" – dvs. minimalt antall. Vanligvis er det snakk om
  - **minimum 0**, dvs. ingen begrensninger på hvor få, **"KAN"**
  - **minimum 1**, m.a.o. at noe må være knyttet til minst en annen, **"MÅ"**

I noen tilfelle er det interessant å oppgi et bestemt antall, f.eks. at en bil kan ha minimum 3, maksimum 4 hjul. Slikt må imidlertid spesialbehandles i databasen (via såkalte triggere), i applikasjonen som bruker databasen eller i program mellom databasen og applikasjonen.

Videreføring av eksempelet over:



### Fra Avdeling til Ansatt:

Hvis der er slik at hver avdeling kan ha fra 0 til uendelig mange ansatte, kan dette mer formelt skrives som:

- **Avdeling har minimum 0, maksimum mange Ansatt**
  - bør vi anta at alle avdelinger har minst en ansatt? P.d.a.s. kan det kanskje finnes nyopprettede avdelinger uten ansatte?
- **Fra Ansatt til Avdeling:**

Hvis hver ansatt må jobbe i en gitt avdeling, og at den ansatte også maksimalt kan jobbe i en avdeling, blir det mer formelt:  
**Ansatt jobber i minimum 1, maksimum 1 Avdeling**

**Men: er dette eneste mulighet? Neppe!** Vi kan tenke oss

- **Ansatt jobber i minimum 1, maksimum 1 Avdeling**
  - (som over)
- **Ansatt jobber i minimum 0, maksimum 1 Avdeling**
  - (det er en eller flere som ikke er koblet til noen avdeling)
- **Ansatt jobber i minimum 1, maksimum mange Avdeling**
  - (fordeler arbeidstiden mellom 1 eller flere avdelinger)
- **Ansatt jobber i minimum 0, maksimum m Avdeling**
  - (fordeler arbeidstiden mellom 0 eller flere avdelinger)

Hva som er ”riktig” vil selvsagt avhenge av hva som er situasjonen i det firmaet vi skal lage databasen for, dvs. for oppdragsgiveren. Dette må vi finne ut ved samtale med oppdragsgiveren, eller fra en skriftlig beskrivelse av det påtenkte systemet.

**NB: Noen er svært nøyne med alltid å spesifisere både maksimum og minimum, andre spesifiserer stort sett bare maksimum.**

### Oppsummert: vanlige minima og maksima

- Minimalt antall: vanligvis 0 (ingen begrensning) eller 1.
- Maximalt antall: vanligvis 1 eller mange (ingen begrensning) mange.

Hvor få minima (Min)



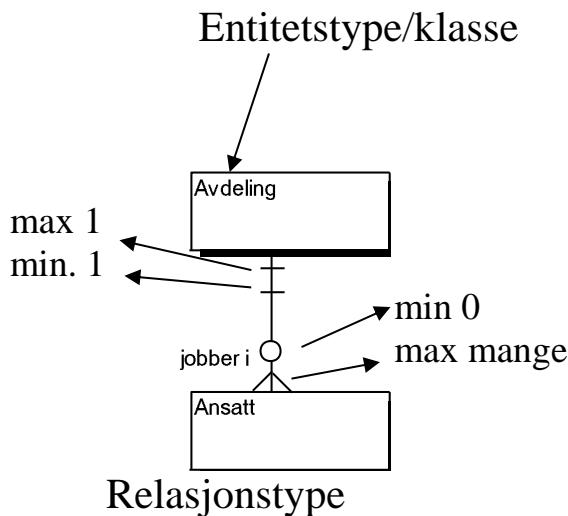
Hvor mange maksimalt (Max)

$\Rightarrow$  Min 0, max mange er egentlig ikke noen begrensning i det hele tatt.

Minimum og maksimum vises ulikt i ulike datamodelleringsdialekter.

## 16.2. Grunnleggende notasjon, som kråkefot hhv. UML-basert.

### ”Kråkefot”



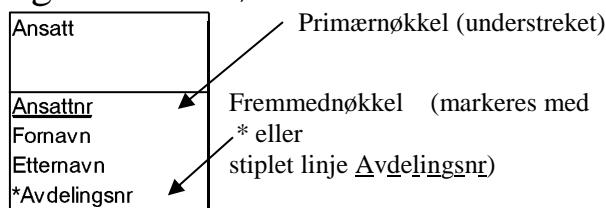
verb/rolle som beskriver hva rel.  
gjelder (en eller begge veier)

- Minimum kan sløyfes, men gir mindre nøyaktig modell.
- Verb eller rolle kan sløyfes, men er ofte nyttig for å forstå hva relasjonstypen gjelder.

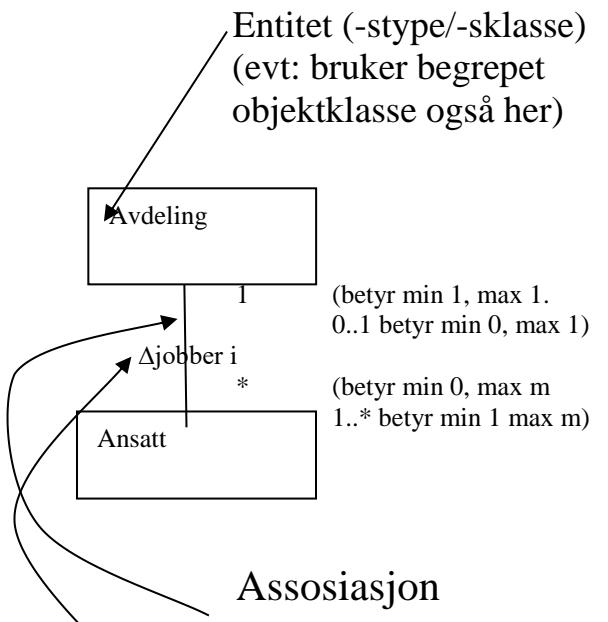
Merk paralleliteten med ”vanlig” språk, f.eks:

Ansatt jobber i min. 1, max. 1 Avdeling  
og  
Avdeling har min 0, max mange Ansatt.

Med primærnøkkel (identifikator)  
og fremmednøkkel:



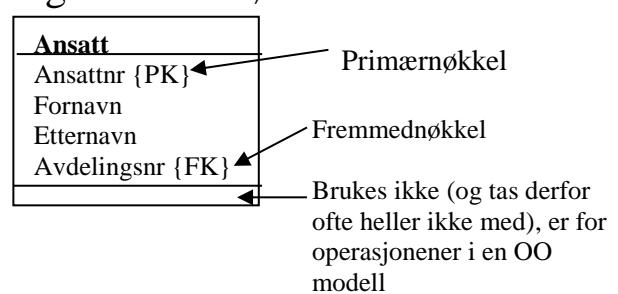
### ”UML-basert”



relasjonsnavn (en eller begge  
veier, Δ viser retningen)

- Hvis minimum sløyfes, blir kan det være usikkert om ”1” betyr min. 0 eller min. 1.
- Relasjonsnavn kan sløyfes, men er ofte nyttig for å forstå hva relasjonstypen gjelder.
- Notasjonen er en delmengde av klassediagram.
- Fordel: samme notasjon.
- Ulempe: man kan lett forledes til å tro at tankemåten ved modelleringen er lik.

Med primærnøkkel (identifikator)  
og fremmednøkkel:



Opplysningene (tilsvarende kolonner i databasen) kalles vanligvis attributter.

## Legg merke til forskjellen

- Entitetstyper er ”**tingene**” i seg selv, f.eks. en Ansatt.
- Attributtene er ”**opplysninger/egenskaper om tingene**”, f.eks. Ansattnr, fornavn, alder, sivilstatus o.l. Dette kan på en eller annen måte gis en verdi.

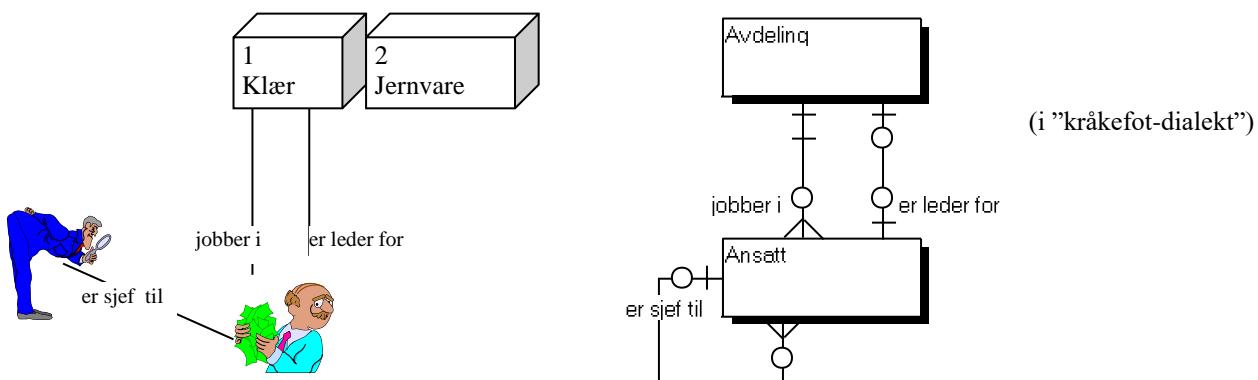
## Primær- og fremmednøkler i modellen?

Det er ulike meninger om man bør ha med primær- og fremmednøkler i datamodellen

- hele tiden (betyr at man tenker relasjonsdatabase hele tiden)
- etter hvert
- bare når (hvis) man skal overføre modellen til en database.

## Flere relasjoner mellom de samme entitetene (og dermed også –typene/klassene)

Det kan godt finnes flere relasjoner mellom de samme entitetene, og til og med relasjoner mellom to entiteter av samme type. Det siste kalles gjerne egenrelasjoner.



På type-nivå har vi dermed

- Ansatt *jobber i* Avdeling,
- Ansatt *er leder for* Avdeling
- Ansatt *er sjef til* Ansatt.

Relasjonstypene kan altså være av flere former:

**en-til-mange**

 **mange-til-mange**  
 **en-til-en**

(må splittes før overføring til database)  
(relativt sjeldent i praksis)

**Dessuten:**  
**alle-til-alle**

(beskrives sjeldent i modeller, men er etter min mening viktig!)

### 16.3. Forholdet mellom ting og navn på ting.

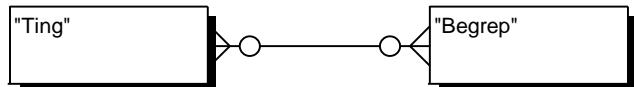
Vi tror gjerne at begrep er temmelig entydige, og at man mener det samme eller samme ting med samme begrep. Dette er ofte ikke tilfelle. Vi bør derfor være nøyne med begrepsbruk.

Noen trivielle eksempler:



Tegningene på venstre side brukes som bilde på selve tingene, mens navnene, eller betegnelsene står til høyre.

Forholdet mellom "ting" og "navn på ting"/begreper er altså, sagt med en "kvasi"-datamodell:



Vi avslutter dette temaet med René Magrittes berømte «Dette er ikke en pipe» (1928/29). Poenget er jo at det vi ser er en tegning av en pipe, ikke en pipe.

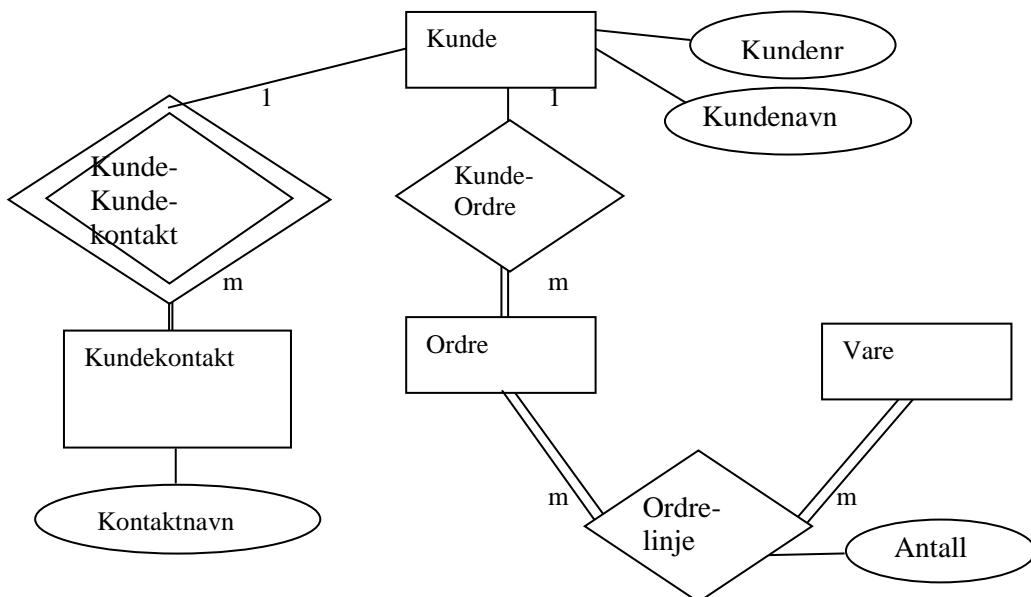
Det vi lagrer i en database, er data om ting, ikke tingene selv!  
Det er jo selvsagt ikke hele Nederland som finnes på denne sidene, men derimot en grovtegning av omrisset av Nederland.



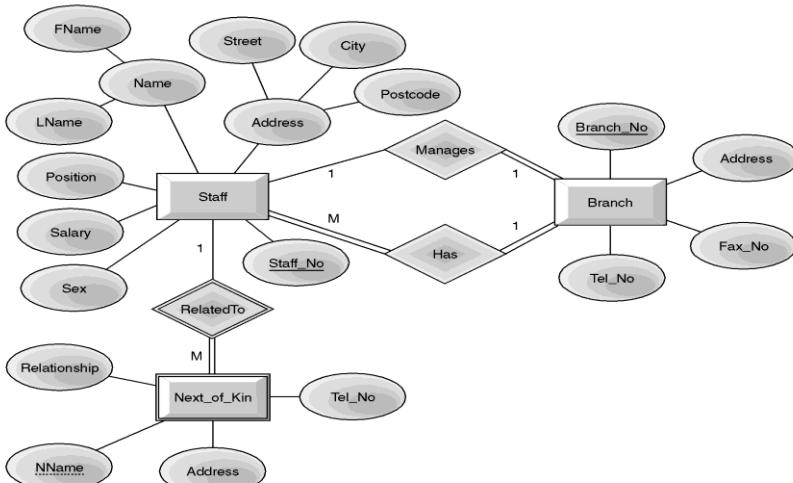
## 16.4. Ulike notasjoner/«dialekter» (orienteringsstoff)

Det finnes en rekke ulike måter å tegne datamodeller på. Her følger en oversikt.

### Chen's opprinnelige notasjon:



Et annet eksempel følger her:

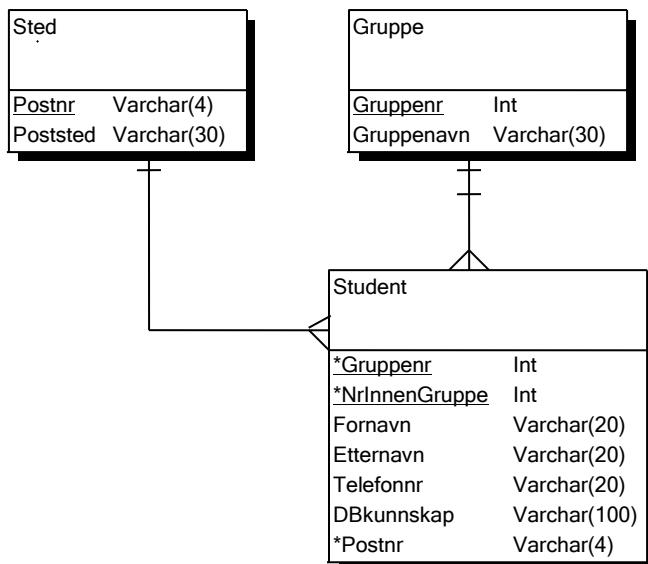


Denne notasjonen har bl.a. super / subtyper, sterke og svake entitetstyper m.m.  
Kan tegne på eller utelate attributter i modellen

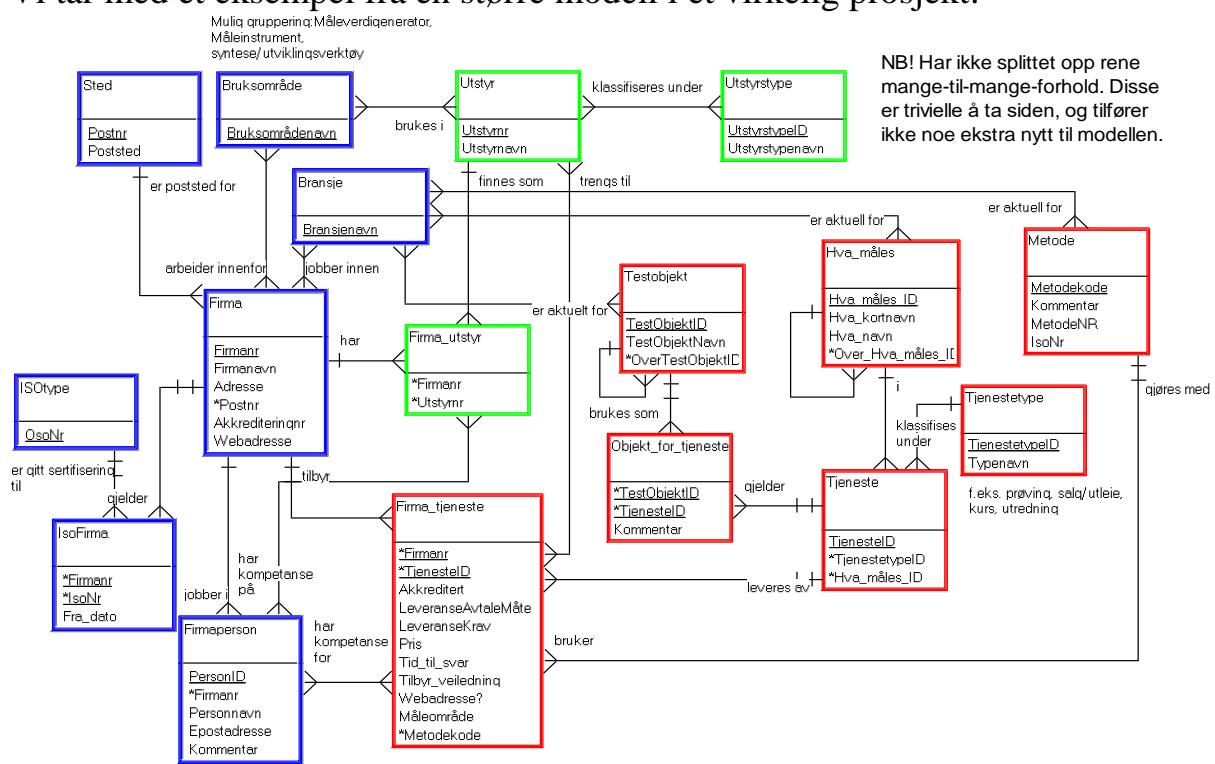
- Se f.eks. Jeffrey A. Hoffer, Mary B. Prescott, Fred R. McFadden: Modern Database Management, 6/e

# Kråkefot:

(Kråkefot er kanskje den enkleste teknikken og den som er nærmest relasjonsdatabasetenkning, men kan ikke beskrive alle avanserte betingelser/skranker.)



Vi tar med et eksempel fra en større modell i et virkelig prosjekt:



Vi kan vurdere om vi tegner attributter direkte i modellen, eller om vi lager det på en egen liste ved siden av.

Databasediagrammer som finnes i noen databasesystemer er i virkeligheten på kråkefot-form (selv om ”mange” kanskje tegnes som  $\infty$  el.l.).

- Se f.eks. Edgar Bostrøm: Datamodellering, praksis og teori.

