Materiale til brug for Datatekniker og IT Supporter

UV-fag 6238

TEC Ballerup 2015

Indhold:

1 Hvad er en database?

I det følgende skal vi se nærmere på, hvad en database egentlig er for en størrelse.
I princippet er en database **en samling data opbevaret på en organiseret måde**.
Databasen har hjemme i et databasesystem, der kan håndtere adskillige databaser.
Dette kaldes ofte DBMS (Database Management System) – eller en databaseserver om man vil.
Oracle, Microsoft SQL, MySQL og PostgreSQL er eksempler på databaseservere/DBMS'er.
Det er vigtigt at man skelner mellem selve databasestyresystemet og en enkelt database.
En database er faktisk blot en fil (eller flere filer) og DBMS'en styrer så oprettelsen og manipulationen af den.

Lad os nu se på, hvordan data er organiseret i en database.

1.1 Tabeller

Når man gemmer data i en database, så gemmes de i en tabelstruktur, der, sikkert af samme årsag, kaldes en **tabel** (eller *table* hvis man vil tale engelsk).
En tabel er en struktureret model af, hvordan et bestemt element ser ud.
Hvis vi f.eks. ønsker at gemme oplysninger om en elev, så er vi nødt til at afgøre, hvilke oplysninger der er relevante at gemme for en elev. Da tabellen ikke kan ændre sig efter vores forgodtbefindende, er tabellen nødt til at afspejle alle elever.

Lad os se lidt nærmere på det:
Jeg ønsker at gemme følgende oplysninger om elever:

- Fornavn
- Efternavn
- Klasse
- Adresse
- Postnummer
- By

Det er således gældende for alle de elever jeg ønsker at gemme.
Hvis vi tegner tabellen indeholdende 3 elever, så giver det hele måske lidt mere mening:

Elev

Fornavn	Efternavn	Klasse	Adresse	Postnr	By
Søren	Jensen	235	Ved Vejen 6	4000	Roskilde
Edvard	Olsen	235	Gåsevejen 13	2750	Ballerup
Søren	Olsen	235	Borgvej 145	2800	Lyngby

Som vi kan se, så er alle elever strukturelt ens, men deres data er naturligvis forskellige.
Tabellen herover hedder **Elev** – det er **tabelnavnet**.
Et tabelnavn skal være unikt, idet vi bruger det til at skelne mellem de forskellige tabeller i databasen.
Der er forskellige begreber der knytter sig til en tabel – det er vigtigt at få dem på plads, så "vi taler det samme sprog".

1.1.1 Række

Fornavn	Efternavn	Klasse	Adresse	Postnr	By
Søren	Jensen	235	Ved Vejen 6	4000	Roskilde
Edvard	Olsen	235	Gåsevejen 13	2750	Ballerup
Søren	Olsen	235	Borgvej 145	2800	Lyngby

Det markerede udgør en **række**(row).
En række kaldes også en **post**(tuple, record), og svarer her til én elev. Idet tabellen indeholder 3 elever, indeholder den altså 3 rækker/poster.

1.1.2 Kolonne

Fornavn	Efternavn	Klasse	Adresse	Postnr	By
Søren	Jensen	235	Ved Vejen 6	4000	Roskilde
Edvard	Olsen	235	Gåsevejen 13	2750	Ballerup
Søren	Olsen	235	Borgvej 145	2800	Lyngby

Det markerede er en **kolonne**(column).
Kolonnen har et **kolonnenavn** – i dette tilfælde 'Efternavn'.
Et enkelt felt(det lidt mørkere i tabellen) kaldes et **felt**(field) – derfor kaldes kolonnenavn også ofte for **feltnavn**.

2 Datatyper

Ok, så langt så godt.
Vi har altså de mest basale begreber på plads, men der er et stykke vej endnu.
Hvis vi husker på vores tabel fra før, så ser vi at kolonnerne har samme type. Et fornavn er f.eks. altid en samling bogstaver eller karakterer, mens en klasse ser ud til at være tal.
Det er ikke ligegyldigt hvilken type en kolonne har, og en kolonne kan kun indeholde én type.

Som sagt er det ikke ligegyldigt hvilken type vores data har.
Er der tale om tal eller bogstaver og hvor store må disse i givet fald være. Det handler nemlig også om at benytte en datatype, der fylder mindst muligt, så vores database ikke har en masse spildplads.

2.1 Oversigt over typer

Hermed en liste over de mest almindelige typer:

2.1.1 Typer til tekst

Der findes(bl.a.) følgende datatyper til tekst:

Type	Max. længde	Str. bytes
char(*n*)	8000 karakterer	*n* bytes
varchar(*n*)	8000 karakterer	*0-n* bytes
text	2.147.483.647 karakterer	0-2.147.483.647 bytes

char anvendes til tekst med en fast længde, og fylder så meget som man har oprettet den med. F.eks. vil char(20) maksimalt kunne indeholde 20 karakterer og selvom den indeholder mindre, vil den stadig fylde 20 bytes.

varchar anvendes ligeledes til tekst, og som char skal der angives en maksimal længde. Hvis man f.eks. har en varchar(20), kan der skrives tekst på ikke over 20 karakterer, men hvis man skriver mindre, så fylder den kun det antal karakterer man har puttet i den. Ordet 'Søren' fylder derfor 5 bytes hvis typen er varchar(20), mens den som char(20) ville fylde 20 bytes.

text bruges til lang tekst. Der skal ikke angives en maksimal længde, men kan maksimalt fylde 2.147.483.647(~2,15 milliarder) karakterer. Den fylder så mange bytes, som der er karakterer i den.

