IN2090 – Databaser og datamodellering
07 – Datamanipulering

Leif Harald Karlsen
leifhka@ifi.uio.no

Universitetet i Oslo

Komplisert eksempel

Finn kundenavn og productnavn på alle kunder som har bestilt en drikkevare som ikke lenger selges (“discontinued”) [230 rader]

```
SELECT DISTINCT c.company_name, p.product_name
FROM products AS p
     INNER JOIN order_details AS d ON (p.product_id = d.product_id)
     INNER JOIN orders AS o ON (o.order_id = d.order_id)
     INNER JOIN customers AS c ON (c.customer_id = o.customer_id)
     INNER JOIN categories AS g ON (g.category_id = p.category_id)
WHERE p.discontinued = 1 AND
      g.category_name = 'Beverages';
```


Komplisert eksempel med `WITH`

Finn kundenavn og productnavn på alle kunder som har bestilt en drikkevare som ikke lenger selges (“discontinued”) [230 rader]

```
WITH
  beverages AS (
    SELECT p.product_id, p.product_name
    FROM products AS p INNER JOIN categories AS c
         ON (p.category_id = c.category_id)
    WHERE c.category_name = 'Beverages' AND
          p.discontinued = 1
  ),
  company_orders AS (
    SELECT u.company_name, d.product_id
    FROM customers AS u
         INNER JOIN orders AS o ON (u.customer_id = o.company_id)
         INNER JOIN order_details AS d (o.order_id = d.order_id))
SELECT DISTINCT c.company_name, p.product_name
FROM beverages AS b INNER JOIN company_orders AS o
     ON (b.product_id = o.product_id);
```


Flere eksempler: Kombinere aggregater

Finn antall producter som har en pris større eller lik 100 og alle som har en pris mindre enn 100

```
SELECT
  (SELECT count(*) FROM products WHERE unit_price >= 100) AS expensive,
  (SELECT count(*) FROM products WHERE unit_price < 100) AS cheap
```

Typer SQL-spørringer

Som sagt tidligere, SQL kan gjøre mye mer enn bare uthenting av data.
Det første ordet i en spørring sier hva spørringen gjør:

SELECT henter informasjon (svarer på et spørsmål)
CREATE lager noe (f.eks. en ny tabell)
INSERT setter inn rader i en tabell
UPDATE oppdaterer data i en tabell
DELETE sletter rader fra en tabell
DROP sletter en hel ting (f.eks. en hel tabell)

SQLs ulike fuksjoner

De ulike spørringene er egentlig deler av ulike under-språk av SQL. Vi har

- SDL (Storage Definition Language): 3-skjemaarkitekturens fysiske lag
- DDL (Data Definition Language): 3-skjemaarkitekturens konseptuelle lag
- VDL (View Definition Language): 3-skjemaarkitekturens presentasjonslag
- DML (Data Manipulation Language): innlegging, endring og sletting av data
- DQL (Data Query Language): spørrespråk
- DCL (Data Control Language): integritet og sikkerhet

Lage ting

- For å lage tabeller, brukere, skjemaer, osv. bruker vi `CREATE`-kommandoer
- For å lage et skjema gjør vi

```
CREATE SCHEMA northwind;
```

- SQL-kommandoen for å lage tabeller har formen:

```
CREATE TABLE <tabellnavn> ( <kolonner> );
```

- hvor `<tabellnavn>` er et tabellnavn (potensielt prefikset med et skjemanavn)
- og `<kolonner>` er kolonne-deklareringer
- En kolonne-deklarering inneholder
  - et kolonnenavn, og
  - en type,
  - og en liste med skranker (constraints)

`CREATE`-eksempel

- For å lage `Student`-tabellen kan vi kjøre

```
CREATE TABLE Student (
    SID int,
    StdName text,
    StdBirthdate date
);
```

- Nå vil følgende tomme tabell finnes i databasen:

Students

SID (int)	StdName (text)	StdBirthdate (date)

