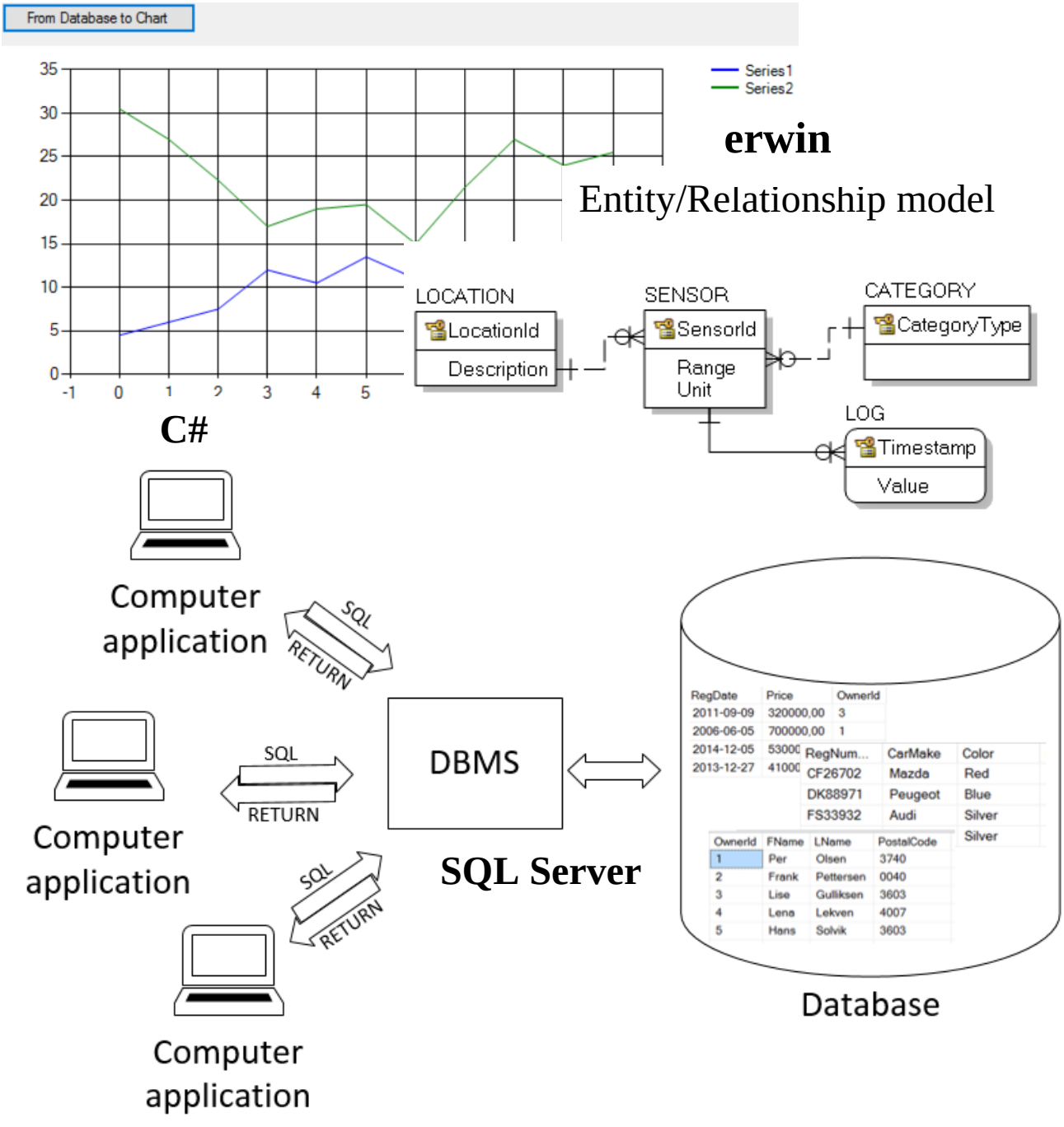
Olav Dæhli

# DATABASEUTVIKLING

Grunnleggende databaseteori, SQL-spørringer, databasemodellering og utvikling av applikasjoner med kobling mellom SQL Server og C#.



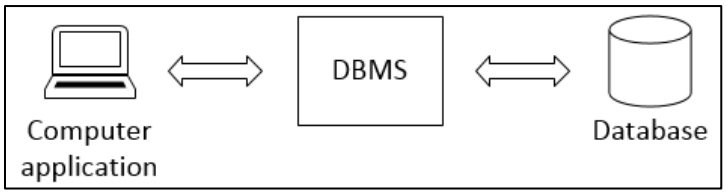
## Innholdsfortegnelse

# 1. Introduksjon

I de fleste applikasjoner er det behov for å lagre data. Ofte kan datamengdene bli store, og det vil da være behov for å kunne lagre og gjenfinne data på en strukturert og hurtig måte. Til slike formål lagres gjerne dataene i det som kalles **databaser** fysisk på et lagringsmedium.

Lagring i databaser, fremfor i filbaserte systemer (jf. metoder i programmeringskompendiet), gir en del fordeler. Det blir lettere å beskytte dataene, dataene blir lettere å holde vedlikeholde, det er lettere å unngå feil grunnet redundans, og dataene kan lettere gjøres tilgjengelig for ulike applikasjoner.

Egne applikasjoner kalt «databasehåndteringssystemer» (**DBHS**), på engelsk «Database Management Systems» (**DBMS**), benyttes til å sørge for effektiv lagring og gjenfinning av data. DBMS blir et mellomledd mellom hovedapplikasjonen og dataene på lagringsmediet, som illustrert i Figur 1-1.



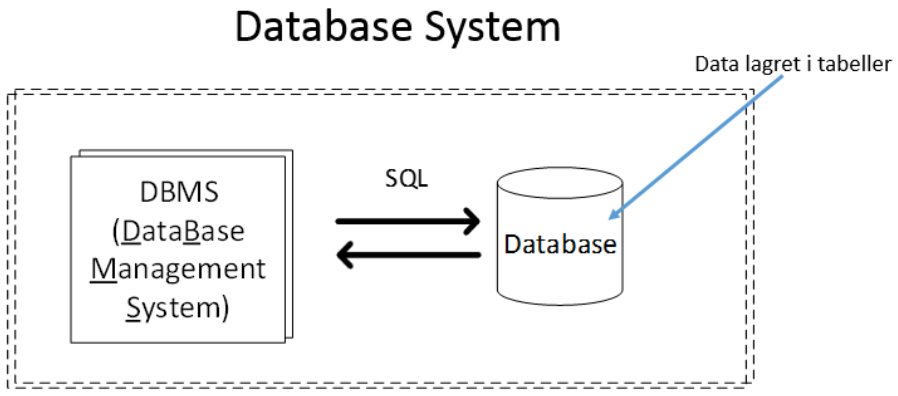

**Figur 1-1: Database Management System (DBMS).**

Det finnes ulike typer databaser. I dette kompendiet benyttes relasjonsdatabaser, som er den typen som er i mest utstrakt bruk. Grunnen til at det kalles relasjonsdatabaser er at prinsippene er basert på en matematisk modell kalt relasjonsmodellen, utviklet av Edgar Frank Codd på 60-70-tallet mens han jobbet for IBM, beskrevet i en artikkel i 1970 [1]. Edgar F. Codd [2] videreutviklet senere modellen i samarbeid med Raymond Boyce [3]

I 1974 beskrev Chamberlin [4] og Raymond Boyce språket SEQUEL for spørringer mot relasjonsdatabaser i en artikkel [5] . Senere endret språket av rettighetsårsaker navn til SQL, som siden 1986 er en definert ISO-standard [6]. Dette kompendiet/kurset baserer seg på en praktisk tilnærming til relasjonsdatabaser, med bruk av SQL for å lage spørringer mot databaser.

Til håndtering av relasjonsdatabaser finnes det en rekke **DBMS**. Blant de mest kjente er **Oracle**, **DB2**, **MySql**, **Access** og **SQL Server**, men det finnes mange flere. I dette kurset benyttes **Microsoft SQL Server** som **DBMS**. Til **modellering** av databaser benyttes en applikasjon kalt **erwin**.

Utveksling av data mellom DBMS og databasen gjøres normalt med et standardisert språk kalt SQL (Standard Query Language). DBMS og databasen utgjør et **databasesystem**, som illustrert i Figur 1-2.




**Figur 1-2: Illustrasjon av et databasesystem med DBMS, database og datahåndtering med SQL.**

**I dette kompendiet ses det på følgende:**

- Grunnleggende teori knyttet til databaser.
- Grunnleggende **SQL**. Dette er et spørrespråk inkludert i DBMS, som gir mulighet til opprette og slette databaser, endre databasestrukturene, lagre og hente ut data fra databasen på en strukturert måte, samt definere tilgangsrettigheter til dataene.

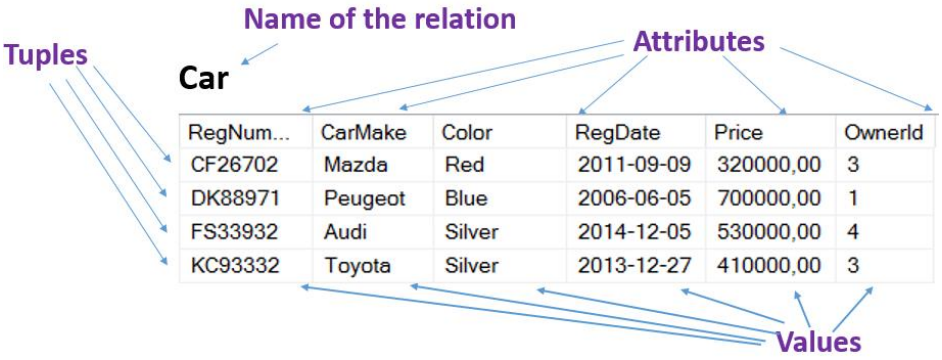
- **Normalisering**. En systematisk fremgangsmåte for å sikre en god datastruktur.
- **Modellering** av databaser. Dette handler om design av databaser på et høyere abstraksjonsnivå enn SQL. Med egne modelleringsverktøy kan databasene fremstilles på blokkskjematisk nivå. Ut fra disse kan så modelleringsverktøy automatisk generere databasestrukturen.
- En databaseapplikasjon inngår gjerne som en del av et større datasystem, der et programmeringsspråk benyttes til andre deler av systemet (som f.eks. brukergrensesnittet). Da det i dette kurset har inngått opplæring i C#, ses det også litt på hvordan C# og SQL Server kan benyttes i kombinasjon.

# 2. Tabeller

I relasjonsdatabaser lagre dataene i 2-dimensjonale tabeller. Matematisk (i relasjonsmodellen) kan en tabell betraktes som en «**relasjon**», hvorav årsaken til navnet relasjonsdatabaser.

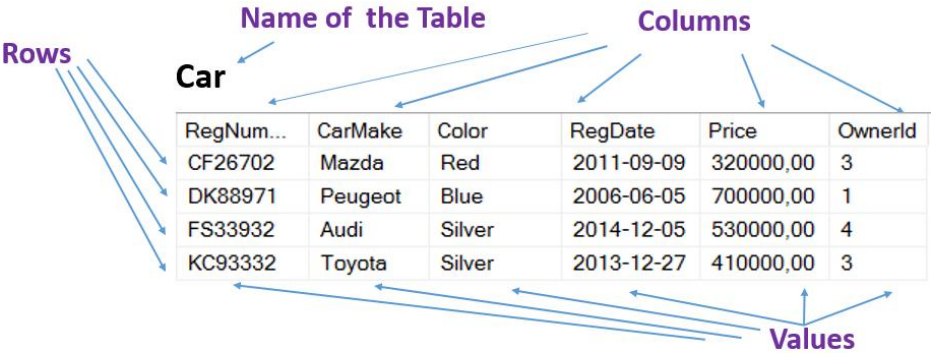
## 2.1. Grunnleggende om tabeller

I Figur 2-1 er det vist et eksempel på en tabell (relasjon), der det er påført navn på tabellen (relasjonen). I tillegg vises det hvordan hver rad i henhold til relasjonsmodellen kalles en «**tuppel**» og hver kolonne et «**attributt**». I hver tuppel lagres det et sett med verdier. Hver tuppel (rad) inneholder her informasjon om én spesifikk bil og dens eier.




**Figur 2-1: Eksempel på en tabell (relasjon).**

I denne litt mer praktiske tilnærming vil relasjonsnavn ofte kalles **tabellnavn**, tupler kalles **rader** og attributter kalles **kolonnenavn**, som vist i Figur 2-2. Det vil i en tabell alltid være et fast antall kolonner, mens antall rader vil variere ut fra behovet. Det kan lagres tusenvis av rader med data i en tabell.




**Figur 2-2: Eksempel på navn ved praktisk bruk av tabeller/relasjoner.**

## 2.2. Logisk skjema og fysisk skjema

Det «logiske nivået» beskriver tabellstrukturen på et nivå som er tilpasset i prinsippet en hvilken som helst relasjonsdatabase. Alle tabeller har nøyaktig én primærnøkkel, som brukes som identifikator for en rad med informasjon. Primærnøkkel forklares nærmere i kapittel 2.4. I tillegg kan en tabell ha én eller

flere fremmednøkler. Fremmednøkler brukes som referanser mellom tabeller når data som hører sammen fordeles på flere tabeller. Fremmednøkler forklares nærmere i kapittel 7.1. En tabell med primærnøkkel, andre attributter, og ev. fremmednøkler, angis ofte med følgende notasjon:

**Tabellnavn (attributt1, attributt2, attributt3*)**

Tabellnavnet angis først. Deretter listes alle attributtnavn opp i en parentes. En primærnøkkel angis med understrekning og en fremmednøkkel angis med en stjerne bak. Består en primær- eller fremmednøkkel av en kombinasjon av attributter, angis disse i en parentes. Her er et eksempel på en tabell med to fremmednøkler, der en av dem består av to attributter og den andre av ett attributt.

**Tabellnavn (attributt1, attributt2, attributt3*, (attributt4, attributt5)*)**

Tabellen som ble vist i Figur 2-2 kan med denne notasjonen angis sånn:

**CAR (RegNumber, CarMake, Color, RegDate, Price, OwnerId*)**

Det er da forutsatt at **OwnerID** er en fremmednøkkel. Ikke alle tabeller har en fremmednøkkel. I dette tilfellet er fremmednøkkelen ment å peke til en annen tabell, der det kan hentes mer informasjon om eieren, eksempelvis navn, e-post, adresse osv. Fremmednøkler behandles grundigere senere.

En database beskrevet på det «**logiske nivået**» kan som nevnt implementeres i en hvilken som helst relasjonsdatabase. Før dette kan gjøres, må det likevel foretas en del tilpasninger til det konkrete systemet den skal implementeres i. Blant annet gjelder dette spesifikasjon av feltenes datatyper, da ulike DBMS kan ha ulik spesifikasjon av datatyper. Dette kalles det «**fysiske nivået**».

Beskrivelse på det «fysiske nivået» er en tilpasning til det konkrete systemet implementasjonen skal skje i, i motsetning til det «logiske nivået». Datatyper, gyldighetskontroll etc. er ting som må spesifiseres for et konkret databasehåndteringssystem.

## 2.3. Datatyper

Alle data i en kolonne må være av samme datatype. Datatypene defineres når tabellene opprettes. Ulike DBMS har støtte for ulike datatyper, så dette må sjekkes for det aktuelle DBMS som benyttes. Datatyper er gjennomgått i programmeringskompendiet.

Selv om det som sagt kan avvike noe fra system til system, er det grunnleggende sett de samme hovedtypene som går igjen, som tekst, heltall, flyttall, boolske verdier osv.

**De vanligste datatypene som benyttes for databaser i dette kurset er:**

- **int**: Heltall (32 bit). Eksempler: 1, 5, 100, 55000 osv.
  Det finnes alternative heltallsdatatyper, som **smallint**, **bigint** m.m., som ikke beskrives her.
- **identity (seed, increment)**: Ikke direkte en datatype, men en autonummerering av heltall, der «seed» angir startnummer og «increment» angir stegene det skal økes med.
  Eksempel: identity (1,1).
- **float**: Flyttall, dvs. tall med desimaler. Eksempel: 1.53421, 2.34 osv.
  (Det finnes alternative flyttallstyper, som **numeric**, **decimal** m.m., som ikke beskrives her) .
- **varchar (x)**: variabel «character», dvs. tekststreng med variabel lengde. Dersom **varchar (25)**, kan det lagres en tekststreng med inntil 25 karakterers lengde, men det avsettes ikke mer plass enn det som reelt benyttes. Navnet "Per" vil eksempelvis ta mindre plass enn "Kristian".
- **char (x)**: Dette er et alternativ til **varchar (x)**. Det avsettes da plass til x antall tekstkarakterer i minnet, uavhengig av hvor lang tekst som lagres. Med **char (25)** vil da navene "Per" og "Kristian" ta like mye plass i minnet. **Varchar** vil kreve litt mer tid å prosessere, men ubetydelig for våre formål. Benytt derfor gjerne **varchar** gjennomgående i dette kurset.

- **date**: Lagring av data på dataformat. Dette gir bl.a. mulighet for å benytte innebygde datofunksjoner på dataene som lagres i kolonnen. Det finnes andre varianter, som f.eks. **datetime**.
- **bit**: Lagring av boolske data (true /false)
- **money**: Valutaformat, ved eksempelvis lagring i en kolonne som **Price** i tabellen **CAR**.

Det ses nærmere på datatyper når det skal lages og brukes tabeller senere i kompendiet.

## 2.4. Primærnøkkel

Hver tabell må en **primærnøkkel** (**Primary Key**), som unikt kan identifisere enhver rad i tabellen. Dette så det vil være mulig å skille radene fra hverandre, da alle radene må være forskjellige fra hverandre. Måten dette gjøres på er å definere én eller en kombinasjon av kolonnenavn (attributter) i en rad som **primærnøkkel**, som unikt benyttes til å skille radene fra hverandre.

Det må derfor gjøres en vurdering av dataenes betydning, for å finne en **primærnøkkel**. Som eksempel betraktes tabellen vist i Figur 2-2. I denne tabellen er det registrert data om biler. Det kan da antas at bilens registreringsnummer er unikt, hvilket vil si at det i to rader aldri vil kunne finne sted registrering av biler med samme registreringsnummer.

Kolonnen **RegNumber** kan derfor velges som primærnøkkel, da den unikt vil kunne identifiser enhver rad i tabellen.

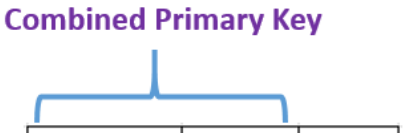
Det er ikke alltid at ***ett*** attributt i tabellen har den egenskapen at den unikt kan identifisere enhver rad. Da må det sjekkes om det finnes kombinasjoner av attributter, som samlet vil ha denne egenskapen.

I Figur 2-3 er det vist en tabell med tre attributter (kolonnenavn). I alle de tre kolonnene ses det at det forkommer **repeterende verdier**. I den første kommer tallet 1 to ganger og tallet 2 tre ganger. I den andre kommer både tallet 1 og tallet 2 to ganger og i den tredje kommer tallet 60 to ganger. Dette betyr at ingen av de tre kolonnene alene har den egenskapen at de unikt kan identifisere enhver rad.

Ved en analyse av dataenes betydning, vil man se at én turnering vil kunne bestå av mange spillere, mens én spiller kan delta i mange turneringer. For hver turnering vil hver enkelt spiller som deltar få registrert en poengsum.

Ovennevnte betyr at kombinasjonen av turnering ***og*** spiller alltid vil være unik. Turnering 1 og spiller 1 vil for eksempel bare kunne forekomme én gang i tabellen. Disse to attributtene i kombinasjon «TOURNAMENT, PLAYER», vil derfor kunne benyttes som en primærnøkkel, som vist i Figur 2-3.

En tabell ***må*** alltid ha en primærnøkkel og den har ***aldri*** flere primærnøkler. Selv om en primærnøkkel eventuelt består av en kombinasjon av flere felt, betyr dette altså ***ikke*** at den har flere primærnøkler.




| TOURNAMENT | PLAYER | POINTS |
|---|---|---|
| 1 | 1 | 60 |
| 1 | 2 | 70 |
| 2 | 1 | 65 |
| 2 | 2 | 80 |
| 2 | 3 | 60 |

**Figur 2-3: Valg av «TOURNAMENT, PLAYER» som primærnøkkel.**

## 2.5. Kolonners og raders plassering/rekkefølge er uvesentlig

Primærnøkkelen gir som vist en unik identifisering av hver enkelt rad i tabellen, om det så måtte være tusenvis av dem. Av denne grunn er det helt uvesentlig hvilken rekkefølge radene med data kommer i. I en database er dataene per definisjon usortert. Det finnes en SQL-kommando som kan sortere dataene som vi måtte ønske, men «rådataene» er altså ikke sortert.

På tilsvarende vis er det uvesentlig i hvilken rekkefølge attributtene (kolonnenavnene) i en tabell plasseres, men det kan i spørringer angis i hvilken rekkefølge man vil ha dem vist i resultatet.

## 2.6. Nullmerker

I tabellene vist til nå, har det vært verdier i alle tabellenes felter for enhver rad. I noen tilfeller hender det at det ikke er noen verdi å registrere. Noen ganger skal det aldri registreres en verdi der, mens andre ganger vil det kanskje først senere bli registrert en verdi der.

I felt der det ikke er registrert noen verdi, angis det såkalte nullmerker (**NULL VALUES**). Dette er ***ikke*** det samme som 0, men bare en indikasjon på at det faktisk ikke finnes noen verdi der. Normalt ønskes det å unngå for mye bruk av nullmerker, men det kan ikke alltid unngås. Datamodellering, som behandles senere, vil medvirke til gode datastrukturer der bl.a. behovet for nullmerker reduseres.

Nullmerker kan blant annet skape problemer ved enkelte beregninger, for eksempel gjennomsnittsberegninger av tall i en kolonne. Beregnes et gjennomsnitt som summen av antall verdier delt på antall rader, kan det utgjøre en forskjell om nullmerker tas med eller ikke. Eksempler på dette vises senere.

## 2.7. Entitetsintegritet (Entity Integrity) – Gjelder primærnøkler

**Entitetsintegritet** er regler som skal sikre at kravene som gjelder for primærnøkler ikke brytes. Dette er regler som DBMS vil håndtere for oss. DBMS vil gi feilmelding dersom reglene brytes.

**For primærnøklers krav til entitetsintegritet gjelder to hovedregler.**

1. Enhver rad må unikt kunne identifiseres, hvilket innebærer at det ***ikke*** tillates duplikater (repeterte verdier) i noen av radene for det eller de attributtene som inngår i primærnøkkelen.
2. Det tillattes aldri nullmerker i rader for ett eller flere attributter som inngår i primærnøkkelen.

## 2.8. Navnekonvensjon

Som for programmering, er det også mange meninger om hvordan en navnekonvensjon bør være for navngivning av tabeller og annet. Det aller viktigste er at man ***har*** en navnekonvensjon, og derfor vil stort sett alle bedrifter ha dette for sine programmeringsprosjekter. I dette kurset benyttes følgende:

- Tabellnavn: **Store bokstaver** og **entallsform** (eksempel: **CAR**, ikke CARS). Består navnet av flere ord, benyttes eventuelt understrekning mellom hver del, for eksempel **CAR_OWNER**.
- Kolonnenavn (attributter): **PascalCase** og **entallsform**: (eksempel: **CarMake**, **RegistrationDate**).

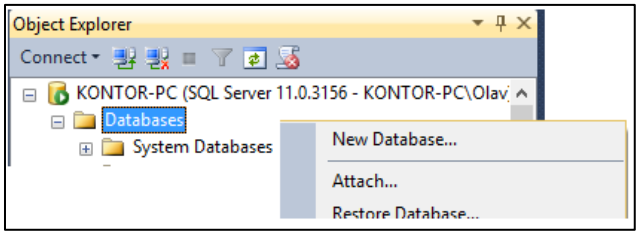
**Annet:**

- Benytt kun engelsk språk i navnsettingen.
- Ikke forkort navnene på en måte som gjør det vanskelig å forstå hva de representerer.
- Benytt store bokstaver for SQL-kommandoer (**WHERE**, **FROM** etc.).

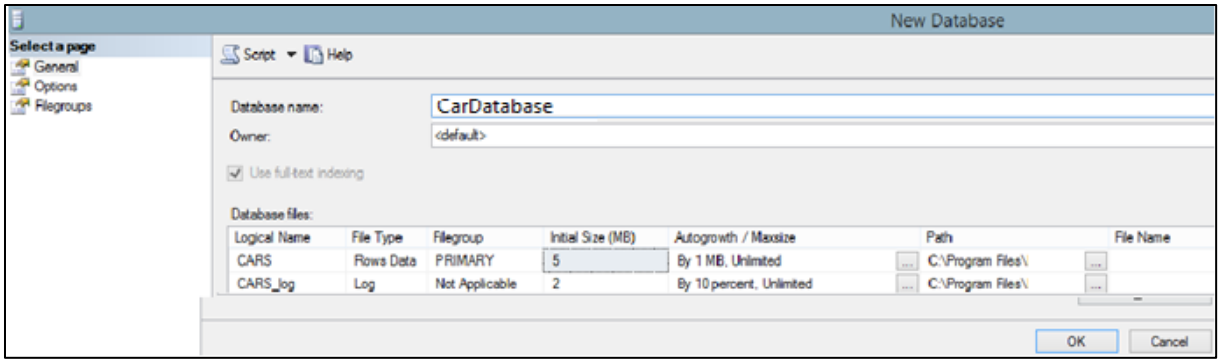
## 2.9. Opprette en database med det «grafiske» verktøyet i SQL Server

Nedenfor er det vist trinn for trinn hvordan tabellen fra Figur 2-2 kan opprettes med det grafiske verktøyet i SQL Server. Senere vil det vises hvordan det samme kan gjøres med SQL-kommandoer.

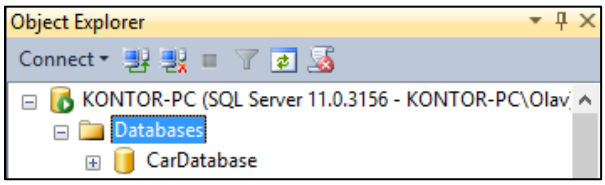
**Trinn 1:** Høyreklikk over «Databases» og velg «New Database…».



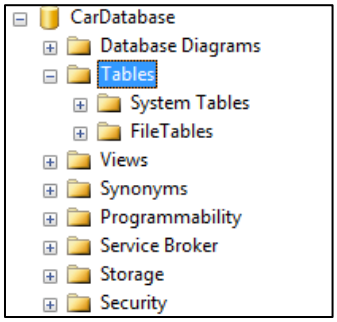
**Trinn 2:** Skriv databasenavn (CarDatabase) og velg «OK».



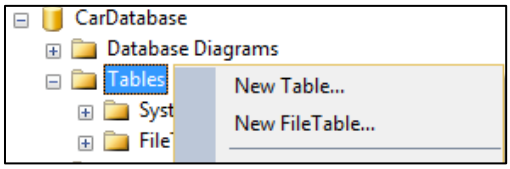

**Trinn 3:** Databasen skal nå være å finne i «Object Explorer», som vist nedenfor.



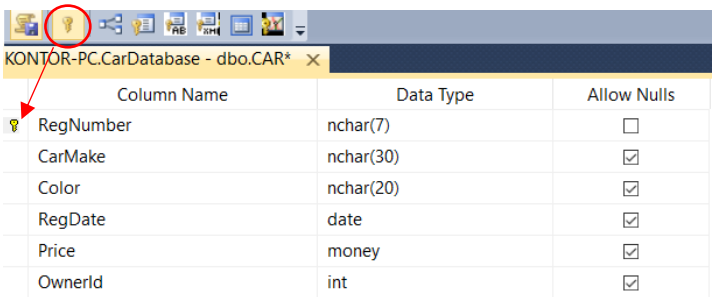

Klikk på pluss(+)-symbolene for å utvide mappene, som vist nedenfor.




**Trinn 4:** Høyreklik over «Tables» og velg «New Table».

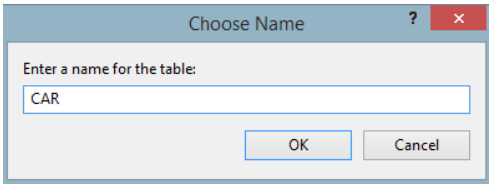



**Trinn 5:** Opprett en tabell med navn «**CAR**» i databasen. Definer felt og datatyper som vist nedenfor. Flytt kursoren til den øverste raden (**RegNumber**) så denne blir avmerket. Klikk så på nøkkelsymbolet i menyen (**Set Primary Key**). Feltet blir da angitt som primærnøkkel, med nøkkelsymbolvisning.

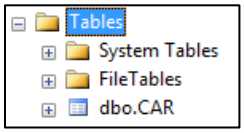



**Trinn 6:** Tabellen er nå definert. Klikk da på lagresymbolet (Save Table) på menylinjen:  .

Velg navnet **CAR** for tabellen som skal lages.




**Trinn 5:** Marker «**Tables**» og klikk så på **F5**-tasten (eventuelt **View -> Refresh**). Tabellen **dbo.Car** vil da vises. SQL Server opererer med mulighet for å definere ulike «skjemaer» og påfører prefikset «**dbo**» (**d**ata**b**ase **o**wner) som standardverdi på databasene som lages.




## 2.10. Endre tabellstruktur etter at tabellen er opprettet

Dersom tabellstrukturen senere ønskes endret, høyreklikk da over **dbo.CAR** i «Object Explorer». Velg så «Design» og definisjonsvinduet for kolonnenavn og datatyper vil fremkomme.

Ved eksempelvis endringer av datatype i etterkant, må det sørges for at det ikke endres til en datatype som gjør at det mistes data eller presisjon i allerede eksisterende data.

## 2.11. Legge data inn i tabellen via det grafiske brukergrensesnittet

For å legge data inn i tabellrader, høyreklikk over **dbo.CAR** i «**Object Explorer**» og velge «**Edit Top 200 Rows**». Da fremvises tabellen for editering, som vist i Figur 2-4.

KONTOR-PC.Union_I..._Except - dbo.CAR

| | RegNumber | CarMake | Color | RegDate | Price | OwnerId |
|---|---|---|---|---|---|---|
| * | *NULL* | *NULL* | *NULL* | *NULL* | *NULL* | *NULL* |

**Figur 2-4: Tabell klar for registrering av data.**

Legg inn eksempeldataene vist i Figur 2-5.

KONTOR-PC.Union_I..._Except - dbo.CAR

| RegNumber | CarMake | Color | RegDate | Price | OwnerId |
|---|---|---|---|---|---|
| CF26702 | Mazda | Red | 2011-09-09 | 320000,0000 | 3 |
| DK88971 | Peugeot | Blue | 2006-06-05 | 250000,0000 | 1 |
| FS33932 | Audi | Black | 2014-12-05 | 530000,0000 | 4 |
| KC93332 | Toyota | Silver | 2013-12-27 | 410000,0000 | 3 |
| LY13335 | Mercedes | Silver | 2003-12-09 | 350000,0000 | 6 |
| NL50034 | Volvo | Red | 2000-02-16 | 230000,0000 | 4 |
| XX45999 | Peugeot | Green | 2015-03-12 | 470000,0000 | 1 |
| YC43229 | Honda | Blue | 2012-03-12 | 460000,0000 | 2 |
| YZ33892 | Peugeot | Blue | 2012-07-11 | 430000,0000 | 1 |

**Figur 2-5: Eksempel på innlegging av data i tabellen med navn "CAR".**

Siden **RegNumber** er definert som en **primærnøkkel**, vil det gis feilmelding dersom det forsøkes å registrere en ny bil med samme registreringsnummer som allerede finnes. Dette da primærnøkkelen ikke tillater **repeterende verdier**.

På tilsvarende vis gis det feilmelding om det forsøkes å registrere data om en bil ***uten*** at det fylles ut noe i feltet **RegNumber**. Dette da det ikke tillates **nullmerker** i primærnøkkelfelt.

## 2.12. Vise data i tabellen via det grafiske brukergrensesnittet

For å vise data fra en tabell, høyreklikk over **cbo.CAR** og velg «**Select Top 1000 Rows**». Da vil det vises inntil 1000 rader med data. I vårt tilfelle er det bare lagt inn 9 rader, så da vil disse 9 radene vises, slik det fremgår av Figur 2-6.

Results | Messages

| | RegNumber | CarMake | Color | RegDate | Price | OwnerID |
|---|---|---|---|---|---|---|
| 1 | CF26702 | Mazda | Red | 2011-09-09 | 320000,00 | 3 |
| 2 | DK88971 | Peugeot | Blue | 2006-06-05 | 250000,00 | 1 |
| 3 | FS33932 | Audi | Black | 2014-12-05 | 530000,00 | 4 |
| 4 | KC93332 | Toyota | Silver | 2013-12-27 | 410000,00 | 3 |
| 5 | LY13335 | Mercedes | Silver | 2003-12-09 | 350000,00 | 6 |
| 6 | NL50034 | Volvo | Red | 2000-02-16 | 230000,00 | 4 |
| 7 | XX45999 | Peugeot | Green | 2015-03-12 | 470000,00 | 1 |
| 8 | YC43229 | Honda | Blue | 2012-03-12 | 460000,00 | 2 |
| 9 | YZ33892 | Peugeot | Blue | 2012-07-11 | 430000,00 | 1 |

**Figur 2-6: Visning av inntil 1000 rader med kommandoen «Select Top 1000 Rows».**

Øverst i samme skjermbildet vil det samtidig vises SQL-kommandoen som er utført. Se Figur 2-7.

```
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT TOP 1000 [RegNumber]
      ,[CarMake]
      ,[Color]
      ,[RegDate]
      ,[Price]
      ,[OwnerId]
  FROM [CarDatabase].[dbo].[CAR]
```

**Figur 2-7: Kode som er utført for å vise inntil 1000 rader med data.**

Endre «**TOP 1000**» til «**TOP 2**», og trykk så på **Execute** i menyen: Execute. Trykk på **F5**-tasten vil også starte kjøringen av spørringen.

Da blir spørringen kjørt på nytt med visning av bare 2 rader med data, som vist i Figur 2-8.

Results | Messages

| | RegNum... | CarMake | Color | RegDate | Price | OwnerId |
|---|---|---|---|---|---|---|
| 1 | CF26702 | Mazda | Red | 2011-09-09 | 320000,00 | 3 |
| 2 | DK88971 | Peugeot | Blue | 2006-06-05 | 250000,00 | 1 |

**Figur 2-8: Resultat etter kjøring av SQL-spørring med visning av kun 2 rader.**

Endre så «scriptet» vist i Figur 2-7, så det står 3 istedenfor 1000, og fjern i tillegg **RegDate**, **Price** og **OwnerId**. Scriptet skal da bli som vist i Figur 2-9.

```
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT TOP 3 [RegNumber]
      ,[CarMake]
      ,[Color]
  FROM [CarDatabase].[dbo].[CAR]
```

**Figur 2-9: SQL-script for visning av 3 kolonner og 3 rader med data.**

Ved kjøring vil det nå vises kun de 3 kolonnene **RegNumber**, **CarMake** og **Color**, samt 3 rader med verdier, som vist i Figur 2-10. Dette er eksempler på kjøring av SQL-spørringer.

Results | Messages

| | RegNumber | CarMake | Color |
|---|---|---|---|
| 1 | CF26702 | Mazda | Red |
| 2 | DK88971 | Peugeot | Blue |
| 3 | FS33932 | Audi | Black |

**Figur 2-10: Visning av tre kolonner og tre rader med data.**

# 3. SQL – Structured Query Language

**SQL** (**Structured Query Language**) er et spørrespråk som gir mulighet for å definere tabeller, oppdatere data og hente ut data fra tabeller på en strukturert og effektiv måte, uten å måtte detaljprogrammere med et programmeringsspråk.

SQL er en standard som er implementert i de fleste DBMS. Standarden kom i sin første versjon i 1986 (kalt SQL-87) og den nyeste standarden er fra 2016 [6].

Selv om det er en standard, er det likevel sånn at de ulike DBMS-leverandørene som regel ikke følger denne fullt og helt. Spesielt legges det gjerne til ikke-standardiserte utvidelser med tilleggsfunksjoner som ønskes. Mye av dette tas så gjerne senere opp i standardene, men da kan ulike systemer allerede ha implementert det på ulikt vis.

Den grunnleggende funksjonaliteten er likevel stort sett lik i alle DBMS.

SQL inndeles gjerne i tre hovedgrupper med SQL-instruksjoner, «Data Definition Language» (**DDL**), «Data Manipulation Language» (**DML**) og «Data Control Language» (**DCL**).

## 3.1. TRANSACT SQL (T-SQL)

For SQL Server, som benyttes i dette kurset, har Microsoft kalt sin SQL-implementasjon for «**Transact SQL**» (**T-SQL**) [7]. I det følgende forklares de mest sentrale SQL-kommandoene i T-SQL (der det meste er generell SQL). For mer kompleks bruk ligger det utførlig dokumentasjon tilgjengelig på nett.

## 3.2. Data Definition Language (DDL)

Definition Language (**DDL**) er den delen av SQL som gir mulighet for definisjon og senere endring av selve tabellstrukturen, med definisjon av tabellnavn, attributtnavn, datatyper, primærnøkkel, fremmednøkler m.m. De tre viktigste operasjonene er:

- Opprette tabeller (**CREATE TABLE**)
- Endre en eksisterende tabellstruktur (**ALTER TABLE**)
- Slette en tabell (**DROP TABLE**)

## 3.3. Data Manipulation Language (DML)

Data Manipulation Language (**DML**) er den delen av SQL som gir mulighet for å utføre operasjoner på selve dataene som skal lagres i tabellene. Det er her fire viktige typer operasjoner.

Operasjoner for å:

- legge inn data (**INSERT**)
- hente ut data (**SELECT**)
- modifisere og oppdatere data (**UPDATE**)
- slette data (**DELETE**)

## 3.4. Data Control Language» (DCL).

Data Control Language (**DCL**) er den delen av SQL som gir mulighet for å styre brukertilgang til tabeller og disses innhold. De to viktigste operasjonene er:

- Tildele rettigheter (**GRANT**)
- Fjerne rettigheter (**REVOKE**)

# 4. Lage, modifisere og slette tabeller

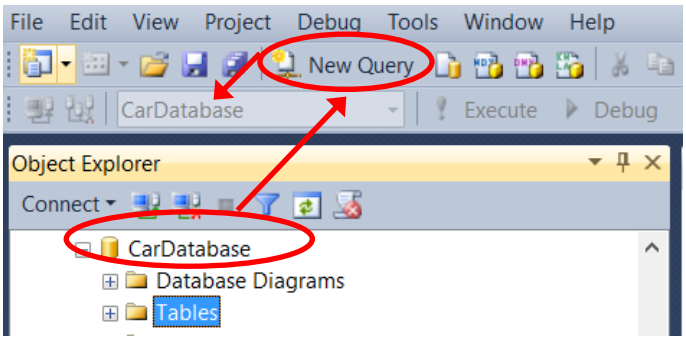
Den første SQL-delen det ses på ligger innunder «Data Definition Language» (DDL), der det ses på hvordan tabeller (og disses struktur) kan lages, modifiseres og slettes med SQL-instruksjoner.

## 4.1. CREATE TABLE

I kapittel 2.9 ble det vist hvordan det kunne opprettes en tabell ved hjelp av det grafiske verktøyet i **SQL Server**. Tabellen som ble opprettet fikk navnet **CAR**.

Nå skal det opprettes en tilsvarende tabell med SQL. Siden det allerede finnes en tabell med navn **CAR**, gis den nye tabellen navnet **CAR2**. Det finnes en rekke tilleggsopsjoner i strukturen. For å se alle disse mulighetene, må det slås opp i Microsoft-dokumentasjonen. I vårt tilfelle ses det kun på de grunnleggende elementene, og særskilt de som er i tråd med den generelle SQL-standarden.

Det første som må gjøres for å kunne utføre en SQL-spørring fra SQL Server er å åpne et konsollvindu for spørringer. Avmerk da først databasen (i vårt tilfelle **CarDatabase**) som tabellen skal opprettes i. Klikk så på «**New Query**», som vist i Figur 4-1, og sjekk at **CarDatabase** er valgt. I nedtrekksmenyen for dette feltet kan det eventuelt velges annen database spørringen skal utføres mot.




**Figur 4-1: Åpning av et konsollvindu for å lage en SQL-spørring.**

Det vil åpne seg et konsollvindu der det kan skrives SQL-kommandoer (jf. Figur 4-3). Syntaks for opprettelse av en tabell med **CREATE TABLE** er vist i Figur 4-1.

```
CREATE TABLE TABLENAME
(
    attribute1 datatype [DEFAULT] / [NULL] / [NOT NULL] [UNIQUE],
    attribute2 datatype [DEFAULT] / [[NULL] / [NOT NULL], [UNIQUE],
    /* more attributes */,
    CONSTRAINT PK_TABLENAME PRIMARY KEY (attributt1)
)
```

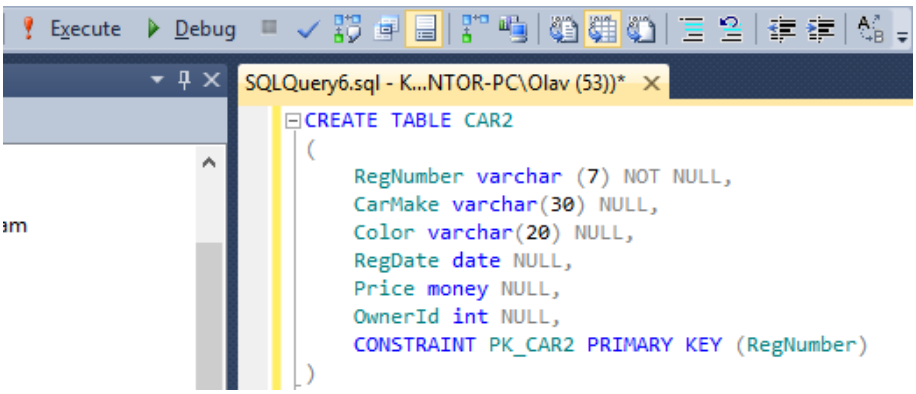
**Figur 4-2: Opprettelse av en tabell med CREATE TABLE.**

I syntaksen inngår det også en primærnøkkeldefinisjon. Det finnes i T-SQL flere måter å gjøre dette på, men måten som er vist er SQL-standarden. De valgfrie «attributt-constraints»-elementene **DEFAULT**, **NULL**, **NOT NULL** og **UNIQUE**, vist i Figur 4-2, forklares i kapittel 4.3.

Først angis nøkkelordet **CONSTRAINT**, deretter angis en identifikator og så nøkkelordene «**PRIMARY KEY**». Til slutt angis den eller de attributt(-er) som skal utgjøre primærnøkkelen.

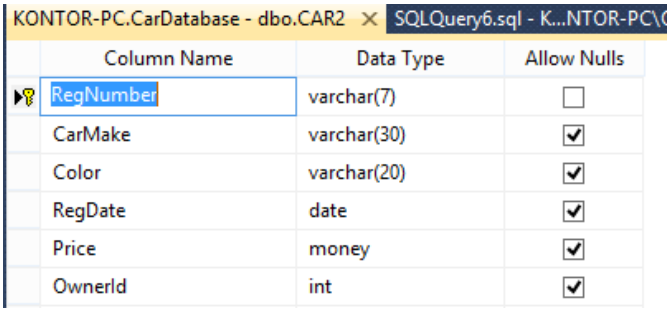
Det er ikke noe krav til hvordan identifikatoren skal navnsettes, men i dette kurset benyttes følgende navnekonvensjon: **PK** for å vise at det gjelder en «**Primary Key**», deretter en understrekning, og til slutt tabellnavnet, altså **PK_TABELLNAVN**. Da er det enkelt å vite hvilken tabell nøkkelen gjelder, hvilket er nyttig dersom den senere skal slettes eller endres via SQL.

SQL for å lage tabellen **CAR2** kan nå skrives i konsollvinduet, som vist i Figur 4-3. Når spørringen er skrevet, klikk på «**Execute**» (som er vist i menyen i samme figur) for å utføre spørringen (eller **F5**-tasten). Etter kjøring skal tabellen være tilgjengelig i «**Object Explorer**». Tabellen vises trolig først etter en «**refresh**», som kan utføres ved å avmerke mappen **CarDatabase** i «**Object Explorer**» og så trykke på **F5**-tasten (eller via menyvalget ***View -> Refresh***).




**Figur 4-3: SQL for å opprette tabellen CAR2. Trykk "Execute" for å kjøre spørringen.**

Ved å høyreklikke over den nye tabellen **CAR2** i «Object Explorer» og så velge «Design», vil det fremkomme en oversikt over tabelldefinisjonen, jf. Figur 4-4. Legg merke til at «Allow Nulls» samsvarer med det som er definert for hvert enkelt attributt som **NULL**/**NOT NULL** i Figur 4-3.




**Figur 4-4: Designmodus for tabellen som ble laget med CREATE TABLE-instruksjonen.**

Hvis man sammenligner med tabellen som tidligere ble laget grafisk, ses det at datatypene der var **nchar** istedenfor **varchar**. Det kunne også vært benyttet **nchar** istedenfor **varchar** i **CREATE TABLE**-definisjonen, da disse fungerer likt, men der **varchar** er SQL-standarden.

## 4.2. Utføre og lagre SQL-kommandoer i konsollvinduet

Før det gås videre, er det nyttig å kjenne til følgende når det gjelder kjøring av SQL-instruksjoner.

Dersom **CREATE TABLE**-setningen som ble laget ønskes lagret, for senere å kunne kjøres på nytt, gjøres dette via menyvalget ***File -> Save* Name.sql**.

Eventuelt kan ***File -> Save As* Name.sql** benyttes, dersom man ønsker å spesifisere/endre navn (eksempelvis **CreateTableCAR2**, for spørringen laget ovenfor). Opprett en fornuftige mappestrukturer for lagring av spørringer, så det blir enkelt å finne og gjenbruke dem senere.

Det kan skrives og kjøres flere SQL-instruksjoner i samme konsollvindu. Uten avmerking, utføres alle instruksjonene direkte etter hverandre. Dersom én eller flere SQL-instruksjoner avmerkes, er det bare denne/disse som blir utført ved klikk på **Execute**.

## 4.3. CONSTRAINTS – CHECK, DEFAULT, NULL, NOT NULL og UNIQUE

I **CREATE TABLE**-definisjonen vist i Figur 4-2 fremgår det noen valgfrie «constraints» etter attributtene. Disse er her forklart nærmere.

**NULL:**

Dette betyr ***at*** det tillates å la kolonnen stå tom når det registreres verdier i en rad.

**NOT NULL:**
Det vil da ***ikke*** tillates **NULL**-merker i raden for denne kolonnen. En verdi ***må*** legges inn. Et felt definert som primærnøkkel blir av «SQL Server» alltid automatisk definert som NOT NULL.

**DEFAULT:**

Dersom det for en rad ikke fylles ut en verdi i en kolonne som er definert med en **DEFAULT**-verdi, fylles isteden automatisk inn den definerte **DEFAULT**-verdien.

**UNIQUE:**

I en kolonne (et attributt) definert som **UNIQUE**, tillattes det ikke duplikater (repeterte verdier) i denne kolonnen. Når et felt defineres som **PRIMARY KEY** blir den automatisk **NOT NULL** og **UNIQUE** (uten av dette angis spesifikt). **DBMS** vil grunnet referanseintegritet ikke tillate **NULL**-merker og duplikater i primærnøkler (som ble forklart i kapittel 2.7).

**CHECK:**

**CHECK** gir anledning til å legge til ytterligere begrensninger på dataene som skal lagres i en kolonne enn bare datatypen. Det er egentlig et **domene** (sett med verdier) som defineres for hvert attributt, ikke bare datatypen (men ofte er datatypen tilstrekkelig til å definere domenet). Det kan bl.a. angis begrensninger i hvilke typer verdier som skal tillates registrert.

I Figur 4-5 er det laget en tabell kalt **CAR3**, med diverse «constraints» innlagt.

```
CREATE TABLE CAR3
(
    RegNumber varchar (7), --Primærnøkkel blir pr. def. NOT NULL og UNIQUE
    CarMake varchar(30) NULL, -- Nullmerker tillates
    Color varchar(20) NULL DEFAULT 'Unknown', --'Unknown' fylles inn når verdi ikke angis
    RegDate date NULL,
    Price money NULL,
    OwnerId int NOT NULL UNIQUE, --Tillater ikke duplikater og NULL-verdier
    CONSTRAINT PK_CAR3 PRIMARY KEY (RegNumber), --Definerer RegNumber som primærnøkkel
    CHECK ((Price >= 0) AND (Price < 2000000)), --Pris må være i området 0-2000000
    CHECK (Color IN ('Unknown','Blue', 'Red')) -- Bare utvalgte "farger" tillates
)
```

**Figur 4-5: Eksempel på CREATE TABLE med ulike typer constraints lagt til på attributtene/kolonnene.**

## 4.4. ALTER TABLE

Med **ALTER TABLE** kan det gjøres endringer i strukturen til en eksisterende tabell. Nedenfor vises en del eksempler på bruk av denne instruksjonen. For andre behov, slå opp i dokumentasjonen.

**Endring av datatype:**

La oss si at det for tabellen **CAR3** ønskes å endre datatypen til feltet **CarMake** fra **varchar (30)** til **varchar (40)**, så kan dette utføres med **ALTER COLUMN**-kommandoen, som vist i Figur 4-6.

```
ALTER TABLE CAR3
ALTER COLUMN CarMake varchar (40)
```

**Figur 4-6: SQL for å endre datatypen til feltet CarMake i tabellen CAR3.**

Vær oppmerksom på risikoen ved å endre datatype for en eksisterende tabell. Dersom det f.eks. allerede ligger lagret data i tabellen, kan en innskrenking av datatypen medføre tap av data.

**Legge til en kolonne:**

En ekstra kolonne kan legges til tabellen med **ADD**-kommandoen, som vist med SQL-setningen i Figur 4-7 (der kolonnen **Country** tilføyes).

```
ALTER TABLE CAR3
ADD Country varchar (50);
```

**Figur 4-7: Tilføyelse av en kolonne kalt Country i tabellen CAR3.**

**Slette en kolonne:**

Kolonnen **Country**, som ble lagt med kommandoen vist i Figur 4-7, kan slettes med SQL-kommandoen **DROP**, som vist i Figur 4-8.

```
ALTER TABLE CAR3
DROP COLUMN Country;
```

**Figur 4-8: Sletting av kolonnen Country i tabellen CAR3.**

**ALTER TABLE** kan også benyttes til å slette eller legge til en primærnøkkel. Da benyttes **ADD** for å legge til og **DROP** for å slette. **NB!** Dersom et felt legges til som primærnøkkel, må det først være definert som **NOT NULL**.

Som eksempel skal det nå legges til et nytt felt kalt **CarId**. Deretter skal primærnøkkelen endres fra **RegNumber** til **CarId**.

For å utføre dette, må det kjøres flere SQL-instruksjoner. Den første må legge til det nye feltet **CarId**, den neste må slette primærnøkkelen til feltet **RegNumber** og den siste må lage en primærnøkkel for attributtet **CarId**. De tre instruksjonene som må utføres er vist i Figur 4-9.

```
ALTER TABLE CAR3 /* ALTER TABLE for å endre en eksisterende tabell */
DROP CONSTRAINT PK_CAR3; /* Sletter fremmednøkkelen */

ALTER TABLE CAR3 /* ALTER TABLE for å endre en eksisterende tabell */
ADD CarId int NOT NULL /* Legger til en ny kolonne (atributt) kalt CarId */

ALTER TABLE CAR3 /* ALTER TABLE for å endre en eksisterende tabell */
ADD CONSTRAINT PK_CAR3 PRIMARY KEY (CarId); /*Legger til primærnøkkel */
```

**Figur 4-9: SQL-instruksjoner for å legge til felt og endre tabellens primærnøkkel.**

I koden vist i Figur 4-9 er det lagt inn kommentarer som kompilatoren vil ignorere. Det er to måter å legge inn kommentarer på. De ene er å starte det som skal kommenteres bort med to bindestreker (- - kommentar). Alternativt kan et område kommenteres bort sånn: /* kommentar */.

Legg merke til hvordan identifikatoren **PK_CAR3** i Figur 4-9 benyttes til å angi primærnøkkelen som skal slettes.

Etter at instruksjonene er utført, ta en refresh av «**Object Explorer**». Høyreklikk så på **CAR3** og velg «**Design**». Nytt felt med primærnøkkelangivelse som vist i Figur 4-10, skal da vises.

| | Column Name | Data Type | Allow Nulls |
|---|---|---|---|
| | RegNumber | varchar(7) | ☐ |
| | CarMake | varchar(30) | ☑ |
| | Color | varchar(20) | ☑ |
| | RegDate | date | ☑ |
| | Price | money | ☑ |
| | OwnerId | int | ☑ |
| 🔑 | CarId | int | ☐ |

**Figur 4-10: Nytt felt (CarId) er lagt til i tabellen og er gjort til primærnøkkel.**

## 4.5. DROP TABLE

Instruksjonen **DROP TABLE** sletter en hel tabell og alle dens data. Vær derfor sikker på at dette er ønskelig, før den utføres. I dette tilfellet kan det gjøres, da det i det videre skal jobbes med den opprinnelige tabellen kalt **CAR**.

For å slette tabellen **CAR3**, kan SQL-instruksjonen vist i Figur 4-11 utføres.

```
DROP TABLE CAR3
```

**Figur 4-11: DROP TABLE-instruksjon for å slette hele tabellen CAR3.**

# 5. SQL – Innsetting, sletting og endring av data

I de tidligere kapitlene ble det sett på **DDL**-delen (Data Definition Language) av SQL.

Nå skal det lages SQL-spørringer med instruksjoner fra **DML**-delen (Data Manipulation Language).

I kapittel 3.3 ble det forklart hvordan dette er instruksjoner som gir tilgang til å legge inn data, hente ut data, endre data og slette data. I motsetning til **DDL**, som gjaldt selve datastrukturen, er det nå altså selve dataene som manipuleres.

I kapittel 5 ses det på innsetting, sletting og endring av data, og i kapittel 6 ses det på ulike typer SQL-instruksjoner for å hente ut data fra tabeller. Alt dette ligger innunder **DDL**.

Det vil benyttes konkrete, praktiske eksempler i forklaringen av SQL-instruksjonene.

## 5.1. INSERT INTO

En **INSERT**-setning kan benyttes til å sette inn én eller flere rader med data i én tabell. Med én **INSERT**-setning kan det ikke legges data i flere tabeller. Da må det utføres flere **INSERT**-operasjoner.

For å angi hvilken tabell dataene skal inn i benyttes «**INSERT INTO** tabellnavn». Deretter benyttes nøkkelordet **VALUES** til å angi hvilke verdier som skal settes inn.

Det forutsettes i det videre at de ni radene med data, som ble vist i Figur 2-5, allerede ***er*** lagt inn i tabellen **CAR** via det grafiske brukergrensesnittet i SQL Server.

Med **INSERT**-instruksjonen skal det legges til en tiende rad i tabellen **CAR**. Denne skal inneholde følgende data: **AA11111**, **Audi**, **Red**, **19/4-2015**, **600 000** og **4**.

Avmerk tabellen i «Object Explorer» og klikk på «New Query». Spørringen er vist i Figur 5-1. Dersom det skal angis verdier for alle kolonnene, er det ikke påkrevd å angi kolonnenavnene.

```
INSERT INTO CAR
VALUES ('AA11111', 'Audi', 'Red', '2015-04-19', 600000, 4)
```

**Figur 5-1: Innsetting av en ny rad med data i tabellen CAR.**

**NB!** Dersom datatypen er av en tekstdatatype (jf. eksempelvis **varchar**), må dataene som skal legges inn angis med apostrofer rundt (eks: `'Audi'`). Datoer kan være komplisert og kan registreres på ulike måter, blant annet som vist i Figur 5-1 (`'yyyy-mm-dd'`), der **y** står for year, **m** for month og **d** for day.

Å oppgi dato ***uten*** bindestrekene i rekkefølgen (`'yyyymmdd'`) skal fungere ***uavhengig*** av system.

Dersom det i en rad ønskes lagt inn data bare i ***noen*** av kolonnene, kan de aktuelle kolonnenavnene angis ***etter*** tabellnavnet. I annet fall forventes det verdier i alle kolonnene. Husk at for kolonner definert som «**NOT NULL**» (inklusiv alle primærnøkkelfelt), ***må*** det angis verdier.