Som man kan se af det ovenstående, er det ikke lige meget hvilken type man vælger til kolonnen fornavn. Hvis man vælger char(20) vil vores 3 elevers fornavne fylde 60 bytes. Hvis vi derimod havde valgt typen varchar(20) ville de blot fylde 16 bytes – og det er jo en stor forskel selv for så lille en mængde data.
Med mindre der er et fast antal karakterer for hver post, bør du altid vælge varchar.

2.1.2 Typer til heltal

Et heltal er et tal uden decimaler.
Både negative og positive hele tal er heltal.
Et heltal kaldes på computer'sk for **integer**.

Herunder en række af de mest almindelige typer til at håndtere denne type tal:

Type	Min og max. størrelse	Str. bytes
int	-2.147.483.648 og 2.147.483.647	4 bytes
smallint	-32.768 og 32.767	2 bytes
tinyint	0 og 255	1 byte
bigint	-9.223.372.036.854.775.808 og 9.223.372.036.854.775.807	8 bytes

Igen ser vi, at der er stor forskel på, hvilke typer vi vælger.
Til vores klasse vil det være rigeligt med en smallint – ligeså til postnr.
Skulle vi derimod have et telefonnummer med, ja så blev den for lille, og vi måtte vælge en int.
Spørgsmålet er så, om det overhovedet er godt at anvende en heltalstype til et telefonnummer, men det kan vi vende tilbage til senere.

2.1.3 Typer til decimaltal

Vi kan naturligvis ikke nøjes med hele tal, så derfor findes der også typer til decimaltal.
På computer'sk benævnes disse som **float**.
Databasen har dog forskellige typer til at holde decimaltal, og forskellen handler om hvor præcist man vil have sit tal.
Lad os se på dem:

Type	Min og max. størrelse	Str. bytes
decimal(p, s)	$-10^{38}+1$ og $10^{38}-1$ (afhænger af p)	► p=1-9 : 5 bytes ► p=10-19 : 9 bytes ► p=20-28 : 13 bytes ► p=29-38 : 17 bytes
float(*n*)	- 1,79E + 308 og 1,79E + 308 ($-1{,}79 * 10^{308}$ og $1{,}79 * 10^{308}$)	► n=1-24 : 4 bytes ► n=25-53 : 8 bytes

Hvis vi gerne vil angive et antal decimaler som der maksimalt må gemmes med – f.eks. prisen på en vare (kr. 27,95), så kunne vi passende vælge typen decimal(6, 2), hvilket vil sige, at der kan være 6 tal og 2 af dem er decimaler. Det højeste tal vi således kan gemme er 9999,99.

Nå vi skal skrive decimaltal ind i en tabel, så skal man brug punktum som adskillelse mellem heltalsdelen og decimaldelen. Vores 27 kroner og 95 øre fra før, skal skrives 27.95 .
Når vi kommer til SQL, vil det give god mening.

2.1.4 Typer til dato og tid

Vi skal også kunne gemme datoer og tid. Man kunne jo anvende en teksttype til det, men så bliver det svært at sortere og sammenligne. Derfor er der særlige typer til det:

Type	Min og max. størrelse	Str. bytes
datetime	1. januar 1753 og 31. december 9999 - præcision: 3.33 millisekunder	8 bytes (2 * 4 bytes)
smalldatetime	1. januar 1900 og 6. juni 2079 - præcision: 1 minut	4 bytes (2 * 2 bytes)

datetime og **smalldatetime** forventer datoer i formatet mm/dd/åååå tt:nn:ss.zzz eller åååå-mm-dd tt:nn:ss.zzz (m: måned, d: dag, å: år, t: timer, n: minutter, s: sekunder, z: millisekunder).
smalldatetime dog uden millisekunder – der rundes op el. ned til nærmeste minut.

Hvilken af de to typer man vælger, afhænger naturligvis af hvad man skal bruge sin tidsangivelse til. Skal man bare bruge en dato uden præcis tidsangivelse og inden for det understøttede interval, så skal man selvfølgelig vælge den mindste af de to typer.

3 Primærnøgler og fremmednøgler

For at kunne kende de forskellige poster fra hinanden, så skal hver post have noget, der er unikt for den.
Vi har også behov for at kunne knytte tabeller til hinanden – skabe relationer imellem dem.
Til at håndtere disse to behov, har vi **primærnøgler**(primary keys) og **fremmednøgler**(foreign keys).

Lad os se nærmere på primærnøglen først:

3.1 Primærnøgle

Hvis vi kigger på vores tabel fra før – herunder – så ser man, at der ikke er noget der er helt unikt for hver enkelt post.