Skranker: NOT NULL

- I mange tilfeller ønsker vi å ikke tillate NULL-verdier i en kolonne
- For eksempel dersom verdien er påkrevd for at dataene skal gi mening
  - F.eks. vi vil aldri legge inn en student dersom vi ikke vet navnet på studenten
- eller verdien er nødvendig for at programmene som bruker databasen skal fungere riktig
- Vi kan da legge til en NOT NULL-skranke til kolonnen
- For eksempel:

```
CREATE TABLE Student (
    SID int,
    StdName text NOT NULL,
    StdBirthdate date
);
```


Skranker: UNIQUE

- Dersom vi ønsker at en kolonne aldri skal gjenta en verdi (altså inneholde duplikater)
- kan vi bruke UNIQUE-skranken
- For eksempel, student-IDen SID er unik
- Så for at databasen skal håndheve dette kan vi lage tabellen slik:

```
CREATE TABLE Student (
    SID int UNIQUE,
    StdName text NOT NULL,
    StdBirthdate date
);
```


Skranker: PRIMARY KEY

- I tillegg til å være unik, så må SID-verdien aldri være ukjent, ettersom det er primærnøkkelen i tabellen
- Så vi burde derfor ha både UNIQUE og NOT NULL, altså:

```
CREATE TABLE Student (
    SID int UNIQUE NOT NULL,
    StdName text NOT NULL,
    StdBirthdate date
);
```

- Men, det finnes også en egen skranke for dette, nemlig PRIMARY KEY som inneholder UNIQUE NOT NULL. Så,

```
CREATE TABLE Student (
    SID int PRIMARY KEY,
    StdName text NOT NULL,
    StdBirthdate date
);
```

  er ekvivalent som over
- Merk, kan kun ha én PRIMARY KEY per tabell, må bruke UNIQUE NOT NULL dersom vi har flere kandidatnøkler

Alternaiv syntaks for skranker

- Man kan også skrive skrankene til slutt, slik:

```
CREATE TABLE Student (
    SID int,
    StdName text NOT NULL,
    StdBirthdate date,
    CONSTRAINT sid_pk PRIMARY KEY (SID)
);
```

- Nå har skrankene navn (sid_pk, name_nn)
- Denne syntaksen er nødvendig om vi ønsker å ha skranker over flere kolonner
- F.eks. om kombinasjonen av StdName og StdBirthdate alltid er unik:

```
CREATE TABLE Student (
    SID int,
    StdName text NOT NULL,
    StdBirthdate date,
    CONSTRAINT sid_pk PRIMARY KEY (SID),
    CONSTRAINT name_bd_un UNIQUE (StdName, StdBirthdate)
);
```

Skranker: REFERENCES

- Det er vanlig i relasjonelle databaser at en kolonne refererer til en annen
- Fremmednøkler er eksempler på dette
- I slike tilfeller ønsker vi å begrense de tillatte verdiene i kolonnen til kun de som finnes i den den refererer til
- Dette kan gjøres med REFERENCES-skranken
- F.eks. for å lage TakesCourse-tabellen, kan vi gjøre følgende:

```
CREATE TABLE TakesCourse (
    SID int REFERENCES Student (SID),
    CID int REFERENCES Course (CID),
    Semester text
);
```

- Nå vil man kun kunne legge inn SID-verdier som allerede finnes i Students(SID) og kun CID-verdier som allerede er i Courses(CID)

Sette inn data

- For å sette inn data i en tabell bruker vi INSERT-kommandoen
- INSERT brukes på følgende måte:

```
INSERT INTO <tabell>
VALUES (<rad>),
       (<rad>),
       ...,
       (<rad>);
```

- Så, for å sette inn radene
  - (0, 'Anna Consuma', '1978-10-09'), og
  - (1, 'Peter Young', '2009-03-01')
- inn i Students, kan vi gjøre:

```
INSERT INTO Students
VALUES (0, 'Anna Consuma', '1978-10-09'),
       (1, 'Peter Young', '2009-03-01');
```


Andre måter å sette inn data

- Vi kan bruke resultatet fra en SELECT-spørring i stedet for VALUES
- For ekesmpel:

```
CREATE TABLE Students2018 (
    SID int PRIMARY KEY,
    StdName text NOT NULL
);

INSERT INTO Students2018
SELECT S.SID, S.StdName
  FROM Students AS S INNER JOIN TakesCourse AS T
       ON (S.SID = T.SID)
 WHERE T.Semester LIKE '%18';
```