I Figur 5-2 er det vist hvordan det kan legges inn en rad med verdier bare for kolonnene **RegNumber**, **CarMake** og **Color**. Husk at kolonnen **RegNumber** ***må*** gis verdi, siden den er primærnøkkel. I annet fall vil DBMS gi en feilmelding ved forsøk på å kjøre instruksjonen.

```
INSERT INTO CAR (RegNumber, CarMake, Price)
VALUES ('AA22222', 'Nissan', 520000);
```

**Figur 5-2: INSERT-instruksjon med innlegging av verdier bare i utvalgte kolonner.**

De to radene som er lagt inn nå vil ha data som vist i Figur 5-3. Legg merke til hvordan det er merkert **NULL**-merker i felt der det ikke ble satt inn data.

| | | | | | |
|---|---|---|---|---|---|
| AA11111 | Audi | Red | 2015-04-19 | 600000,00 | 4 |
| AA22222 | Nissan | NULL | NULL | 520000,00 | NULL |

**Figur 5-3: Resultat av de to INSERT-setningene som er utført.**

Det er også mulig å sette inn flere rader samtidig. I Figur 5-4 er det vist innsetting av tre rader.

```
INSERT INTO CAR (RegNumber, CarMake, Color)
VALUES ('AA33333', 'Volvo', 'LightBlue'),
       ('AA44444', 'Volvo','DarkGreen'),
       ('AA55555', 'Volvo', 'Silver')
```

**Figur 5-4: Insetting av tre rader i én INSERT-operasjon.**

Resultatet av innsettingen av de tre radene er vist i Figur 5-5.

| | | | | | |
|---|---|---|---|---|---|
| AA33333 | Volvo | LightBlue | NULL | NULL | NULL |
| AA44444 | Volvo | DarkGreen | NULL | NULL | NULL |
| AA55555 | Volvo | Silver | NULL | NULL | NULL |

**Figur 5-5: Resultatet av innseting av tre rader med én INSERT-setning.**

## 5.2. DELETE

SQL-instruksjonen **DELETE** benyttes for å slette data i en tabell. Dersom **DELETE**-instruksjonen brukes uten tilleggskriterier, slettes absolutt alle verdiene i tabellen. Gjør derfor ikke dette med mindre du har backup av dataene og lett kan importere disse igjen om nødvendig. Vær altså klar over at det ikke finnes noen «undo»-mulighet etter sletting av data med **DELETE**. Sletting av alle data fra tabellen CAR er vist i Figur 5-6.

```
DELETE
FROM CAR
```

**Figur 5-6: SQL-instruksjon som sletter absolutt alle radene med data i tabellen CAR.**

Dersom det ønskes slettet bare spesifikke data, kan dette gjøres ved å legge til en **WHERE**-betingelse. For eksempelvis å slette bare den første raden vist i Figur 5-3, kan det lages en **DELETE**-instruksjon med betingelsen **WHERE RegNumber = 'AA11111'**, som vist i Figur 5-7.

```
DELETE
FROM CAR
WHERE RegNumber = 'AA11111'
```

**Figur 5-7: Instruksjon for å slette raden der RegNumber er eksakt lik AA11111.**

### 5.2.1. Logiske operatorer (benyttet med DELETE-instruksjonen)

I betingelsen kan det også benyttes logiske operatorer som **OR** og **AND**. Ønskes det slettet forekomster både der **RegNumber** er 'AA11111' og 'AA22222', kan dette gjøres ved å legge inn begge betingelsene med **OR**-operatoren mellom, som vist i Figur 5-8.

```
DELETE
FROM CAR
WHERE RegNumber = 'AA11111' OR RegNumber = 'AA22222'
```

**Figur 5-8: Sletting av utvalgte rader ved bruk av logisk operator (OR).**

### 5.2.2. IN-operator

Med **IN**-operatoren kan det i en spørring spesifiseres et utvalg av verdier det skal sjekkes mot. Dersom verdiene det sjekkes mot finnes i utvalget, returnerer delspørringen **True**, i annet fall **False**. I spørringen testes det så rad for rad, og de rader der resultatet blir **True** returneres.

I Figur 5-9 er det laget en spørring der bilene med de to registreringsnumrene 'AA11111' og 'AA22222' isteden slettes med en **IN**-operator. Det kan listes mange verdier, og det kan alternativt også lages en delspørring (behandles i kapittel 6.14) som returnerer verdier som skal brukes med **IN**.

```
DELETE
  FROM CAR
  WHERE RegNumber IN ('AA11111', 'AA22222')
```

**Figur 5-9: Spørring der IN-operator brukes til utvelgelse av**

### 5.2.3. Sammenligningsoperatorer (benyttet med DELETE-instruksjonen)

Sammenligningsoperatorer som <, <=, =, !=, >, >= (mindre enn, mindre eller lik, lik, forskjellig fra, større enn, større eller lik) kan også benyttes for å spesifisere hvilke poster som skal slettes.

Dersom det bare ønskes å slette biler der prisen er minst 600 000, kan dette gjøres ved å angi dette i **WHERE**-betingelsen, som vist i Figur 5-10.

```
DELETE
FROM CAR
WHERE Price >= 600000
```

**Figur 5-10: Sletting av alle rader der bilprisen er kr 600 000 eller mer.**

I dette tilfellet er det kun én bil med pris på 600 000 eller mer, vist i Figur 5-11, som dermed slettes. Dersom spørringen kjøres, kan denne posten legges inn igjen med **INSERT**-instruksjonen i Figur 5-1.

| RegNumber | CarMake | Color | RegDate | Price | OwnerID |
|---|---|---|---|---|---|
| AA11111 | Audi | Red | 2015-04-19 | 600000,00 | 4 |

**Figur 5-11: Følgende rad slettes i dette tilfellet.**

Mer komplekse betingelser kan også lages, som for eksempel i spørringen vist i Figur 5-12.

```
DELETE
FROM CAR
WHERE (Price >= 600000) AND (Price <= 800000)
```

**Figur 5-12: Sletting av alle poster der prisen ligger fra f.o.m. kr 600 000 t.o.m. kr 800000.**

Her kan også spørringen i Figur 5-1 kjøres, dersom posten ønskes lagt inn igjen etter sletting.

### 5.2.4. LIKE-operator (benyttet med DELETE-instruksjonen)

Istedenfor å spesifisere eksakte forekomster som skal slettes, kan det benyttes en **LIKE**-operator. **LIKE**-operatoren gir mulighet for å spesifisere søket i **WHERE**-betingelsen. I disse søkene kan det benyttes et **%**-symbol for å angi generelle tegn (wildcards) som skal med i søket uansett.

«**WHERE Color LIKE %blue**» angir alle ord som slutter på «blue» uansett hva som står foran, f.eks. «lightblue», «darkblue», «blue» osv.

«**WHERE Color LIKE dark%**» angir alle ord som starter med «dark» uansett hva som kom etter, f.eks. «darkred», «darkblue», «darkgreen» osv.

Alle radene som ble satt inn med **INSERT**-instruksjoner i kapittel 5.1, fikk et **RegNumber** som startet med teksten 'AA' (AA11111, AA22222, …, AA55555). Dersom det ønskes å slette alle disse 5 forekomstene samlet, kan **LIKE** benyttes. SQL-spørringen for dette er vist i Figur 5-13.

De kan da enkelt settes inn igjen med **INSERT**-spørringene fra Figur 5-1, Figur 5-2 og Figur 5-4.

```
DELETE
FROM CAR
WHERE RegNumber LIKE 'AA%' /*Sletter bare RegNumber som starter med 'AA' */
```

**Figur 5-13: Sletting av alle poster der RegNumber starter med bokstavene 'AA'.**

## 5.3. UPDATE og SET

**UPDATE** kan benyttes til å endre verdien på dataene i en tabell. På samme måte som med **DELETE**, må man være klar over at endringene ikke kan omgjøres. Benyttes **UPDATE** uten **WHERE**-angivelse, påføres endringen ***alle*** radene for den/de kolonnene som spesifiseres. På tilsvarende måte som for **DELETE** kan det også her benyttes **WHERE**-betingelser til å spesifisere hva som ønskes slettet.

Nøkkelordet **SET** benyttes for å angi én eller flere verdier som skal endres/settes.

Spørringen i Figur 5-14 vil endre fargen (**Color**) til blå i alle radene, så ikke gjør dette med mindre det foreligger en backup som kan gi dataene tilbake. For de øvrige spørringene på denne siden er det i laget tilleggsspørringer som reverserer effekten.

```
UPDATE CAR
SET Color = 'Blue' /* Alle bilene vil bli registrert med fargen 'Blue'*/
```

**Figur 5-14: Endring av fargen til 'Blue' i samtlige rader, uavhengig av hva den er fra før.**

Endring av alle biler med fargen «Silver» til fargen «Black» kan gjøres som vist i Figur 5-15.

```
UPDATE CAR /*Endrer alle med farge 'Silver' til 'Black' */
SET Color = 'Black' /*Setter fargen til 'Black' */
WHERE Color = 'Silver' /*Endrer bare til 'Black' der fargen er 'Silver' */
```

**Figur 5-15: Endrer biler med farge 'Silver' til fargen 'Black'.**

Spørringen vist i Figur 5-16 endrer tilbake igjen til fargen 'Silver', som var den opprinnelige.

```
UPDATE CAR /*Setter dataene tilbake til hva de var før endringen */
SET Color = 'Silver' /*Setter fargen til 'Silver' */
WHERE Color = 'Black' /*Endrer bare til 'Silver' der fargen er 'Black' */
```

**Figur 5-16: Endrer biler med farge 'Black' tilbake til fargen 'Silver'.**

Det er også mulig å endre flere felt, for eksempel ***både*** farge og pris. I Figur 5-17 er det gjort endring både av fargen (til 'Black' der den er 'Silver'), samtidig som prisen (Price-kolonnen) økes med kr 500 000 for de samme forekomstene (altså for biler med fargen 'Silver').

```
UPDATE CAR /*Endrer både farge og pris */
SET Color = 'Black', Price = Price + 500000 /*Farge til 'Black' og prisøkning på kr 500 000 */
WHERE Color = 'Silver' /*Endrer bare de forekomster der fargen er 'Silver' */
```

**Figur 5-17: Endrer både farge og pris for biler med fargen 'Silver'.**

Resultatet fra Figur 5-17 kan reverseres med spørringen vist i Figur 5-18.

```
UPDATE CAR /*Endrer farge og pris tilbake til hva det var */
SET Color = 'Silver' , Price = Price - 500000 /*Farge til 'Silver' og prisred. på kr 500 000 */
WHERE Color = 'Black' /*Endrer bare de forekomster der fargen er 'Black' */
```

**Figur 5-18: Endrer fargen og prisen tilbake til den opprinnelige.**

# 6. SQL-instruksjoner for å hente ut data fra tabeller

I kapittel 5 ble det sett på innsetting, sletting og endring av data i tabeller. Nå ses det på den siste og mest omfattende delen av **DML** (Data Manipulation Language), som går på å hente ut data fra databasen. I retur vil man få et spørreresultat i form av en resultattabell (en relasjon).

Det ses først på spørringer rettet mot kun én tabell. Senere ses det på spørringer også mot flere tabeller. Da behøves flere SQL-instruksjoner, men de som gjelder for én tabell vil fortsatt gjelde.

Nedenfor vises den grunnleggende oppstillingen/rekkefølgen for instruksjonene i SQL-spørringer der det skal hentes ut data. Elementene som er satt i klammer ikke er obligatoriske, men brukes ved behov. **SELECT** og **FROM *må*** alltid tas med. I underkapitlene forklares de ulike elementene grundigere.

**SELECT** kolonner [, kalkulasjoner] [,mengdeoperasjoner]
**FROM** tabell/tabeller
[**ON** tabellkoblinger (**JOIN**), dersom flere tabeller]
[**WHERE** radbetingelse]
[**GROUP BY** grupperingskolonne]
[**HAVING** gruppebetingelse]
[**ORDER BY** kolonne/kolonner stigende/fallende (**ASC**/**DESC**)]

## 6.1. SELECT og FROM – Grunnleggende spørrestruktur

Alle spørringer der data skal hentes ut, starter med en **SELECT**-del, der det angis hvilken/hvilke kolonner som ønskes med i spørreresultatet. Angis en stjerne (*), istedenfor spesifikke kolonner, returneres alle kolonnene. Spørringene i dette kapittelet er basert på at tabellen **CAR** inneholder verdiene vist i Figur 2-5 (9 verdier), Figur 5-1 (1 verdi), Figur 5-2 (1 verdi) og Figur 5-4 (3 verdier).

Siden det normalt lages spørringer mot ulike tabeller, kreves det også at det angis hvilken/hvilke tabeller(-er) spørringen retter seg mot. Dette angis med **FROM** etterfulgt av tabellnavn.

Spørringen vist i Figur 6-1 returnerer samtlige rader og kolonner fra tabellen **CAR**.

```
SELECT *      /* Alle kolonner tas med i resultatet */
FROM CAR      /* Dataene hentes fra tabellen CAR */
```

**Figur 6-1: SELECT-spørring som returnerer alle rader og kolonner fra tabellen CAR.**

Et utdrag av resultatet fra kjøring av spørringen er vist i Figur 6-2.

Results | Messages

| | RegNumber | CarMake | Color | RegDate | Price | OwnerID |
|---|---|---|---|---|---|---|
| 1 | CF26702 | Mazda | Red | 2011-09-09 | 320000,00 | 3 |
| 2 | DK88971 | Peugeot | Blue | 2006-06-05 | 250000,00 | 1 |
| 3 | ES33932 | Audi | Silver | 2014-12-05 | 530000,00 | 4 |

**Figur 6-2: Et utdrag av resultatet etter kjøring av spørringen vist i Figur 6-1.**

## 6.2. TOP-instruksjonen

Dersom det ikke ønskes å vise alle radene fra et spørreresultat, men f.eks. bare de 2 første radene, kan dette angis med **TOP**-instruksjonen. Dette er vist i Figur 6-3.

```
SELECT TOP 2 *    /* Alle kolonner tas med, men bare to rader */
FROM CAR      /* Dataene hentes fra tabellen CAR */
```

**Figur 6-3: Antall rader i resultatet begrenset til 2, med instruksjonen TOP 2.**

Kun de to første radene fra Figur 6-2 vil da bli med i resultatet.

## 6.3. Projeksjon - Utvelgelse av kolonner

Normalt ønskes ikke alle kolonnene tatt med i resultatet. Ønskes f.eks. bare **RegNumber** og **Price** tatt med, gjøres dette som vist i Figur 6-4. Dette kalles en «**projeksjon**», og man sier at tabellen «projiseres» på attributtene **RegNumber** og **Price**.

```
SELECT RegNumber, Price /* Bare kolonnene RegNumber og Price tas med i resultatet */
FROM CAR                /* Dataene hentes fra tabellen CAR */
```

**Figur 6-4: Spørring med retur fra bare to utvalgte kolonner.**

Et utdrag av resultatet fra kjøring av spørringen er vist i Figur 6-5.

Results | Messages

| | RegNumber | Price |
|---|---|---|
| 1 | CF26702 | 320000,00 |
| 2 | DK88971 | 250000,00 |
| 3 | ES33932 | 530000,00 |

**Figur 6-5: Et utdrag av resultatet etter kjøring av spørringen vist i Figur 6-4.**

## 6.4. Seleksjon – Utvelgelse av rader med WHERE-betingelser

På tilsvarende måte som det ofte ønskes bare et utvalg av kolonner, ønskes det i de fleste tilfeller også bare et utvalg av rader. En **WHERE**-betingelse gir mulighet for å velge/filtrere ut hvilket rader som ønskes returnert. I det følgende gis det en del eksempler.

I Figur 6-6 er det vist en spørring med retur av alle rader med fargen 'Blue' i kolonnen **Color**.

```
SELECT *
FROM CAR
WHERE Color = 'Blue' /*I resultatet skal bare rader med 'Blue' i Color-kolonnen tas med */
```

**Figur 6-6: Spørring der det ønskes retur av alle rader der Color er 'Blue'.**

Resultatet etter kjøring av spørringen er vist i Figur 6-7.

Results | Messages

| | RegNumber | CarMake | Color | RegDate | Price | OwnerID |
|---|---|---|---|---|---|---|
| 1 | DK88971 | Peugeot | Blue | 2006-06-05 | 250000,00 | 1 |
| 2 | YC43229 | Honda | Blue | 2012-03-12 | 460000,00 | 2 |
| 3 | YZ33892 | Peugeot | Blue | 2012-07-11 | 430000,00 | 1 |

**Figur 6-7: Resultat etter kjøring av spørringen vist i Figur 6-6. Bare rader med Color = 'Blue' er tatt med.**

Logiske operatorer som **AND**, **OR** og **NOT** kan tas med for å lage mer komplekse seleksjoner.

Spørring med retur av alle rader med Color 'Blue' ***eller*** Color 'Red' kan gjøres som vist i Figur 6-9.

```
SELECT *
FROM CAR
/*Retur av alle rader med Color = 'Blue' eller Color = 'Red' */
WHERE Color = 'Blue' OR Color = 'Red'
```

**Figur 6-8: Seleksjon der rader med Color = 'Red' eller Color = 'Blue' returneres.**

Resultatet etter kjøring av spørringen er vist i Figur 6-9.

Results | Messages

| | RegNumber | CarMake | Color | RegDate | Price | OwnerID |
|---|---|---|---|---|---|---|
| 1 | CF26702 | Mazda | Red | 2011-09-09 | 320000,00 | 3 |
| 2 | DK88971 | Peugeot | Blue | 2006-06-05 | 250000,00 | 1 |
| 3 | NL50034 | Volvo | Red | 2000-02-16 | 230000,00 | 4 |
| 4 | YC43229 | Honda | Blue | 2012-03-12 | 460000,00 | 2 |
| 5 | YZ33892 | Peugeot | Blue | 2012-07-11 | 430000,00 | 1 |
| 6 | AA11111 | Audi | Red | 2015-04-19 | 600000,00 | 4 |

**Figur 6-9: Resultat etter kjøring av spørringen vist i Figur 6-9.**

Dersom man ønsker ***bare*** de radene der fargen er rød eller blå ***og*** kun de rader prisen i tillegg er større eller lik kr 400 000 og mindre enn kr 600 000, kan betingelsen lages som vist i Figur 6-10.

**NB!** Benytt parenteser for å gjøre betingelsen entydig, selv om dette ikke alltid er påkrevd. De «indre» parentesene vil utføres/kalkuleres før den ytre. I dette tilfellet kreves det at fargen må være ***enten*** blå ***eller*** rød, ***samtidig*** som det bare ønskes biler i prisklassen fra og med kr 400 000 til kr 600 000.

```
SELECT *
FROM CAR
WHERE ((Color = 'Blue' OR Color = 'Red') AND (Price >= 400000 AND Price < 600000))
```

**Figur 6-10: Seleksjon der bare rader med angitte farger og angitt prisintervall returneres.**

Det kan også i **WHERE**-seleksjonene benyttes **LIKE**-operator istedenfor =, som ble introdusert i anledning **DELETE**-instruksjonen i kapittel 5.2.4. Husk da på at likhetstegnet betyr «**eksakt lik**», mens **LIKE** spesifiserer et «**søkemønster**».

I Figur 6-11 vises det en spørring der det er en **projisering** på kolonnene **RegNumber** og **CarMake**, samt en **seleksjon** der bare rader med **RegNumber** som starter med bokstavene «**AA**» skal med.

```
SELECT RegNumber, CarMake /*Projeksjon: Ønsker bare kolonnene RegNumber og CarMake */
FROM CAR
WHERE RegNumber LIKE 'AA%' /*Seleksjon: Ønsker bare rader der RegNumber starter med "AA" */
```

**Figur 6-11: Projisering på to kolonner og seleksjon av alle biler med bilnummer som starter med «AA».**

Resultatet etter kjøring av spørringen er vist i Figur 6-12.

Results | Messages

| | RegNumber | CarMake |
|---|---|---|
| 1 | AA11111 | Audi |
| 2 | AA22222 | Nissan |
| 3 | AA33333 | Volvo |
| 4 | AA44444 | Volvo |
| 5 | AA55555 | Volvo |

**Figur 6-12: Resultat etter kjøring av spørringen vist i Figur 6-11.**

## 6.5. ORDER BY – Sortere dataene i resultatsettet

Som tidligere nevnt er radenes rekkefølge per definisjon usorterte i en relasjonsdatabase, da det er primærnøkkelen som benyttes til å skille radene fra hverandre. Når det lages en spørring, er det likevel vanlig at man ønsker resultatet sortert på én eller flere kolonner. Til dette benyttes en instruksjon kalt «**ORDER BY**», etterfulgt av den eller de kolonnene det ønskes sortert på.

Etter sorteringskolonnen kan det angis om resultatet ønskes sortert «**stigende**» (**ASC**) eller «**fallende**» (**DESC**), der **ASC** på engelsk er forkortelse for «ascending» og **DESC** for «descending». Om det ikke angis noe, setter systemet **ASC** som «default» (standard), men det anbefales å angi for å tydeliggjøre.

I spørringen vist i Figur 6-13 returneres alle bilmerker sortert stigende på verdiene i **CarMake**-kolonnen. Dataene er projisert på attributtene (kolonnene) **CarMake** og **Price**.

```
SELECT CarMake, Price
FROM CAR
ORDER BY CarMake DESC /*Sorterer fallende på verdiene i CarMake-kolonnen */
```

**Figur 6-13: Fallende sortering på kolonnen CarMake.**

Resultatet er vist til venstre i Figur 6-15. Dersom det i tillegg ønskes sortert stigende på pris (Price-kolonnen) etter at bilmerkene er sortert fallende, kan dette gjøres som vist i Figur 6-14.

```
SELECT CarMake, Price
FROM CAR
ORDER BY CarMake DESC, Price ASC /*Sorterer fallende på CarMake og så stigende på Price */
```

**Figur 6-14: Sortering først fallende på CarMake og deretter stigende på Price.**

Resultatet er vist til venstre i Figur 6-15. Legg merke til hvordan **NULL**-merkene håndteres.

Results | Messages

| | CarMake | Price |
|---|---|---|
| 1 | Volvo | 230000,00 |
| 2 | Volvo | NULL |
| 3 | Volvo | NULL |
| 4 | Volvo | NULL |
| 5 | Toyota | 410000,00 |
| 6 | Peugeot | 250000,00 |
| 7 | Peugeot | 470000,00 |
| 8 | Peugeot | 430000,00 |
| 9 | Nissan | 520000,00 |
| 10 | Mercedes | 350000,00 |
| 11 | Mazda | 320000,00 |
| 12 | Honda | 460000,00 |
| 13 | Audi | 600000,00 |
| 14 | Audi | 530000,00 |

Results | Messages

| | CarMake | Price |
|---|---|---|
| 1 | Volvo | NULL |
| 2 | Volvo | NULL |
| 3 | Volvo | NULL |
| 4 | Volvo | 230000,00 |
| 5 | Toyota | 410000,00 |
| 6 | Peugeot | 250000,00 |
| 7 | Peugeot | 430000,00 |
| 8 | Peugeot | 470000,00 |
| 9 | Nissan | 520000,00 |
| 10 | Mercedes | 350000,00 |
| 11 | Mazda | 320000,00 |
| 12 | Honda | 460000,00 |
| 13 | Audi | 530000,00 |
| 14 | Audi | 600000,00 |

**Figur 6-15: Sorterte kolonner.**

I tillegg til at kolonner angis i **SELECT**-delen, kan det der også utføres kalkulasjoner med disse. La oss si at det skal beskattes 10 % av bilenes registrerte pris. For å beregne beløpene, kan det lages en spørring som vist i Figur 6-17. For å begrense antall rader i visningen er kun biltypen Audi tatt med.

```
SELECT RegNumber, CarMake, Price, Price * 0.10
FROM CAR
WHERE CarMake = 'Audi'
```

**Figur 6-16: Spørring som inkluderer en kalkulasjon der 10 % av prisen i Price-kolonnen beregnes.**

Resultatet blir som vist i Figur 6-17

Results | Messages

| | RegNumber | CarMake | Price | (No column name) |
|---|---|---|---|---|
| 1 | FS33932 | Audi | 530000,00 | 53000.000000 |
| 2 | AA11111 | Audi | 600000,00 | 60000.000000 |

**Figur 6-17: Spørreresultat som inkluderer en kalkulert kolonne.**

Legg merke til at kalkulasjonskolonnen ikke har fått noe fornuftig kolonnenavn. Det er fordi dette er en kalkulert kolonne. SQL Server vet da ikke hvilket navn den skal gi den. Derfor angis bare «**No column name**» som overskrift.

I spørringen kan det angis kolonnenavn for slike kolonner. Etter funksjonen angis da nøkkelordet **AS** og så et egenvalgt navn (**AS** kan også droppes). Dersom det ønskes navn bestående av flere ord med mellomrom mellom, angis disse i hakeparenteser. Eks.: **Price * 0.10 AS [Tax to be paid]**. Er navnet bare ***ett*** sammenhengende ord, kan hakeparentesene utelates. Spørringen er vist i Figur 6-18.

```
SELECT RegNumber, CarMake, Price, (Price * 0.10) AS [Tax to be paid]
FROM CAR
WHERE CarMake = 'Audi'
```

**Figur 6-18: Spørring der kalkulasjonskolonnen er gitt navnet «Tax to be paid».**

I Figur 6-19 vises kalkulasjonskolonnen med kolonneoverskriften «**Tax to be paid**».

Results | Messages

| | RegNumber | CarMake | Price | Tax to be paid |
|---|---|---|---|---|
| 1 | FS33932 | Audi | 530000,00 | 53000.000000 |
| 2 | AA11111 | Audi | 600000,00 | 60000.000000 |

**Figur 6-19: Den kalkulerte kolonnen vist med en overskrift.**

## 6.6. Formatering av valutaformat og datoformat

Som det ses av Figur 6-18 er ikke visningen av beløpene formatert på et fornuftig kroneformat. Det som vises i resultattabellene er formatert som tekststrenger, selv om det opprinnelig var verdier. Disse tekststrengene kan formateres før resultatvisning.

Det finnes flere måter å formatere på, der en av dem er å benytte **FORMAT**-funksjonen. I Figur 6-20 vises noen eksempler på bruk med dato- og valutaformat. Argumentet «**no**» kalles «Culture». Byttes «**no**» med «**en-GB**» blir det britisk valutaformat. Endres det til «**en-US**» blir det amerikansk osv. Søk på **FORMAT**-funksjonen og «**Culture**» for grundigere informasjon.

Merk at i datoformatering er «**day**» angitt med liten **d** mens «**month**» er angitt med stor «**M**». Disse verdiene er «case-sensitive» og det må slås opp i dokumentasjon for å vite hva som må brukes.

```
SELECT RegNumber, CarMake,
       FORMAT(RegDate, 'd/M-yyyy') AS RegDate, /* dato og format angis */
       FORMAT(Price, 'C', 'no') AS Price, /* Pris, C for Currency og no for norsk format angis) */
       FORMAT (Price * 0.10, 'C', 'no') AS [Tax to be paid] /* Samme som over */
FROM CAR
WHERE CarMake = 'Audi'
```

**Figur 6-20: Eksempler på formatering av valuta- og dato-verdier i resultatvisning.**

Figur 6-21 viser resultatet av formateringen. Sammenlign dette med f.eks. dato- og valutformatene tidligere i kapittelet vist i Figur 6-9.

Results | Messages

| | RegNumber | CarMake | RegDate | Price | Tax to be paid |
|---|---|---|---|---|---|
| 1 | FS33932 | Audi | 5/12-2014 | kr 530 000,00 | kr 53 000,00 |
| 2 | AA11111 | Audi | 19/4-2015 | kr 600 000,00 | kr 60 000,00 |

**Figur 6-21: Resultatet etter kjøring av SQL-spørringen vist i Figur 6-20.**

## 6.7. Datohåndtering

I spørringer kan det sammenlignes direkte mot datoer, og det er også i SQL Server tilgjengelig ulike funksjoner for håndtering av data. Eksempelvis vil **Year( )**, **Month( )** og **Day( )** trekke ut år, måned og dag fra en verdi angitt med datoformat. Funksjonen **GetDate()** vil returnere dagens dato.

**RegDate** er i tabellen **CAR** angitt med datatypen **Date**. Nedenfor er det angitt noen eksempler på spørringer mot denne datokolonnen.

Det ønskes en liste over alle biler registrert i **2015**. Spørring og resultat er vist i Figur 6-22.

```
SELECT *
FROM CAR
WHERE Year(RegDate) = 2015
```

Results | Messages

| | RegNumber | CarMake | Color | RegDate | Price | OwnerId |
|---|---|---|---|---|---|---|
| 1 | XX45999 | Peugeot | Green | 2015-03-12 | 470000,00 | 1 |
| 2 | AA11111 | Audi | Red | 2015-04-19 | 600000,00 | 4 |

**Figur 6-22: Spørring som returnerer alle biler registrert i år 2015.**

Det ønskes en liste over alle biler registrert i **april 2015**. Spørring og resultat er vist i Figur 6-23.

```
SELECT *
FROM CAR
WHERE Year(RegDate) = 2015 /* Bare biler fra 2015 */
AND Month(RegDate) = 4 /* Bare biler fra april */
```

Results | Messages

| | RegNumber | CarMa... | Color | RegDate | Price | OwnerId |
|---|---|---|---|---|---|---|
| 1 | AA11111 | Audi | Red | 2015-04-19 | 600000,00 | 4 |

**Figur 6-23: Spørring som returnerer alle biler registrert i april 2015.**

Det ønskes en liste over alle biler registrert i 1/4-20/4 i 2015. Spørringen er vist i Figur 6-24. Resultatet blir det samme som vist i Figur 6-23, da kun denne bilen er registrert i denne perioden.

```
SELECT *
FROM CAR
WHERE RegDate >= '2015-04-01'  /* Datoformat: 'yyyy-mm-dd' eller 'yyyymmdd'*/
AND   RegDate <= '2015-04-20'  /* fra og med 1/4-2015 til og med 20/4-2015 */
```

**Figur 6-24: Spørring som returnerer alle biler registrert i perioden 1/4-20/4 i 2015.**

Det ønskes en tilleggskolonne med beregning av bilenes alder. Spørringen er vist i Figur 6-25.

```
SELECT RegNumber, CarMake, RegDate, (Year(GetDate())-Year(RegDate)) AS CarAge
FROM CAR
```

**Figur 6-25: Spørring der en tilleggskolonne med beregning av bilenes alder er tatt med.**

Resultatet av spørringen er vist i Figur 6-26.

Results | Messages

| | RegNum... | CarMake | RegDate | CarAge |
|---|---|---|---|---|
| 1 | CF26702 | Mazda | 2011-09-09 | 4 |
| 2 | DK88971 | Peugeot | 2006-06-05 | 9 |
| 3 | FS33932 | Audi | 2014-12-05 | 1 |
| 4 | KC93332 | Toyota | 2013-12-27 | 2 |
| 5 | LY13335 | Mercedes | 2003-12-09 | 12 |
| 6 | NL50034 | Volvo | 2000-02-16 | 15 |
| 7 | XX45999 | Peugeot | 2015-03-12 | 0 |
| 8 | YC43229 | Honda | 2012-03-12 | 3 |
| 9 | YZ33892 | Peugeot | 2012-07-11 | 3 |
| 10 | AA11111 | Audi | 2015-04-19 | 0 |
| 11 | AA22222 | Nissan | NULL | NULL |
| 12 | AA33333 | Volvo | NULL | NULL |
| 13 | AA44444 | Volvo | NULL | NULL |
| 14 | AA55555 | Volvo | NULL | NULL |

**Figur 6-26: Resultat av spørring der også bilens alder er beregnet og tatt med i en kolonne.**

Se kapittel 6.8 for hvordan **NULL**-merkene kan håndteres.

## 6.8. Angivelse av NULL og NOT NULL i spørringer

I spørringsresultatet i Figur 6-26 ses det at det er tatt med en rekke **NULL**-merker. Dette er der det ikke er registrert noen registreringsdato for bilene, og det dermed følgelig heller ikke kan beregnes noen alder. Dersom disse radene ikke ønskes tatt med i resultatet, kan spørringen fra Figur 6-25 tilføyes en **NOT NULL**-angivelse. Dette er vist i Figur 6-27.

```
SELECT RegNumber, CarMake, RegDate, (Year(GetDate())-Year(RegDate)) AS CarAge
 FROM CAR
 WHERE RegDate IS NOT NULL
```

**Figur 6-27: Spørringen fra Figur 6-25 omskrevet til *ikke* å vise rader med NULL-merker.**

Resultatet vil bli som tabellen vist i Figur 6-26, men uten de 5 radene som inneholder **NULL**-merker i **RegDate**-feltet.

Dersom det tvert imot er ønskelig ***bare*** å vise de 5 radene ***med*** **NULL**-merker, kan **NOT** droppes. Da blir spørringen som vist i Figur 6-28.

```
SELECT RegNumber, CarMake, RegDate, (Year(GetDate())-Year(RegDate)) AS CarAge
FROM CAR
WHERE RegDate IS NULL
```

**Figur 6-28: Spørringen fra Figur 6-27 omskrevet til *bare* å vise rader med NULL-merker.**

## 6.9. Aggregeringsfunksjoner (COUNT, MIN, MAX, AVG og SUM)

I SQL Server finnes det diverse aggregeringsfunksjoner (eng: Aggregate Functions). Mest benyttet er:

- **COUNT** (Teller antall rader)
- **MIN** (Beregner den laveste verdien i en kolonne)
- **MAX** (Beregner den høyeste verdien i en kolonne)
- **AVG** (Beregner gjennomsnittet av verdiene i en kolonne)
- **SUM** (Beregner summen av alle verdiene i en kolonne)

Disse funksjonene opererer på alle radene i angitt kolonne (som angis i parentes). Dette sånn at **SUM(Price)** f.eks. vil gi summen av verdien for samtlige rader til kolonnen **Price**. En spørring der alle de ovennevnte funksjonene benyttes er vist i Figur 1-1.

```
SELECT COUNT(*), COUNT(Price), MIN(Price), MAX(Price), AVG(Price), SUM(Price)
FROM CAR
```

**Figur 6-29: Eksempel på bruk av aggregeringsfunksjoner.**

Resultatet av spørringen er vist i Figur 6-30.

Results | Messages

| | (No column name) | (No column name) | (No column name) | (No column name) | (No column name) | (No column name) |
|---|---|---|---|---|---|---|
| 1 | 14 | 11 | 230000,00 | 600000,00 | 415454,5454 | 4570000,00 |

**Figur 6-30: Resultat av spørring der ulike aggregeringsfunksjoner inngår.**

Ingen av kolonnene har fått noe fornuftig kolonnenavn, hvilket ble forklart i kapittel **Error! Reference source not found.**. I Figur 6-31 utføres samme spørring med selvvalgte kolonnenavn tilføyd.

```
SELECT COUNT(*) AS [Count all rows], COUNT(Price) AS [Count rows with values],
       MIN(Price) AS [Minimum Price], MAX(Price) AS [Maximum Price],
       AVG(Price) AS [Average Price], SUM(Price) AS Sum
```

**Figur 6-31: Spørring med anspesifiserte kolonnenavn.**

Resultatet blir nå en tabell påført de selvvalgte kolonnenavnene, som vist i Figur 6-32.

| | Count all rows | Count rows with values | Minimum Price | Maximum Price | Average Price | Sum |
|---|---|---|---|---|---|---|
| 1 | 14 | 11 | 230000,00 | 600000,00 | 415454,5454 | 4570000,00 |

**Figur 6-32: Spørreresultat med spesifiserte kolonnenavn.**

Legg merke til at **COUNT(*)** returnerer tallet 14 mens **COUNT(Price)** returnerer 11.

I førstnevnte tilfelle telles ***alle*** radene uansett om det måtte være nullmerker i noen av kolonnene.

Når **COUNT**-funksjonen angis med et spesifikt kolonnenavn i parentesen, som f.eks. **COUNT(Price)**, telles ***bare*** rader der det ***ikke*** ligger **NULL**-merker. Fra Figur 6-15 ses det at tre av de fjorten radene inneholder **NULL**-merker, hvilket er årsaken til at verdien **11** returneres. Det er viktig å kjenne til slike forskjeller, da kalkulasjoner ellers kan gi feil resultater.

Beregnes gjennomsnittet ved å bruke **SUM/ COUNT(*)**, vil man fått et annet resultat enn om man beregner det med **SUM(Price)/ COUNT(Price)**, dersom noen av radene inneholder **NULL**-merker. Det førstnevnte vil gi feil resultat, da det inkluderer **NULL**-merker i tellingen.

Benyttes **AVG**-funksjonen, ekskluderes **NULL**-merker sånn at resultatet blir korrekt. I Figur 6-33 vises en SQL-setning for de tre ovennevnte gjennomsnittsberegningene.

```
SELECT SUM(Price)/COUNT(*) AS [Average when NULL-values included],
       SUM (Price)/COUNT(Price) AS [Average when NULL-values not included],
       AVG (Price) AS [Average calculated with the AVG-function]
FROM CAR
```

**Figur 6-33: Spørring med tre kalkulasjoner av gjennomsnitt, der den første vil gi feil resultat.**

Resultatet av spørringen er vist i Figur 6-34.

Results | Messages

| | Average when NULL-values included | Average when NULL-values not included | Average calculated with the AVG-function |
|---|---|---|---|
| 1 | 326428,5714 | 415454,5454 | 415454,5454 |

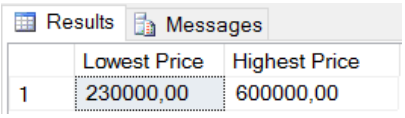
**Figur 6-34: Resultatet av spørringen vist i Figur 6-33.**

De andre funksjonene benyttes på tilsvarende vis, for å finne minste verdi og største verdi. Et eksempel er vist i Figur 6-35, der laveste og høyeste bilpris finnes.

```
SELECT MIN(Price) AS [Lowest Price], MAX(Price) AS [Highest Price]
FROM Car
```

**Figur 6-35: SQL-setning som finner laveste og høyeste bilpris.**

Resultatet av spørringen er vist i Figur 6-36.



| | Lowest Price | Highest Price |
|---|---|---|
| 1 | 230000,00 | 600000,00 |

**Figur 6-36: Spørreresultat for spørring som viser høyeste og laveste bilpris.**

**NB!** Når det gjelder aggregeringsfunksjoner returnerer disse alltid ***én*** verdi. I spørringene som her er vist, er det derfor ikke mulig i tillegg å vise verdier for andre kolonner i **SELECT**-delen. Dette fordi ett konkret bilmerke ikke kan knyttes til maksimalprisen, da det kan være flere biler med samme pris.

Én spesifikk rad kan derfor altså ikke kobles mot et aggregert resultat i en **SELECT**-del.

## 6.10. ROUND-funksjonen

**ROUND**-funksjonen (tilsvarende som i C#/.Net) kan benyttes til å avrunde et resultat til et angitt antall desimaler. Den tar to argumenter, der det første er tallet som skal avrundes og det andre er antall desimaler det skal avrundes til. I Figur 6-34 ses det at alle resultatene er vist med 4 desimaler. I Figur 6-37 er spørringen endret så de tre tallene avrundes til to desimaler.

```
SELECT ROUND(SUM(Price)/COUNT(*),2) AS [Average when NULL-values included],
       ROUND(SUM (Price)/COUNT(Price),2) AS [Average when NULL-values not included],
       ROUND(AVG (Price),2) AS [Average calculated with the AVG-function]
FROM CAR
```

**Figur 6-37: ROUND-funksjonen benyttes til å avrunde resultatene til 2 desimaler.**

Resultatet av avrundingen er vist i Figur 6-38.

| | Average when NULL-values included | Average when NULL-values not included | Average calculated with the AVG-function |
|---|---|---|---|
| 1 | 326428,57 | 415454,55 | 415454,55 |

**Figur 6-38: Resultatene avrundet til 2 desimaler.**

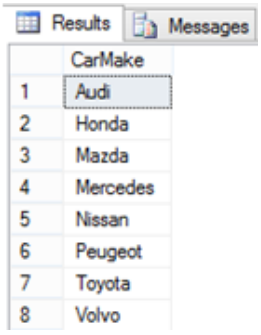
## 6.11. DISTINCT

**DISTINCT** vil fjerne duplikater (gjentatte verdier) i et resultat for en angitt kolonne. Ønskes eksempelvis bare én forekomst av hver biltype tatt med i resultatet (selv om biltypen måtte finnes i flere rader) kan nøkkelordet **DISTINCT** benyttes. **Volvo**, som finnes i fire rader, vil det da bare tas med den første forekomsten av. I Figur 6-39 er det laget en slik spørring.

```
SELECT DISTINCT CarMake /* Bare første forekomst av hver verdi tas med i resultatet */
FROM CAR
```

**Figur 6-39: DISTINCT brukes for å returnere kun første forekomst av hver verdi.**

Resultatet etter kjøring av spørringen er vist i Figur 6-12.



| | CarMake |
|---|---|
| 1 | Audi |
| 2 | Honda |
| 3 | Mazda |
| 4 | Mercedes |
| 5 | Nissan |
| 6 | Peugeot |
| 7 | Toyota |
| 8 | Volvo |

**Figur 6-40: Resultat etter bruk av DISTINCT på attributtet (kolonnen) CarMake.**

## 6.12. GROUP BY

**GROUP BY** grupperer resultatet i en angitt kolonne. Velges kolonnen **CarMake**, vil eksempelvis alle biler av typen **Audi** betraktes som én gruppe, alle av typen **Honda** betraktes som en annen gruppe osv.

Effekten av en enkel gruppering vil tilsynelatende bli det samme som **DISTINCT**, men ved gruppering kan man i tillegg utføre kalkulasjoner på innholdet i gruppen.

```
SELECT CarMake
FROM CAR
GROUP BY CarMake
```

**Figur 6-41: Spørring der det grupperes på innholdet i kolonnen CarMake.**

**GROUP BY**-spørringen i Figur 6-41 vil gi samme resultat (Figur 6-40) som **DISTINCT**-spørringen. Forskjellen er at resultatet nå inneholder en gruppering. For eksempel vil det være 2 Audier, 1 Honda, 4 Volvoer etc. i den underliggende grupperingen (jf. tidligere utlisting vist i Figur 6-15).

Det er mulig å utføre kalkulasjoner og benytte aggregeringsfunksjoner på de underliggende grupperte elementene. Dersom det ønskes å telle hvor mange det er i hver gruppe, dvs. hvor mange Audier, hvor mange Hondaer osv., kan dette gjøres med **COUNT**-funksjonen, som vist i Figur 6-42.

```
SELECT CarMake, COUNT(*) AS [Number of cars from each CarMake]
FROM CAR
GROUP BY CarMake
```

**Figur 6-42: Spørring med gruppering på kolonnen CarMake og telling av forkomstene i hver gruppe.**

Spørringen vil gi resultatet vist i Figur 6-43, der det i en egen kolonne vises antallet av hvert bilmerke.

Results | Messages

| | CarMake | Number of cars from each CarMake |
|---|---|---|
| 1 | Audi | 2 |
| 2 | Honda | 1 |
| 3 | Mazda | 1 |
| 4 | Mercedes | 1 |
| 5 | Nissan | 1 |
| 6 | Peugeot | 3 |
| 7 | Toyota | 1 |
| 8 | Volvo | 4 |

**Figur 6-43: Antall bilmerker gruppert, med angivelse av antall biler i hver gruppe.**

Det totale antall rader er altså først gruppert etter bilmerke. Deretter telles (med **COUNT**) antallet biler i hver gruppe, altså antall Audier (2), antall Hondaer (1) osv.

Det er viktig ***ikke*** å blande gruppering (**GROUP BY**) med sortering (**ORDER BY**). Resultatet i Figur 6-43 kan, om ønskelig, i tillegg sorteres på f.eks. kolonnen det telles forekomster i. Dette kan gjøres som vist i Figur 6-44.

```
SELECT CarMake, COUNT(*) AS [Number of cars from each CarMake]
FROM CAR
GROUP BY CarMake /* Radene grupperes ut fra CarMake-kolonnen */
ORDER BY COUNT(*) ASC /* Grupperingen sorteres stigende på antall */
```

**Figur 6-44: Det grupperte resultatet sortert stigende på antall biler.**

Resultatet sortert på antall er vist i Figur 6-45. **ORDER BY** må da plasseres ***etter*** **GROUP BY**.

Results | Messages

| | CarMake | Number of cars from each CarMake |
|---|---|---|
| 1 | Honda | 1 |
| 2 | Mazda | 1 |
| 3 | Mercedes | 1 |
| 4 | Nissan | 1 |
| 5 | Toyota | 1 |
| 6 | Audi | 2 |
| 7 | Peugeot | 3 |
| 8 | Volvo | 4 |

**Figur 6-45: Resultatet av grupperingen sortert på antall biler i hver gruppe.**

På tilsvarende måte kan andre aggregeringsfunksjoner benyttes på grupperinger. Dersom det ønskes å finne gjennomsnittprisen for bilene i hver gruppe, gjennomsnittspris for Hondaer, for Mazdaer osv., kan dette gjøres som vist i Figur 6-46. Prisen er formatert med norsk valutaformat.

```
SELECT CarMake, FORMAT(AVG(Price),'C','no') AS [Average price for each CarMake]
 FROM CAR
 GROUP BY CarMake /* Radene grupperes ut fra CarMake-kolonnen */
ORDER BY AVG(Price) ASC /* Grupperingen sorteres stigende på antall */
```

**Figur 6-46: Gjennomsnittspris for hvert bilmerke.**

Et utdrag av resultatet er vist i Figur 6-47.

Results | Messages

| | CarMake | Average price for each CarMake |
|---|---|---|
| 1 | Volvo | kr 230 000,00 |
| 2 | Mazda | kr 320 000,00 |
| 3 | Mercedes | kr 350 000,00 |

**Figur 6-47: Gjennomsnittspris for hvert bilmerke, med norsk valutaformat og sortert stigende på pris.**

Det er også mulig å gruppere på flere kolonner. Som et eksempel på dette skal det først grupperes på bilmerke, sånn at hvert bilmerke håndteres på en gruppe. Deretter skal det innenfor hver gruppe grupperes på farge. Målet er å få en oversikt over hvor mange det innenfor hvert bilmerke er biler av fargen **Blue**, **Red** osv. I tillegg til kolonnene benyttes derfor også **COUNT**. Spørringen er vist i Figur 6-48.

```
SELECT CarMake, Color, COUNT(*) AS [Number of cars with this color for the given CarMake]
 FROM CAR
 GROUP BY CarMake, Color /* Radene grupperes ut fra CarMake-kolonnen */
ORDER BY CarMake ASC /* Grupperingen sorteres stigende på antall */
```

**Figur 6-48: Gruppering på to kolonner, CarMake og Color.**

Et utsnitt av resultatet er vist i Figur 6-49. I dette utsnittet ses det at det for Peugeot er 2 stk. blå og 1 grønn. Før øvrige biler er det ikke registrert flere biler med samme farge.

Results | Messages

| | CarMake | Color | Number of cars with this color for the given CarMake |
|---|---|---|---|
| 1 | Audi | Black | 1 |
| 2 | Audi | Red | 1 |
| 3 | Honda | Blue | 1 |
| 4 | Mazda | Red | 1 |
| 5 | Mercedes | Silver | 1 |
| 6 | Nissan | NULL | 1 |
| 7 | Peugeot | Blue | 2 |
| 8 | Peugeot | Green | 1 |
| 9 | Toyota | Silver | 1 |

**Figur 6-49: Utsnitt fra resultatet av en spørring gruppert på to kolonner.**

**NB!** Når det grupperes på flere kolonner, må ***alle*** kolonner som er med i grupperingen også tas med i **SELECT**-delen. I annet fall gir SQL Server feilmelding ved kompilering. I Figur 6-48 ses det at kolonnene **CarMake** og **Color** er med både i grupperingen og i **SELECT**-delen.

## 6.13. HAVING

Instruksjonen **HAVING** benyttes på grupperinger omtrent som **WHERE** på rader. **HAVING** forutsetter derfor at det er gjort en gruppering, og brukes deretter til å filtrere resultatet.

I Figur 6-46 ble det laget en spørring som først grupperte på bilmerke (**CarMake**) og så fant gjennomsnittsprisen for biler innenfor hvert bilmerke. Dersom det var ønskelig å vise ***kun*** de bilmerkene der gjennomsnittsprisen var f.eks. minimum kr 500 000 eller mer, kan en **HAVING**-betingelse legges til ***etter*** grupperingen. Dette er vist i Figur 6-50.

```
SELECT CarMake, FORMAT(AVG(Price),'C','no') AS [Gjennomsnittspris for hvert bilmerke]
FROM CAR
GROUP BY CarMake /* Radene grupperes ut fra CarMake-kolonnen */
HAVING AVG(Price) >= 500000 /* Filtrerer etter at grupperingen er utført */
ORDER BY AVG(Price) ASC /* Grupperingen sorteres stigende på antall */
```

**Figur 6-50: Spørring der HAVING benyttes til å filtrere etter grupperingen.**

Grupperingsresultatet i Figur 6-47 viser at kun to av gruppene innfrir kravet til en gjennomsnittpris større eller lik kr 500 000. Resultatet av **HAVING**-spørringen blir derfor som vist i Figur 6-51.

Results | Messages

| | CarMake | Gjennomsnittspris for hvert bilmerke |
|---|---|---|
| 1 | Nissan | kr 520 000,00 |
| 2 | Audi | kr 565 000,00 |

**Figur 6-51: Resultat av spørringen etter bruk av HAVING-betingelse etter gruppering.**

**NB!** Eventuell **ORDER BY** (sortering) må gjøres ***etter*** **HAVING**, som i Figur 6-50.

Selv om **HAVING** benyttes, er det også mulig å bruke **WHERE**. Husk da at **WHERE** filtrerer rader, hvilket må gjøres ***før*** gruppering. **HAVING** filtrerer deretter ***etter*** **GROUP BY**.

Som et eksempel skal nå først **WHERE** benyttes til å filtrere rader, så alle rader av bilmerket Nissan fjernes fra utvalget. Dette kan gjøres sånn: **WHERE CarMake <> "Nissan"**.

Deretter utføres gruppering på samme måte som Figur 6-50. Det eneste som vil skille disse to spørringene er altså at det legges inn en **WHERE**-betingelse før grupperingen.

Den nye spørringen er vist i Figur 6-52.

```
SELECT CarMake, FORMAT(AVG(Price),'C','no') AS [Gjennomsnittspris for hvert bilmerke]
FROM CAR
WHERE CarMake <> 'Nissan' /* Filtrerer bort alle rader med Nissan som bilmerke */
GROUP BY CarMake /* Radene grupperes ut fra CarMake-kolonnen */
HAVING AVG(Price) >= 500000 /* Filtrerer etter at grupperingen er utført */
ORDER BY AVG(Price) ASC /* Grupperingen sorteres stigende på antall */
```

**Figur 6-52: Spørring der både WHERE og HAVING benyttes.**

Siden «Nissan» er filtrert bort fra radene ***før*** grupperingen utføres, vil resultatet nå bli kun én rad, i motsetning til resultatet vist i Figur 6-51, som inneholdt to rader.

Det nye resultatet er vist i Figur 6-53.

Results | Messages

| | CarMake | Gjennomsnittspris for hvert bilmerke |
|---|---|---|
| 1 | Audi | kr 565 000,00 |

**Figur 6-53: Resultat av spørringen vist i Figur 6-52 der både WHERE og HAVING er benyttet.**

## 6.14. Delspørring (Subquery)

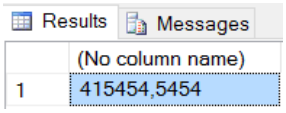
Det er mulig å inkludere delspørringer i en spørring. Disse kan enten være selvstendige eller ha en korrelerende effekt med hovedspørringen. Det ses først på bruk av selvstendige delspørringer.

Det er tidligere vist at gjennomsnittsprisen for alle bilene kan finnes med **AVG**-funksjonen. I Figur 6-54 er det vist en slik spørring.

```
SELECT AVG(PRICE)
FROM CAR
```

**Figur 6-54: Spørring som finner gjennomsnittsprisen for alle bilene.**

Resultatet av spørringen vist i Figur 6-55. Som det ses av figuren returnerer denne spørringen eksakt ***én*** verdi og ikke et sett med rader.




**Figur 6-55: Beregnet gjennomsnittspris for samtlige biler i tabellen CAR.**

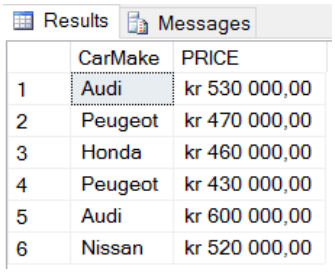
Dersom det er ønskelig å generere en liste over alle biler som har en verdi **større eller lik** gjennomsnittsverdien for samtlige biler, kan spørringen vist i Figur 6-54 benyttes som en delspørring.

Delspørringen regnes da først ut som en selvstendig verdi. Hovedspørringen vil deretter rad for rad sjekke verdien i **Price**-kolonnen mot resultatet av delspørringen. Spørringen er vist i Figur 6-56.

```
SELECT CarMake, FORMAT(Price,'C','no') AS Price /* Price formateres med valutaformat */
FROM CAR
WHERE Price >= /* Sammenligner prisen rad for rad med gjennomsnittsprisen */
    (SELECT AVG(PRICE) FROM CAR) /* Selvstendig delspørring som returnerer snittprisen */
```

**Figur 6-56: Spørring som inkluderer en selvstendig delspørring**

Resultatet (Figur 6-57) blir alle rader der **Price** er større eller lik gjennomsnittsprisen (415454,5454).




**Figur 6-57: Retur av alle rader der «Price»-verdien er større enn gjennomsnittsprisen.**

Det er også mulig å ha «korrelerte delspørringer» (engelsk: correlated subqueries), noe det ses nærmere på i kapittel 8.8.

# 7. Kobling av flere tabeller

I alle spørringene til nå er det kun benyttet én tabell. I de fleste praktiske tilfeller vil det inngå mange tabeller og ofte vil det være behov for spørringer der flere av disse inngår.

Istedenfor å plassere alle data i én stor tabell er det vanlig å dele opp dataene i mindre tabeller med data som «passer sammen». I eksemplene til nå er det benyttet en tabell med data om biler. Denne tabellen inneholder et felt kalt **OwnerId**.

Dersom det ønskes flere data om brukeren enn bare **OwnerId**, f.eks. fornavn, etternavn, adresse osv., vil dette være data det er naturlig å plassere i en egen tabell.

## 7.1. Fremmednøkkel

For å registrere data om bilenes eiere skal det lages en ny tabell. For eksempelets skyld skal det bare lagres bileiernes fornavn, etternavn, fødselsdata og postnummer. Som attributter/kolonnenavn for disse velges **FName**, **LName**, **DateOfBirth** og **PostalNumber**. I tillegg behøves en primærnøkkel som unikt kan identifisere enhver rad i tabellen. Tabellen gis navnet **OWNER**.

Som primærnøkkel velges et attributt med navn **OwnerId**. Merk at dette er samme attributt som også finnes i tabellen **CAR** og skal være koblingen mellom de to tabellene.

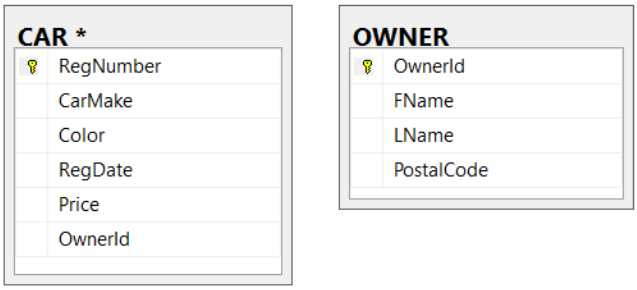
Når samhørende data som dette skal fordeles på to tabeller, benyttes en **fremmednøkkel**. Fremmednøkkelen, som plasseres i én av tabellene, fungerer da som en **referanse** fra den ene tabellen til den andre tabellen, ved at den refererer til (peker til) primærnøkkelen i den andre tabellen. På denne måten kan man via dette/disse attributtene få tilgang til alle samhørende data i de to tabellene.

Det er krav til at fremmednøkkelen må ha samme **datatype** som feltet/feltene den peker til. Tabellen kan lages med **CREATE TABLE**-instruksjonen, på samme måte som i kapittel 4.1. SQL-spørringen for dette er vist i Figur 7-1.

```
CREATE TABLE OWNER
(
    OwnerId int NOT NULL,
    FName varchar(30) NULL,
    LName varchar(30) NULL,
    PostalCode int NULL,
    CONSTRAINT PK_OWNER PRIMARY KEY (OwnerId)
)
```

**Figur 7-1: Tabellen OWNER lages med SQL.**

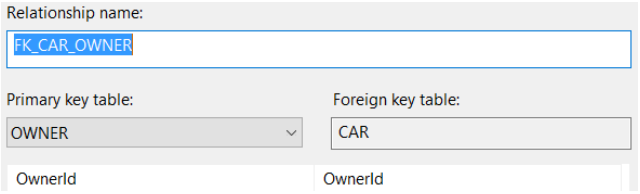
De to tabellene som nå er laget er i Figur 7-2 vist i SQL Server sin diagramvisning.




**Figur 7-2: De to tabellene CAR og OWNER vist i diagramvisning.**

## 7.2. Definere fremmenøkkel via det grafiske brukergrensesnittet

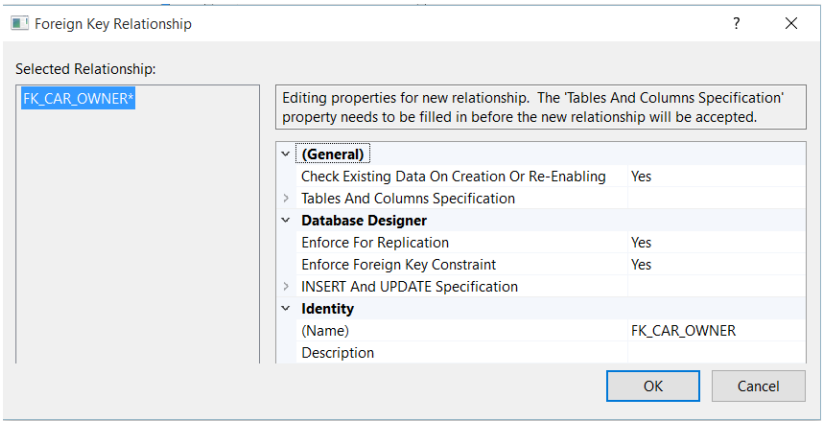
Fremmednøkkel kan defineres på to måter, enten via det grafiske grensesnittet eller med SQL. I det grafiske grensesnittet (vist i Figur 7-2) kan dette gjøres ved å plassere kursoren over **OwnerId** i tabellen **CAR** (der **OwnerId** skal være fremmednøkkel). Hold så venstre musetast nede og dra kursor over til **OwnerID** i **OWNER** (som er primærnøkkelen fremmednøkkelen skal «peke til»/«være referanse til»). Når kursoren slippes, åpnes vinduet vist i Figur 7-3.