Elev

Fornavn	Efternavn	Klasse	Adresse	Postnr	By
Søren	Jensen	235	Ved Vejen 6	4000	Roskilde
Edvard	Olsen	235	Gåsevejen 13	2750	Ballerup
Søren	Olsen	235	Borgvej 145	2800	Lyngby

Der er to der hedder Søren til fornavn, to der hedder Olsen til efternavn og alle går i klasse 235. Ingen af disse tre kolonner er altså unikke for en enkelt elev.
Hvad så med adressen? Jo den er rigtignok unik i denne lille tabel, men kan vi være sikre på, at den til enhver tid vil være unik? Kunne Søren Olsen ikke have en lillebror som på et tidspunkt også kommer til at går på skolen? Jo det kunne man forstille sig, og alene derfor, er adresse uegnet som primærnøgle.
Postnummer og by er ligeledes ubrugelige – det er jo ikke usædvanligt, at der er flere der kommer fra samme by.
Vi har nu undersøgt, om der er en **naturlig primærnøgle**. Vi kunne konstatere at det er der ikke.
Hvad gør vi så?
Vi tildeler bare hver elev et unikt nummer – se her:

Elev

ElevId	Fornavn	Efternavn	Klasse	Adresse	Postnr	By
1	Søren	Jensen	235	Ved Vejen 6	4000	Roskilde
2	Edvard	Olsen	235	Gåsevejen 13	2750	Ballerup
3	Søren	Olsen	235	Borgvej 145	2800	Lyngby

Hver elev er nu identificeret med et nummer. Umiddelbart skal vi ikke bruge nummeret til så meget, men hvad nu hvis jeg ville rette i en bestemt elevs data, så er jeg jo nødt til at være helt sikker på, at det er den korrekte elev jeg retter – ellers kan det går grueligt galt. Hvem ville bryde sig om at få en 13-tal rettet til et 5-tal, fordi man havde en navnebror der havde glemt at læse?
Læg mærke til, at primærnøglen i tabellen er understreget _____ - sådan angiver man netop en primærnøgle. Nogle foretrækker at skrive (PK) bagefter – og det er også fint nok.

Da det ikke har nogen betydning for vores beskrivelse af en elev, hvilket id han har, vil det være passende at lade databasen håndtere tildelingen af id'er. Det er muligt, og vi vender tilbage til det senere i SQL-delen.

Hvis vi f.eks. have haft et cpr-nr. med for hver elev, så havde det jo været en fin naturlig primærnøgle for hver elev, idet der jo ikke er to mennesker der har det samme cpr-nr.. Det er alligevel en dårlig idé, da et cpr-nr. må regnes for personfølsomme oplysninger, og det er derfor ikke hensigtsmæssigt at benytte det.

I langt de fleste tilfælde bør man anvende en primærnøgle, der ikke har noget som helst med noget at gøre i stedet for at bryde sin hjerne med at finde en naturlig en af slagsen.

Alle tabeller SKAL have en primærnøgle.

3.2 Fremmednøgler

I enhver tabel vil man prøve at undgå gentagelse af data – kaldet **redundans**.
Derfor deler man data imellem flere tabeller. For at knytte data sammen refererer tabellerne til hinanden v.h.a. fremmednøgler.
Man kunne måske synes, at det er meget at skulle lave en helt ny tabel bare for at undgå at data bliver gentaget, men det viser sig i at være meget mere pladsbesparende og i det hele taget smartere.

Se på vores gammelkendte tabel – nu med lidt flere data og primærnøgle:

Elev

ElevId	Fornavn	Efternavn	Klasse	Adresse	Postnr	By
1	Søren	Jensen	235	Ved Vejen 6	4000	Roskilde
2	Edvard	Olsen	235	Gåsevejen 13	2750	Ballerup
3	Søren	Olsen	235	Borgvej 145	2800	Lyngby
4	Søren	Jensen	235	Vejvejen 543	2000	Frederiksberg
5	Eddy	Jørgensen	235	Gnustræde 50	2750	Ballerup
6	Arne	Poulsen	235	Abegade 15	2000	Frederiksberg

Vi ser her en masse redundant data – og forestiller vi os en meget større samling data, så er det ikke svært at forestille sig, hvor mange gentagelser vi kan få.
Det er pladskrævende. 'Frederiksberg' fylder 13 karakterer – 13 bytes hver gang. Vi bruger altså 26 bytes alene på 'Frederiksberg'.
Hvis vi laver en tabel for sig til postnumre og bynavne, så ser det således ud:

Elev

ElevId	Fornavn	Efternavn	Klasse	Adresse
1	Søren	Jensen	235	Ved Vejen 6
2	Edvard	Olsen	235	Gåsevejen 13
3	Søren	Olsen	235	Borgvej 145
4	Søren	Jensen	235	Vejvejen 543
5	Eddy	Jørgensen	235	Gnustræde 50
6	Arne	Poulsen	235	Abegade 15

PostnrBy

Postnr	Bynavn
4000	Roskilde
2750	Ballerup

2800	Lyngby
2000	Frederiksberg

Det eneste problem vi nu har, er at vi ikke kan se hvor den enkelte elev bor.
Det klarer vi med fremmednøglen:

Elev

ElevId	Fornavn	Efternavn	Klasse	Adresse	Pnummer
1	Søren	Jensen	235	Ved Vejen 6	4000
2	Edvard	Olsen	235	Gåsevejen 13	2750
3	Søren	Olsen	235	Borgvej 145	2800
4	Søren	Jensen	235	Vejvejen 543	2000
5	Eddy	Jørgensen	235	Gnustræde 50	2750
6	Arne	Poulsen	235	Abegade 15	2000

Pnummer er fremmednøgle og refererer til tabellen PostnrBy, hvor Postnr er blevet primærnøgle.
Fremmednøglen Pnummer som det fremgår af tabellen herover understreget med _ _ _ _ _ _, det angiver at det er en fremmednøgle. Nogle skriver (FK) – det er også helt ok.
I tabellen PostnrBy skal man lægge mærke til, at jeg har ændret kolonnenavnet 'By' til 'Bynavn'. Det skyldes at ordet 'by' er et reserveret ord(blandt andre), og det vil derfor blive tolket forkert.