Ny tabell basert på SELECT direkte

- Vi kan også kombinere de to kommandoene på forige slide slik:

```
CREATE TABLE Students2018 AS
SELECT S.SID, S.StdName
  FROM Students AS S INNER JOIN TakesCourse AS T
       ON (S.SID = T.SID)
 WHERE T.Semester LIKE '%18';
```

- Dette gir samme data, men merk at vi nå ikke har skrankene PRIMARY KEY og NOT NULL
- Disse må da legges til etterpå

Default-verdier

- Vi kan gi en kolonne en standard/default verdi
- Denne blir brukt dersom vi ikke oppgir en verdi for kolonnen
- For ekesmpel:

```
CREATE TABLE personer (
    pid int PRIMARY KEY,
    navn text NOT NULL,
    nationalitet text DEFAULT 'norge'
);

INSERT INTO personer
VALUES (1, 'carl', 'UK');

INSERT INTO personer(pid, navn) --eksplisitte kolonner
VALUES (2, 'kari');
```

vil gi

personer

pid	navn	nationalitet
1	Carl	UK
2	Kari	norge

SERIAL

- For primærnøkler som bare er heltall, så kan vi bruke `SERIAL`
- Dette gjør at databasen automatisk genererer unike heltall for hver rad
- Så med

```
CREATE TABLE Student (
    SID SERIAL PRIMARY KEY,   -- merk ingen type
    StdName text NOT NULL,
    StdBirthdate date
);

INSERT INTO Students(StdName, StdBirthdate) --eksplisitte kolonner
VALUES ('Anna Consuma', '1978-10-09'),
       ('Peter Young', '2009-03-01'),
       ('Anna Consuma', '1978-10-09');
```

vil vi få

Students

SID	StdName	StdBirthdate
1	Anna Consuma	1978-10-09
2	Peter Young	2009-03-01
3	Anna Consuma	1978-10-09

- Merk at man må være sikker på at radene nå faktisk representerer unike ting!

Hvor kommer data fra? (1)

Man skriver som oftest ikke `INSERT`-spørringer direkte

Den vanligste måten å få data inn i en database på er via programmer som eksekverer `INSERT`-spørringer (Se senere i kurset), f.eks.:

- data generert av simuleringer, analyse, osv.
- data skrevet av brukere via en nettside, brukergrensesnitt, osv.
- data fra sensorer (f.eks. værdata), nettsider (f.eks. aksjedata, klikk), osv.

Hvor kommer data fra? (2)

- Man kan også lese data direkte fra filer (f.eks. regneark eller CSV)
- I PostgreSQL har man `COPY`-kommandoen får å laste inn data fra CSV
- Følgende laster inn innholdet fra CSVen `~/documents/people.csv` (med separator `','` og null-verdi `''`) inn i tabellen `Persons`:

```
COPY persons
FROM '~/documents/people.csv' DELIMITER ',' NULL AS '';
```

- Merk, PostgreSQL krever at man er superuser for å lese filer av sikkerhetsgrunner
- Men man kan alltid lese fra Standard Input (`stdin`), f.eks. ved å eksekvere følgende (i Bash):

```
$ cat persons.csv | psql <flag> -c
    "COPY persons FROM stdin DELIMITER ',' NULL AS ''"
```

(hvor `flag` er de vanlige flaggene man bruker for innlogging til databasen)

- I Postgres finnes det også en egen `\copy`-kommando i `psql`

Eksempel: Jentenavn – datainnlasting

- La oss finne ut hvilket jentenavn som økte mest i popularitet fra 2017 til 2018
- SSB har mange dataset, bla.: `https://www.ssb.no/statbank/table/10501/` (velg alle navn, og årene 2017 og 2018)
- Lager så tabell for denne dataen:

```
CREATE TABLE jentenavn(navn text, y17 int, y18 int);
```

- Og laster så inn dataene (må endre enkoding og bytte alle `..` med `.`)

```
$ cat Persons.csv | psql <flag> -c
    "COPY jentenavn FROM stdin DELIMITER ';' NULL AS '.'"
```


