

# Introduksjon til Avanserte Databaser

Tore Mallaug

30.8.2011

Lærestoffet er utviklet for faget LO326D Avanserte databaser

## 1. Introduksjon og bakgrunn

*Resymé: I denne leksjonen introduseres faget og vi ser kort på faginnholdet og læringsmål. I tillegg gis en kort oppsummering av hvilke grunnleggende database- og datamodelleringskunnskaper du bør ha før du går i gang med faget.*

### Innhold

1.1.	INNLEDNING .....	1
1.2.	FORKUNNSKAPER .....	2
1.3.	DATA REPRESENTASJON VS. DATA LAGRING .....	2
1.4.	DATABASEDESIGN .....	4
1.4.1.	3-nivå modell.....	4
1.4.2.	Klient/tjener.....	5
1.4.3.	Normalisering .....	6
1.5.	ULIKE DATAMODELLER .....	6
1.5.1.	ER – diagram .....	6
1.5.2.	Relasjonsdatamodellen.....	8
1.5.3.	XML.....	9
1.5.4.	Andre datamodeller.....	10
1.6.	SPØRRINGER .....	10
1.6.1.	SQL.....	10
1.6.2.	XML.....	10
1.6.3.	Transaksjoner.....	10
1.7.	INNHold I FAGET.....	11
1.7.1.	XML.....	11
1.7.2.	Datavarehus .....	11
1.7.3.	Databaser og Web.....	12
1.7.4.	Andre spesielle databaser .....	12

### 1.1. Innledning

Dette faget omhandler noen videregående emner innen databaser og datamodellering. Hvordan skal data representeres og lagres for å kunne utnyttes best mulig av dataintensive programmer? Hvordan skal ulike programmer og brukergrupper få tilgang til felles dataressurser, f.eks. på Internett? I et moderne informasjonssamfunn er ofte ikke tilgang til data problemet, heller hvordan finne de riktige dataene til riktig tid, og å finne strukturerte data av høy kvalitet uten å måtte bruke komplekse og dyre programvareløsninger.

Databaseløsninger spiller en viktig rolle her, og kan både effektivisere, forenkle og totalt sett gjøre datalagring, datautveksling og datagjenfinning billigere.

Effektiviteten til mange større programmer er avhengig av kvaliteten til den logiske og fysiske dataorganiseringen, og kvaliteten til spørrespråk for å hente ut data.

Generelt kan vi bruke datarepresentasjon og datalagring til:

1. Åpenbart, til å lagre data sikkert i en database, slik at de er tilgjengelige for ulike programmer i ettertid ved bruk av et spørrespråk, f.eks. SQL
2. I tillegg kan datarepresentasjonen brukes til å utveksle, eller sende, dataene mellom ulike aktører, for eksempel i e-handel. XML er et godt eksempel på dette.

I dette faget ser vi på ulike spesialiserte måter å lage databaser på. I tillegg ser vi på hvordan XML kan brukes i databasesammenheng, som støtter opp rundt pkt. 2 over.

## 1.2. Forkunnskaper

For å gjennomføre dette kurset forventes det at du har litt kunnskaper innen databaser og datamodellering fra før. Oppsummert er de viktigste bakgrunnsemnene:

- Relasjonsdatamodellen og relasjonsdatabaser med viktige begreper som tabell/relasjon, attributt, sammenhengstyper (eng. relationships) og nøkkelbegrepet (primær- og fremmednøkler)
- Enkle SQL-spøringer inkl. ”joining” (forening) av tabeller
- Enkel ER-modellering, helst også litt om spesialisering / generalisering av entitetstyper (EER-modellering)
- Gjerne ha litt erfaring fra et databasesystem, som MS-Access, MS-SQL Server, Oracle eller MySQL.
- Fint om du har litt kunnskaper om Internet og HTML. XML kan du lære deg i løpet av kurset, men merk deg at dette ikke er et XML-kurs (vi tilbyr et eget fjernundervisningskurs i XML for de som er interessert).
- Elementær kunnskap om operativsystem, filer og fysisk datalager (harddisk)

Mangler du denne kunnskapen ved kursstart, eller føler behov for repetisjon, kan du lese den norske læreboka ”Databaser” (på Gyldendal/Tisip), eller ei hvilken som helst innføringsbok i databaser på engelsk. I denne leksjonen kommer jeg til å nevne kort det mest grunnleggende, men leksjonen er ikke ment som en fullstendig innføring i databaser.