En fremmednøgle har altid samme type som den primærnøgle den henviser til!

Lad os se på hvad vi har sparet.
Vi har fået en kolonne mere. Hvis den er af typen smallint, fylder den samlet 12 bytes(6 * 2 bytes).
Vi har sparet 1 'Frederiksberg' og 1 'Ballerup' – tilsammen 21 bytes(13 bytes + 8 bytes).
I princippet har vi altså sparet 9 bytes.

Eksemplet er tænkt, for dertil kommer indeksering af fremmednøglen og selve tabellen, der også fylder noget, men tilføjer vi f.eks. 2000 rækker, så er besparelsen betragtelig.

Faktisk burde vi også skille klassen ud, men det gemmer vi til afsnit 4.

3.3 Normalisering

Det vi i virkeligheden har gang i her, kaldes **normalisering.**
Der findes 1., 2. og 3. normalform.
Det er en lidt omstændelig metode til at sørge for at data ligger i databasen på den mest effektive måde.

Jeg vil henvise til *Note vedr. Normalisering* for yderlige om denne fremgangsmåde.
Personligt foretrækker jeg en mere intuitiv fremgangsmåde ved at lave et E/R-diagram – det kræver måske nok lidt rutine, men er efter min ringe mening meget mere sammenhængende med f.eks. måden man tænker objektorienteret programmering på.

Datakompleksiteten skal ikke være ret stor, før normalisering synes meget omstændelig.

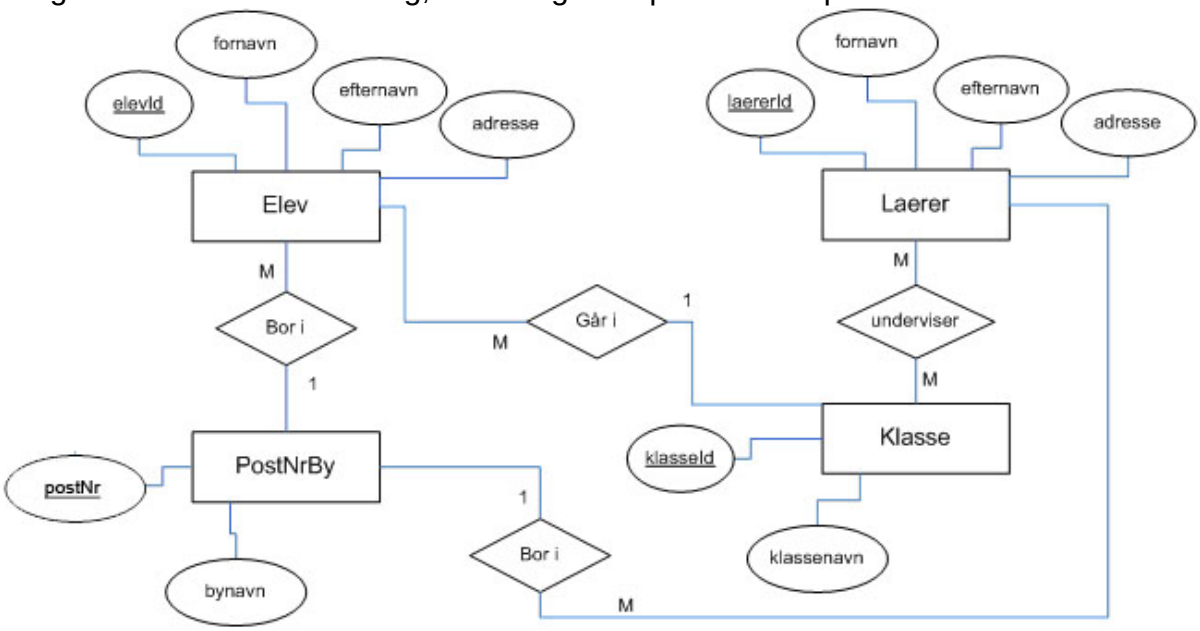
4 Databasemodellering

Vi har allerede været inde på det i foregående afsnit.
En databases opbygning kan og må ikke være tilfældig. Det er vigtigt at man ved præcis hvordan den skal se ud inden man laver den.
Hvis man skulle bygge et hus, så ville man vel også tegne en tegning af det først – måske endda bygge en model.

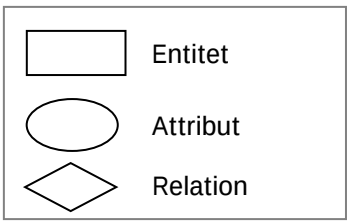
Et glimrende redskab hertil er E/R-diagrammet. Det ser vi nærmere på nu.