# Forenkling av klassediagram fra UML

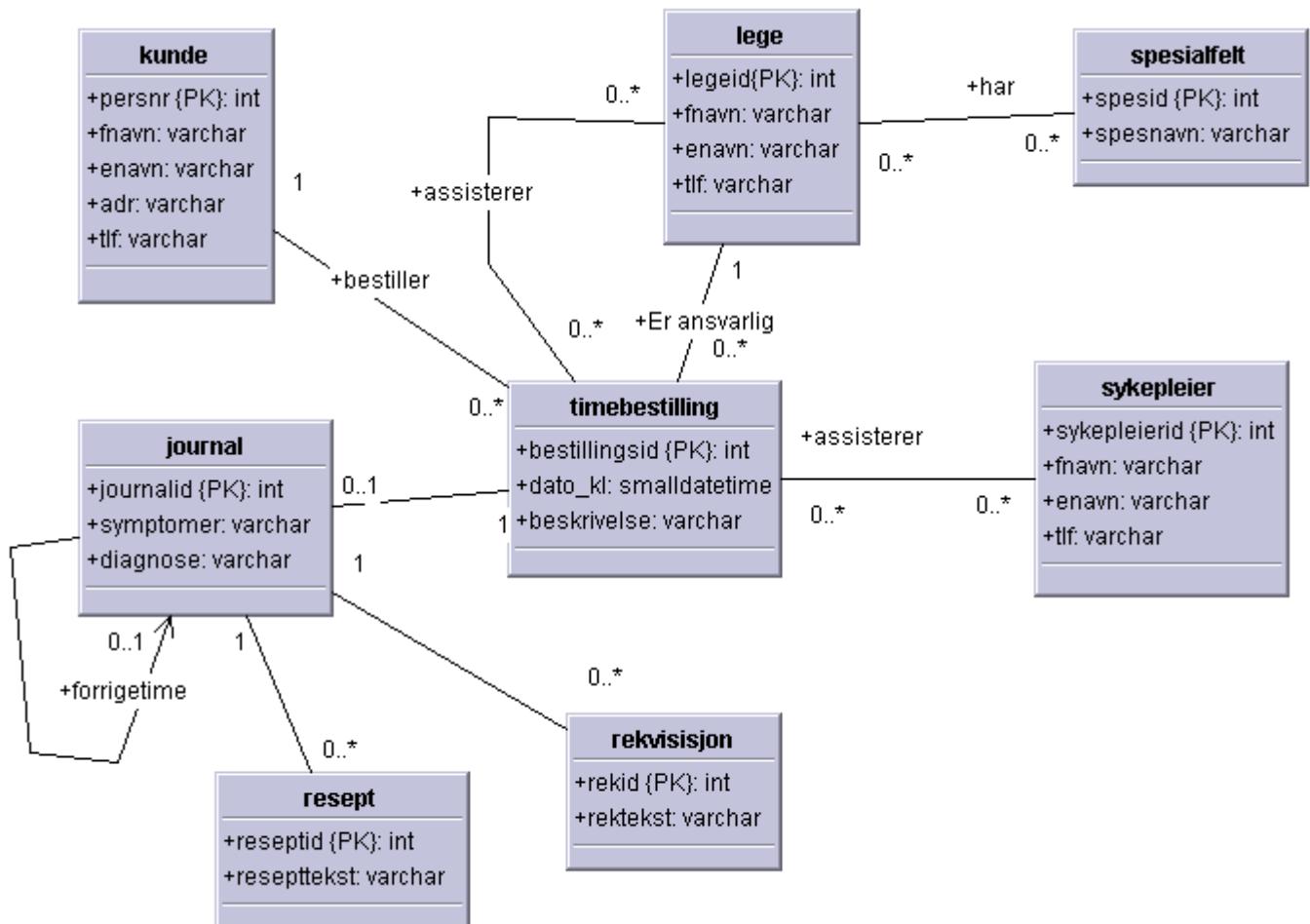
Klassediagram er en sentral del av UML, Unified Modeling Language. Disse beskriver bl.a. objektklasser, med

- klassenavn
- attributter (data)
- operasjoner (metoder)

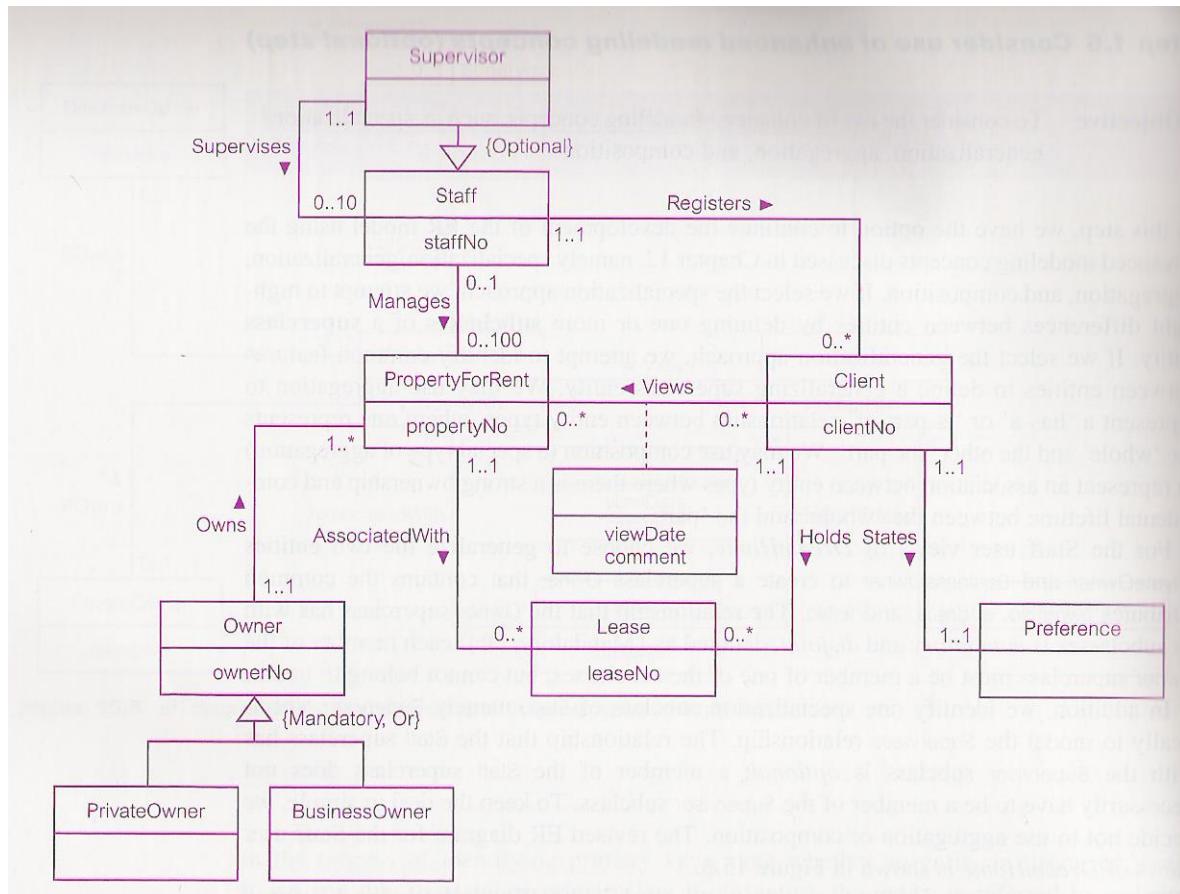
Faktura
Fakturanr
Fakturadato
....
LagNy
SkrivUt
UtsettSending (antallDager)

Hvis vi dropper operasjoner, kan dette ses på som et datamodelldiagram).

En UML-modell hvor bare ”standard datamodellering” er med kan se slik ut:



## Et mer omfattende eksempel:



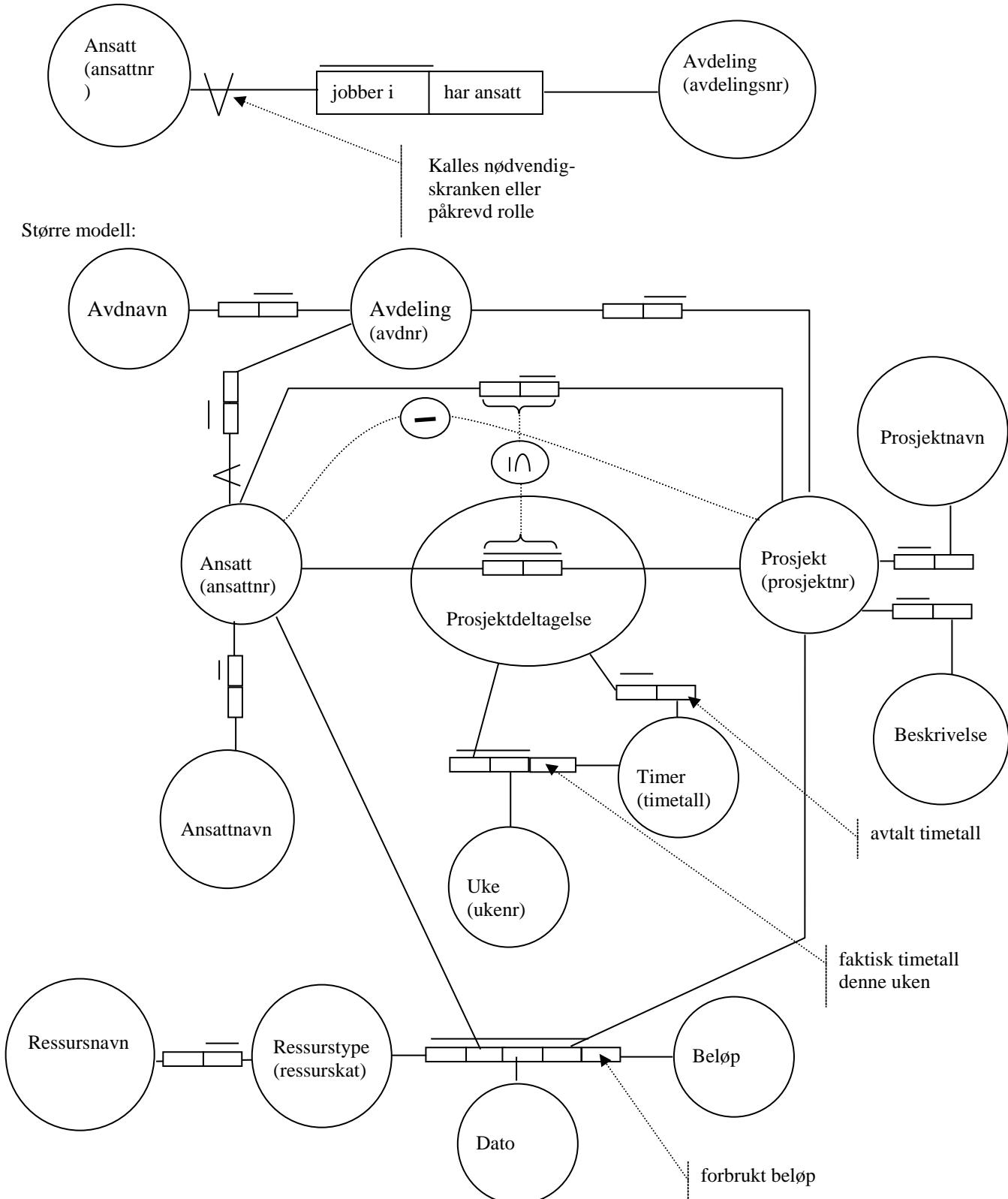
Teknikken er mer omfattende (f.eks. arv, komposisjoner m.m.).

I tillegg til det som er vist over, kan høyere ordens assosiasjoner (relasjoner) markeres med en rombe (som i Chens's notasjon)

- Se f.eks. Connolly & Begg: Database Systems (bruker egentlig en kombinasjon begreper fra UML og Chen).
- I tillegg finnes naturligvis en rekke bøker om UML, men disse er (i alle fall som regel) ikke fokusert på data/databaseaspektet.

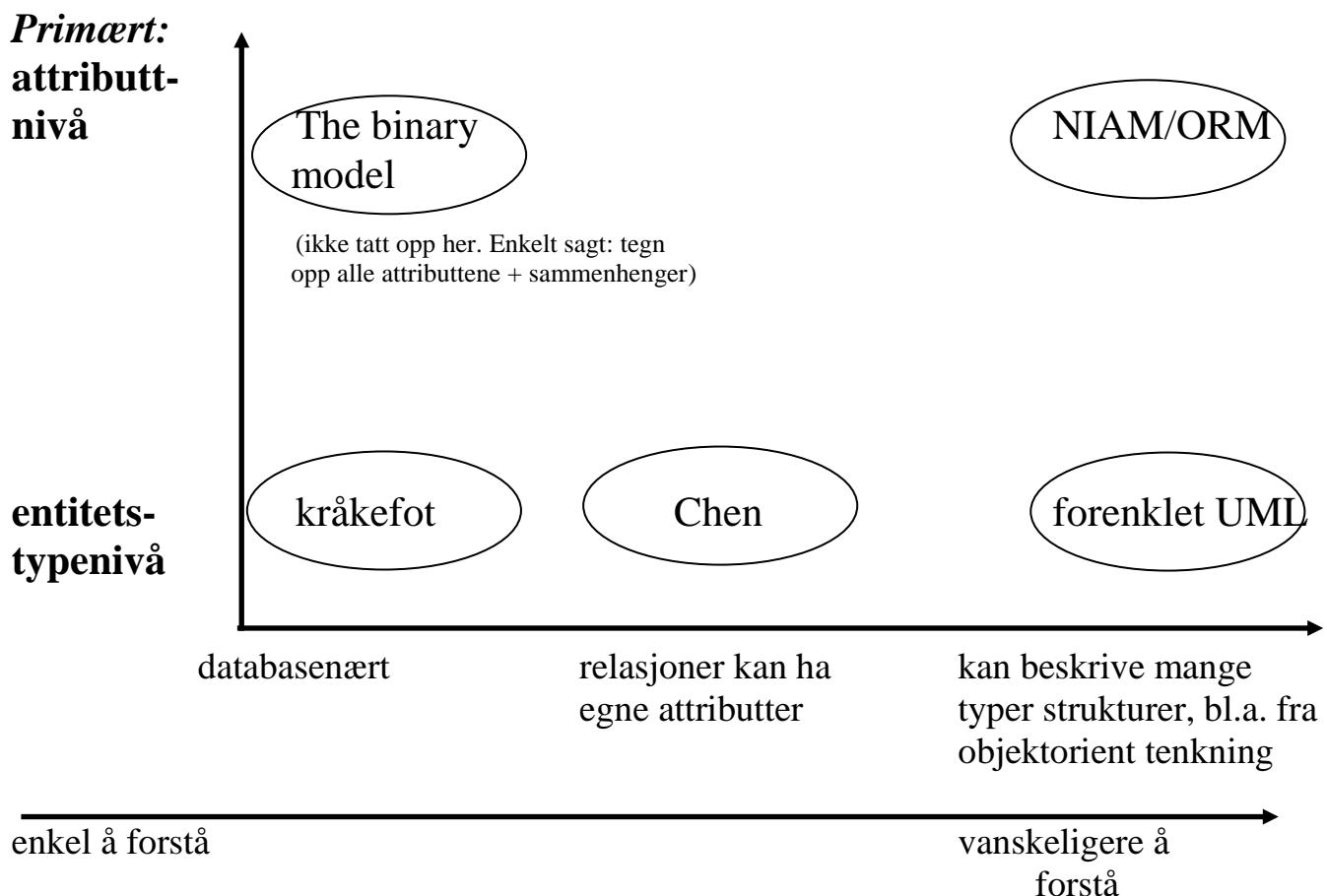
# NIAM (Nijsen's InformasjonsAnalyse Metode) / ORM (Object Role Model) (orienteringsstoff)

(svært presis, på ”attributtnivå”, men kan bli på litt for detaljert nivå)



- Se f.eks. Gerhard Skagestein: Systemutvikling. Cappelen forlag. 1. utgave. 2. utgave bruker UML-basert notasjon.

## Klassifikasjon av datamodelleringsdialekter, litt forenklet (orienteringsstoff)



Samtidig må man være klar over at den reelle utfordringen er å kunne bruke språket i komplekse sammenhenger, ikke å kunne symbolene.

## I tillegg er det ”naturligvis” mange varianter, bl.a.

### Skal det gjøres et skille mellom ulike stadier av utviklingen (jf. tidligere)?

- konseptuell (databasetypeuavhengig)
- logisk (databasetypeavhengig)
- fysisk design (da er vi egentlig dypt nede i databasedesign)

### Symbolbruk og mekanismer.

- hvilke symboler skal i tilfelle brukes?
- hvorledes markeres en og mange?
- skal ”minima” markeres, i tilfelle hvorledes?
- hvilke avanserte mekanismer bør finnes?
- hvilke skranker bør kunne modelleres?
- begrepsbruk: entiteter eller -typer, -klasser? Relasjoner, sammenhenger eller assosiasjoner?

### Og: det finnes verktøy for tegning av datamodeller

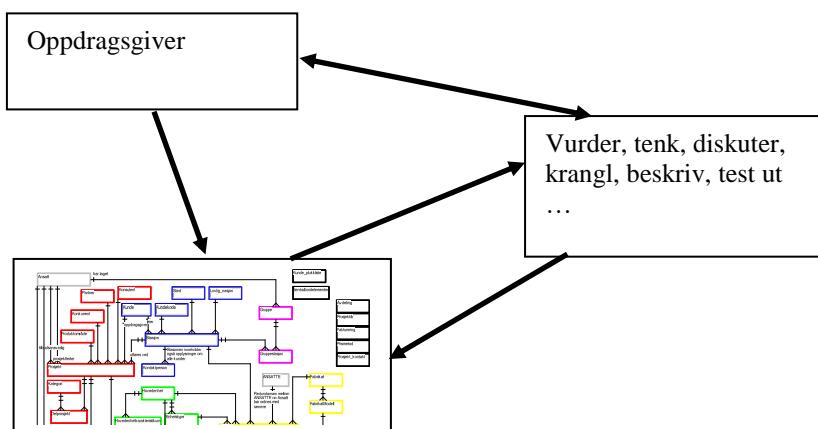
- generelle tegneverktøy (du kan tegne alt mulig)
- verktøy for mange teknikker (f.eks. innen UML)
- spesialverktøy for datamodellering

større eller mindre grad av automatikk, designhjelp osv.

## 16.5. Generell metodikk (kort)

### Metodikk, generelt

- Det er **kunden** som vet hvilken informasjon som trengs i systemet
- Vi er **metodespesialister**, men må passe på at vi ikke overkjører kundene. (Stikkord: **modellmakt**)
- Legg mye vekt på **diskusjon av begreper** (blir evt. tabellnavn siden)
- Jobb på overordnet nivå lengst mulig (normalt entitetstyper/objektklasser og relasjonstyper/assosiasjoner)
- Etter hvert vil da attributter ”falle på plass av seg selv”.
- Vær forberedt på **mange** iterasjoner.
  - **diskuter** med kunden, diskuter med kollegaer, få tak i en hovedkritiker.
  - idealt er at du får full oversikt med en gang, men det er sjeldent realistisk
  - mer realistisk:



## Antall vs. antall unike



- Jubileet har bidratt til at Sarpsborg har blitt mer attraktiv for innbyggere, næringsliv og besøkende, konstaterer rådmannen i sitt saksframlegg for politikerne.

Publikumsoversikten var det knyttet litt spenning til. Sluttrapporten viser at nærmere 104.000 mennesker stilte opp på de store markeringene i regi av kommunen og iSarpsborg.

I tillegg sto lag og frivillige for en rekke arrangementer, totalt ble det registrert 162 jubileumsarrangementer i 2016.

Kommunen har skaffet publikumsoversikt for langt på vei alle disse. Den viser at over 200.000 mennesker besøkte arrangementene i 2016.

Her er det neppe snakk om 200 000 ulike personer, det har temmelig sikkert vært samme personer flere ganger.

Tilsvarende, hentet fra nsb.no

På NSB Region tog på Østfoldbanen (Oslo - Halden) var det 7,1 prosent flere passasjerer i 2015, sammenlignet med 2014. I alt reiste 2.504 000 passasjerer med NSB Region tog på strekningen Halden - Oslo i 2015. Reisende på NSB Lokaltog på Østfoldbanen er ikke regnet inn.

og f-b.no/byfergen

### TALLENE

Så mange har reist med byens ferger hittil i år (2016-tall for samme periode i parentes):

Gamlebyferga: 116.262 (99.876)

Byfergene: 96.213 (65.181)

Selbak-Lisleby: 21.437 (15.436)

Ekstra: 52 (4.706 - Anno, cup og lignende)

Totalt: 233.964 (185.199)



**Han har klippet  
75.000 hoder**

Lik Fredriksstad Blad på Facebook

Poenget alle gangene er at man må skille mellom antall personer og antall reiser (evt. besøk). Det er neppe 116262 ulike personer som har reist med Gamlebyferga! Ordet passasjer er etter mitt syn også flertydig.

## Begreper i ulike abstraksjoner, f.eks. Vare

**Begrepet Vare kan det bety ulike ting, bl.a. (tegn gjerne datamodell!).**

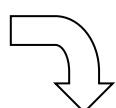
- Varegruppe (f.eks.: hvilke varer selger Elkjøp – jo hvitevarer, kjøkken, elektro..).
- Varetype (f.eks. TV, PCer, vaskemaskiner)
- Vare/artikkel (f.eks. Asus VivoStick PC TS10-B004D)
- Vareeksemplar/individ (f.eks. min TS10-B004D som har serienr 88266611-121)  
... og sikkert flere muligheter. Tegn datamodell selv!).

Tilsvarende f.eks. med begrepet Kurs (Kurs?, Kursavholdelse?, ....) m.m.

# Tenk gjerne prototypeorientert!

- I forbindelse med databaser må du kanskje lage en eller flere databaseprototyper, for dermed å vinne erfaringer.
- Noen ganger kan det lønne seg å bruke "plan to throw away"-tenkning, dvs. at man lager modellen for å få idéer, men så kaster den for at man ikke skal binde seg for mye til prototypen.