**Figur 7-3: Generering av fremmednøkkel.**

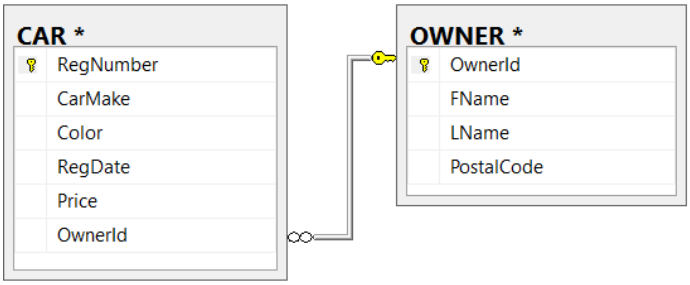
«Relationship name» er et valgfritt navn. Forslaget «**FK_CAR_OWNER**» benyttes som navnestandard i dette kurset. **FK** angir at det er en fremmednøkkel, **CAR** angir at tabellen det pekes fra (altså tabellen fremmednøkkelen er i) og **OWNER** er tabellen fremmednøkkelen refererer til (altså «**FK_fra tabell_til tabell**»). Alternativt kan det brukes bare «**FK_til tabell**» (**FK_OWNER**), men benytt én av metodene konsekvent, så det er lett å vite hva referansen er, om f.eks. en nøkkel senere skal slettes.

Ved klikk på «OK» kommer dialogboksen «Foreign Key Relationship» som vist i Figur 7-4.




**Figur 7-4: Definisjon av regler for fremmednøkkelen.**

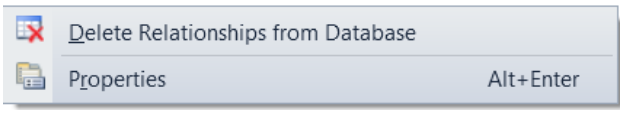
Det behøves ikke gjøres noen endringer av foreslåtte innstillinger. Det ønskes at det skal sjekkes mot eventuelle data i eksisterende tabeller, så det sikres at disse innfrir kravene til en fremmednøkkel. Trykk OK og fremmednøkkelen vil illustreres grafisk, som vist i Figur 7-5.




**Figur 7-5: Tabellene CAR og OWNER i diagramvisning, med fremmednøkkel påført.**

Nøkkelsymbolet indikerer primærnøkkelen og uendelig-symbolet (∞) indikerer fremmednøkkelen.

Ved å høyreklikke over fremmednøkkellinjen fremkommer en meny der det, om ønskelig, er mulig å slette fremmednøkkelen. Se Figur 7-6. Via den samme menyen kan også «Properties» velges, der egenskapene tidligere vist i Figur 7-4 ved behov kan endres.




**Figur 7-6: Ved høyreklikk over fremmednøkkellinjen gis det mulighet for å slette fremmednøkkelen.**

**NB!** Slett fremmednøkkelen som er laget, da den i neste kapittel skal lages på nytt med SQL.

## 7.3. Lage tabell med fremmednøkler med SQL

I Figur 7-1 ble det laget en primærnøkkel i tabellen **OWNER** med nøkkelordet **CONSTRAINT**. På tilsvarende måte kan **CONSTRAINT** brukes til angivelse av en fremmednøkkel. Syntaksen er sånn:

```
CONSTRAINT identifikator FOREIGN KEY (fremmednøkkel) REFERENCES tabell (primærnøkkel)
```

I vårt tilfelle blir dette sånn:

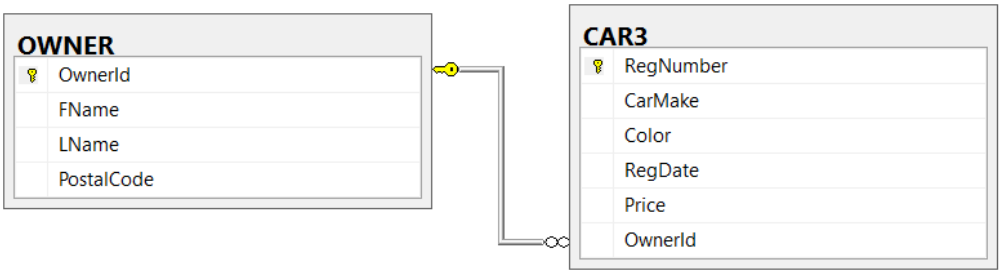
```
CONSTRAINT FK_CAR_OWNER FOREIGN KEY (OwnerId) REFERENCES OWNER (OwnerID)
```

Dersom hele tabellen **CAR** ønskes laget ***med*** primærnøkkel og fremmednøkkel, vil **CREATE TABLE**-spørringen bli som vist i Figur 7-7. Siden dette nå er for testformål, og det tidligere er opprettet noen **CAR**-tabeller, kalles denne for **CAR3** for å unngå konflikt med eventuell eksisterende tabell.

```
CREATE TABLE CAR3
(
    RegNumber varchar (7) NOT NULL,
    CarMake varchar(30) NULL,
    Color varchar(20) NULL,
    RegDate date NULL,
    Price money NULL,
    OwnerId int NULL,
    CONSTRAINT PK_CAR3 PRIMARY KEY (RegNumber),
    CONSTRAINT FK_CAR3_OWNER FOREIGN KEY (OwnerId) REFERENCES OWNER(OwnerID)
)
```

**Figur 7-7: CREATE TABLE-spørring med definisjon av primær- og fremmednøkkel.**

Diagramvisning av de to tabellene vil vise fremmednøkkelen mellom dem, som vist i Figur 7-8.




**Figur 7-8: Tabellene etter at fremmednøkkel er definert i OWNER-tabellen.**

Dersom en tabell allerede ***er*** definert, og det i ettertid ønskes å legge til en fremmednøkkel, kan dette gjøres med **ALTER TABLE**-metoden, som vist i Figur 7-9.

```
ALTER TABLE CAR
ADD CONSTRAINT FK_CAR_OWNER FOREIGN KEY (OwnerId) REFERENCES OWNER(OwnerID)
```

**Figur 7-9: ALTER TABLE benyttes til å legge til en fremmednøkkel til en eksisterende tabell.**

Dersom det kommer feilmelding knyttet til at det allerede ***foreligger*** en nøkkel med identifikator **FK_CAR_OWNER**, kommer feilmeldingen vist i Figur 7-10

```
Messages
Msg 2714, Level 16, State 5, Line 1
There is already an object named 'FK_CAR_OWNER' in the database.
```

**Figur 7-10: Feilmelding dersom nøkkelidentifikatoren allerede finnes.**

Da må denne først slettes, hvilket kan gjøres med SQL-setningen vist i Figur 7-11.

```
ALTER TABLE OWNER
DROP CONSTRAINT FK_CAR_OWNER /* Sletter eventuell fremmednøkkel med dette navn */
```

**Figur 7-11: Sletting av eventuelt eksisterende nøkkelidentifikator med navn FK_CAR_OWNER.**

Kjøres SLQ-spørringen i Figur 7-9 på nytt, vil det komme enda en feilmelding, som vist i Figur 7-12.

```
Messages
Msg 547, Level 16, State 0, Line 1
The ALTER TABLE statement conflicted with the FOREIGN KEY constraint "FK_CAR_OWNER".
The conflict occurred in database "CarDatabase", table "dbo.OWNER", column 'OwnerId'.
```

**Figur 7-12: Feilmelding som følge av brudd på reglene for fremmednøkler.**

Årsaken er reglene som gjelder for fremmednøkler (**referanseintegritet**), som blant annet sier at en fremmednøkkel ***må*** peke til (være referanse til) noe som eksisterer, dersom den peker til noe. Begrepet referanseintegritet forklares nærmere i kapittel 7.4.

Tabellen **CAR** inneholder i sitt fremmednøkkelfelt (**OwnerId**) verdiene 1, 2, 3, 4 og 6 (jf. Figur 2-5), som ***ikke*** finnes i feltet **OwnerId** i **OWNER**. Derfor gis det feilmelding for brudd på reglene.

For å legge inn verdier i tabellen **OWNER** via SQL Servers grafiske brukergrensesnitt, høyreklikk over tabellen i Object Explorer og velg «**Edit Top 200 Rows**». Legg der inn verdiene vist i Figur 7-3.

| OwnerId | FName | LName | PostalCode |
|---|---|---|---|
| 1 | Per | Olsen | 3740 |
| 2 | Frank | Pettersen | 0040 |
| 3 | Lise | Gulliksen | 3603 |
| 4 | Lena | Lekven | 4007 |
| 5 | Hans | Solvik | 3603 |
| 6 | Gunn | Helgesen | 3740 |

**Figur 7-13: Innlegging av data i "edit"-modus.**

Nå eksisterer verdiene det refereres til fra tabellen **CAR**, og det tillates da å lage fremmednøkkelen fra **OwnerId** i **CAR** til **OwnerId** i **OWNER**. SQL-spørringen vist i Figur 7-9 skal derfor nå virke.

## 7.4. Referanseintegritet (Reference Integrity) – Gjelder fremmednøkkel

I kapittel 2.7 ble begrepet **entitetsintegritet** introdusert, hvilket gjaldt **primærnøkler**.

**Referanseintegritet** gjelder **fremmednøkler** og sikres av DBMS-systemet. I kapittel 7.3 ble det vist et praktisk eksempel med brudd på referanseintegritetsregelen, med påfølgende feilmelding. Det ble forsøkt å referere til bileiere som det ennå ikke var lagret noen data om.

Referanseintegritet brukes til å sikre at bare gyldige data legges inn i databasen i fremmednøkkelfelt. Det gir f.eks. ikke mening å referere til data om en bileier, når det ikke er registrert noen data om denne. Det gis derfor en feilmelding dersom en slik referering forsøkes utført.

**Grunnregler for referanseintegritet:**

- Fremmednøkkelfeltet og primærnøkkelfeltet dette peker til, må ha samme datatype.
- Registreres det en verdi i et fremmednøkkelfelt, ***må*** denne verdien ***først*** være registrert i primærnøkkelfeltet referansen peker til.
- Det er tillatt at hele fremmednøkkelen inneholder **NULL**-merker. Dette betyr da at det ***ikke*** refereres til noen verdi. Består fremmednøkkelen av flere felt, må enten alle være **NULL** eller alle ha verdier. Dette fordi en primærnøkkel det pekes til ikke kan inneholde **NULL**-merker.

Disse reglene vil også hindre sletting og endring av data i primærnøkkelfelt dersom det finnes fremmednøkkelverdier som peker til disse.

## 7.5. Redundans

Redundans er et begrep i databaser på overflødige data. Årsaken til at det er viktig ikke å registrere overflødige (redundante) data er blant annet at det kan:

- føre til lagring av mer data enn nødvendig
- føre til registreringsfeil når data legges inn
- vanskeliggjøre søk etter data

Det kan se ut som lagring av de samme dataene i både et **primærnøkkelfelt** og et **fremmednøkkelfelt** er unødvendig dobbeltlagring av data, men dette er data som er nødvendige for å bevare sammenhengen mellom data ved oppsplitting i flere tabeller. De er derfor ikke **redundante**, men nødvendig.

Et eksempel på **redundans** er å lagre en pris både ***med*** og ***uten*** merverdiavgift. Årsaken er at den ene prisen kan avledes av den andre. Dersom disse lagres i to kolonner, må begge verdier holdes oppdatert i forhold til hverandre, hvilket kan gi risiko for oppdateringsfeil.

Et annet eksempel på redundans er å registrere to verdier (i samme tabell) som er funksjonelt avhengig av hverandre. At de er funksjonelt avhengig, vil si at én bestemt verdi (eller kombinasjon av verdier) i det ene feltet ***alltid*** gir en og samme verdi i det andre feltet/feltene.

Én spesifikk emnekode på et studium vil for eksempel alltid gi samme emnenavn. Dersom det derfor registreres karakterer for kandidater, og det for hver gang det registreres en emnekode gjentas respektive emnenavn, er dette **redundant** informasjon. Da bør det heller lages en egen tabell med emnekoder og respektive emnenavn og heller ved behov lage fremmednøkler til emnekoden.

En parallell til dette skal nå legges inn i vårt eksempel. I tabellen **OWNER** er det lagt inn et felt kalt **PostalCode**, der postnummeret registreres. Hadde det i tillegg vært lagt til et felt kalt **City**, ville bynavnet blitt repetert hver gang samme postnummer ble gjentatt. Dette er **redundante** data.

For å unngå dette, skal det opprettes en egen tabell kalt **POSTALADDRESS**. Fra tabellen **OWNER** skal det så legges til en fremmednøkkel fra feltet **PostalCode** til **PostalCode** i **POSTALADDRESS**.

Tabellen **POSTALADDRESS** kan lages som vist i Figur 7-14.

```
CREATE TABLE POSTALADDRESS
(
    PostalCode char(4),
    City Char(35),
    CONSTRAINT PK_POSTALADRESS PRIMARY KEY (PostalCode)
)
```

**Figur 7-14: Lager en ny tabell for lagring av postnummer og poststed.**

Det skal så lages en fremmednøkkel fra feltet **PostalCode** i tabellen **OWNER** til **PostalCode** i tabellen **POSTALADDRESS**. I **OWNER**-tabellen vist i Figur 7-13 ses det at det ble lagt inn postnumrene '3740', '0040', '3603, og '4007'. Som konsekvens av referanseintegritetsreglene, vil det komme feil-melding dersom det lages en fremmednøkkel til til **PostalCode** i tabellen **POSTALADDRESS**, uten at disse verdiene der allerede finnes i dennes primærnøkkel.

Høyreklikk derfor over den nye tabellen **POSTALADDRESS**, velg «Edit Top 200 Rows» og legg inn verdiene vist i Figur 7-15.

KONTOR-PC.CarDa...o.POSTALADDRESS

| | PostalCode | City |
|---|---|---|
| | 3740 | SKIEN |
| | 0040 | OSLO |
| | 3603 | KONGSBERG |
| | 4007 | STAVANGER |

**Figur 7-15: Tabellen POSTALADRESS med noen innlagte radverdier.**

Nå skal det være mulig å legge inn den aktuelle fremmednøkkelen i tabellen **OWNER**, ved å utføre spørringen vist i Figur 7-16.

```
ALTER TABLE OWNER
ADD CONSTRAINT FK_OWNER_POSTALADDRESS FOREIGN KEY (PostalCode)
               REFERENCES POSTALADDRESS (PostalCode)
```

**Figur 7-16: SQL-kode for å legge til en fremmednøkkel i tabellen OWNER.**

I prosjektet er det nå tre tabeller, **CAR**, **OWNER** og **POSTALADDRESS**. Det er fremmednøkkel fra feltetet **OwnerId** i tabellen **CAR** til primærnøkkelen **OwnerID** i tabellen **OWNER**. I tillegg er det fremmednøkkel fra feltet **PostalCode** i tabellen **POSTALADDRESS** til primærnøkkelen **PostalCode** i tabellen **OWNER**. Sammenhengene er vist i diagramvisningen i Figur 8-1 i neste kapittel.

## 7.6. Fremmednøkler med angivelse av DELETE eller CASCADE

Grunnet **referanseintegritet** må en **fremmednøkkel** enten ***ikke*** peke til noe (bare ha **NULL**-merker) eller peke til en **primærnøkkel** i en annen tabell. Verdien det pekes til ***må*** i så fall eksistere, i annet fall gir DBMS feilmelding. DBMS passer på at disse reglene følges.

Et problem som likevel kan inntreffe, er at verdien det pekes til i den andre tabellene senere slettes eller endres. Da vil ikke lenger kravet til referanseintegritet være innfridd. Ved opprettelse av fremmednøkler kan det angis hvordan slike tilfeller skal håndteres. Det er to ting som kan skje, enten at verdien det pekes til slettes eller at den endres. Med «**ON DELETE**» og «**ON UPDATE**» kan disse to situasjonene håndteres.

**Ved sletting av primærnøkkelverdien det pekes til:**

**ON DELETE CASCADE**: Slettes primærnøkkelen det pekes til, slettes også fremmednøkkelen.

**Ved endring av primærnøkkelverdien det pekes til:**

**ON UPDATE** [**CASCADE**] [**SET NULL**] [**SET DEFAULT**]

- **CASCADE**: endrer fremmednøkkelen til samme verdi som primærnøkkelen endres til
- **SET NULL**: Fremmednøkkelen settes til **NULL**, så den ikke lenger peker til noe
- **SET DEFAULT**: Fremmednøkkel settes til en forhåndsdefinert verdi, som det da må sikres at ligger som verdi i primærnøkkelfeltet

I figur Figur 7-17 er tabellen fra Figur 7-7 laget på nytt med **ON UPDATE**- og **ON DELETE**-angivelser for fremmednøkkelen. Det er valgfritt om dette skal gjøres.

```
CREATE TABLE CAR3
(
    RegNumber varchar (7) NOT NULL,
    CarMake varchar(30) NULL,
    Color varchar(20) NULL,
    RegDate date NULL,
    Price money NULL,
    OwnerId int NULL,
    CONSTRAINT PK_CAR3 PRIMARY KEY (RegNumber),
    CONSTRAINT FK_CAR3_OWNER FOREIGN KEY (OwnerId) REFERENCES OWNER(OwnerID)
    ON DELETE CASCADE
    ON UPDATE CASCADE
)
```

**Figur 7-17: Tabellopprettelse med angivelse av «constraints» for fremmednøkkelen.**

Dersom ovennevnte ønskes testet, kan det lages en ny tabell. Dersom det allerede finnes en **CAR3**-table, så lag heller en **CAR4**, **CAR5** eller tilsvarende.

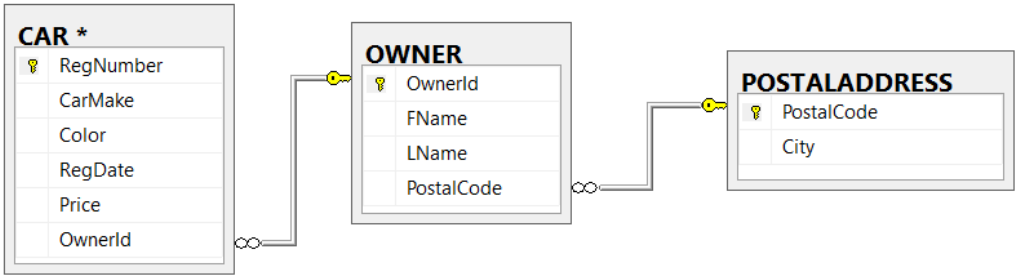
Fines det allerede en **CAR**-table uten nevnte restriksjoner, må dennes fremmednøkkel til **OWNER** først slettes. I annet fall vil det komme en feilmelding grunnet eksisterende referanseentigritetsbegrensninger, når det gjøres endring i **OWNER**-tabellen. Selve tabellen bør ikke slettes, bare fremmednøkkelen mellom denne og **OWNER**-tabellen.

**Test:** Legg inn en rad i den nye **CAR**-tabellen med **OwnerId** satt til nr. 1 i fremmednøkkelfeltet. Når dette er utført, endre da for eksempel **OwnerId** nr 1 i **OWNER**-tabellen til nr. 1000. Sjekk deretter om verdien fremmednøkkelfeltet i den nye **CAR**-tabellen automatisk har endret seg til 1000 (grunnet **CASCADE**).

# 8. Spørringer der flere tabeller inngår

I kapittel 6 ble de vanlige SQL-instruksjonene for uthenting av data fra én tabell gjennomgått. Disse gjelder også når det skal lages spørringer mot flere tabeller, men da kommer det en del i tillegg.

Eksempelet fra de tidligere kapitlene videreføres med de tre tabellene vist i Figur 8-1. I disse tre tabellene er all **redundans** fjernet. Mer om dette kommer i kapittelet om **normalisering**.




**Figur 8-1: De tre tabellene påført primær- og fremmednøkler.**

Når tabeller skal koble, er det for oversiktens skyld fornuftig å ha et diagram, som i Figur 8-1, lett tilgjengelig. Dette vil gjøre det enklere å se hvilke koblingsbetingelser som behøves i spørringer.

## 8.1. Refresh av IntelliSense

Innledningsvis vil det gjøres oppmerksom på at det av og til kommer noen feilmeldinger i SQL-koden i form av understrekninger under tabeller uten at det er noe feil i spørringen. Disse kan i såfall fjernes ved å utføre en «refresh» av **IntelliSense**. Ved mistanke om dette, utfør følgende menyvalg:

**Edit -> IntelliSense -> Refresh Local Cache**

## 8.2. Likekobling (med WHERE og INNER JOIN)

Dersom det skal lages en spørring der det ønskes i retur alle kolonnene fra tabellen **OWNER** og poststed (**City**) for angitt eier, er det behov for å lage en spørring mot de to tabellene **OWNER** og **POSTALADDRESS**.

Dersom en slik spørring lages med ***** i **SELECT**-delen, ***uten*** en koblingsbetingelse for de to tabellene, returneres ***alle*** kolonnene fra begge tabellene. I tillegg lages det for hver rad i den ene tabellen en kobling mot hver rad i den andre tabellen. Totalt antall rader blir derfor: **rader i tabell 1 * rader i tabell 2**. Dette kalles **kryssproduktet** eller det «**kartesiske produktet**».

En slik spørring er vist i Figur 8-2. Siden **OWNER** inneholder 6 rader og **POSTALADRESS** inneholder 4 rader, returneres totalt 24 rader. Figuren viser bare et lite utdrag av disse.

```
SELECT *
FROM OWNER, POSTALADDRESS
```

**Figur 8-2: Spørring vil returnere kryssproduktet av tabellene OWNER OG POSTALADDRESS.**

Et utdrag av de 24 radene i resultatet (kryssproduktet/det kartesiske produkt) er vist i Figur 8-3.

Results | Messages

| | OwnerId | FName | LName | PostalCode | PostalCode | City |
|---|---|---|---|---|---|---|
| 1 | 1 | Per | Olsen | 3740 | 0040 | OSLO |
| 2 | 2 | Frank | Pettersen | 0040 | 0040 | OSLO |
| 3 | 3 | Lise | Gulliksen | 3603 | 0040 | OSLO |
| 4 | 4 | Lena | Lekven | 4007 | 0040 | OSLO |
| 5 | 5 | Hans | Solvik | 3603 | 0040 | OSLO |
| 6 | 6 | Gunn | Helgesen | 3740 | 0040 | OSLO |
| 7 | 1 | Per | Olsen | 3740 | 3603 | KONGSBERG |
| 8 | 2 | Frank | Pettersen | 0040 | 3603 | KONGSBERG |

**Figur 8-3: Resultatet av krysskoblingen er på totalt 24 rader (6*4), men bare et utvalg er vist her.**

En slik spørring gir normalt liten mening, da det som oftest er ønskelig å filtrere kolonner/rader.

Med referanse til Figur 8-1, ses det at det som bør gjøres er å koble hver eiers postnummer med det respektive postnummeret i **POSTADDRESS**-tabellen, sånn at man via dette postnummeret kan hente ut riktig poststed (**City**). Dette kan gjøres ved å lage en likekobling mellom de to tabellene.

Når to tabeller **likekobles**, sjekkes verdiene i én spesifisert kolonne i en tabell mot verdiene i tilsvarende spesifiserte kolonne i den andre tabellen. For hver rad der det finnes en samsvarende verdi i koblingsfeltet i den andre tabellen, tas raden med i resultatet. Der det ikke finnes en samsvarende koblingsverdi, utelates raden fra resultatet.

Det er mulig å lage likekoblinger mellom **ikke-nøkkel**-felt, men i de aller fleste tilfeller er en likekobling en kobling mellom **primærnøkkel** i én tabell og **fremmednøkkel** i en annen tabell.

### 8.2.1. Likekobling av to tabeller med WHERE

En likekobling utført med en **WHERE**-betingelse gjøres med syntaksen vist i Figur 8-4.

```
SELECT kolonner
FROM TABELL1, TABELL2
WHERE TABELL1.koblingskolonne = TABELL2.Koblingskolonne
```

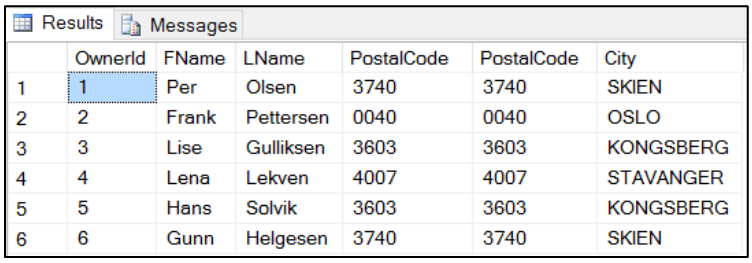
**Figur 8-4: Syntaks for likekobling med WHERE.**

Likekoblingen som skal lages, kan derfor med SQL skrives som vist i Figur 8-5.

```
SELECT *   /*Viser alle kolonnene fra begge tabellene */
FROM OWNER, POSTALADDRESS /* Tabellene som inngår i koblingen */
WHERE OWNER.PostalCode = POSTALADDRESS.PostalCode /* Likekobling */
```

**Figur 8-5: Likekobling mellom tabellene OWNER og POSTALADDRESS.**

Resultatet av spørringen er vist i Figur 8-6. Istedenfor 24 rader blir resultatet nå 6 rader, én for hver bileier. Verdiene fra **City**-kolonnen i tabellen **POSTALADDRESS** er også med.



Results | Messages

| | OwnerId | FName | LName | PostalCode | PostalCode | City |
|---|---|---|---|---|---|---|
| 1 | 1 | Per | Olsen | 3740 | 3740 | SKIEN |
| 2 | 2 | Frank | Pettersen | 0040 | 0040 | OSLO |
| 3 | 3 | Lise | Gulliksen | 3603 | 3603 | KONGSBERG |
| 4 | 4 | Lena | Lekven | 4007 | 4007 | STAVANGER |
| 5 | 5 | Hans | Solvik | 3603 | 3603 | KONGSBERG |
| 6 | 6 | Gunn | Helgesen | 3740 | 3740 | SKIEN |

**Figur 8-6: Resultatet av spørringen med likekobling.**

Det ses at kolonnen **PostalCode** er tatt med to ganger i resultatet, hvilket for så vidt er unødvendig. Dette kan unngås ved i **SELECT**-delen å angi hvilke kolonner som ønskes. Dersom det angis * for den ene, og det for den andre ikke tas med koblingskolonnen, vil koblingskolonnen vises kun én gang. En spørring som gir dette resultatet er vist i Figur 8-7.

```
SELECT OWNER.*, POSTALADDRESS.City  /*Viser alle koblingskolonnen kun én gang */
 FROM OWNER, POSTALADDRESS /* Tabellene som inngår i koblingen */
 WHERE OWNER.PostalCode = POSTALADDRESS.PostalCode /* Likekobling */
```

**Figur 8-7: Spørring der koblingskolonnen kun tas med én gang i resultatet.**

## 8.2.2. Likekobling av to tabeller med INNER JOIN

Det er i SQL-standarden laget en standardnotasjon spesifikt for indre likekobling, som et alternativ til bruk av **WHERE**. Syntaksen er som vist i Figur 8-8. Det benyttes da isteden **INNER JOIN**:

```
SELECT kolonner
FROM TABELL1 INNER JOIN TABELL2
ON TABELL1. koblingskolonne = TABELL2. koblingskolonne
```

**Figur 8-8: Generell syntaks for likekobling mellom to tabeller med «INNER JOIN».**

Denne syntaksen gir mulighet for å skrive spørringen vist i Figur 8-7 på den alternative måten med **INNER JOIN**, som vist i Figur 8-9.

```
SELECT *   /*Viser alle kolonnene fra begge tabellene */
 FROM OWNER INNER JOIN POSTALADDRESS /* Tabellene som inngår i koblingen */
 ON OWNER.PostalCode = POSTALADDRESS.PostalCode /* Likekobling */
```

**Figur 8-9: Likekobling mellom tabellene OWNER og POSTALADDRESS med «INNER JOIN».**

Resultatet av spørringen blir identisk med resultatet vist i Figur 8-6.

Ved visning av spørringer generert automatisk av SQL Server, vil det ses at **INNER JOIN** er benyttet. Selv om **WHERE**-likekoblinger nok kan synes noe enklere, er det derfor viktig å kjenne til begge disse syntaksene for likekobling. **JOIN** benyttes også i andre koblinger (**OUTER JOIN** og **FULL JOIN**) som det ses på senere, der det ikke finnes noen alternativ metode med **WHERE**.

## 8.2.3. Likekobling av tre tabeller med WHERE

Dersom det skal lages en spørring som gir en oversikt over hvilke biler de ulike bileeierne eier, og det ønskes tatt med data fra kolonnene **RegNumber**, **CarMake**, **OwnerID**, **LName** og **City**, må det lages en spørring som kobler **tre** tabeller. Dette kan ses av diagramfiguren vist i Figur 8-1.

Spørringen kan lages som en likekobling der **WHERE** benyttes til å koble tabellene. Det kan ikke benyttes flere **WHERE**-instruksjoner etter hverandre, men det kan isteden tilføyes flere betingelser til **WHERE**-delen ved å benytte **AND**-operatoren.

En spørring som kobler de tre tabellene, og returnerer angitte kolonner, er vist i Figur 8-10. Det er i spørringen også valgt å sortere resultatet på kolonnen **OwnerId**.

Merk at kolonnen **OwnerId** er angitt med «**TABELLNAVN**.kolonnenavn», dvs. **OWNER.OwnerID**. Når kolonner med samme navn finnes i flere tabeller, ***må*** tabellnavnangivelse benyttes.

```
SELECT OWNER.OwnerId, LName, City, RegNumber, CarMake
 FROM CAR, OWNER, POSTALADDRESS /* Tre tabeller inngår i koblingen */
 WHERE OWNER.PostalCode = POSTALADDRESS.PostalCode /* Likekobling */
 AND OWNER.OwnerId = CAR.OwnerId /* Likekobling */
ORDER BY Owner.OwnerId /* Sorterer resultatet på kolonnen OwnerId */
```

**Figur 8-10: Spørring som kobler tre tabeller.**

Resultatet av spørringen der de tre tabellene kobles, er vist i Figur 8-11. Det ses at resultatet er sortert på **OwnerId**, der det for hver eier listes opp hvilket poststed vedkommende er tilknyttet og hvilke biler den enkelte eier. For **OwnerId** nr 1 er det f.eks. tre rader, én for hver av bilene vedkommende eier. Det er i alt 14 biler i databasen, men kun 10 av dem er registrert med eiere.

Results | Messages

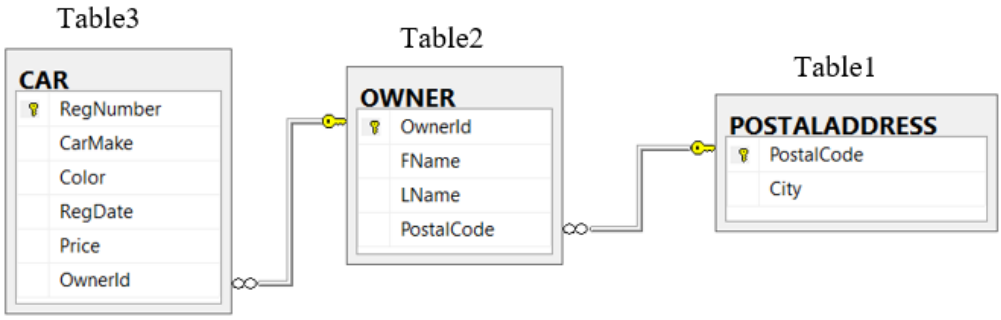
| | OwnerId | LName | City | RegNumber | CarMake |
|---|---|---|---|---|---|
| 1 | 1 | Olsen | SKIEN | DK88971 | Peugeot |
| 2 | 1 | Olsen | SKIEN | XX45999 | Peugeot |
| 3 | 1 | Olsen | SKIEN | YZ33892 | Peugeot |
| 4 | 2 | Pettersen | OSLO | YC43229 | Honda |
| 5 | 3 | Gulliksen | KONGSBERG | CF26702 | Mazda |
| 6 | 3 | Gulliksen | KONGSBERG | KC93332 | Toyota |
| 7 | 4 | Lekven | STAVANGER | FS33932 | Audi |
| 8 | 4 | Lekven | STAVANGER | NL50034 | Volvo |
| 9 | 4 | Lekven | STAVANGER | AA11111 | Audi |
| 10 | 6 | Helgesen | SKIEN | LY13335 | Mercedes |

**Figur 8-11: Resultat av spørring der data hentes fra tre tabeller.**

På tilsvarende vis kan flere tabeller enn tre **likekobles**, ved å tilføre flere likekoblinger med **AND**-operatoren i **WHERE**-delen av spørringen.

## 8.2.4. Likekobling av tre tabeller med INNER JOIN

På tilsvarende måte som med **WHERE**, kan det lages flere enn to likekoblinger med **INNER JOIN**. For å lette arbeidet er det lurt på nytt å betrakte diagrammet vist i Figur 8-12. **Table1**, **Table2** og **Table3** er påført for å lettere å se hvordan koblingene må lages.




**Figur 8-12: De tre tabellene i designvisning, med relasjoner påført.**

En generell syntaks for likekobling mellom tre tabeller med **INNER JOIN** kan skrives som vist i Figur 8-13. **TABLE1** er tabellen med primærnøkkelkoblingsfeltet ytterst til høyre i lenken.

```
SELECT * FROM TABLE1
INNER JOIN TABLE2 ON TABLE1.PrimaryKey = TABLE2.ForeignKey
INNER JOIN TABLE3 ON TABLE2.PrimaryKey = TABLE3.ForeignKey
```

**Figur 8-13: Generell syntaks for INNER JOIN-kobling med tre tabeller.**

Av Figur 8-12 ses det at **POSTALADDRESS** kan betraktes om **TABLE1**, **OWNER** som **TABLE2** og **CAR** som **TABLE3**. Da kan spørringen lages som vist i Figur 8-14.

```
SELECT OWNER.OwnerId, LName, City, RegNumber, CarMake
FROM POSTALADDRESS
INNER JOIN OWNER ON POSTALADDRESS.PostalCode = OWNER.PostalCode
INNER JOIN CAR ON OWNER.OwnerId = CAR.OwnerId
ORDER BY OWNER.OwnerId /* Sorterer resultatet på kolonnen OwnerId */
```

**Figur 8-14: INNER JOIN-kobling med tre tabeller.**

Resultatet av spørringen blir identisk med resultatet vist i Figur 8-11. På tilsvarende vis kan flere tabeller likekobles ved å tilføre flere **INNER JOIN**-koblinger.

## 8.3. Bruk av alias

Lange tabellnavn kan erstattes av **alias**. Disse defineres samtidig som tabellene introduseres i **FROM**-delen. Nøkkelordet **AS** benyttes til å angi **alias**. Nøkkelordet **AS** kan i **SQL Server** droppes, og definisjonen vil likevel virke. Siden **AS** er en del av standard SQL, anbefales det tatt med.

Spørringen fra Figur 8-14 er i Figur 8-15 omskrevet med bruk av **aliaser**. Selv om **aliasene** først introduseres i **FROM**-delen, ***må*** de benyttes allerede fra og med **SELECT**-delen av spørringen. Etter at de er introdusert, kan ***ikke*** lenger de opprinnelige tabellnavene benyttes ***noe*** sted i spørringen.

```
SELECT O.OwnerId, LName, City, RegNumber, CarMake
FROM POSTALADDRESS AS P
INNER JOIN OWNER AS O ON P.PostalCode = O.PostalCode
INNER JOIN CAR AS C ON O.OwnerId = C.OwnerId
ORDER BY O.OwnerId
```

**Figur 8-15: Spørring der aliaser er brukt for de tre tabellnavene som inngår i spørringen.**

## 8.4. Gruppering (GROUP BY) i spørring mot flere tabeller

**GROUP BY**, som i kapittel 6.12 ble benyttet i spørringer mot én tabell, kan også benyttes i spørringer mot flere tabeller. I Figur 8-11, ses det at det for mange bileiere er registrert flere rader. Dette betyr at disse bileierne eier flere biler. I tillegg har hver bileier et unikt **OwnerId**, et etternavn (**LName**) og en postadresse (**City**). Det skal derfor nå lages en spørring der man finner antallet biler hver eier har, og der verdier for kolonnene **OwnerId**, **LName**, **City** og «**Number of cars**» skal returneres for hver eier.

For å finne antall biler, må resultatet grupperes på kolonnen **OwnerId**. Siden det i tillegg til **OwnerId**, ønskes vist **LName** og **City**, må disse med i grupperingsdelen. Selv om det her ikke er behov for å gruppere videre på disse, stiller SQL krav at alle kolonner i **SELECT**-delen må med i grupperingen.

I tillegg må det i **SELECT**-delen legges inn telling av antall forekomster for hver gruppe. Dette gjøres med **COUNT**-funksjonen. Det ønskes også kolonneoverskriften «**Number of cars**» for denne kolonnen. Resultatet sorteres til slutt på feltet **OwnerId**. Spørringen er vist i Figur 8-16.

```
SELECT O.OwnerId, LName, City, COUNT(*) AS [Number of cars]
FROM POSTALADDRESS P
INNER JOIN OWNER O ON P.PostalCode = O.PostalCode
INNER JOIN CAR C ON O.OwnerId = C.OwnerId
GROUP BY O.OwnerId, LName, City
ORDER BY O.OwnerId ASC
```

**Figur 8-16: Spørring som viser informasjon om eierne og antall biler de eier.**

Resultatet av spørringen er vist i Figur 8-17.

Results | Messages

| | OwnerId | LName | City | Number of cars |
|---|---|---|---|---|
| 1 | 1 | Olsen | SKIEN | 3 |
| 2 | 2 | Pettersen | OSLO | 1 |
| 3 | 3 | Gulliksen | KONGSBERG | 2 |
| 4 | 4 | Lekven | STAVANGER | 3 |
| 5 | 6 | Helgesen | SKIEN | 1 |

**Figur 8-17: Resultat som viser informasjon om hver bileier, inklusiv antall biler de eier.**

## 8.5. Bruk av HAVING i gruppert spørring mot flere tabeller

Dersom det ønskes en oversikt over eiere som eier mer enn 1 bil, kan **HAVING**-betingelsen, som ble introdusert for spørringer mot én tabell i kapittel 6.13, benyttes. Spørringen er vist i Figur 8-18.

```
SELECT O.OwnerId, LName, City, COUNT(*) AS [Number of cars]
FROM POSTALADDRESS P
INNER JOIN OWNER O ON P.PostalCode = O.PostalCode
INNER JOIN CAR C ON O.OwnerId = C.OwnerId
GROUP BY O.OwnerId, LName, City
HAVING COUNT(*) > 1 /* Filtrerer ut grupper med mer enn 1 bil */
ORDER BY O.OwnerId
```

**Figur 8-18: HAVING-betingelse lagt til, for kun å ta med grupper med mer enn 1 bil.**

Resultatet av spørringen blir nå som vist i Figur 8-19. Merk at sorteringen (**ORDER BY**) må oppføres ***etter*** de nye instruksjonene (**GROUP BY** og **HAVING**) som er lagt til.

Results | Messages

| | OwnerId | LName | City | Number of cars |
|---|---|---|---|---|
| 1 | 1 | Olsen | SKIEN | 3 |
| 2 | 3 | Gulliksen | KONGSBERG | 2 |
| 3 | 4 | Lekven | STAVANGER | 3 |

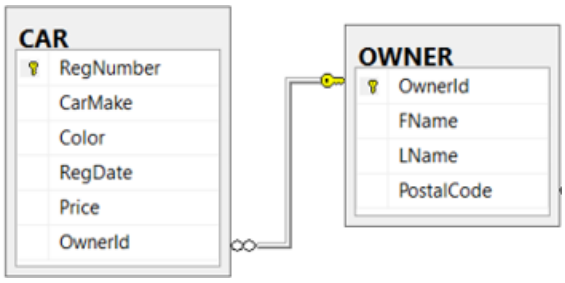
**Figur 8-19: Bare grupper med mer enn 1 bil er tatt med i resultatet.**

## 8.6. OUTER JOIN (LEFT- , RIGHT- og FULL OUTER JOIN)

I kapittel 8.2 ble det vist hvordan **INNER JOIN** kunne benyttes til å **likekoble** to, tre, eller flere tabeller i en spørring. I en likekobling lages det en kobling der verdiene i spesifiserte kolonner er lik hverandre. Dersom det da for én av verdiene i likekoblingen ***ikke*** finnes en samsvarende verdi i koblingskolonnen i den andre tabellen, tas ***ikke*** den aktuelle raden med i søkeresultatet.

I noen tilfeller ønskes det å ta med i resultatet ***alle*** verdiene fra én av de to kolonnene, selv om det ikke måtte finnes en «match» i koblingsfeltet for tabellen den kobles mot. Siden det da ***ikke*** finnes noen «match», må det i så fall for disse radene returneres **NULL**-merke for de aktuelle kolonnene.

Tabellene vist i Figur 8-20 skal brukes som eksempel for å illustrere **OUTER JOIN**.




**Figur 8-20: Tabeller som skal kobles med OUTER JOIN.**

En likekobling mellom de to tabellene vist i Figur 8-20, kan lages som vist i Figur 8-21.

```
SELECT OWNER.OwnerId, LName, RegNumber
FROM CAR INNER JOIN OWNER
ON CAR.OwnerId = OWNER.OwnerId
ORDER BY OWNER.OwnerId
```

**Figur 8-21: Indre likekolbing (INNER JOIN) mellom CAR og OWNER.**

Resultatet blir som vist i Figur 8-22

Results Messages

| | OwnerId | LName | RegNumber |
|---|---|---|---|
| 1 | 1 | Olsen | DK88971 |
| 2 | 1 | Olsen | XX45999 |
| 3 | 1 | Olsen | YZ33892 |
| 4 | 2 | Pettersen | YC43229 |
| 5 | 3 | Gulliksen | CF26702 |
| 6 | 3 | Gulliksen | KC93332 |
| 7 | 4 | Lekven | FS33932 |
| 8 | 4 | Lekven | NL50034 |
| 9 | 4 | Lekven | AA11111 |
| 10 | 6 | Helgesen | LY13335 |

**Figur 8-22: Resultatet av likekobling mellom CAR og OWNER.**

Av Figur 8-22 fremgår det at det er laget koblinger for personer med **OwnerId** 1, 2, 3, 4 og 6. Fra innlegging av data i tabellen **OWNER** (jf. Figur 8-13) ses det at det også er registrert en eier med **OwnerId** 5. At denne ikke er med i Figur 8-22, betyr at det her ikke er funnet en 'match' i likekoblingen. Dette skyldes at denne personen ikke er registrert med noen bil.

Dersom alle eiere ønskes vist med minst én rad, uansett om det finnes noen 'match' i respektive tabell som inngår i likekoblingen, kan **OUTER JOIN** benyttes.

Benyttes **LEFT OUTER JOIN**, tas alle forekomster fra tabellen plassert til venstre i spørringen med i resultatet. I Figur 8-20 betyr dette at alle verdier fra **CAR** tas med, selv om et ikke finnes en «match» i tabellen **OWNER**. Benyttes isteden **RIGHT OUTER JOIN**, tas alle verdiene fra tabellen til høyre (**OWNER**) med.

I vårt tilfelle ønskes alle eiere tatt med, også de som ***ikke*** er registrert med en bil. Det må derfor benyttes en **RIGHT OUTER JOIN** (jf. Figur 8-13) for å få med alle verdiene fra **OWNER** til høyre.

```
SELECT OWNER.OwnerId, LName, RegNumber
FROM CAR RIGHT OUTER JOIN OWNER /* Tar med alle verdiene fra OWNER */
ON CAR.OwnerId = OWNER.OwnerId
ORDER BY OWNER.OwnerId
```

**Figur 8-23: RIGHT OUTER JOIN som returnerer alle eiere, også om de ikke er registrert med bil.**

Resultatet av denne spørringen er vist i Figur 8-24. Nå er også eier 5 tatt med i resultatet, selv om denne ikke hadde sin **OwnerId** koblet mot noen av bilene i **CAR**. Siden det ikke finnes noen bil å koble mot, vil det heller ikke være noe **RegNumber** (eller andre verdier) for denne raden i tabellen **CAR**. Derfor er **RegNumber**-verdien for denne forekomsten markert med et **NULL**-merke.

Results Messages

| | OwnerId | LName | RegNumber |
|---|---|---|---|
| 1 | 1 | Olsen | DK88971 |
| 2 | 1 | Olsen | XX45999 |
| 3 | 1 | Olsen | YZ33892 |
| 4 | 2 | Pettersen | YC43229 |
| 5 | 3 | Gulliksen | CF26702 |
| 6 | 3 | Gulliksen | KC93332 |
| 7 | 4 | Lekven | FS33932 |
| 8 | 4 | Lekven | NL50034 |
| 9 | 4 | Lekven | AA11111 |
| 10 | 5 | Solvik | NULL |
| 11 | 6 | Helgesen | LY13335 |

**Figur 8-24: Resultatet av RIGHT OUTER JOIN-spørringen.**

Tilsvarende kan **LEFT OUTER JOIN** benyttes dersom det er registrert biler som ikke ennå er koblet mot eiere, når alle bilene likevel ønskes tatt med i resultatet. En slik spørring er vist i Figur 8-25.

```
SELECT OWNER.OwnerId, LName, RegNumber
FROM CAR LEFT OUTER JOIN OWNER /* Tar med alle verdiene fra CAR */
ON CAR.OwnerId = OWNER.OwnerId
ORDER BY OWNER.OwnerId
```

**Figur 8-25: LEFT OUTER JOIN returnerer her alle biler, også dem som ikke er registrert med eier.**

Resultatet av spørringen er vist i Figur 8-26, med **NULL**-merker for biler der eier ikke er registrert.

Results | Messages

| | OwnerId | LName | RegNumber |
|---|---|---|---|
| 1 | NULL | NULL | AA22222 |
| 2 | NULL | NULL | AA33333 |
| 3 | NULL | NULL | AA44444 |
| 4 | NULL | NULL | AA55555 |
| 5 | 1 | Olsen | YZ33892 |
| 6 | 1 | Olsen | DK88971 |
| 7 | 1 | Olsen | XX45999 |
| 8 | 2 | Pettersen | YC43229 |
| 9 | 3 | Gulliksen | CF26702 |
| 10 | 3 | Gulliksen | KC93332 |
| 11 | 4 | Lekven | FS33932 |
| 12 | 4 | Lekven | NL50034 |
| 13 | 4 | Lekven | AA11111 |
| 14 | 6 | Helgesen | LY13335 |

**Figur 8-26: Resultatet av LEFT OUTER JOIN-spørringen.**

Dersom alle verdier fra begge tabellene ønskes tatt med, selv om det ikke finnes noen 'match' i like-koblingen, kan en **FULL OUTER JOIN** benyttes. En slik spørring er vist i Figur 8-27.

```
SELECT OWNER.OwnerId, LName, RegNumber
FROM CAR FULL OUTER JOIN OWNER /* Tar med alle verdiene fra begge tabellene */
ON CAR.OwnerId = OWNER.OwnerId
ORDER BY OWNER.OwnerId
```

**Figur 8-27: FULL OUTER JOIN som returnerer alle biler og eiere, uavhengig om 'match' eller ikke.**

Resultatet av spørringen er vist i Figur 8-28.

Results | Messages

| | OwnerId | LName | RegNumber |
|---|---|---|---|
| 1 | NULL | NULL | AA22222 |
| 2 | NULL | NULL | AA33333 |
| 3 | NULL | NULL | AA44444 |
| 4 | NULL | NULL | AA55555 |
| 5 | 1 | Olsen | YZ33892 |
| 6 | 1 | Olsen | DK88971 |
| 7 | 1 | Olsen | XX45999 |
| 8 | 2 | Pettersen | YC43229 |
| 9 | 3 | Gulliksen | CF26702 |
| 10 | 3 | Gulliksen | KC93332 |
| 11 | 4 | Lekven | FS33932 |
| 12 | 4 | Lekven | NL50034 |
| 13 | 4 | Lekven | AA11111 |
| 14 | 5 | Solvik | NULL |
| 15 | 6 | Helgesen | LY13335 |

**Figur 8-28: Resultatet av FULL OUTER JOIN-spørringen.**

Det er også mulig å bruke **OUTER JOIN** når det er flere enn to tabeller, selv om spørringen da blir noe mer kompleks. I spørringen i Figur 8-16 ble ikke eier 5 tatt med i resultatet vist i Figur 8-17. For å ta med alle eiere, kan spørringen omskrives med **LEFT OUTER JOIN**, som vist i Figur 8-29.

```
SELECT O.OwnerId, LName, City, COUNT(*) AS [Number of cars]
FROM POSTALADDRESS AS P
INNER JOIN OWNER AS O ON P.PostalCode = O.PostalCode
LEFT OUTER JOIN CAR AS C ON O.OwnerId = C.OwnerId
GROUP BY O.OwnerId, LName, City
ORDER BY O.OwnerId ASC
```

**Figur 8-29: Spørringen fra Figur 8-16 omskrevet til bruk av OUTER JOIN mellom OWNER og CAR.**

Legg merke til at det er likekoblingen mellom **OWNER** og **CAR** som er omgjort til **LEFT OUTER JOIN**, ikke likekoblingen mellom **POSTALADDRESS** og **OWNER**.

Results | Messages

| | OwnerId | LName | City | Number of cars |
|---|---|---|---|---|
| 1 | 1 | Olsen | SKIEN | 3 |
| 2 | 2 | Pettersen | OSLO | 1 |
| 3 | 3 | Gulliksen | KONGSBERG | 2 |
| 4 | 4 | Lekven | STAVANGER | 3 |
| 5 | 5 | Solvik | KONGSBERG | 1 |
| 6 | 6 | Helgesen | SKIEN | 1 |

**Figur 8-30: Resultatet av spørringen fra Figur 8-16 omskrevet med LEFT OUTER JOIN.**

Nå ses det at også eier 5 tas med i resultatet, selv om denne ikke var registrert som eier av noen bil. Problemet er bare at denne eieren nå står oppført med 1 i kolonnen «**Number of cars**», hvilket er feil. Årsaken til dette er at **COUNT(*)** teller ***alle*** rader, og med den ytre likekoblingen er det jo nå ***én*** rad med informasjon om eier 5, selv om det ikke er registrert noen bil. Dette er en feil som er fort å gjøre.

Løsningen er å endre **COUNT(*)** til **COUNT(RegNumber)**. Da telles det isteden forekomster der det er registrert bilnummer, hvilket det ikke er for eier 5. Dette er vist i spørringen i Figur 8-31.

```
SELECT O.OwnerId, LName, City, COUNT(RegNumber) AS [Number of cars]
FROM POSTALADDRESS AS P
INNER JOIN OWNER AS O ON P.PostalCode = O.PostalCode
LEFT OUTER JOIN CAR AS C ON O.OwnerId = C.OwnerId
GROUP BY O.OwnerId, LName, City
ORDER BY O.OwnerId ASC
```

**Figur 8-31: COUNT(*) endret til COUNT(RegNumber) for å unngå tellefeil.**

Resultatet av spørringen, vist i Figur 8-32, viser at eier nr 5 nå står oppført med 0 biler istedenfor 1.

Results | Messages

| | OwnerId | LName | City | Number of cars |
|---|---|---|---|---|
| 1 | 1 | Olsen | SKIEN | 3 |
| 2 | 2 | Pettersen | OSLO | 1 |
| 3 | 3 | Gulliksen | KONGSBERG | 2 |
| 4 | 4 | Lekven | STAVANGER | 3 |
| 5 | 5 | Solvik | KONGSBERG | 0 |
| 6 | 6 | Helgesen | SKIEN | 1 |

**Figur 8-32: Resultatet viser nå riktig antall biler registrert på hver person.**

I SQL Server er det mulig å droppe **OUTER** og bare skrive **RIGHT JOIN**, **LEFT JOIN** og **FULL JOIN**, men **RIGHT OUTER JOIN** osv. er standardsyntaksen for SQL.

## 8.7. Koblingsspørringer med tilleggsbetingelser

Når koblinger mellom flere tabeller er laget, kan flere tilleggsbetingelser legges til. I Figur 8-33 er det vist kobling av tre tabeller, der det i tillegg er lagt til betingelser om at bare forekomster der byen er «Stavanger» ***og*** bilmerket er «Audi» skal med i resultatet. Legg merke til at det kun er tillatt med én **WHERE**, så ved behov for flere tilleggsbetingelser tilføyes disse med **AND**-operatoren.

```
SELECT RegNumber, CarMake, O.OwnerId, LName, City
FROM POSTALADDRESS AS P, OWNER AS O, CAR AS C
WHERE P.PostalCode = O.PostalCode
AND O.OwnerId = C.OwnerId
AND City = 'STAVANGER'
AND CarMake = 'Audi'
```

**Figur 8-33: Kobling av tre tabeller med «WHERE»-syntaks og tilleggsbetingelser.**

I Figur 8-34 er den samme spørringen vist i med «**INNER JOIN**»-syntaks.

```
SELECT RegNumber, CarMake, O.OwnerId, LName, City
FROM POSTALADDRESS AS P
INNER JOIN OWNER O ON P.PostalCode = O.PostalCode
INNER JOIN CAR AS C On O.OwnerId = C.OwnerId
WHERE City = 'STAVANGER'
AND CarMake = 'Audi'
```

**Figur 8-34: Kobling av tre tabeller med «INNER JOIN»-syntaks og tilleggsbetingelser.**

Resultatet av spørringene er vist i Figur 8-35.

Results | Messages

| | RegNumber | CarMake | OwnerId | LName | City |
|---|---|---|---|---|---|
| 1 | FS33932 | Audi | 4 | Lekven | STAVANGER |
| 2 | AA11111 | Audi | 4 | Lekven | STAVANGER |

**Figur 8-35: Resultatet av spørringene fra Figur 8-33 og Figur 8-34.**

## 8.8. Korrelert spørring (Correlated Query)

En korrelert spørring inneholder en **hovedspørring** og en **delspørring** der delspørringen avhenger av en verdi fra hovedspørringen. Det er altså en avhengighet (korrelasjon) mellom de to spørringene. Den indre spørringen blir utført for hver eneste rad av den ytre spørringen.

Dersom det er x rader i den «ytre» tabellen som inngår i spørringen, og y rader i den andre, blir totalt antall iterasjoner x • y. Dette betyr også at spørringen «kan» være tidkrevende å kjøre, dersom det er mange tabellrader.

Som et eksempel skal det ses på en spørring der det for hver bileier ønskes å finne sum pris for alle bilene vedkommende eier.

Det skal for dette formålet lages en spørring der den ytre spørringen utføres rad for rad mot tabellen **OWNER**. For hver rad i **OWNER** i den ytre løkka, skal alle radene i tabellen **CAR** itereres. Alle rader der **OwnerId**-verdiene samsvarer, tas med i summen. Deretter går den ytre løkka (**OWNER**) til neste rad, og finner så på samme måte sum pris for denne radens **OwnerId**.

For å løse oppgaven lages det en delspørring som en del av **SELECT**-delen, som for hver rad vil returnere den ønskede summen. Spørringen er vist i Figur 8-36.

```
SELECT OwnerId, LName, (SELECT SUM(Price)
                        FROM CAR
                        WHERE CAR.OwnerId = OWNER.OwnerId) AS [Sum pris for alle eierens biler]
FROM OWNER
ORDER BY OWNER.OwnerId ASC
```

**Figur 8-36: Spørring med korrelert delspørring. For hver eier returneres samlet pris for alle eierens biler.**

Resultatet av spørringen er vist i Figur 8-37. Tenk gjennom hvordan spørringen utføres!

Results | Messages

| | OwnerId | LName | Sum pris for alle eierens ... |
|---|---|---|---|
| 1 | 1 | Olsen | 1150000,00 |
| 2 | 2 | Pettersen | 460000,00 |
| 3 | 3 | Gulliksen | 730000,00 |
| 4 | 4 | Lekven | 1360000,00 |
| 5 | 5 | Solvik | NULL |
| 6 | 6 | Helgesen | 350000,00 |

**Figur 8-37: Resultatet av en korrelert delspørring for å finne sum pris for bilene til hver enkelt bileier.**

Spørringen i i Figur 8-36 kan alternativt lages med «**GROUP BY**», som vist i Figur 8-38. Det må da benyttes «**RIGHT OUTER JOIN**» dersom **OwnerId** nr 5 (med **NULL**-merke) ønskes tatt med i resultatet. Med en likekobling vil det ikke være noe «treff» på **OwnerId** for denne forekomsten.

```
SELECT OWNER.OwnerId, LName, SUM(Price)
FROM CAR RIGHT OUTER JOIN OWNER
ON CAR.OwnerId = OWNER.OwnerId
GROUP BY OWNER.OwnerId, LName
ORDER BY OWNER.OwnerId ASC
```

**Figur 8-38: Spørringen fra Figur 8-36 laget med en alternativ løsning (GROUP BY).**

## 8.9. Korrelerte spørring (Correlated Query) med egenkobling

I kapittel 8.8 ble det vist en korrelert spørring der det i **SELECT**-delen ble lagt inn en delspørring. Det er også mulig å lage en delspørring i **FROM**-delen av en spørring.

I dette eksempelet skal det lages en spørring som lister opp bilene med høyest pris innenfor hvert bilmerke. Det skal inngå en ytre løkke der alle biler i tabellen **CAR** skal itereres rad for rad. For hver rad skal så en indre løkke iterere alle radene i den samme tabellen (**CAR**) og beregne høyeste pris for bilene i den **CarMake**-raden det finnes likekobling. I den ytre løkka sjekkes det så rad for rad om raden er større eller lik den største prisen for dette bilmerket.

For å få til dette behøves det en egenkobling. Dette betyr at det må lages to instanser av samme tabell (**CAR**). Dette gjøres ved å definere samme tabell med to ulike **aliaser**, her kalt **C1** og **C2**. En spørring for dette formålet er vist i Figur 8-39.

```
SELECT CarMake, RegNumber, Price
FROM CAR AS C1     /* Definerer den første instansen av CAR-tabellen */
WHERE Price >=
      (SELECT MAX(C2.Price)          /* Finner den høyeste prisen i hver "gruppering" */
       FROM CAR AS C2                /* Definerer den andre instansen av CAR-tabellen */
       WHERE C1.CarMake = C2.CarMake) /* Kobler de to instansene */
```

**Figur 8-39: Spørring som finner bilmerket med høyeste pris innenfor hver bilmerkekategori (CarMake).**

Resultatet av spørringen er vist i Figur 8-40.