## 1.3. Data representasjon vs. data lagring

Egentlig snakker vi grovt sett om to fagområder: 1) databaser 2) datamodellering. I begge tilfeller brukes *datamodeller* til å beskrive, eller representere data som skal lagres. En datamodell brukes som et verktøy for å representere data på en logisk måte. Ulike datamodeller er tilpasset ulike behov. Generelt er disse behovene litt forskjellige i datamodellering vs. databaser. Mer spesifikt kan en ha datamodeller med spesielle egenskaper til bruk for spesifikk programvare eller informasjonssystemer. For eksempel har datamodeller for å representere XML-dokumenter, eller datamodeller som brukes for datavarehus, særegenheter tilpasset bruksområdet til programmene og brukerne.

*Datamodellering* prøver å finne måter for å representere data på en slik måte at semantikken til dataene kommer best mulig frem. Semantikken er knyttet til den "virkeligheten" dataene er hentet fra, i videste forstand ontologier som brukes i den aktuelle "virkeligheten". Tradisjonelt er datamodeller som brukes i datamodellering, som ER-modellen, såkalte konseptuelle datamodeller. Slike modeller lar seg ikke nødvendigvis implementere direkte i et databasesystem. Derfor må en ha en mapping, eller oversetting, fra datarepresentasjonen til lagringen av dataene i selve databasen. F.eks. en mapping mellom et ER-diagram og en relasjonsdatabase.

*Databaser* handler om hvordan dataene logisk sett bør lagres, eller organiseres, slik at de blir lette å hente ut igjen fra fysisk medium (internminne og harddisk), samtidig som de er trygt lagret. Databaser administreres av databasesystem (DBMS) som typisk tilbyr flere brukere å aksessere samme database samtidig. Dette er ideelt for deling av data mellom ulike brukere/brukergrupper, samt at det gir en effektiv måte å lagre data på, f.eks. kan vi unngå at samme data lagres flere plasser og at det blir lettere å holde dataene oppdatert (endringer kan skje kontrollert og kun i en felles database). Datamodeller her tar derfor typisk hensyn til to faktorer:

- At dataene skal kunne lagres effektivt i en bestemt type databasesystem, f.eks. relasjonsdatabasesystem.
- At dataene skal kunne aksesseres raskt i databasesystemet ved hjelp av et spørrespråk, f.eks. SQL.

Den absolutt mest vanlige datamodellen på databasenivået er relasjonsdatamodellen. Her lagres data som tupler i tabeller (eller egentlig relasjoner) som lett lar seg mappe ned til f.eks. filer på en harddisk. Datateknisk gjøres dette av databasesystemet i samspill med operativsystemet, filsystemet og maskinvare på lagringsmediet.

Vi kan altså skille mellom datarepresentasjon og datalagring. I dette faget heller vi mot datarepresentasjon, men siden både XML og datavarehus som utgjør størstedelen av faget også kan brukes i datalagring, blir det litt både óg. Hvis faget var et rent datalagringsfag ville emner som optimalisering av spørring (eng. query optimization) og "tuning" av databasen vært viktige tema. Dette omtaler bare sporadisk i leksjonene. Derimot hvordan data kan representeres for at de lett kan brukes i e-handel, er et fremtidsrettet og interessant tema som omtales grundigere.

*XML-teknologi* representerer data uavhengig av program- og maskinvare (betegnelsen XML-teknologi brukes her for alle teknologier knyttet til det XML-miljøet på Internett, mens XML brukes om selve språket for å representere et XML-dokument). I et XML-dokument er selve XML-koden gitt i ren tekst strukturert på en bestemt måte (en samling av elementer), og derfor er koden (eller rettere datainnholdet den representerer) mulig å lagre i en database. Så skillet mellom datamodellering og database er her noe uklart. Skillet er i større eller mindre grad avhengig av valgt databaseløsning. Til tross for at dataene i XML-dokumentet ikke er representert som tabeller i relasjonsdatamodellen, kan de fortsatt lagres i en relasjonsdatabase hvis det finnes en mapping mellom XML og tabellene. Dette er et av temaene vi skal se nærmere på i kurset.

Et nytt emne er hvordan data fra Web / Web 2.0 kan lagres og representeres på databasenivå, f.eks. data fra sosiale verktøy som en Wiki eller ulike nettportaler ("nettsamfunn") for utveksling av informasjon og kunnskap mellom deltakerne (eng. Community Information Management Systems, forkortes CIM systems). Data fra slike systemer vil være semistrukturelle, og kan derfor knyttes til XML-teknologi (selv om dette neppe er så vanlig

foreløpig). Videre vil det ofte være ønskelig å lagre dataene i flere (historiske) versjoner for og f.eks. vise endringer gjort av brukere underveis i en Wiki. Dette kan generelt knyttes til temporale aspekter rundt dataene tilsvarende en temporal database. Dette kommer vi tilbake til senere i kurset.