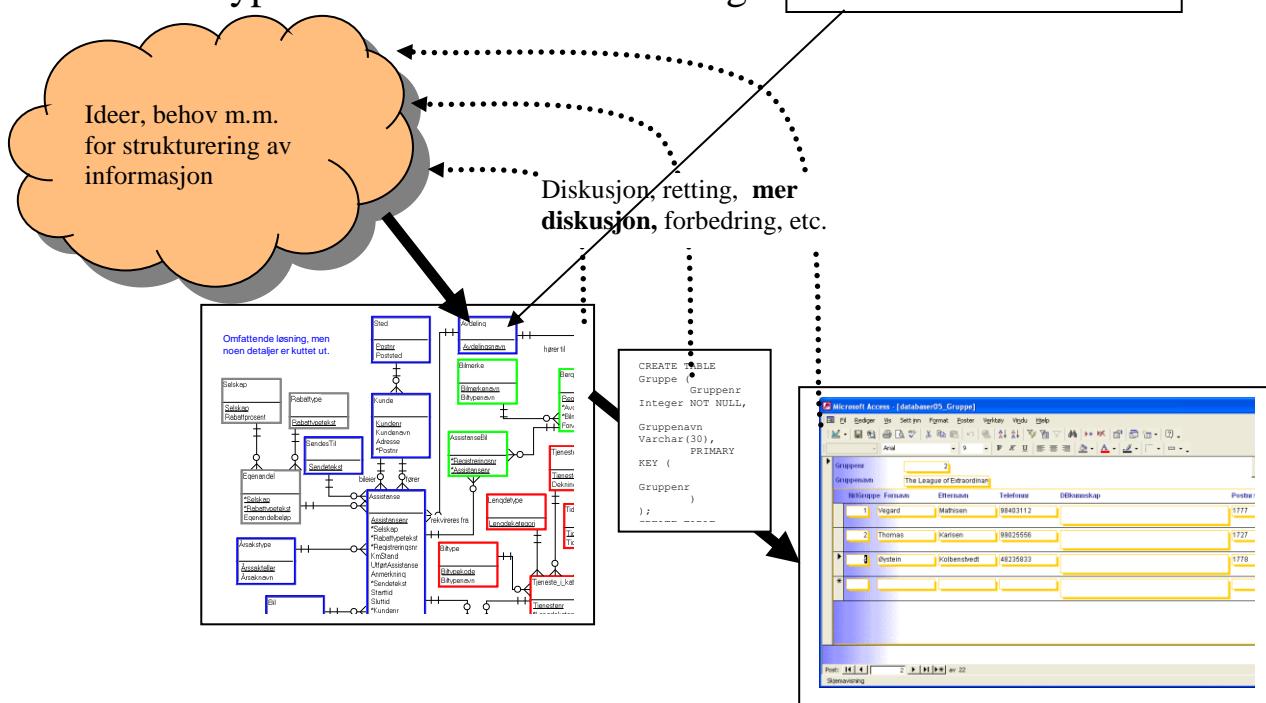
M.a.o.



(revidert fra starten av kapittelet):

Erfaring, metodekunnskaper, strukturer

## Prototypeorientert databaseutvikling



## **16.6. Vanlige strukturer**

**(kun på stikkordsnivå, mer følger i forelesningene)**

### **Ikke alt kan beskrives i en datamodell!**

- ulike notasjoner har ulike muligheter (og kompleksitet)
- behov for skriftlig beskrivelse ved siden av
  - ting som ikke lar seg beskrive i den grafiske modellen
  - designbeslutninger som er tatt og vurdering av alternativer

#### **1) Datamodeller og nøkler**

- bør primær- og evt. også fremmednøkler være med (bl.a. i forhold til detaljeringsgrad, konseptuell-logisk-fysisk osv). NB! PK & FK er en del av relasjonsmodellen, ikke modellering generelt.
- regler hvis fremmednøkler skal være med

#### **2) Hode/linje (hoved-del) -strukturer**

og graden av binding mellom disse (UML: komposisjon evt. aggregering).

Identifiserende og ikke-identifiserende

#### **2) Hode/linje (hoved-del) -strukturer**

#### **3) Vare/ordre-strukturer, og hvordan de brukes i andre systemer.**

#### **4) Mange-til-mange og eventuell entitetisering**

- mange-til-mange (beholdes eller entitetiseres)
- mange-til-mange og attributter (attributter på relasjonene eller ikke)
- selve teknikken
- 2. ordens entitetisering (viser seg at en allerede entitetisert inneholder en mange-til-mange, slik at man må splitte på nytt).

## **5) Flere relasjonstyper mellom de samme entitetstypene**

- problemstillingen
- bruk i forbindelse med historikk
- modellering i forbindelse med en hoved- og flere tilleggs-.....

## **6) Kontroll av lovlige verdier**

- i ett nivå
- i flere nivåer, inkl. "alle-til-alle" eller "mange-til-mange"
- standardverdier og unntak
- kontroll av et bestemt max. Antall

## **7) Relasjonstyper mellom 3 eller flere**

- inkl. ulike muligheter for dette.
- viktig: ulike skranner ut fra fremmednøkkelegenskaper

## **8) Flere-rolle-problematikk**

- nøyaktig to roller
- fler-roller og hvorledes legge struktur til data

## **9) Hierarki**

- fast antall nivåer (evt. med ulike data)
- variabelt antall nivåer (med mest mulig like data)

## **10) Nettverk**

- nettverk generelt
- nettverk med symmetri (ikke-rettede grafer), og problem med disse
- sammenheng mellom representasjoner: datamodeller, grafer og matriser

## **11) Vanlige feil og hvorledes løse disse**

- vifte-bommerten (= "mist ikke tråden") - engelsk: fan trap
- kløft-bommerten (="mist ikke mellomnivå") - engelsk: chasm trap

## **12) Kort om arv og muligheter i ulike modellingsdialekter**

## **13) Kort om ener-stier og ekvivalente stier**

## **16.7. Plassering av entitetstyper i modellen.**

- Det kan være lurt å plassere entitetstyper som danner en naturlig helhet nær hverandre, slik at det blir færre streker på kryss og tvers. Også lettere å lese en helhet og se en totalstruktur.
- Oftest er det nødvendig med flere runder med tegning.
- Kan lønne seg å farge de ulike delene av en modell med ulike farger, mye lettere å lese.
- Hvis det er en liten modell kan det være en fordel å sette på alle attributtene. For en større modell blir det for detaljert. NB! Hvis modellen fremdeles endres en god del, kan det lønne seg å vente med attributter til du er ”nesten ferdig”.
- Noen velger å modellere først og fremst med 1-ere øverst, mange nederst hvis mulig. Det kan gjøre modellen lettere å lese og mer entydig, men spiller ingen rolle for innholdet.

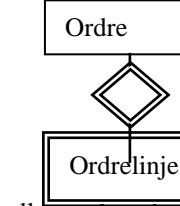
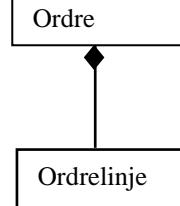
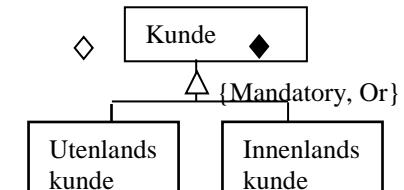
## 16.8. Notasjon og variasjoner i 3 av dialektene: Chen, kråkefot og nedskalert UML.

En del detaljer og variasjoner mangler.

(Viktig å kunne den dialekten du velger, de andre er orienteringsstoff)

	Chens ER	Kråkefot	nedskalert UML
<b>Grunnleggende.</b>	<p>For alle dialekter:</p> <ul style="list-style-type: none"> <li>attributter kan tas med eller utelates (avh. av hvor langt i prosessen og hvor stor modellen blir)</li> <li>ditto for domener/datatyper</li> <li>det finnes varianter for å vise min./max.</li> </ul> <p><b>Begrep:</b> Entitet(styper), relasjon(styper), attributter.</p> <pre> classDiagram     class Avdeling     class Person     class Personnavn     class Rollenavn      Avdeling "1" -- "m" Person : arbeidssted     Person --&gt; Personnavn     Avdeling -.-&gt; Rollenavn   </pre>	<p><b>Avdeling</b></p> <p><b>Person</b></p> <p><b>Personnavn</b></p> <p><b>Rollenavn / relasjonsnavn</b></p> <p><b>Begrep:</b> Entitet(styper), relasjon(styper), attributter.</p> <p>Verbal beskrivelse. Kan evt. settes på begge sider. Alternativt brukes en rolle som ”relasjonsnavn”</p> <p>Max. nærmest entitetstypen, evt. min. lengre unna</p> <p>Eksempel med attributter</p>	<p><b>Avdeling</b></p> <p><b>Person</b></p> <p><b>PersID</b></p> <p><b>1 er (og kan skrives) 1..1 0..1 skrives 0..1</b></p> <p><b>Ver arbeidssted for</b></p> <p><b>jobber i</b></p> <p><b>Verbal beskrivelse. På en eller begge sider. Pil viser retning</b></p> <p><b>* er (og kan skrives) 0..*</b></p> <p><b>1..* betegner 1..m.</b></p> <p><b>Begrep.</b> Entitet(styper) eller objektklasser, (multiplisitets)assosiasjoner, attributter.</p>
<b>Er repetisjoner tillatt?</b>	Ja, på konseptuelt nivå	Nei – splittes ut i egne entitetstyper	Ja, på konseptuelt nivå
<b>Eventuelle primær- og fremmednøkler</b>	Tas gjerne ikke med	Hvis det tas med: Markeres f.eks. med primærnøkkelen: <u>understrekning</u> , fremmednøkkelen: <u>prikket linje</u> , *, el.l.	Hvis det tas med: markeres gjerne med {PK} hhv. {FK} bak attributnavnet. Hvis (del av) begge deler: {PK,FK}
<b>Entitetisering</b>	<p>Kan gjøres, men vanligvis settes det bare på attributter på relasjonen.</p> <p>Bare nødvendig ved 2. ordens entitetisering.</p>	<p>Gjøres dersom ”relasjonen skal inneholde attributter”.</p> <pre> classDiagram     class Person     class Kurs     class Deltagelse      Person "*" -- "*" Kurs : Deltagelse     Person "++" --&gt; Deltagelse     Deltagelse --&gt; Kurs   </pre>	<p>Kan gjøres, men bare nødvendig ved det som ellers ville vært 2. ordens entitetisering.</p> <p>Associative entitetstyper m/ attributter kan legges på:</p> <pre> classDiagram     class Person     class Kurs     class Deltagelse      Person "*" -- "*" Kurs : Deltagelse     Person "++" --&gt; Deltagelse     Deltagelse --&gt; Kurs   </pre>
<b>n-ære relasjonstype / assosiasjoner (n &gt;2)</b>	Innebygd i notasjonen, ingen forskjell på binære og n-ære.	Evt. entitetisering gjøres først, deretter henges nye entitetstyper på denne.	Bruk Assosiativer for å knytte dem sammen. Assosiativer entitetstyper kan brukes

## Datamodellering

<p><b>Avhengighet av andre entitetstyper</b> (en entitet er avhengig av eksistensen av en annen entitet)</p>	 <p>kalles svak entitet / weak entity</p>	<p>Markeres ved at fremmednøkkelen er en del av primærnøkkelen (på mange-siden)</p>	 <p>kalles komposisjon. Finnes også en mindre sterk kobling som kalles aggregering (markeres med i stedet for ).</p>
<p><b>Arv</b></p>	<p>Finnes ikke, må i tilfelle beskrives som 1:1, men gir ikke egentlig arv.</p>	<p>Finnes ikke, må i tilfelle beskrives som 1: 1, men gir ikke egentlig arv.</p>	 <p>I tillegg: kan beskrive kombinasjoner av mandatory/optional og om en overordnet kan kobles til max. 1 eller til flere underordnede (or eller and), se over. Kan også være arv med "ett barn", f.eks. bare "Kunde" og "Utenlandskunde".</p>
<p><b>Forhold til normalisering</b></p>	<p>Må evt. gjøre utsplittinger</p>	<p>Er normalisert</p>	<p>Må gjøre evt. utsplittinger</p>
<p><b>Overføring til relasjonsdatabaser</b></p>	<p>Overføres til kråkefot el.l. først (fra konseptuelt til logisk nivå) Alternativt: Legg på primær- og fremmednøkler Evt. repetisjoner må tas bort. Entitetstyper blir til tabeller. Relasjoner som gjelder 1:m tas bort, relasjoner som gjelder m:m blir egne tabeller.</p>	<p>Evt. mange-til-mange må entitetiseres. Ellers: entitetstyper blir til tabeller</p>	<p>Evt. repetisjoner må tas bort. Entitetstyper/objektklasser blir til tabeller. Assosiasjonsattributter i m:m blir egne tabeller, andre m:m entitetiseres. Høyere ordens relasjonstyper blir til tabeller. Arv må omformuleres (flere alternativer finnes, ingen er helt gode). Dersom man bruker ORDB-utvidelser i systemer som har dette, kan arv implementeres.</p>

## 17. Normalisering - t.o.m. BCNF.

### 17.1. Hva er normalisering?

- Enkelt sagt: regler som kan brukes for å oppdage enkelte former for uheldig struktur i en database eller en datamodell.
- I praksis: splitting av en tabell el.l. i to eller flere tabeller, men slik at den opprinnelige tabellen kan gjenskapes ved å koble de splitte tabellene.

### Eksempel på en dårlig struktur:

Vi skal lage en bibliotekdatabase med mulighet for å registrere låntakere til bøker. Følgende struktur ble foreslått:

BOKLÅN

Låntakernr	Boknr	ISBN-nr	Boknavn	Lånedato
1	453	82-891-1	Ole Brumm	17.12.03
1	142	82-119-3	Le med oss	17.12.03
2	891	82-891-1	Ole Brumm	12.12.03
4	453	82-891-1	Ole Brumm	29.12.03

Vi antar at det i tillegg finnes en Låntaker-tabell.

Denne strukturen har imidlertid en rekke ulemper både med hensyn til innsetting, sletting og endring (hvilke)?

Problemet er først og fremst at siden

- ISBNnr og boktittel kun er avhengig av boknr
- Låntakernavn, Adresse, telefonnr er kun avhengig av Låntakernr, må disse gjentas mange ganger.

Vi bør derfor se på avhengigheter mellom de ulike attributter i tabellen.

## 17.2. Funksjonelle avhengigheter / determineringer.

### Funksjonell avhengighet:

Hvis det til enhver verdi av attributtet  $x$  finnes bare en verdi av  $y$ , dvs.  $y = f(x)$ , sier vi at  $y$  er funksjonelt avhengig av  $x$ .

Dette skrives som  $x \rightarrow y$ .

Det er ofte lettere å formulere det samme ved å bruke uttrykket determinering.

### Determinering (funksjonell determinering).

Hvis det til enhver verdi av attributtet  $x$  finnes bare en verdi av  $y$ , sier vi at  $x$  funksjonelt determinerer  $y$ , eller  $x \rightarrow y$ .

### Eksempler:

Leses: Boknr determinerer Boknavn,  
evt: Boknavn er funksjonelt avhengig av boknr.

Boknr $\rightarrow$ Boknavn	flere boknr kan ha samme boknavn, men ikke motsatt
Boknr $\rightarrow$ ISBNNr	flere boknr kan ha samme ISBNNr (flere eksemplarer). Derfor: ikke motsatt
Låntakernr $\rightarrow$ Låntakernavn	
Låntakernr $\rightarrow$ Telefonnr	Forutsetter da at hver låntaker bare har ett telefonnr.
(Boknr, Låntakernr) $\rightarrow$ Lånedato	Forutsetter at ikke samme bok blir lånt ut flere ganger til samme person - antagelig urealistisk...

### Determinant.

Et attributt som determinerer et annet attributt kalles en determinant.

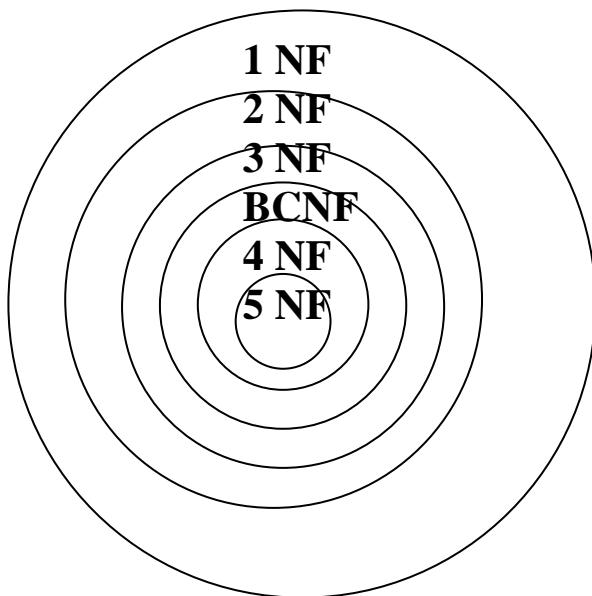
### 17.3. Normaliseringsregler – oversikt

Kort sagt er kravene til de 3 første normalformene (NF) + Boyce-Codds normalform slik:

<b>1NF</b>	<b>Alle attributter skal være atomiske</b>
<b>2NF</b>	<b>1NF +</b> <b>Deler av nøkkelen<sup>18</sup> skal ikke kunne determinere ikke-nøkkel-attributter</b>
<b>3NF</b>	<b>2NF +</b> <b>Ikke-nøkkelattributter skal ikke være transitivt (dvs. indirekte) avhengig av primærnøkkelen,</b> <b>alternativt formulert:</b> <b>Ikke-nøkkel-attributter skal ikke kunne determinere andre ikke-nøkkel-attributter.</b>
<b>BCNF</b>	<b>1NF + Enhver determinant er en kandidatnøkkel.</b>

**4. normalform og 5. normalform tas ikke opp her.**

De ulike normalformene stiller strengere og strengere krav til en relasjon, slik:



<sup>18</sup> Med nøkkel menes her kandidatnøkkel.

## 17.4. Eksempel på normalisering

### Fra unormalisert til 1NF:

Tabellen over inneholder bare atomiske attributter. Eksempel på struktur som ikke er atomisk:

Ansattnr	Ansattnavn	.....	Oppgaver
872	Ole Olesen		Karosseri, Elektisk
821	Anne Annesen		Motor, Elektrisk, Vinduer

Repeterende

Kan diskuteres om denne er atomisk eller ikke

For å gjøre den om til 1NF, må vi fjerne repetisjonen, og vi får:

Ansattnr	Fnavn	Enavn	.....	Oppgave
872	Ole	Olesen		Karosseri
872	Ole	Olesen		Elektisk
821	Anne	Annesen		Motor
821	Anne	Annesen		Elektrisk
821	Anne	Annesen		Vinduer

Denne har imidlertid andre ulemper, og er bare på 1NF. (Det er vel opplagt hva som er problemet !)

Hvis vi "går et skritt tilbake" med hensyn til boksystemet, kan vi tenke oss følgende:

BOKLÅN		
Boknr	ISBN-nr / navn	Låntakernr og når
453	82-891-1 / Ole Brum	1 : 17.12.03, 4 : 29.12.03
142	82-119-3 / Lek med oss	1:17.12.03
891	82-891-1 / Ole Brum	2:12.12.03

Sammensatt.

Sammensatt og  
repeterende.

Dette tilsvarer jo "gamle tiders" lånekort på biblioteket:

453
82-891-1
Ole Brum
.....
1 : 17.12.03
4 : 29.12.03

men det er viktig at vi ikke overtar opplegg fra manuelle systemer ukritisk !

## Oppsummering - 1NF:

Over så vi 3 typer brudd på atomær-prinsippet:

- Repeterende attributt
- Sammensatt attributt
- Kombinasjon av disse

For å være på 1NF, må kreves det at alle attributter er atomære.

## To kommentarer:

- For oss er det nesten en selvfølge at man ikke har repeterende attributter/attributtgrupper. Imidlertid tillot noen filorganiseringsformer (f.eks. såkalt ISAM-filer) at man hadde en repetisjon på slutten av en post. Dermed måtte 1NF-kravet brukes for å "flate ut" slike repeterende grupper.
- Kravet mht. ikke sammensatt attributt må sees ut fra sammenhengen. Eksempler:
- Det 11-sifrede fødselsnummeret kan f.eks. splittes opp, men det er sjeldent vi deler det i ulike attributter i en database.
- I noen tilfelle er det svært uheldig at etternavn og fornavn utgjør ett attributt, i andre tilfelle er det i praksis helt i orden.

## Fra 1NF til 2NF:

Vi har dermed tabellen

BOKLÅN

Låntakernr	Boknr	ISBN-nr	Boknavn	Lånedato
1	453	82-891-1	Ole Brumm	17.12.03
1	142	82-119-3	Le med oss	17.12.03
2	891	82-891-1	Ole Brumm	12.12.03
4	453	82-891-1	Ole Brumm	29.12.03

NB! Kombinasjonen  
av 3 attributter er  
primærnøkkel.

Tabellen er på 1NF. For å fjerne attributter som er avhengige av deler av nøkkelen: Splitt i 2 tabeller:

BOK

Boknr	ISBN-nr	Boknavn

UTLÅN

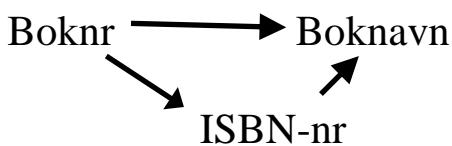
Boknr	Låntakernr	Lånedato

## Fra 2NF til 3NF.

Hva er uheldig med denne strukturen?

- Utlån kan det ikke gjøres noe med. Den eneste determineringen er (Boknr , Låntakernr) → Lånedato, som er OK.
- For BOK har vi  
Boknr → Boknavn  
Boknr → ISBNnr  
ISBNnr → Boknavn

Den siste gjør at vi får en indirekte determinering (jfr. begrepet transitivitet i matematikken, kap. 2.2).



Dermed: Splitt igjen. Vi får:

### BOK1

<u>ISBN-nr</u>	Boknavn
----------------	---------

### BOK2

<u>Boknr</u>	ISBN-nr
--------------	---------

## Fra 3NF til BCNF:

For å illustrere dette, må vi snu litt på eksempelet:

Anta at hver låntaker kan låne en bokutgivelse (dvs. med samme ISBN-nr) bare en gang. Han kunne altså ikke låne både 453 og 891, fordi disse har samme ISBN-nr. Da kan et utlån karakteriseres ved:

### UTLÅN3:

<u>ISBN-nr</u>	<u>Låntakernr</u>	Boknr
----------------	-------------------	-------



Hva er ulempen med denne?