4.1 E/R-diagram

E/R betyder **E**ntitet/**R**elation.
Det giver vist lidt mere mening, hvis vi lige ser på et eksempel:



Og hvad skal det så betyde.
Lad os først få de forskellige elementer på plads:



Entitet betegner det element vi ønsker at beskrive. Svarer til en tabel i databasen.

Attribut betegner egenskaberne ved elementet. Svarer til en kolonne i tabellen.

Relation viser at der er sammenhæng(relation) mellem to tabeller. Teksten inde i relationen beskriver denne – og kan sagtens udelades uden at diagrammet mister noget.

1 og **M** der er placeret ved relationerne betyder henholdsvis **én** og **mange**.

Dette beskriver hvordan de to tabeller er afhængige af hinanden. En sådan betegnelse kaldes **kardinalitet**.
Entiteten *Elev* og entiteten *Klasse* har en relation der kan beskrives som 1→M (læses: én til mange), hvilket vil sige at:
Én elev går i **én** klasse.
Én klasse kan have **mange** elever.
Mellem Lærer og Klasse er der en M→M - relation (læses: mange til mange), hvilket vil sige at:
Én lærer underviser **mange** klasser.
Én klasse kan have **mange** lærere.

Hvis man støder på 1→1-relationer, så er det som regel tegn på, at der er noget galt. Den ene entitet kunne måske så passende holde informationerne fra den anden – men det er altså ikke *altid* tilfældet.

4.2 Fra diagram til tabeller i databasen

Et diagram er jo ikke meget værd, hvis det ikke kan omsættes til noget vi kan bruge til noget. Lad os derfor omforme vores diagram til tabeller i databasen.

Først skal i lige have et par regler på plads:

- ✔ **1→M-relationer giver ALTID en fremmednøgle på mange-siden.**
- ✔ **M→M-relationer giver ALTID en ny tabel med to fremmednøgler.**

Det vil altså sige, at vi skal have to fremmednøgler på tabellen *Elev*, en fremmednøgle på *Laerer* og lave en ny tabel.
Dette fordi der på *Elev* er "mange" i forhold til to entiteter – nemlig *Klasse* og *PostNrBy*, fordi *Laerer* er "mange" i forhold til *PostNrBy* og fordi der er en mange til mange relation mellem *Laerer* og *Klasse*.

Dette giver os i alt 5 tabeller, der ser således ud:

Elev

elevId	**fornavn**	**efternavn**	**adresse**	**postNr**	**klasse**
int	varchar(30)	varchar(30)	varchar(50)	smallint	smallint

Laerer

laererId	**fornavn**	**efternavn**	**adresse**	**postNr**
int	varchar(30)	varchar(30)	varchar(50)	smallint

Klasse

klasseId	**klassenavn**
int	varchar(10)

PostNrBy

postnr	**bynavn**
smallint	varchar(30)

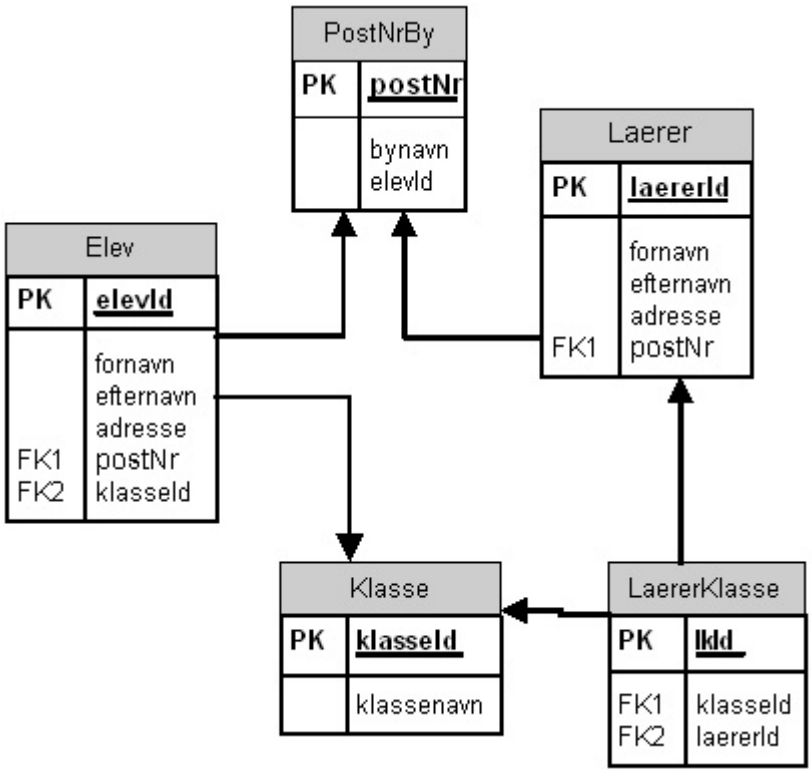
LaererKlasse

lkId	**laererId**	**klasseId**

int	int	int

Der er valgt passende typer, om end man kunne diskutere om man kunne vælge smallint til nogle af id'erne, men over tid(f.eks. 10 år), er det svært at vide, hvor mange elever man f.eks. vil nå op på.