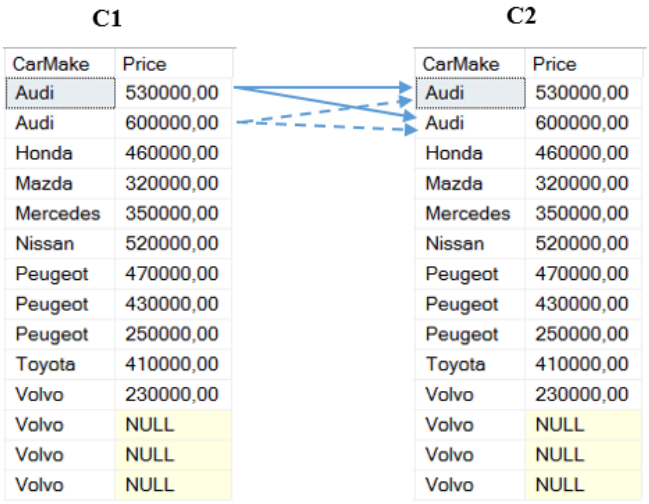
| | CarMake | RegNumber | Price |
|---|---|---|---|
| 1 | Audi | AA11111 | 600000,00 |
| 2 | Honda | YC43229 | 460000,00 |
| 3 | Mazda | CF26702 | 320000,00 |
| 4 | Mercedes | LY13335 | 350000,00 |
| 5 | Nissan | AA22222 | 520000,00 |
| 6 | Peugeot | XX45999 | 470000,00 |
| 7 | Toyota | KC93332 | 410000,00 |
| 8 | Volvo | NL50034 | 230000,00 |

**Figur 8-40: Resultatet av spørringen som finner dyreste bil innenfor hver bilmerkekategori.**

I Figur 8-41er prosessen med de to instansene **C1** og **C2** av tabellen **CAR** illustrert. Instansen med **CarMake** lik **Audi**, vist i øverste rad i **C1**, vil likekobles og sjekkes mot alle **Audi** i **C2** (2 stk. i dette tilfellet). Maksimalprisen finnes ut fra alle **Audi**-forekomstene i **C2** og sammenlignes mot den første **Audi**-verdien i **C1**. Dersom **C1**-raden er høyere eller lik maksimalprisen i **C2**, tas raden med. Deretter

gås det til neste rad i **C1** (rad 2, der det også ligger en Audi) og tilsvarende sammenligning utføres på nytt mot alle Audi-radene i **C2**. Resultatet blir til slutt som vist i Figur 8-40.

NB! Dersom det skulle være flere biler av samme merke med samme pris (dersom f.eks. begge de to Audiene i dette tilfellet hadde kostet kr 600 000) ville begge blitt tatt med i sluttresultatet.




**Figur 8-41: Rad for rad i den venstre instansen likekobles mot tilsvarende rader i den andre instansen.**

## 8.10. EXISTS

I kapittel 5.2.2 ble **IN**-operatoren introdusert. Denne kan brukes både som sjekk på om en angitt mengde verdier (f.eks. fargene 'Blue', 'Red' og 'Green') finnes, eller som en sjekk mot en delspørring som returnerer et sett med verdier.

**EXISTS**- operatoren derimot, inngår alltid i en delspørring (med **SELECT**). For øvrig fungerer den omtrent som en **IN**-operatoren.

Som eksempel skal det lages en spørring som skal returnere en liste over eiere som er innehaver av minst én blå bil.

```
SELECT *  /* Velger kolonner for rader med blå biler (der disse "exists") */
FROM OWNER
WHERE EXISTS (SELECT *     /* Delspørring som finner rader med blå biler */
              FROM CAR
              WHERE CAR.OwnerId = OWNER.OwnerId /* Korrelert likekobling */
              AND Color = 'Blue');
```

**Figur 8-42: Spørring som finner alle eiere av blå biler, der EXISTS benyttes i delspørringen.**

Resultatet av spørringen er vist i Figur 8-43.

Results Messages

| | OwnerId | FName | LName | PostalCode |
|---|---|---|---|---|
| 1 | 1 | Per | Olsen | 3740 |
| 2 | 2 | Frank | Pettersen | 0040 |

**Figur 8-43: Resultat der eiere som har minst én blå bil er listet.**

Denne spørringen kunne alternativt (uten **EXISTS**) vært laget som vist i Figur 8-44.

```
SELECT DISTINCT OWNER.* /* DISTINCT da det kun ønskes én rad for hver eier av blå bil */
FROM OWNER, CAR
WHERE OWNER.OwnerId = CAR.OwnerId
AND Color = 'Blue'
```

**Figur 8-44: Alternativ (til EXISTS) spørring for å finne alle som eier minst én blå bil.**

**EXISTS**-operatoren kan også brukes sammen med **NOT**-operatoren. Dersom spørringen ønskes endret til å returnere alle eiere som ***ikke*** har noen blå biler, kan dette gjøres som vist i Figur 8-45.

```
SELECT *  /* Velger kolonner for rader med blå biler (der disse "exists") */
FROM OWNER
WHERE NOT EXISTS (SELECT *     /* Delspørring som finner rader med blå biler */
                  FROM CAR
                  WHERE CAR.OwnerId = OWNER.OwnerId /* Korrelert likekobling */
                  AND Color = 'Blue');
```

**Figur 8-45: Delspørring med NOT EXISTS.**

**NOT EXISTS** blir her true dersom det ***ikke*** finnes noen forekomster av **Color = 'Blue'** for likekoblingen mellom **OwnerId** i de to tabellene. Dermed returneres alle rader for eiere som ***ikke*** eier blå biler.

Results | Messages

| | OwnerId | FName | LName | PostalCo... |
|---|---|---|---|---|
| 1 | 3 | Lise | Gulliksen | 3603 |
| 2 | 4 | Lena | Lekven | 4007 |
| 3 | 5 | Hans | Solvik | 3603 |
| 4 | 6 | Gunn | Helgesen | 3740 |

**Figur 8-46: Resultatet av spørring som finner alle eiere som ikke eier noen blå bil.**

## 8.11. ALL og SOME (eventuelt ANY)

**ALL** og **SOME** er logiske operatorer som må brukes sammen med sammenligningsoperatorene =, !=, >, <, <=, >=. (Det finnes i tillegg en operator kalt **ANY**, som er ekvivalent med **SOME**).

**ALL** og **SOME** brukes alltid mot delspørringer.

Disse operatorene brukes til å teste én og én verdi i en ytre spørring opp mot et sett med verdier i en delspørring. Operatorene forklares gjennom noen eksempler.

Operatoren **ALL** er en sammenligning der ***alle*** verdiene det sammenlignes mot i delspørringsresultatet må innfris for at operatoren skal bli «**true**».

La oss si at det ønskes å finne ***alle*** biler som er dyrere enn den dyreste Peugeoten. Det lages først en delspørring som finner alle Peugeot-prisene. En slik spørring, med resultat, er vist i Figur 8-47.

```
SELECT Price
FROM CAR
WHERE CarMake= 'Peugeot'
```

Results | Messages

| | Price |
|---|---|
| 1 | 250000,00 |
| 2 | 470000,00 |
| 3 | 430000,00 |

**Figur 8-47: Spørring som finner prisen til alle (tre) Peugeotene.**

Dersom «> **ALL**» brukes mot denne delspørringen, sjekkes det at det på venstresiden er større enn alt på høyresiden, hvilket her vil si større enn 250 000 **og** større end 470 000 ***og*** større enn 430 000. Det betyr at verdien på venstresiden må være større enn ***alle*** (**ALL**) verdiene delspørringen returner, hvilket underforstått innebærer at den må være større enn den største av disse verdiene (470 000).

En spørring som benytter denne delspørringen er vist i Figur 8-48. Resultatet er sortert på **Price**.

```
SELECT *
FROM CAR
WHERE Price > ALL
    (SELECT Price
     FROM CAR
     WHERE CarMake= 'Peugeot')
     ORDER BY Price ASC
```

**Figur 8-48: ALL brukes her til å returnere alle biler som har høyere pris enn alle Peugeoter.**

Resultatet av denne spørringen blir som vist i Figur 8-49. Det ses at alle bilene har en pris større enn 470 000.

Results | Messages

| | RegNum... | CarMa... | Color | RegDate | Price | OwnerId |
|---|---|---|---|---|---|---|
| 1 | AA22222 | Nissan | NULL | NULL | 520000,00 | NULL |
| 2 | FS33932 | Audi | Silver | 2014-12-05 | 530000,00 | 4 |
| 3 | AA11111 | Audi | Red | 2015-04-19 | 600000,00 | 4 |

**Figur 8-49: Resultatet av spørringen vist i Figur 8-49.**

Dersom større enn (>) i spørringen i Figur 8-48 endres til større eller lik (>=), vil spørringen også returnere biler med pris **lik** den største verdien. I dette tilfellet tas også en Peugeot med, jf. Figur 8-50.

Results | Messages

| | RegNum... | CarMake | Color | RegDate | Price | OwnerId |
|---|---|---|---|---|---|---|
| 1 | XX45999 | Peugeot | Green | 2015-03-12 | 470000,00 | 1 |
| 2 | AA22222 | Nissan | NULL | NULL | 520000,00 | NULL |
| 3 | FS33932 | Audi | Silver | 2014-12-05 | 530000,00 | 4 |
| 4 | AA11111 | Audi | Red | 2015-04-19 | 600000,00 | 4 |

**Figur 8-50: Spørringen i Figur 8-48 med sammenligningsoperatoren endret fra > til >=.**

Operatoren **SOME** (eller **ANY**, da disse er ekvivalenter) er en sammenligning der ***minst én*** av verdiene det sammenlignes med i delspørringsresultatet ***må*** innfris for at operatoren skal bli «**true**».

Dersom «**> SOME**» brukes mot denne delspørringen, sjekkes det at verdien på venstresiden er større enn minst én av verdiene på høyresiden. I dette tilfellet vil det si ***enten*** større enn 250 000 ***eller*** større end 470 000 ***eller*** større enn 430 000. Det betyr at verdien på venstresiden ***må*** være større enn den minste av disse verdiene (250 000), for da innfris «***eller***-kravene».

Spørringen vil derfor returnere alle biler med pris større enn 250 000, eller sagt på en annen måte, alle biler som er dyrere enn den billigste Peugeoten. Spørring og resultat er vist i Figur 8-51.

```
SELECT *
FROM CAR
WHERE Price >= SOME
    (SELECT Price
     FROM CAR
     WHERE CarMake= 'Peugeot')
     ORDER BY Price ASC
```

Results | Messages

| | RegNumber | CarMake | Color | RegDate | Price | OwnerId |
|---|---|---|---|---|---|---|
| 1 | CF26702 | Mazda | Red | 2011-09-09 | 320000,00 | 3 |
| 2 | LY13335 | Mercedes | Silver | 2003-12-09 | 350000,00 | 6 |
| 3 | KC93332 | Toyota | Silver | 2013-12-27 | 410000,00 | 3 |
| 4 | YZ33892 | Peugeot | Blue | 2012-07-11 | 430000,00 | 1 |
| 5 | YC43229 | Honda | Blue | 2012-03-12 | 460000,00 | 2 |
| 6 | XX45999 | Peugeot | Green | 2015-03-12 | 470000,00 | 1 |
| 7 | AA22222 | Nissan | NULL | NULL | 520000,00 | NULL |
| 8 | FS33932 | Audi | Silver | 2014-12-05 | 530000,00 | 4 |
| 9 | AA11111 | Audi | Red | 2015-04-19 | 600000,00 | 4 |

**Figur 8-51: Spørring med "SOME" som finner alle biler dyrere enn den billigste Peugeoten (250 000)**

Disse operatorene kan være litt vanskelig å forstå og er ikke så hyppige i bruk. En huskeregel er:

- **SOME (ANY)**: Tenk «**OR**», dvs. at ***minst ett*** av resultatene fra delspørringen må være sanne.
- **ALL**: Tenk «**AND**», dvs. at ***alle*** resultatene fra delspørringen må være sanne.

# 9. Mengdeoperatorer (UNION, INTERSECT og EXCEPT)

I SQL er det også tilgjengelig mengdeoperatorer. Det ses i det videre på de tre operatorene **UNION**, **INTERSECT** (snitt) og **EXCEPT** (minus). I forklaringen tas det utgangspunkt i to mengder, A og B.


**Figur 9-1: Venn-diagram for to mengder kalt A og B.**

Mengdene A og B skal i det praktiske eksempelet representeres med to tabeller med data, en for studenter (**STUDENT**) og en for lærere (**TEACHER**).

I dette tilfellet er noen av studentene stipendiater og er i en type ansettelsesforhold der de betraktes som ***både*** studenter og lærer. De ligger derfor i begge tabellene. Tabellene kan opprettes med **CREATE TABLE** som vist i Figur 9-2.

```
CREATE TABLE TEACHER
(TeacherId int,
 FirstName varchar (40),
 LastName varchar (40)
 CONSTRAINT PK_TEACHER PRIMARY KEY (TeacherId))
```

```
CREATE TABLE STUDENT
(StudentId int,
 FirstName varchar (40),
 LastName varchar (40)
 CONSTRAINT PK_STUDENT PRIMARY KEY (StudentId))
```

**Figur 9-2: Opprettelse av tabellene TEACHER og STUDENT.**

Innlegging av et utvalg testverdier i de to tabellene er vist i Figur 9-3.

```
INSERT INTO TEACHER
VALUES (1, 'Per', 'Larsen'),
       (2, 'Fredrik', 'Olsen'),
       (3, 'Kathrine', 'Fredriksen'),
       (4, 'Gro', 'Leikvoll')
```

```
INSERT INTO STUDENT
VALUES (1001, 'Trude', 'Furulund'),
       (1002, 'Ellen', 'Lyse'),
       (1, 'Per', 'Larsen'),
       (1003, 'Franz', 'Drevland'),
       (4, 'Gro', 'Leikvoll')
```

**Figur 9-3: Innlegging av data med «INSERT» i tabelleen TEACHER og STUDENT.**

Tabellene med innhold etter «**SELECT * FROM <tabel name>**» er vist i Figur 9-4. Legg spesielt merke til at to av kandidatene (1 og 4) finnes i begge tabellene.

| | StudentId | FirstName | LastName |
|---|---|---|---|
| 1 | 1 | Per | Larsen |
| 2 | 4 | Gro | Leikvoll |
| 3 | 1001 | Trude | Furulund |
| 4 | 1002 | Ellen | Lyse |
| 5 | 1003 | Franz | Drevland |

| | TeacherId | FirstName | LastName |
|---|---|---|---|
| 1 | 1 | Per | Larsen |
| 2 | 2 | Fredrik | Olsen |
| 3 | 3 | Kathrine | Fredriksen |
| 4 | 4 | Gro | Leikvoll |

**Figur 9-4: Tabellen til venstre er STUDENT og tabellen til høyre er TEACHER.**

I de påfølgende underkapitlene ses det på hvordan **UNION**, **INTERSECT** og **EXCEPT** kan brukes.

## 9.1. UNION

A UNION B (A∪B) vil gi som resultat alle rader (tupler) som finnes enten i mengden A, i mengden B eller i begge. Dette er illustrert i Figur 9-5.


**Figur 9-5:** Venn-diagram for mengden A UNION B **(A∪B)**.

Dersom det ønskes liste over alle personene som ligger ***enten*** i tabellen **STUDENT** ***eller*** i tabellen **TEACHER** ***eller*** i begge, kan det lages en SQL-spørring med UNION, som vist i Figur 9-6.

```
SELECT * FROM STUDENT
UNION
SELECT * FROM TEACHER
```

**Figur 9-6: SQL-spørring med UNION.**

I resultatet vil det fås en samlet liste over alle ansatte og studenter, som vist i Figur 9-7.

Results Messages

| | StudentId | FirstName | LastName |
|---|---|---|---|
| 1 | 1 | Per | Larsen |
| 2 | 2 | Fredrik | Olsen |
| 3 | 3 | Kathrine | Fredriksen |
| 4 | 4 | Gro | Leikvoll |
| 5 | 1001 | Trude | Furulund |
| 6 | 1002 | Ellen | Lyse |
| 7 | 1003 | Franz | Drevland |

**Figur 9-7: Resultatet av SQL med UNION for de to tabellene TEACHER og STUDENT.**

Kandidater som finnes i begge mengdene tas kun med én gang i resultatet.

## 9.2. INTERSECT

EXCEPT (A∩B) vil gi som resultat alle rader som er ***både*** i A og B (snittet), som vist i Figur 9-8.


**Figur 9-8:** Venn-diagram for mengden A INTERSECT B **(A∩B)**. Dette er snittet av de to mengdene.

Dette kan benyttes til å finne alle rader som finnes i to mengder. I vårt eksempel kan det benyttes til å finne alle som er stipendiater, da disse som nevnt innledningsvis er registrert som både student og ansatt. Dette gjelder ingen andre. Det lages derfor en spørring som vist i Figur 9-9.

```
SELECT * FROM STUDENT
INTERSECT
SELECT * FROM TEACHER
```

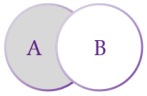
Results | Messages

| | StudentId | FirstName | LastName |
|---|---|---|---|
| 1 | 1 | Per | Larsen |
| 2 | 4 | Gro | Leikvoll |

**Figur 9-9: INTERSECT-spørring som returnerer bare dem som er stipendiater.**

## 9.3. EXCEPT

Except (A - B) vil gi som resultat alle rader (tupler) som finnes i den ene mengden minus det som også måtte finnes i den andre mengden, som illustrert i Figur 9-10.




**Figur 9-10:** Venn-diagram for mengden A EXCEPT B **(A-B)**.

Som et eksempel lages en liste over alle ansatte uten stipendiatene. Spørringen er vist i Figur 9-11.

```
SELECT * FROM TEACHER
EXCEPT /* Tar ikke med stipendiatene, som finnes i begge mengdene */
SELECT * FROM STUDENT
```

**Figur 9-11: EXCEPT-spørring som returnerer alle ansatte med unntak av stipendiater.**

Stipendiatene blir ikke med i resultatet vist i **Figur 9-12**, da disse er med i begge mengdene.

Results | Messages

| | TeacherId | FirstName | LastName |
|---|---|---|---|
| 1 | 2 | Fredrik | Olsen |
| 2 | 3 | Kathrine | Fredriksen |

**Figur 9-12: INTERSECT-spørring som returnerer alle ansatte bortsett fra stipendiater.**

# 10. VIEWS

I tillegg til tabellene generert med «**CREATE TABLE**» kan det med «**CREATE VIEW**» lages «virtuelle tabeller» (**Views**). Et View er et tilpasset «innblikk» i de virkelige dataene i en underliggende tabell. Når et View er opprettet fungerer dette derfor (med noen unntak) som en virkelig tabell.

Husk at at View alltid jobber mot de virkelige underliggende dataene. Slettes for eksempel data via et View, slettes også de underliggende dataene i de virkelige tabellene.

Noen av fordelene/egenskapene ved bruk av Views er blant annet:

- Views kan etableres som et utsnitt av én eller flere tabeller, hvilket kan forenkle spørringer som skal lages.
- Flere små tabeller kan samles til ***ett*** View, når dette er hensiktsmessig for å lage spørringer.
- De to ovennevnte gir mulighet for å gi brukere oversiktlig tilgang til akkurat de dataene de behøver, uten å måtte ha mye databasekunnskap og kjennskap til de underliggende strukturer.
- Det kan gjøres begrensninger i hva brukere skal få tilgang til, ved at det lages et View (utsnitt) med data en bruker har behov for, istedenfor å gi tilgang til alle tabeller og data.
- Ved å lage et View, og så lage spørring mot dette, kan i mange tilfeller ellers mer komplekse spørringer forenkles.
- Views er ikke egne tabeller med lagrede data, men en type definisjon som «kjøres» ved behov. Det kreves derfor ikke mye lagringsplass for å lagre et View.
- Views kan inngå i koblede spørringer sammen med virkelige tabeller.

## 10.1. CREATE VIEW

Syntaksen for å lage et **View**:

**CREATE VIEW** ViewName **AS**
**SELECT** Column 1, Column 2, etc.
**FROM** Tablename
**WHERE** (conditions)

Det kan ***ikke*** brukes «**ORDER BY**» (sortering) når **View** lages. Dette fordi **Views** håndteres som tabeller, og tabeller er per definisjon ikke sorterte (jf. forklart i kapittel 2.5 og kapittel 6.5).

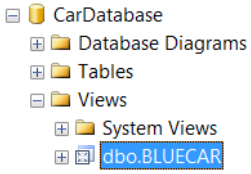
Når det lages spørringer mot et View er det tillatt å bruke «**ORDER BY**». Et resultatsett fra en spørring mot et View kan derfor returnere dataene sortert som det måtte ønskes.

Som eksempel skal det lages et View fra tabellen vist i Figur 10-1. Dette skal lages ut fra tabellen **OWNER** og skal inneholde kolonnene **RegNumber**, **CarMake**, **Price** og **Owner**. I tillegg skal det bare inneholde blå (**Blue**) biler og gis navnet **BLUECAR**. Spørringen kan lages som vist i 10.1.

```
CREATE VIEW BLUECAR AS
SELECT RegNumber, CarMake, Price, OwnerId
FROM CAR
WHERE Color = 'Blue'
```

**Figur 10-1: Lage et View kalt BLUECAR ut fra tabellen CAR**

**Viewet** skal deretter være tilgjengelig (etter en refresh) via «Object Explorer», som vist i Figur 10-2.




**Figur 10-2: Et View er opprettet og vil vises i Object Explorer**

For å vise innholdet av Viewet kan det lages en spørring mot dette, som om det skulle vært en tabell. En slik spørring er vist i Figur 10-3.

```
SELECT *
FROM BLUECAR
```

Results | Messages

| | RegNumber | CarMake | Price | OwnerId |
|---|---|---|---|---|
| 1 | DK88971 | Peugeot | 250000,00 | 1 |
| 2 | YC43229 | Honda | 460000,00 | 2 |
| 3 | YZ33892 | Peugeot | 430000,00 | 1 |

**Figur 10-3: Spørring mot et View, der bilene som listes er bare de som er blå.**

Et View kan også inngå i koblinger med andre tabeller. Dersom det eksempelvis ønskes å få en oversikt som inneholder etternavn på dem som eier de blå bilene, kan tabellene **OWNER** og **BLUECAR** kobles. Dette er vist i Figur 10-4, der resultatet også er sortert på **OwnerId**.

```
SELECT OWNER.OwnerId, LName, RegNumber, CarMake
FROM OWNER, BLUECAR
WHERE OWNER.OwnerId = BLUECAR.OwnerId
ORDER BY OWNER.OwnerId
```

Results | Messages

| | OwnerId | LName | RegNum... | CarMake |
|---|---|---|---|---|
| 1 | 1 | Olsen | DK88971 | Peugeot |
| 2 | 1 | Olsen | YZ33892 | Peugeot |
| 3 | 2 | Pettersen | YC43229 | Honda |

**Figur 10-4: Likekobling mellom et View (BLUECAR) og en tabell (OWNER)**

Et View kan også opprettes med det grafiske verktøyet. Dette gjøres ved å høyreklikke over Views-mappen vist i Object Explorer-strukturen i Figur 10-2. Fra menyen velges så «**New View**».

## 10.2. UPDATE, INSERT og DELETE via Views

Oppdatering av Views gjøres med **UPDATE**-kommandoen. Som det ses av tabellen i Figur 10-3 er prisen på bilen med **RegNumber** 'DK88971' kr 250 000. Denne prisen kan via Viewet **BLUECAR** oppdateres fra kr 250 000 til kr 700 000 i **SET**-delen, som vist med spørringen i Figur 10-5.

```
UPDATE BLUECAR
SET Price = 700000
WHERE RegNumber = 'DK88971'
```

**Figur 10-5: Prisen til bilen med regnummer 'DK88971' oppdateres via Viewet BLUECAR.**

For å vise at prisen ikke er oppdatert bare i Viewet, kan det kjøres en **SELECT**-spørring både mot Viewet **BLUECAR** og mot tabellen **CAR**, hvilket vil gi resultatene vist i Figur 10-6.

```
SELECT * FROM BLUECAR
WHERE RegNumber = 'DK88971'

SELECT * FROM CAR
WHERE RegNumber = 'DK88971'
```

Results | Messages

| | RegNumber | CarMake | Price | OwnerId |
|---|---|---|---|---|
| 1 | DK88971 | Peugeot | 700000,00 | 1 |

| | RegNumber | CarMake | Color | RegDate | Price | OwnerId |
|---|---|---|---|---|---|---|
| 1 | DK88971 | Peugeot | Blue | 2006-06-05 | 700000,00 | 1 |

**Figur 10-6: SELECT-spørringer med resultat, etter at Price er oppdatert til 700 000 for 'DK88971'.**

Det ses at både Viewet (**BLUECAR**), og den underliggende tabellen (**CAR**) som Viewet er laget ut fra, har fått prisen endret til 700 000. Dette er de samme dataene, bare sett gjennom litt ulike «briller».

Dersom originalverdien ønskes tilbakestilt igjen, gjøres dette med samme spørring som vist i Figur 10-5, men med **SET**-delen endret til «**SET Price = 250000**».

Så lenge et View er oppdaterbart (se kapittel 10.4 for restriksjoner) kan dataene oppdateres via dette. Det er da også mulig å sette inn nye data med **INSERT** og slette data med **DELETE**.

Disse instruksjonene er tidligere forklart i kapittel 5 og brukes på samme måte mot Views som mot tabeller, men vær som nevnt oppmerksom på at de ikke vil fungere på alle Views.

## 10.3. Slette et View (Drop)

SQL-kommandoen for å slette et View er tilsvarende som for en tabell: **DROP VIEW name**.

Eksempel: `DROP VIEW BLUECAR`

## 10.4. Restriksjoner for Views

Det er en del restriksjoner når det gjelder oppdatering av Views. De viktigste begrensningene for Views i SQL Server er:

- Dersom kolonner i et View skal oppdateres med **INSERT**-, **UPDATE**- og **DELETE**-kommandoer (jf. kapittel 5) må oppdateringen kun gjelde kolonner fra én av de underliggende tabellene.
- Avledete kolonner er ikke oppdaterbare, eksempelvis kolonner som er resultat av en aggregeringsfunksjon (**AVG**, **SUM**, **MIN**, **MAX**, **COUNT** osv.). Dersom dette var tillatt, er det nokså innlysende at dette også lett ville kunnet bli feil i forhold til dataunderlaget. Det samme gjelder kolonner som er et resultat av kalkulasjoner.
- Kolonner kan ikke modifiseres dersom de er påvirket av **GROUP BY**, **HAVING** eller **DISTINCT**.
- Viewet må inkludere primærnøkkelen til tabellen/tabellene den er basert på.
- Det gjelder også andre restriksjoner. Ved behov for mer utfyllende og detaljert spesifikasjon anbefales det å sjekke Microsofts nettbaserte SQL Server-dokumentasjon.

# 11. Stored Procedure

En «**Stored Procedure**» tilsvarer en **metode** (method) i et programmeringsspråk. Det er en programmert og kompilert modul på serveren, som senere kan kalles opp med Procedure-navnet ved behov. Dette gir derfor mulighet for gjenbruk av kode (i dette tilfellet gjenbruk av spørringer). I tillegg kan det gis rettighetstildeling til utførelse av en prosedyre, hvilket kan forbedre sikkerheten.

På samme måte som en metode kan lages med parametere, kan en «Stored Procedure» også lages med parametere. Dette gir spørringene mer fleksibilitet, da det kan sendes med parameterverdi(-er) som kan inngå i spørringene når disse kjøres.

## 11.1. Stored Procedure uten parametere

En **Stored Procedure** uten parametere vil utføre en forhåndslagret spørring. Som et eksempel tenker vi oss at det stadig er et behov for å utføre en SQL-spørring som beregner gjennomsnittsprisen for samtlige biler, med valutaformatering. En spørring som utfører denne oppgaven er vist i Figur 11-1.

```
SELECT FORMAT(AVG(Price), 'C', 'no') AS [Average Price]
FROM CAR
```

**Figur 11-1: SQL-kode for å finne gjennomsnittsprisen (valutaformatert) av alle bilene i tabellen CAR.**

Resultatet vil bli én kolonne, som vist i Figur 11-2.

Results | Messages

| | Average Price |
|---|---|
| 1 | kr 456 363,64 |

**Figur 11-2: Resultat av spørringen som finner gjennomsnittspris for alle biler valutaformatert.**

Syntaks for å opprette en «**Stored Procedure**»:

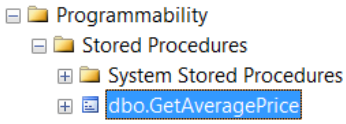
**CREATE PROCEDURE** ProcedureName **AS** `/* Angivelse av selvvalgt prosedyrenavn */`
**Spørringen defineres** `/* SELECT … osv. */`
**GO** `/* Prosedyren opprettes og lagres */`

Spørringen i Figur 11-1 kan derfor lages/lagres som en «**Stored Procedure**» som vist i Figur 11-3.

```
CREATE PROCEDURE GetAveragePrice
 AS /* Definerer spørringen: */
SELECT FORMAT(AVG(Price), 'C', 'no') AS [Average Price]
 FROM CAR
GO /* Oppretter GetAveragePrice som en "Stored Procedure"
```

**Figur 11-3: Spørring for beregning av gjennomsnittspris for alle biler, lagret som en "Stored Procedure".**

Den lagrede prosedyren vil vises i **Object Explorer** (etter en «refresh»), som vist i Figur 11-4.




**Figur 11-4: Visning av «Stored Procedures» i Object Explorer.**

For å utføre spørringen må det skrives «`Exec GetAveragePrice`» i konsollvinduet, eller bare «`GetAveragePrice`», for så å klikke på «**Execute**» i menyen. Resultatet blir som i Figur 11-2.

Man vil se større nytte av dette når det lages et program som kombinerer programmeringsspråk (f.eks. C#) med spørringer mot databaser. Dette da prosedyrene da kan kalles opp via programmet.

## 11.2. Stored Procedure med parametere

Akkurat som for metoder i programmeringsspråk, er det av stor nytteverdi å kunne overføre verdier via **parametere** til en «**Stored Procedure**». Dette gjør prosedyrene mer fleksible.

Parametere angis med navn og datatype. Eksempel: **@parameterName integer**

I Figur 11-1 ble det laget en «Stored Procedure» som fant gjennomsnittsprisen for alle bilene. Dersom det ønskes gjennomsnittspris kun for blå (Blue) biler, blir selve spørringen som vist i Figur 11-5.

```
SELECT FORMAT(AVG(Price), 'C', 'no') AS [Average Price]
 FROM CAR
 WHERE Color = 'Blue'
```

**Figur 11-5: Spørring som finner gjennomsnittsprisen for de blå bilene i tabellen CAR.**

For å lage en «Stored Procedure» som er mer fleksibel enn den fra Figur 11-1, skal farge angis via parameter. Dette så det kan sendes med en farge som argument når prosedyren kalles, som så kan brukes i spørringen. En «Stored Procedure» som løser dette med parameter, er vist i Figur 11-6.

```
CREATE PROCEDURE GetAveragePriceForSpesificCarColor
 @carColor varchar(20)  /* Deklarerer en parameter kalt carColor */
 AS   /* Definerer spørringen: */
SELECT FORMAT(AVG(Price), 'C', 'no') AS [Average Price]
 FROM CAR
 WHERE Color = @carColor   /* Fargen oversendes via parameter */
GO -- "GO" starter opprettelsen av GetAveragePrice som en "Stored Procedure"
```

**Figur 11-6: Stored Procedure med parameter (@carColor).**

**NB!** Merk at det i kommentaren til **GO**-instruksjonen i Figur 11-6 er brukt to streker (- -) som kommentarsymbol istedenfor /* comments */. Årsaken til dette er at det er en «bug» i SQL Server som gjør at sistnevnte kommentartype plassert etter **GO**-instruksjonen vil gi feilmeldingen: «*A fatal scripting error occurred. Incorrect syntax was encountered while parsing GO*».

Den lagrede prosedyren (Stored Procedure) med parameter, kan kalles opp/kjøres som vist i Figur 11-7, der gjennomsnittsprisen for blå biler beregnes. Test selv med noen andre farger.

```
exec GetAveragePriceForSpesificCarColor @carColor = 'Blue'
```

Results | Messages

| | Average Price |
|---|---|
| 1 | kr 530 000,00 |

**Figur 11-7: Kjøring av lagret prosedyre kalt opp med 'Blue' som parameter.**

Det kan også brukes flere parametere i en spørring. I Figur 11-8 er spørringen modifisert for bruk av to parametere, én for å angi biltypen (**CarMake**) og en annen for å angi fargen (**Color**). Parameterne er gitt navnene **carMake** og **carColor**. Disse blir da brukt som variabler i **WHERE**-betingelsen.

«**Camel case**»-notasjon brukes i dette kurset som navnekonvensjon for parametere, dvs. liten forbokstav i første ord og store forbokstaver for resterende ord, dersom parameternavnet er sammensatt av flere ord (som eksempelvis **carColor**).

```
CREATE PROCEDURE GetAveragePriceForSpesificCarMakeAndCarColor
@carColor varchar(20), @carMake varchar(30)
AS
SELECT FORMAT(AVG(Price), 'C', 'no') AS [Average Price]
FROM CAR
WHERE CarMake = @carMake
AND Color = @carColor
GO
```

**Figur 11-8: «Stored Procedure» med to parametere.**

Eksempeldatabasen inneholder de tre Peugeotene vist i Figur 11-9.

Results | Messages

| | RegNumber | CarMake | Color | RegDate | Price | OwnerId |
|---|---|---|---|---|---|---|
| 1 | DK88971 | Peugeot | Blue | 2006-06-05 | 700000,00 | 1 |
| 2 | XX45999 | Peugeot | Green | 2015-03-12 | 470000,00 | 1 |
| 3 | YZ33892 | Peugeot | Blue | 2012-07-11 | 430000,00 | 1 |

**Figur 11-9: Peugeotene i eksempeldatabasen.**

Ønskes det å finne gjennomsnittsprisen for **blå Peugeoter**, kan den lagrede prosedyren utføres med angivelse av 2 argumenter, som vist i Figur 11-10.

```
exec GetAveragePriceForSpesificCarMakeAndCarColor @carMake = 'Peugeot', @carColor = 'Blue'
```

**Figur 11-10: Stored Procedure utført med 2 argumenter.**

Resultatet blir kr 565 000 (jf. Figur 11-11), som er gjennomsnittsprisen for de to blå Peugeotene.

Results | Messages

| | Average Price |
|---|---|
| 1 | kr 565 000,00 |

**Figur 11-11: Gjennomsnittsprisen for blå Peugeoter i eksempeldatabasen.**

«Stored Procedures» kan også brukes til andre formål, f.eks. for å sette inn data i tabeller. I spørringen vist i Figur 11-12 lages det en lagret prosedyre for innsetting av verdier i **POSTALADDRESS**-tabellen. Det brukes to parametere til å angi henholdsvis postkode (**PostalCode**) og poststed (**City**).

```
CREATE PROCEDURE InsertPostalAddress
@postalCode char(4), @city char(35) -- Deklarer to paramatere
AS
INSERT INTO POSTALADDRESS (PostalCode, City) -- Angir kolonner
VALUES (@postalCode, @city) -- Setter inn parameterverdiene tabellen
GO
```

**Figur 11-12: Stored Procedure for innsetting av data i tabellen POSTALADDRESS**

For å teste prosedyren skal det settes inn en ny postadresse (4808 ARENDAL) i tabellen. Den lagrede prosedyren utføres ved å kjøre instruksjonen vist i Figur 11-13.

```
InsertPostalAddress @postalCode = '4808', @city = 'ARENDAL'
```

**Figur 11-13: Innsetting av en ny postadresse i tabellen POSTALADDRESS med en Stored Procedure.**

Med spørringen «**SELECT * FROM POSTALDDRESS**» vil den nye raden vises (jf. Figur 11-14).

| PostalCode | City |
|---|---|
| 0040 | OSLO |
| 3603 | KONGSBERG |
| 3740 | SKIEN |
| 4007 | STAVANGER |
| 4808 | ARENDAL |

**Figur 11-14: POSTALADDRESS-tabellen etter at raden «4808 ARENDAL» er lagt til i tabellen.**

En parameter kan også angis med **OUT**, hvilket gir mulighet for en returverdi. Dette behandles ikke i dette kompendiet.

# 12. Trigger

En **trigger** er en operasjon som utføres automatisk direkte før eller etter en **INSERT**-, **DELETE**- eller **UPDATE**-operasjon. Ved hjelp av **triggere** kan det derfor innføres automatikk knyttet til kode som ønskes utført automatisk direkte i forkant og/eller etterkant av de nevnte operasjonene.

Det er to hovedtyper triggere:

1. **INSTEAD OF**-triggere utføres i ***forkant*** av **INSERT**-, **DELETE**- og **UPDATE**- oppdateringer.
2. **AFTER**-triggere utføres i ***etterkant*** av **INSERT**-, **DELETE**- og **UPDATE**- oppdateringer.

I kapittel 12.1 og kapittel 12.2 forklares de to typene gjennom hvert sitt eksempel.

## 12.1. INSTEAD OF-trigger

Figur 11-14 viser radene som til nå er lagt inn i **POSTADDRESS**-tabellen. Den siste posten, **4008 ARENDAL**, ble lagt inn via en **Stored Procedure**. Dersom «ARENDAL» i **INSERT**-delen hadde vært skrevet «Arendal» ville denne posten hatt et avvik fra skrivemåten til de andre poststedene.

Det skal lages en trigger for å sikre at alle poststeder (i **City**-kolonnen) legges inn med ***kun*** store bokstaver. (Dette kunne også vært gjort i den lagrede prosedyren, men nå er poenget å forklare en triggers virkemåte). Triggeren skal brukes mot den lagrede prosedyren **InsertPostalAddress** (Figur 11-12).

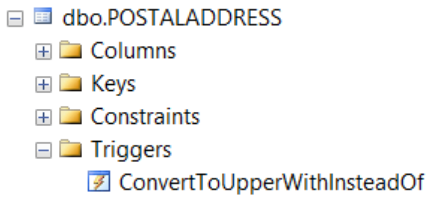
Siden endringen til store bokstaver må skje ***før*** eventuell innsetting, må det benyttes en «**INSTEAD OF**»-trigger. Denne vil da utføres i forkant av **INSERT**-operasjonen, og det må foretas en alternativ (instead of)-innsetting i den aktuelle kolonnen. I SQL Server er det tilgjengelig en **UPPER**-metode som kan benyttes til å konvertere bokstavene i et ord til bare store bokstaver.

I en trigger kan det opereres med de «midlertidige» verdiene fra den «egentlige» **INSERT**-spørringen som kjøres via den lagrede prosedyren. Disse verdiene mellomlagrer SQL Server i en tabell kalt **INSERTED**. Når man vet dette, kan verdier hentes derfra og manipuleres før endelig lagring i triggerens **INSERT**-del. Triggeren er vist i Figur 12-1, påført forklaringer.

```
CREATE TRIGGER ConvertToUpperWithInsteadOf ON POSTALADDRESS
INSTEAD OF INSERT /* Angir at det skal være INSTEAD OF og gjelde ved INSERT */
AS
    DECLARE @postalCode char(4) /* Deklarerer parametere */
    DECLARE @city char(35)

    /* Henter verdier fra den midlertidige tabellen INSERTED til parameterne: */
    SELECT @postalCode = PostalCode FROM INSERTED /* Verdi fra INSERTED til parameter */
    SELECT @city = UPPER(City) FROM INSERTED /* Omgjør til store bokstaver. Lagrer i parameter */

    INSERT INTO POSTALADDRESS (PostalCode, City) /* Triggeren legger inn oppdaterte verdier */
    VALUES (@postalCode, @city) /* Parameterverdiene legges inn i tabellen */
GO
```

**Figur 12-1: Trigger der alle poststeder konverteres til bare store bokstaver før INSERT utføres.**

Selve triggeren vil finnes igjen i **Object Explorer** under **Tables -> PostalAddress -> Triggers**, som vist i Figur 12-2. Ved å høyreklikke over triggeren, fås blant annet menyvalg for å endre triggeren.




**Figur 12-2: Triggerne lagres og kan ses i Object Eplorer i strukturene til for den aktuelle tabellen.**

Triggeren skal testes ved å legge inn enda en postadresse via den lagrede prosedyren (Stored Procedure) kalt **InsertPostalAddress**. Mens spørringen i Figur 11-13 la inn «**4808 ARENDAL**», skal det nå legges inn «**5700 Voss**». Dette for å teste om triggeren virker. Virker den, skal «**Voss**» omgjøres til «**VOSS**» (med store bokstaver) før **INSERT** (i triggeren) legger dataene inn som en ny rad i tabellen **POSTALADDRESS**.

Den lagrede prosedyren utføres med instruksjonen vist i Figur 12-3.

```
InsertPostalAddress @postalCode = '5700', @city = 'Voss'
```

**Figur 12-3: Den lagrede prosedyren brukes til å legge ny post inn i tabellen POSTALADDRESS.**

Dersom triggeren har fungert skal «**Voss**» etter utførelsen være endret til «**VOSS**». Utføres det deretter en «**SELECT * FROM POSTALADDRESS**» skal ny rad («5700 VOSS») vises som i Figur 12-4.

Results | Messages

| | PostalCo... | City |
|---|---|---|
| 1 | 0040 | OSLO |
| 2 | 3603 | KONGSBERG |
| 3 | 3740 | SKIEN |
| 4 | 4007 | STAVANGER |
| 5 | 4808 | ARENDAL |
| 6 | 5700 | VOSS |

**Figur 12-4: «5700 Voss» lagt inn i POSTALADDRESS, med Voss omgjort til VOSS av triggeren.**

## 12.2. AFTER-trigger

**AFTER**-trigger fungerer også ved **INSERT**, **DELETE** og **UPDATE**, men med triggerutførelse ***etter*** at oppdatering har funnet sted.

Som et eksempel på bruk av en **AFTER**-trigger skal det lages en loggetabell der det skal logges informasjon om tidspunktene hver gang det legges inn nye rader i **POSTALADDRESS**-tabellen.

Loggedataene skal lagres i en tabell kalt **POSTALADDRESSLOG**, som skal inneholde kolonnene **PostalCode**, **City** og **LogTime**. Det må derfor først lages en ny tabell, der **POSTALADDRESSLOG** velges som tabellnavn. SQL for tabelloppretteLsen er vist i Figur 12-5.

```
CREATE TABLE POSTALADDRESSLOG
(  PostalCode char(4),
   City char(35),
   LogTime DateTime  ) /* DateTime er datatype for å lagre både dato og tid */
```

**Figur 12-5: Lager ny tabell for å loggføre INSERT-operasjoner mot POSTALADDRESS -tabellen.**

Det skal så lages en **trigger**. Hver gang det utføres en **INSERT**-operasjon mot tabellen **POSTALADDRESS** skal triggeren ***etter*** dette også legge dataene i **POSTALADDRESSLOG**.

I tillegg til **PostalCode** og **City**, som kan legges over direkte, skal et «**timestamp**» genereres når **INSERT** finner sted. **CURRENT_TIMESTAMP** er en metode i SQL Server som kan utføre dette. Definisjon av triggeren, med kommentarer innlagt, er vist i Figur 12-6.

```
CREATE TRIGGER LogInsertsWithAfter ON POSTALADDRESS
AFTER INSERT /* Skal utføres etter hver INSERT i POSTALADDRESS-tabellen */
AS
    DECLARE @postalCode char(4) /* Deklarerer parametere */
    DECLARE @city char(35)
    DECLARE @logTimestamp DateTime

    SELECT @postalCode = PostalCode FROM INSERTED /* Fra INSERTED-tabell til parameter */
    SELECT @city = City FROM INSERTED /* Fra INSERTED-tabell til parameter */
    SELECT @logTimestamp = CURRENT_TIMESTAMP /* øyeblikkstid lagres i parameter */

    INSERT INTO POSTALADDRESSLOG (PostalCode, City, LogTime)
    VALUES (@postalCode, @city, @logTimestamp) /* Verdier til loggetabellen */
GO
```

**Figur 12-6: Trigger for logging av data i en loggetabell ved INSERT-operasjoner i POSTALADDRESS**.

For å teste triggeren som er laget, skal det på nytt legges inn data via den lagrede prosedyren, sånn det ble gjort i Figur 11-13. Denne gang legges det inn tre rader. Siden det registreres «timestamp», så la det gå litt tid (i hvert fall noen sekunder) mellom kjøring av hver de tre radene.

De tre radene med data som skal legges inn via den lagrede prosedyren er vist i Figur 12-7.

```
Exec InsertPostalAddress @postalCode = '5701', @city = 'Voss'
Exec InsertPostalAddress @postalCode = '5702', @city = 'Voss'
Exec InsertPostalAddress @postalCode = '5703', @city = 'Voss'
```

**Figur 12-7: Tre rader med data skal legges inn i tabellen POSTALADDRESS via den lagrede prosedyren.**

Dersom den første triggeren har fungert, skal alle «**Voss**» være omgjort til «**VOSS**». Sjekk at dette har skjedd i tabellen **POSTALADDRESS**.

Dersom den andre triggeren har fungert, skal dataene også være lagt inn i den nye tabellen kalt **POSTALADDRESS**. I tillegg til **PostalCode** og **City** skal triggeren ha generert et «**timestamp**» for den tredje kolonnen (**LogTime**). Resultatet skal bli som vist i Figur 12-8 (med andre tidsdata).

Results Messages

| | PostalCode | City | LogTime |
|---|---|---|---|
| 1 | 5701 | VOSS | 2015-08-12 20:38:00.160 |
| 2 | 5702 | VOSS | 2015-08-12 20:38:45.893 |
| 3 | 5703 | VOSS | 2015-08-12 20:40:05.437 |

**Figur 12-8: innholdet generert av AFTER-triggeren i den nye tabellen POSTALADDRESSLOG.**

Det ses at det for hver **INSERT**-operasjon mot tabellen **POSTADDRESS**, samtidig er lagt inn en rad i tabellen **POSTALADDRESS**, med blant annet informasjon om dato/tid for **INSERT**-operasjonen.

# 13. Brukerrettigheter (Data Control Language (DCL)

Til nå er det sett på SQL-mulighetene i SQL Data Definition Language (**DDL**) og Data Manipulation Language (**DML**). Den tredje og siste delen av SQL er Data Control Language (**DCL**). I denne ligger det muligheter for opprettelse av brukere og grupper, samt å tildele/fjerne rettigheter knyttet til disse.

Det gås ikke grundig inn på denne delen, men gis en kort introduksjon med noen eksempler knyttet til noen viktige prinsipper for rettighetsstyring med SQL.

Det skilles gjerne på **systemrettigheter** og **objektrettigheter**.

Systemrettigheter gjelder **DDL**-delen av SQL, eksempelvis rettighet til å lage nye tabeller/objekter. **CREATE TABLE**-rettighet er et eksempel på dette.

Objektrettigheter gjelder **DML**-delen, der brukere gis rettigheter knyttet til det å utføre operasjoner mot dataene. Eksempler på objektrettigheter er **SELECT**, **INSERT**, **UPDATE** og **EXECUTE**.

## 13.1. Opprettelse av brukere (users) og grupper (roles)

Det kan opprettes **brukere** (users), som tildeles rettigheter. I tillegg kan det opprettes **grupper** (roles). Grupper gir mulighet for å samle brukere, for så å tildele rettigheter til grupper istedenfor enkeltbrukere. Alle brukerne som er gitt medlemskap i en gruppe, vil da få samme rettigheter.

Syntaksen for å opprette en bruker (user) med **SQL**:

- **CREATE USER** username **IDENTIFIED BY** password.
- Eksempel: `CREATE LOGIN testUser1 WITH PASSWORD = 'P123@321'`

Syntaksen for å opprette en gruppe (role) med SQL:

- **CREATE ROLE** _rolename
- Eksempel: `CREATE ROLE carowners`

Når en gruppe ***er*** opprettet, er SQL-syntaksen for å gi brukere tilgang til gruppen sånn:

- **ALTER ROLE** roleName **ADD MEMBER** _username
- Eksempel: `ALTER ROLE carOwners ADD MEMBER testUser1`

## 13.2. Ulike typer rettigheter tildeles med GRANT

DML-operasjonene i SQL kan kategoriseres i fire hovedoperasjoner: Legge inn data, lese data, oppdatere/endre data og slette data. Dette prinsippet refereres ofte til som **CRUD**, som er en huskeregel for denne typen operasjoner. **CRUD** står for **Create**, **Read**, **Update** og **Delete**, og gjelder ikke bare databaser. I SQL er disse implementert med **INSERT**, **SELECT**, **UPDATE** og **DELETE**.

I rettighetssammenheng benyttes derfor disse til å angi hva slags rettigheter en gruppe og/eller bruker skal gis. Det kan gis rettigheter til blant annet kun å lese data (**SELECT**), legge inn data (**INSERT**) oppdatere data (**UPDATE**) og slette data (**DELETE**).

Eksempel på å gi leserettighet (altså tilgang til å kjøre **SELECT**-setninger) til en enkeltbruker:

- `GRANT SELECT ON CAR to testUser1`

Eksempel på å gi leserettighet (altså tilgang til å kjøre **SELECT**-setninger) til en gruppe:

- `GRANT SELECT ON CAR TO carOwner`

Brukeren **tesUser1**, som er medlem av gruppen **carOwner**, skal da kunne utføre denne spørringen:

```
SELECT *
FROM CAR
```

Dersom **tesUser1** forsøker å sette inn verdier (**INSERT**), som vist under, gis det rettighetsfeilmelding:

```
INSERT INTO CAR (RegNumber)
VALUES ('zz9999')
```

Skal gruppen **carOwner** få rettighet til ***både*** å lese og å sette inn data, gjøres dette sånn:

- `GRANT SELECT, INSERT ON CAR TO carOwner`

For å gi tilgang på **DDL**-nivå, for å kunne opprette tabeller etc., kan rettigheter som **CREATE TABLE** m.m. benyttes. Her er noen eksempler:

- `GRANT CREATE TABLE TO carOwner`
- `GRANT CREATE VIEW TO carOwner`
- `GRANT ALTER TABLE TO carOwner`
- `GRANT DROP TABLE TO carOwner`
- `GRANT CREATE PROCEDURE TO carOwner`

Flere rettigheter kan også tildeles i én instruksjon

- `GRANT CREATE TABLE, CREATE VIEW, DROP TABLE TO carOwner`

For mer detaljert informasjon om rettighetsstyring i T-SQL, se Microsofts MSDN-nettsider.

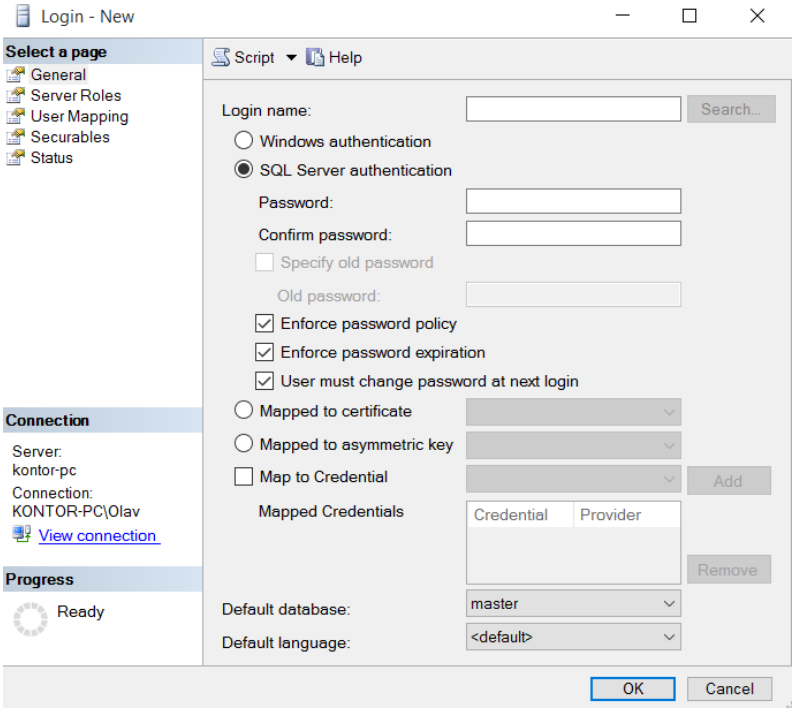
## 13.3. Rettigheter fjernes med REVOKE

For å **fjerne** tildelte rettigheter, brukes **REVOKE**. I eksempelet nedenfor fjernes rettigheten til å opprette tabeller og Views, som ble tildelt gruppen **carOwner** i forrige kapittel. Merk at det da benyttes **FROM**, istedenfor **TO** som ble benyttet i SQL-syntaksen ved **tildeling** av rettigheter:

- `REVOKE CREATE TABLE, CREATE VIEW FROM carOwner`

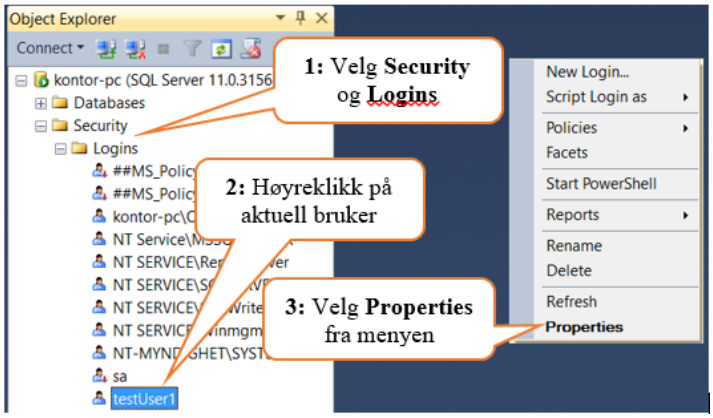
## 13.4. Bruker- og rettighetsstyring via det grafiske brukergrensesnittet

Rettigheter kan også tildeles via det grafiske brukergrensesnittet. En bruker kan opprettes ved å høyre-klikke over **Logins**-mappen (vist i Figur 13-2). Velg så «SQL Server authentication», og opprett en bruker med et valgt brukernavn og et passord.



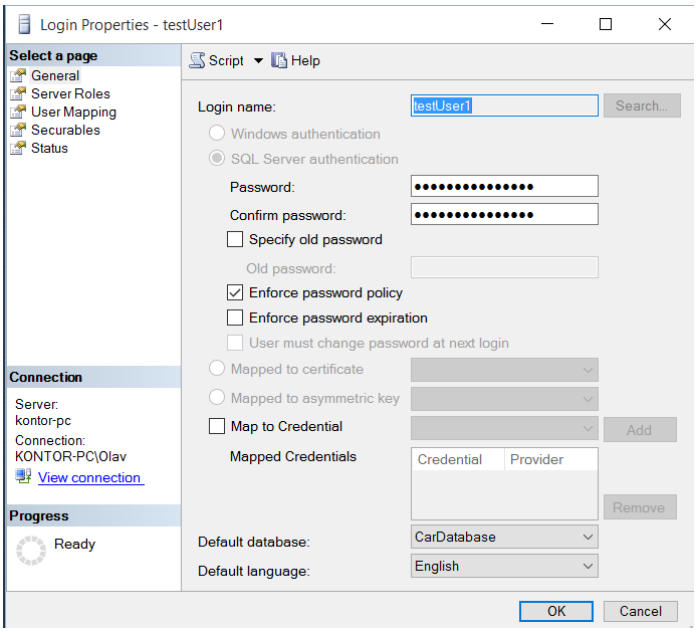

**Figur 13-1: Opprettelse av en bruker som kan logge seg på SQL Serveren.**

I Figur 13-2 vises en testbruker kalt **testUser1**, som er opprettet med SQL Server Management Studio.



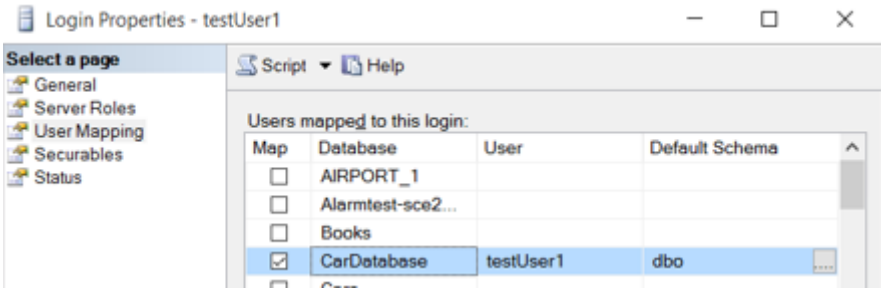

**Figur 13-2: Tildeling av rettigheter via SQL Server Management Studio.**

Ved behov for senere endringer knyttet til denne brukeren, høyreklikk da over brukernavnet (**testUser1** i dette tilfellet), som vist i Figur 13-3, og velg «**Properties**». Da fremkommer dialogvinduet vist i Figur 13-1. Fra menyen «**Select a page**» i venstrekanten, kan det velges «**General**», «**Server Roles**» etc. «**General**»-vinduet er det som først åpnes.

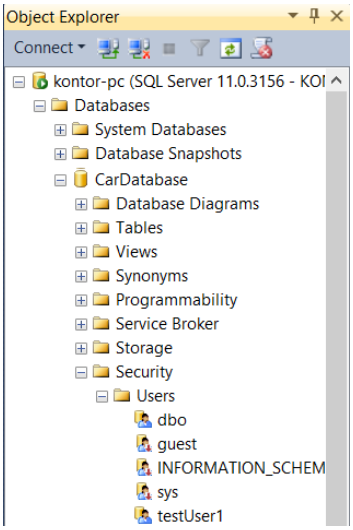



**Figur 13-3: Dialogvindu for konfigurering av brukerrettigheter etc.**

Under «**User Mapping**» (Figur 13-4Figur 13-4), kan brukeren «mappes»/knyttes opp mot en database. I dette tilfellet er brukeren knyttet opp mot databasen **CarDatabase**.