Hvis flere låntakere låner samme bok (samme ISBN-nr), vil opplysningen om at denne har boknr xyz kunne finnes flere ganger. Hvis ikke hvert boknr er lånt ut minst en gang, kan vi imidlertid ikke vite sammenhengen mellom boknr og ISBN-nr.

Determineringer her:

ISBN-nr, Låntakernr → Boknr

Boknr → ISBN-nr

Den første er OK, mens den siste er brudd på prinsippet om at "enhver determinant er en kandidatnøkkel". Dermed: splitt!

ISBN-nr	Låntakernr	Boknr	ISBN-nr
---------	------------	-------	---------

NB! For at BCNF skal være forskjellig fra 3NF, må man ha

- en sammensatt primærnøkkel &
- en kandidatnøkkel som har minst ett attributt felles med primærnøkkelen  
==> Sjeldent i praksis.

## Bruk kun av BCNF - praksis.

BCNF inneholder i virkeligheten både 2NF, 3NF og "litt til". Siden BCNF er et enklere prinsipp, vil man ofte i praksis bare

- teste på 1NF
- teste på BCNF

### 17.5. Spiller det noen rolle hvorledes man splitter?

Ja - eksempel:

Gitt tabellen under med avdkode som primærnøkkel.

avdkode	etasjenr	avdnavn
51	3	Ost
22	1	Kjøkken
83	3	Sko

Dersom denne tabellen skal splittes, kan det gjøres på flere måter, f.eks.:

-----  
Splitt - forslag 1.

avdkode	avdnavn
51	Ost
22	Kjøkken
83	Sko

avdkode	etasjenr
51	3
22	1
83	3

### Normalisering - t.o.m. BCNF.

Ved sammenføyning (kobling) av disse på likhet i verdier (husk at rekkefølge mellom radene ikke spiller noen rolle) får:

avdkode	etasjenr	avdnavn
51	3	Ost
22	1	Kjøkken
83	3	Sko

### Splitt - forslag 2.

avdkode	etasjenr
51	3
22	1
83	3

Tilsvarende gjøres som er felles attributt, nemlig etasjenr. Dette gir:

etasjonenr	avdnavn
3	Ost
1	1 Kjøkken
3	Sko

en kobling på det

avdkode	etasjenr	avdnavn
51	3	Ost
51	3	Sko
22	1	Kjøkken
83	3	Ost
83	3	Sko

Vi fikk ekstra/kunstige tupler i forhold til den opprinnelige. Denne splitt / sammenføyningsoperasjonen var dermed ikke reversibel. Årsaken er naturligvis at etasjenr ikke er unik, slik at den kobles med rader den ikke skulle kobles med.

Konklusjon:

En splitting og kobling av tabeller kan gjøres på grunnlag av **primærnøkkelen eller annen unik og minimal nøkkel (kandidatnøkkel)**.

### 17.6. Oppsummering / sammenstilling:

#### Determinering

Fra	Til	Brudd på
Del av ID	Ikke-ID	2NF
Ikke-ID	Ikke-ID	3NF
Ikke-ID	Del av ID	BCNF
Del av ID	Annен del av ID	ID er ikke minimal

**Det eneste som tillates er altså:**

**Determinering fra hele primær- eller annen kandidatnøkkel til alle ikke-nøkkel-attributter.**

---

Som kjent har man i amerikanske rettssaker følgende ed:

**The truth, the whole truth, and nothing but the truth, so help me God.**

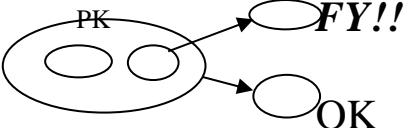
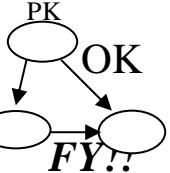
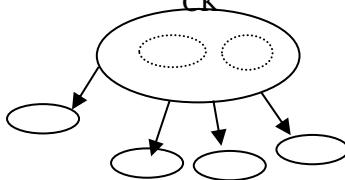
I databasesammenhenger har vi dermed, etter Edgar D. Codd (som bl.a. oppdaget Boyce-Codds normalform), fått følgende "ed":

**The key, the whole key, and nothing but the key, so help me Codd.**

Det kan være lurt å gjøre en systematisk analyse av normalformer. Denne er laget i to varianter, en hvor det bare gjøres en «grovanalyse» for å se om systemet er bra normalisert eller ikke, en med «finanalyse», der man kan se detaljert hvilken normalform som systemet er på.

Bruk dette!

## 17.7. Testskjema – normalisering

"Grov-analyse"	"Finanalyse"	Konklusjon:
	<b>Test for 1NF</b> (alle attributter må være atomiske)	Strukturen er unormalisert
	ikke-OK OK	
	<b>Test for 2 NF (gjøres bare hvis sammensatt nøkkel)</b> 	Strukturen er på 1NF
	Tillater ikke determinering fra deler av nøkkelen til ikke-nøkkel	ikke-OK OK
	<b>Test for 3NF</b> 	Strukturen er på 2NF
	Tillater ikke determinering fra ikke-nøkkel til annen ikke-nøkkel	ikke-OK OK
<b>Test for BCNF</b> 	-kun slike determineringer	Ved "grovanalyse": Vet kun at dårlig normalisert.  Ved "finanalyse": Strukt. er på 3NF.
Enhver determinant er en kandidatnøkkel (alle determineringene må gå ut fra en primær- eller annen kandidatnøkkel)		Strukturen er på BCNF

## 17.8. Litt om 4. normalform-problemer, eksempel.

Vi skal lage en oversikt over freelance-journalister, hvilke aviser de skriver for, og hvilke fagområder de dekker. Foreslått struktur:

Persid	Avisnavn	Spesialitet
1001	VG	Sport
1001	Dagbladet	Sport
1001	Aftenposten	Sport
1001	Rabarbladet	Sport
1001	VG	Politikk
1001	Dagbladet	Politikk
1001	Aftenposten	Politikk
1001	Rabarbladet	Politikk
1002	VG	Sport
1003	VG	Kultur
1003	Dagbladet	Kultur
1003	VG	Politikk
1003	Dagbladet	Politikk

Som vi ser, må hver avis kobles med hver spesialitet. Skal vi legge til en ny spesialitet for nr. 1001, må vi ha 4 nye rader, en avis gir 3 nye osv.

Ser vi på denne strukturen, kan vi konkludere:

- Den er langt fra bra, bl.a. svært mye redundans.
- Likevel: den er på 3NF og BCNF.

Problemet ligger i at det er to "uavhengige 1:m" fra Persid til hhv. Avisnavn og Spesialitet, og vi må ha med alle kombinasjoner hver gang.

Løsningen blir selvsagt å splitte tabellen i to:

Persid	Avisnavn
1001	Dagbladet
1001	VG
1001	Aftenposten
1001	Rabarbladet
1002	VG
1003	VG
1003	Dagbladet

Persid	Spesialitet
1001	Sport
1001	Politikk
1002	Sport
1003	Kultur
1003	Politikk

Begrepsdannelse og teori mht. dette utelates.

**5. normalform:** langt fra lett å forstå, og utelates her.

## 17.9. Ulike utgangspunkt for å lage en database:

- Universalrelasjon + normalisering.

Dersom vi skal lage et system, kan vi tenke oss å samle alle opplysninger vi trenger i en tabell. Denne kalles universalrelasjonen.

Deretter kan man bruke normalisering for å splitte denne opp i en passende tabelloppbygging.

- Tabeller på grunnform + gruppering.

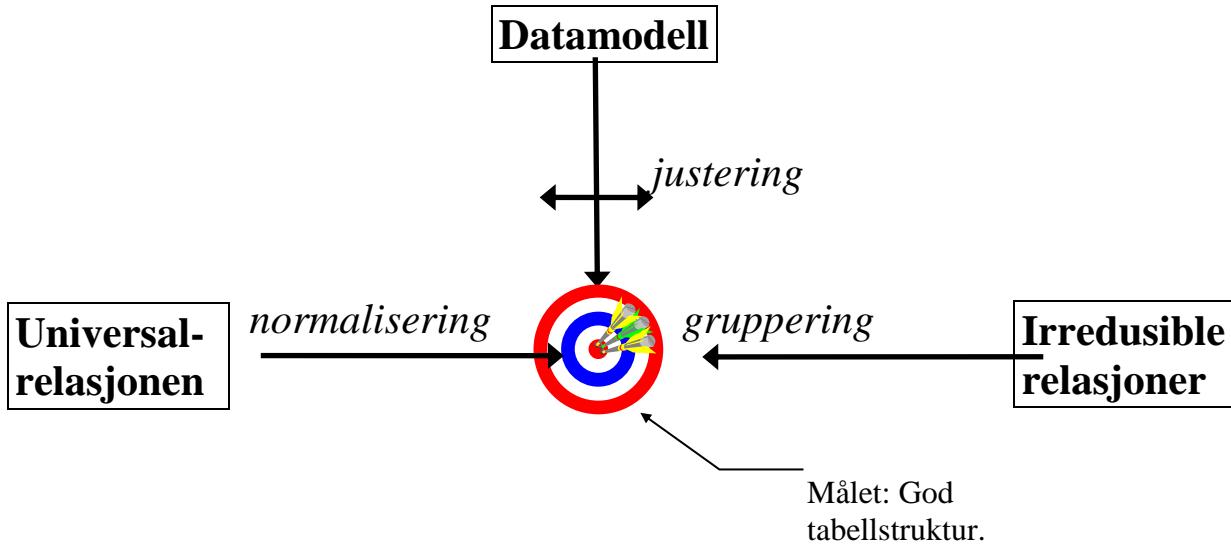
Den andre ytterligheten ville være at man splittet i flest mulig tabeller, slik at hver tabell kun inneholdt primærnøkkel + evt. ett attributt. En slik tabell sies å være irredusibel - eller å være på grunnform - den kan ikke splittes mer.

For å få en fornuftig database må man slå sammen tabellene – bl.a. slik at tabeller som har samme primærnøkkel blir slått sammen til en. Dette kalles gruppering.

- Datamodellering.

Med datamodellering forsøker vi å beskrive den logiske strukturen i informasjonen, og bruke den som utgangspunkt for en databasestruktur. Normalisering blir da retningslinjer for utforming av datamodellen og en formalisering for å kontrollere at strukturen innen hver tabell er fornuftig.

Altså:



Påstand:

Dersom man er grundig med datamodelleringen, vil man som regel ha kommet fram til en fornuftig struktur. Normalisering kan derfor brukes som en sjekk på strukturen i hver entitetstype.

### 17.10. Normalisering i praksis.

- Bruk datamodellering for å få en bra struktur - normalisering for å sjekke denne.
- Normalisering er mest en presis formulering av naturlig design.
- Kommer langt med intuisjon, men bør kunne reglene.
- Enklest å bruke BCNF, og en rask sjekk på 4NF og 5NF - hvis nødvendig.
- Av og til må man "denormalisere", f.eks. for å få en rask nok database. I tilfelle er arbeidsgangen:
  - Normaliser mest mulig.
  - Denormaliser hvis nødvendig, og dokumenter alle normaliseringssunntak.
- Legg gjerne inn triggere o.l. som sørger for konsistens på tross av manglende normalisering.
- Stabile data i database → denormalisering pga. ytelse, men gir mer plassforbruk.  
Typisk eksempel: Datavarehus.  
Mye endringer i data → normalisering gir lettere oppdatering.
- Det finnes en rekke designproblemer som ikke oppdages av normaliseringsregler.
- Strukturer er svært ofte "nesten normalisert". Typisk: normalverdier og unntaksverdier

Normalisering handler om semantikk. Automatisk normalisering er dermed umulig med mindre f.eks. alle funksjonelle avhengigheter blir beskrevet.

## 18. Ulike modeller - prinsipiell oppbygging av en database

### “PRE-RELASJONELLE”

- **Hierarkisk modell**

- Data legges i fysisk rekkefølge etter bruk
- Kun ett hieraki

Eks (Avdelingsnr, Avdelingsnavn, Ansattnr, Ansattnavn):

1 Kjøttavd → 2 Ole Jensen → 22 Anne Hansen → 11 Finn Jensen → 2 Osteavd → 3 Kjøkkenavd → 32 Jon Jonsen

- Rask, men for begrenset struktur for å beskrive virkeligheten
- Dog: ekstra pekere gjør det mye mer fleksibelt.

- **Nettverksmodell** (NB! Betyr ikke database brukt i nettverk)

- Strukturen realiseres via pekere
- Ligner multilister som brukes i programmering m/ pekere

Eks:  
0: peker ikke på noen

1	Kjøkkenavd	4
2	Osteavd	0
3	Kjøttavd	2
..	..	..

peker til 1.  
ansatt i avd.

peker i 1. med  
denne still.typen

SL	Slakter	3
EK	Ekspeditør	2
..	..	..

Loka-
1
2
3
4

Ansnr	Fornavn	Etternavn	nxt_avdnr	nxt_stillingstype
22	Anne	Hansen	3	0
32	Jon	Jonsen	0	4
11	Finn	Jenssen	0	1
21	Ole	Jenssen	1	0

0: ingen neste

Av dette kan vi f.eks. se at Kjøkkenavd. har ansatt Ole Jensen, Anne Hansen og Finn Jensen. Slaktere er Finn Jensen og Anne Hansen.

- Er fysisk effektiv på maskinen, men mye jobb å programmere.

## “RELASJONELL”

- Prerelasjonelle bygget på “fysiske” lagringsstrukturer, relasjonell på en “logisk” struktur og matematikk
- Enklest mulig struktur: kun tabeller, med rader og kolonner
- Ingen rekkefølge mellom radene (ingen “ neste”),
- Enkelt å programmere mot
- Etterhvert utvidet med litt mer struktur: domener, fremmednøkler etc.
- I databaseprodukter også utvidet med noe prosedyrell logikk lagt ned i databasen: lagrede prosedyrer og avtrekkere (triggere)

## “POST-RELASJONELLE”

### Objektorientert

- **Grunnlag:** Relasjonell er **“for enkel”**. Trenger bl.a. arv.
- Kan legge metoder (handlingsmønstre, operasjoner) sammen med tabellene.
- Data identifiseres ikke via primærnøkler, men via peker til objektet. Ditto for fremmednøkler. Men: disse vedlikeholdes av systemet.

### Objekt-relasjonell

- **Grunnlag:** Bør ikke gjøre utvidelser som bryter med relasjonsdatabaser. Mye bra i denne, bl.a. enkelhet. Må gjøre utvidelser av relasjonell i objektorientert retning.
- Sentralt: Mulighet for komplekse datastrukturer, arv og metoder.
- En del av dette er inkludert i nyere versjoner av databasesystemer.

### noSQL

- NB: Betyr **not-only SQL**. Andre strukturer eller spesialstrukturer for noen typer anvendelser, spesielt ved svært store datamengder.

### Systemer for «Big Data»

- Mest kjent: Hadoop

## STATUS – ULIKE MODELLER

- Pre-relasjonelle finnes bare i gamle systemer.
- Relasjonelle er så å si enerådende
- Objekt-relasjonelle og objekt-orienterte har ikke blitt noen stor suksess, bl.a. fordi det er blitt laget OO rammeverk rundt relasjonsdatabaser. Disse blir ”mellomvare” mellom et OO utviklingsmiljø og RDBMS.
- noSQL og BigData er i ferd med å bli store innen svært store datamengder eller spesialanvendelser.

## 19. Datauavhengighet. ANSI/SPARC-arkitekturen.

### 19.1. Datauavhengighet.

Prinsipp:

Datastrukturen skal være uavhengig av lagring av dataene.

Noen definerer det folkelig som at

databasen skal være immun mot vekst

- Endringer i lagringsstrukturen skal ikke merkes av omgivelsene (bortsett fra evt. hastighetsmessig)
- Endringer av databaseskjemaet skal ikke merkes av omgivelsene (bortsett fra evt. hastighetsmessig).

I SQL: Søkes oppnådd bl.a. ved

- Indekser.  
Disse er ikke nødvendige for å kunne manipulere i databasen og kan endres etter ønske.
- Utsnitt (view).  
Kan ofte endre tabelloppbyggingen uten at applikasjoner som bruker denne merker noe.
- Triggere.  
Denormaliseringer etc. blir maskert ved at triggere holder strukturen ved like.

## 19.2. ANSI-SPARC-arkitekturen

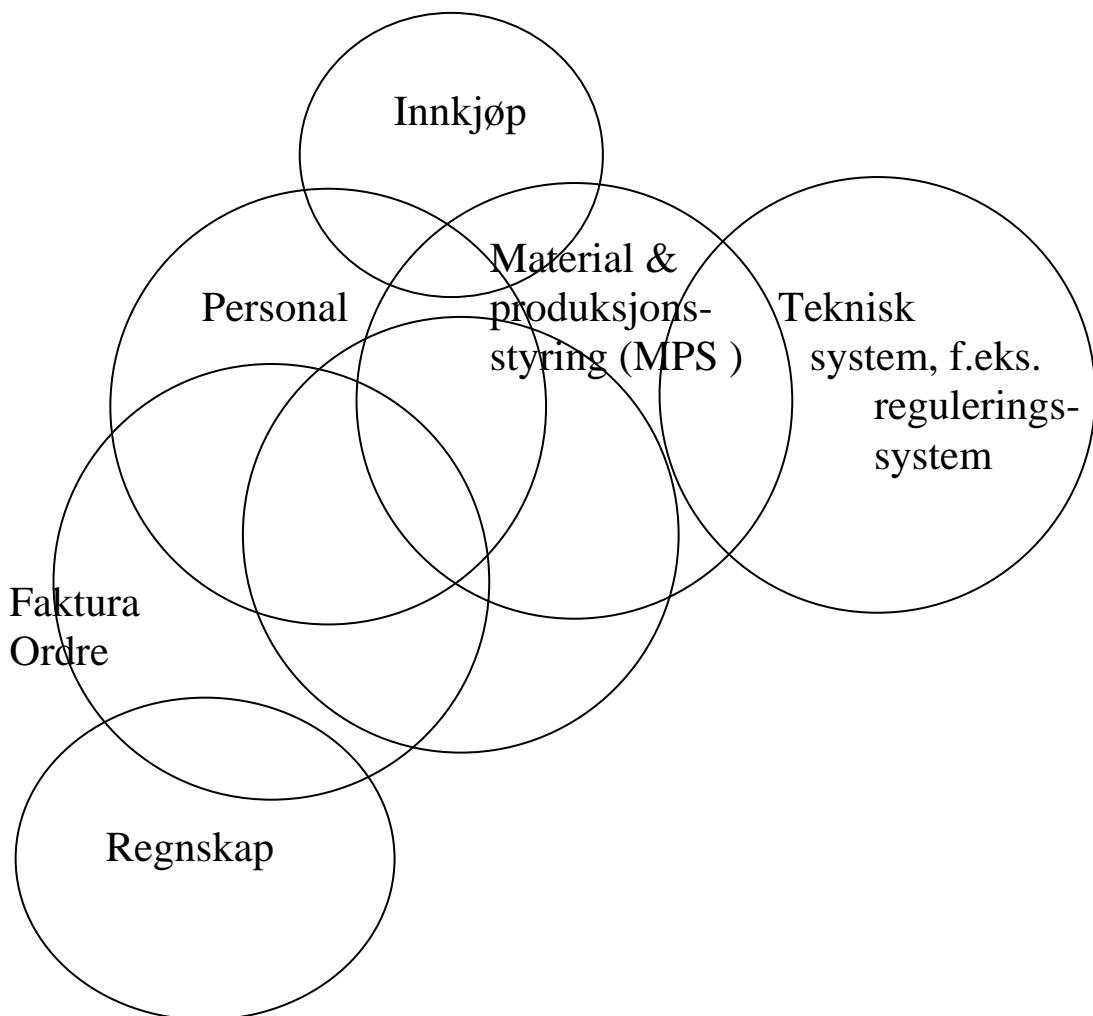
ANSI: American National Standards Institute

SPARC: Standards Planning and Requirement Committee:

---

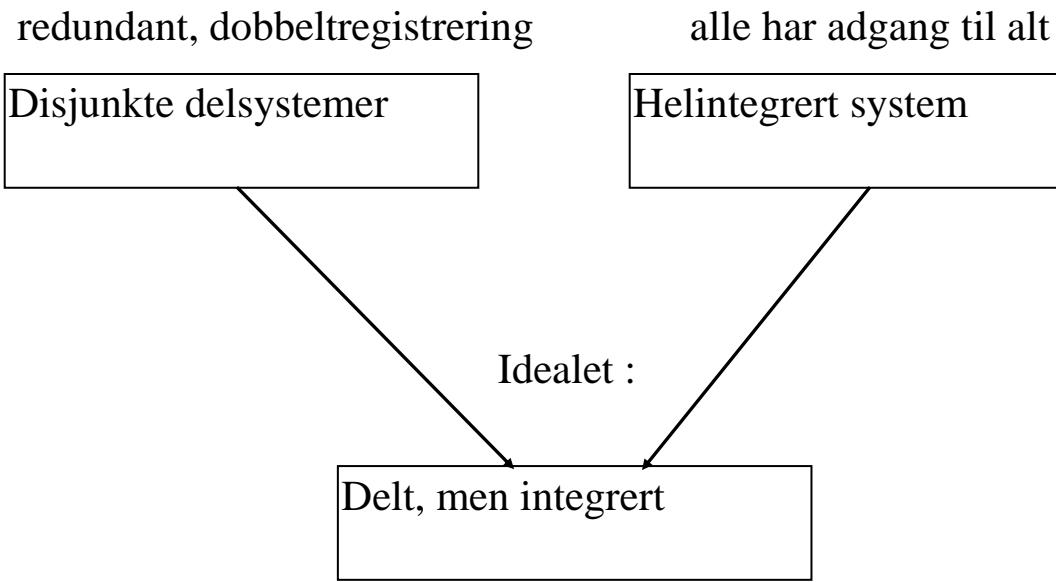
Bakgrunn:

Virkeligheten i de fleste firmaer er gjerne slik:



Ulike delsystemer, men med en god del overlapp mht. hvilke data som trengs.