For at tydeliggøre relationerne mellem tabeller, kan man lave en anden type relationsdiagram:



Den største forskel mellem de to diagramtyper er vel, at den første bedst egner sig til den fase, hvor man skal finde tabeller, attributter og kardinalitet(hvordan de relaterer (1 og M) – det man kunne kalde udviklingsfasen. Den er også mere konceptuel end den anden, idet man ikke har splitter M→M-relationerne ud, og ej heller angiver fremmednøgler.

Den anden type(herover) egner sig nok bedre som dokumentation for en færdig database, altså "hvordan kom den til at se ud".
Her tegnes alle tabeller og samtlige nøgler angives.

Nu kan vi endeligt oprette vores tabeller i databasen.
Til det skal vi benytte SQL, og det er netop hvad næste afsnit handler om.

5 SQL

For at 'snakke' med vores database, bruger man sproget **SQL** – **S**tructured **Q**uery **L**anguage.
Med SQL kan man **hente** data, **indsætte** data, **rette** data og **slette** data.
Derudover kan man håndtere(oprette, rette, slette) tabeller, håndtere brugere og meget meget mere.

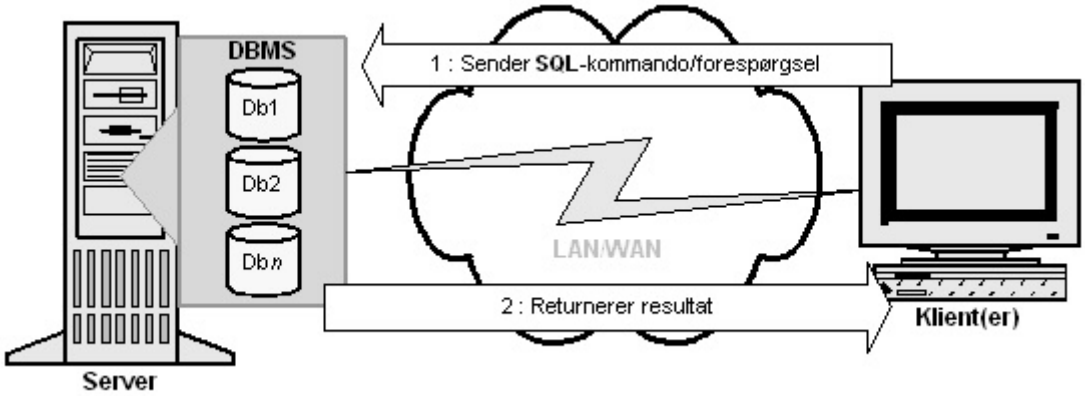
SQL er et dejligt sprog på den måde, at man skriver det man ønsker udført med "almindelige ord". Hvis vi vil vælge hedder det *select*, slette hedder *delete*, opdatere hedder *update* – og sådan kunne man blive ved.

SQL er IKKE et programmeringssprog, men et forespørgselssprog(deraf 'query').

Selvom SQL altså er syntaks for at trække data fra databasen, kan man manipulere og definere tabeller og databaser – ja sågar administrere adgangen til disse.

SQL har altså tre dele:
- **Query Language – forespørgsler – returnere data.**
- **Data Manipulation Language(DML – manipulere data (insert, update, delete)).**
- **Data Definition Language(DDL – definere data(create table, alter table, drop table m.m.)).**

Kommunikationen med et databasesystem(DBMS) kan illustreres således:



Man kan selvfølgelig køre et databasesystem lokalt, men det er ikke systemerne er beregnet til at dele data – helst på en dedikeret server.

Basal SQL er syntaksmæssig den samme på tværs af databasesystemer.
De enkelte databasesystemer har dog alle (mange) forskelle, og det skal man være meget opmærksom på!
Dette på trods af, at man kalder SQL en standard.

Heldigvis har alle de databasesystemer jeg har stødt på glimrende online dokumentation (el. reference), og med den 'i hånden', når man langt – også selvom man er vant til et specifikt system.

Microsoft anvender *Transact SQL* til deres databasesystemer.
Deres online dokumentation findes på
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_tsqlcon_6lyk.asp

Transact SQL er grundlaget for dette materiale.

Vi vil fokusere på datadelen, der er den, som man vil bruge allermest. Den anden funktionalitet kan i høj grad varetages af interfaces.

Det giver dog en meget god grundlæggende forståelse at oprette sine tabeller selv i SQL – så det vil vi gøre…

5.1 Opret en tabel

Lad os oprette en simpel tabel med SQL.
Vi forudsætter at vi har en database til rådighed, er logget på og har rettigheder til at oprette en tabel. Som sagt vil dette materiale ikke fokusere på adminstration.