Eksempel: Jentenavn – finne svaret

Kan så finne svaret vårt:

```
SELECT *
FROM jentenavn
WHERE y18-y17 = (SELECT max(y18-y17)
                 FROM jentenavn)
```

Svaret er altså “Ada”

Eksempel: Jentenavn – alternativ løsning

```
ALTER TABLE jentenavn
ADD COLUMN diff int;

UPDATE jentenavn
SET diff = y18 - y17

SELECT *
FROM jentenavn
WHERE diff = (SELECT max(diff)
              FROM jentenavn)
```


Eksempler på skrankeovertredelser (violations)

Som sagt tidliere, man har ikke lov til å overtre databaseskjemaet, så hvis vi har

```
CREATE TABLE Students (
    SID int PRIMARY KEY,
    StdName text NOT NULL,
    StdBirthdate date
);
```

så vil

- ```
  INSERT INTO Students
  VALUES (0, 'Anna Consuma', '1978-10-09', 1);
  ```

  gi `ERROR: INSERT has more expressions than target columns`
- ```
  INSERT INTO Students
  VALUES ('zero', 'Anna Consuma', '1978-10-09');
  ```

  gir `ERROR: invalid input syntax for integer: "zero"`
- ```
  INSERT INTO Students
  VALUES (0, NULL, '1978-10-09');
  ```

  gir `ERROR: null value in column "stdname"violates not-null constraint`
  ```

Eksempler på skrankeovertredelser

Og gitt:

Students

SID	StdName	StdBirthdate
0	Anna Consuma	1978-10-09
1	Anna Consuma	1978-10-09
2	Peter Young	2009-03-01
3	Carla Smith	1986-06-14
4	Sam Penny	NULL

Vil

```
INSERT INTO Students
VALUES (0, 'Peter Smith', '1938-11-11');
```

gi `ERROR: duplicate key value violates unique constraint "students_pkey"`

Slette ting

- For å slette ting (tabeller, skjemaer, brukere, osv.) fra databasen bruker vi `DROP`
- For å slette en tabell gjør vi `DROP TABLE <tablename>;`, f.eks.:

```
DROP TABLE Students;
```

- Tilsvarende for skjemaer, f.eks. `DROP SCHEMA northwind;`
- Av og til avhenger ting vi ønsker å slette på andre ting (f.eks. en tabell er avhengig av skjemaet den er i eller tabellene den refererer til)
- Vi kan ikke slette ting som andre ting avhenger av, uten å også slette disse
- For å slette en ting og alt som avhenger av den tingen kan vi bruke `CASCADE`
- Så for å slette `northwind`-skjemaet og alle tabeller som det inneholder kan vi gjøre

```
DROP SCHEMA northwind CASCADE;
```


Slette data

- For å slette rader fra en tabell bruker vi `DELETE`:

```
DELETE
  FROM <tabellnavn>
 WHERE <betingelse>
```

- Så sletting av alle studenter født etter `1990-01-01` gjøres slik:

```
DELETE
  FROM Students
 WHERE StdBirthdate > '1990-01-01'
```


Oppdatere ting

- For å oppdatere skjemaelementer bruker vi `ALTER`
- Mens data oppdateres med `UPDATE`
- Vi kan f.eks. gjøre følgende:

```
ALTER TABLE Students
RENAME TO UIOStudents;
```

  for å omdøpe `Students`-tabellen til `UIOStudents`
- Eller

```
ALTER TABLE Courses
ADD COLUMN Teacher text;
```

  for å legge til en kolonne `Teacher` med type `text` til `Courses`-tabellen
- Alt i skjemaet kan endres med `ALTER`, se PostgreSQL-siden[1] for en oversikt

[1] `https://www.postgresql.org/docs/current/sql-altertable.html`

Legge til skranker i ettertid

- Vi kan også legge til skranker etter at en tabell er laget
- Dette gjøres med kombinasjonen av `ALTER TABLE` og `ADD CONSTRAINT`
- For eksempel:

```
ALTER TABLE courses
ADD CONSTRAINT cid_pk PRIMARY KEY (cid);
```


Oppdatere data

- `UPDATE` lar oss oppdatere verdiene i en tabell:

```
UPDATE <tabellnavn>
   SET <oppdateringer>
 WHERE <betingelse>
```

  hvor `<oppdateringer>` er en liste med oppdateringer som blir eksekvert for hver rad som gjør `<betingelse>` sann
- For eksempel:

```
UPDATE Students
   SET StdBirthdate = '1987-10-03'
 WHERE StdName = 'Sam Penny'
```

  oppdaterer fødselsdatoen til studenten Sam Penny til `'1987-10-03'`
- Mens

```
UPDATE northwind.products
   SET price = price * 1.1
 WHERE quantityperunit LIKE '%bottles%'
```

  øker prisen med 10% på alle produkter som selges i flasker

Views

- Merk at vi nesten aldri er interessert i dataene slik de er lagret
- Vi må nesten alltid joine tabeller, filtrere vekk rader, projisere vekk kolonner, osv. for å få interessant data ut
- F.eks. i Filmdatabasen må man joine 3 tabeller for å finne ut hvilken skuespiller som spiller i hvilken film
- Hvorfor er det slik?
- Jo, fordi vi ønsker å representere dataene på slik måte at:
  - vi aldri repeterer data (gjør det enkelt å vedlikeholde, mer effektivt, osv.)
  - dataene kan brukes på mange forskjellige måter
- Vi bruker så spørringer for å få ut interessant data
- Av og til vil en bestemt spørring bli eksekvert veldig ofte
- Det er da upraktisk å måtte skrive den ut hver gang
- I slike tilfeller kan man lage et `VIEW`

Å lage views

- Et view er egentlig bare en navngitt spørring, og lages slik:

```
CREATE VIEW StudentTakesCourse ( StdName text, CourseName text )
AS
  SELECT S.StdName, C.CourseName
  FROM Students AS S,
       Courses AS C,
       TakesCourse AS T
   WHERE S.SID = T.SID AND C.CID = T.CID
```

- Et view kan så brukes som om det var en vanlig tabell
- Men blir beregnet på nytt hver gang den brukes
- Så et view tar ikke opp noe plass og trengs ikke oppdateres
- Så,

```
SELECT *
FROM StudentTakesCourse AS s
WHERE s.StdName = 'Anna Consuma'
```

$\Rightarrow$

```
SELECT *
FROM (
    SELECT S.StdName, C.CourseName
      FROM Students AS S, Courses AS C,
           TakesCourse AS T
     WHERE S.SID = T.SID AND
           C.CID = T.CID) AS s
WHERE StdName = 'Anna Consuma'
```

Views som abstraksjoner

- Views kan også brukes for å bygge lag med abstraksjoner over tabellene
- F.eks. gitt følgende tabeller:

students

sid	name	mail
1	Anna Consuma	anna@mail.no
2	Peter Young	py@uio.no
3	Mary Smith	smith@ifi.no

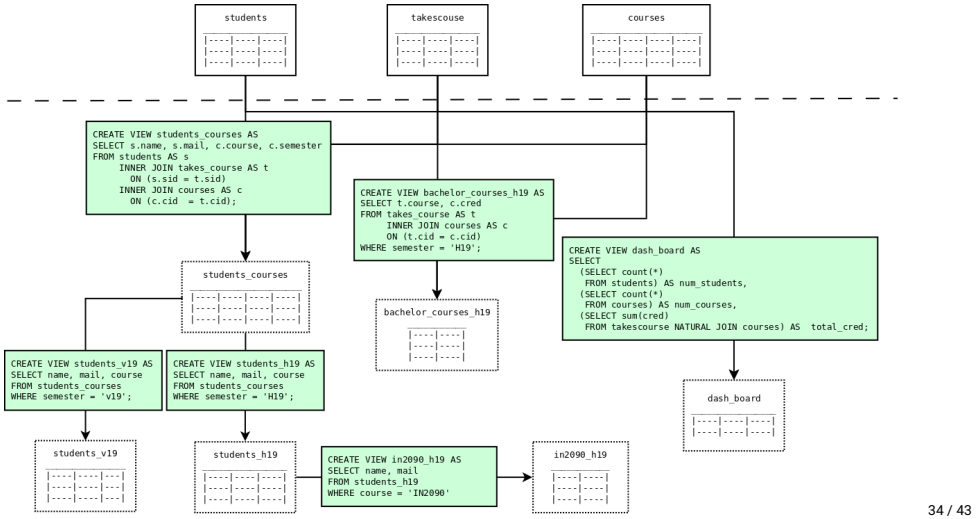
takescourses

sid	cid	semester
1	1	h18
1	2	v18
2	3	v18
3	2	v19
3	1	h19

courses

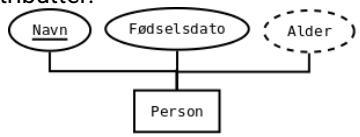
cid	name	cred	lvl
1	Databases	10	B
2	Programming 101	5	B
3	Advanced SQL	10	M

Views som abstraksjoner



Views for utledbare verdier

- I ER har vi utledbare attributter:



- Med views kan vi introdusere disse attributtene igjen
- Uten at vi trenger å lagre dem, holde dem oppdatert, osv.

person

navn	fødselsdato
Anna Consuma	1989-08-17
Peter Young	1991-02-29
Mary Smith	1993-01-01

person_alder

navn	fødselsdato	alder
Anna Consuma	1989-08-17	30
Peter Young	1991-02-29	28
Mary Smith	1993-01-01	26