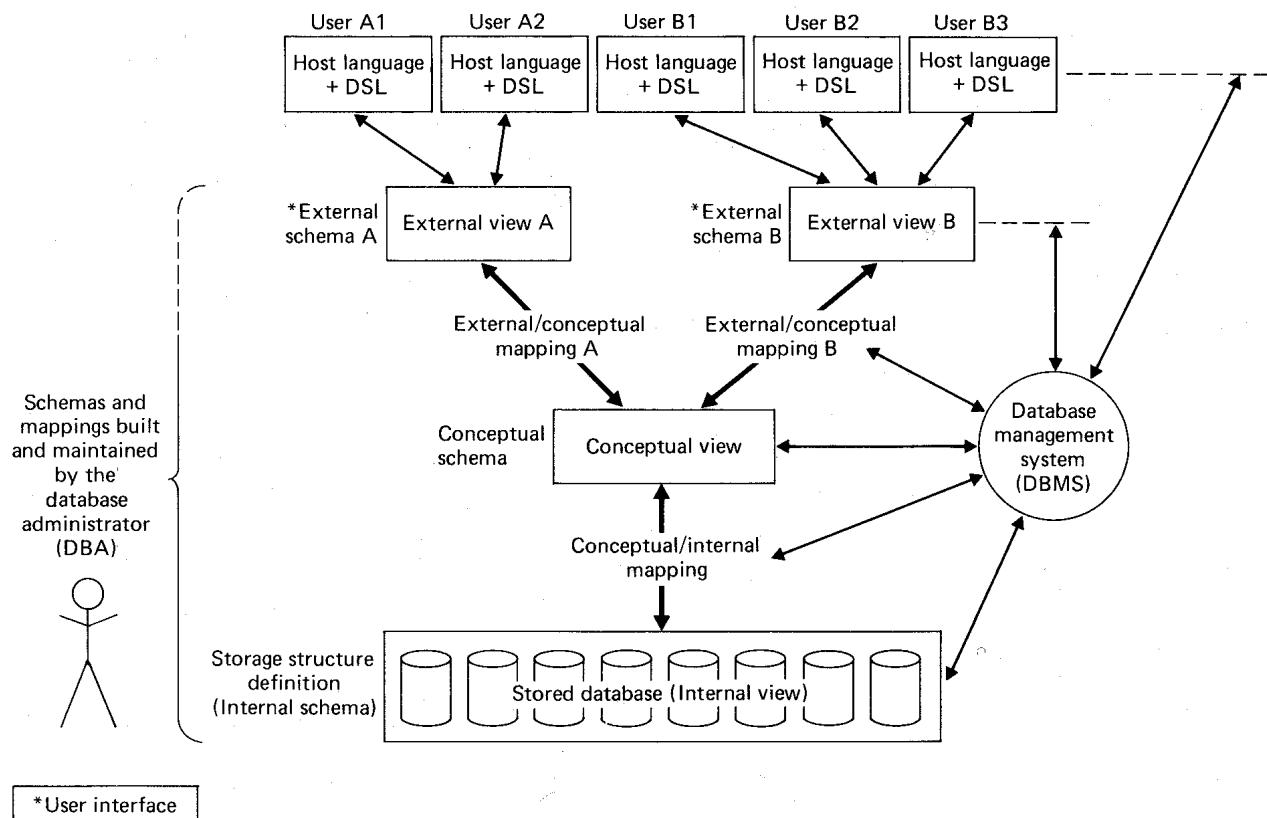
## Hvordan håndtere overlappet? Mulige strategier:



ANSI ga i 1975 ut et forslag til arkitektur for databasesystemer, hvor de skilte mellom **3 nivåer**:

Eksterne nivåer	slik applikasjonsprogram/spørrespråket ser databasen - en delmengde av totaldatabasen Kan være <u>mange</u> eksterne utsnitt.
Begrepsmessig nivå	én felles, logisk beskrivelse av dataene
Internt nivå	én felles, fysisk beskrivelse av dataene, dvs. lagringsformater, indeks etc.

## Arkitekturen - mer detaljert:



(fra Date: Introduction to Database system)

I prinsippet kan man gjøre "alle" omdefineringer av data mellom hvert av lagene, f.eks.

- nye tabeller kan legges til / tabeller kan endres, bare de involverte eksterne skjemaer berøres.
- en ekstern bruker/gruppe ser på aggregeringer (f.eks. summer) av data, mens de andre ser på ikke-aggregerte dataene
- noe oppfattes som heltall et sted, som streng et annet sted
- man lager en komplisert "oppspeedingsmekanisme" på internt nivå, mens de andre delene oppfatter systemet som like enkelt

Som vi ser: ANSI-SPARC er ideell for å få til datauavhengighet.

### 19.3. Sammenligning ANSI/SPARC v.s SQL.

Viktig å være klar over at bare deler av ANSI-SPARC er implementert i SQL.

Endel forskjeller:

ANSI-SPARC	SQL
Skille internt og begrepsmessig	Indekser (internt nivå) og logisk def. er i samme språk
Skille internt og begrepsmessig	Kluss mellom logisk struktur (f.eks. entydighet) og fysisk (f.eks. indekser)
Utsnitt er hele ”virkeligheten” slik en delappl. ser det	Utsnitt gjelder ETT utsnitt fra en eller flere tabeller.
Utsnitt er alltid oppdaterbare	Utsnitt er langt fra alltid oppdaterbare
Kan sette opp ”mappinger” mellom nivåene	Mangler slike mekanismer.
Teorimodell, ideell	Gjennomført praktisk standard, med sine svakheter

## 20. Transaksjoner & samtidighet.

### Hva?

- En transaksjon er en eller flere handlinger fra en enkelbruker inn mot (f.eks.) en database.
  - Ønske:
    - Transaksjonene bør kunne foregå mest mulig i parallel.
  - Problem:
    - En/flere brukere ønsker å oppdatere data samtidig som
    - en /flere ønsker å lese eller oppdatere de samme data
- kan risikere feil eller inkonsistenser hvis transaksjonene overlapper mht. berørte data.

Transaksjoner skal ha egenskapene **ACID**:

<b>Atomicity</b>	Transaksjonen er en enhet som enten er gjennomført fullt ut eller ikke utført i det hele tatt ("all or nothing")
<b>Consistency</b>	Databasen må være konsistent før og etter transaksjonen.
<b>Independence / Isolation</b>	Transaksjonen skal være uavhengig av andre transaksjoner. Utenforverdenen skal oppfatte transaksjonen som <u>en</u> enhet.
<b>Durability</b>	En gjennomført transaksjon skal ikke kunne mistes pga. senere feil

**Transaksjonsstart / slutt:**

**Start:**

- Noen systemer: BEGIN TRANSACTION
- Andre: ved et transaksjonsbasert uttykk, f.eks. INSERT eller UPDATE.

**Slutt:**

- COMMIT - gjennomfør (ugjenkallelig) **eller**
- ROLLBACK - tilbakefør til transaksjonsstart, alt ugjort.

## 20.1. Eksempler på samtidighetsproblemer: NB! Hvis vi ikke har transaksjoner.

1 - "Lost update" (dirty write)

Tid	T <sub>1</sub>	T <sub>2</sub>	x-verdi i basen
0			100
1		les x	
2	les x 100	x := x + 100 200	
3	x := x - 10 90	skriv x	200
4	skriv x	commit	90 (evt. 200)
5	commit		90

Skulle vært 190

2 - "Uncommitted dependency" (dirty read/read incommited)

Tid	T <sub>1</sub>	T <sub>2</sub>	x -verdi i basen
0			100
1		les x	
2		x := x + 100	
3		skriv x	200
4	les x		
5	x := x - 10	rollback	100
6	skriv x		190
7	commit		

Skulle vært 90

3 - "Inconsistent analysis" - summér x og y fra databasen (unrepeatable read)

Tid	T <sub>1</sub>	T <sub>2</sub>	sum	Verdier i basen	
0		sum := 0	0	x = 100	y = 50
1		les x			
2		sum := sum + x	100		
3	les x, y				
4	Overfør y til x: x := x + y y := 0				
4	skriv x, y			150	0
5		les y			
6	commit	sum := sum + y	100		
7		commit			

Skulle vært 150

## 20.2. Serialiserbarhet.

Prinsipp: Dersom flere transaksjoner gjøres i parallel, skal resultatet være som om de hadde blitt gjort i en eller annen rekkefølge.

Ikke problem:

- flere leser de samme datene
- flere behandler data som ikke overlapper i løpet av transaksjonen

Kan være problem:

- en eller flere leser de samme dataene som en eller flere skriver.

NB! Dette betyr at transaksjonene ikke må "blande seg i hverandre", ikke nødvendigvis at rekkefølgen er uviktig.

Eksempel: Gitt tabellen

ansattnr	lønn
1002	50000
1003	45000
9834	10

T<sub>1</sub>: Øk lønningene med 5% og

T<sub>2</sub>: Øk lønningene med 5000 kr

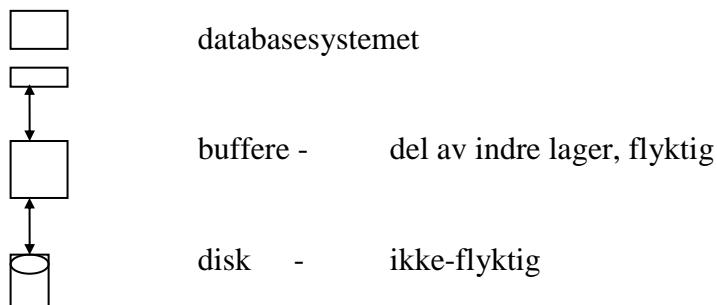
gir forskjellig resultat avhengig av rekkefølge de utføres i.

Hvis T<sub>1</sub> og T<sub>2</sub> kjørt mot databasen samtidig, vil det altså være databasesystemet / nettverkstrafikken el.l. som avgjør hva som blir resultatet. Hvis de ikke "blander seg i hverandre" er imidlertid resultatet konsistent.

## Schedule = timeplan - begrepet.

Databasesystemet må ha en mekanisme for å lage en schedule = timeplan / kjøreplan for rekkefølgen av transaksjonene.

## Buffere - et ekstra problem.



Hva hvis data er forandret, men enda ikke skrevet til disk?

## 20.3. Låsing.

Vanlig teknikk: Lås større eller mindre deler av databasen. Låsningsformer:

Delt låsing	= Leselås	= S-lock
Eksklusiv låsing	= Skrivelås	= X-lock

### Hvilke låsningsformer kan kombineres?

Andre kan læse  
raden med

	X-lås	S-lås	ikke låst
X-lås	X	X	OK
S-lås	X	OK	OK

Du har låst  
raden med

Men: dette gir fremdeles problemer i forhold til de 3 klassiske oppdateringsproblemene over.  
Løsning: se 2PL under.

### Låsing av hva ?

- hele databasen (evt. hele tiden: en-om-gangen)
- tabell
- (kolonne)
- den fysiske siden (page-locking)
- raden (record-locking)
- enkeltverdi

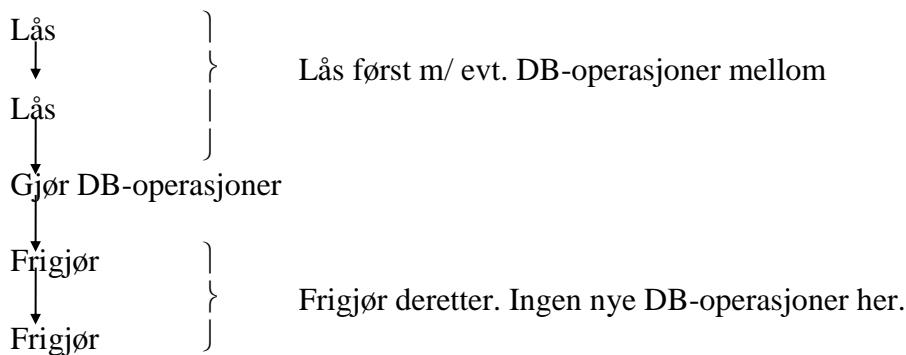
finere **granularitet**

==> bedre samtidighet, men mere tidsforbruk til låsing & frigjøring

NB! Dersom DBMS tillater ulike nivåer av låsing: Låsing på et nivå (f.eks. tabell) må låse alle forekomster av underliggende nivå (kolonne, side, rad, enkeltverdi).

## 20.4. 2-fase-låsing.

For å unngå problemene over: Bruk 2-faselåsing (2PL):



Altså: ikke bland låsing og frigjøring.

- Sjekk at dette prinsippet løser de 3 problemtypene over!  
(lost update, uncommitted dependency, incorrect analysis)

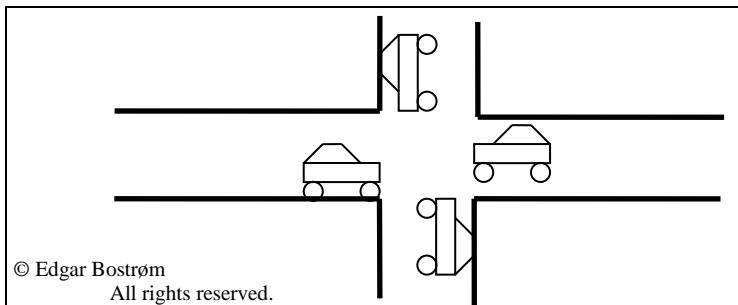
**NB!** 2PL kan føre til kaskade av tilbakerullinger:

Tid	T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>
0	Lås x, X-lås		
1	Les x		
2	x := x + 10		
3	skriv x		
4	frigjør x		
		Lås x	
		Les x	
		x := x + 100	
		skriv x	
		frigjør x	
			Lås x
	rollback		
		Må gjøre rollback	
			Må gjøre rollback

Løsning: Hold alle låser helt til transaksjonsslutt.

**NB!** 2PL kan føre til vranglåsproblemer - se neste side.

## 20.5. Vranglås (deadlock).



2PL kan gi opphav til "alle venter på hverandre", jfr. trafikkryss.

**T<sub>1</sub>:** Ønsker å låse x , deretter y.    **T<sub>2</sub>:** Ønsker å låse y, deretter x.

Tid	T <sub>1</sub>	T <sub>2</sub>
0	Lås x, X-lås	Lås y, X-lås
1	Forsøk å låse y	
2	vent	
3	.	Forsøk å låse x
4	.	.

Eksempel:

AVDELING

avdkode	avdnavn	etasjenr	areal
1	Ost	3	100
4	Hvite potteplanter	5	1000

ANSATT

ansattnr	avdkode	lønn
1002	1	30000
1003	1	40000
9834	4	10

Hvis tabellåsing:

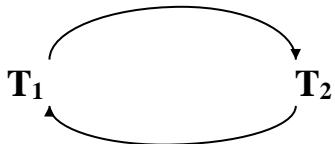
**T<sub>1</sub>:** Slett avd 4, kaskade. **T<sub>2</sub>:** Øk lønninger i Osteavd. med 10%.

Tid	T <sub>1</sub>	T <sub>2</sub>
0	Lås ANSATT	Lås AVDELING
1	Forsøk å låse AVDELING	Forsøk å låse ANSATT
2	vent	vent
3	..	..

- Sidelås kunne ført til samme problem, radlås løser problemet.
- Her: Kunne omformulert rekkefølgen for å hindre dette, men ikke alltid mulig, og normalt utenfor vår kontroll.

## Hvorledes oppdage vranglås?

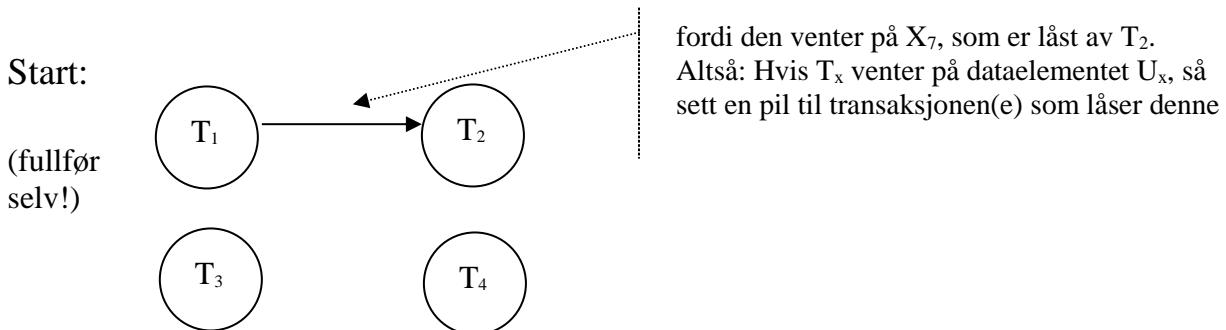
Lag en "ventegraf" engelsk: (wait-for graph). Vi har en vranglås hvis og bare hvis denne inneholder en sykel.



Litt større eksempel:

	Låser dataelementene	Venter på
T <sub>1</sub>	X <sub>5</sub>	X <sub>7</sub>
T <sub>2</sub>	X <sub>7</sub>	X <sub>2</sub>
T <sub>3</sub>	X <sub>3</sub>	X <sub>5</sub> , X <sub>7</sub>
T <sub>4</sub>	X <sub>2</sub>	X <sub>3</sub>

Sjekk om vranglås!



## Hvorledes løse opp vranglås?

- enten: **Føre var**  
sjekk på forhånd for mulig vranglås i aktuelle transaksjoner
- eller: **Etter snar**  
løs opp vranglåser som måtte forekomme.

Vranglås kan bare oppstå dersom vi har paralleisme ==> serialiser den på en annen måte, f.eks. ved at sist startede transaksjon tilbakerulles, samtidig som denne får nytt starttidspunkt. NB! Dette forutsetter en tidsstemppling på transaksjonene.

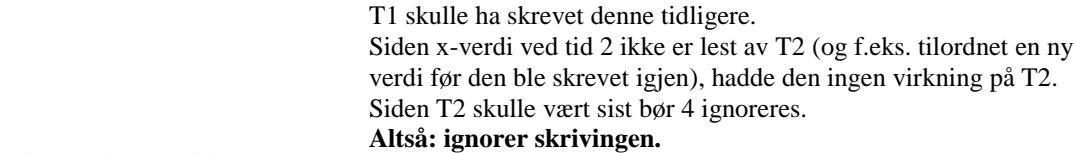
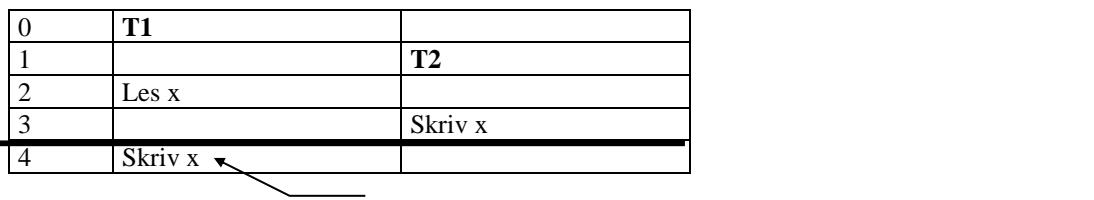
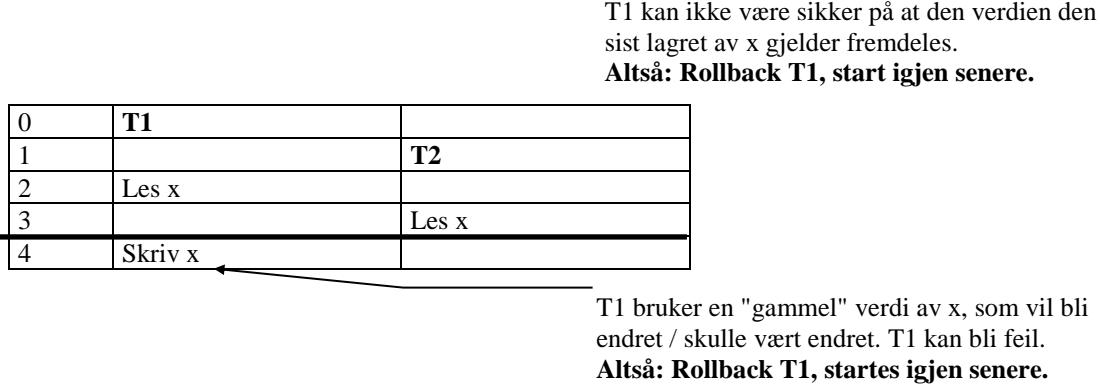
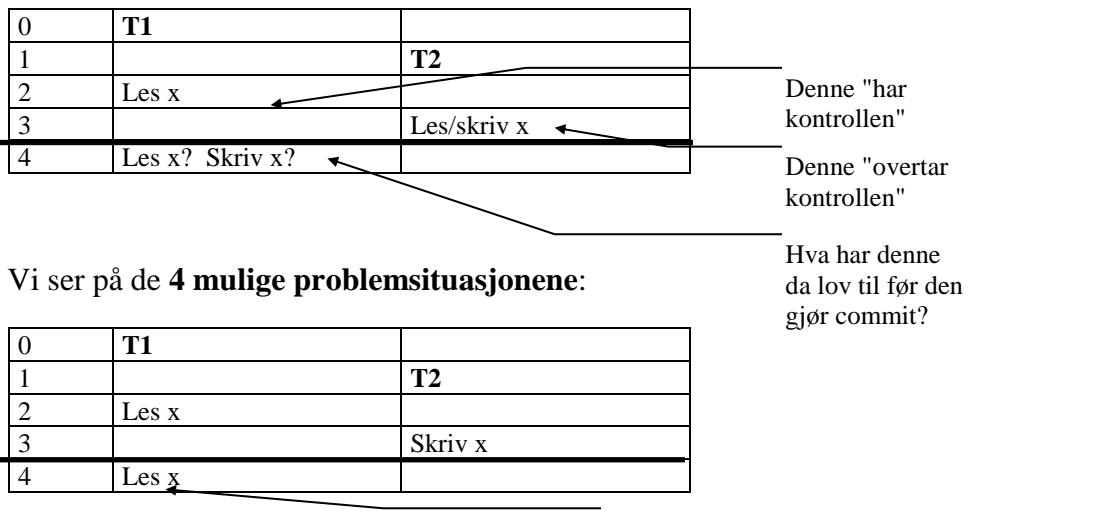
## 20.6. Tidsstemping av transaksjoner.