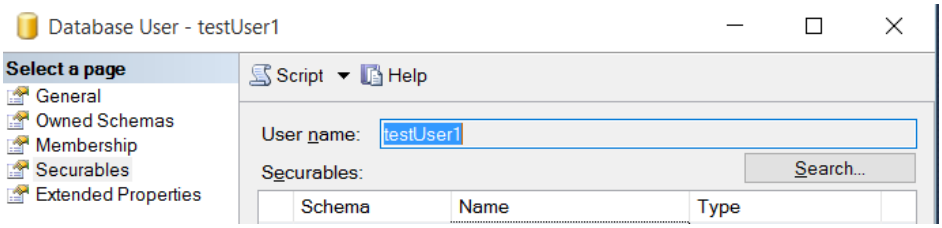



**Figur 13-4: User Mapping: Brukeren testUser1 "mappet" mot databasen CarDatabase.**



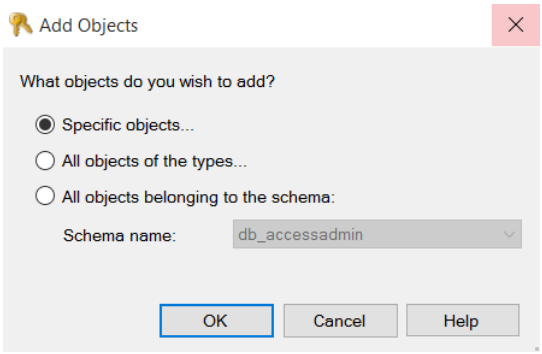

**Figur 13-5: Brukeren testUser1 tilgjengelig under Databases -> CarDatabase -> Security -> Users**

Finn så **testUser1** under ***Databases*** -> ***CarDatabase*** -> ***Security*** -> ***Users***. Høyreklikk over brukeren og velg **Properties**. Velg **Securables**, som vist i Figur 13-6.



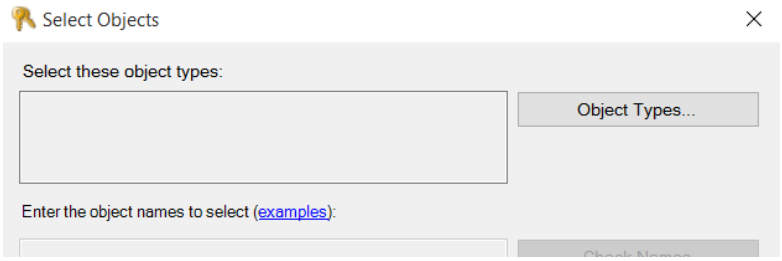

**Figur 13-6: Vinduet "Securables" for å gi tilgang for en bruker til aktuelle elementer.**

Klikk så på **Search**-knappen (Figur 13-6). Da vises menyen i objekt-valgmenyen i Figur 13-7.



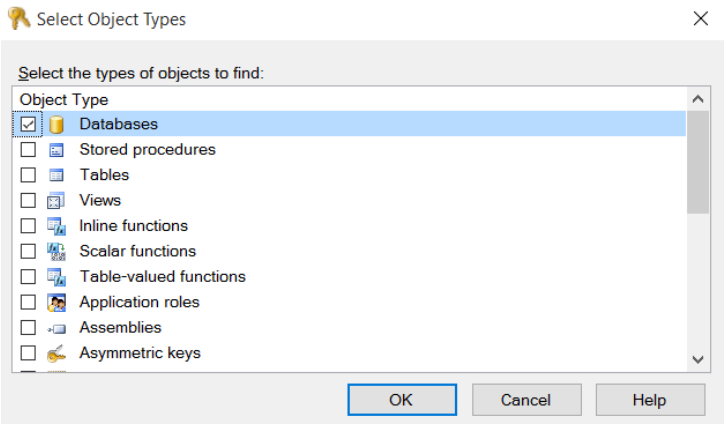

**Figur 13-7: Meny for å angi objekter det ønskes satt rettigheter for.**

Velg «**Specific objects…**» og klikk på **OK**. Da vises vinduet i Figur 13-8.



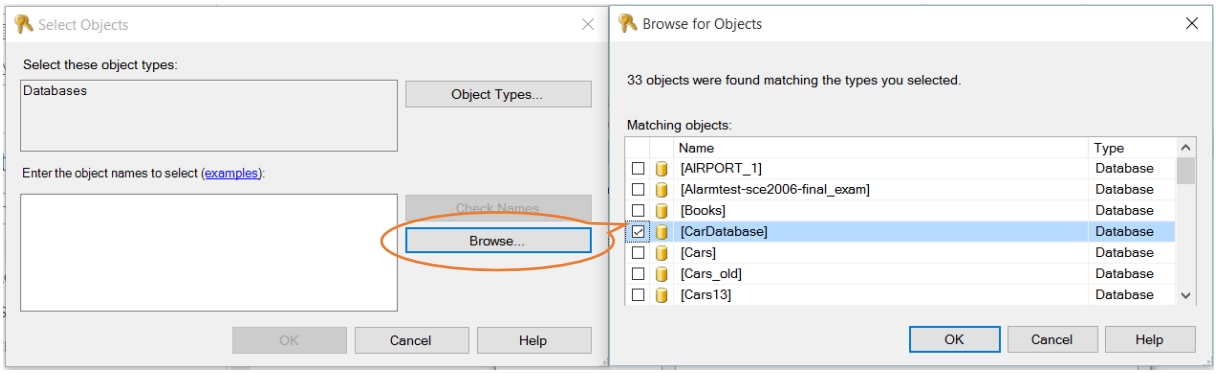

**Figur 13-8: Meny for å velge aktuelle objekttyper.**

Klikk på knappen «**Object Types**» i Figur 13-8. En meny med mulige objekttyper det kan settes rettigheter på, vil da fremkomme, som vist i Figur 13-9.



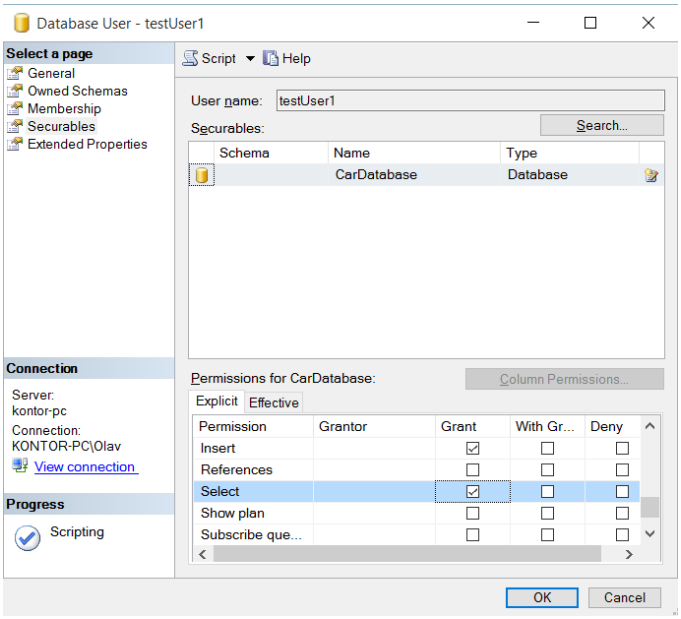

**Figur 13-9: Meny for valg av spesifikk (-e) objekttype (-r).**

For dette eksempelet, kryss av for sjekkboksen foran «**Databases**», som vist i Figur 13-9.




**Figur 13-10: Resultat etter Browse-operasjon for søk etter "Databases".**

Klikk OK igjen. Da vises en lang liste med avkryssningsbokser. I den kan det spesifiseres hvilke rettigheter brukeren skal ha til **CarDatabase**-tabellen. I Figur 13-11 er **Select**- og **Insert**-rettigheter gitt til brukeren. Test en for en av disse, og sjekk hvordan de fungerer når innlogget som **testUser1**.




**Figur 13-11: Valg av rettigheter for brukeren testUser1 til tabellen «CarDatabase».**

Sikkerhet knyttet til databaser er et eget (og omfattende) tema og ikke en vesentlig del av dette kurset, men de grunnleggende prinsippene forklart i dette kompendiet, bør man være kjent med.

# 14. Databasemodellering

I kapitlene frem til nå er det sett på hvordan SQL og SQL Server kan brukes til å lage databasestrukturer og å utføre spørringer mot disse. Dette er en typisk bottom-up planlegging, der det startes med detaljene, for så å jobber seg opp mot en helhet. Normalt vil en database utvikles ut fra en top-down-strategi, der det startes med en overordnet planlegging av databasestrukturen, for så å utføre detaljene når den overordnet strukturen er på plass.

På det øverste nivået opererer man gjerne med en **konseptuell modell**, der man utelater alle detaljer og konsentrerer seg om den helt overordnede strukturen. Deretter utvides modellen med mer detaljer, inntil den til slutt er klar til fysisk implementasjon i et konkret databasesystem.

I vårt tilfelle skal en applikasjon kalt **erwin** (med liten **e**) [8] benyttes til å modellere databasene. **erwin** er et såkalt CASE-tool (Computer Aided Software Engineering), tilgjengelig gratis for studenter og ansette ved høgskoler og universiteter, men man må registrere seg, for deretter å få tilsendt en lisenskode.

I **erwin** modelleres først det erwin kaller «**logisk modell**». Deretter omsetter **erwin** modellen til en «**fysisk modell**». Når den fysiske modellen er klar, kan **erwin** automatisk generere et **SQL-script**, som kan brukes til å generere en database i et valgt databasehåndteringssystem, eksempelvis i **SQL Server**.

Hva som vises/modelleres på de to nivåene kan konfigureres, og konfigurering som vil brukes i våre eksempler når det gjelder logisk og fysisk nivå, er beskrevet i kapittel 14.3.

Alle modelleringstrinnene er forklart i et sett med fire **videoer** publisert (på engelsk) på **YouTube**. I dette kapittelet forklares de ulike trinnene kort, men det anbefales å gjennomgå videoserien, der mer detaljer og praktiske eksempler demonstreres.

**Kort forklaring til videoinnholdet og lenker til videoene på YouTube:**

**erwin – Basic notation, relationships and cardinalities (17:01 minutes)**
Link to video: https://www.youtube.com/watch?v=0h7olmsGHwo

**Topics:** E/R-modeling basics: The basic symbols and notation of the E/R-model, based on the Information Engineering-method (with Crow Coot Notation), are explained. Examples on how to use these symbols to create a logical E/R-model, and thereafter transform the model to a physical model, from where a table structure can be created, are illustrated. Cardinality and one-to-many-, many-to-many- and one-to-one-relationships are explained and exemplified.

---

**Configuring erwin and creating templates (05:31 minutes)**
Link to video: https://www.youtube.com/watch?v=56i5z43Mlbk

**Topics:** The video shows **erwin**-configuration of what model elements to display when modelling both at a Conceptual (Logical) and Physical level. When the configuration is completed, a template for later use is created. The video also shows configuration of the Forward Engineering-process, where CREATE TABLE-scripts can be generated. When configured, a template is also here created for later use.

---

**A sensor-example – Creating a logical E/R-model and a physical model (18:02 minutes)**
Link to video: https://www.youtube.com/watch?v=mVwiq0kPKE4&t=4s

**Topics:** This video shows the entire process from the analyze phase, via E/R-diagram (Conceptual/Logical model) to a Physical model. It explains how to use the different E/R-notation elements (Entities, Entity Names, Identifier attributes, Non-identifier attributes, Identifying

relationships (one-to-many), Non-identifying relationships (one-to-many) and Many-to-Many-relationships (and how to resolve them). The differences between a Conceptual (Logical) model and a Physical model are explained.

**A sensor-example: From physical model to a database in SQL Server – (04:30 minutes)**
Link to video: https://www.youtube.com/watch?v=rUK081XkztQ

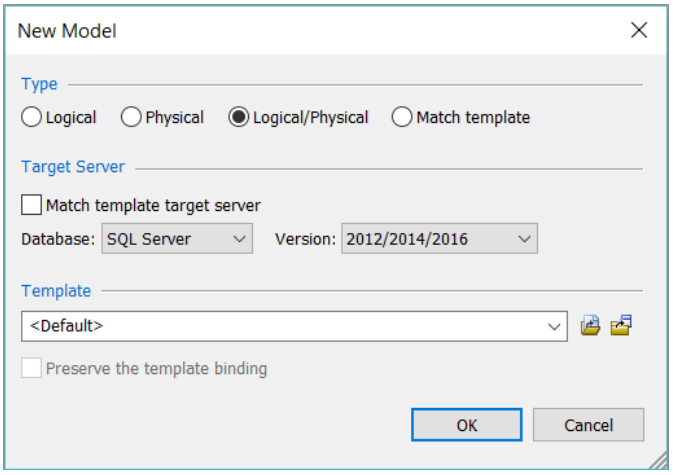
**Topics:** This video shows how to create a database in SQL Server based on a Physical model in erwin. The Physical model from the sensor example created in an earlier video, will be used as a starting point for creating the necessary script to create the database.

---

## 14.1. Opprette et erwin-prosjekt

Ved å velge ***File*** → ***New*** fra menyen, kan man opprette et nytt **erwin**-prosjekt. Etter at systemet er konfigurert, kan det lønne seg å lagre oppsettet som en mal (template). For neste prosjekt man skal starte, velger man da i så fall ***File*** → ***Open***, og velger malen som er lagret.

Det bør velges kombinasjonen **Logical/Physical**, som vist i Figur 14-1, da det først skal lages en **logisk E/R-modell**, for så å la erwin benytte denne til å generere en **fysisk modell**. I samme menybilde velges hvilken database man ønsker modellen skal implementeres i, hvilket for vår del er **SQLServer**.




**Figur 14-1: Opprettelse av erwin-prosjekt med Logical/Physical modell for implementering i SQL Server.**

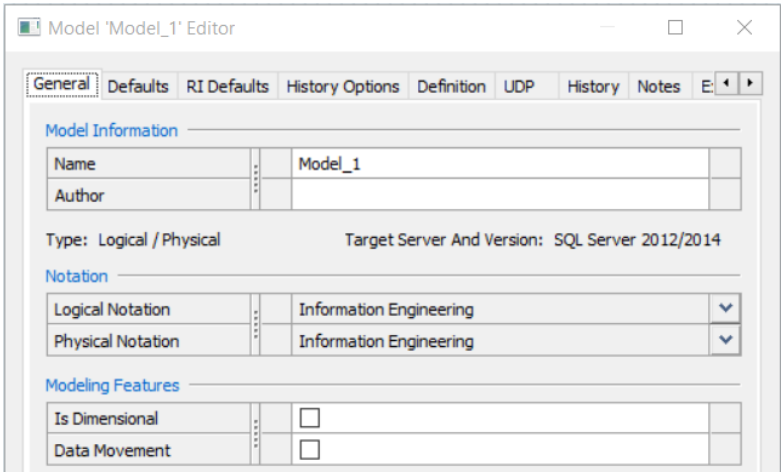
## 14.2. Konfigurasjon av erwin

Det første som bør gjøres er å konfigurere **erwin**, deriblant velge notasjonssett som skal benyttes.

Den opprinnelige modellen for E/R-diagrammer ble utviklet av Peter Chen i 1976 [9]. Senere er det utviklet andre tilsvarende modeller med andre notasjonssett. Blant de mest benyttede er **Information Engineering**, populært kalt **kråkefotnotasjon** (**Crow's foot notation**), fordi et av de sentrale symbolene ligner på en kråkefot. Symbolene er her mer kompakte i den grafiske utformingen enn **Chen**-modellen, hvilket gjør den enklere å bruke når modellene vokser seg større.

**Information Engineering**-notasjonen blir benyttet i dette kompendiet og er et tilgjengelig valg i **erwin**.

Velg **Model → Model Properties** fra **erwin**-menyen vist i Figur 14-2, og velg modellen «**Information engineering**». Denne modellen benytter den såkalte «**kråkefot-notasjon**», det vil si dette symbolet: ⊲




**Figur 14-2: erwin-skjermbilde for å konfigurere innstillinger.**

I samme skjermbilde (Figur 14-2) kan også andre innstillinger gjøres. Under **Defaults**-fanen kan det for eksempel settes hva som ønskes som standardvalgt (default) datatype for attributter. Denne er som standard satt til char (18). Andre konfigurasjoner kan også gjøres.

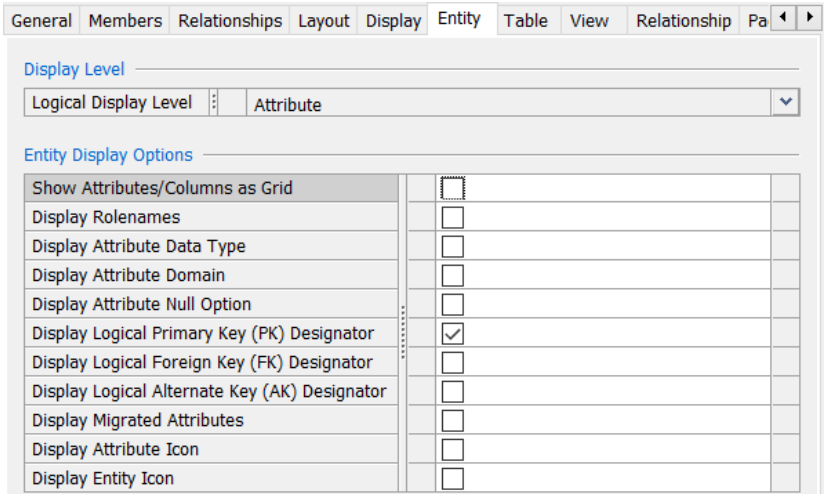
Klikk på **Close**-knappen nede til høyre for å gå ut av menyen. **NB!** Klikker man isteden på **X** oppe i høyre hjørne, blir innstillingene som er utført ***ikke*** lagret.

Ved så å dobbeltklikke på et tomt område i skjemaet, får man frem en annen konfigurasjonsmeny. Velg der **Entity**-fanen, som vist i Figur 14-3.

Innstillingene som settes under **Entity**-fanen, gjelder den **logiske modellen**. Dette skal i vårt tilfelle være en **konseptuell modell**, der bare **identifikatorer** skal vises, ikke **fremmednøkler**. Sistnevnte vil fremkomme automatisk i **fysisk modell**, ut fra de **relasjonsforhold** som påføres i den **logiske modellen**.

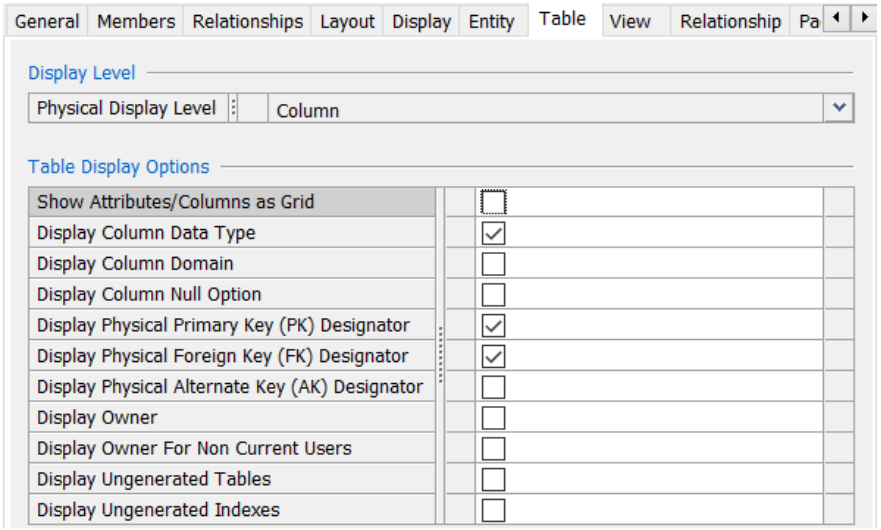
For at kun identifikatorer skal vises, avmerkes bare «*Display Logical Primary Key (PK) Designator*».

Når man går fra logiske modell til fysisk modell, blir en identifikator i en **entitet** ofte til en primærnøkkel i en **tabell**, men noen ganger blir primærnøkkelen en sammensetning av identifikatorer fra flere entiteter. Av den grunn brukes gjerne navnet **identifikator** i den **logiske tabellen** og **primærnøkkel** i den **fysiske modellen**, da disse ikke nødvendigvis er like.



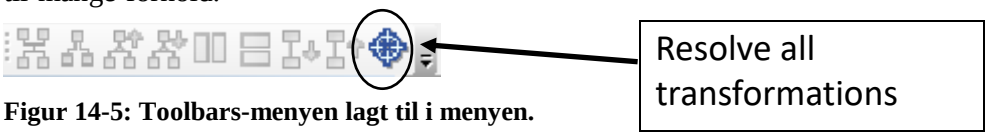

**Figur 14-3: I Entity-menyen settes egenskaper for den logiske modellen.**

Velg deretter **Table**-fanen til høyre for **Entity**-fanen. Under **Table**-fanen gjøres konfigurasjoner som gjelder den **fysiske modellen**. I den fysiske modellen ønsker vi at ***både* primærnøkler**, **fremmednøkler** og **datatyper** skal vises. Konfigurer **Table**-valgene som vist i Figur 14-4.




**Figur 14-4: I Entity-menyen settes egenskaper for den fysiske modellen.**

Legg også til menyen «**Transformations**» under **View → Toolbars**. Menyen er vist i Figur 14-5.Denne menyen inneholder verktøyet «**Resolve all transformations**», som skal benyttes til å løse opp mange-til-mange-forhold.




**Figur 14-5: Toolbars-menyen lagt til i menyen.**

## 14.3. Hva skiller E/R-diagram (logisk modell) fra fysisk modell

Normalt startes modelleringen med en konseptuell modell. Da tar man gjerne ikke med eksempelvis fremmednøkler og datatyper. I konseptuelle modeller på det høyeste abstraksjonsnivået, tar man heller ikke med attributter, men ser bare på entitetsnavn.

I ulike systemer opereres det også med litt ulike definisjoner av hva som er konseptuell modell, E/R-diagram, logisk modell og fysisk modell. I erwin-eksemplene som her benyttes, skiller vi bare mellom logiske (her også kalt E/R-diagrammer) og fysiske modeller (som representerer den endelige database-strukturen som skal implementeres i et databasehåndteringssystem).

I vårt tilfelle benyttes begrepsapparatene beskrevet i Figur 14-6 i henholdsvis logisk og fysisk modell. Dette samsvarer med konfigureringene gjort i erwin.

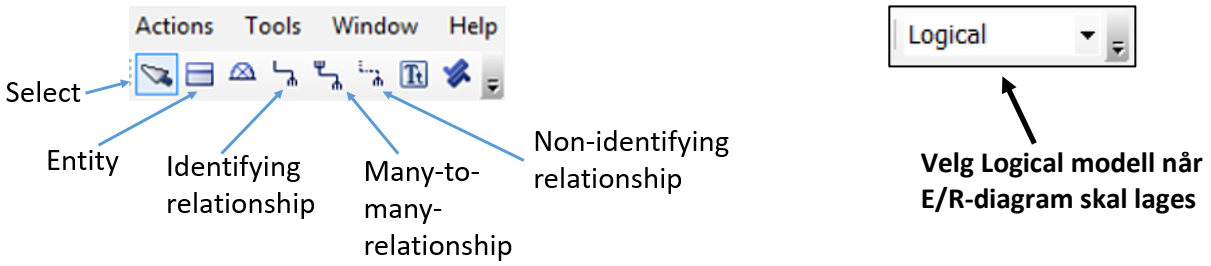
| Logical model: |
|---|
| **Elements** |
| Entity names |
| Entity relationships |
| Attributes |
| Identifiers |

| Physical model: |
|---|
| **Elements** |
| Table names |
| Table relationships |
| Columns |
| Datatypes |
| Primary key |
| Foreign keys |

**Figur 14-6: Begreper benyttet i henholdsvis logisk og fysisk modell i erwin.**

## 14.4. Grunnleggende E/R-symboler i logisk modell i erwin

E/R-diagrammer tegnes i logisk modell (Logical). De grunnleggende symbolene som skal benyttes til å lage E/R-modeller ligger i øverste menyrad og er vist i Figur 14-7.




**Figur 14-7: De grunnleggende symbolene som skal brukes ved tegning av E/R-diagrammer.**

## 14.5. Eksempel som skal brukes til å modellere en E/R-modell

I de kommende kapitlene skal et modellering av en database for billøp benyttes til å eksemplifisere ulike modelleringsteknikker relatert til E/R-modellering. Ulike typer modelleringsteknikker blir presentert etter hvert som E/R-modellen utvidet. Til slutt lages en konkret databaseinstallasjon ut fra resultatet.

## 14.6. Entiteter

Entiteter er hovedelementet i et **E/R-diagram** (**Entity/Relationship-diagram**). Entiteter skal bli til tabeller når man går fra logisk til fysisk modell. Entiteter inneholder attributter, der noen av dem er identifikatorer (nøkkel-attributter), mens andre er ikke-nøkkel-attributter.

Figur 14-8 viser en entitet med navnet **BILEIER**. Denne entiteten inneholder tre attributter, **EierId**, **Fornavn** og **Etternavn**. **EierId** er identifikator og er derfor markert med et nøkkelsymbol. Skriver man diagrammer for hånd, benyttes ofte heller en understrekning, for å markere identifikatorattributter.

Entiteten har rette hjørner, hvilket er erwins måte å markere at det er en sterk entitet på.


**Figur 14-8: Entitet med navnet EMPLOYEE og tre attributter, med EmployeeId som identifikator.**

## 14.7. Ikke-identifiserende relasjonsforhold (Non-Identifying relationship)

**Ikke-identifiserende relasjonsforhold** er forhold mellom to entiteter, der hver entitet alene kan identifisere seg selv. Dette innebærer at hver entitet har identifikator som ikke trenger «arve» fra den andre entitetens identifikator for å forme en unik primærnøkkel, ved overgang fra logisk til fysisk modell.

Ikke-identifiserende relasjonsforhold kalles også et sterkt forhold, for det knytter sammen såkalte sterke entiteter (strong entities) som kan identifisere seg selv. Slike entiteter markeres i erwin med



**Figur 14-9: En stiplet linje er symbol for et ikke-identifiserende relasjonsforhold.**

## 14.8. Kardinalitet (Cardinality)

Ved bruk av relasjonsforhold mellom entiteter, angis **kardinalitet** for å angi hvor mange det kan være av en forkomst. Man kunne tenkt seg at man for eksempel oppga at én bileier hadde 3–5 biler, men i E/R-modellering er det ikke av verdi for tabellen å vite det eksakte antallet, for å modellere en database.

Det som er av interesse er å vite om det kan registreres:

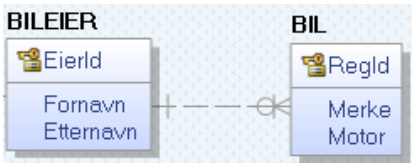
0: Dette vil si at et felt ***kan stå tomt*** når verdier registreres i en rad, altså at NULL-verdi tillates.

1: Dette innebærer at det ***må*** registreres en verdi i feltet. I CREATE TABLE benyttes da NOT NULL.

(mange): Dette innebærer at det kan registreres mange verdier, hvilket angis med en kråkefot.

## 14.9. Ikke-identifiserende en-til-mange forhold (1:M-forhold)

En-til-mange-relasjonsforhold skrives ofte **1:M**-forhold (one-to-many). Et ikke-identifiserende relasjonsforhold er et forhold mellom to entiteter som hver kan identifisere seg selv fullt og helt. Figur 14-10 viser et **1:M**-forhold mellom entitetene **BILEIER** og **BIL**.

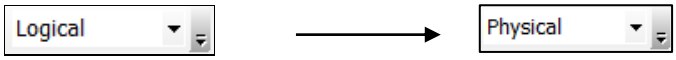



**Figur 14-10: Et ikke-identifiserende 1:M-forhold (stiplet linje) mellom to sterke entiteter.**

Dette forholdet kan leses sånn: ***Én*** bileier eier ***null, en eller flere*** biler, mens ***én*** bil har ***én*** eier. Det er altså et 1:M-forhold mellom entitetene.

Når ikke-identifiserende 1:M-forhold skal omgjøres til fysisk modell, vil identifikatoren på 1-siden komme inn som fremmednøkkel på mange siden som et nytt attributt. Denne fremmednøkkelen blir da en referanse som refererer til (pekker mot) primærnøkkelen på 1-siden.

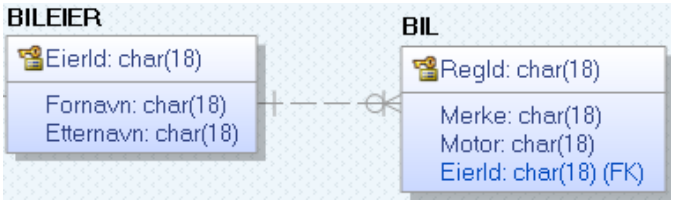
Endre fra «Logical» til «Physical» i menyen, som vist i Figur 14-11, for å se fremmednøkkelen.




**Figur 14-11: Endre fra logisk modell (E/R) til fysisk modell for å se fremmednøkkelen som er påført.**

I den fysiske modellen vist i Figur 14-1, fremkommer det nye fremmednøkkelattributtet på mange-siden, det vil si som et kolonnenavn i **BIL**-tabellen. Legg merke til det nye kolonnenavnet **EierId** er markert med **FK**, for **Foreign Key**. Dette er en referanse til **EierId** i **BIL**.

**NB!** I fysisk modell snakker man normalt om **tabeller**, **kolonner** og **primærnøkler**, istedenfor **entiteter**, **attributter** og **identifikatorer** (som i E/R-diagram/logisk modell). Dette fordi den fysiske modellen representerer den reelle tabellstrukturen som skal genereres i databasen.




**Figur 14-12: E/R-modellen konvertert til fysisk modell.**

## 14.10. Definering av datatyper

I den fysiske modellen vist i Figur 14-1 ses datatypene som vil bli benyttet i erwins generering av script for opprettelse av den fysiske databasen i SQL Server. Der ses det at alle datatypene satt til char (18), fordi dette er satt som default-verdi, som beskrevet i kapittel 14.2.

Datatypene kan omdefineres både i logisk modell og fysisk modell. Selv om datatypene ikke er relevante i den logiske modellen (E/R-diagrammet), er det likevel ofte greiest å definere dem der. Da blir modelldataene konsistente gjennom hele modellen.

Der det foreligger en fremmednøkkel i ***én*** entitet, som peker til en primærnøkkel i en ***annen*** entitet, skal bare datatypen til primærnøkkelen endres. Fremmednøkkelen vil da automatisk få samme datatype.

Høyreklikk over **BILEIER** i logisk model, og velg «**Attribute properties**», for å få visningen vist i Figur 14-3.

| Name | Parent Domain | Logical Data Type | Primary Key | Foreign Key | Logical Only |
|---|---|---|---|---|---|
| EierId | <default> | CHAR(18) | ☑ | ☐ | ☐ |
| Fornavn | <default> | CHAR(18) | ☐ | ☐ | ☐ |
| Etternavn | <default> | CHAR(18) | ☐ | ☐ | ☐ |

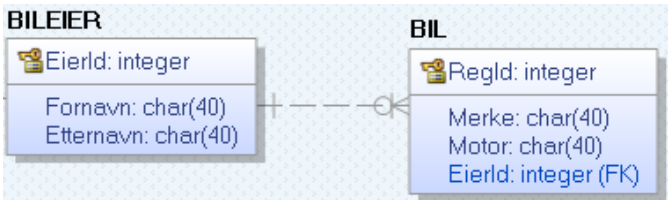
**Figur 14-13: Attribute properties for entiteten BILEIER og BIL.**

Endre **EierId** til **INTEGER** og **Fornavn** og **Etternavn** til **CHAR(40)**, som vist i Figur 14-4.

| Name | Parent Domain | Logical Data Type | Primary Key | Foreign Key | Logical Only |
|---|---|---|---|---|---|
| EierId | <default> | INTEGER | ☑ | ☐ | ☐ |
| Fornavn | <default> | CHAR(40) | ☐ | ☐ | ☐ |
| Etternavn | <default> | CHAR(40) | ☐ | ☐ | ☐ |

**Figur 14-14: Attributtverdiene i entiteten BILEIER er endret.**

Endre deretter (i logisk modell) også attributtene i **BIL** til **INTEGER** for **RegId** og **CHAR(40)** for **Merke** og **Motor**. Visningen i fysisk modellvist i Figur 14-15.




**Figur 14-15: Fysisk modell påført endrete datatyper.**

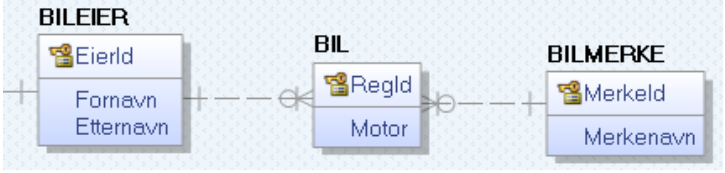
## 14.11. Oppslagstabell

Innføring av oppslagstabeller gjøres når det er spesielle kolonner det ønskes å søke særskilt på, koble til for eksempel til «ComboBox»-er i brukergrensesnitt for å få listet opp innholdet for en bruker eller for å sikre gjennom fremmednøkkel til oppslagstabellen at bare gyldige verdier registreres i fremmednøkkelen som refererer til primærnøkkelen i oppslagstabellen.

I Figur 14-2 var Merke (underforstått bilmerke) registrert som et attributt i entiteten **BIL**. Dette kan være et aktuelt attributt å skille ut i en koblingstabell. I en koblingstabell kan enten bare den aktuelle kolonnen benyttes (i dette tilfellet Merke, der bilmerker som Volvo, Fiat, Opel osv. kan registreres), eller det kan innføres et løpenummer i tillegg, hvilket ofte gjøres. I vårt eksempel gjøres dette.

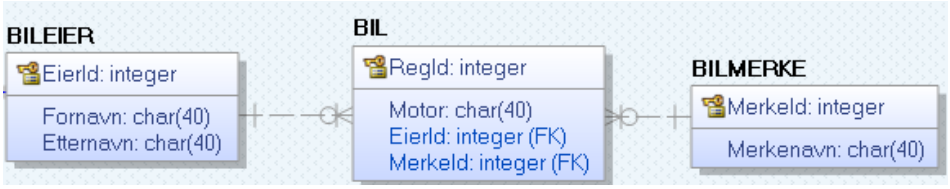
Det første som gjøres er å slette attributtet **Merke** i entiteten **BIL**. Dette gjøres i den logiske modellen, og vil da automatisk også bli gjort i den fysiske modellen. Sletting gjøres ved å klikke en gang på attributtet, så det blir grått, for så å klikke på **Delete**-knappen på tastaturet.

Legg så inn en ny entitet kalt **BILMERKE**, som vist i Figur 14-16.




**Figur 14-16: Bilmerke er i logisk modell skilt ut i en egen entitet. I fysisk modell blir det en oppslagstabell.**

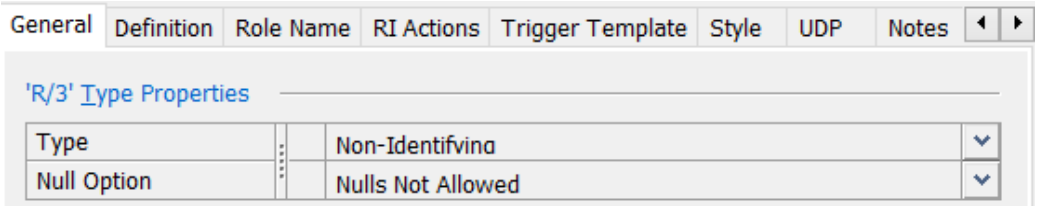
I fysisk modell vil dette bli som vist i Figur 14-17 (der enkelte datatyper først er endret i logisk modell)




**Figur 14-17: Koblingstabellen BILMERKE vist i fysisk modell.**

Som det ses av de to **1:M**-forholdene fra **BIL** mot henholdsvis **BILEIER** og **BILMERKE**, er begge disse markert med | på 1-siden, for å markere at en ***bil*** gjelder nøyaktig ***én*** bileier og ***én*** bil er av nøyaktig ***ett*** bilmerke.

Når forholdet påføres vil det stå 0 istedenfor 1, der 0 betyr «0 eller 1», altså ikke påkrevd (not mandatory). For å endre dette til eksakt 1, dobeltklikk på relasjonen og sett «**Null Option**» til «**Null Not Allowed**» istedenfor «**Nulls Allowed**», som er standardoppsettet. Dette er illustrert i Figur 14-8.




**Figur 14-18: «Null Option» satt til «Nulls Not Allowed», så forholdet påføres | istedenfor 0.**

## 14.12. Identifiserende en-til-mange forhold (1:M-forhold)

I kapittel 14.9 ble det sett på **ikke-identifiserende** 1:M-relasjonsforhold. Disse endte opp i tabeller der identifikatoren på 1-siden ble til en ny kolonne på mange-siden. Denne kolonnen ble samtidig en fremmednøkkel mot identifikatoren den var avledet fra. Relasjonsforholdet ble tegnet med stiplet linje.

Et **identifiserende relasjonsforhold** (**Identifying Relationships**) tegnes i logisk E/R-diagram mellom entiteter der en av dem ikke kan identifisere sine attributter unikt uten «hjelp» fra den andre entiteten. Slike entiteter kalles derfor **svake entiteter** (**Weak Entities**) eller **avhengige entiteter** (**Dependent Entities**). Dette er også grunnen til at relasjonsforholdet kalles **identifiserende** (**Identifying**) fordi det angir at ***én*** entitet er med på å identfisere ***en annen*** entitet. En heltrukken linje benyttes som symbol for å angi et identifiserende forhold, jf. Figur 14-19.

Den svake entiteten markeres i tillegg av erwin med avrundete istedenfor rette hjørner.



**Figur 14-19: En heltrukken linje benyttes som symbol for et identifiserende relasjonsforhold.**

For å eksemplifisere et identifiserende relasjonsforhold, utvides E/R-diagrammet med en entitet for å registrere data om konkurranser (billøp) som gjennomføres. En del av billøpene også har premier for et utvalg plasseringer, for eksempel for 1, 2 og 3. plass og noen ganger kanskje for flere plasseringer. Det er ønskelig å kunne registrere disse premiene i databasen før løpene gjennomføres, og de legges derfor i en egen entitet. Som følge av dette utvides diagrammet med entitetene vist i Figur 14-20.

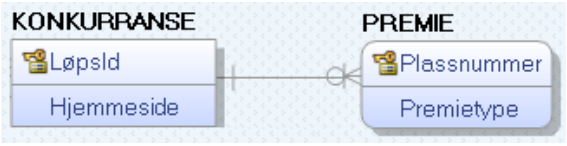

**Figur 14-20: E/R-diagrammet utvides med de to entitetene KONKURRANSE og PREMIE.**

Forholdet mellom KONKURRANSE og PREMIE vil være et 1:M-forhold. Én konkurranse vil kunne ha mange premier og én premie (1. premie, 2. premie, 3. premie, osv.) vil benyttes i mange konkurranser.

Plassnummer er i dette tilfellet et tall i stigende rekkefølge: 1, 2, 3 osv., mens Premietypene vil kunne variere. Siden mange av konkurransene vil ha like plassnummer (1, 2, 3, …), vil ikke PREMIE være i

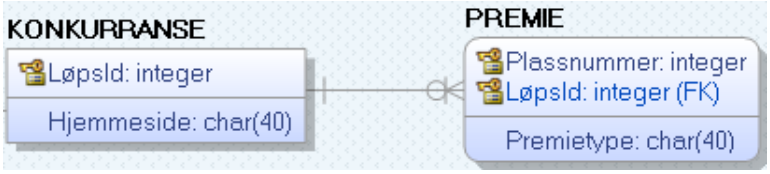
stand til å identifisere seg selv alene. Den er en «svak entitet». For å identifisere seg, må **PREMIE** vite hvilken **KONKURRANSE** den skal knyttes opp mot. Dette betyr at **LøpsId** må inngå som del av primærnøkkelen til **PREMIE**, når denne skal bli til en tabell i den fysiske modellen.

Dette løses ved å benytte et identifiserende relasjonsforhold (Identifying Relationship) mellom de to entitetene. Ved å gjøre **PREMIE** svak, vil den arve identifikatoren fra **KONKURRANSE** og gjøre den til en del primærnøkkelen i den fysiske modellen. Det identifiserende relasjonsforholdet påføres med en heltrukken linje, som vist i Figur 14-21. Legg også merkes til at **PREMIE** for avrundete hjørner, som en konsekvens av at den nå er definert som en svak (dependent) entitet.




**Figur 14-21: Et identifiserende 1:M-forhold er påført mellom entitetene KONKURRANSE og PREMIE.**

Dette kan leses sånn at én konkurranse kan registreres med 0, 1 eller mange premier, mens én konkret premie som registreres, skal knyttes til én spesifikk konkurranse. Den fysiske modellen er vist i Figur 14-22. Datatypene er endret i den logiske modellen før visning av den fysiske modellen.

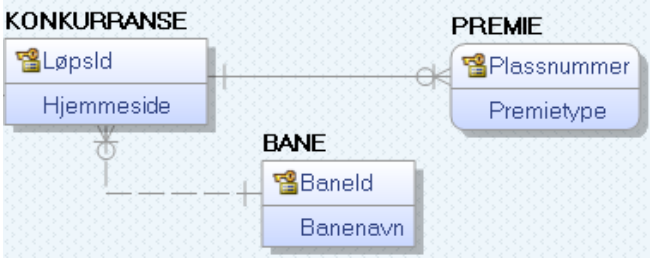



**Figur 14-22: Overgang til fysisk modell fra logisk modell (E/R-diagram) med identifiserende forhold.**

**LøpsId** har kommet inn som en kolonne i tabellen **PREMIE** og denne kolonnen har blitt til ***både*** en **fremmednøkkel** (FK: Foreign Key) og en **del av primærnøkkelen** (som en referanse til **LøpsId** i tabellen **KONKURRANSE**).

## 14.13. Legge til en ny oppslagstabell

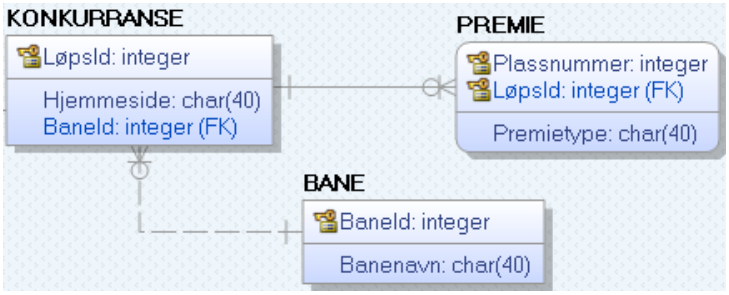
Det ønskes å registrere hvilke baner konkurransen skal foregå på. Banene ønskes registrert i en egen tabell, så de er enkelt tilgjengelig for registrering i **KONKURRANSER**. De legges derfor i en egen tabell kalt **BANE**, som kobles til **KONKURRANSE**. Dette er to tabeller som hver for seg kan identifisere seg selv unikt med sin egen identifikator, så her kan det benyttes et ikke-identifiserende 1:M-relasjonsforhold mellom entitetene, som vist i __.




**Figur 14-23: E/R-diagrammet utvides med entiteten BANE.**

Det nye 1:M-relasjonsforholdet kan leses sånn: ***Én*** konkurranse arrangeres på ***én*** spesiell bane, mens én bane kan benyttes i mange konkurranser (billøp).

Resultatet blir at **KONKURRANSE** får **BaneId** som en ny kolonnen. Denne blir en fremmednøkkel (FK), men blir ikke en del av primærnøkkelen, siden det er et ikke-identifiserende relasjonsforhold. Den fysiske modellen er vist i Figur 14-24.

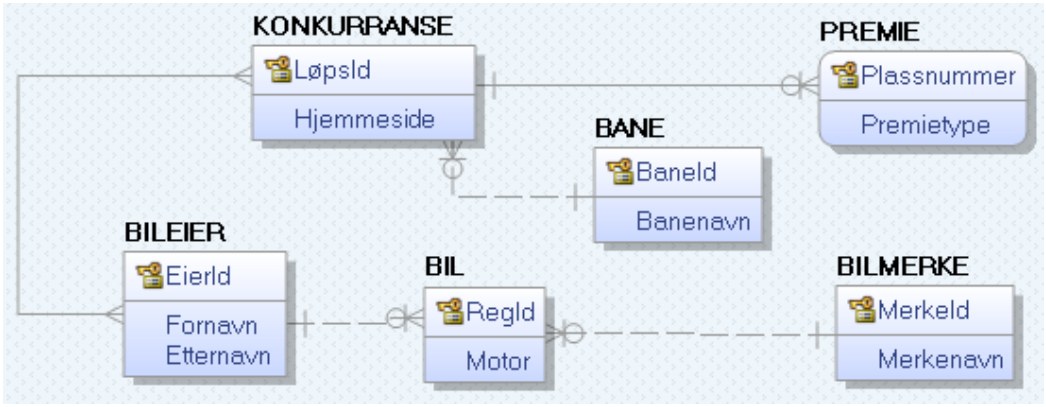



**Figur 14-24: Fysisk modell som viser utvidelsen av modellen med tabellen BANE.**

## 14.14. Mange-til-mange-forhold (M:N-forhold)

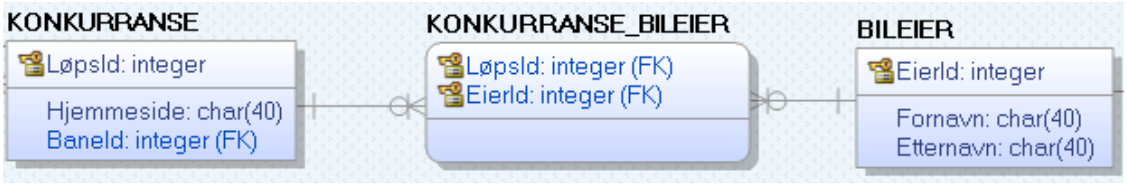
I en del tilfeller vil det være et en-til-mange-forhold mellom entiteter (ofte skrevet M:N-forhold). Dette skal eksemplifiseres ved av det skal registreres bileiere i konkurransen. Én bileiere kan delta i mange konkurranser, og én konkurranse kan ha mange deltakere. Dette representere derfor et M:N-forhold.

I erwin-menyen er det tilgjengelig et verktøy for å påføre «Many-to-many-relationship», som tidligere illustrert i kapittel 14.4. Legg til et M:N-forhold mellom entitetene KUNKURRANSE og BILEIER, så E/R-diagrammet blir som vist i Figur 14-25.




**Figur 14-25: E/R-diagram (logisk modell) med et mange-til-mange-forhold inkludert.**

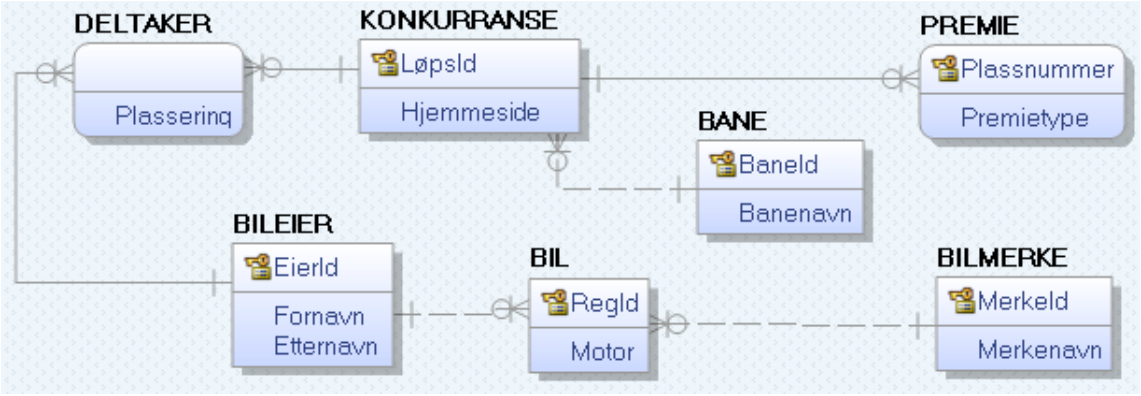
M:N-forhold lar seg ***ikke*** realisere i en database og må derfor løses opp i 2 stk. 1:M-forhold. Dette vil man se er utført automatisk i den fysiske modellen, som vist i Figur 14-26. Den nye tabellen har erwin kalt KONKURRANSE_BILEIER, som refererer til de to navnene entiteten refererer til. Dette navnet kan endres manuelt. Det anbefales i så fall å gjøre dette i den logiske modellen, fordi navnet da blir gjeldende både i den logiske og den fysiske modellen.




**Figur 14-26: I den fysiske modellen er M:N-forholdet løst opp i to stk. 1:M-forhold.**

Legg merke til at erwin har gjort de to 1:M-forholdene til identifiserende relasjonsforhold. Derfor har også entiteten **KONKURRANSE_BILEIER** fått avrundete hjørner, for å markere at dette nå er en svak entitet (weak/dependent entity).

Vi skal nå omforme E/R-diagrammet til å samsvare med den fysiske modellen. Gå til logisk modell og klikk på M:N-forholdet mellom **KONKURRANSE** og **BILEIER** så dette blir markert. Klikk deretter på «Resolve All Transformations»-verktøyet i menyen (illustrert i kapittel 14.4). Det etableres da en ny entitet mellom **KONKURRANSE** og **BILEIER**. Gi denne navnet **DELTAKER** og legg til et ikke-nøkkel-attributt kalt **Plassering**. Resultatet er vist i Figur 14-27.



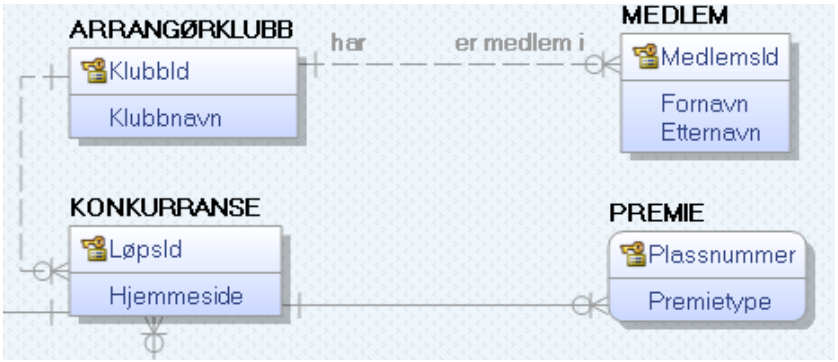

**Figur 14-27: E/R-diagrammet (logisk modell) utvidet med DELTAKER-entiteten.**

I E/R-diagrammet har **DELTAKER** ingen identifikator påført. Dette er fordi den arver sine identifikator-attributter fra henholdsvis **KONKURRANSE** (attributtet **LøpsId**) og **BILEIER** (attributtet **EierId**). Årsaken til det, er det to identifiserende relasjonsforholdene.

Disse to attributtene i kombinasjon, vil i den fysiske tabellen bli primærnøkkel i tabellen **DELTAKER**. **LøpsId** vil i tillegg bli en fremmednøkkel mot tabellen **KONKURRANSE** og **EierId** en fremmednøkkel mot tabellen **BILEIER**. Attributtet **Plassering**, vil bli en ikke-nøkkel-kolonne i den samme tabellen.

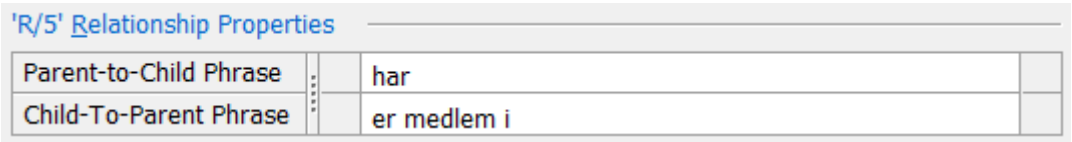
## 14.15. Påføring av roller

Det er ønskelig å utvide E/R-modellen med data om arrangørklubbene som arrangerer konkurransene og hvilke medlemmer som erknyttet til disse. Dette skal gjøres ved å innføre av to nye entiteter kalt **MEDLEM** og **ARRANGØRKLUBB**. Relasjonene mellom disse og mellom **ARRANGØR** og **KONKURRANSE** blir **ikke-identifiserende** 1:M-relasjonsforhold, som vist i Figur 14-28.




Enkelte ganger kan det være av interesse å påføre roller på et relasjonsforhold, for å tydeliggjøre hva relasjonen gjelder. Dobbeltklikk da på relasjonsforholdet. Under fanen «General» kan roller så påføres, som vist i Figur 14-29.

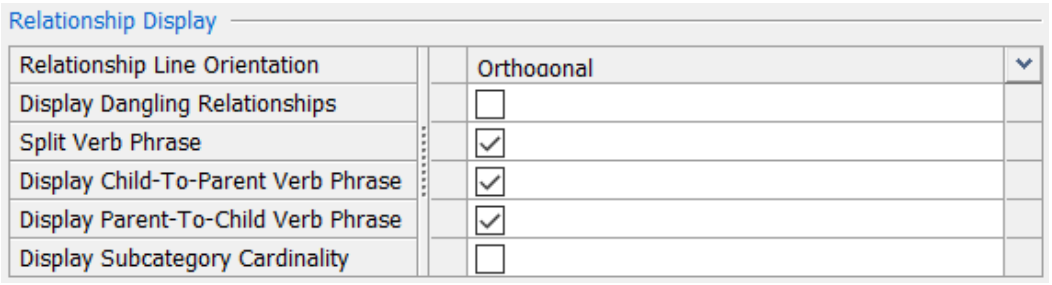
**Figur 14-28: To nye entiteter er lagt til og ikke-identifiserende 1:M-relasjonsforhold er påført.**




**Figur 14-29: Definisjon av roller for et relasjonsforhold.**

For at rollene skal vises i skjemaet, som vist i Figur 14-28, dobbeltklikk på et tomt område på skjemaet, velg fanen «Display» og avkryss de tre sjekkboksene vist i Figur 14-30.

Rollene i figuren kan leses sånn: En **ARRANGØRKLUBB** «har» 0, 1 eller mange medlemmer, mens ett **MEDLEM** «er medlem i» én arrangørklubb.




**Figur 14-30: Konfigurasjon for visning av roller påført relasjonsforhold.**

## 14.16. En-til-en-relasjoner (1:1)

En-til-en-relasjoner inntreffer ikke så ofte, men er nyttige i noen situasjoner. I vårt eksempel skal det legges til et 1:1-forhold mellom ARRANGØRKLUBB og MEDLEM, der det allerede er et 1:M-forhold. 1:M-forholdet representerer medlemskapet mellom medlemmer og klubbene, mens 1:1-forholdet skal modellere hvem som er leder av hver arrangørklubb. Det settes som forutsetning at en leder av en klubb også er medlem av klubb.

1:1 forhold tegnes normalt med | på den ene siden og 0 på den andre. | betyr da eksakt 1, mens 0 betyr 0 eller 1. Det å ha eksakt 1 i hver ende av et 1:1-forhold er svært sjeldent, da det da normalt i de fleste tilfeller vil være mer naturlig at alle entitetene ligger i en og samme entitet.

Modelleringen skal fremstille følgende relasjon: Ett medlem kan, men må ikke, være leder av en klubb. Dette blir derfor 0-siden av relasjonsforholdet. En klubb har samtidig eksakt ***ett*** medlem som sin leder. Dette blir dermed 1-siden av forholdet.

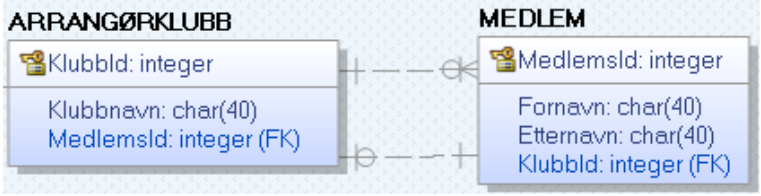
Start med å trekke en ikke-identifiserende relasjon ***fra* MEDLEM *til* ARRANGØRKLUBB**. Dobbeltklikk deretter på relasjonsforholdet. Under fanen «General», sett «Null Option» to «Nulls not allowed» og Cardinality til «Zero or one».

Legg også på rollenavn på samme måte som det ble gjort i kapittel 14.15. Dette gjør det mulig å skille de to relasjonsrollene mellom **ARRANGØRKLUBB** og **MEDLEM** fra hverandre, ved å beskrive hva hver av dem gjør. Det nye 1:1-relasjonsforholdet mellom entitetene kan da leses sånn: ***Ett* MEDLEM** «er leder i» 0 eller 1 **ARRANGØRKLUBB** (det vil si at vedkommende ikke nødvendigvis er en leder, men ***kan*** være det) og ***én* ARRANGØRKLUBB** «ledes av» nøyaktig ***ett* MEDLEM**.


**Figur 14-31: Relasjonsforhold påført roller i E/R-diagrammet (logisk modell).**

Man leser altså først et entitetsnavn, deretter rollen som står nærmest denne, og man relaterer det så til den andre entiteten, sammen med dennes kardinalitet.

Når det foreligger et 1:1-forhold med 0 på den ene siden av forholdet, vil denne 0-siden få identifikatoren fra 1-siden (i dette tifellet **MedlemsId**) som en ny kolonne i den fysiske modellen. Siden det er et ikke-identifiserende forhold, blir den ikke en del av primærnøkkelen, men den blir en fremmednøkkel med referanse til **MedlemsId** i **MEDLEM**. I Figur 14-32 vises den fysiske modellen.