Vi tager udgangspunkt i denne simple tabel:

Person

personId	**fornavn**	**efternavn**
int	varchar(20)	varchar(30)

Vi fyrer nu følgende SQL-kommando af sted:

```
CREATE TABLE Person(
                personId INT PRIMARY KEY IDENTITY(1,1),
                fornavn VARCHAR(20) NOT NULL,
                efternavn VARCHAR(30) NOT NULL
                )
```

Og hvad betyder det så?
Jo lad os tage det i små bidder:

1. **CREATE TABLE** betyder at vi vil lave en ny tabel.
2. **Person(***kolonner***)** er navnet på tabellen – i den efterfølgende parentes står, hvad tabellen skal indeholde – i det her tilfælde tre kolonner:
   - 2.1. **personId INT PRIMARY KEY IDENTITY(1,1)** er den første kolonne og den er defineret således:
     - 2.1.1. **personId** er navnet på kolonnen
     - 2.1.2. **INT** er typen for data i den pågældende kolonne – i dette tilfælde et heltal
     - 2.1.3. **PRIMARY KEY** angiver at kolonne indeholder primærnøgle for hver række/post
     - 2.1.4. **IDENTITY(1,1)** fortæller at værdien i denne kolonne tælles automatisk op 1 hver gang der indsættes en ny række i tabellen. Vi skal altså ikke selv holde styr på at tildele et unikt id.
       Det første 1-tal fortæller at første værdi er 1 – det andet 1-tal, at der tælles op med 1 hver gang. Vi kunne altså starte i 1342 og tælle op med 17 hver gang, hvis vi havde lyst.
       Hvis man sletter en række, så erstattes værdien **ikke**. Det vil sige at hvis vi har id'erne 1, 2, 3, 4, 5, 6 og 7 og vi sletter rækken med id 4, så bliver det næste indsatte id **ikke** 4, men der tælles bare videre og derfor bliver næste id lig med 8.
       IDENTITY er specifik for Transact SQL – andre databaser anden syntaks.
   - 2.2. **fornavn VARCHAR(20) NOT NULL** er den anden kolonne og den er defineret således:

2.2.1. **fornavn** er navnet på kolonnen
2.2.2. **VARCHAR(20)** er typen for kolonnens data – her er der tale om karakterer(bogstaver) – max. længde på 20.
2.2.3. **NOT NULL** fortæller at der ikke må være tomme(NULL) felter i kolonnen – man skal altså angive et fornavn når man opretter en person/række. Alternativt kunne man skrive **NULL**, hvis det var ok at feltet stod tomt – det er default, men mange angiver det alligevel af hensyn til dokumentationen.
Et felt der er primærnøgle er pr. definition NOT NULL.

2.3. **efternavn VARCHAR(30) NOT NULL** er den tredje kolonne – den vil jeg ikke beskrive nærmere – se pkt. 2.2.

Bid endeligt mærke i, at hver kolonnedefinition er adskilt af et komma(,).

Har man lavet en fejl i oprettelsen af sin tabel – f.eks. angivet en forkert type – så kan man ændre tabellen med **ALTER TABLE** og **MODIFY TABLE** - dette naturligvis også, hvis man ad åre får behov for en ekstra kolonne eller andet.

Som udgangspunkt er det en dårlig idé at rette i tabeller i oprettelsesfasen. Det er meget nemmere at gemme sin oprettelse i et script – rette scriptet til – slette alle tabeller, og oprette det hele på ny. Det lyder måske lidt besværligt, men det er og bliver den sikreste måde at få en korrekt tabelstruktur.
Man sletter nemt en tabel således:

```
DROP TABLE tabelnavn
```

Alt data tabellen måtte indeholde forsvinder sammen med den.

5.2 Oprettelse af flere tabeller med relationer

Som vi tidligere har kigget på, så har vi oftest behov for, at data afhænger af hinanden på tværs af tabeller. Vi lavede denne relation med en fremmednøgle.
Lad os se det i praksis:
Vi vil lave følgende tabeller

Ansat

ansatId	**fornavn**	**efternavn**	**afdId**
int	varchar(20)	varchar(30)	int

Afdeling

afdelingsId	**afdelingsNavn**
int	varchar(20)

Ansat refererer til afdeling, hvilket altså vil sige, at én ansat arbejder i én afdeling og at én afdeling kan have mange ansatte. En 1→M-relation med M på ansat-siden.

I SQL ser det således ud:

```
CREATE TABLE Afdeling(
                    afdelingsId INT PRIMARY KEY IDENTITY(1,1),
                    afdelingsNavn VARCHAR(20) NOT NULL UNIQUE
                    );
```

```
CREATE TABLE Ansat(
                        ansatId INT PRIMARY KEY IDENTITY(1,1),
                        fornavn VARCHAR(20) NOT NULL,
                        efternavn VARCHAR(30) NOT NULL,
                        afdId INT NOT NULL,
                        FOREIGN KEY(afdId) REFERENCES Afdeling(afdelingsId)
                        )
```

I den første tabeloprettelse, er der sådan set ikke noget nyt under solen. Vi støder dog på et enkelt nyt ord – nemlig **UNIQUE**. Dette betyder, at ikke kan være to felter i kolonnen med samme værdi – ethvert afdelingsnavn skal være unikt.

I tabeloprettelsen for ansat, sker der først noget nyt med:
FOREIGN KEY(afdId) REFERENCES Afdeling(afdelingsId)
Hvis man forstår engelsk, så siger det næsten sig selv, men lad os prøve at skrive linien på jævnt dansk:
afdId er fremmednøgle. Den refererer til tabellen Afdelings primærnøgle afdelingsId.

Man bør endvidere ligge mærke til semikolon imellem de to CREATE TABLE. Dette for at skille hver SQL-sætning fra hinanden.
På den måde, kan man skrive mange SQL-sætninger på en gang – de udføres én ad gangen i rækkefølge.