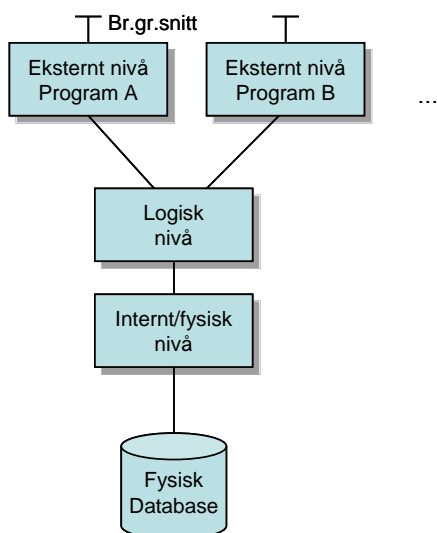
I et datavarehus er situasjonen litt annerledes. Her har vi egne datamodeller som er spesielt beregnet for å bygge opp et datavarehus og for å utføre søk mot det. Et datavarehus er et spesialisert system som har utvidede muligheter i forhold til en "vanlig" database. Det samme kan sies om andre spesialiserte databasesystem, som temporale og romlige. XML-tenkologi er på en måte et eksempel på det motsatte: en teknologi som prøver å generalisere datarepresentasjon - en standard for å utveksle data mellom ulike aktører på Internet, hvor systemene rundt på nettet er såkalt heterogene, d.v.s. det er ikke sikkert de snakker det samme språket, kanskje de bruker ulike standardene eller terminologier.

I alle disse tre tilfellene er det i databasesammenheng snakk om å utvide den tradisjonelle databasen til å inneholde mer enn kun ren tekst- tekst som vi tradisjonelt ikke vet mye om. Ofte vet vi bare datatypen til et attributt i et tuppel; f.eks. tekst, tall eller dato. Behovet for både å lagre og representere noe mer enn simple datatyper er stigende i et voksende informasjonsamfunn.

Under nevner jeg kort noen hovedpunkter for databasedesign, datamodeller og spørringer.

## 1.4. Databasedesign

### 1.4.1. 3-nivå modell



Figur: ANSI/SPARC-arkitekturen

Generelt kan vi betrakte data i en database på 3 nivåer. Denne fremstillingen er kalt ANSI/SPARC-arkitekturen (se evt. Databaser-boka s. 31-33). Mellom hvert nivå må det finnes en "mapping". Hvert nivå kort forklart:

- Eksternt nivå

Dette er hvordan databasen ser ut for vanlige sluttbrukere av et program / applikasjon som lagrer og henter ut data fra databasen. Ofte vil et program kun bruke ("se") deler av databasen, og kan ha et brukergrensesnitt spesialtilpasset et utvalg sluttbrukerne og de operasjoner de har lov til å gjøre mot (deler av) databasen. **Vi kan ha flere ulike programmer som bruker den samme, felles, databasen.**

På dette nivået brukes et DSL (eng. Data Sub Language) til å kommunisere med logisk nivå under. DSL deles opp i *DDL* (datadefineringspråk) for å opprette nye elementer i en database og *DML* (datamanipuleringspråk) for å lese, skrive, slette og oppdatere datainnholdet i databasen. Eksempel på et DSL-språk er SQL. I DDL-delen av SQL kan du opprette nye tabeller i en relasjonsdatabase med kommandoen CREATE TABLE. I DML-delen bruker du

kommandoene SELECT for å lese data og UPDATE, INSERT og DELETE for å manipulere en database.

- Logisk nivå

Dette er hvordan databasesystem representerer den (fysiske) databasen på et konseptuelt nivå. D.v.s. det er en abstrakt fremstilling av databasen, men dog mindre abstrakt enn f.eks. et ER-diagram, siden DBMS bruker en såkalt implementeringsdatamodell (ER-modellen er ikke det). I relasjonsdatabaser er denne modellen tabeller, med kolonner/tupler, primær- og fremmednøkler, og integritetsregler.

Brukere på dette nivået er DBA – Databaseadministrator.