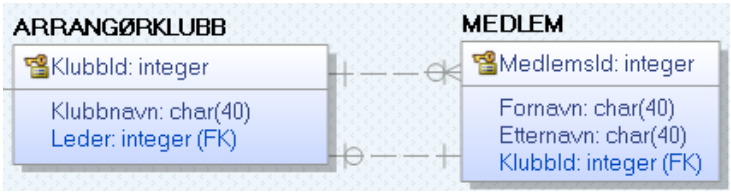

**Figur 14-32: Fysisk modell med primærøkler**

## 14.17. Ulikt navn på fremmednøkkel og primærnøkkelen den refererer

Normalt har vi operert med samme navn på fremmednøkkelen og primærnøkkelen den refererer. Dette er ofte praktisk, men av og til gir det bedre beskrivende kolonnenavn og ha ulike navn på disse.

I tilfellet illustrert i Figur 14-32, kunne det for eksempel vært mer beskrivende å lag fremmednøkkelen MedlemsId i ARRANGØRKLUBB få navnet «Leder» istedenfor «MedlemsId». Selv om det fortsatt vil være et medlemsnummer, sier det mer om akkurat dette medlemmets funksjon og kan dermed egne seg bedre som kolonnenavn. Den vil fortsatt referere til kolonnen med navn MedlemsId i MEDLEM, der den er primærnøkkel.

En endring av navnet på en kolonne, må gjøres i den fysiske modellen. Klikk på attributtet MedlemsId i ARRANGØRKLUBB og gi den isteden navnet «Leder», som vist i _.



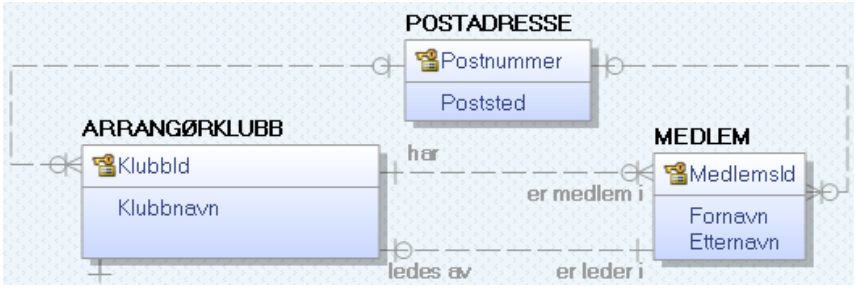

**Figur 14-33: Kolonnen MedlemsId i ARRANGØRKLUBB er omdøpt til Leder i den fysiske modellen.**

## 14.18. Koble en entitet til flere andre entiteter

I noen tilfeller kan entiteter være så generelle at de kan benyttes i flere relasjoner. Som et eksempel, tenker vi oss at både arrangørklubb og medlemmer ønskes registrert med postnummer og postadresse.

Dette kan gjøres i E/R-diagrammet (logisk modell) ved å lage en entitet kalt **POSTADRESSE**, som inneholder attributtene **Postnummer** og **Poststed**, der postnumre er unike og kan være identifikator.

I Figur 14-34 vises det hvordan **POSTADRESSE** er koblet til de to entitetene. Dette kan leses sånn: Én postadresse kan gjelde mange arrangerklubber og medlemmer, mens hvert medlem og hver arrangør-klubb har hver sin spesifikke postadresse.

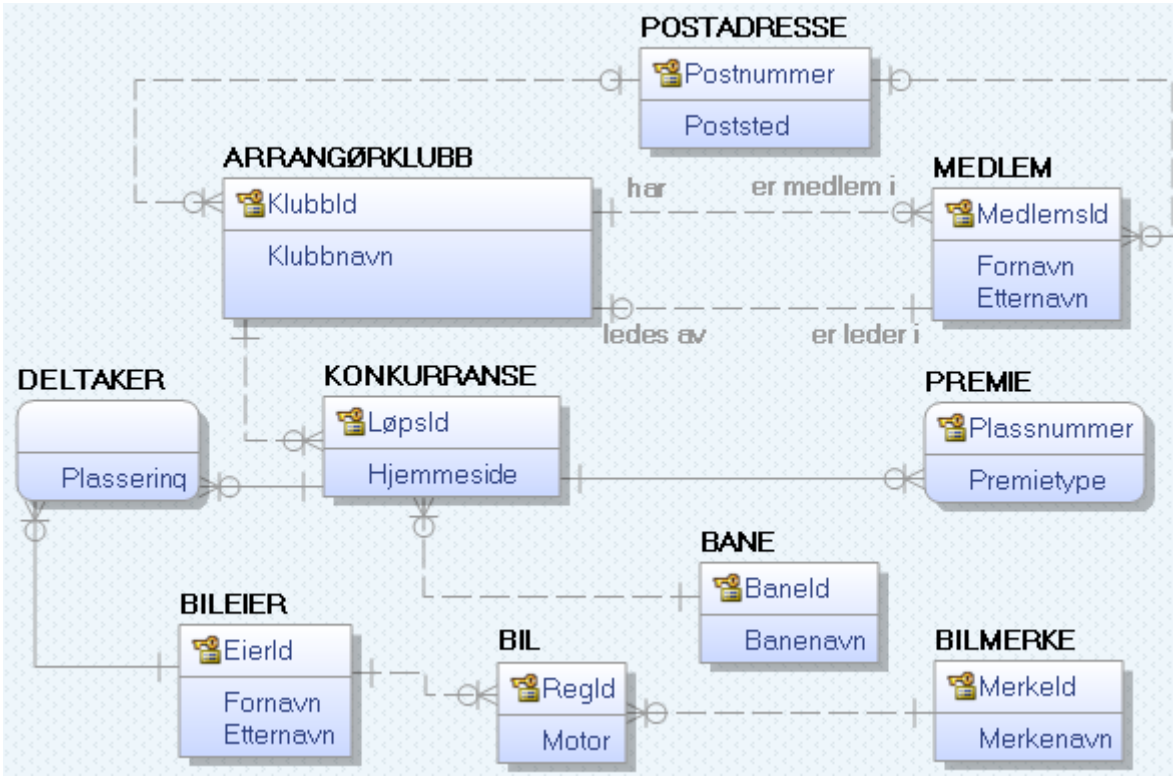



**Figur 14-34: Ny entitet POSTADRESSE er koblet til både ARRANGØRKLUBB og MEDLEM.**

## 14.19. Det endelige E/R-diagrammet

E/R-modelleringen er nå avsluttet. Det er vist hvordan man stadig kan utvide diagrammet med ny funksjonalitet, dersom man modellerer fornuftig underveis. Man må alltid anta at det vil kunne komme behov for å lagre mer data enn det som er tenkt på i den opprinnelige modellen, og en strukturert modellert database vil da gjøre dette mulig uten å ødelegge den eksisterende strukturen.

I Figur 14-35 vises den helhetlige E/R-modellen vi har kommet frem til gjennom modelleringen.




**Figur 14-35: Ferdig modellert E/R-diagram.**

## 14.20. Fysisk tabell – Tabeller m.m. som inngår i sluttresultatet

Med regelverket for relasjoner som er etablert og beskrevet, er det enkelt å gå fra logisk til fysisk modell, for å se hvilke tabeller, primærnøkler, fremmednøkler (og datatyper) man ender opp med. Dette kan man også lette se ved å gå til «fysisk modell» i erwin. I dette tilfellet ender vi med følgende tabeller, der understrekning markerer primærnøkkel og * markerer fremmednøkkel:

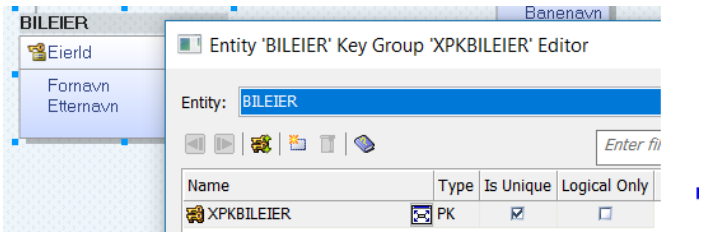
- BILEIER (EierId, Fornavn, Etternavn)
- BIL (RegiId, Motor, EierId*, MerkeId*)
- BILMERKE (MerkeId, Merknavn)
- BANE (BaneId, Banenavn)
- KONKURRANSE (LøpsId, Hjemmeside, KlubbId*, BaneId*)
- DELTAKER (EierId*, LøpsId*, Plassering)
- ARRANGØRKLUBB (KlubbId, Klubbnavn, Postnummer*, Lede nr*)
- MEDLEM (MedlemsId, Fornavn, Etternavn, Postnummer*, KlubbId*)
- PREMIE (Plassnummer, LøpsId*, Premietype)

## 14.21. Hvordan endre identifikatornavn for primærnøkler

Når det skal genereres tabeller, benytter erwin følgende navnekonvensjon: Er for eksempel kolonnen **EierId** primærnøkkel i tabellen **BILEIER**, navngir erwin identifikatoren som brukes i **CREATE TABLE**-spørringen **XPKBILEIER**, mens vi har benyttet **PK_BILEIER** som navn (jf. kapittel 4.1).

Det er noen muligheter for å lage script for å styre navnekonvensjonen i erwin, men for endre navn på identifikator uten å måtte lage slike konfigureringer, beskrives en fremgangsmåte her.

Stå i logisk modell, høyreklikk over en entitet, for eksempel **BILEIER**, og velg så «**Key Group Properties**». Da vil det fremkomme en dialogboks som vist i 14-36. I denne kan identifikatoren, som i dette tilfellet heter **XPBILEIER**, endres manuelt til for eksempel **PK_BILEIER**.




**14-36: Dialogboks der fremmednøkkelidentifikatoren (XPKBILEIER) vises og kan endres.**

I Figur 14-37 vises eksempel på en erwin-generert tabell (fra script) etter navnsetting av primær- og fremmednøkkelidentifikatorer i E/R-diagrammet (jf. kapittel 14.20 og kapittel 14.21).

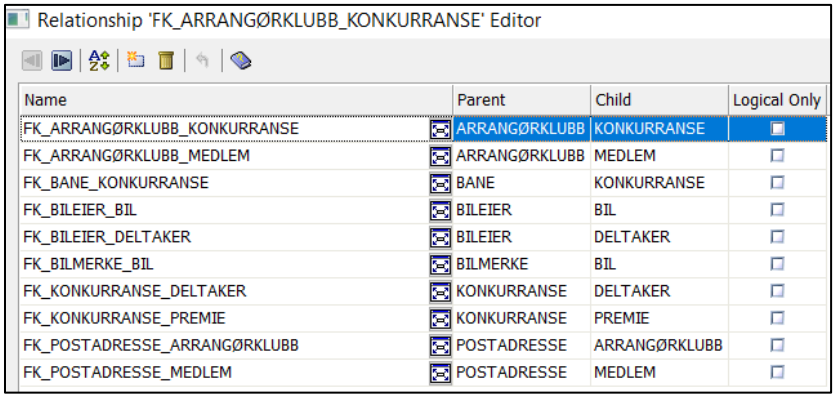
```
CREATE TABLE ARRANGØRKLUBB
(
        KlubbId          integer  NOT NULL ,
        Klubbnavn         char(40)  NULL ,
        Postnummer        char(18)  NULL ,
        CONSTRAINT PK_ARRANGØRKLUBB PRIMARY KEY  CLUSTERED (KlubbId ASC),
        CONSTRAINT FK_POSTADRESSE_ARRANGØRKLUBB FOREIGN KEY
                          (Postnummer) REFERENCES POSTADRESSE(Postnummer)
)
```

**Figur 14-37: Eksempel på tabell generert av erwin med egendefinerte nøkkelidentifikatorer.**

## 14.22. Hvordan endre identifikatornavn for fremmednøkler

I kapittel 14.21 ble det vist hvordan man kunne endre navn på primærnøkkelidentifikatorer, for å få de til å følge navnekonvensjonen. Tilsvarende kan gjøres for fremmednøkler.

Dobbeltklikk (i logisk modell) på et av relasjonsforholdene. Da kommer det opp et vindu, der relasjons-identifikatorenes navn vises, der disse også kan endres. I 14-38 er endringer utført i henhold til navne-konvensjonen: **FK_fraTabell_tilTabell**, som ble beskrevet i kapittel 7.2. Endringene er relativt raskt å utføre, da det for alle identifikatorene vises «**Parent-table**» (det vil si tabellen fremmednøkkelen ligger i) og «**Child-table**» (det vil si tabellen fremmednøkkelen refererer til) oppført til høyre.



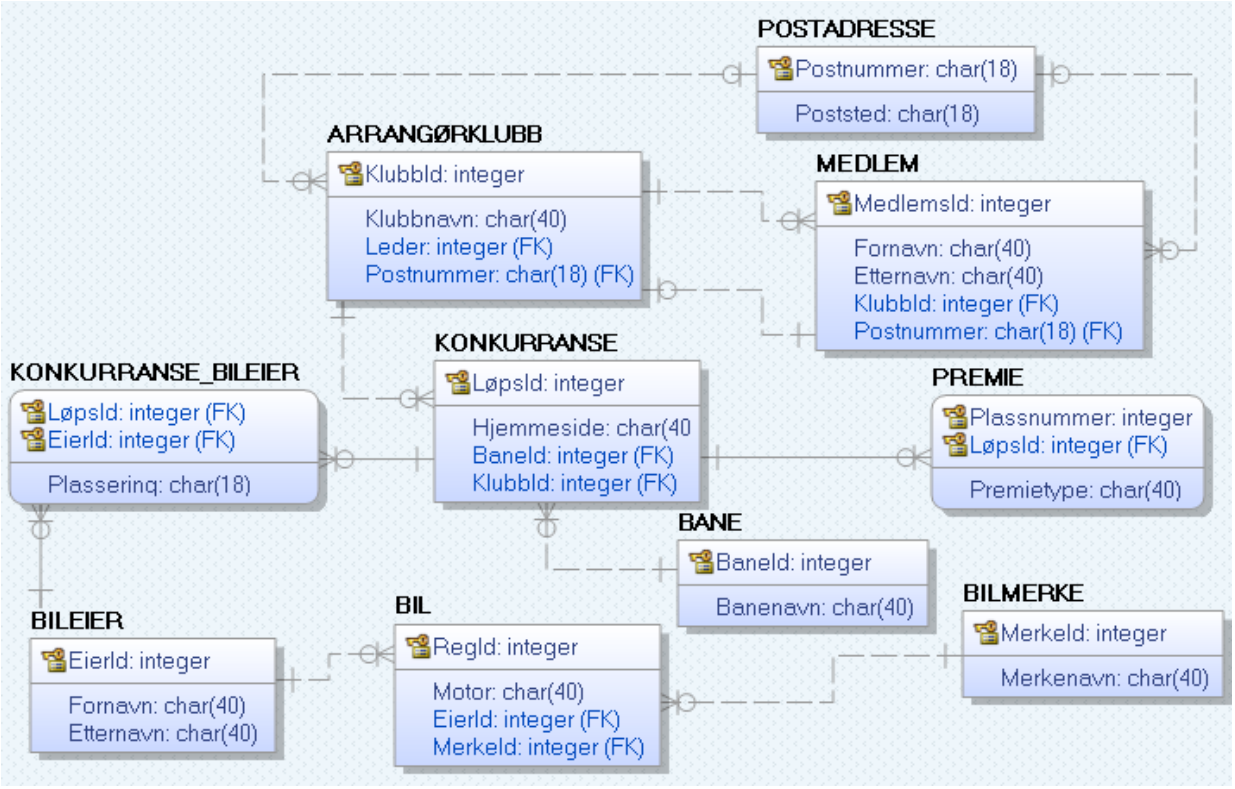

**14-38: Navnsetting av fremmednøkkelidentifikatorer.**

## 14.23. Generere en databasestruktur ut fra den fysiske erwin-modellen

Ut fra den fysiske modellen **erwin** har laget ut fra E/R-diagrammet (den logiske modellen), kan det enkelt genereres et script som kan benyttes til å lage databasestrukturen i SQL Server. Man må stå i «Physical mode» når dette skal gjøres.

Man kan lage en kobling mot databasen og kjøre scriptet direkte mot denne. Da velger man menyene ***Actions → Target Database → Database Connection*** og konfigurerer oppkoblingen. Man bør da huske å gå tilbake og velge «**Disconnect**» når databasen er ferdig opprettet.

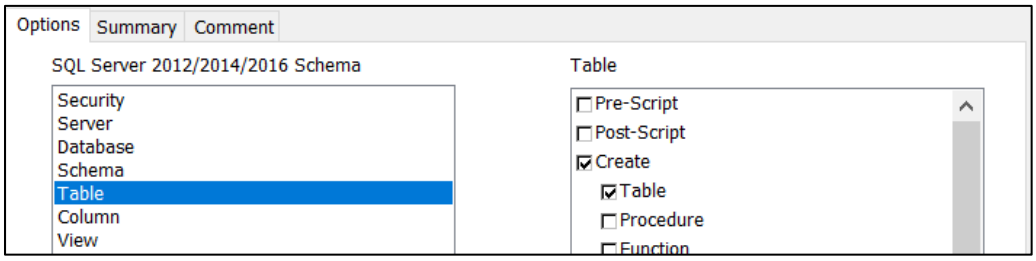
Ofte er det likevel enklere bare å generere et script, for deretter å kopiere dette inn i et SQL Server-spørrevindu og kjøre det derfra. Da slipper man eventuelle oppkoblingsproblemer mot databasen. Her ses det derfor på denne metoden. Den fysiske modellen er vist i Figur 14-39.




**Figur 14-39: Fysisk modell med primærnøkler, fremmednøkler og datatyper påført.**

Gjør (i fysisk modell) menyvalget ***Actions → Forward Engineering → Schema***. Konfigurer så disse: **Table** (Figur 14-40), **Referential Integrity** (Figur 14-41) og **Other Options** (Figur 14-42).

For alle de øvrige valg (Security, Server osv.) kan alle avkrysninger fjernes.

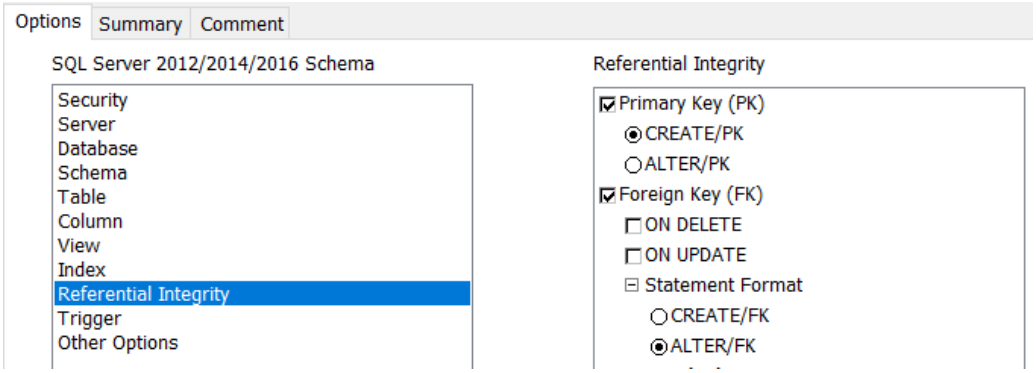



**Figur 14-40: Konfigurering av «Table».**

I konfigureringen vist i Figur 14-41, er det spesielt viktig å avmerke «**ALTER/FK**» for «**Foreign Key**». Dette gjelder spesielt når man han entiteter der det går fremmednøkkel fra den ene tabellen til primærnøkkelen i den andre ***og*** tilsvarende motsatt veg. Dette er en type relasjonsforhold får mellom **MEDLEM** og **ARRANGØRKLUBB**, etter at 1:1 forholdet ble lagt til (i tillegg til 1:M-forholdet).

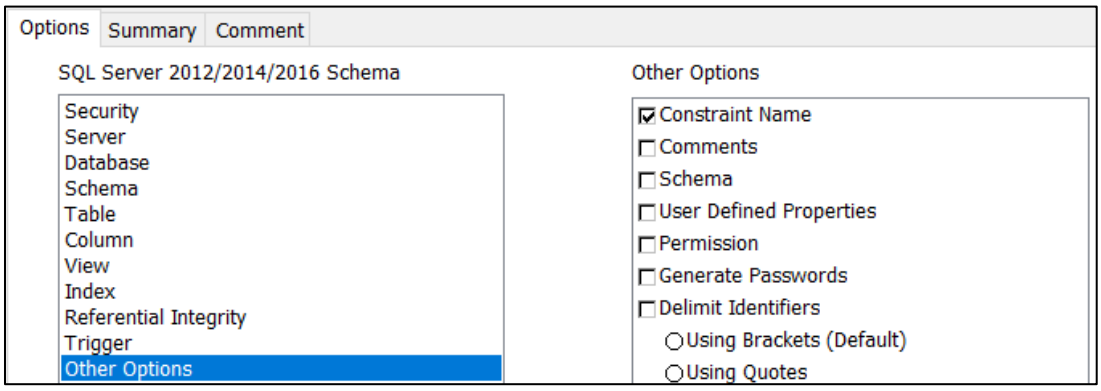
Problemet skyldes at hvis fremmednøklene forsøkes etablert som en integrert del av **CREATE TABLE**-spørringene, vil fremmednøklene peke til tabeller som foreløpig ikke finnes. Dermed inntreffer det en referanseintegritetsfeil, og databasen blir ikke laget fullstendig.

Dersom man istedenfor **CREATE/FK** velger **ALTER/FK**, blir alle tabellene opprettet først, og ***etter*** dette legges så fremmednøklene til med **ALTER TABLE**-instrukser. Når fremmednøklene da legges til, eksisterer allerede alle tabellene, og referanseintegritetsfeil inntreffer ikke. **ALTER/FK** er derfor det tryggeste alternativet, selv om **CREATE/FK** vil fungere i de fleste tilfeller (men ikke i vårt tilfelle).



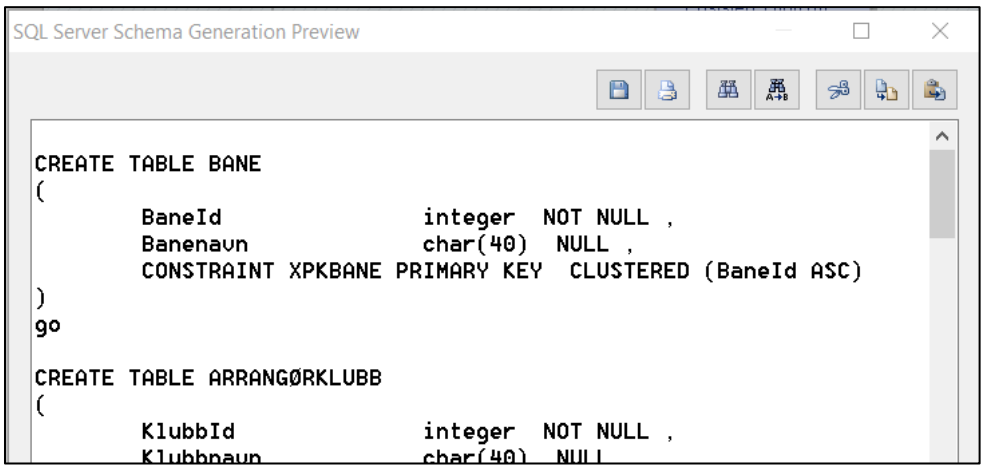

**Figur 14-41: Konfigurering av «Referential Integrity»**

Under «**Other Options**» behøves det bare å velges «**Constraint Names**» (jf. Figur 14-42).




**Figur 14-42: Konfigurering av «Other Options».**

Når konfigurereringen er ferdig, klikk da på **Preview**-knappen, så genereres scriptet. Se Figur 14-43.




**Figur 14-43: Utdrag av scriptet som erwin genererer.**

Dette scriptet inneholder **CREATE TABLE**-spørringer for å generere alle tabellene som er modellert, inkludert alle kolonner, primærnøkler, fremmednøkler og datatyper. Ønskes det at scriptet også skal generere databasen, så legg til linjene vist i Figur 14-4 øverst i scriptet (som det første som utføres).

```
CREATE DATABASE BilraceDatabase
go
USE BilraceDatabase
go
```

**Figur 14-44: Instruksjoner lagt til øverst i scriptet, for eventuelt å opprette ny database og stille seg i den.**

Kopier så scriptet til utklippstavla (Copy/Ctrl+C). Gå så til SQL Server, åpne en «new Query» og lim inn scriptet (Paste/Ctrl+V). Litt av dette er vist i Figur 14-45 (men det inneholder langt flere tabeller).

```
CREATE DATABASE CarRaceDatabase
GO
USE  CarRaceDatabase
GO

CREATE TABLE ARRANGØRKLUBB
(
        KlubbId             integer  NOT NULL ,
        Klubbnavn           char(40)  NULL ,
        Postnummer          char(18)  NULL ,
        Leder               integer  NOT NULL ,
        CONSTRAINT PK_ARRANGØRKLUBB PRIMARY KEY  CLUSTERED (KlubbId ASC)
)
go
        Øvrige CREATE TABLE-spørringer utføres alle før ALTER TABLE-spørringene starter


ALTER TABLE ARRANGØRKLUBB
        ADD CONSTRAINT FK_POSTADRESSE_ARRANGØRKLUBB FOREIGN KEY (Postnummer) REFERENCES POSTADRESSE(Postnummer)
go

ALTER TABLE ARRANGØRKLUBB
        ADD CONSTRAINT R_28 FOREIGN KEY (Leder) REFERENCES MEDLEM(MedlemsId)
go
        Øvrige ALTER TABLE-spørringer
```

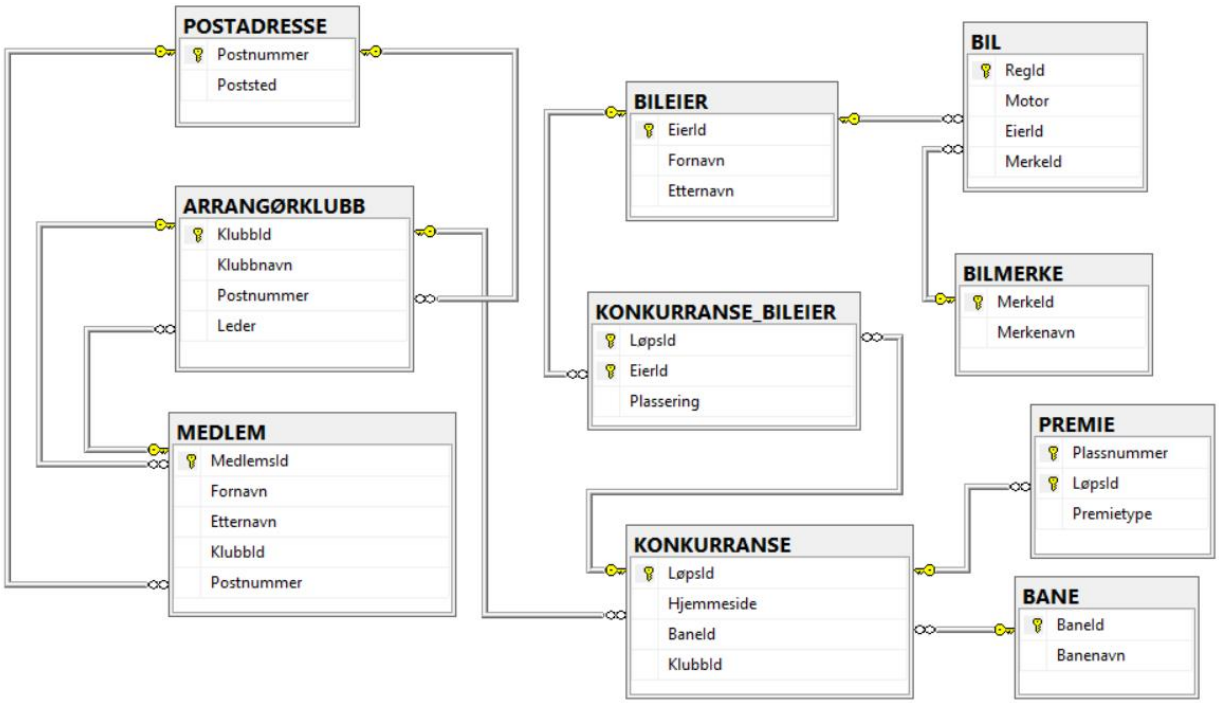
Kode lagt inn manuelt for å opprette tabell og «plassere seg» i denne.

Eksempel på én av **CREATE TABLE**-spørringene

Eksempel på to av **ALTER TABLE**-spørringene

**Figur 14-45: Utdrag fra scriptet som "klippes ut" fra erwin og som deretter utføres i SQL Server.**

Etter å ha kjørt scriptet, som bør kunne kjøres uten feilmeldinger, dersom beskrivelsene er fulgt, så lag en diagramvisning i SQL Server for å se at modellen er i overensstemmelse med E/R-modelleringen. I _ er resultatet vist i diagramvisning.

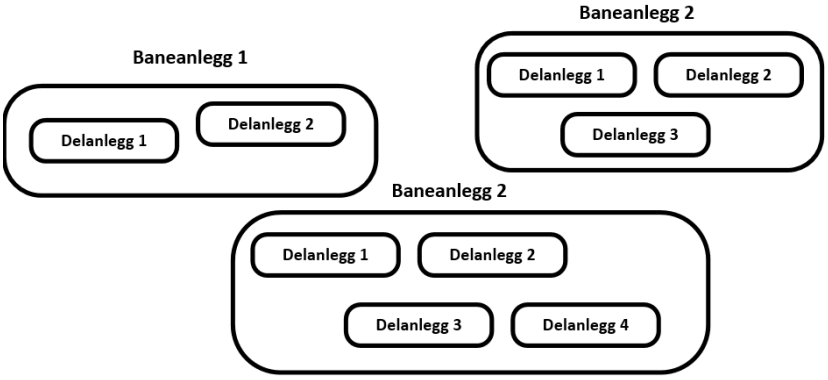



**Figur 14-46: Tabellstruktur vist i diagramverktøyet til SQL Server etter databaseopprettelsen.**

I de to neste kapitlene forklares rekursive relasjonsforhold og relasjonsforhold der flere 1:M-forhold eksisterer mellom to entiteter. Dette er litt sjeldnere relasjonsforhold, men nyttige i noen sammenhenger. Samme bil-race-tema benyttes, men det inkluderes ikke i tabellen som til nå er laget.

## 14.24. Rekursive relasjonsforhold (Recursive Relationships)

Egenforhold, eller rekursive forhold, er forhold der samme entitet inngår i et relasjonsforhold ***fra*** seg selv ***til*** seg selv. Det skal ses på et eksempel, der **BANE**-tabellen omdøpes til **BANEANLEGG**. Det antas videre at ett baneanlegg kan bestå av flere **delanlegg**, som illustrert i Figur 14-47

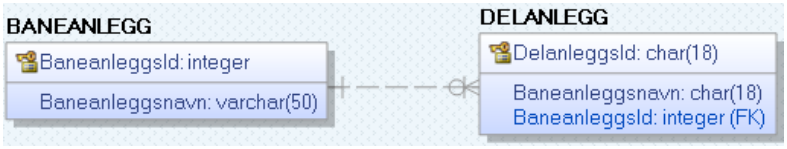



**Figur 14-47: Illustrasjon av tre baneanlegg som hver består av flere delanlegg.**

E/R-diagram kan da modelleres som et 1:M-forhold, der ett baneanlegg består av 0, 1 eller mange delanlegg og der ett konkret delanlegg ligger innunder ett baneanlegg, som vist i Figur 14-48.


**Figur 14-48: E/R-diagram for baneanlegg som består av flere delanlegg.**

Den fysiske modellen blir som vist i Figur 14-49.




**Figur 14-49: Fysisk modell for baneanlegg som består av flere delbaneanlegg.**

Ved å se nærmere på entitetene, ser man at et delanlegg ikke er noe annet enn et baneanlegg, bare at det er underordnet et annet anlegg. De inneholder prinsipielt de samme attributtene, dvs. ett attributt som identifiserer anlegget og ett som angir navnet.

Av den fysiske modellen ses det at man vil ende opp med følgende to tabeller:

- BANEANLEGG (BaneanleggsId, Baneanleggsnavn)
- DELANLEGG (DelanleggsId, Baneanleggsnavn, BaneanleggsId*)

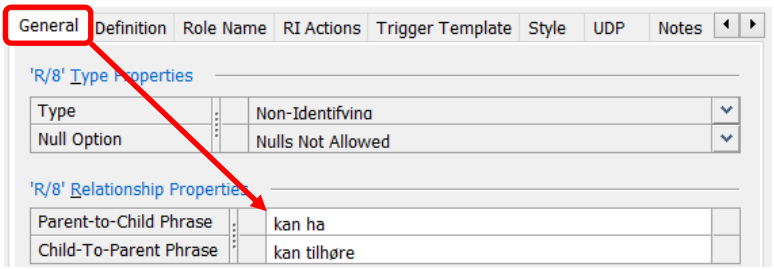
Siden dette handler om prinsipielt like entiteter, bare med overordnede og underordnede nivåer, kan dette også modelleres som et rekursivt forhold. Her er det bare snakk om to nivåer, men prinsipielt kunne det vært enda flere nivåer, ved f.eks. at det delanlegg igjen hadde sine delanlegg osv.

Det rekursive forholdet er vist i Figur 14-50, der **BANEANLEGG** er vist med et relasjonsforhold til seg selv. ***Ett*** baneanlegg (delanlegg) «**kan tilhøre**» ***ett*** overordnet baneanlegg (hovedanlegg), mens ett baneanlegg (hovedanlegg) «**kan ha**» 0 ett eller mange baneanlegg (delanlegg).

For å tegne det rekursive forholdet, velges **1:M non-identifying relationship**. Klikk så to ganger på entiteten, med litt pause mellom klikkene. Da påføres det rekursive forholdet. Klikk deretter på relasjonsforholdet og konfigurer kardinalitetene (med ***0, 1 eller mange*** på ene siden og ***1*** på andre siden).

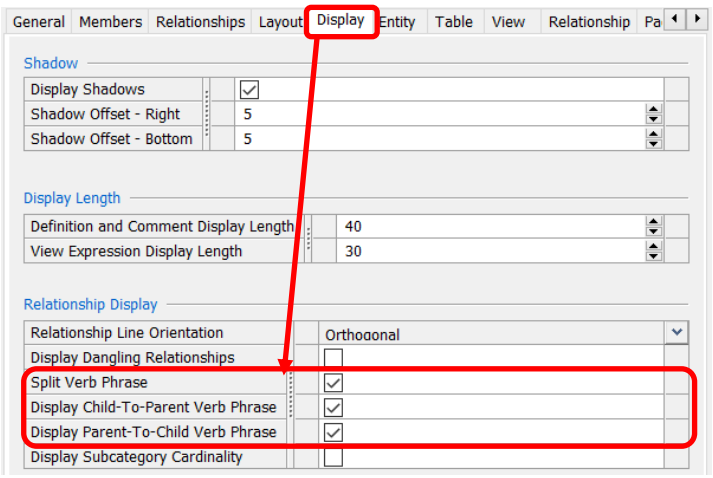

**Figur 14-50: E/R-diagram med rekursivt forhold i BANEANLEGG. Et rollenavn er påført relasjonen.**

Rolleforholdet er her tydeliggjort ved å påføre modellen rollebeskrivelsene «**kan ha**» og «**kan tilhøre**». Dette gjøres ved å dobbelt klikke på relasjonsforholdet og under «**General**» definere «**Parent-to-Child Phrase**» og «**Child-to-Parent Phrase**» som vist i Figur 14-51.




**Figur 14-51: Definisjon av rollebeskrivelser for relasjonsforholdet.**

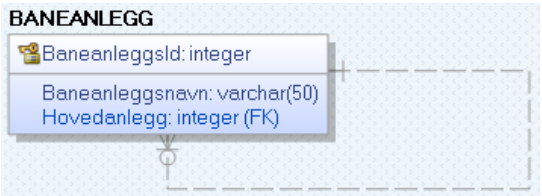
For at rollebeskrivelsene skal vises i E/R-diagrammet, dobbeltklikk på et tomt område på skjemaet, velg Display og kryss av for «**Split Verb Phrase**», «**Display Child-To-Parent Verb Phrase**» og «**Display Parent-To-Child Verb Phrase**» som vist i Figur 14-52.




**Figur 14-52: Konfigurering for å vise roller som er påført E/R-diagrammet.**

En overgang til **fysisk modell** krever noe redigering i modellen, hvilket snart beskrives. Resultatet blir som vist i Figur 14-53, med **Hovedanlegg** som fremmednøkkel. Denne peker til **BaneanleggsId**. Både **Hovedanlegg** og **BaneanleggsId** representerer ID-numre for **baneanlegg**, der det ene refererer til det andre innenfor samme tabell.

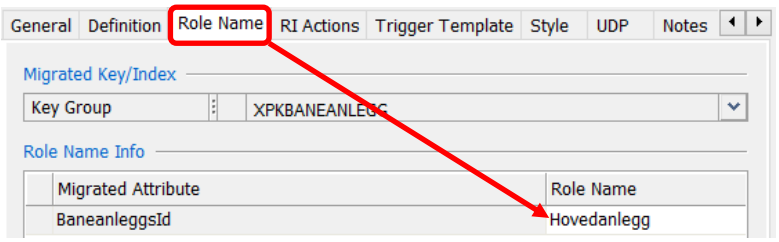
Selv om begge kolonnene for så vidt representerer **BaneanleggsId**, er det et krav at disse må ha ulike navn, da en tabell ikke kan ha flere kolonner med samme navn.




**Figur 14-53: Fysisk modell av det rekursive forholdet som ble modellert i E/R-diagrammet.**

For å få til dette i **erwin**, må man ***etter*** at E/R-diagrammet i Figur 14-48 er tegnet (i den logiske modellen), dobbeltklikke på relasjonsforholdet. Under «**Role Name**», som vist i Figur 14-54, defineres det et

attributt kalt **Hovedanlegg** (valgfritt navn), som skal være et fremmednøkkelattributt mot **Baneanleggs-Id** i den samme tabellen, altså i tabellen **BANEANLEGG** (altså et rekursivt relasjonsforhold).




**Figur 14-54: Hovedanlegg defineres som fremmednøkkel mot BaneanleggsId.**

Ut fra den fysiske modellen vist i Figur 14-53, vil man nå ende opp med én tabell:

- BANEANLEGG (BaneanleggsId, Baneanleggsnavn, Hovedanlegg*)

Fremmednøkkel

I Figur 14-55 er tabellen vist med eksemplifisering av data innlagt. Der ses det at verdiene i kolonnen **Hovedanlegg** er fremmednøkkelreferanser som refererer til **BaneanleggsId** i den samme tabellen. Det som altså er primærnøkkelen i den første raden (**BaneanleggsId** 1) er for eksempel en fremmednøkkel i den neste raden (som da peker til verdien oppført i rad 1).

**Tabellnavn: BANEANLEGG**

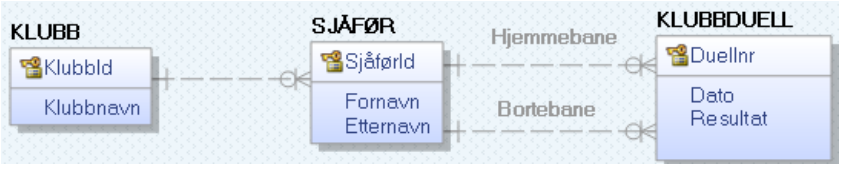
| BaneanleggsId | Baneanleggsnavn | Hovedanlegg |
|---|---|---|
| 1 | Baneanlegg 1 | |
| 2 | Delanlegg 1 | 1 |
| 3 | Delanlegg 2 | 1 |
| 4 | Baneanlegg 2 | |
| 5 | Delanlegg 1 | 4 |
| 6 | Delanlegg 2 | 4 |
| 7 | Delanlegg 3 | 4 |
| 9 | Baneanlegg 3 | |
| 10 | Delanlegg 1 | 9 |
| 11 | Delanlegg 2 | 9 |
| 12 | Delanlegg 3 | 9 |
| 13 | Delanlegg 4 | 9 |

**Figur 14-55: Innlegging av verdier i den «rekursive» modellen.**

## 14.25. Flere en-til-mange-relasjonsforhold mellom to entiteter

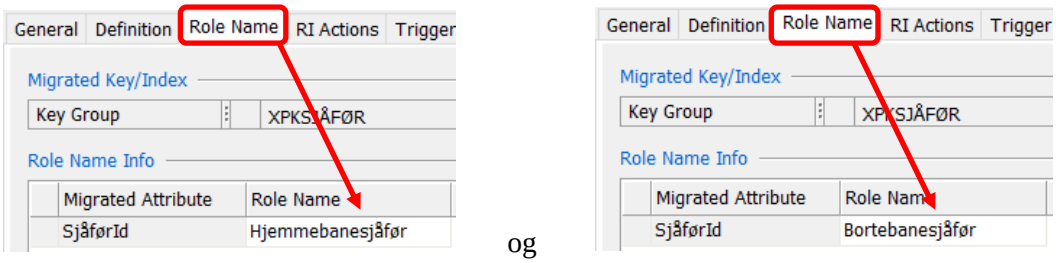
Enkelte ganger vil det være hensiktsmessig å etablere flere 1:M-relasjonsforhold mellom to entiteter.

Som eksempel, skal vi tenke oss at det på stevner arrangeres noen klubbdueller mellom en sjåfør som har hjemmebane og en som kjører på bortebane. Det skal registreres hvilke klubber sjåførene tilhører, hvem av dem som har henholdsvis hjemmebane og bortebane i duellen og om resultatet ble hjemme-seier (H), borteseier (B) eller uavgjort (U). I Figur 14-56 er det laget et E/R-diagram for dette. Som det fremgår, er det definert 2 stk. 1:M-forhold mellom tabellene **SJÅFØR** og **KLUBBDUELLEN**.




**Figur 14-56: Modellering av transporttilbud, med logisk modell til venstre og fysisk modell til høyre.**

Rollene defineres etter fremgangsmåten vist i Figur 14-54, nå med rollenavn som vist i Figur 14-57 for de to forholdene. To fremmednøkkelkolonner, med disse navnene, opprettes da i den fysiske modellen. Begge disse blir fremmednøkkelreferanser mot kolonnen **SjåførId** i tabellen **SJÅFØR**.




**Figur 14-57: Rollenavn påføres de to relasjonsforholdene.**

Den fysiske modellen blir da som vist i Figur 14-58.


**Figur 14-58: Fysisk modell der to fremmednøkler er opprettet som referanser til SjåførId i KLUBB.**

Tabellene man ender opp med er:

- KLUBB (**KlubbId**, Klubbnavn)
- SJÅFØR (**SjåførId**, Fornavn, Etternavn, KlubbId*)
- KLUBBDUELLEN (**Duellnr**, Hjemmebanesjåfør*, Bortebanesjåfør*, Resultat, Dato)

I Figur 14-59 er det lagt inn noen eksempeldata. Dataene i kolonnene **Hjemmebanesjåfør** og **Bortebanesjåfør** i tabellen **KLUBBDUELL** er fremmednøkler til en og samme kolonne i tabellen **SJÅFØR**, dvs. til kolonnen **SjåførId**.

Tabell: **SJÅFØR**

| SjåførId | Fornavn | Etternavn | KlubbId |
|---|---|---|---|
| 1 | Huey | Duck | 1 |
| 2 | Dewey | Duck | 1 |
| 3 | Louie | Duck | 1 |
| 4 | Practical | Pig | 2 |
| 5 | Fiddler | Pig | 2 |
| 6 | Fifer | Pig | 2 |

Tabell: **KLUBBDUELLEN**

| Duellnr | Hjemmebanesjåfør | Bortebanesjåfør | Resultat | Dato |
|---|---|---|---|---|
| 1 | 1 | 4 | H | 3.3.__ |
| 2 | 5 | 2 | H | 17.3.__ |
| 3 | 2 | 6 | B | 29.3.__ |
| 4 | 3 | 5 | H | 12.4.__ |

**Figur 14-59: Noen eksempeldata i de to tabellene SJÅFØR og KLUBBDUELLEN.**

# 15. Normalisering

Normalisering er en metode knyttet til dekomponering av tabeller (splitting av tabeller/relasjoner i mindre enheter), for å sikre en god tabellstruktur. Grunnlaget for teorien bak relasjonsdatabaser og normalisering ble lagt av Codd og er beskrevet i en artikkel i 1970 [1]. Normalformene (1., 2. og 3. normalform) ble utover 70-tallet utvidet med normalformen **BCNF** (Boyce Codd Normal Form), der navnet refererer til Raymond Boyce [3] og E. F. Codd.

I dette kapittelet ses det kort på noen av de viktigste prinsippene knyttet til dette, så man har noen «verktøy» til å sjekke tabellstrukturen med, før det lages en database. Målet er å unngå redundans og sikre dataintegritet.

Det å lage et godt E/R-diagram er gjerne det første bidraget til en god tabellstruktur, og mye av normaliseringsarbeidet kan ivaretas der. Kaula Rajeev, professor ved Missouri State University, gir noen retningslinjer for dette i denne artikkelen [10]. Etter dette bør normalisering utføres. To viktige ting normalisering bidrar til er å unngå:

- redundans (dvs. lagring av overflødig informasjon)
- anomalier (dvs. uregelmessigheter i databasen som konsekvens av oppdateringer i en dårlig tabellstruktur)

## 15.1. Redundans

Redundans kan betraktes som overflødige data lagret i databasen. Et klassisk eksempel er lagring av postnummer og poststed når det lagres persondata, som vist et eksempel på i Tabell 1.

**Tabell 1: Persontabell**

| | | | | |
|---|---|---|---|---|
| 1 | Per | Hansen | 3918 | Porsgrunn |
| 2 | Lise | Olsen | 3918 | Porsgrunn |
| 3 | Frank | Olsen | 3720 | Skien |
| 4 | Hanne | Fjell | 3720 | Skien |

I dette eksempelet ses det at det at for hver nye person som registreres med postnummeret 3918, gjentas Porsgrunn. På tilsvarende vis gjentas Skien for hver forekomst av personer med postnummer 3720. Dette er et typiske eksempler på redundante data. Lagring på denne måten har flere utfordringer:

- Det kan inntreffe anomalier (uregelmessigheter/feil), ved at ikke alle data oppdateres korrekt Dette ses det nærmere på i kapittel 15.2.
- Unødvendig dobbeltlagring av data kan inntreffe når ett datasett kan avledes av et annet. Er det for eksempel nødvendig å lagre pris både *uten* og *med* mva.?
- Unødvendig dobbeltlagring krever ekstra lagringsplass.

I enkelte tilfeller kan det være ønskelig eller påkrevd med redundante data, eksempelvis for å sikre raske nok oppslag i databasen. Håndteringen må da gjøres på en måte som sikrer at feil ikke kan inntreffe, det vil si kontrollert redundans.

I Tabell 1 kan redundansen unngås ved å skille ut postnummer og poststed i en egen tabell. Dette kalles en normalisering, noe vi senere skal se nærmere på reglene for.

## 15.2. Anomalier (uregelmessigheter etter oppdateringer i databasen)

Anomalier er uregelmessigheter, avvik eller feil som kan inntreffe som en konsekvens av en tabellstruktur som ikke er normalisert.

Det er vanlig å inndele anomalier i tre typer:

1. Innsettingsanomalier (Insertion anomalies)
2. Sletteanomalier (Delete anomalies)

3. Oppdateringsanomalier (Update anomalies)

I det følgende forklares disse anomalitypene med henvisning til dataene i Tabell 2.

**Tabell 2: Tabell for registrering av personer som melder seg på ulike kurs**

| PersonId | Fornavn | Etternavn | Postnr | Poststed | KursId | Kursnavn | Kursdato |
|---|---|---|---|---|---|---|---|
| 1 | Per | Hansen | 3918 | Porsgrunn | 1 | Databaser | 3. mai |
| 1 | Per | Hansen | 3918 | Porsgrunn | 7 | C# | 30. mai |
| 3 | Frank | Olsen | 3720 | Skien | 7 | C# | 30. mai |
| 4 | Hanne | Fjell | 3720 | Skien | 1 | Databaser | 3. mai |
| 3 | Frank | Olsen | 3720 | Skien | 1 | Databaser | 3. mai |
| 1 | Per | Hansen | 3918 | Porsgrunn | 12 | Java | 5. juli |
| 5 | Lise | Karlsen | 3740 | Skien | 19 | Operativsystemer | 2. februar |

### 15.2.1. Innsettingsanomalier (Insertion anomalies)

Dersom innlegging av én type data i en tabell, gjør at man samtidig er nødt til å registrere en annen type ikke-samhørende data i den samme tabellen, foreligger det en innsettingsanomali.

Av Tabell 2 ses det at radene ikke kan identifiseres unikt kun med attributtene KursId, Kursnavn og Kursdato. Dette betyr at dersom det skal registreres et nytt kurs, må det samtidig registreres en person. I annet fall brytes det med entitetsintegritetsreglene, da det vil bli stående nullmerker i attributter som kreves for å identifisere en rad. Dette er ett eksempel på en innsettingsanomali.

Skal det påmeldes en ny person på et kurs, må samtidig all informasjon om kurset (KursId, Kursnavn og Kursdato) registreres i tillegg til persondataene. Dette må gjøres for hver eneste person som skal påmeldes et kurs. Dette gir mulighet for feilregistrering av data i noen av radene, sånn at det kan bli liggende ulik informasjon registrert om ett og samme kurs. Dette er et eksempel på en innsettingsanomali.

### 15.2.2. Sletteanomalier (Delete anomalies)

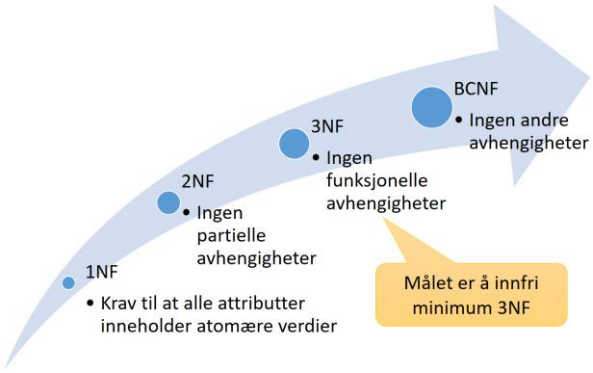
Ønskes det å slette kurset «Operativsystemer», f.eks. fordi det kun er én påmeldt, kan dette ikke gjøres uten samtidig å slette all informasjon om personen påmeldt dette kurset. Dersom denne personen kun er påmeldt dette kurset, har man mistet all personinformasjonen. Dette kalles en sletteanomali.

### 15.2.3. Oppdateringsanomalier (Update anomalies)

La oss si at kursnavnet «Databaser» ønskes endret til «Relasjonsdatabaser». Da må alle forekomster der noen er påmeldt dette kurset endres. Dette gir mulighet for feil, dersom ikke alle radene oppdateres, og er et eksempel på en oppdateringsanomali.

## 15.3. Normalisering av tabeller - Normalformer

For å løse problemet med redundans og anomalier normaliseres tabellene. Det finnes ulike normalformer, første, andre, tredje, Boyce Codd, fjerde og femte normalform, gjerne forkortet til 1NF, 2NF, 3NF, BCNF, 4NF og 5NF. Vi ønsker i våre eksempler å normalisere tabellene så de innfrir kravene til 3NF. Gjør de det, innfrir de også kravene til 1NF og 2NF. Normaliseringstrinnene er illustrert i Figur 15-1, der man først sjekker mot 1NF, så 2NF osv.




**Figur 15-1: Normaliseringstrinnene.**

## 15.4. 1NF (Første normalform)

For at en tabell skal innfri 1NF-kravet, kreves det at alle attributter inneholder atomære verdier, hvilket vil si at dataene må representere entydige verdier. I Tabell 3 er det vist et eksempel på en tabell der dette kravet brytes. For kurset med KursId 1 (med kursnavn «Databaser»), er det registrert tre ulike kursdatoer, hvilket ikke er en atomær verdi.

**Tabell 3: Tabell som bryter med kravet til kun atomære verdier, altså brudd på 1NF**

| KursId | Kursnavn | Kursdato |
|---|---|---|
| 1 | Databaser | 3. mai<br>4. mai<br>6. mai |

For å løse dette problemet, kan det opprettes en ny tabell kalt KURSAVVIKLING. I denne blir kombinasjonen av KursId og Kursdato en kombinert primærnøkkel og KursId en fremmednøkkel mot den opprinnelige tabellen. Da kan det for hver dato registreres flere kurs og hvert kurs kan oppføres på flere datoer. Den nye tabellen er vist i Tabell 4, med en del eksempeldatoer innlagt. Tabellen KURSAVVIKLING blir da en koblingstabell, som representerer en kobling mellom datoer og kurs.

**Tabell 4: Tabell som innfrir 1NF-kravet om atomære verdier.**

| KURSAVVIKLING | |
|---|---|
| **Kursdato** | **KursId** |
| 3. mai | 1 |
| 4. mai | 1 |
| 4. mai | 10 |
| 5. mai | 17 |
| 6. mai | 1 |

Før det gås videre for å finne ut om det foreligger noen brudd på 2NF, defineres begrepene Supernøkkel, Kandidatnøkkel og funksjonell avhengighet

## 15.5. Supernøkkel (Super Key)

En supernøkkel er en kombinasjon av attributter i en tabell, som kan identifisere samtlige attributter i tabellen. Dette betyr at kombinasjonen av ***alle*** attributtene i tabellen, alltid vil være en supernøkkel. I tillegg vil alle andre kombinasjoner av attributter, med tilsvarende egenskaper, også være supernøkler. En tabell vil derfor normalt ha mange supernøkler.

## 15.6. Kandidatnøkkel (Candidate Key)

En kandidatnøkkel er en minimal supernøkkel, hvilket gjør den til en **kandidat** til å velges til primærnøkkel. At supernøkkelen er minimal, betyr at det ikke kan fjernes attributter fra supernøkkelen ***uten*** at den ***mister*** sin egenskap som supernøkkel. Ofte har en tabell bare ***én*** kandidatnøkkel, men den ***kan*** ha flere.

Dersom en tabell for studenter har både attributtene Studentnummer og Personnummer, vil begge være supernøkler. Begge vil unikt kunne identifisere alle de øvrige attributtene. Samtidig kan det ikke fjernes

noe fra dem uten at de mister sin egenskap som supernøkler. I dette tilfellet består begge nøklene av kun ***ett*** attributt, og da er det innlysende at ikke noe kan fjernes fra dem. Begge er dermed kandidatnøkler, det vil si mulige primærnøkler, i en slik tabell.

En tabell må alltid ha nøyaktig ***én*** primærnøkkel, aldri flere, men primærnøkkelen kan bestå av ***flere*** attributter, hvilket i så fall gjør den til en kombinasjonsnøkkel (Combined Primary Key).

Primærnøkkelen velges blant kandidatnøklene. Er det bare én kandidatnøkkel, er valget enkelt, for da ***må*** denne brukes som primærnøkkel. Er det flere kandidatnøkler, må en av dem velges som primærnøkkel. I Studenttabelleksempelet vil Studentnummer trolig være et naturlig valg som primærnøkkel, fordi det er knyttet en god del restriksjoner fra Datatilsynet når det gjelder bruk av en persons personnummer.

## 15.7. Funksjonelle avhengigheter

For å finne kandidatnøkler må man først definere funksjonelle avhengigheter. For å kunne gjøre dette, kan man ikke bare se på ***noen*** av radene med data, men man må ***vite*** noe om hvordan attributtene henger sammen. Dette henger sammen med spesifikasjonen av systemet, etter oppdragsgivers behov.

Dersom det er sånn at A, her definert som ett attributt eller en kombinasjon av flere attributter, alltid gir B, definert som et annet attributt eller kombinasjon av attributter, sier vi at det en funksjonell avhengighet fra A til B, hvilket skrives A → B (A gir B). Den funksjonelle avhengigheten må i så fall gjelde for enhver rad i tabellen. I en slik avhengighet er A en **determinant**, som gir B.

I Tabell 3 kan man tenke seg flere funksjonelle avhengigheter. For eksempel vil det være naturlig å anta at én PersonId alltid vil gi det samme fornavnet og etternavnet: PersonId → Fornavn, Etternavn

Videre vil det være sånn at ett spesifikt postnummer alltid vil gi samme poststed. Dette er altså også en funksjonell avhengighet: Postnummer → Poststed.

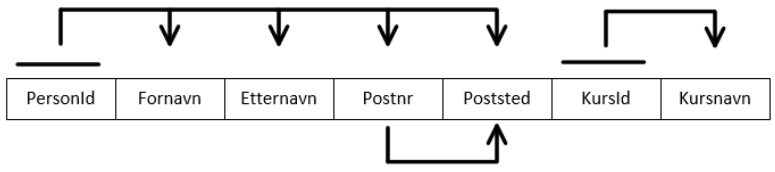
Én KursId vil høyst sannsynlig alltid gi samme kursnavn: KursId → Kursnavn. Dersom det også er sånn at ett konkret kurs alltid settes opp på én konkret dato, vil også følgende avhengighet gjelde: KursId -> Kursnavn, KursDato, men det er ikke sikkert at det foreligger en slik avhengighet. Det må derfor utføres en analyse m.h.p. før det kan konkluderes med hvilke avhengigheter som foreligger.

For å gjøre normaliseringen mer oversiktlig, erstattes ofte attributtnavnene med A, B, C osv. Dette fordi at attributtnavnene ikke er av betydning for normaliseringen etter at de funksjonelle avhengighetene og primærnøkkelen er påført. Deretter følger normaliseringen et fast sett med regler.

## 15.8. 2NF (Annen normalform)

For at en tabell skal innfri kravet til 2NF, må det ikke foreligge noen partielle avhengigheter. Partielle avhengigheter vil si at det finnes funksjonelle avhengigheter med attributter ***partielt*** avhengig av primærnøkkelen, eller sagt på en annen måte, avhengig av ***en del*** av primærnøkkelen.