Slutteligt er det vigtigt at skrive sig rækkefølgen for oprettelse bag øret. Havde vi ikke oprettet tabellen afdeling først, så kunne vi ikke referere til den fra tabellen Ansat – vi kunne altså ikke have erklæret vores fremmednøgle.
Havde vi forsøgt, ville databasen have givet en fejl.

5.3 SELECT – hent data

Nå, nu kan vi altså oprette vores tabeller – lad os lige fylde noget data i dem.

Ansat

ansatId	fornavn	efternavn	afdId
1	Morten	Jensen	2
2	Ib	Nielsen	1
3	Jørgen	Petersen	1
4	Herbert	Mogensen	3
5	Søren	Rasmussen	2
6	Morten	Petersen	3
7	Viggo	Nielsen	3
8	Rolf	Madsen	1
9	Birger	Alfonso	2
10	Lotte	Åbo	2

Afdeling

afdelingsId	afdelingsNavn
1	Sjælland
2	Fyn
3	Jylland

Ja, ok jeg snød lidt og hældte bare noget i dem – vi ser senere på hvordan vi gør det i SQL.

Lad os prøve at vælge alle ansattes fornavne:

```
SELECT fornavn FROM Ansat
```

Dejligt enkelt ikke? ”Vælg fornavn fra Ansat”, ville den oversættelsen være, og det var jo lige det vi ville.
Vi får følgende resultat – også kaldet resultatsæt(resultset) - tilbage:

Fornavn
Morten
Ib
Jørgen
Herbert
Søren
Morten
Viggo
Rolf
Birger
Lotte

Vi får altså også svar tilbage fra databasen i en tabelform – i hver fald opfatter vi det således.

Hvis vi vil vælge både efternavn og fornavn(og i den rækkefølge) sender vi denne SQL-forspørgsel:

```
SELECT efternavn, fornavn FROM Ansat
```

Det giver dette resultat:

efternavn	fornavn
Jensen	Morten
Nielsen	Ib
Petersen	Jørgen
Mogensen	Herbert
Rasmussen	Søren
Petersen	Morten
Nielsen	Viggo
Madsen	Rolf
Alfonso	Birger
Åbo	Lotte

Og hvis vi bare vil vælge alt fra tabellen kunne vi sende den her af sted:

```
SELECT * FROM Ansat
```

- og få flg. resultat:

ansatId	fornavn	efternavn	afdId
1	Morten	Jensen	2
2	Ib	Nielsen	1
3	Jørgen	Petersen	1
4	Herbert	Mogensen	3
5	Søren	Rasmussen	2
6	Morten	Petersen	3
7	Viggo	Nielsen	3
8	Rolf	Madsen	1
9	Birger	Alfonso	2
10	Lotte	Åbo	2

Altså en kopi af hele vores tabel.
* betyder altså ”alt”.

Ville vi vælge alle efternavne, men undgå gentagelser, kunne vi spørge således:

```
SELECT DISTINCT efternavn FROM Ansat
```

Med følgende resultat:

efternavn
Alfonso
Jensen
Madsen
Mogensen
Nielsen
Petersen
Rasmussen
Åbo

En Nielsen og en Petersen er altså ikke med – vi ville jo undgå dubletter.
Desuden er data sorteret, men det ligger egentlig ikke i DISTINCT, men min forespørgsel returnerede dette. Jeg har ikke fundet dokumentation der kan understøtte denne egenskab.

5.4 WHERE – betingelser

Nu er det jo ikke synderligt interessant altid at vælge alle.
For at få noget at arbejde med, tilføjer vi lige elegant en alder til vores ansatte – og vupti vores ansat-tabel ser nu sådan her ud med indhold:

ansatId	fornavn	efternavn	afdId	alder
1	Morten	Jensen	2	34
2	Ib	Nielsen	1	65
3	Jørgen	Petersen	1	23
4	Herbert	Mogensen	3	27
5	Søren	Rasmussen	2	32
6	Morten	Petersen	3	29
7	Viggo	Nielsen	3	19
8	Rolf	Madsen	1	18
9	Birger	Alfonso	2	36
10	Lotte	Åbo	2	32

Måske kunne det være interessant at vælge alle de ansatte, der er ældre end 25 år.

```
SELECT * FROM Ansat WHERE alder > 23
```

Vi ville få det her tilbage:

ansatId	fornavn	efternavn	afdId	alder
1	Morten	Jensen	2	34
2	Ib	Nielsen	1	65
4	Herbert	Mogensen	3	27
5	Søren	Rasmussen	2	32
6	Morten	Petersen	3	29
9	Birger	Alfonso	2	36
10	Lotte	Åbo	2	32

Det vil altså sige, at vi kan betinge vores forespørgsel – der er noget der skal være opfyldt.

Vi har en række matematiske operatorer, som vi kan benytte:
> større end
< mindre end
<> forskellig fra

!= forskellig fra(samme som <>)
<= mindre end el. lig med
>= større end el. lig med
= lig med

Vi kan også vælge at opfylde flere betingelser:

```
SELECT fornavn, efternavn FROM Ansat WHERE alder < 30 AND afdId = 1
```

fornavn	efternavn
Jørgen	Petersen
Rolf	Madsen

6 Indeks