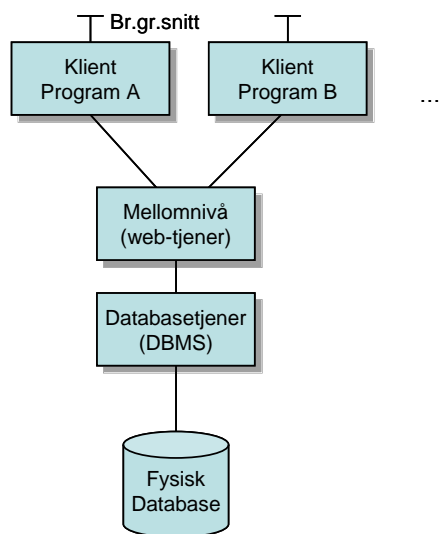
Ved å ha en logisk datamodell oppnår vi *logisk datauavhengighet*. Dette er et viktig prinsipp fordi vi da har en representasjon av databasen som er uavhengig av hvordan databasen fysisk er lagret på lagringsmedium (uavhengig av operativsystem og maskinvare). Dette gjør det enklere å flytte databasen til f.eks. ei ny maskin, et nytt operativsystem eller et nytt DBMS (gitt at det nye DBMS-et bruker den samme logiske datamodellen som det gamle).

- Internt/fysisk nivå

Dette nivået sier helt konkret hvordan databasen lagres på filer på fysisk medium. Når vi bruker et DBMS vil vi ikke se dette nivået, siden DBMS tar seg av dette internt (det er ett av hovedpoengene med å bruke et DBMS, at det tilbyr et spørrespråk (DSL) er det andre poenget). Et større DBMS som Oracle vil ha funksjonalitet som gjør det mulig for DBA å påvirke den fysiske lagringen, f.eks. valg av lagringsstruktur og opprettelse av indekser som gjør enkelte søk i en tabell raskere.

*Fysisk datauavhengighet* gjør at vi kan endre den fysiske lagringen uten at det logiske skjemaet på logisk nivå over påvirkes.

### 1.4.2. Klient/tjener



Figur: 3-lags klient/tjener-løsning

Nevner bare kort at de fleste databaser i dag inngår i en klient/tjener-arkitektur. Databasesystemet ligger da på en sentral maskin i et datanett (Internett eller et internt lukket nett). Denne maskina kalles databasetjener, eller database-server. Flere ulike programmer kan da aksessere en felles database via databasetjeneren over nettet. Disse programmene kan kalles klienter. Dette samsvarer med ANSI/SPARC-arkitekturen i kap. 1.4.1 over. Klientene trenger ikke å vite hvordan databasen fysisk er lagret. Den vanlige er nok at databasen fysisk er lagret på databasetjeneren, men i prinsippet kan den fysiske lagringen være ekstern på en eller flere maskiner, f.eks. på en ekstern ”databank” som leier ut lagringskapasitet

(inkl. sikkerhetskopiløsninger) til de som måtte trenge.

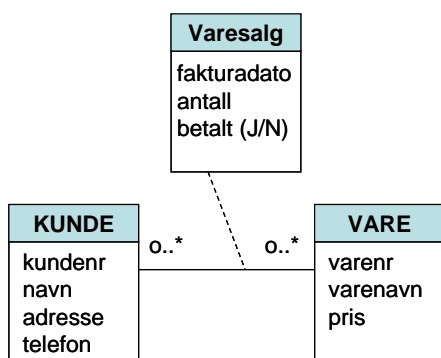
På Internett er det vanlig å ha en såkalt 3-lagsløsning for klient/tjener (eng. 3-tier). Her har vi et mellomnivå mellom klientene og databasetjeneren. Dette mellomnivået er typisk en web-tjener. En klient som kommuniserer med databasetjeneren i en slik løsning, kommuniserer via web-tjeneren i stede for direkte med DBMS. 3-lagsløsninger gjør det også mulig å lage et distribuert system som inkluderer flere databasetjenerer. I dette faget begrenser vi oss til å se på datarepresentasjonen og lagringen; datakommunikasjonen dekkes av andre fag, f.eks. web-programmering i ASP.NET, JSP eller PHP. Er du interessert kan du lese om klient/tjener i kap. 9 og 10 i Databaser-boka (2.utg.).

### 1.4.3. Normalisering

Normalisering er et veldig bra prinsipp for god databasedesign. Kort sagt unngår en normalisert database dobbeltlagring av data internt i databasen. F.eks. at persondata om de ansatte kun ligger på en (felles) plass i databasen. Dette gjør at dataene er lettere å vedlikehold – hvis en ansatt flytter, trenger firmaet kun å endre adressen en gang på en plass i det totale informasjonssystemet, ikke i mange tabeller eller i mange små lokale databaser. Dette øker kvaliteten på lagret data – faren for feil data i databasen reduseres.