Alternativ til låsing:

- Alle transaksjoner får
- et **tidsstempel**, f.eks. fra maskinens klokke, evt. et transaksjonsnr.
- Alle dataelementer har
  - et **lesestempel** (start for **trans.** som sist leste dataelementet)
  - et **skrivestempel** (start for **trans.** som sist skrev dataelementet)

Protokoll - se dog problemsituasjoner under:

En transaksjon får lov til å lese eller skrive data som sist ble endret av en eldre transaksjon enn den selv.



Siste av de 4 mulighetene:

## *Transaksjoner & samtidighet.*

0	<b>T1</b>	
1		<b>T2</b>
2	Les x	
3		Les x
4	Les x	

Baserer seg på samme korrekte verdi av x. T2 kan lese og skrive x.  
Men: T1 kan ikke skrive x.

**Altså: Alt ok, fortsett.**

Større grad av samtidighet kunne nås dersom T1 også kunne skrive x først, men

- det er sjeldent aktuelt å skrive, deretter lese eller skrive på nytt.
- Det vil medføre at T1 stopper T2's transaksjon, noe som kan være problematisk og som bryter mot prinsippet om at alle transaksjoner er uavhengig av hverandre.

Transaksjonene startes i verste fall bare på nytt, ingen låsing ==> kan ikke risikere vranglås.

## **Annet alternativ - dobbeltlagring av verdier:**

Ved endring lagres både ny og gammel versjon av dataene i samme tabell. En tidsmekanisme sjekkes for å avgjøre om gammel eller ny versjon skal brukes. Tidsmekanismen oppdateres når databasen er i en stabil tilstand.

## 20.7. Pessimistiske vs. optimistiske teknikker.

**Pessimistisk låsing.** Filosofi: det går antagelig gærnt!

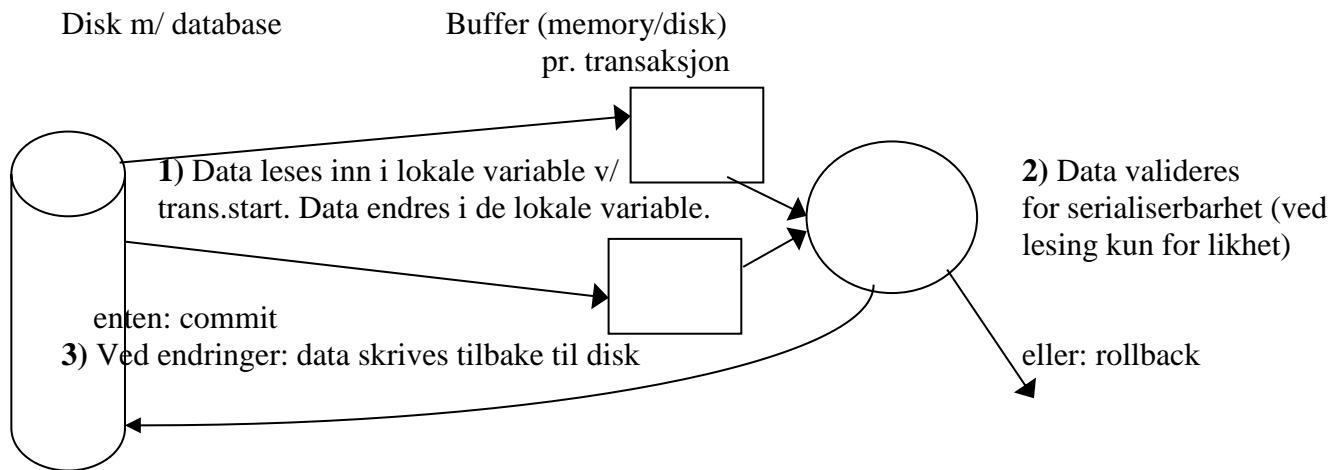
- Lås på forhånd / gjør tidsstemplinger på forhånd for å oppdage evt. konflikter.
  - Løs disse opp på forhånd, f.eks. ved en omserialisering.
- 

**Optimistisk løsning**      Filosofi: det går antagelig bra!

Kjør transaksjonen, sjekk ved commit om alt har gått bra.

Kan føre til mye tilbakerulling (også av andre transaksjoner) dersom noe har gått galt ==> brukes bare dersom liten sjanse for problemer.

## Kort om tankemåten ved optimistisk låsing:



Validering skjer ved:

Hver transaksjon får tre tidsstempel:

Start(T)              Validation(T)              Finish(T)

Evt. overlapp:

SSSSSSSSSSSSSSSS  
TTTTTTTTTTTTTTTT

En transaksjon T er godkjent hvis:

enten:

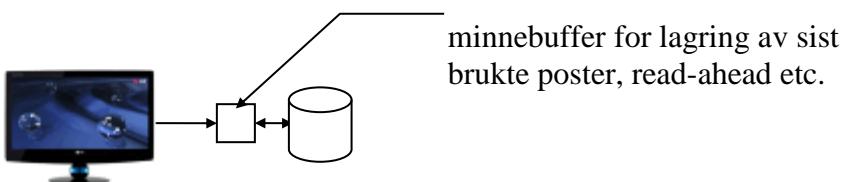
- Alle andre transaksjoner S er avsluttet før T er startet, dvs.  $\text{Start}(T) > \text{Finish}(S)$  - inget overlapp.

eller, T starter før S er avsluttet, dvs.  $\text{Start}(T) < \text{Finish}(S)$

- dataene som skrives av S er ikke de samme som de som leses av T      &
- S er ferdig med å skrive før T valideres, dvs.  
 $\text{Start}(T) < \text{Finish}(S) < \text{Validation}(T)$

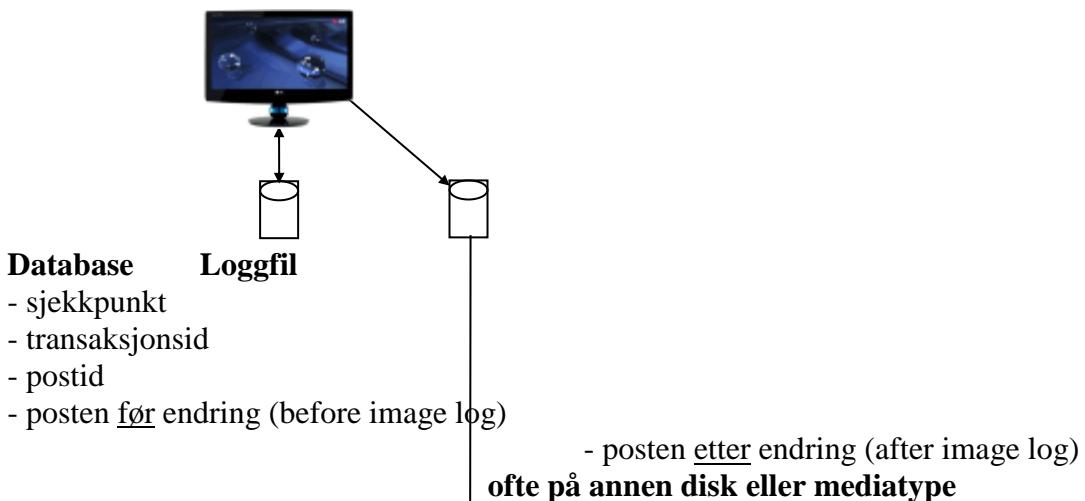
## 20.8. Gjenopprettelse av databaser.

Hvis skikkelig systemkrasj (diskkrasj, alvorlig HW- eller SW- feil, brann etc), må databasen reableres. Ekstra problem: Poster kan ligge igjen i buffer:

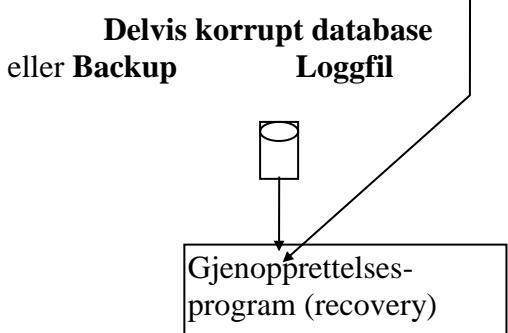


Defineres sjekkpunkter, (f.eks. med noen minutters mellomrom) - alle data fysisk skrives til disk ("flush"). Noteres i loggfilen.

### Ved normal bruk oppdatering av databasen + loggfil:

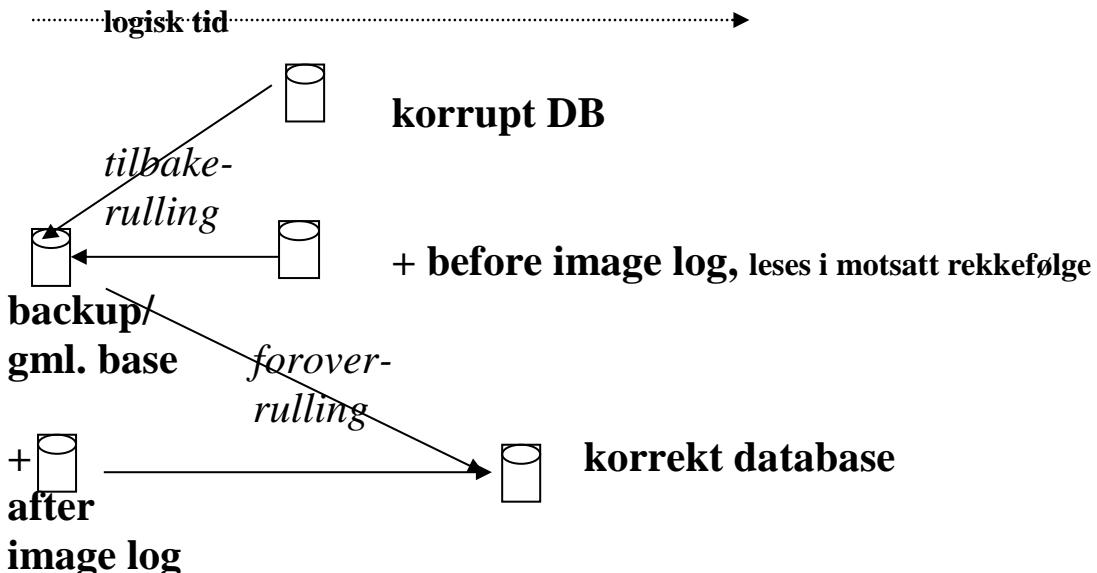


### Etter krasj:



## Gjenopprettelsen kan være

- bakoverrulling (fra delvis korrupt database og bakover)
- foroverrulling (fra backup og framover)<sup>19</sup>
- begge deler



## Alternativer mht. skriving av data:

Utsatt (deferred) skriving til hovedfil:

- skriving skjer først kun til loggfil
  - etter commit skjer skriving til hovedfil
  - hvis feil før commit: ikke skriv til hovedfil = rollback
  - hvis feil etter commit: skriv på nytt (redo)
- ==> trenger bare after image log

Imiddelbar (immediate) skriving til hovedfil:

## Alternativ loggingsmåte - "shadow paging":

- Kopier alle diskpages først
- Hvis OK, bruk de nye som kopi igjen, osv.

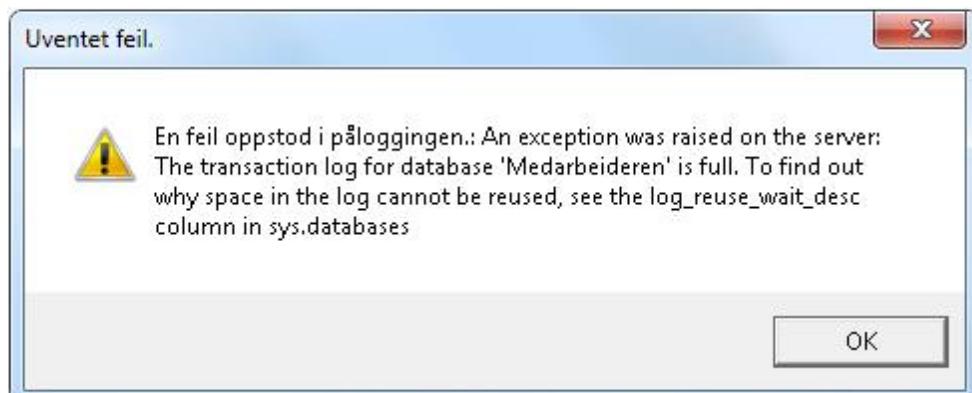
---

<sup>19</sup> Merk likheten mellom before/after-image-log og hhv. angre og gjenta f.eks. i en tekstbehandler.

## *Transaksjoner & samtidighet.*

Backup har også menneskelige sider .....

The screenshot shows a Microsoft Internet Explorer window with the title "Et tastetrykk stanset bankene - Microsoft Internet Explorer". The address bar shows the URL <http://www.dagbladet.no/dinside/2001/08/07/273342.html>. The page content is an article titled "Et tastetrykk stanset bankene" (A keyboard stroke stopped the banks). The article discusses a break-in at EDB Fellesdata where a keyboard was used to delete data from bank databases. It includes a photograph of a server room and several small thumbnail images of books. A sidebar on the right contains an advertisement for "DOMENEREGISTERET" with the text "Få hele Historien om Norge gratis!\*" (Get the whole history of Norway for free!). The left sidebar lists various news categories like "nyheter", "oppdatert meninger", "sport", "kultur", "manual", and "på din side". The bottom of the page has a footer with links to "data stata E" and "Internett".



på datamaskinen hans ved et uhell - før de hadde sikret seg eventuelle beviser.

Etter det VG får opplyst, tok politiet i Asker og Bærum beslag i tobarnsfarens datamaskin og la den i et lokale for såkalt «speiling» (kopiering, journ. ann.) i april 2010. Samtidig som denne operasjonen pågikk, ble brannalarmen i lokalene utløst og man valgte å koble fra strømmen.

#### Vet du noe? Tips oss!



Tiltaket førte til at alt innholdet på harddisken forsvant, og det var ikke mulig å gjenskape materialet etterpå - noe som betyr at politiet ikke vet om sentrale bevis gikk tapt.

## 20.9. Databaser og sårbarhet.

Databaser er kanskje den viktigste ressursen som et firma har. Forsvinner / stjeles kundedatabasen, vil det i verste fall føre til at et firma går konkurs. Vi kan snekke om mange former for sårbarhet:

- Fysisk innbrudd
- Ureglementert tilgang til maskiner og data (stjeling av passord, personer som gir passord til andre .....)
- Hardware og software-krasj.
- Hacking på ulikt vis. En kjent teknikk er såkalt SQL-injection, hvor man lager en SQL-setning som inneholder noe annet enn det som forventes, og som utfører uønskede handlinger i databasen.



(<http://xkcd.com/327/>)

## 21. Ulike temaer - databaser.

### 21.1. Utviklingstrender.

Databasebransjen har på den ene siden vært relativt stabil i mange år, med relasjonsdatabaser som nesten enerådende. Det har vært mye utvikling og eksperimentelle systemer. Noen av disse teknologiene har kommet og gått, andre har stor innflytelse på bransjen. De siste årene har det imidlertid vært en rivende utvikling, både innenfor hardware, software og modeller. Ikke minst har «big data» vært en sterk endringsdriver på avanserte systemer. Vi nevner bare noen eksempler:

Maskinemessig har vi gått fra enprosessorsystemer til flerprosessorsystemer, hvor en maskin utfører flere oppgaver samtidig, og dermed får en mangedoblet ytelse. Vi har gått fra at alt foregår på en sentral maskin til at data og prosessering skjer på ulike steder, ofte over store avstander. En naturlig videreutvikling er at man legger programvare og databaser i «skyen».

Vi har gått fra enkle klient-tjener-systemer til systemer med mange lag, f.eks. selve databasen (kanskje distribuert), OO-lag rundt databasen, databaseorienterte forretningsregler, lag med applikasjonsorienterte forretningsregler, lag med brukergrensesnitt. Og ikke minst har vi gått fra moderate mengder av data til enormt store mengder.

### 21.2. Datavarehus

Vi tar først opp en type teknologi som har vært etablert i relativt lang tid, nemlig datavarehus. Litt enkelt sagt er det ut fra et behov om å lagre mye historiske data, for dermed å kunne bruke de til analyse og prediksjon.

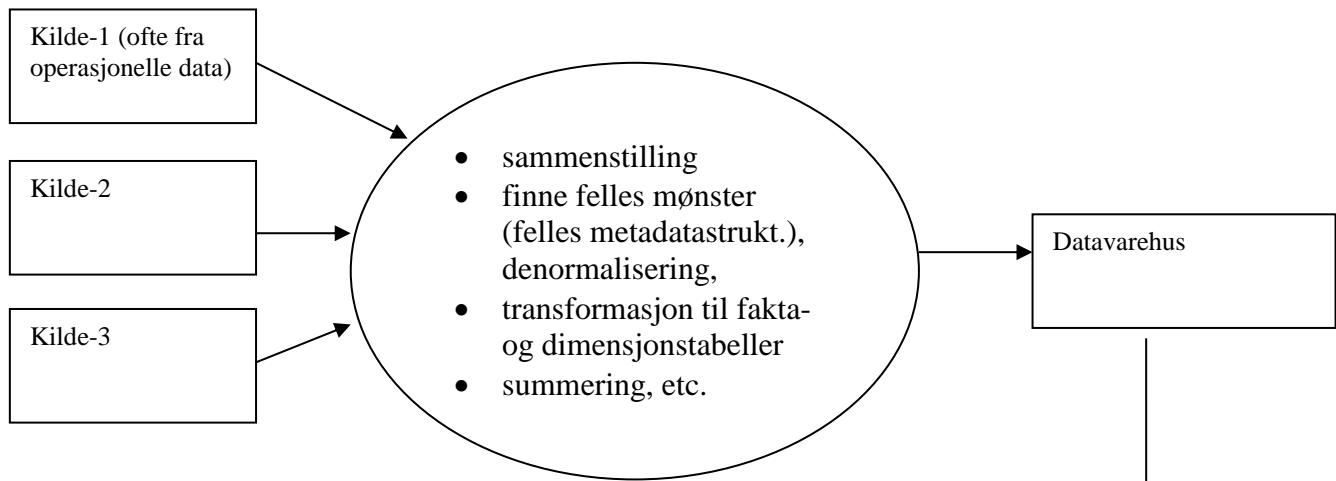
Noen stikkord:

- viktig skille mellom bruk av databaser: OLTP (On Line Transactional Processing, vanlige, transaksjonsbaserte systemer) og OLAP On Line Analytical Processing, systemer for analyse av data)
- ingen registrering direkte i datavarehuset, data hentes fra andre databaser, ofte fra mange kilder, med strukturomforming.
- "multidimensjonale databaser"
- ikke-normaliserte strukturer
- optimalisert på henting/analyse av summerte (aggregerte) data
- datadrilling (drilldown og drillup).
- ofte svært store datamengder
- "minivariant" kalles datamarked (data mart)

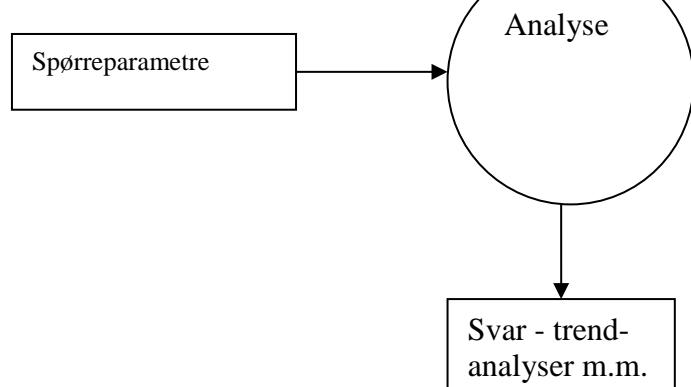
Fra et bedriftsperspektiv snakker vi ofte om Business Intelligence, altså det å kunne bruke dataene til å forutse trender, bruke de til å finne mønster m.m.

## Svært forenklet kan man skissere:

Laging av et datavarehus:



Bruk av et datavarehus



### 21.3. XML og JSON

Vi skal først se på XML, Xtendable Markup Language, som er en standard som både er et språk i seg selv og et metaspåk, dvs. slik at vi kan utvkle nye standarder på grunnlag av XML.

Bruksområdene er mange, f.eks.