For å avgjøre om en tabell er på 2NF (andre normalform), må det avklares hva som skal være tabellens primærnøkkel, fordi denne inngår i definisjonskravet for 2NF. Etter en analyse er de funksjonelle avhengighetene, illustreres avhengighetene med piler, som vist i Figur 15-2.




**Figur 15-2: Tabellen påført funksjonelle avhengigheter og primærnøkkel.**

**Kandidatnøkkel/kandidatnøkler:**

Det første som må gjøres er å finne tabellens kandidatnøkkel/-nøkler. ***PersonId og KursId*** i kombinasjon er en supernøkkel. Det kan ikke fjernes noe fra denne nøkkelen, uten at den mister sin egenskap som supernøkkel. Dermed er dette en kandidatnøkkel.

**Primærnøkkel:**

Det er ingen andre kandidatnøkler enn kombinasjonen ***PersonId*** og ***KursId*** i denne tabellen, og dermed må denne kombinasjonen velges som primærnøkkel. Primærnøkkelen er markert i Figur 15-2 ved å påføre en strek over attributtene som inngår i denne.

**Primærnøkkel:** PersonId, KursId → Fornavn, Etternavn, Postnr, Poststed, Kursnavn.
(PersonId, KursId gir underforstått også seg selv, men dette tas normalt ikke med i oppstillingen).

Andre funksjonelle avhengigheter:

PersonId → Fornavn, Etternavn, Postnr, Poststed
Postnr → Poststed
KursId → Kursnavn

**Er tabellen på 2NF?**

Det sjekkes så om noen av de funksjonelle avhengighetene utenom primærnøkkelen bryter med kravet til at det ikke skal foreligge noen partielle avhengigheter.

Det ses da at Fornavn, Etternavn, Postnr, Poststed er funksjonelt avhengig av PersonId, som er en del av primærnøkkelen. Dette er dermed et brudd på kravet til 2 NF.

I tillegg er Kursnavn funksjonelt avhengig av KursId, som også er en del av primærnøkkelen.

Den siste avhengigheten, Postnr → Poststed bryter ***ikke*** med 2NF, da Postnr ikke er en del av primærnøkkelen.

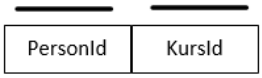
Siden det foreligger to avhengigheter som bryter med 2NF, betyr dette at tabellen er på 1 NF.

**Hvordan løse brudd på 2NF:**

For å løse problemet med brudd på 2NF, normaliseres tabellen. Dette gjøres ved å skille ut attributtene som er funksjonelt avhengige av en del av primærnøkkelen i egen/egne tabeller. De nye tabellene er vist i Figur 15-3.


**Figur 15-3: De funksjonelle avhengighetene som bryter med 2NF er skilt ut i to nye tabeller.**

PersonId og KursId må samtidig bevares i den opprinnelige tabellen, så man ikke mister den logiske sammenhengen mellom de opprinnelige attributtene. For å bevare denne sammenhengen blir disse attributtene fremmednøkler som refererer/peker til de nye tabellene. Den resterende delen av den opprinnelige tabellen blir etter dette en tabell kun med attributtene PersonId, KursId, som vist i Figur 15-4.




**Figur 15-4: Resterende del av den opprinnelige tabellen.**

Den opprinnelige tabellen er nå normalisert til tre tabeller. Alle innfrir kravene til 2NF, da det ikke lenger er noen attributter som er partielt avhengig av primærnøkkelen.

Tabellene kan skrives med følgende notasjon, der primærnøkler angis med understrekning og fremmednøkler er angitt med en stjerne bak (*). Dersom primærnøkler og/eller fremmednøkler består av flere attributter, settes en parentes rundt attributtene som inngår i nøkkelen. Gjelder dette en fremmednøkkel, settes stjernen i så fall bak parentesen.

De normaliserte tabellene kan da stilles opp sånn, der passende tabellnavn er valgt:

PERSON (PersonId, Fornavn, Etternavn, Postnr, Poststed)

KURS (KursId, Kursnavn)

PÅMELDING (PersonId*, KursId*)

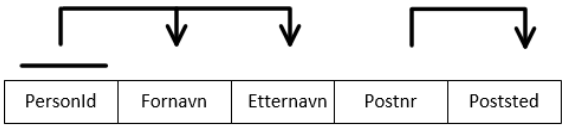
## 15.9. 3NF (Tredje normalform)

I 3NF er kravet at det ikke foreligger noen transitive avhengigheter. Dette betyr at ingen attributter kan være indirekte avhengig av primærnøkkelen. I praksis vil dette si at det ikke må foreligge noen funksjonelle avhengigheter mellom ikke-nøkkelattributter.

I den nye tabellen PERSON, som ble opprettet gjennom normaliseringen til 2NF, er det en funksjonell avhengighet mellom Postnr og Poststed (Postnr → Poststed). Dette er vist med pil i Figur 15-5. Ingen av disse to attributtene inngår som en del av primærnøkkelen. Dette er dermed en avhengighet mellom ikke-nøkkelattributter, hvilket innebærer et brudd på 3NF.

Den funksjonelle avhengigheten her, kalles transitiv avhengighet. Dette fordi Postnr er avhengig av PersonId og Poststed er avhengig av Postnr. Dette kan stilles opp sånn: PersonId → PostNr og PostNr → Poststed. Implisitt gjelder da at PersonId → Poststed, hvilket er den transitive avhengigheten.

Tabellen er dermed på 2NF, fordi den bryter med 3NF.




**Figur 15-5: Avhengigheten Postnr → Poststed bryter med kravet til 3NF. Tabellen er dermed på 2NF.**

## 15.10. BCNF (Boyce Codd normalform)

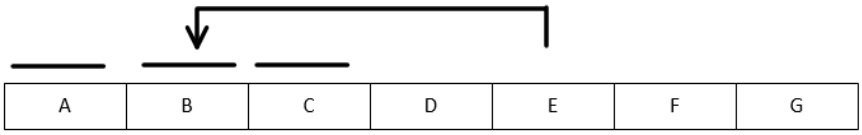
Dersom en tabell er normalisert til 3NF, vil den i de fleste tilfeller også innfri kravene til BCNF. Brudd på BCNF er sjeldne og litt vanskelige å håndtere på en god måte, så derfor stopper man ofte normaliseringen når tabellen er på 3NF.

Kravet til BCNF er at enhver determinant (attributt/attributter i en funksjonell avhengighet som bestemmer det/de andre attributtet/attributtene) skal være en kandidatnøkkel.

En konsekvens av et brudd på BCNF er at det vil være en determinant som peker tilbake til en del av primærnøkkelen (da alle transitive avhengigheter jo ble fjernet med 3NF).

I tabellene vi har operert med i eksemplene i dette kapittelet har det ikke vært noen brudd på BCNF.

Et BCNF-brudd kan illustreres som vist i Figur 15-6. Her vises det en funksjonell avhengighet (E → B), med en determinant (E) som ikke er en kandidatnøkkel (da ikke alle attributtene i tabellen er funksjonelt avhengig av denne). Dette bryter med BCNF-kravet, og tabellen er dermed på 3NF.




**Figur 15-6: En illustrasjon av et brudd på BCNF. Tabellen er på 3NF.**

## 15.11. Ulemper med normalisering:

Konsekvensen av normalisering er at en tabell splittes opp i mindre tabeller. Dette betyr at det ved spørringer hyppigere vil forekomme «joining» av tabeller, hvilket kan gjøre spørringene tregere, samt mer kompliserte å lage.

Å lage VIEWs kan kompensere noe for problemet med hyppig «joining». Likevel vil det av og til være hensiktsmessig ikke å normalisere fullt ut av slike hensyn. Som en konsekvens vil det da lagres redundante data i databasen, som må håndteres kontrollert.

Tilsvarende gjelder når data lagret i én kolonne, kan avledes/beregnes ut fra data i en annen/andre kolonne/kolonner. Dersom det kreves omfattende beregninger med data som må hentes fra en annen/ andre kolonner, vil dette kunne bli svært tidkrevende. I noen tilfeller vil det da velges bevisst å dobbeltlagre data i slike sammenhenger.

Et eksempel kan være bankkontoer der det stadig skal hentes ut saldo. Det må der hver gang det skal hentes ut en saldo, utføres beregninger basert på en rekke posteringer, dersom ikke saldo fortløpende

lagres som tilleggsdata i en egen kolonne. Da kan et valg om å innføre en redundant saldo-kolonne være et alternativ.

Ved kontrollert redundans blir det naturlig nok svært viktig at det programmeringsmessig sikres at det ikke kan inntreffe feil/uregelmessigheter som konsekvens av den redundante datastrukturen.

# 16. Kobling mellom C# og database

I dette notatet ses det på eksempler der det lages brukergrensesnitt i C# med kobling mot en SQL Server-database. Via dette brukergrensesnittet skal det utføres ulike operasjoner mot databasen. Data brukeren oppgir i tekstbokser skal lagres, og utvalgte data skal hentes ut fra databasen og vises på ulike måter i brukergrensesnittet.

## 16.1. Eksempeldatabase

Det skal lages en tabell for å registrere bilmerker og biler, som vist med E/R-diagrammet i Figur 16-1.


**Figur 16-1: E/R-diagram i erwin for databasen som skal lages.**

Den logiske modellen (E/R-diagrammet) i Figur 16-1 blir omgjort til den fysiske modellen vist i Figur 16-2.


**Figur 16-2: E/R-diagram omgjort til fysisk modell i erwin.**

## 16.2. SCRIPT for å generere tabellene CAR og CARMAKER

Opprett en i SQL Server en database med navn «VehicleRegistration», enten via «Object Explorer» eller ved å utføre følgende SQL-kode:

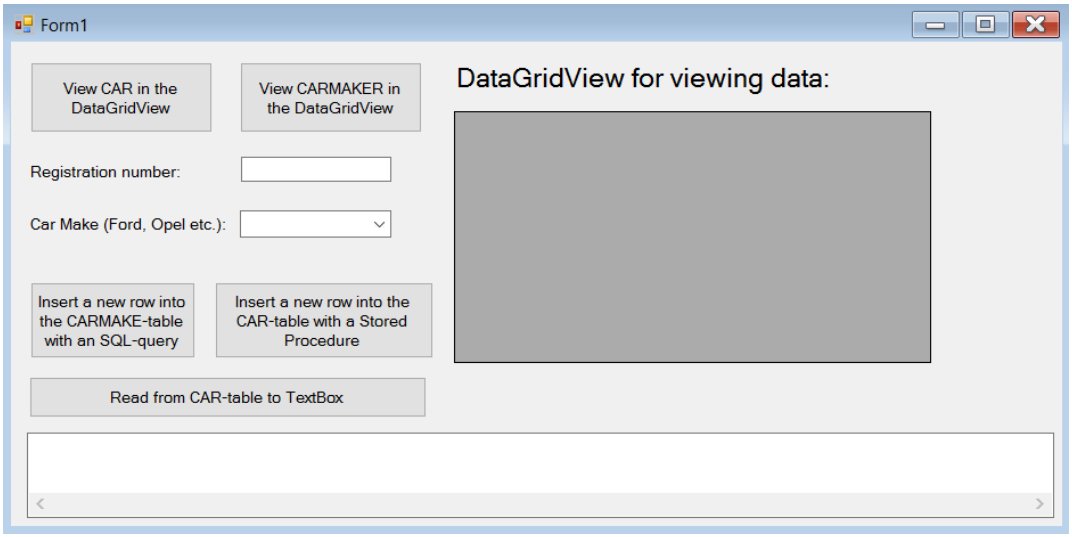
```
Create Database VehicleRegistration
Go
```

Et script for deretter å generere tabellene i SQL Server kan genereres fra erwin eller lages manuelt som vist med følgende kode:

```
use VehicleRegistration
go

CREATE TABLE CARMAKER
(
	CarMake	char(40)  NOT NULL ,
	CONSTRAINT PK_CARMAKER PRIMARY KEY (CarMake)
)

CREATE TABLE CAR
(
	RegNumber char(7)  NOT NULL ,
	CarMake  char(40)  NOT NULL ,
	CONSTRAINT PK_CAR PRIMARY KEY  (RegNumber),
	CONSTRAINT FK_CAR_CARMAKER FOREIGN KEY (CarMake) REFERENCES CARMAKER(CarMake)
)
```

## 16.3. Brukergrensesnitt (GUI) i C#

Brukergrensesnittet i C# skal være som vist i Figur 16-3.




**Figur 16-3: Brukergrensesnitt**

Lage en kobling (connection) mellom C# og databasen. Connection-stringen skal i dette eksempelet defineres i «App.config».

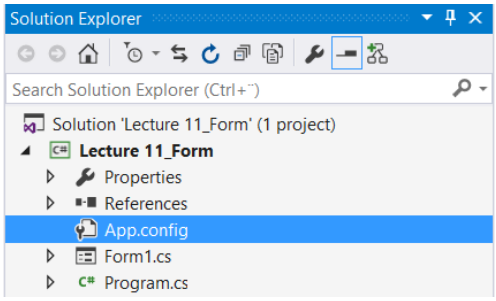
Legg inn data i databasen via brukergrensesnittet, ved å sette sammen tekst til en SQL-spørring som sendes til databasen for utførelse. NB! Dette er ikke en sikker måte å kommunisere med databasen på, da hackere i visse tilfeller kan klare å manipulere spørringen (SQL Injection).

## 16.4. Kobling mot databasen

For å få til en kobling mellom C# og databasen må det lages en «connection string». Et alternativ er å lage denne direkte i koden, men en mer sikker og generell måte å gjøre det på, er å definere denne i App.config. Da blir det enkelt ved behov å endre «connectiong string» uten å måtte gjøre endringer i koden. Det kan defineres flere «connection strings» mot ulike databaser, men i dette eksempelet behøves bare en.

### 16.4.1. App.config

App.config finner man i Solution Explorer, som vist I Figur 16-4.




**Figur 16-4: "Connection strings" kan defineres i App.config.**

Dersom ikke App.config allerede ligger i Solution Explorer, kan den legges til ved å høyreklikke over prosjektet (i dette tilfellet Lecture11_Form) i Solution Explorer (jf. Figur 16-4) og velge følgende:

*Add → New Item → General (under Visual C# Items) → Application Configuration File.*

Klikk deretter på **Add** og en **App.config**-fil bli lagt til i Solution Explorer.

Høyreklikk over **App.config** og velg **Open**. Følgende kode skal legges til i **App.config**.

```
<connectionStrings>
    <add name="conVehicle" connectionString="Data Source=kontor-pc;
    Initial Catalog =   VehicleRegistration; Integrated Security = True"/>
</connectionStrings>
```

Ovennevnte kode er for pålogging med «Windows Authentication», og det behøves dermed ikke angivelse av brukernavn og passord.

I dette tilfellet er «kontor-pc» navnet på min SQL Server. For å gjøre «connection string»-en mer generell, dersom den skal åpne default SQL Server på samme server som C#-programmet ligger, kan server-navnet byttes ut med «**localhost**» eller «**.**» (punktum). Da kan man teste C#-fila på en annen PC som har SQL Server installert, ved å kopiere direkte over til den andre PC-en. Koblingen skal da fungere med en gang, uten at man trenger å endre server-navnet i «connection string»-en.

```
<connectionStrings>
    <add name="conVehicle" connectionString="Data Source=localhost;
    Initial Catalog =   VehicleRegistration; Integrated Security = True"/>
</connectionStrings>
```

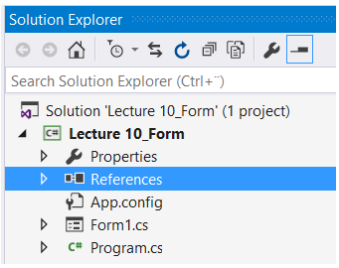
### 16.4.2. Nødvendige using-statements som må legges til

For databaseoperasjonene som skal utføres behøves følgende using-statements lagt til:

```
using System.Data.SqlClient;
using System.Configuration;
```

### 16.4.3. Referanse til System.Configuration må legges til

For å kunne bruke metoder fra System.Configuration, må denne legges til som en referanse. Gjøres ikke dette, vil man se at metoder som skrives *ikke* blir blåfarget, samt at de markeres med en rød under-strekning og feilmelding. Høyreklikk over «References» i Figur 16-5.




**Figur 16-5: System.Configuration må legges til som referanse under "References"**

Velg så «Add Reference».

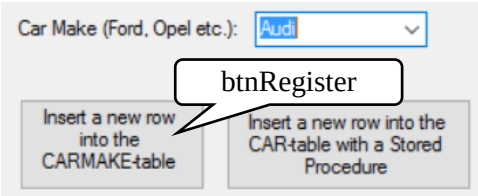
Avmerk sjekkboksen for «System.Configuration» og klikk deretter OK. Deretter kan følgende string legges til som instansvariabel for å koble til databasen definert i **App.config**:

```
string conVehicle =
ConfigurationManager.ConnectionStrings["conVehicle"].ConnectionString;
```

Senere i koden kan man da opprette et connection-objekt (her gitt navnet «con») på følgende måte:

```
SqlConnection con = new SqlConnection(conVehicle);
```

## 16.5. Innlegging av bilmerke i tabellen CARMAKE med en sql-string




**Figur 16-6: Brukergrensesnitt med knappen btnRegister for å legge inn nye bilmerker i CARMAKE**

I det første eksempelet skal det legges inn bilmerker som skrives inn i comboboxen som vist i Figur 16-6: Brukergrensesnitt med knappen btnRegister for å legge inn nye bilmerker i CARMAKE skal gjøres ved sende over en SQL-string til serveren. Dette er ikke en sikkerhetsmessig god måte å kommunisere med databasen på, da (som tidligere nevnt) «hackere» i noen tilfeller vil kunne klare å manipulere sql-spørringen (SQLInjection) før den utføres i databasen (og da f.eks. slette noe isteden).

Koden som skal legges inn i knappen **btnRegister**:

```
private void btnRegister_Click(object sender, EventArgs e)
{
    string carMake, sqlQuery;
    try
    {
        //Oppretter en connection mot databasen med string definert i App.config:
        SqlConnection con = new SqlConnection(conVehicle);
        carMake = cboCarMake.Text; //Verdien som skal inn i databasen
        //hentes fra combobox og lagres i carMake-variabelen

        /* Lagrer spørringen legger en ny "CarMake"-verdi i CARMAKER-tabellen */
        sqlQuery = String.Concat(@"INSERT INTO CARMAKER (CarMake)
           VALUES ('", carMake, "')"); //Setter variabelen carMake inn i sql-spørringen
        con.Open();
        SqlCommand command = new SqlCommand(sqlQuery, con);
        command.ExecuteNonQuery();
        con.Close();
    }
    catch (Exception error)
    {
        MessageBox.Show(error.Message);
    }
}
```

Legg inn noen bilmerker og test med en SELECT-setning i SQL-server mot CARMAKE-tabellen for å se at verdiene har blitt lagt inn. (SELECT * FROM CARMAKE).

## 16.6. Spørring som viser innholdet i CARMAKER i en DataGridView

Det skal lages kode for knappene og btnViewCarInDataGridView og
btnViewCarMakerInDataGridView
vist i Figur 16-7.


**Figur 16-7: Knappen som skal starte innhenting av data fra CAR og CARMAKER og vise disse i en DataGridView.**

Det lages først en generell metode som kan motta en sql-string via en parameter, for så å vise resultatet denne returnerer fra databasen i en DataGridView:

```
void ViewSqlResultInDataGridView (string sqlQuery)
{
    try
    {
        SqlConnection con = new SqlConnection(conVehicle);
        SqlDataAdapter sda;
        DataTable dt;
        con.Open();
        sda = new SqlDataAdapter(sqlQuery, con);
        dt = new DataTable();
        sda.Fill(dt);
        dgvCars.DataSource = dt;
        con.Close();
    }
```

```
    catch (Exception error)
    {
        MessageBox.Show(error.Message);
    }
}
```

Lager så metoden for knappen btnViewCarMakerInDataGridView:

```
private void btnViewCarMakerInDataGridView_Click(object sender, EventArgs e)
{
    string sqlQuery = @"SELECT * FROM CARMAKER ORDER BY CarMake ASC";
    ViewSqlResultInDataGridView (sqlQuery);
}
```

Lager deretter metoden for knappen btnViewCarInDataGridView etter samme prinsipp:

```
private void btnViewCarInDataGridView_Click(object sender, EventArgs e)
{
    string sqlQuery = @"SELECT * FROM CAR ORDER BY RegNumber ASC";
    ViewSqlResultInDataGridView (sqlQuery);
}
```

## 16.7. En «Stored Procedure» for innlegging av data i CAR-tabellen

Følgende sql-spørring skal lages på SQL-serveren:

```
--Lager en "brukerdefinert" Stored Procedure (usp: User Defined Procedure)
CREATE PROCEDURE uspInsertCAR
@regNumber char(7), @carMake char(35) -- Deklarer to paramatere
AS
INSERT INTO CAR (regNumber, carMake) -- Angir kolonner
VALUES (@regNumber, @carMake) -- Setter inn parameterverdiene tabellen
go
```

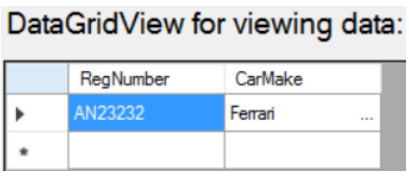
Den lagrede prosedyren skal testes ved å legge inn en bil med et angitt registreringsnummer og bilmerke. Siden CarMake i CAR-tabellen er en fremmednøkkel til CarMake i CARMAKER-tabellen, må bilmerket som skal registreres først legges inn i CARMAKER-tabellen. Dette kan gjøres med denne SQL-setningen:

```
INSERT INTO CARMAKER (CarMake)
VALUES('Ferrari')
```

følgende sql-spørring kan så utføres for å teste den lagrede prosedyren på SQL-serveren:

```
--Test av Stored Procedure kalt uspInsertCAR
exec uspInsertCAR 'AN23232', 'Ferrari'
```

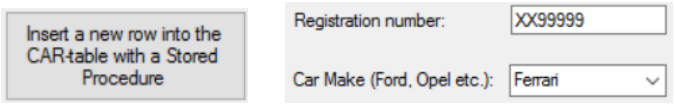
Klikk på knappen btnViewCarInDataGridView i C#-brukergrensesnittet, for å se at denne bilen nå listes opp i DataGridView-en vist i Figur 16-8.




**Figur 16-8: Visning av innlagt bil i DataGridView-en i C#-skjermaet.**

## 16.8. Kalle opp en «Stored Procedure» fra C#-skjemaet

En Stored Procedure skal kalles opp for å legge inn en ny bil, som vist i Figur 16-9.




**Figur 16-9: Knapp for å kalle opp Stored Procedure og legge inn ny bil. Eksempeldata er vist til høyre.**

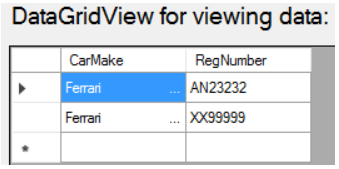
Husk at for å kunne legge inn en ny bil, må bilmerket først være registrert i CARMAKER-tabellen, siden CarMake i tabellen CAR er fremmednøkkel til CarMake i tabellen CARMAKER.

Sørg derfor i dette eksempelet først å legge inn bilmerket «Ferrari» i CARMAKER før det trykkes på knappen btnStoredProcedure (eller du kan prøve å legge inn og se hvilken feilmelding som kommer, såfremt try-catch-blokker er benyttet).

Følgende metode viser hvordan en Stored Procedure med parametere kan kalles opp fra et C#-skjema:

```
private void btnStoredProcedure_Click(object sender, EventArgs e)
{
    try
    {
    string carMake, regNumber;
    carMake = cboCarMake.Text; //Verdi hentes fra tekstboks og lagres i carMake-
variabelen
         regNumber = txtRegNumber.Text; //Verdien som skal inn i databasen hentes fra
                                        //tekstboks og lagres i regNumber-variabelen
         SqlConnection con = new SqlConnection(conVehicle);
         SqlCommand sql = new SqlCommand("uspInsertCAR", con);
         sql.CommandType = CommandType.StoredProcedure;
         con.Open();
         sql.Parameters.Add(new SqlParameter("@carMake", carMake));
         sql.Parameters.Add(new SqlParameter("@regNumber", regNumber));
         sql.ExecuteNonQuery();
         con.Close();
    }
    catch (Exception error)
    {
        MessageBox.Show(error.Message);
    }
}
```

Vis deretter dataene i DataGridView for å se at de er lagt inn, som vist i Figur 16-10.




**Figur 16-10: De innlagte dataene skal kunne ses ved å klikke på knappen som viser CAR-tabellen i en DataGridView.**

## 16.9. Fylle data fra en tabell inn i en ComboBox-liste

Dersom man har en tabell med data som ønskes vist i en ComboBox-nedtrekksmeny, kan det lages kode for dette. Her er det vist en kode som fyller en ComboBox med alle verdiene fra CARMAKER-tabellen:

```
void ImportToComboBox()
{
    SqlConnection con = new SqlConnection(conVehicle);
    string sqlQuery = "SELECT CarMake FROM CARMAKER ORDER BY CarMake ASC";
    SqlCommand sql = new SqlCommand(sqlQuery, con);
    con.Open();
    SqlDataReader dr = sql.ExecuteReader();
```

```
    while (dr.Read() == true)
    {
        sqlQuery = dr[0].ToString();
        cboCarMake.Items.Add(sqlQuery);
    }
    con.Close();
}
```

Koden leser dataene linje for linje og legger dem inn i List-strukturen til en ComboBox (kalt **cboCarMake**) ved å benytte List-strukturens Add-metode.

Metoden kan så kalles opp i constructoren (etter InitializeComponent), sånn at combobox-lista fylles opp med en gang programmet starter. Her er constructor-koden med denne modifikasjonen:

```
public Form1()
{
    InitializeComponent(); //Systeminitieringer (buttons, textbox-er etc. opprettes)
    ImportToComboBox(); //Kaller metoden som viser verdier i comboboxen
    cboCarMake.SelectedIndex = 0; //For å la det første elementet vises I tekstboksen
}
```

Dersom det blir et blått felt i combobox-menyen, så er følgende én måte å omgå dette på:

```
private void cboCarMake_SelectedIndexChanged(object sender, EventArgs e)
{
    label2.Focus();//Legger inn for å unngå blått felt i combobox-menyen
}
```

## 16.10. Lese data fra tabell til List-variabler og vise dem i tekstboks

I dette eksempelet opprettes to stk. List-variabler. Samtlige data hentes så fra de to kolonnene i tabellen CAR og legges i hver sin liste. Deretter skrives verdiparene til en tekstboks. Hadde dette isteden vært tallverdier, kunne disse verdiene deretter vært plottet fra List-variablene til et C#-Chart.

Koden under bør også settes inn i try-catch-blokker, for å fange opp eventuelle feil, men dette er ikke prioritert i dette eksempelet (men legg det gjerne til selv).

```
private void btnReadFromCarTableToTextBox_Click(object sender, EventArgs e)
{
    SqlConnection con = new SqlConnection(conVehicle);
    string sqlQuery = "SELECT * FROM CAR ORDER BY RegNumber ASC";
    SqlCommand sql = new SqlCommand(sqlQuery, con);
    con.Open();
    SqlDataReader dr = sql.ExecuteReader();
    while (dr.Read() == true)
    {
        sqlQuery = dr[0].ToString();
        regNumberValues.Add(sqlQuery);
        sqlQuery = dr[1].ToString();
        carMakeValues.Add(sqlQuery);
    }
    con.Close();

    txtShow.Clear(); //Fjerner eventuelt tekstboksinnhold før start
    foreach (string x in regNumberValues) //Skriver ut alle x1-verdiene
    {
        txtShow.AppendText(x + "\t\t\t");
    }
    txtShow.AppendText("\r\n"); //Linjeskift
```

```
    foreach (string y in carMakeValues)//Skriver ut alle y1-verdiene
    {
        txtShow.AppendText(y + "\t");
    }

    regNumberValues.Clear();
    carMakeValues.Clear();
}
```

To lister må defineres som instansvariabler (direkte etter class …-linjen), som vist her:

```
public partial class Form1 : Form
{
    List<string> regNumberValues = new List<string>();
    List<string> carMakeValues = new List<string>();
```

Foreach-løkker er i koden brukt for å iterere gjennom alle elementene i hver av de to listene. Resultatet etter innlegging av fire testverdier er vist i Figur 16-11.


**Figur 16-11: Verdier fra CAR-tabellen hentes fra databasen, legges i List-variabler og vises i en tekstboks.**

# 17. C# Charts

Chart-verktøyet i CSharp kan benyttes til å lage grafiske fremstillinger av måledata. Dataene kan for eksempel hentes fra en fil eller fra en database. Det er også mulig å plotte data fra for eksempel en målesensor. Da behøves det et «interface» og en «driver», så dataen kan digitaliseres og leses inn i C#-koden, men dette ses det ikke på her.

## 17.1. Opprette SQL Server-tabell med data som skal plottes i et C#-chart

Det første vi skal se på er hvordan data kan leses fra en tabell i SQL Server og plottes i et C# Chart.

Som et utgangspunkt er det opprettet en database kalt **ChartDatabase**. I denne lages så en tabell som gis navnet **CHARTDATA**, definert med to kolonner kalt **XColumn** og **YColumn**, begge av datatypen **float**, som vist i Figur 17-1. (Høyreklikk i Object Explorer over «**Tables**» i databasen «**ChartDatabase**» og velg «**New Table**»).

| | Column Name | Data Type | Allow Nulls |
|---|---|---|---|
| 🔑 | XColumn | float | ☐ |
| | Y1Column | float | ☑ |
| | Y2Column | float | ☑ |

**Figur 17-1: Definisjon av en tabell kalt CHARTDATA.**

Høyreklikk så over tabellen **CHARTDATA** og velg «**Edit Top 200 Rows**». Legg inn verdiene vist i Tabell 5 i de to kolonnene **XColumn** og **YColumn**.

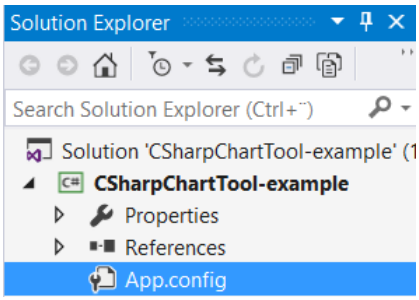
**Tabell 5: Verdiene som skal legges inn i tabellen CHARTDATA.**

| XColumn | YColumn |
|---|---|
| 0 | 4,5 |
| 1 | 6 |
| 2 | 7,5 |
| 3 | 12 |
| 4 | 10,5 |
| 5 | 13,5 |
| 6 | 11 |
| 7 | 9,5 |
| 8 | 10 |
| 9 | 5,5 |
| 10 | 3 |

Det skal lages et C#-program der verdiene fra CHARTDATA-tabellen skal plottes i et C#-diagram.

## 17.2. Kobling fra C# mot databasen dataene skal hentes fra

For å koble seg mot databasen, dobbeltklikk på **App.config** i **Solution Explorer** (vist Figur 17-2).




**Figur 17-2: App.config er tilgjengelig i Solution Explorer.**

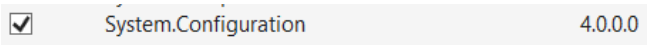
Definer «**connectionstringen**» i **App.config**, som vist i Figur 17-3.

```
<configuration>
    <connectionStrings>
        <add name="conChartDatabase" connectionString="Data Source=localhost;
        Initial Catalog = ChartDatabase; Integrated Security = True"/>
    </connectionStrings>
</configuration>
```

**Figur 17-3: Definisjon av connectionstring i App-config-fila.**

I Form-koden må **using**-statement `System.Data.SqlClient` og `System.Configuration` legges til.

Høyreklikk deretter over «**References**» i «Solution Exlorer», vist i Figur 17-2. Klikk «**Add references**» og sørg for å krysse av i sjekkboksen «**System.Configuration**», vist i Figur 17-4.




**Figur 17-4: System.Configuration må avkrysses under "References" for å legges til som referanse.**

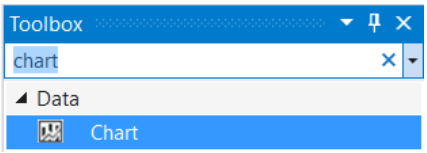
Deklarer så connection-variabelen (**conChart**) vist i Figur 17-5, som en instansvariabel direkte etter den første klammeparentesen i klassen (etter linja med class Form1 …).

```
string conChart =
ConfigurationManager.ConnectionStrings["conChartDatabase"].ConnectionString;
```

**Figur 17-5: Deklarasjon av connection-variabel i skjemaet (Form) der denne skal benyttes.**

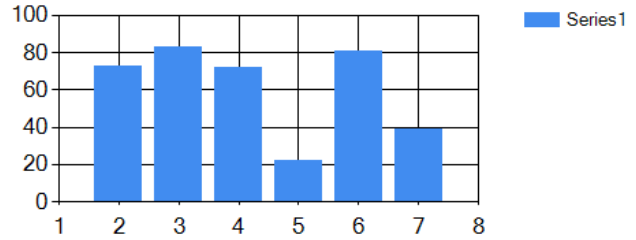
## 17.3. Chart-verktøyet i Visual Studios Toolbox

Chart-komponenten hentes fra Visual Studio's Toolbox (Figur 17-6) og dras inn på et skjema (Form).



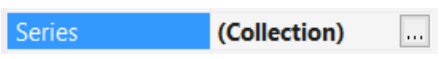

**Figur 17-6: Chart-tool hentes fra Visual Studio's Toolbox og dras ut på skjemaet (Form1).**

Et diagram tilsvarende diagrammet vist i Figur 17-7 vil bli plassert på skjemaet.



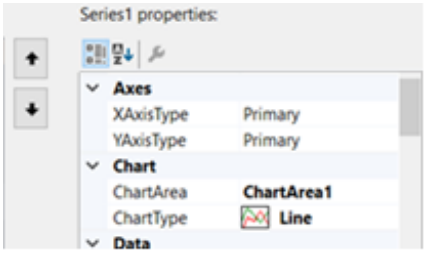

**Figur 17-7: Nytt diagram opprettet med chart-verktøyet i Visual Studio's Toolbox.**

Avmerk diagrammet. Velg så i Properties-vinduet «Series», og klikk deretter på knappen med tre prikker, vist i Figur 17-15.




**Figur 17-8: Collection-knappen i "Properties"-vinduet.**

Da vises et sett med Properties, som kan brukes til å konfigurere diagrammet. Litt av dette Properties-vinduet er vist i Figur 17-9.




**Figur 17-9: Properties-vindu for å konfigurere diagrammet.**

Sett **ChartType** til «**Line**» i Properties-vinduet vist i Figur 17-9.

## 17.4. Konfigurasjon av diagramoppsett fra C#-koden.

Det er mulig å utføre diverse konfigurasjoner av diagrammene inn i selve koden. I metoden vist i Figur 17-10, er noen av konfigurasjonsmulighetene vist. Noen av mulighetene er kommentert bort (med //).

```
void InitializeGraph()
{
   chart1.ChartAreas[0].AxisX.MajorGrid.LineColor = Color.Red;//Grid-farge x-akse
   chart1.ChartAreas[0].AxisY.MajorGrid.LineColor = Color.Blue;//Grid-farge y-akse
   chart1.ChartAreas[0].AxisX.Interval = 10.0;//Intervallinndeling (grids) x-akse
   chart1.ChartAreas[0].AxisY.Interval = 5.0;//Intervallinndeling (grids) y-akse
   //Min. og maks. akseverdi kan settes. I annet fall blir disse autokonfigurert:
   //chart1.ChartAreas[0].AxisY.Minimum = 5;//Minimumsverdi x-akse
   //chart1.ChartAreas[0].AxisY.Maximum = 40;//Maksimumsverdi x-akse
   chart1.Series["Series1"].Color = Color.DarkOrange;//Kurvefarge "Series1"
   chart1.Series["Series2"].Color = Color.Red;//Kurvefarge "Series2"

   /*Koden nedenfor viser eksempel på konfigurering av datoakse: */
   /*
   chart1.ChartAreas[0].AxisX.IntervalType = DateTimeIntervalType.Hours;//Viser timer
   chart1.ChartAreas[0].AxisX.LabelStyle.Format = "dd-MM-yyyy hh:mm";//Format på tekst

   //Nedenfor settes minimumsdato og maksimumsdato for den horsontale aksen (x-aksen)
 //Se:https://msdn.microsoft.com/en-us/library/system.datetime.tooadate(v=vs.110).aspx
   chart1.ChartAreas[0].AxisX.Minimum = new DateTime(2016, 9, 1, 1, 0, 0).ToOADate();
   chart1.ChartAreas[0].AxisX.Maximum = new DateTime(2016, 9, 2, 23, 0, 0).ToOADate();
    */
}
```

**Figur 17-10: Eksempler på konfigureringsmuligheter av chart-komponenten.**

Metoden vist i Figur 17-10 kan eksempelvis kalles opp fra «constructoren», dersom konfigurasjonene ønskes utført ved programoppstart. I annet fall kalles den opp der den behøves. I Figur 17-11 er det vist et eksempel der metoden **InitializeGraph** kalles opp fra «constructoren» til **Form1**.

```
public Form1()
{
    InitializeComponent();
    InitializeGraph();//Kaller opp metoden som konfiugrerer diagrammet chart1
}
```

**Figur 17-11: Kall på InitializeGraph-metoden fra constructoren til Form1.**

## 17.5. Plotting av verdiene fra databasetabellen i et chart

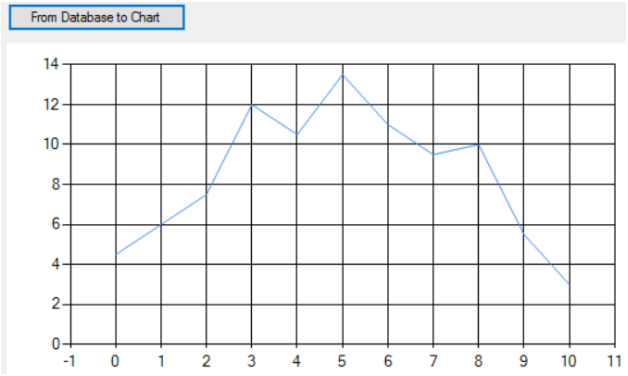
Ved klikk på knappen **btnFromDatabaseToChart** skal x- og y-verdiene leses rad for rad fra database-tabellen og fortløpende plottes i diagrammet. Koden for dette er vist i Figur 17-12.

Ved kobling mot database, er det fornuftig å omslutte koden med **try-catch**, som vist Figur 17-12, så eventuelle unntak i form av tilkoblingsfeil og annet fanges opp og kan håndteres på kontrollert vis.

```
private void btnFromDatabaseToChart_Click(object sender, EventArgs e)
{
    double valueX, valueY; //Deklarer to variabler for å ta vare på x- og y-verdi
    SqlConnection con = new SqlConnection(conChart); //Oppretter et connection-objekt
    string sqlQuery = "SELECT * FROM CHARTDATA"; //SQL for å hente databaseverdiene
    SqlCommand sql = new SqlCommand(sqlQuery, con); //Knytter spørring til connection
    con.Open();//Åpner databaseforbindelsen
    SqlDataReader dr = sql.ExecuteReader();//SqlDataReader for å lese fra databasen
    while (dr.Read() == true) //Leser rad for rad inntil det ikke er flere rader
    {
        valueX = Convert.ToDouble(dr[0]); //Leser fra kolonne 0 (kalt «XColumn»)
        valueY = Convert.ToDouble(dr[1]); //Leser fra kolonne 1 (kalt «YColumn»)
        chart1.Series["Series1"].Points.AddXY(valueX, valueY); //Plotter “x,y”-verdi
    }
    con.Close();//Stenger databaseforbindelsen
}
```

**Figur 17-12: Kode for lesing fra database med plotting av verdiene som leses i et C#-chart.**

Resultatet etter trykk på knappen **btnFromDatabaseToChart** er vist i diagrammet i Figur 17-13.




**Figur 17-13: Plotting av dataene fra tabellen CHARTDATA i et C#-chart.**

## 17.6. Plotting av flere verdier i samme diagram

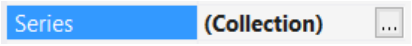
For å illustrere plotting av to verdier som en funksjon av én annen verdi, legges det til en ekstra kolonne i Tabell 5. Navnet **YColumn** fra den opprinnelige tabellen, endres til **Y1Column**. I tillegg legges det til en ny kolonne med navn **Y2Column**. Den nye tabellen med verdier er vist i Tabell 6. Tabellen beholder samme navn, dvs. **CHARTDATA**.

**Tabell 6: Tabellen CHARTDATA utvidet med en ny kolonne.**

| XColumn | Y1Column | Y2Column |
|---|---|---|
| 0 | 4,5 | 30,5 |
| 1 | 6 | 27 |
| 2 | 7,5 | 22,3 |
| 3 | 12 | 17 |
| 4 | 10,5 | 19 |
| 5 | 13,5 | 19,5 |
| 6 | 11 | 15 |
| 7 | 9,5 | 21,5 |
| 8 | 10 | 27 |
| 9 | 5,5 | 24 |
| 10 | 3 | 25,5 |

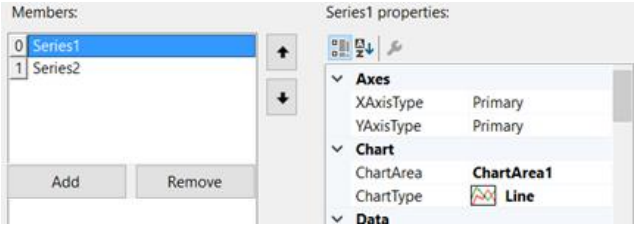
**Figur 17-14: Tabellen CHARTDATA utvidet med kolonnen Y2Column. YColumn er endret til Y1Column.**

For å plotte to kurver i samme chart, klikk da først på diagrammet i designmodus, så dette blir avmerket. I Properties-vinduet, klikk på **Collection**-knappen, som er påført tre prikker (jf. Figur 17-15).




**Figur 17-15: Collection-knappen i "Properties"-vinduet.**

Velg så **Add**-knappen vist i Figur 17-16. Det legges da til en ny dataserie («**Series2**») som kan plottes i tillegg til den opprinnelige «**Series1**». La begge serienes «**ChartType**» være av typen «**Line**».



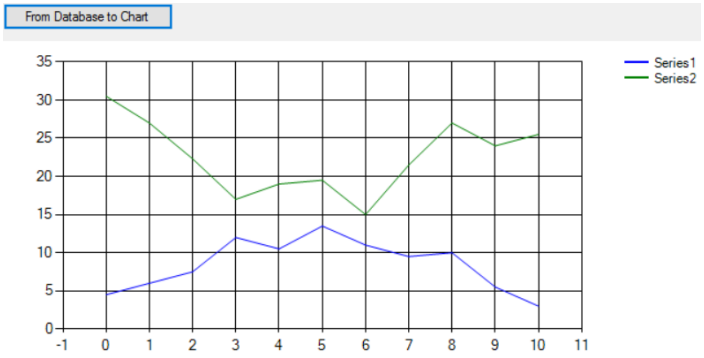

**Figur 17-16: Chart-Property der det kan legges til en ny "Series".**

For å plotte begge samtidig, kan koden fra Figur 17-12 omskrives som vist i Figur 17-17.

```
private void btnFromDatabaseToChart_Click(object sender, EventArgs e)
{
    double valueX, valueY1, valueY2; //Variabler for verdier som skal plottes
    string sqlQuery = "SELECT * FROM CHARTDATA"; //SQL for å hente tabelldata
    try
    {
        SqlConnection con = new SqlConnection(conChart); //Connection-objekt
        SqlCommand sql = new SqlCommand(sqlQuery, con); //SQLCommand-objekt
        con.Open();//Åpner databaseforbindelsen
        SqlDataReader dr = sql.ExecuteReader();//SqlDataReader for databaselesing
        while (dr.Read() == true) //Leser rad for rad inntil det ikke er flere rader
        {
            valueX = Convert.ToDouble(dr[0]); //Leser fra kolonne 0 (kalt «XColumn»)
            valueY1 = Convert.ToDouble(dr[1]); //Leser fra kolonne 1 (kalt «YColumn»)
            valueY2 = Convert.ToDouble(dr[2]); //Leser fra kolonne 2 (kalt «YColumn»)
            chart1.Series["Series1"].Points.AddXY(valueX, valueY1); //Plotter (x,y1)
            chart1.Series["Series2"].Points.AddXY(valueX, valueY2); //Plotter (x,y2)
        }
        con.Close();//Stenger databaseforbindelsen
    }
    catch (Exception error)
    {
        MessageBox.Show(error.Message);
    }
}
```

**Figur 17-17: Kode for lesing fra database med plotting av verdiene som leses i et C#-chart.**

Resultatet vil bli som vist i Figur 17-18.




**Figur 17-18: Plott av to y-verdier som funksjon av én x-verdi.**

## 17.7. Alternativ plotting med mellomlagring i List-variabler

Data skal nå leses fra databasen og lagres i **List**-variabler før de plottes i diagrammet. Da kan de samme dataene også brukes i flere sammenhenger i programmet, hvis det for eksempel skal gjøres beregninger på dem (gjennomsnittsberegninger eller annet), uten at de må hentes fra databasen på nytt.

**List**-variabler har en **Add**-metode som gjør det enkel å legge til flere elementer i den dynamiske **List**-strukturen (i motsetning til en **Array**, som har et på forhånd bestemt antall plasser).

Først deklareres tre **List**-variabler som vist i Figur 17-19. Istedenfor metoden **AddXY** må nå metoden **DataBindXY** benyttes, så hele dataserien blir bundet til grafen og plottet.

Modifisert kode er vist i Figur 17-19. Et klikk på knappen **btnChartViaListVariables** skal gi samme diagramresultat som det som tidligere ble vist i Figur 17-18.

```
private void btnChartViaListVariables_Click(object sender, EventArgs e)
{
    List<double> xValues = new List<double>();//List-variabel for x-verdier
    List<double> y1Values = new List<double>();//List-variabel for y1-verdier
    List<double> y2Values = new List<double>();//List-variabel for y2-verdier
    try
    {
        SqlConnection con = new SqlConnection(conChart); //Oppretter connection-objekt
        string sqlQuery = "SELECT * FROM CHARTDATA"; //SQL for å hente databaseverdier
        SqlCommand sql = new SqlCommand(sqlQuery, con); //Lager SQLCommand-objekt
        con.Open();//Åpner databaseforbindelsen
        SqlDataReader dr = sql.ExecuteReader();//SqlDataReader for databaselesing
        while (dr.Read() == true) //Leser rad for rad inntil det ikke er flere rader
        {
            xValues.Add(Convert.ToDouble(dr[0])); //Fra kolonne 0 til List-variabel
            y1Values.Add(Convert.ToDouble(dr[1])); //Fra kolonne 1 til List-variabel
            y2Values.Add(Convert.ToDouble(dr[2])); //Fra kolonne 2 til List-variabel
        }
        chart1.Series["Series1"].Points.DataBindXY(xValues, y1Values); //Plot (x,y1)
        chart1.Series["Series2"].Points.DataBindXY(xValues, y2Values); //Plot (x,y2)
        con.Close();//Stenger databaseforbindelsen
    }
    catch (Exception error)
    {
        MessageBox.Show(error.Message); //Feilmelding dersom unntak inntreffer
    }
    ShowListValuesInTextBox(xValues, y1Values, y2Values); //Lag metode for å vise data
}
```

**Figur 17-19: Data leses fra tabell til List-variabler og plottes i C#-chart.**

## 17.8. Skrive ut List-variablenes innhold til en tekstboks

I koden i Figur 17-19 lagres dataene fra databasen i de tre List-variablene **xValues**, **y1Values** og **y2Values**. I Figur 17-20 er det vist en **metode** som kan motta disse tre **List**-variablene som parametere, for så å skrive disse til en tekstboks kalt **txtShow**.

```
void ShowListValuesInTextBox(List<double> xValues, List<double> y1Values, List<double>
y2Values)
{
    txtShow.Clear(); //Fjerner eventuelt tekstboksinnhold før start
    foreach (double x in xValues) //Skriver ut alle x1-verdiene
    {
        txtShow.AppendText(x + "\t");
    }
    txtShow.AppendText("\r\n"); //Linjeskift
    foreach (double y1 in y1Values)//Skriver ut alle y1-verdiene
    {
        txtShow.AppendText(y1 + "\t");
    }
```

```
    txtShow.AppendText("\r\n"); //Linjeskift
    foreach (double y2 in y2Values) //Skriver ut alle y2-verdiene
    {
        txtShow.AppendText(y2 + "\t"); //Legger til tabulatortegn mellom hver verdi
    }
}
```

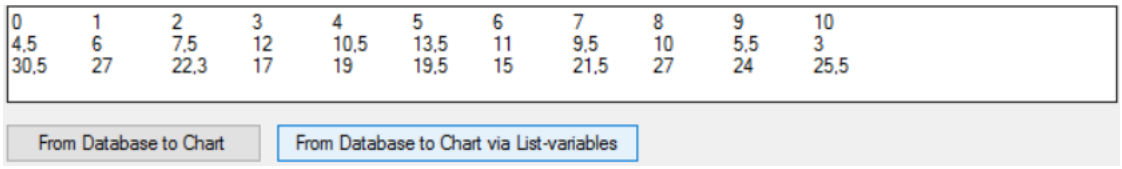
**Figur 17-20: Metode for å vise innholdet i de tre List-variablene i en tekstboks.**

Metoden kan kalles opp fra koden vist i Figur 17-19, ved å legge til metodekallet vist i Figur 17-21 i en linje direkte etter instruksjonen **con.Close** ().

```
ShowListValuesInTextBox(xValues, y1Values, y2Values);
```

**Figur 17-21: Metodekall som må legges til etter linja med «con.Close()» i koden vist i Figur 17-19.**

Resultatet av verdiene som skrives til tekstboksen er vist i Figur 17-22.




**Figur 17-22: Innholdet i de tre List-variablene xValues, y1Values og y2Values vist i tekstboksen txtShow.**

## 17.9. Legg til nye List-verdier og lag nytt plott med alle verdiene.

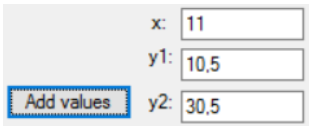
Flere List-verdier kan legges til ved å benytte **Add**-metoden. Det skal nå leses verdier fra tre tekstbokser inn i **List**-variablene ved trykk på knappen **btnAdd**. Diagrammet tegnes deretter på nytt.

For at dette skal fungere må de tre **List**-variablene legges «globalt», altså som instansvariabler, så de blir tilgjengelig i alle metodene. Dette betyr at de tre **List**-variablene vist i Figur 17-19, må flyttes så de blir liggende øverst i koden, direkte under klassenavnet. Dette er illustrert i Figur 17-23.

```
public partial class FormFromDatabaseTableToChart : Form
{
        string conChart =
        ConfigurationManager.ConnectionStrings["conChartDatabase"].ConnectionString;
        List<double> xValues = new List<double>();
        List<double> y1Values = new List<double>();
        List<double> y2Values = new List<double>();
```

**Figur 17-23: De tre List-variablene flyttes og gjøres til instansvariabler (dvs. globale variabler i klassen).**

Brukergrensesnittet utvides som vist i Figur 17-24.




**Figur 17-24: Brukergrensesnittet utvides med en knapp og tre tekstbokser, for registrering av nye verdier.**

Hendelsesmetoden til knappen **btnAdd** kan deretter lages som vist i Figur 17-25.

```
private void btnAdd_Click(object sender, EventArgs e)
{
    double x, y1, y2;
    x = Convert.ToDouble(txtX.Text);
    y1 = Convert.ToDouble(txtY1.Text);
    y2 = Convert.ToDouble(txtY2.Text);
    xValues.Add(x);
    y1Values.Add(y1);
    y2Values.Add(y2);
    chart1.Series["Series1"].Points.DataBindXY(xValues, y1Values); //Plot (x,y1)
    chart1.Series["Series2"].Points.DataBindXY(xValues, y2Values); //Plot (x,y2)
```

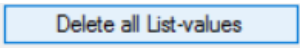
```
}
```

**Figur 17-25: Kode for å legge til nye verdier til de tre List-variablene xValues, y1Values og y2Values.**

Det kan nå legges til nye verdier med trykk på «Add values»-knappen (**btnAdd**). For hver nye innlagte verdi tegnes diagrammet på nytt med både nye og gamle verdier.

## 17.10. Slette alle verdiene fra en liste og plott diagrammet på nytt

Dersom verdiene ønskes slettet fra en liste, gjøres dette med **List**-variabelens **Clear**-metode. Legg til knappen **btnDelete**, vist i Figur 17-26.




**Figur 17-26: Knapp (Name. btnDelete) for å slette innholdet i de tre List-variablene).**

Hendelsesmetoden til **btnDelete**-knappen blir som vist i Figur 17-27. Verdiene i **List**-variablene slettes, og diagrammet tegnes på nytt. Grafene som var der i utgangspunktet vil da bli borte.

```
private void btnDelete_Click(object sender, EventArgs e)
{
    xValues.Clear();
    y1Values.Clear();
    y2Values.Clear();
    chart1.Series["Series1"].Points.DataBindXY(xValues, y1Values); //Plot (x,y1)
    chart1.Series["Series2"].Points.DataBindXY(xValues, y2Values); //Plot (x,y2)
}
```

**Figur 17-27: Kode for å slette alt innhold i de tre List-variablene xValues, y1Values og y2Values.**

## 17.11. Diagram med datoverdier langs den horisontale aksen.

Det skal lages et eksempel med bruk av dataverdier langs den horisontale aksen og flyttall langs den vertikale aksen.

Legg til en ny tabell kalt **CHARTWITHDATEVALUES** i SQL Server-databasen **ChartDatabase**, med tabelldesign som vist i Figur 17-28.

| | Column Name | Data Type | Allow Nulls |
|---|---|---|---|
| 🔑 | DateValues | datetime | ☐ |
| | Y1Column | float | ☑ |
| | Y2Column | float | ☑ |

**Figur 17-28: Design av tabellen CHARTWITHDATEVALUES.**

Registrer dataene vist i Tabell 7.

**Tabell 7: Data som skal legges inn i tabellen CHARTWITHDATEVALUES.**

| DateValues | Y1Column | Y2Column |
|---|---|---|
| 2016-05-09 | 10,5 | 24,5 |
| 2016-08-01 | 5 | 30 |
| 2016-08-20 | 7,6 | 32,7 |
| 2016-09-27 | 13,2 | 27 |
| 2016-10-07 | 12,1 | 24,5 |
| 2016-11-12 | 7 | 19,7 |

Legg inn knappen vist i Figur 17-29 på skjemaet, og gi den «Name»: **btnDatevaluesFromDatabaseToChart**.


**Figur 17-29: Knapp som skal aktivere plott i chart med datoverdier langs den horisontale aksen.**

Koden for knappens hendelsesrutine er vist i Figur 17-30.

For mer informasjon om hvordan datoaksen kan formateres, se Figur 17-10 i Kapittel 17.4.

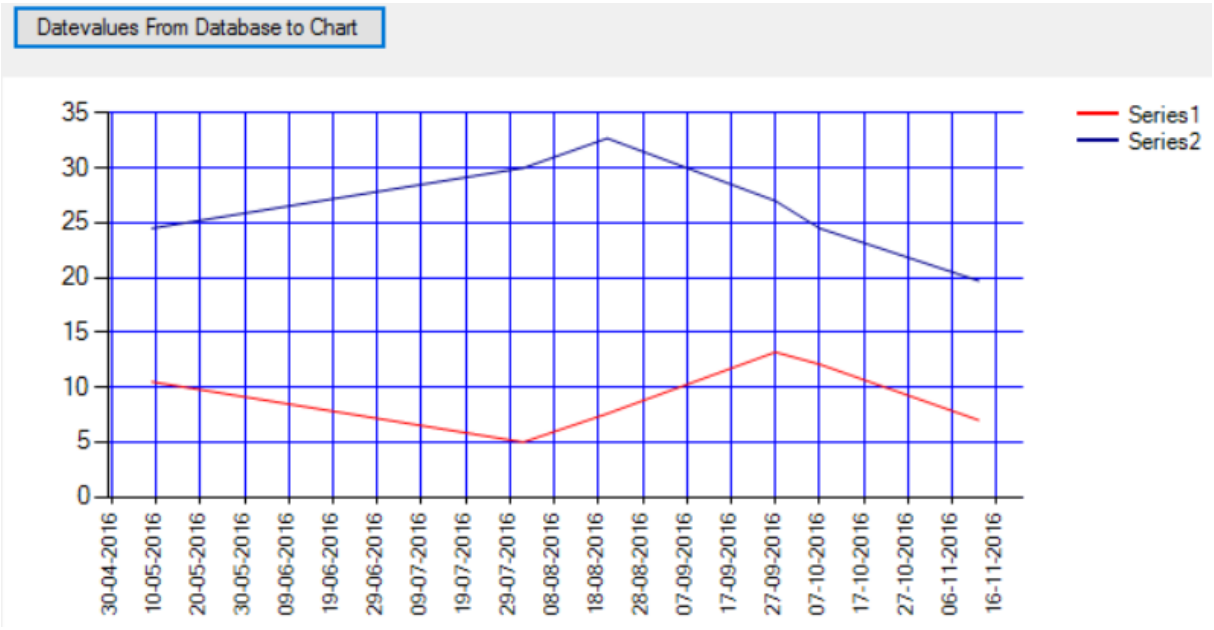
```
private void btnDatevaluesFromDatabaseToChart_Click(object sender, EventArgs e)
{
    DateTime valueX; //Horisontal akse skal være dato-akse
    double valueY1, valueY2; //Deklarer variabler for verdier som skal plottes
    string sqlQuery = "SELECT * FROM CHARTWITHDATEVALUES"; //SQL som skal utføres
    InitGraph();
    try
    {
        SqlConnection con = new SqlConnection(conChart); //Lager connection-objekt
        SqlCommand sql = new SqlCommand(sqlQuery, con); //Lager SqlCommand-objekt
        con.Open();//Åpner databaseforbindelsen
        SqlDataReader dr = sql.ExecuteReader();//SqlDataReader for databaselesing
        while (dr.Read() == true) //Leser rad for rad inntil det ikke er flere rader
        {
            valueX = Convert.ToDateTime(dr[0]); //Leser fra kolonne 0 (kalt «XColumn»)
            valueY1 = Convert.ToDouble(dr[1]); //Leser fra kolonne 1 (kalt «Y1Column»)
            valueY2 = Convert.ToDouble(dr[2]); //Leser fra kolonne 1 (kalt «Y2Column»)
            chart1.Series["Series1"].Points.AddXY(valueX, valueY1); //Plotter (x,y1)
            chart1.Series["Series2"].Points.AddXY(valueX, valueY2); //Plotter (x,y2)
        }
        con.Close();//Stenger databaseforbindelsen
    }
    catch (Exception error)
    {
        MessageBox.Show(error.Message); //Fanger eventuelle unntak og gir feilmelding
    }
}
```

**Figur 17-30: Kode for å lage et chart med dato-verdier langs den horisontale aksen.**

Diagrammet vist i Figur 17-31 viser resultatet av koden fra Figur 17-30.