I relasjonsdatamodellen ble en normalisert løsning formalisert allerede tidlig på 70-tallet av Codd. Han satte opp et sett med regler kalt normalformer (1., 2., 3. og Boyce-Codd normalform er mest kjent). Ved å undersøke en tabell kan denne dekomponeres, eller splittes opp, i flere mindre tabeller slik at dobbeltlagring unngås (med unntak av nøkler som må dobbeltlagres for at tabellene skal henge sammen i den totale databasen).

I dette faget kommer vi ikke til å fokusere på normalisering, men jeg nevner dette prinsippet innledningsvis fordi det er viktig. I XML-teknologi finnes det ingen kjente normaliseringsregler enda, men det forskes på området. Det kan finnes dobbeltlagring av data i et XML-dokument også – de samme elementene kan forekomme flere ganger i samme dokument. Det finnes i dag forskning på hvordan slik dobbeltlagring kan oppdages for at XML-dokumenter kan ”trimmes” til å passe bedre inn i god databasedesign. Så normalisering er altså ikke et prinsipp som bare angår store ”gamle” relasjonsdatabaser – prinsippet kan tas med inn i andre datamodeller også.



Figur: ER-diagram med mange-til-mange

## 1.5. Ulike datamodeller

Nevner her kort de aktuelle datamodellene for dette faget. ER og relasjonsdatamodellen antas kjent fra før. XML og de andre spesielle datamodellene kommer vi tilbake til i de resterende leksjonene.

### 1.5.1. ER – diagram

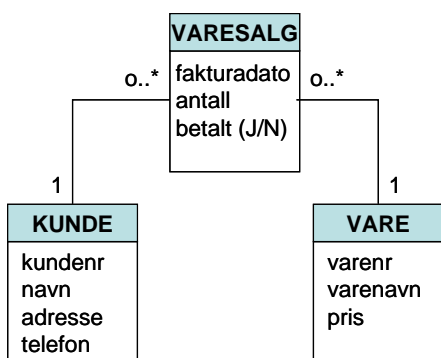
ER-modellen er en konseptuel datamodell som brukes i datamodellering. Den ligner på relasjonsdatamodellen, men har noen ulikheter:

- I ER snakker vi om *entitetstyper* i stede for

tabeller. En entitetstype beskriver en *entitet*, typisk et substantiv i den virkelige verden, f.eks. en person, en bil, en faktura, en ordre. En entitet vil ha et sett med attributter, og sammenhenger (relasjoner) til andre entiteter.

- Det er ikke noe krav at entitetstypene i et ER-diagram skal ha en entydig nøkkel. Valg av primærnøkkel kan gjøres ved oversetting til tabeller, så vi trenger ikke å tenke på dette i modelleringsfasen. I relasjonsdatamodellen kreves det i praksis at hver tabell har en primærnøkkel. I ER er det heller ikke nødvendig å spesifisere datatypen til attributtene. Dette må gjøres i relasjonsdatamodellen.
- Vi har også et klassebegrep. En klasse er en samling av entiteter av samme entitetstype. Dette blir ofte i praksis det samme som en tabell i relasjonsdatamodellen. Klassebegrepet er dog viktigst hvis du skal drive med objektorientert programmering som Java eller C++. I vårt fag har vi ikke så mye bruk for klasser.
- I ER tillates tre ulike sammenhengstyper (bruker ordet 'sammenheng' i stede for 'relasjon', fordi relasjon er det egentlige navnet på en tabell i relasjonsdatamodellen). Disse er en-til-en, en-til-mange og mange-til-mange. ER-diagrammet i figuren over viser to entitetstyper, KUNDE og VARE. Mange-til-mange sammenhengstypen Varesalg gjør det mulig for en kunde å handle null, en eller flere varer. For hver handel blir fakturadato, antallet av varer som kjøpes registrert, pluss attributtet 'betalt' som viser om antallet er betalt eller ikke (boolean datatype). Hver forekomst av Varesalg kan være ei linje på en samlefactura for kunden. Hver vare kan handles av ingen, en eller flere kunder.

En relasjonsdatabase kan ikke representere mange-til-mange, så slike sammenhengstyper må omskrives til to en-til-mange i stede. Se eksempel i figuren under.