- definisjon av initialverdier for programmer
- definisjon av standarder f.eks. for et yrkesområde eller en vitenskap (MathXML, ChemXML, .....)
- lagringsformat for data for vanlige standardprogrammer
- definisjon av dokumenter, f.eks. standarddokumenter
- utvekslingsformat for data

Det er spesielt det siste som er interessant i vår sammenheng. De fleste databasesystemer kan i dag hente data inn på XML-format og/eller skrive data på XML-format.

XML er definert hierarkisk, og har derfor begrensninger i forhold til den strukturen som er mulig i en relasjonsdatabase. Bruksområdet er typisk mindre datamengder og ikke nødvendigvis helt fast strukturerte data (semistrukturerte data).

På den annen side kan relasjonsdatabaser fungere som fast lagring for data som plukkes ut og legges i en XML-struktur. Relasjonsstrukturen kan dermed brukes for en generell struktur for dataene, mens de enkelte XML-utplukkene er hierarkiske. Det finnes standardiserte mekanismer for å lage slike stukturer, bl.a. såkalte DOM-trær, og det finnes egne spørrespråk, Xquery, inn mot denne stukturen.

Det finnes en rekke verktøy som støtter ulike deler av XML-standarden.

Mer informasjon finnes bl.a. på [www.xml.org](http://www.xml.org).

JSON (JavaScript Object Notation) er en forenklet standard, som også er tag-basert, og som har blitt populær i mange (database)systemer. Vi henter et eksempel fra nettet på en datastruktur i JSON<sup>20</sup>:

```
{  
    "title": "Example Schema",  
    "type": "object",  
    "properties": {  
        "firstName": {  
            "type": "string"  
        },  
        "lastName": {  
            "type": "string"  
        },  
        "age": {  
            "description": "Age in years",  
            "type": "integer",  
            "minimum": 0  
        }  
    },  
    "required": ["firstName", "lastName"]  
}
```

### Sample JSON Schema

Se ellers [www.json.org](http://www.json.org).

<sup>20</sup>[https://www.google.no/search?q=&tbm=isch&tbs=rimg:CaYUDuzZ27cXIjzm4ap4x8DNI2AKp5RV1\\_1E6xEiFwhsmOiIebPYLVbm5WTz8no137Z-QWFJyAYEc\\_1I6mOrXgxqfioSCbObhqnjHwM0EbP7XCCidrjMKhIJjYAqnIFXX8QRa91tmUK82gAqEgnrESIXCGyY6BHfEMiY0Q-ELyoSCYh5s9gtVublEY2cXUNNeQmUKhIJZPPyejXftn4RyXUvh7GkNrQqEglBYUnIBgRyfxGQRcjETN6aiioSCcjY6teDGp-EZx13ATg\\_1dmU&tbo=u&sa=X&ved=0ahUKEwju2cTNisTYAhXEDZoKHfYbCO8Q9C8IHw&biw=1280&bih=590&dpr=1.5#imgrc=bzelDEA2PBO3HM](https://www.google.no/search?q=&tbm=isch&tbs=rimg:CaYUDuzZ27cXIjzm4ap4x8DNI2AKp5RV1_1E6xEiFwhsmOiIebPYLVbm5WTz8no137Z-QWFJyAYEc_1I6mOrXgxqfioSCbObhqnjHwM0EbP7XCCidrjMKhIJjYAqnIFXX8QRa91tmUK82gAqEgnrESIXCGyY6BHfEMiY0Q-ELyoSCYh5s9gtVublEY2cXUNNeQmUKhIJZPPyejXftn4RyXUvh7GkNrQqEglBYUnIBgRyfxGQRcjETN6aiioSCcjY6teDGp-EZx13ATg_1dmU&tbo=u&sa=X&ved=0ahUKEwju2cTNisTYAhXEDZoKHfYbCO8Q9C8IHw&biw=1280&bih=590&dpr=1.5#imgrc=bzelDEA2PBO3HM)

## 21.4. OO og OR-databasesystemer.

En naturlig følge av utviklingen innenfor objektorientert programmering, var at man utviklet objekt-orienterte databasesystemer. Den var også drevet av andre faktorer, som

- Behov for å lagre mer kompliserte strukturer (f.eks. bilder, kart, filmer) og tilhørende behandling av data.
- Behov for å representere nesten like, men ikke helt like data (arv), knytte handlinger og data tettere sammen etc.

Vi kan skille mellom to typer systemer:

Rent objektorienterte databasesystemer (OODB):

- Tenkningen med primærnøkler, fremmednøkler etc. forkastes.
- Alt er objekter med pekere mellom. Bruker ofte mer kompliserte lagringsstrukturer ("bags"/multimengder, lister, arrays etc.).
- Nærmere implementering, blir ofte raskere, men mer kompliserte.  
Se f.eks. <http://www.objectivity.com>, db4o, nå Versant,  
<http://supportservices.actian.com/versant/default.html>

Kombinasjon av relasjonell og objekt-orientert teknologi (ORDB).

- Beholder relasjonssturen, men utvider med arv, metoder osv.
- Se bl.a. Oracle, PostgreSQL.

## 21.5. noSQL-systemer

NB! Dette betyr not-only SQL. noSQL er en samlebetegnelse på ikke-SQL-baserte systemer, som på en eller annen måte kompletterer vanlige relasjonsdatabaser. Vi kan snakke om mange typer av noSQL-systemer<sup>21</sup> som

- objektorienterte,
- nettverksbaserte,
- grafbaserte,
- (nøkkel,verdi)-baserte,
- dokument-baserte,
- distribusjonsbaserte,
- kombinasjoner.

Ofte er slike spesialstrukturer nødvendige for å kunne lagre og behandle massivt med data, altså Big Data. Det finnes en rekke systemer innenfor hver av disse kategoriene. Det mest kjente er antagelig Hadoop, som jobber med massiv distribusjon av data til mange, opp til flere tusen maskiner for rask prosessering av data. I tillegg brukes ofte med en algoritme for komprimering av data, kalt MapReduce.

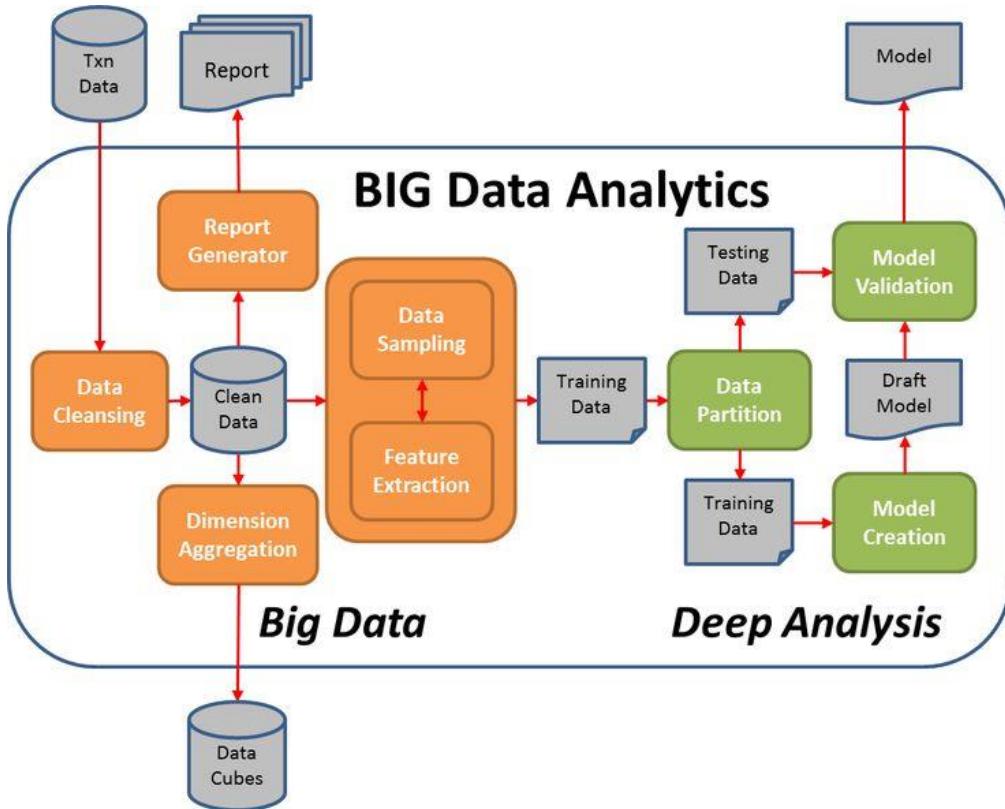
<sup>21</sup> se f.eks. <http://nosql-database.org/>.

## 21.6. Big Data

Big Data er en samlebetegnelse på ulike teknikker hvor man har massivt med data som skal behandles på kort tid. Det kan være snakk om data fra analyser f.eks. av realtime data fra Nordsjøen, samling av data for analyse av strømforbruk, samling og analyse av data fra vår egen bruk av sosiale medier. En rekke ulike systemer, bl.a. noSQL-systemer er i bruk. Noen eksempler på systemer finner vi i oversikten under, (se ComputerWorld Norge, 28.11.2016):



Ofte integreres dette med visualisering av trender, mønstre, osv., i mange tilfeller slik at man ser utviklingen i realtime. Big Data kombineres gjerne med maskinlæring (ofte dyp-læring), altså at maskinen leter etter og lærer mønstre i store datamengder, og ut fra det kan trekke konklusjoner, f.eks. foreslå produkter du antagelig kan være interessert i, utvikle «superintelligent» oppførsel osv.



En figur som kombinerer datavarehusenkning med «Big Data» finnes her<sup>22</sup>.

Noen mener at dette er et av de viktigste framskrittene for menneskeheden, mens andre er skremt av utviklingen, og redd for at dette blir ødeleggende for mennesker i framtiden.

<sup>22</sup><https://www.google.no/search?q=big+data+development+model&tbo=isch&tbo=u&source=univ&sa=X&ved=0ahUKEwitxOTfpMTYAhUNS5oKHbMrAXgQsAQI0g&biw=1152&bih=514#imgrc=McKL0Wr3CWDYhM>:

## 22. Noen bøker innen databaser

**Edgar Bostrøm:** *Datamodellering – praksis og teori.*

MetodeData a.s. 1999. ISBN: 82-91915-21-0. 140 s.

Tar kun opp datamodellering, og er dermed et komplement til disse transparentene.

**Thomas Connolly, Carolyn Begg:**

*Database Systems. A practical Approach to Design, Implementation and Management.*

6. utgave. Pearson, 2016 1374 s. ISBN: 1-292-06118-9.

**Chris J. Date:** *An introduction to Database Systems,*

8.utg. Addison-Wesley 2000. 983 s. ISBN: 0-321-18956-6

**Barry Eaglestone, Mick Ridley:** *Object databases: An introduction.*

McGraw Hill, 1998. 380 s. ISBN: 0-07-709354

**Ramez Elmasri, Shamkant B. Navathe:** *Fundamentals of database systems*

Addison Wesley, 6. utgave 2011. 1155 s. ISBN: 978-0-13-214498-8, 0-13-214498-0

**Kjell Toft Hansen, Tore Mallaug:** *Databaser.*

Gyldendal, 2008, 2. utg. 380 s. ISBN/EAN: 9788205381056

**Bjørn Kristoffersen:** *Databasesystemer.* Universitetsforlaget 2016, 4. utg. 423 s. ISBN 978-82-15-02708-1

**Rich van der Lans:** *Introduction to SQL.* Addison-Wesley 2000. 700 s. ISBN: 0-201-59618-0

**Gerald V. Post:** *Database Management Systems.* 2. utg. 2002. 600 s. ISBN 0-07-112113-7

**Gunter Saake, Andreas Heuer, Kai-Uwe Sattler:** *Datenbanken: Implementierungstechniken.*

2. utg. 2005. 870 s. ISBN: 3-8266-1438-0

**Gunter Saake, Andreas Heuer:** *Datenbanken: Konzepte und Sprachen.*

2. utg. 2000. 700 s. ISBN: 3-8266-0619-1

**Richard T. Watson:** *Data Management. Databases and Organizations.*

Wiley, 4.utg. 2004. 603 s. ISBN: 0-471-34711-6

**TILLEGG:** Om og og eller, eller eller og og, og om alle, noen og ingen, om sannhet og usannhet, om tilhørighet, om allting og ingenting (!).

## 23. TILLEGG: Om og og eller, eller eller og og, og om alle, noen og ingen, om sannhet og usannhet, om tilhørighet, om allting og ingenting (!).

Den vanligste formen for databaser i dag, relasjonsdatabaser, bygger fullt ut på diskret matematikk. Det er derfor lurt å vise grunnlaget og sammenhengen mellom databaser og diskret matematikk

- Det viktigste i denne sammenhengen er å forstå begrepene og den grunnleggende tenkningen bak.
- Dessuten kan noen regler, f.eks. deMorgans lover, brukes til å forenkle f.eks. spørninger.
- For å forstå videregående databaseteori kan også en del annen matematikk være nyttig.

### Om disse sidene

- Oppsettet under er i hovedsak stikkord, med selve begrepene og notasjon. Eksempler, anvendelser etc. må komme i tillegg for å forstå det.
- Noe fra SQL er tatt med for i vise sammenhengen mellom logikk, mengdelære og SQL.
- Deler er markert med *kursiv og skygge*. Det er ting som er mindre viktige i vår sammenheng.

### Grunnleggende lover

- Kommutativitet, for tall gjelder f.eks.  $x + y = y + x$ ,  $x \bullet y = y \bullet x$ , men ikke  $x - y = y - x$ .
- Assosiativitet, ” f.eks.  $x + (y+z) = (x+y) + z$
- Distributivitet, ” f.eks.  $x(y+z) = xy + xz$
- Refleksivitet ” f.eks.  $x = x$
- Anti-refleksivitet Eks.:  $x < x$  stemmer aldri. Både  $<$  og  $\neq$ -operatorene er anti-refleksive.

x, y etc. kalles operander.  
+, -, <,  $\neq$  etc. kalles operatorer.

Matriser				
12	Jenco	Aveien	3	43
26	Asco	Bveien	4	21
15	Kelko	Cveien	8	3

kolonner

rader

- For tallmatriser kan man under visse forutsetninger definere sum (+) og produkt ( $\bullet$ )
- For slike er + kommutativ, mens  $\bullet$  ikke er kommutativ (det er tom. slik at selv om  $x \bullet y$  har mening, behøver ikke  $y \bullet x$  å ha mening)<sup>23</sup>.

<sup>23</sup> For detaljer: se lærebøker som omhandler matriser. Poenget her er at egenskaper som kommutativitet ikke er en selvfølge.

**TILLEGG:** Om og og eller, eller eller og og, og om alle, noen og ingen, om sannhet og usannhet, om tilhørighet, om allting og ingenting (!).

## 23.1. Logikk, lukkede og åpne utsagn



### Utsagn

Vi bruker p, q, r osv. om logiske utsagn (ytringer, påstander), dvs utsagn som kan være sanne eller usanne.

Beskrivelse	(Matematiske) notasjon	Notasjon i SQL	
Sant	s, sann, true, t, <sup>24</sup> ✓	TRUE	
Usant	u, usann, false, f, ✗	FALSE	Brukes sjeldent direkte, fordi et utsagn i seg selv er sant eller usant
Og	$p \wedge q$	AND	AND
Eller	$p \vee q$	OR	OR
eksklusiv eller	$p \vee\! q$ (evt. $p \oplus q$ )	XOR	XOR
Ikke	$\neg p$ (evt. $\sim p$ eller $\bar{p}$ )	NOT	NOT

Flere av disse har presise betydninger i matematikk som ikke nødvendigvis stemmer med dagligspråket. Hva betyr f.eks. dette skiltet?



### Sannhetsverditabeller, bl.a

S betyr at et utsagn er sant, U betyr at et utsagn er usant.

$p$		$\text{NOT } p$
S		U
U		S

$p$	$q$	$p \text{ AND } q$	$p \text{ OR } q$	$p \text{ XOR } q$	...	$p \Rightarrow q$	$p \Leftrightarrow q$
S	S	S	S	U		S	S
S	U	U	S	S		U	U
U	S	U	S	S		S	U
U	U	U	U	U		S	S

### Prioritet av operatorer ("hva binder sterkest sammen")

logiske operatorer  
4) parenteser  
3)  $\neg /$  NOT  
2)  $\wedge /$  AND  
1)  $\vee, \vee\!$  OR/XOR

parallel til aritmetikk:  
parenteser  
- som fortegn  
 $\bullet, /$   
 $+, -$

<sup>24</sup> Noen ganger brukes også binære tall, f.eks. 1 = sann, 0 = usann.

**TILLEGG:** Om og og eller, eller eller og og, og om alle, noen og ingen, om sannhet og usannhet, om tilhørighet, om allting og ingenting (!).

## Grunnleggende lover

- kommutativ lov       $p \text{ AND } q \equiv q \text{ AND } p$ , tilsvarende for OR.
- assosiativ lov  $p \text{ AND } (q \text{ AND } r) \equiv (p \text{ AND } q) \text{ AND } r$ , tilsvarende for OR
- distributiv lov  $p \text{ AND } (q \text{ OR } r) \equiv p \text{ AND } q \text{ OR } p \text{ AND } r$
- noen andre lover:     $p \text{ AND NOT } p = U$ ,  $p \text{ OR NOT } p = S$ ,  $p \text{ AND } U = U$ ,  $p \text{ OR } U = p$
- deMorgans lover:     $\text{NOT}(p \text{ AND } q) \equiv \text{NOT } p \text{ OR NOT } q$ ,  
 $\text{NOT}(p \text{ OR } q) \equiv \text{NOT } p \text{ AND NOT } q$

## Åpne utsagn

Vi bruker  $P(x)$ ,  $Q(x)$ , osv. om åpne logiske utsagn, dvs. der  $P(x)$  er sann/usann avhengig av hva  $x$  er.

Eksempler:

- $P(x) = \text{«Person } x \text{ er gift»}$ . Dette er sant eller usant avhengig av hvem personen  $x$  er.
- $Q(x) = \text{«Det regner i Bergen på dag } x\text{»}$ . Hvis  $x$  er en tilfeldig dag, er  $Q(x)$  svært ofte sant (!!)
- $R(x) = \text{«Byen } x \text{ har flere enn 50000 innbyggere pr. 1.1. i år»}$ . Sant for noen byer, usant for andre.

Vi kan også ha åpne utsagn i flere variable, f.eks.

- $P(x,y) = \text{«Personen } x \text{ bor i kommune } y\text{»}$

Dette er det samme som det vi bruker i betingelser, f.eks. where-setninger i SQL.

### 23.2. Åpne utsagn og kvantorer

En kuantor er en operator som kan brukes på et åpent utsagn. Hvis  $P(x)$  er et åpent utsagn, kan vi ha behov for å si at

- det er alltid slik at  $P(x)$  er sann, dvs. at for alle  $x$  så er  $P(x)$  sann. Vi skriver det slik:  
 $\forall x: P(x)$ . (*Tips: A - alltid - speilet, eller om vi vil, skrevet opp/ned*).
- det eksisterer minst en  $x$  slik at  $P(x)$ . Vi skriver det slik:  $\exists x: P(x)$ . (*Tips: E - speilet*)
- det finnes ingen  $x$  slik at  $P(x)$  er sann, dvs.  $\neg \exists x: P(x)$  eller  $\forall x: \neg P(x)$ .

**Eksempel 1:** Vi har ulike blomster  $x$ .  $P(x)$  er usagnet "Blomst  $x$  er rød".

Vi kan da tenke oss ulike muligheter:

- $\forall x: P(x)$  - betyr at alle blomstene er røde. **For alle** blomster  $x$  er det sant at  $x$  er rød.
- $\exists x: P(x)$  - betyr at minst en blomst er rød. Det eksisterer **minst en** blomst  $x$  som er rød.
- $\neg \exists x: P(x)$  eller  $\forall x: \neg P(x)$  - betyr at det **ikke eksisterer** noen blomster som er røde. Mer folkelig sagt: ingen av blomstene er røde-
- $P(x)$  - betyr at vi ikke vet, antagelig er noen  $P(x)$  sanne, andre ikke.

**Eksempel 2:** Vennegjengen har vært ute og fisket.  $P(x)$  er utsagnet "Person  $x$  har fått fisk (som ble tatt over bord)". Vi kan uttrykke:

Alle har fått fisk:  $\forall x: P(x)$

Det er minst en / eksisterer minst en som har fått fisk:  $\exists x: P(x)$

Ingen har fått fisk:  $\neg \exists x: P(x)$  (evt. skrevet med eget symbol,  $\nexists x: P(x)$  ).

### Eksempel 3:

Tabellen under viser rom (101-) som er opptatt eller ikke på et hotell en bestemt natt. Oppt betyr opptatt

101	102	103	104	105	...	...	...
Oppt	...						

Vi definerer det åpne utsagnet **P(x) = «Rom x er opptatt»**

At alle rom er opptatt betyr:  $\forall x: P(x)$ , dvs. at det ikke eksisterer rom som ikke er opptatt.

At minst ett rom er opptatt:  $\exists x: P(x)$ , dvs. det er ikke slik at alle rom er ledige.

At ingen rom er opptatt:  $\neg \exists x : P(x)$  eller  $\nexists x : P(x)$ , det vil si at alle rom er ledige.

For de som kan programmere: hvordan programmerer vi dette?