Resultatet viser et plot der tallet 10 er angitt som intervall langs den horisontale aksen, sånn at det tegnes en vertikal grid-linje (rutenettlinje) for hver tiende dag. Dataene som plottes er vis i Tabell 5.

Langs den vertikale aksen er intervallet 5.




**Figur 17-31: Et diagram plottet med datoverdier langs den horisontale aksen.**

# 18. Transaksjoner

I dette kapittelet gis det en kort introduksjon til transaksjoner, med et påfølgende transaksjonseksempel med C# og SQL Server. Transaksjoner involverer gjerne en av de tre SQL-operasjonene **INSERT INTO**, **UPDATE** eller **DELETE**, der resultatet er en endring av dataene i databasen etter utført transaksjon.

Noen ganger er det sånn at **én** SQL-spørring påvirker bare **én** rad i en tabell, som for eksempel ved bruk av «**INSERT INTO**», som kan benyttes til å sette ***én*** ny rad inn i en tabell. Andre ganger kan ***én*** SQL-setning påvirke mange rader, som for eksempel «**DELETE FROM TABLENAME**», som vil slette alle rader i en tabell. I tillegg utføres ofte flere SQL-setninger etter hverandre, på en samhørende måte som gjør at de ønskes betraktet som én transaksjon.

Et klassisk transaksjonseksempel er en situasjon der et beløp skal flyttes fra én bankkonto til en annen. La oss eksempelvis si at kr 1000 skal flyttes fra kontonummer 32325577154 til kontonummer 44327832156. Da må den ene kontoens balanse debiteres med kr 1000,- og den andre kontoens balanse krediteres med kr 1000,-. Man kan tenke seg dette utføres med to SQL-setninger.

Et **UPDATE**-statement benyttes først for å ta kr 1000,- ut av kontonummer 32325577154, og et nytt **UPDATE**-statement benyttes så for å sette kr 1000,- inn på konto 44327832156, illustrert i Figur 18-1.

```
UPDATE ACCOUNT
SET Balance = Balance - 1000
WHERE AccountNumber = 32325577154

UPDATE ACCOUNT
SET Balance = Balance + 1000
WHERE AccountNumber = 44327832156
```

**Figur 18-1: UPDATE-instruksjoner brukt til å overføre kr 1000,- fra én konto til en annen.**

Hvis det skulle inntreffe en feil i applikasjonen etter den første av disse to transaksjonene, så den andre ikke blir utført, vil dette innebære at kr 1000,- er trukket fra den ene kontoen, men ikke godskrevet den andre. Kr 1000,- er da «forsvunnet». Dette er naturlig nok ikke en akseptabel risiko.

## 18.1. BeginTransaction og Commit

For å unngå feil i databasen som konsekvens av at ikke alle samhørende operasjoner utføres riktig, innføres begrepet **transaksjon** [11] En transaksjon er en eller flere operasjoner mot databasen, som blir definert og håndtert som én enhet. Før en transaksjon utføres, er databasen i en konsistent (gyldig) tilstand. En transaksjon er ansvarlig for å føre databasen over i en ny konsistent tilstand eller å ved feil å avbryte transaksjonen og føre databasen tilbake til den siste konsistente tilstand.

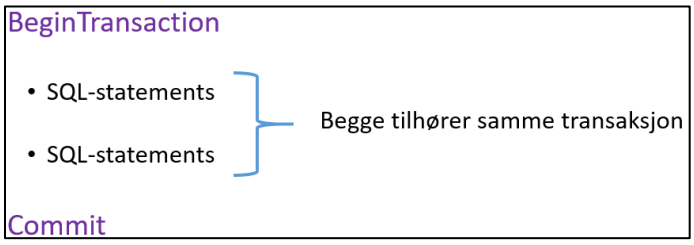
For å være sikre på at kontoenes balanse, jf. Figur 18-1, oppdateres korrekt, må enten ***ingen*** av de to SQL-setningene eller ***begge*** SQL-setningene utføres. Vi kan derfor se på disse to som ***én*** samhørende enhet og definerer dem som én transaksjon.

I SQL Server finnes det ulike klassebiblioteker med transaksjonsmetoder. Der finnes det blant annet metoder for å definere starten og slutten av en transaksjon [11]:

1. BeginTransaction()
2. Commit()

Det er programmereren som må ta stilling til hvilke instruksjoner som tilhører en transaksjon. Når det er tatt stilling til dette, benyttes metoden **BeginTransaction** til å markere starten av en transaksjon og metoden **Commit** til å markere slutten, dersom alt har gått bra. Det viktige er at databasen etter transaksjonen er ført til en ny konsistent tilstand, hvilket vil si at alle operasjoner mot databasen er fullført i tråd med planen. I annet fall må databasen føres tilbake til den tilstanden den var i før transaksjonen (eventuelt at transaksjonen fullføres ut fra en transaksjonsplan).

De to SQL-operasjonene fra Figur 18-1 kunne da vært samlet i én transaksjon, etter prinsippet illustrert i Figur 18-2. Dette sånn at systemet kun ville registrere at en konsistent tilstand er nådd dersom begge de to operasjonene i transaksjonen har blitt fullført og databasen er oppdatert med de riktige dataene.




**Figur 18-2:Illustrasjon på hvordan samhørende SQL-instruksjoner kan defineres som én transaskjon.**

## 18.2. Transaksjonslogg

Mens en disk (eksempelvis en SSD, dvs. en Solid State Disk, eller en harddisk) er et lagringsmedium for permanent lagring, der data fortsatt er tilgjengelig når strømmen brytes, er RAM (Random Access Memory) et såkalt flyktig (volatile) minne, som kun beholder dataene mens programmet utføres.

Under en databasetransaksjonen vil noen data befinne seg i **RAM** (Random Access Memory) og noen data befinne seg på **disk**. Dataene leses til/fra fra disk og behandles i **RAM**, der operasjoner kan utføres langt raskere. Først når transaksjonen utfører en **Commit**, blir dataene lagret permanent på disk.

Når et program kjøres, kan ulike typer feil inntreffe. Eksempelvis kan det inntreffe en minnefeil (for eksempel på grunn av et strømbrudd), diskkrasj eller andre typer feil.

For at et databasehåndteringsystemet (DBHS) skal være i stand til å «rydde opp» dersom det inntreffer feil under en transaksjon, benyttes en **transaksjonslogg** [12]. En **transaksjonslogg** lagres som én eller flere filer i databasen, der det registreres informasjon om transaksjonen.

Ved feil kan DBHS benytte **transaksjonsloggen** til å bringe systemet tilbake til den siste **konsistente** tilstanden systemet var i før feilen inntraff eller loggen kan eventuelt benyttes til å fullføre database-operasjonene som ikke ble utført grunnet feilen som inntraff. Dette er altså to ulike tilnærminger:

1. **Rollback** (undo operations): «Ruller» databasen tilbake til tilstanden den var i før transaksjonen ble påbegynt.

2. **Roll forward** (Redo/Resubmitt). «Ruller» databasen frem ved fullføring av operasjonene, ved bruk av transaksjonsloggen der kommende operasjoner er registrert før utførelse.

| TransactionId | Operation | Table | Attribute | Row ID | Before | After |
|---|---|---|---|---|---|---|
| 0000:00000232 | BeginTrans | | | | | |
| 0000:00000232 | UPDATE | ACCOUNT | Balance | 12355 | 10 000 | 9 000 |
| 0000:00000232 | UPDATE | ACCOUNT | Balance | 12363 | 1000 | 2000 |
| 0000:00000232 | Commit | | | | | |

**Figur 18-3: Eksempel på en transaksjonslogg.**

Tabellen i Figur 18-3 viser en forenklet fremstilling av en transaksjonslogg for loggføring av transaksjonen som inkluderer de to operasjonene illustrert i Figur 18-1. En slik logg vil ha flere kolonner enn dem som er vist her, blant annet for starttid og sluttid. Ut fra dataene registrert i tabellen vil systemet kunne rulle databasen tilbake eller bruke loggen til å fullføre ufullførte transaskjoner.

Alle databaser trenger en slik logg for å kunne sikre at databasen inneholder korrekte data og en backup-rutine på en loggfiler vil derfor også være viktig, i tilfelle for eksempel diskkrasj.

## 18.3. Rollback

I kapittel 18.1 ble det vist hvordan C# har klasser som inkluderer metodene **BeginTransaction** og **Commit**. Dersom en feil inntreffer som gjør at en påbegynt transaksjon ikke kan fullføres, finnes det en metode kalt **Rollback**, som kan «rulle tilbake» instruksjonene som er utført ***etter*** at transaksjonen startet. Dette så man ved å kalle opp denne metoden, kan få databasen ført tilbake til den tilstanden den var i da transaksjonen ble påbegynt. **DBHS** vil da bruke transaksjonsloggen til å tilbakestille operasjonene.

En måte å lage et C#-program på for å sikre at en transaksjon ikke fører til feil, er å legge transaksjonskoden som skal utføres i en **try**-block og håndtere eventuelle feil i en **catch**-block. I **try**-blokken defineres transaksjonens start og slutt med henholdsvis **BeginTransaction** og **Commit**.

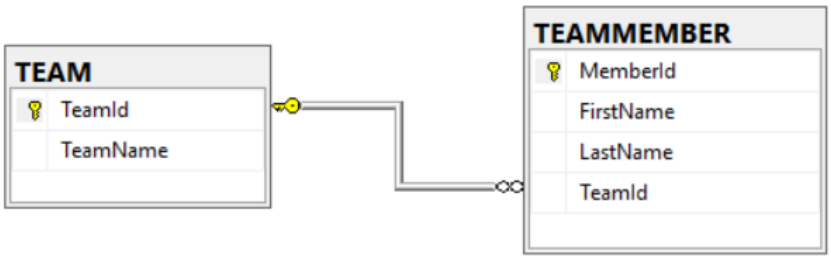
Inntreffer feil ***før* Commit** er nådd, sikres tilbakerulling til forrige konsistente tilstand ved å kalle opp **Rollback**-metoden i **catch**-blokken. **Rollback** vil bare utføres dersom en transaksjon ***er*** påbegynt og ***ikke fullført*** dersom en feil inntreffer. Den prinsipielle løsningen blir som vist i Figur 18-4.

```
try
{
    //BeginTransaction

    //Update ... Hvis denne delen av transaksjonen feiler, hopper
    //           programmet til catch-blokken

    //Update ... Denne utføres bare hvis den første Update lykkes

    //Commit (Kommer programmet hit, er transaksjonen fullført)
}
catch (Exception)
{
    //Rollback (Inntreffer feil, rulles transaksjonen tilbake)
}
```

**Figur 18-4: Prinsipp for utførelse av en transaksjon med rollback ved eventuell feil.**

## 18.4. Et transaksjonseksempel med bruk av C# og SQL Server

Som illustrasjon på transaksjoner, skal det utføres en transaksjon relatert til tabellene vist i Figur 18-5. I tabellen **TEAM** registreres ulike team/grupper. Hvert team består av et antall medlemmer, registrert i tabellen **TEAMMEMBER**, og hvert medlem (team member) tilhører ***ett*** team. Det er altså et en-til-mange-forhold mellom tabellene, slik det fremgår av SQL Server-diagramvisning i Figur 18-5. Fremmednøkkelen **TeamId** i **TEAMMEMBER** er en referanse til primærnøkkelen **TeamId** i **TEAM**.




**Figur 18-5: Tabellene TEAM og TEAMMEMBER i SQL Server.**

I tabellene **TEAM** legges forekomstene (verdiene) vist i Figur 18-6 inn.

| TeamId | TeamName |
|---|---|
| 1 | Team 1 |
| 2 | Team 2 |
| 3 | Team 3 |
| 4 | Team 4 |

**Figur 18-6: Fire rader med verdier er lagt inn i tabellen TEAM, en for hvert team.**

I tabellene **TEAMMEMBER** legges forekomstene (verdiene) vist i Figur 18-7 inn.

| MemberId | FirstName | LastName | TeamId |
|---|---|---|---|
| 1 | Huey | Duck | 1 |
| 2 | Dewey | Duck | 1 |
| 3 | Louie | Duck | 1 |
| 4 | Practical | Pig | 2 |
| 5 | Fiddler | Pig | 2 |
| 6 | Fifer | Pig | 2 |

**Figur 18-7: Seks rader, med tre ender og tre griser, er lagt inn i tabellen TEAMMEMBER.**

Som det fremgår av Figur 18-7, er de tre endene lagt til i **Team 1** (TeamId = 1) og de tre grisene er lagt til i Team 2 (TeamId 2). I **Team 3** og **Team 4** er det ikke lagt inn noen medlemmer.

For å få minst én gris og én and på hvert team, skal en av grisene bytte team med en av endene. Til å utføre dette, skal det benyttes to **UPDATE**-instruksjoner, én for å flytte anden **Louie Duck** fra **Team 1** til **Team 2** og en annen for deretter å flytte grisen **Practical Pig** fra **team 2** til **team 1**.

Først defineres en «**connectionString** i **App.Config**, på tilsvarende måte som forklart i kapittel 16.4.1. Connectionstringen i App.Config er vist i Figur 18-8.

```
<connectionStrings>
    <add name="conProjectDatabase" connectionString="Data source =
     localhost; Initial Catalog = ProjectDatabase; Integrated Security = True" />
</connectionStrings>
```

**Figur 18-8: Definering av connectionstring i App.Config.**

I Form1 deklareres deretter en connectionstring-variabel, som vist i Figur 18-9.

```
public partial class Form1 : Form
{
    string conProject = ConfigurationManager.
           ConnectionStrings["conProjectDatabase"].ConnectionString;
```

**Figur 18-9: Deklarasjon av en connectionstring-variabel med navnet conProject.**

**NB!** For at ovennevnte koblinger skal virke, må **using**-statement og **System.Configuration** legges til, som tidligere beskrevet i henholdsvis kapittel 16.4.2 og 16.4.3.

Det skal så lages en «**click event handler**» for knappen **btnSwapTeams**. Ved trykk på denne knappen, skal anda og grisen bytte plass. Nedenfor er koden vist, plassert i en **try-catch-finally**-struktur.

Innledningsvis i metoden opprettes først noen variabler. Deretter defineres de to SQL-setningene som skal utføres (to stk. **UPDATE**-instruksjoner), som lagres i to string-variabler. Se Figur 18-10.

```
private void btnSwapTeams_Click(object sender, EventArgs e)
{
    string sqlQuery1, sqlQuery2;
    SqlTransaction transaction = null; //Lager en variabel for transaksjonen
    SqlConnection con = null; //Lager en connection-variabel
    SqlCommand command = null; //Lager en variable for sql-operasjoner mot basen
    //Definerer og lagrer den første spørringen. Flytter medlem 3 til Team 2:
    sqlQuery1 = String.Concat(@"UPDATE TEAMMEMBER
                               SET TEAMID = 2
                               WHERE MemberId = 3");
    //Definerer og lagrer den andre spørringen. Flytter medlem 4 til Team 1:
    sqlQuery2 = String.Concat(@"UPDATE TEAMMEMBER
                               SET TEAMID = 1
                               WHERE MemberId = 4");
```

**Figur 18-10: Innledende kode i "click event handleren" til knappen btnSwapTeams.**

Koden er vist i Figur 18-11. **Try-catch** er benyttet i tilfelle noe går galt under databaseoperasjonene. Lukking av databasen utføres i en **Finally**-blokk, sånn at dette utføres uansett feil eller ikke.

```
try
{
    con = new SqlConnection(conProject); //Et connection-objekt opprettes.
    con.Open(); //Forbindelsen til databasen åpnes
    command = new SqlCommand(sqlQuery1, con); //Oppretter SqlCommand-objekt for query 1
    command.ExecuteNonQuery(); //Den første UPDATE-spørringen utføres
    command = new SqlCommand(sqlQuery2, con); //Oppretter SqlCommand-objekt for query 2
    command.ExecuteNonQuery(); //Den andre UPDATE-spørringen utføres
}
catch (Exception error)
{
    MessageBox.Show(error.Message + " : " + error.GetType());//Viser melding dersom feil
}
finally
{
    if (con.State == ConnectionState.Open) //Sjekker om databaseforbindelsen er åpen
    {
        con.Close(); //Dersom åpen databaseforbindelse, så stenges den
    }
}
```

**Figur 18-11:Oppretter forbindelse til databasen og utfører databaseoperasjonene.**

Det er nå laget et program der **Louie Duck** bytter team med **Practical Pig**. Når det trykkes på knappen med navn **btnSwapTeams**, utføres byttet. Ved å utføre en **SELECT**-setning for å vise tabellinnholdet i **SQL SERVER** før og etter trykk på knappen, skal resultatet bli som vist i Figur 18-12.

| MemberId | FirstName | LastName | TeamId |
|---|---|---|---|
| 1 | Huey | Duck | 1 |
| 2 | Dewey | Duck | 1 |
| 3 | Louie | Duck | 1 |
| 4 | Practical | Pig | 2 |
| 5 | Fiddler | Pig | 2 |
| 6 | Fifer | Pig | 2 |

| MemberId | FirstName | LastName | TeamId |
|---|---|---|---|
| 1 | Huey | Duck | 1 |
| 2 | Dewey | Duck | 1 |
| 3 | Louie | Duck | 2 |
| 4 | Practical | Pig | 1 |
| 5 | Fiddler | Pig | 2 |
| 6 | Fifer | Pig | 2 |

**Figur 18-12: Bildet til venstre viser tabellinnholdet til TEAMMEMBER før endring og til høyre etter.**

Dette tyder på at begge de to **UPDATE**-instruksjonene er utført riktig, så databasen er ført over i en ny konsistent tilstand.

Men hva hvis bare den første **UPDATE**-setningen hadde blitt utført og ikke den andre? Dette kan testes ved for eksempel å endre litt på **UPDATE**-setning to, så denne utfører en operasjon som vil feile.

Får å gjøre dette, omgjøres den andre **UPDATE**-setningen til å flytte deltakeren til **Team 5** istedenfor til **team 1**. Siden **Team 5** ikke finnes i tabellen **TEAM**, jf. Figur 18-13, betyr dette at **UPDATE**-setningen vil feile. En «**run time-error**» vil inntreffe, og systemet vil hoppe til **catch**-blokken, der en feilmelding vil skrives til en **MessageBox**.

| MemberId | FirstName | LastName | TeamId |
|---|---|---|---|
| 1 | Huey | Duck | 1 |
| 2 | Dewey | Duck | 1 |
| 3 | Louie | Duck | 1 |
| 4 | Practical | Pig | 2 |
| 5 | Fiddler | Pig | 2 |
| 6 | Fifer | Pig | 2 |

| TeamId | TeamName |
|---|---|
| 1 | Team 1 |
| 2 | Team 2 |
| 3 | Team 3 |
| 4 | Team 4 |

Team 5 finnes ikke

**Figur 18-13: Den andre UPDATE-setningen prøver å flytte MemberId 4 til Team 5, som ikke finnes.**

For å utføre denne endringen, omgjøres koden fra Figur 18-10 til koden vist i Figur 18-14.

```
private void btnSwapTeams_Click(object sender, EventArgs e)
{
    string sqlQuery1, sqlQuery2;
    SqlTransaction transaction = null; //Lager en variabel for transaksjonen
    SqlConnection con = null; //Lager en connection-variabel
    SqlCommand command = null; //Lager en variable for sql-operasjoner mot basen
    //Definerer og lagrer den første spørringen. Flytter medlem 3 til Team 2:
    sqlQuery1 = String.Concat(@"UPDATE TEAMMEMBER
                               SET TEAMID = 2
                               WHERE MemberId = 3");
    //Definerer og lagrer den andre spørringen. Flytter medlem 4 til Team 5:
    sqlQuery2 = String.Concat(@"UPDATE TEAMMEMBER
                               SET TEAMID = 5
                               WHERE MemberId = 4");
```

**Figur 18-14: Koden fra Figur 18-10 omgjøres så MemberId 4 flyttes til Team 4 istedenfor til Team 1.**

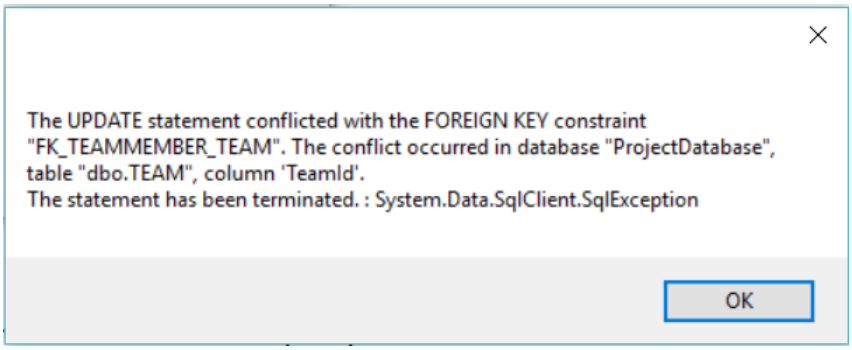
Før programmet kjøres på nytt, kan SQL-setning vist i kjøres SQL-server for å tilbakestille tabellen som ble endret til sin opprinnelige tilstand (med tre ender i Team 1 og tre griser i Team 2).

```
UPDATE TEAMMEMBER
SET TeamId = 1
WHERE MemberId = 3

UPDATE TEAMMEMBER
SET TeamId = 2
WHERE MemberId = 4
```

**Figur 18-15: SQL som kjøres i SQL Server for å tilbakestille TEAMMEMBER til sine opprinnelige verdier.**

Ved nå å trykke på **btnSwapTeams**-knappen, utføres programmet på nytt med de nye **UPDATE**-spørringene vist i Figur 18-14.

Resultatet ved kjøring er at den første UPDATE-spørringen utføres riktig, mens det inntreffer en feil når den andre kjøres. Et unntak (Exception) blir kastet (thrown) og feilmeldingen vist i Figur 18-16 vises.




**Figur 18-16: Feilmelding som vises når den andre UPDATE-spørringen utføres.**

Som det ses, står det at transaksjonen ender med en fremmednøkkelkonflikt, fordi det forsøkes å registrere en person som medlem av **Team 5**, som ikke er registrert i tabellen **TEAM**. Man prøver altså å referere noe som ikke finnes, hvilket er et brudd på referanseintegritet (jf. kapittel 7.4).

Dette betyr at **DBHS** avbryter denne spørringen. Hvis man deretter lager en spørring for å vise innholdet i **TEAMMEMBER**-tabellen etter oppdateringen (jf. Figur 18-17), ser man at resultatet ikke er riktig.

Siden disse to UPDATE-spørringene ikke er definert som én transaksjon, har vi nå en situasjon der en av spørringene er utført og den andre ikke. Da den første **UPDATE**-spørringen ble utført, inntraff ingen feil. Louie Duck, med **MemberId** 3 ble i den overført til **Team 2**. Når så neste **UPDATE** skulle utføres, feilet programmet. Practical Pig, med **MemberId** 4, ble derfor ikke flyttet til **Team 1.** Databasen har derfor ikke kommet til en ny «konsistent» (riktige) tilstand og er heller ikke rullet tilbake til siste konsistente tilastand. Resultatet blir som vist i Figur 18-17, altså en halvveis utført transaksjon.

| MemberId | FirstName | LastName | TeamId |
|---|---|---|---|
| 1 | Huey | Duck | 1 |
| 2 | Dewey | Duck | 1 |
| 3 | Louie | Duck | 2 |
| 4 | Practical | Pig | 2 |
| 5 | Fiddler | Pig | 2 |
| 6 | Fifer | Pig | 2 |

**Figur 18-17: Resultat etter kjøring av programmet. Én av UPDATE-spørringene er utført og en annen ikke.**

Årsaken er altså **DBHS** betrakter hver av **UPDATE**-spørringene som separate operasjoner. Dette fordi «vi» som programmere ikke har definert noe annet. Det fornuftige her ville vært å definere utførelsen av ***begge*** disse **UPDATE**-spørringene som ***én*** transaksjon. Først da er vi sikret en konsistent tilstand etter utførelse av operasjonene, jamfør forklaringen av transaksjonsdefinisjon gitt i kapittel 18.1.

Det må altså sikres at enten ***begge*** eller ***ingen*** av de to **UPDATE**-instruksjonene utføres, og vi vil gjøre dette ved å bruke C#-metodene **BeginTrans** og **Commit** til å definere hva som skal inngå i transaksjonen. I tillegg vil **Rollback**-metoden kalles opp i **catch**-delen av **try-catch-finally**-strukturen, så databasen rulles tilbake til sin siste konsistente tilstand før transaksjonen, dersom noe går galt underveis.

I Figur 18-18 er «click event handleren» til knappen **btnSwapTeams** endret til å inkludere transaksjonhåndtering, der det to **UPDATE**-spørringene defineres om én transaksjon. Kommentarer som forklarer transaksjonselementene som inngår i koden, er lagt til i koden.

```
private void btnSwapTeams_Click(object sender, EventArgs e)
{
    string sqlQuery1, sqlQuery2;
    SqlTransaction transaction = null; //Lager en transaksjonsvariabel
    SqlConnection con = null; //Lager en connection-variabel
    SqlCommand command = null; //Lager en variable for sql-operasjoner mot basen
    //Definerer og lagrer den første spørringen. Flytter medlem 3 til Team 2:
    sqlQuery1 = String.Concat(@"UPDATE TEAMMEMBER
                                SET TEAMID = 2
                                WHERE MemberId = 3");
    //Definerer og lagrer den andre spørringen. Flytter medlem 4 til Team 1:
    sqlQuery2 = String.Concat(@"UPDATE TEAMMEMBER
                                SET TEAMID = 1
                                WHERE MemberId = 4");
    try
    {
        con = new SqlConnection(conProject);
        con.Open();
        transaction = con.BeginTransaction(); //Transaksjonen starter her
        command = new SqlCommand(sqlQuery1, con);
        //assosierer command-variabelen med transaksjonen:
        command.Transaction = transaction;
        command.ExecuteNonQuery();
        command = new SqlCommand(sqlQuery2, con);
        //assosierer command-variabelen med transaksjonen:
        command.Transaction = transaction;
        command.ExecuteNonQuery();
        //Transaksjonen avsluttes og endringer registreres permanent i basen:
        transaction.Commit();
    }
    catch (Exception error)
    {
        MessageBox.Show(error.Message + " : " + error.GetType());
        transaction.Rollback();
        //MessageBox.Show("Transaction was rolledback: " + error.Message);
        //numMembersTeam1++;
        //numMembersTeam2--;
    }
```

```
    finally
    {
        if (con.State == ConnectionState.Open) //Tester om databaseforbindelsen er åpen
        {
            con.Close(); //Stenger databaseforbindelsen dersom den er åpen
        }
    }
}
```

**Figur 18-18:Koden modifisert til å inkludere transaksjonshåndtering.**

Test koden med og uten feil.

Ved kjøring av koden vist i Figur 18-18, skal transaksjonen utføres uten feil. Resultatet skal bli det samme som tidligere vist i tabellen til høyre i Figur 18-12, altså at to av medlemmene har byttet team.

Modifiser så koden så den andre **UPDATE**-spørringen forsøker å flytte medlemmet med **MemberId** 4 til **Team 5** (istedenfor til **Team 1**), så transaksjonen feiler. Endringen er vist i Figur 18-19.

```
    //Definerer og lagrer den andre spørringen. Flytter medlem 4 til Team 5:
    sqlQuery2 = String.Concat(@"UPDATE TEAMMEMBER
                               SET TEAMID = 5
                               WHERE MemberId = 4");
```

**Figur 18-19. Endring utført for at den andre UPDATE-funksjonen i transaksjonen skal feile.**

En feil inntreffer da i den andre **UPDATE**-spørringen. Programmet hopper da til **catch**-delen, der **Rollback**-metoden kalles opp og utføres. Eventuelle endringer transaksjonen har forårsaket i databasen inntil feilen inntraff, rulles da tilbake til den siste konsistente tilstanden før transaksjonen startet.

Kjør koden og sjekk at ingen endringer har blitt utført i **TEAMMEMBER**-tabellen. Resultatet skal bli som tidligere vist i den venstre tabellen i Figur 18-12. Transaksjonshåndteringen har altså sikret at databasen er i en konsistent tilstand, uavhengig av om transaksjonen ble vellykket eller ikke. Dersom feil inntraff, har DBHS benyttet transaksjonsloggen (jf. kapittel 18.2) til å rulle tilbake transaksjonen.

## 18.5. ACID-egenskaper for håndtering av samtidighetsproblemtikk

Når flere transaksjoner utføres samtidig, kan det inntreffe samtidighetsproblematikk (Concurrency Problems). Dette kan skje når ulike transaksjoner jobber mot de samme transaksjonene.

Det er viktig at transaksjoner ikke gis mulighet til å ødelegge for andre transaksjoner. Theo Harder and Andreas Reuter beskrev i en artikkel i 1983 et sett med egenskaper omtalt som **ACID** (Atomocity, Consistency, Isolation and Duarbility) [13].

Det å følge disse **ACID**-egenskapene, skal sikre at transaksjoner utføres korrekt, uten fare for feil i databasen etter utført transaksjon. Nedenfor er **ACID**-egenskapene kort forklart.

**Atomicity**

En transaksjon må utføres i sin helhet eller ikke utføres i det hele tatt. Dette kan i C# sikres med metodene **BeginTransaction**, **Commit** og **Rollback**, som illustrert i kapittel 18.2 – 18.4.

**Consistency**

En transaksjon må føre databasen fra én konsistent (gyldig) tilstand til en annen konsistent tilstand. Dette vil si at alle deler av transaksjonen må utføres i tråd med alle regler som måtte være satt for databasen. DBHS sikrer dette ved gjennom oppfølging av metodikken presentert i kapittel 18.2 – 18.4 (**BeginTransaction**, **Commit** og **Rollback**), samt gjennom å følge opp alle øvrige regler (constraints) knyttet til databasen, som entitets- og referanseintegritet og eventuelle andre restriksjoner.

**Isolation**

Utføres flere transaksjoner samtidig, må disse isoleres fra hverandre. Dette siden en transaksjon under utførelse kan være i en inkonsistent tilstand, som innebærer en risiko for at andre transaksjoner kan komme til å utføre operasjoner mot ugyldige data.

**Durability**

Når en transaksjon er fullført, må dataene lagres på et varig (non-volatile), på for eksempel en disk eller annet permanent lagringsmedium. Dette så man ikke risikerer at data går tapt. Backup av disker etc. er også en viktig del av dette, så ikke for eksempel en diskkrasj fører til at data går varig tapt. Både transaksjonsloggen og andre data må sikres mot slike tilfeller.

I de påfølgende kapitlene ses det på noen kjente samtidighetsproblemer som kan inntreffe dersom transaksjoner ikke isoleres fra hverandre ved samtidig utførelse [14].

1. Tapt oppdatering (The lost update problem)
2. Angret oppdatering (The uncommited data problem) (Dirty read)
3. Inkonsistent oppdatering (The inconsistent data problem) (Incorrect summary)

For å forklare disse problemene tenker vi oss en lignende problemstilling som ble programmert i kapittel 18.4, der medlemmer ble flyttet fra ett team til et annet.

Problemene som nå skal illustreres skal det ikke lages programkode, da dette er noe DBHS tar seg av i bakgrunnen. Det er derfor prinsippene som ligger til grunn, som skal illustreres. Låsing for lesing og/eller skriving fra/til dataelementer en sentral del av denne teknologien.

Som eksempel skal det ses på ulike transaksjoner som kjører samtidig, men der fletting av operasjonene kan skje på forskjellig vis, og der noen av kombinasjonene kan føre til at feil verdier lagres i databasen.

I tillegg til endring av **TeamId**, som ble gjort i kapittel 18.4, tenker vi oss at det nå i tillegg skal lagres et nummer i databasen som viser **antall** team-medlemmer det er i teamet. Dette innebærer at dette tallet må endres hver gang et team-medlem går ut av et team eller kommer inn i et team.

Lagring av **antall** kan betraktes som **redundante** (overflødige) data, da det kan avledes fra eksisterende data gjennom en spørring mot tabellen, men er her brukt for å illustrere samtidighetsproblemetikk.

## 18.6. Tapt oppdatering (The lost update problem)

To transaksjoner kalt **T1** og **T2** skal utføres samtidig.

**T1:** Transaksjon 1 skal flytte et team-medlem fra **Team 2** til **Team 3**.

**T2:** Transaksjon 2 skal samtidig legge til et nytt team-medlem i **Team 2**.

I Figur 18-20 er to transaksjonene **T1** og **T2** stilt opp ved siden av hverandre for å illustrere samtidighet.

| Transaction 1 (T1) | Transaction 2 (T2) |
|---|---|
| ReadDB (team2NumT1) | ReadDB (team2NumT2) |
| team2NumT1-- | team2NumT2++ |
| WriteDB (team2NumT1) | WriteDB (team2NumT2) |
| ReadDB (team3NumT1) | |
| team3NumT1++ | |
| WriteDB (team3NumT1) | |

**Figur 18-20: Illustrasjon av «The lost update problem».**

**Forklaring på utførelsen trinn for trinn:**

Det forutsettes at det i utgangspunktet er tre medlemmer i hvert team.

1. **T1** leser antall team2-medlemmer (3) fra databasen og lagrer verdien i variabelen **team2NumT1**.
2. **T2** leser antall team2-medlemmer (3) fra databasen og lagrer verdien i variabelen **team2NumT2**.
3. **T1** reduserer antall medlemmer i **team2NumT1**-variabelen med 1, så **team2NumT1** blir 2.
4. **T2** øker antall medlemmer i **team2NumT2** med 1, så **team2NumT2** blir 4.
5. **T1** skriver **team2NumT1** (verdien 2) til databasen.
6. **T2** skriver **team2NumT2** (verdien 4) til databasen (som dermed overskriver verdien 2).

7. **T1** leser antall team3-medlemmer (3) fra databasen og lagrer verdien i variabelen **team3NumT1**.
8. **T1** øker antall medlemmer i **team3NumT1** med 1, så **team3NumT1** dermed blir 4.
9. **T1** skriver **team3NumT1** (verdien 4) til databasen.

**Resultater av transaksjonene:**

**T1** har redusert team 2-antallet fra 3 til 2 og økt team 2-antallet fra 3 til 4, hvilket isolert er korrekt, siden ett medlem skulle flyttes fra ett team til et annet.

**T2** har økt **Team 2**-antallet fra 3 til 4 (fordi det ikke leste **T1** sin reduksjon til 2). Denne blir feil, siden transaksjonen ***skulle*** ha lest tallet 2 og så økt dette til 3. Nå skrives isteden 4-tallet til databasen, og antall medlemmer i **Team 2** blir dermed registrert som 4 istedenfor 3.

Dette er et eksempel på at samtidighet kan føre til «**tapt oppdatering**», med feil verdi registrert i databasen som en konsekvens av dette. **Team 2** er oppdatert av **T1** og deretter overskrevet av **T2**.

## 18.7. Angret oppdatering (The uncommitted data problem) (Dirty read)

I eksempelet vist i Figur 18-21, skal de samme transaksjonene som i kapittel 18.6 utføres, men nå med litt annen «fletting» i tid og med en situasjon der én av transaksjonene feiler og må rulles tilbake.

**T1:** Transaksjon 1 skal flytte et team-medlem fra **Team 2** til **Team 3**.

**T2:** Transaksjon 2 skal samtidig legge til et nytt team-medlem i **Team 2**.

| Transaction T1 | Transaction T2 |
|---|---|
| ReadDB (team2NumT1) | |
| team2NumT1 -- | |
| WriteDB (team2NumT1) | |
| | ReadDB (team2NumT2) |
| | team2NumT2++ |
| | WriteDB (team2NumT2) |
| Exception thrown | |
| Rollback (team2NumT1) | |

**Figur 18-21: Illustrasjon av problemet «Tapt oppdatering» ved samtidighet mellom transaksjoner**

1. **T1** leser antall **Team 2**-medlemmer (3) fra databasen og lagrer verdien i variabelen **team2NumT1**.
2. **T1** reduserer antall medlemmer i **team2NumT1**-variabelen med 1, så **team2NumT1** blir 2.
3. **T1** skriver **team2NumT1** (verdien 2) til databasen.
4. **T2** leser antall **Team 2**-medlemmer (2) fra databasen og lagrer verdien i variabelen **team2NumT2**.
5. **T2** øker antall medlemmer i **team2NumT2** med 1, så **team2NumT2** blir 3.
6. **T2** skriver **team2NumT2** (verdien 3) til databasen.
7. **T1** får en **Exception** med **rollback**, fordi en feil har inntruffet. Verdien 3, som ble skrevet til databasen i punkt 6, rulles tilbake, så databaseverdien endres tilbake til 2 (fra 3).

**T1** oppdaterer (endrer tilbake grunnet rollback) en verdi som **T2** allerede har lest. **T2** leser altså feil verdi som følge av den går glipp av en oppdatering som **T2** kommer til å gjøre.

Dette er samtidighetsproblem som kalles «**angret oppdatering**» (eller «**dirty read**»), da det leses data som ikke er klare for permanent lagring/Commit.

## 18.8. Inkonsistent oppdatering (The inconsistent data problem)

I dette eksempelet skal to transaksjoner kalt **T1** og **T2** utføres samtidig, som vist i 18-22.

**T1:** Transaksjon 1 skal flytte et team-medlem fra **Team 2** til **Team 3** og oppdatere **antall** medlemmer.

**T2:** Transaksjon 2 skal summere antall team-medlemmer i **Team 1**, **Team 2** og **Team 3**.

| Transaction T1 | Transaction T2 |
|---|---|
| | ReadDB(team1NumT2) |
| | sum = team1NumT2 |
| ReadDB (team2NumT1) | |
| team2NumT1-- | |
| WriteDB (team2NumT1) | |
| | ReadDB(team2NumT2) |
| | sum := sum + team2NumT2 |
| | ReadDB(team3NumT2) |
| | sum := sum + team3NumT2 |
| ReadDB (team3NumT1) | |
| team3NumT1++ | |
| WriteDB (team3NumT1) | |

**18-22: Eksempel på inkosistent oppdatering, der det opereres mot noen nye og noen gamle data.**

1. **T2** leser antall **Team 1**-medlemmer (3) fra databasen og lagrer verdien i variabelen **team1NumT2**.
2. **T2** tilordner verdien som er lest (3) til en variabel kalt **sum** (**3**).

3. **T1** leser antall **Team 2**-medlemmer (3) fra databasen og lagrer verdien i variabelen **team2NumT2**.
4. **T1** reduserer antall medlemmer i **team2NumT1**-variabelen med 1, så **team2NumT1** blir 2.
5. **T1** skriver **team2NumT1** (verdien 2) til databasen.

6. **T2** leser antall **Team 2**-medlemmer (2) fra databasen og oppdaterer verdien i **sum** til **5** (3 + 2).
7. **T2** leser antall **Team 3**-medlemmer (3) fra databasen og oppdaterer verdien i **sum** til **8** (5 + 3).

8. **T1** leser antall team3-medlemmer (3) fra databasen og lagrer verdien i variabelen **team3NumT1**.
9. **T1** øker antall medlemmer i **team3NumT1** med 1, så **team3NumT1** dermed blir 4.
10. **T1** skriver **team3NumT1** (verdien 4) til databasen.

Det som har skjedd ved fletting av disse to transaksjonene, er at summeringen T2 har gjort, har blitt gjort med noen oppdaterte og noen ikke-oppdaterte data, hvorav navnet «**inkonsistent oppdatering**». T2 får ikke med seg den siste oppdateringen T1 gjør av Team 3-antallet fra 3 til 4, og ender derfor med en sum på 8 istedenfor 9, der i er det korrekte.

Hvert team hadde i utgangspunktet 3 medlemmer, sånn at det var totalt 9 medlemmer. Det eneste **T1** har gjort er å flytte et medlem fra ett team til et annet, sånn at summen av medlemmer fortsatt er 9.

**T2** fikk altså med seg den første oppdateringen **T1** gjorde, men ikke den andre, og dermed ble resultatet i **sum**-variabelen feil (8 istedenfor 9). Flettingen har altså ført til at **T2** har operert med en blanding av oppdaterte og ikke-oppdaterte data, hvorav navnet «**inkonsistent oppdatering**» kommer fra.

## 18.9. Innføring av låser (locks) for å løse samtidighetsproblemene

For å unngå samtidighetsproblemene (Concurrency Problems) illustrert i kapittel 18.6 – 18.8, innføres bruk av låser ved utførelse av transaksjoner. DBHS sørger for låsing og opplåsing av dataelementer og i databasesystemer er det normalt mulig å angi hva slags type låsemekanisme som skal benytte.

I dette kapittelet vises det hvordan låsing løser de nevnte typer samtidighetsproblemer.

**Det benyttes to typer låser:**

1. Leselåser (Read locks / Shared locks)
2. Skrivelåser (Write locks / Exclusive locks)

**Leselås**

Noen transaksjoner trenger bare tilgang til å ***lese*** dataelementer. Hvis en transaksjon ikke skal gjøre noen endringer på dataene, kan det da sette en **leselås** på de aktuelle dataene. Dette innebærer at andre transaksjoner samtidig kan få lov til å lese dataene, men ikke gjøre endringer på dem. Dette gjør at flere transaksjoner kan jobbe mot de samme dataene samtidig.

**Skrivelås**

En transaksjon som har behov for å gjøre ***endringer*** på dataene, kan sette en **skrivelås** på disse. Dette kalles også en **eksklusiv lås**, fordi transaksjonen får en eksklusiv rett til både å lese fra og skrive til dataene som er låst. Andre transaksjoner som trenger tilgang til dataene, må da vente til låsene frigjøres, før de kan gjøre lese- eller skriveoperasjoner mot disse.

Når en transaksjon er ferdig med lese- og/eller skriveoperasjoner mot dataene, må den frigjøre disse ved å låse opp eventuelle låser som er påført dataelementer.

I Figur 18-23 er problemet med «**tapt oppdatering**» løst ved bruk av låser. Når T2 ber om skrivelås på antall medlemmer i Team 2 som ligger lagret i databasen, så har allerede T1 påført en skrivelås på disse dataene. Dette innebærer at T2 må vente med å få tilgang til disse dataene til T1 er ferdig med dem og låser dem opp. T2 forhindres derfor nå fra å lese feil data og begge transaksjonene utføres korrekt og bringer databasen fra én konsistent tilstand til en annen.

| Transaction T1 | Transaction T2 |
|---|---|
| Ask for writelock (W.lock team2 number) | |
| ReadDB (team2NumT1) | Ask for writelock on team2 number (waiting) |
| team2NumT1-- | Ask for writelock on team2 number (waiting) |
| WriteDB (team2NumT1) | Ask for writelock on team2 number (waiting) |
| Unlock (team2NumT1) | Ask for writelock on team2 number (waiting) |
| Ask for writelock (team3 number) | W.lock team2 number: ReadDB (team2NumT2) |
| W.lock team3: ReadDB(team3NumT1) | team2NumT2++ |
| team3NumT1++ | WriteDB (team2NumT2) |
| WriteDB (team3NumT1) | Unlock (team2 number) |
| Unlock (team3 number) | |

**Figur 18-23: Illustrasjon av løsning på "Tapt oppdatering"-problemet ved bruk av låsing.**

De to andre samtidighetsproblemene «angret oppdatering» og inkonsistent analyse» kan løses med samme metodikk. Dette illustreres ikke her, men forsøk gjerne selv å tenke ut hvordan påføring av låser også vil løse disse samtidighetsproblemene.

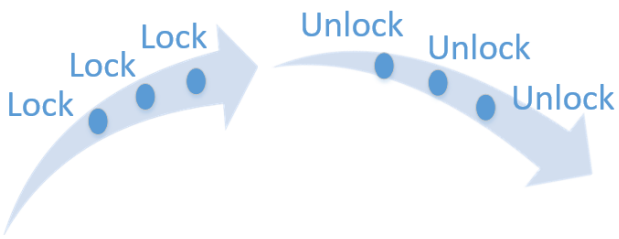
## 18.10. Serialiserbarhet (Serializability)

Selv når det benyttes låser, kan samtidighetsfeil inntreffe. Dette kan eksempelvis skje når oppdatering av data utført i én rekkefølge, fører til et annet resultatet enn når dataene oppdateres i en annen rekkefølge, for eksempel ved bruk av beslutninger (som if-setninger eller tilsvarende).

Dersom to transaksjoner er sikret å oppnå det samme resultat når de flettes (samtidig kjøring), som de ville gjort ved å bli utført en etter en alene, sier man at transaksjonene er **serialiserbare**.

## 18.11. To-fase låsing (Two-phase locking)

To-fase låsing er teknikk som sikrer serialiserbarhet. Prinsippet går ut på at en transaksjon må påføre alle låser den skal påføre, før den første låsen frigjøres. En transaksjon går dermed gjennom to faser, en fase der låser påføres og en annen fase der låser frigjøres, som illustrert i Figur 18-24.



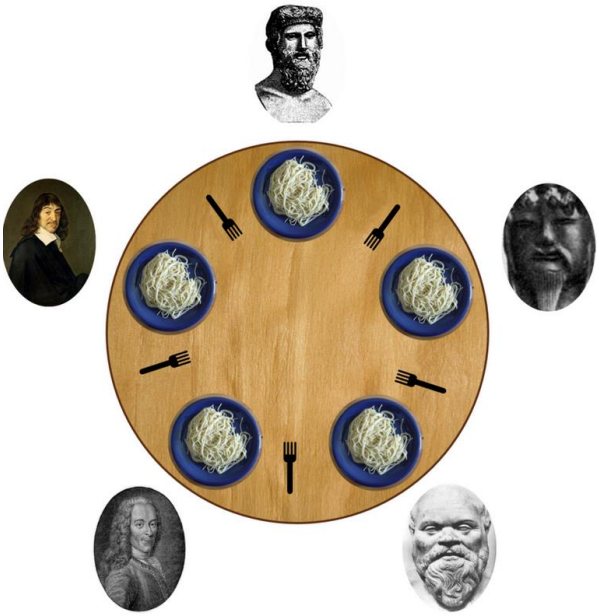

**Figur 18-24: To-fase låsing innebærer en fase med påføring av låser og en fase med frigjøring av låser.**

## 18.12. Vranglås (Deadlock)

Selv om transaksjoner er serialiserbare, kan det fortsatt inntreffe samtidighetsproblemer. Dette kan skje dersom flere transaksjoner har låst et dataelement som de andre trenger tilgang til. Resultatet kan da bli at alle de involverte transaksjonene blir stående og vente på hverandre, som resulterer i det som kalles en vranglås (Deadlock).

Dette fenomenet ble beskrevet av Edsger Dijkstra [15] (1930–2002) i 1965, som del av en eksamensoppgave der han illustrerte samtidighetsproblem og vranglås med det han kalt «The dining philosophers problem» [16].

Illustrasjonen i Figur 18-25 viser fem filosofer som skal spise sammen. Problemet er at alle trenger to gafler for å spise og hver av dem må dermed dele én av gaflene med en av de andre filosofene. Dersom alle derfor skal spise samtidig, og alle for starter med å ta opp sin venstre gaffel, vil alle filosofene måtte vente til en av de andre frigjør én av gaflene. Først da vil en av filosofene kunne få både en høyre- og venstregaffel å spise med.



**Figur 18-25: The dining philosophers problem). (Bilde: Benjamin. D. Esham / Wikimedia Commons)**

Filosof-problemstillingen er naturligvis nokså søkt, men den illustrer situasjonen som inntreffer når ulike transaksjoner havner i en vranglås. Vranglås (deadlock) er altså en situasjon der et antall transaksjoner stopper opp, fordi alle blir stående og vente på frigjøring av en ressurs de alle deler.

Problemstillingen er beskrevet og drøftet i en rekke sammenhenger, og her er referanse til en av de nyere artiklene som beskriver problemet og mulige algoritmiske løsninger [17].

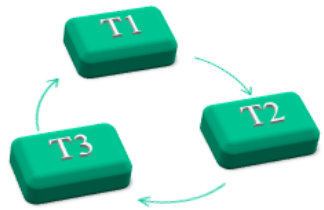
## 18.13. To strategier for å håndtere vranglåser – Oppdage eller forhindre

I dette kapitelet ses det på to ulike strategier for å håndtere vranglåser. Dette er strategier som implementeres i **DBHS**, så det er **DBHS** som tar seg av dette.

**Strategi 1: Detektere og løse opp.**

**DBHS** kan kan konfigureres til å detektere vranglåser og løse dem opp når de oppdages. **DBHS** har da algoritmer som innebærer å bygge opp ventegrafer (Wait-for-graph), som illustrert i Figur 18-26.

Figuren illustrerer hvordan transaksjon **T1** venter på transaksjon **T2**, **T2** venter på **T3** og **T3** venter **T1**. Alle transaksjonene venter altså på hverandre, så det har dermed inntruffet en vranglås.




**Figur 18-26: Illustrasjon av en vranglås som har inntruffet, der tre transaksjoner alle venter på hverandre.**

Strategien **DBHS**-algoritmen bruker når en vranglås oppdages, er prinsipielt å avbryte en av transaksjonene, deretter fullføre de andre og så til slutt fullføre den som ble avbrudt.

**Strategi 2: Forhindre at vranglås inntreffer**

Istedenfor å vente til til en vranglås inntreffer og så løse den opp, er dette en strategi som tar sikte på at vranglåser inntreffer. Dette gjøres ved bruk av tidsstempler (timestamps) på transaksjonene.

I forklaringene som følger, benyttes begrepene yngre og eldre transaksjoner. Alle transaksjonene påføres et tidsstempel som inneholder informasjon om tidspunktet transaksjonen ble opprettet. Dette tidsstempelet beholder transaksjonen i hele sin levetid. Dette innebærer at jo lenger tid det går før transaksjonen blir utført, jo eldre blir den. Man kan altså betrakte transaksjonens alder som øyeblikkstidspunktet minus tidspunktet transaksjonen ble opprettet.

Prinsipielt forhindres vranglåser etter følgende prinsipper.

1. DBHS gir alle transaksjoner et tidsstempel når de starter.
2. Dersom en yngre transaksjon prøver å sette lås på en transaksjon som allerede ***ér*** låst av en eldre transaksjon, avbrytes den nyeste transaksjonen (altså den av dem som sist fikk sitt tidsstempel). Neste gang den nyeste transaksjonen prøver å låse elementet, vil den ha blitt litt eldre.
3. Dersom en eldre transaksjon prøver å sette lås på en transaksjon som allerede ***ér*** låst av en nyere transaksjon, stilles den eldre transaksjonen i kø.
4. Resultatet av punkt 1–3 er at yngre transaksjoner aldri settes i kø (siden de isteden avbrytes), og en vranglås kan dermed aldri inntreffe.

Prinsippene er illustrert i Figur 18-27.


**Figur 18-27: Illustrasjon av prinsipp for å forhindre at vranglåser inntreffer.**

I illustrasjonen til venstre prøver **T1** å låse et dokument som allerede ***er*** låst av **T2**. Siden **T1** (som er opprettet 12:30:31) er eldre enn **T2** (dvs. opprettet før **T2**, som er opprettet 12:30:35), så stilles **T1** i kø. **T1** kan derfor låse elementet når dette blir frigjøres av **T2**.

I figuren til venstre prøver også **T1** å opprette en lås på et element som allerede er låst av **T2**. Denne gangen har de motsatte tidsstempler, hvilket vil si at **T1** er yngre enn **T2** (altså opprettet senere enn **T2**). **T1** blir da ikke satt i kø, men avbrutt, og må prøve igjen senere (og vil da ha blitt eldre).

## 18.14. Optimistisk samtidighetskontroll (Optimistic Concurrency Control)

Dersom det sjeldent inntreffer en situasjon med vranglås, eksempelvis når det fortrinnsvis utføres lese-operasjoner mot dataelementene, er optimistisk samtidighetskontroll en aktuell strategi. Det satses da på at transaksjonene i all hovedsak går bra, sånn at låsing kan begrenses eller unnlates. Transaksjoner kan dermed utføres raskere, enn hva tilfellet er når eksempelvis låser må påføres. Denne strategien benyttes derfor når sannsynligheten for at konflikter skal inntreffe er lav.

Dersom det i sjeldne tilfeller inntreffer konflikter, håndteres disse ved for eksempel å rulle tilbake en transaksjon. Dersom slike tilfeller inntreffer ofte, vil denne strategien bli lite effektiv. Bruk av tids-stempel (timestamp) er én måte å foreta optimistisk samtidighetskontroll på (jf. kapittel 18.13). Da håndteres konflikter først ***når*** de inntreffer.

https://en.wikipedia.org/wiki/Optimistic_concurrency_control

## 18.15. Pessimistisk samtidighetskontroll (Pessimistic Concurrency Control)

Ved pessimistisk samtidighetskontroll (Pessimistic Concurrency Control) er man føre var. Det vil si at det gjøres tiltak for å ***forhindre*** at konflikter inntreffer, istedenfor å måtte håndtere konflikter ***etter*** at de inntreffer. Dette kan eksempelvis gjøres ved bruk av låser og tidsstempler, jf. teknikkene som er forklart i de tidligere kapitlene. Dette for å forhindre at transaksjoner «forstyrrer» andre transaksjoner.

Strategien med å benytte tidsstempler på transaksjoner for å forhåndre at eventuelle vranglåser inntref-fer, jf. forklaring i kapittel 18.13, er også et eksempel på en pessimistisk tilnærming. Denne metoden egner seg derfor når konflikter inntreffer ofte og alternativet er å stadig måtte håndtere disse.

Dersom det eksempelvis sjeldent eller aldri inntreffer vranglåser, er det bedre med en optimistisk tilnær-ming, som eksempelvis oppbygging av ventegrafer for å detektere og håndtere eventuelle vranglåser. Da aksepteres det at vranglåser inntreffer, og de håndteres først da.

http://www.sqlpassion.at/archive/2015/08/31/pessimistic-concurrency-in-sql-server/

# Referanser


[1] Codd, E. F. (1970). A relational model of data for large shared data banks, Communications of the ACM. 13(6), pp. 377-387.

[2] Wikipedia. (2017, 22.11.2017). Edgar F. Codd. Available: https://en.wikipedia.org/wiki/Edgar_F._Codd

[3] IT HISTORY SOCIETY. (2017). Dr. Raymond (Ray) F. Boyce. Available: http://www.ithistory.org/honor-roll/dr-raymond-ray-f-boyce

[4] Wikipedia. (2017). Donald D. Chamberlin. Available: https://en.wikipedia.org/wiki/Donald_D._Chamberlin

[5] Chamberlin, D. D. and Boyce, R. F., "SEQUEL: A Structured English Query Language," in ACM-SIGMOD Workshop on Data Description, Access and Contro, Ann Arbor, Michigan, 1974.

[6] ISO (International Organization for Standardization). (2016, 22.11.2017). ISO/IEC 9075-1:2016 – Information technology -- Database languages -- SQL -- Part 1: Framework (SQL/Framework). Available: https://www.iso.org/standard/63555.html

[7] Microsoft. (2017, 22.01.2017). Transact-SQL Reference (Database Engine). Available: https://docs.microsoft.com/en-us/sql/t-sql/language-reference

[8] erwin. (2017, 22.11.2017). erwin Data Modeler. Available: https://erwin.com/products/data-modeler/

[9] Chen, P. (1976). The Entity–Relationship Model – Toward A Unified View of Data, ACM Transactions on Database Systems. New York: Association for Computing Machinery 1(1),

[10] Kaula, R. (2007) Normalizing with Entity Relationship Diagramming. TThe Data Administration Newsletter (TDAN.com). Available: http://tdan.com/normalizing-with-entity-relationship-diagramming/4583

[11] Microsoft. (2017, 21.11.2017). Transactions (Transact-SQL), . Available: https://docs.microsoft.com/en-us/sql/t-sql/language-elements/transactions-transact-sql

[12] Microsoft. (2017, 22.11.2017). The Transaction Log (SQL Server). Available: https://docs.microsoft.com/en-us/sql/relational-databases/logs/the-transaction-log-sql-server

[13] Haerder, T. and Reuter, A. (1983). Principles of transaction-oriented database recovery, ACM Computing Surveys (CSUR) 15(4), pp. 287-317.

[14] Microsoft Technet. (22.11.2017). Concurrency Effects. Available: https://technet.microsoft.com/en-us/library/ms190805(v=sql.105).aspx

[15] Wikipedia. (2017, 22.11.2017). Edsger W. Dijkstra. Available: https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

[16] Wikipedia. (2017, 22.11.2017). Dining philosophers problem. Available: https://en.wikipedia.org/wiki/Dining_philosophers_problem

[17] Ali, D. Z. and Ali, A. (2016). Analysis of the Dining Philosophers Problem in Concurrency Control between Processes, International Journal of Technology Management & Humanities (IJTMH) 1(4),

# Stikkordregister