Figur: ER-diagram med en-til-mange

- I ER (eller egentlig en utvidet utgave kalt EER (Extended ER)) er det også mulig å modellere spesialiseringer av en entitetstype. I en vanlig "enkel" relasjonsdatabase er spesialisering ikke mulig, men en utvidet SQL-versjon, SQL3, åpner for dette. SQL3 er ikke pensum i dette faget, så hvis vi bruker "vanlig" SQL vil spesialiseringer forsvinne ved oversetting til tabeller. Men vi har fortsatt muligheten til å lagre dataene fra hver entitetstype i tabeller (forskjellen er at det kommer ikke fram i tabellene at det er snakk om spesialisering).

### 1.5.2. Relasjonsdatamodellen

Relasjonsdatamodellen bygger på et sett med matematiske operasjoner, kalt relasjonsalgebra. Veldig kort går algebraen ut på å betrakte dataene som mengder, eller sett, med tupler som har noe felles. En slik mengde kalles en relasjon, men i dagligtale blir dette en tabell. Hver tabell, eller relasjon, har et entydig navn. Tabellen består av tupler (kolonner). Hvert tuppel har et sett med attributter. Ett (evt. et sett) av attributtene velges til primærnøkkel. En primærnøkkel er entydig i tabellen og brukes til å identifisere hvert enkelt tuppel. Et tuppel kan ha en eller mange sammenhenger til andre tupler enten i samme tabell (rekursive sammenhenger) eller mer vanlig til tupler i andre tabeller. For å vise sammenhenger brukes fremmednøkler. En fremmednøkkel refererer til en primærnøkkel i samme tabell eller i en annen tabell. På denne måten kan alle tabellene i en database ”limes” sammen for å finne riktig resultat i spørringer mot databasen – dette kalles forening av tabeller, eller ”join-ing”.

Merk deg i tillegg følgende:

- *Dataintegritet*: Alle relasjonsdatabasesystemer på markedet i dag tilbyr dataintegritet for å lette vedlikeholdet av tuplene i tabellene. Mest kjent er referanseintegritet. Denne settes på sammenhengstypene mellom tabellene, og medfører at det ikke er mulig å slette et tuppel som refereres av andre tupler i databasen. For eksempel i tabellene under er det ikke mulig å slette en kunde i KUNDE som har kjøpt varer. Kjøpene kunden har registrert i VARESALG må slettes først.
- *Automatisk primærnøkkel*: Valg av primærnøkkel i en tabell kan i praksis ofte være vanskelig. Kanskje det ikke finnes noe attributt som naturlig er entydig. I tillegg er det uheldig å endre verdien på en primærnøkkel, for eksempel hvis en har valgt kundennummer som primærnøkkel og man senere ønsker å endre på kundennummeret blir det fort mye rot. Derfor tilbyr de fleste relasjonsdatabasesystemer automatisk genererte primærnøkler som brukerne ikke trenger å tenke på. Dette er som regel et heltall (integer) som DBMS sørger for å holde entydig for alle tuplene i en tabell. Velg automatisk primærnøkkel i tilfeller hvor det ikke er noen annen naturlig nøkkel.

De oversatte tabellene fra ER-diagrammet i kap. 1.5.1 med noen tupler kan se slik ut.

KUNDE			
kundenr	navn	adresse	telefon
1	Tore Mallaug	Trondheim	
2	Ole Olsen	Moss	
3	Eva Larsen	Levanger	

VARE		
varenr	varenavn	pris
1	hammer	kr 49,90
3	sag	kr 79,00
4	slagbor	kr 395,00

VARESALG					
ID	Kundenr*	Varenr*	fakturadato	antall	betalt
1	1	3	01.09.2004	3	Ja
3	2	1	01.09.2004	200	Nei
4	1	3	02.09.2004	5	Nei
8	2	4	02.09.2004	1	Nei
9	1	4	02.09.2004	1	Nei

Understrekne attributt er valgte primærnøkler. \* indikerer fremmednøkler. Datatyper er ikke tatt med her. I tabellen VARESALG har jeg latt databasesystemet (i dette tilfellet var det MS-Access) generere en primærnøkkel automatisk, her ID, siden det ikke finnes noen naturlig entydig nøkkel her (kombinasjonen av Kundenr + Varenr er ikke entydig siden en kunde kan bestille samme varen flere ganger).

### 1.5.3. XML

XML beskrives i de kommende leksjonene, så bare veldig kort her.