Vi systematiserer:

	I matematikk	I SQL
Alle, det gjelder for alle	$\forall x: P(x)$ <sup>25</sup>	ALL, FORALL (gjerne underforstått)
Eksistens, finnes minst en	$\exists x : P(x)$ <sup>26</sup>	EXISTS, i noen sammenhenger ANY,SOME
Ikke-eksistens, finnes ingen	$\neg \exists x: P(x)$ eller $\nexists x : P(x)$	NOT EXISTS
<i>Sammenh. mellom <math>\forall</math> og <math>\exists</math></i>	$\forall x: P(x) \equiv \neg \exists (x: \neg P(x))$	dvs.: alle-operatoren er overflødig
+1 en til ....		

Kommenter annonseteksten!

Ikke entydig utsagn.....



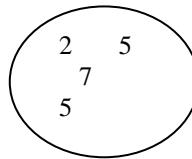
**SOS Rasisme-saken:**  
tiltalte sa seg ikke  
skyldig på alle punkter

Definer  $P(x) = \text{«Personen er skyldig i punkt } x\text{»}$ , og formuler to mulige fortolkninger av utsagnet.

<sup>25</sup>  $\forall x: P(x)$  uttales: for alle  $x$  er det slik at  $P(x)$ , dvs. slik at  $P(x)$  er sann.

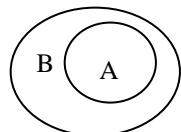
<sup>26</sup>  $\exists x : P(x)$  uttales: det finnes minst en  $x$  slik at  $P(x)$ , dvs. slik at  $P(x)$  er sann.

## Mengder



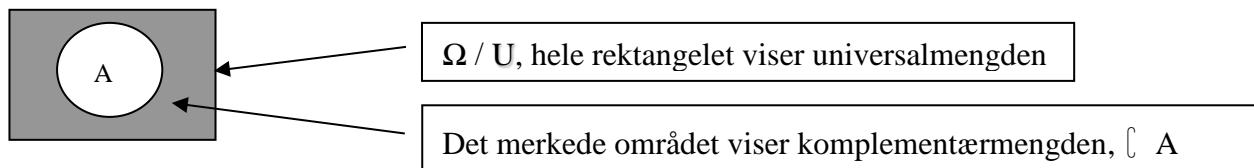
### Grunnleggende begrep

- **Elementer** er "ting" som finnes i en mengde
- Vi gir gjerne **navn på mengder** (ofte brukes M, A eller B, men fritt valg) og sekker/bager
- **Listeform**, f.eks.  $M = \{2, 7, 5\}$ , NB! Ingen rekkefølge, multiplisitet spiller ingen rolle.
- **Mengder** kan ikke ha duplikater (multiplisitet). Dermed er f.eks.  $\{2, 7, 5\} = \{2, 5, 7, 5\}$
- **Mengder** kan ha et **endelig eller uendelig** antall elementer. Eksempler på uendelige mengder er
  - $\mathbb{N}$ : mengden av naturlige tall
  - $\mathbb{Z}$ : mengden av heltall
  - $\mathbb{Q}$ : mengden av brøker
  - $\mathbb{R}$ : mengden av reelle tall.
- **Domene-begrepet.** Et domene er det "havet av verdier" som mengder og elementer kan tas fra.
- **Kardinalitet** (antall elementer) for en mengde. Skrives  $\text{card}(A)$  eller  $|A|$ .
- **Mengdelikhet**,  $A = B$ , hvis de inneholder de samme elementene. Mengdeulikhet:  $\neq$ .
- **Delmengder**,  $A \subseteq B$ ,  $A \supseteq B$ . NB: Her kan A og B være like (jf.  $x \leq y$  for tall).
- **Ekte delmengder, ikke-delmengde**  $\subset, \supset, \not\subset$ .  $A \subset B \equiv (A \subseteq B) \wedge (A \neq B)$



$A \subseteq B$ , mer presist  $A \subset B$

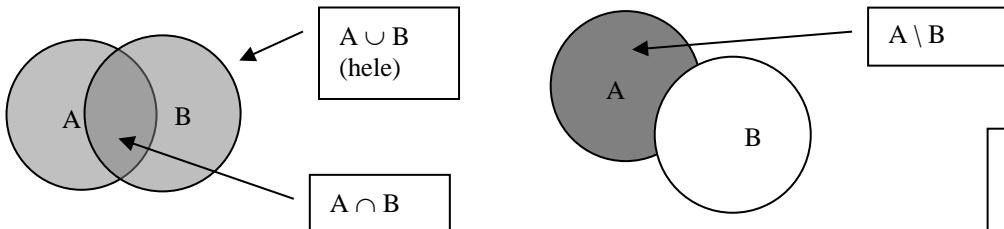
- **Universalmenge** ( $\Omega$  eller  $U$ ), komplementærmengde ( $\complement A$  eller  $\overline{A}$ ).
- **Den tomme mengde**,  $\emptyset$ .  $\text{card}(\emptyset) = 0$ .



Beskrivelse	Matematisk notasjon	Notasjon i SQL
medlemsskap (element i)	$\in$ , f.eks. $x \in M$	<variabel> in (select ....)
ikke-medlemsskap	$\notin$ , f.eks. $x \notin M$	<variabel> not in (select ....)

<i>mengdebygger</i>	$\{x \in D / P(x)\}$ hvor $D$ er et domene	
snitt	$A \cap B$ , med i begge	A INTERSECT B
union	$A \cup B$ , med i minst en	A UNION B
differanse	$A \setminus B$ , mengdedifferanse	A DIFFERENCE B, evt. A MINUS B
<i>symmetrisk differanse</i>	$A \Delta B$	kan uttrykkes via de andre operatorene

Ofte tegnes mengder og mengdesammenhenger via såkalte Venn-diagram



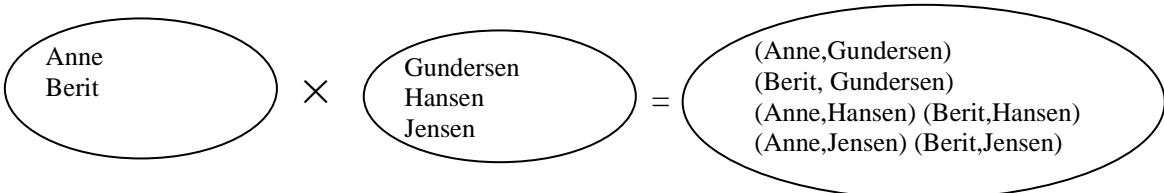
Mer kompliserte mengder kan tegnes på samme måte.

**Prioritet av operatorer**  
 4) parenteser  
 3)  $\setminus$   
 2)  $\cap$   
 1)  $\cup, \setminus$

## Noen begreper og lover om mengder:

- *Disjunkte mengder = ikke-overlappende mengder*
- *Potensmengden = mengden av alle delmenger av en mengde, inkl.  $\emptyset$*
- *Kommutativ og assosiativ lov gjelder for  $\cap$ ,  $\cup$  men ikke for  $\setminus$*
- *Distributiv lov  $A \cap (B \cup C) = A \cap B \cup A \cap C$*
- *deMorgans lover for mengder  $\complement(A \cap B) = \complement A \cup \complement B$ ,  $\complement(A \cup B) = \complement A \cap \complement B$*
- $\text{card}(A \cup B) = \text{card}(A) + \text{card}(B) - \text{card}(A \cap B)$  (tilsvarende regel i sannsynlighetsregning)
- Mengdeprodukt (kartesisk produkt), "alle-mot-alle".  
 SQL: Det vi setter opp i FROM-setningen er i utgangspunktet et mengdeprodukt.

Merk parallelliteten med logikk



### 23.3. Relasjoner og funksjoner

Dette er tatt opp litt i hoveddelen (kap. 2.2, 2.3, 2.4, 2.5 og 17). Her nevnes bare noen få begreper

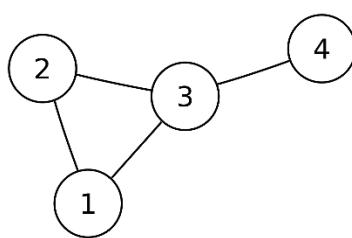
- relasjoner
- funksjonsbegrepet
- *differensligninger (rekursivt definerte funksjoner)*
- determineringer

- en:en, en:mange og mange:mange

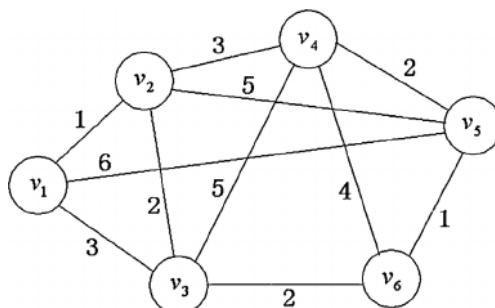
## 23.4. Nettverk

### Generelt om nettverk

Et nettverk (en graf) er et system med **noder** (sirkler, eller kvadrater) og **kanter** (her streker) mellom, som f.eks. Hver kant er nyttet til nøyaktig to noder. Kantene gis ofte et nummer eller et navn.

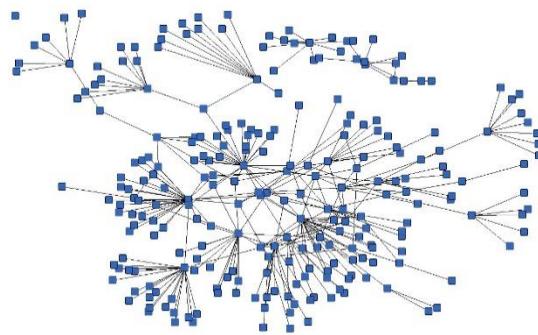


1) Enkel graf



2) Graf med noder ( $v_i$ ), kanter og vekter (tallene)<sup>27</sup>

Nettverket viser som regel en eller annen sammenheng, f.eks. mellom hendelser, personer, datamaskiner, hjerneceller, gatenettverk, flyruter, .... Ofte er vektene et mål for grad av sammenheng, styrke eller antall av noen mellom kantene, avstander el.l.



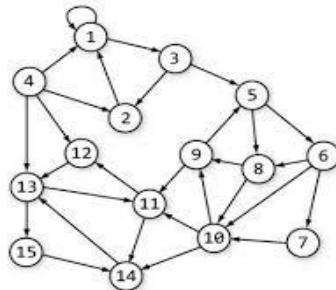
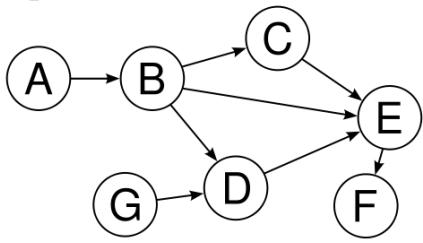
Omfattende graf<sup>28</sup>

Det finnes masse teori rundt grafer, men det er et par enkle begrep som kan være nyttige for oss. En urettet graf (som eksemplene over) viser bare hvilke koblinger som finnes, uten retninger. En rettet graf er en graf som inneholder en retning på kantene, fra en kant til en

<sup>27</sup> Søk på graph, undirected graph, directed graph m.m. gir massevis av resultater. Tegningene er valgt fra dette.

<sup>28</sup> [https://www.google.no/search?biw=1135&bih=505&tbo=isch&sa=1&ei=H1PW076CIKgsgG0norwBg&q=network&oq=network&gs\\_l=psy-ab.3..0l10.57395.59947.0.60448.7.4.0.3.3.0.174.645.0j4.4.0....0...1c.1.64.psy-ab.0.7.673..0i67k1.0.4mMfIOSMUZ0#imgrc=fNUH5hA8C0RhSM](https://www.google.no/search?biw=1135&bih=505&tbo=isch&sa=1&ei=H1PW076CIKgsgG0norwBg&q=network&oq=network&gs_l=psy-ab.3..0l10.57395.59947.0.60448.7.4.0.3.3.0.174.645.0j4.4.0....0...1c.1.64.psy-ab.0.7.673..0i67k1.0.4mMfIOSMUZ0#imgrc=fNUH5hA8C0RhSM):

annen kant (gjerne illustrert med en pil). Rettede grafer kan være asyklike eller sykliske ("piler i sirkel").

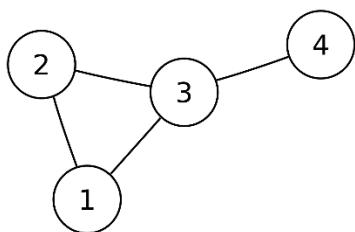


1) Asyklik graf (dermed kan vi finne startpunkter, hvilke?) 2) Syklist graf (hvor mange cykler finner du?)

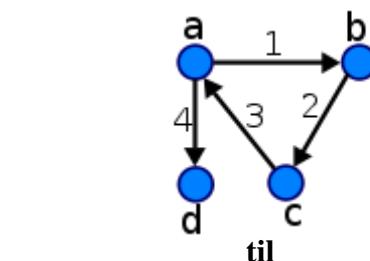
I en asyklik graf kan man finne startpunkter/røtter (hvilke?), slik at man går fra "topp til bunn". I en IT-sammenheng gir sykliske grafer ofte et problem, fordi man må passe på at man ikke "søker i sirkel".

## Nettverk på matriseform

Data i et nettverk kan settes opp som en todimensjonal matrise, se under.



	1	2	3	4
1		x	x	
2	x		x	
3	x	x		x
4			x	



fra      til

	a	b	c	d
a		1		4
b			2	
c	3			
d				

Kommentarer:

- Legg merke til at en urettet graf alltid er symmetrisk om diagonalen, mens en rettet graf normalt ikke er det. Den urettede grafen kan godt ha vekter, men dette gir et problem for den tilsvarende matrisen. Enten må vektene skrives på begge sider av diagonalen (og man må sjekke at de har samme verdi), eller så settes de bare på den "øverste trekanten" (en øvre triangulær matrise), men da må man passe på hvilken del av matrisen man leter i.
- En rettet graf godt kan ha kanter begge veier mellom to noder.
- Dersom man har kanter mellom alle nodene i en rettet graf, kalles det en fullstendig graf, som tilsvarer at alle cellene i matrisen er fylt ut, og det tilsvarer det vi i datamodeller kaller for alle-til-alle relasjonstyper.

## Nettverk på relasjonell form.

Nettverk kan representeres som data i en relasjonsdatabase. Vi tar matrisen til høyre som eksempel. Vi tenker oss at hver av nodene også har en beskrivelse.

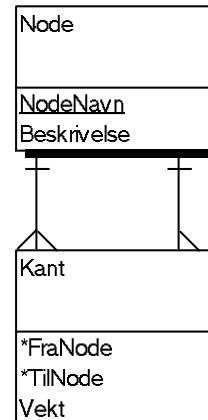
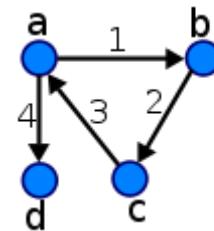
NODE

NodeNavn	Beskrivelse
a	Dette gjelder ...
b	
c	
d	

KANT

FraNode	TilNode	Vekt
a	b	1
a	d	4
b	c	2
c	a	3

1) Data på relasjonell form.

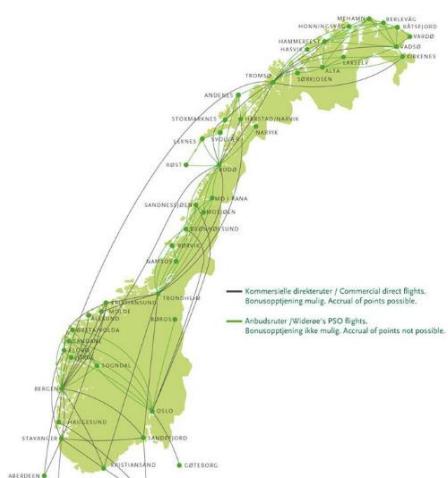


2) Tilsvarende datamodell

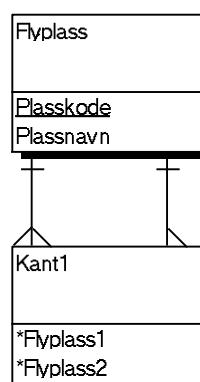
Vi ser at det er de samme dataene, representert på ulik måte. Vi sier at det er en isomorfi (strukturlikhet) mellom de ulike beskrivelse av dataene:



Nettverk brukes for å illustrere en rekke former for data. Her: Widerøes flynettverk:



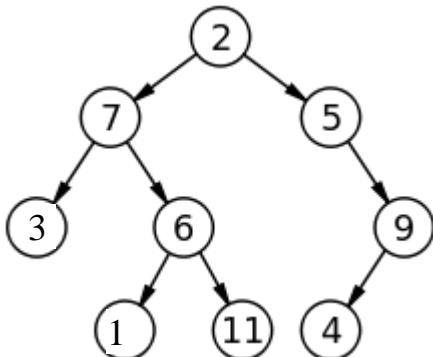
Vi kan naturligvis vise disse dataene både som en matrise og på relasjonell form. I det siste tilfellet blir det



Hvis det bare er hvilke fly som går mellom hvilke flyplasser, blir det en urettet graf. Hvis vi har med avganger, er det en rettet graf. Rutenr blir primærnøkkelen, avg./ank.tid blir vekter.

## 23.5. Trær

Trær er et spesialtilfelle av en rettet graf, hvor man for hver node kan ha et vilkårlig antall underordnede noder, men bare en overordnet node.



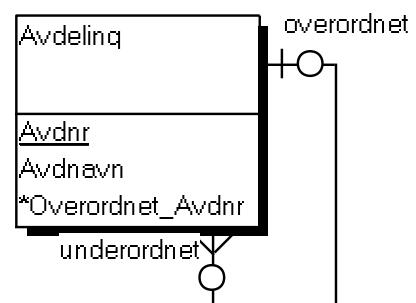
Eksempel på et tre.

Når vi snakker om trær, er det vanlig at vi tenker at vi snur treet opp ned. 2 blir da "rota" i treet, mens "bladene" blir 3, 1, 11, 4 og 9. Et tre kan ha et vilkårlig antall underordnede, og i et vilkårlig antall nivåer. Dersom det aldri har mer enn to underordnede (dvs. to grener), kalles det et binært tre.

En trestruktur kan settes på tabellform / relasjonell form. Hver node har jo bare **en** overordnet, så vi tar med denne, og kaller den overordnet node. Mer konkret kan vi tenke på dette som et hierarki av avdelinger, hvor 2 er den overordnede.

Vi får dermed:

<b>Avdnr</b>	<b>Avdnavn</b>	<b>Overordnet_avdnr</b>
2	Hovedkontor	
7	Produksjon	2
5	.....	2
3	Smådeler	7
6	Store deler	7
9	.....	5
1	Kapping	6
11	Sammensetning	6
4	.....	9



Dette forutsetter at alle nodene har ulik verdi, men det er som regel tilfelle. I databaser vil Avdnr være primærnøkkelen, mens Overordnet\_avdnr vil være en (intern) fremmednøkkelen. Også her er det isometri mellom de ulike representasjonene. Hvorfor må det være det?

På samme måte som for nettverk finnes det en masse teori og praksis rundt trær, både innenfor matematikk, informatikk og andre fagområder. Et filesystem i vanlige operativsystemer er et enkelt eksempel på et tre. (Legger vi til pekere / snarveier i systemet, så bryter det den grunnleggende trestrukturen).

## 23.6. Logikk, med matematiske symboler (alternativ til kap. 23.1).

Beskrivelse	(Matematisk) notasjon	Notasjon i SQL	
Sant	s, sann, true, t, <sup>29</sup> ✓	TRUE	
Usant	u, usann, false, f, ✗	FALSE	Brukes sjeldent direkte, fordi et utsagn i seg selv er sant eller usant
Utsagn	f.eks. p		
og	$p \wedge q$	AND	
eller	$p \vee q$	OR	
eksklusiv eller	$p \veebar q$ (evt. $p \oplus q$ )	XOR	
ikke	$\neg p$ (evt. $\sim p$ eller $\bar{p}$ )	NOT	

Flere av disse har presise betydninger i matematikk som ikke nødvendigvis stemmer med dagligspråket.

### Sannhetsverditabeller, bl.a

<b>p</b>		<b><math>\neg p</math></b>
s		u
u		s

<b>p</b>	<b>q</b>	<b><math>p \wedge q</math></b>	<b><math>p \vee q</math></b>	<b><math>p \veebar q</math></b>		<b><math>p \Rightarrow q</math></b>	<b><math>p \Leftrightarrow q</math></b>
s	s	s	s	u		s	s
s	u	u	s	s		u	u
u	s	u	s	s		s	u
u	u	u	u	u		s	s

### Prioritet av operatorer ("hva binder sterkest sammen")

logiske operatorer

- 4) parenteser
- 3)  $\neg$
- 2)  $\wedge$
- 1)  $\vee, \veebar$

parallel til aritmetikk:

- parenteser  
- som fortegn  
 $\bullet, /$   
 $+, -$

Det finnes en rekke lover/regler for logikk. Noen av de er:

- kommutativ lov  $p \wedge q \equiv q \wedge p$ , tilsvarende for  $\vee$ .
- assosiativ lov  $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r$ , tilsvarende for  $\vee$
- distributiv lov  $p \wedge (q \vee r) \equiv p \wedge q \vee p \wedge r$
- noen andre lover:  $p \wedge \neg p = u$ ,  $p \vee \neg p = s$ ,  $p \wedge u = u$ ,  $p \vee u = p$  m.fl.
- deMorgans lover:  $\neg(p \wedge q) \equiv \neg p \vee \neg q$ ,  $\neg(p \vee q) \equiv \neg p \wedge \neg q$
- Sannhetsverditabeller kan brukes for å finne ut av mer kompliserte utsagn, f.eks.  $(\neg p \wedge q) \vee (p \wedge \neg q)$

<sup>29</sup> Nøn ganger brukes også binære tall, f.eks. 1 = sann, 0 = usann.