XML-dokumenter er ikke representert som tabeller, men som elementer, eller ”tags” i tekstformat. Dette kan virke litt uoversiktlig, men typisk vil elementene være satt sammen i et hierarki som grafisk kan vises som en trestruktur. Tanken er at XML skal kunne brukes til såkalte semistrukturelle data – data som ikke nødvendigvis følger en fast struktur. I databasesammenheng er det dog en stor fordel at dataene har en viss struktur, en viss forutsigbarhet om du vil. Helt ustrukturelle data må fort lagres som ren tekst i en database. Derfor vil de XML-dataene vi ser på i dette kurset i praksis være strukturert etter et skjema, som i praksis er en enkel datamodell.

Tre basisbegrep i XML-teknologien:

- XML-dokumentet (.xml-fila) som inneholder selve dataene
- Stilark, f.eks. Cascading Stylesheet (CSS) eller XSL, som beskriver hvordan et XML-dokument skal vises på skjermen (web-browsersen). En av forskjellene på HTML og XML er at XML-dokumentet ikke inneholder tags som beskriver hvordan teksten skal vises på skjermen, derfor er det behov for stilark. I databasesammenheng er dette greit, for vi har skjelden behov for å lagre om f.eks. et kundenavn eller en varebeskrivelse skal vises med fet skrift eller ikke på skjermen, så like greit å overlate dette til stilark.
- Skjemaer som spesifiserer hvordan elementene i XML-dokumentet skal eller bør struktureres. Dette er svært interessant hvis dataene skal lagres i en database!

De to mest kjente skjemastandardene som kan brukes i databasesammenheng er DTD og XML Schema (heretter kalt XML-skjema). Den enkleste er DTD. Denne har spesielt begrensinger når det gjelder valg av datatyper. Den andre er XML-skjema. Denne har flere muligheter med tanke på datatyper og er mer fleksibel på hvilke rekkefølge elementene i et XML-dokument har. De finnes andre skjemastandarder også, f.eks. RELAX NG, ment som et noe enklere alternativ til XML-skjema.

Et databasespørsmål vi kan stille er hvordan representere sammenhenger i XML. Så lenge vi opererer ”internt” i et XML-dokument, vil elementene i dokumentet henge sammen i en trestruktur, og sammenhengene er da gitt. Verre blir det hvis vi ønsker å sammenhenger på tvers av flere dokument. XML-teknologien har en løsning for dette kalt XLink.

#### 1.5.4. Andre datamodeller

I kurset kommer vi i tillegg innom datavarehus og temporale databaser. Disse kan ha egne spesialiserte datamodeller bygd på utvidelser av relasjonsdatamodellen.

### 1.6. Spørringer

#### 1.6.1. SQL

Det blir alt for omfattende å gå igjennom de ulike spørringene i SQL her. Jeg beskriver heller spørringene etter hvert som de brukes i de ulike leksjonene (dette blir aktuelt i forbindelse med eksempler i Oracle). Generelt er SQL-spørringene knyttet opp mot relasjonsalgebraen, selv om du som sluttbruker ikke trenger å tenke så mye på det. Det er likevel viktig å vite at alle utvalgsspørringer, SELECT i SQL, alltid returnerer en tabell, eller relasjon, som svar. Hvis du lagrer svaret som et såkalt VIEW (CREATE VIEW) kan svaret senere brukes i nye spørringer som om det var en tabell.

Oppsummert er de viktigste SQL-kommandoene, eller spørringene, disse:

- CREATE – Opprette en database, en tabell, et VIEW eller en indeks
- ALTER – Endre på strukturen til en tabell
- DROP – Slette en hel tabell, både struktur og innhold (tuplene) forsvinner
- INSERT – Sette inn et nytt tuppel, rad, i en tabell
- SELECT – Hent ut et utvalg av data fra en eller flere tabeller, evt. fra VIEWS
- DELETE – Slette en eller flere tupler
- UPDATE – Oppdatere en eller flere tupler. Bare angitte attributter (felt) endres.

Henviser til Web eller bøker for innføring i SQL.

#### 1.6.2. XML

XML-teknologien mangler (så langt) en fullgod standard tilsvarende SQL. For rene XML-databaser er dette en svakhet. Relasjonsdatabaser forsøker å lagre XML-data i tabeller, og kan derfor "lure seg til" å bruke SQL også mot disse dataene (men hvor effektivt dette er er et annet spørsmål).

MEN XML-miljøet prøver å kompensere ved å introdusere sitt eget spørrespråk spesielt beregnet for XML, kalt *XQuery*. Vi kommer tilbake til dette språket i en egen leksjon.

#### 1.6.3. Transaksjoner

Nevner veldig kort at mot større databaser er det vanlig å kjøre transaksjoner i stede for enkeltspørringer. En transaksjon er et (lite) program som kan inneholde en samling spørringer som henger sammen. F.eks. ved uttak av kontanter fra en bankkonto er det i alle fall to spørringer som må utføres;

- Finne aktuelle bankkonto i databasen, og sjekke at det finnes dekning på kontoen
- Hvis dekning, beregn ny saldo etter uttaket og oppdater denne

Det første punktet krever en SELECT-spørring, så må programmet foreta litt beregninger, før den i siste punktet kan utføre en UPDATE-spørring.

Kjøres hele dette programmet som en transaksjon i DBMS, er det mulig å sikre at andre brukere ikke oppdaterer den samme kontoen samtidig (f.eks. ved å låse hele eller deler av tabellen for andre brukere mens transaksjonen pågår). Dette er viktig i flerbrukerdatabaser for å unngå rot ved oppdaterer (hvis brukerne kun har leseaksess er dette ikke så viktig). Videre skal en transaksjon betraktes som en atomisk operasjon mot databasen. Dette betyr at enten utføres hele transaksjoner, eller ingenting. Dette kan høres åpenbart ut, men i et flerbrukersystem kan en transaksjon bli avbrutt av operativsystemet (p.g.a. deling av maskinressurser), og det er da viktig at transaksjoner gjør seg ferdig og lagrer resultatet ”stabil” i tabellene etter slike avbrudd, eller alternativt kanselleres. I SQL kan dette løses dette ved å utføre kommandoen COMMIT ved avslutning av transaksjoner – før COMMIT er ikke endringene lagret i databasen, eller ROLLBACK ved kansellering (ingen endringer transaksjonen har gjort blir da lagret). Kommersielle relasjonsdatabasesystem i dag har i tillegg egne løsninger for prosessering av transaksjoner internt i databasesystemet. Eksempler er automatisk utførelse av COMMIT (uten at "brukeren" trenger å tenke på dette), og ulike såkalte isolasjonsnivåer som påvirker graden av hvor atomisk en transaksjon faktisk kjøres internt i databasesystemet (dette for å effektivisere prosesseringen). Skal du programmere, f.eks. i Java, mot et bestemt relasjonsdatabasesystem må du derfor lese manualen og sette deg grundig inn i hvordan DBMS-et håndterer transaksjoner i praksis!

Transaksjonsbehandling er et eget fag i seg selv. Det er mer naturlig å knytte transaksjoner til programmeringsfag. Det er allikevel viktig å nevne begrepet siden det er et av de sentrale innen databaser.

## 1.7. Innhold i faget

Under ramser jeg opp hovedpunkter fra innholdet i faget. Alt dette kommer vi selvsagt tilbake til i leksjonene fremover.

### 1.7.1. XML

- XML-dokumenter
- Document Type Definition (DTD)
- XML-skjema (XML Schema)
- Mulige måter å lagre XML-dokumenter på
- Rene XML-databaser
- Koblingen mellom XML og relasjonsdatabaser
- Lagring av XML i RDBMS (Oracle som eksempel)
- Publisering av tabeller fra RDBMS som XML-dokumenter
- Utføre spørring mot XML-data, både i SQL og XQuery
- Bruk av databaser i B2B (eller e-handel, f.eks. den norske standarden e2b.no)

### 1.7.2. Datavarehus

- Datavarehus

- Grunnleggende begreper
- OLAP og OLAP-spørringer
- Datavarehusarkitektur
- Ulike typer data, datatransformasjon, datamarts
- Stjernemodell
- Bestemme seg for riktig detaljnivå
- Oracle datavarehus
- En introduksjon til datautvinning ("data mining")

### 1.7.3. Databaser og Web

- Databaser og Web 1.0
- Databaseløsninger for å lagre data fra nettsamfunn, f.eks. en Wiki (Web 2.0)
- Kan data fra f.eks. en Wiki kobles/brukes i andre sammenhenger? F.eks. kobles til en "sentral" database
- Hvilke typer kunnskap bør lagres i en database, og hvilke bør evt. ikke lagres (formell vs. uformell kunnskap, "taus" kunnskap ("tacit knowledge"))
- "Flytting" av data mellom gammel og ny teknologi ("emerging technologies")

### 1.7.4. Andre spesielle databaser

- Temporal database