```
CREATE VIEW person_alder AS
SELECT navn,
       fødselsdato,
       EXTRACT(year FROM age(current_date,fødselsdato)) AS alder
FROM person
```


Materialiserte Views

- Dersom et view brukes veldig ofte kan det lønne seg å materialisere det
- Et materialisert view lagres som en vanlig tabell på disk
- De er derfor like effektive å kjøre spørringer mot som en vanlig tabell
- Lages slik:

```
CREATE MATERIALIZED VIEW person_alder AS
SELECT navn,
       fødselsdato,
       EXTRACT(year FROM age(current_date,fødselsdato)) AS alder
FROM person
```

- Men, den kan enkelt oppdateres når de tabellene den avhenger av oppdateres
- Dette skjer derimot ikke automatisk, man må kjøre følgende for å oppdatere det:

```
REFRESH MATERIALIZED VIEW person_alder;
```

SQL-scripts

- Når man lager en database vil man vanligvis lage et script som inneholder alle SQL-kommandoene som lager skjemaene, tabellene, viewsene, osv.
- Man kan så heller eksekvere dette scriptet, fremfor å kjøre hver spørring manuelt
- Følgende er et eksempel-script som lager `Students`-databasen

```
CREATE SCHEMA uio;
CREATE TABLE uio.students (sid SERIAL PRIMARY KEY, stdname text NOT NULL, stduiorthdate date);
CREATE TABLE uio.courses (cid SERIAL PRIMARY KEY, coursename text NOT NULL, credits int);
CREATE TABLE uio.takescourse (cid int REFERENCES uio.courses(cid),
                              sid int REFERENCES uio.students(sid), semester text);
CREATE VIEW studenttakescourse ( stdname text, coursename text )
AS SELECT s.stdname, s.coursename
   FROM uio.students AS s INNER JOIN uio.takescourse AS t ON (t.sid = t.sid)
        INNER JOIN uio.courses AS c ON (t.cid = c.cid);
INSERT INTO uio.students(stdname, stduiorthdate)
VALUES ('Anna Consuma', '1978-10-09'), ('Anna Consuma', '1978-10-09'),
       ('Peter Young', '2009-03-01'), ('Carla Smith', '1986-06-14');
INSERT INTO uio.courses(coursename, credits)
VALUES ('Data Management', 6), ('Finance', 10);
INSERT INTO uio.takescourse(sid, cid, semester)
VALUES (0,0,'A18'), (1,1,'S17'), (2,1,'S18'),
       (2,0,'S18'), (3,0,'A18');
```


Dump

- Et databasesystem kan også lage et script som gjenskaper dens database(r)
- I PostgreSQL gjøres dette med et eget program `pg_dump` på følgende måte:

```
pg_dump [flag] db > fil
```

  hvor `[flag]` er de vanlige tilkobligsflaggene, `db` er navnet på databasen man vil dumpe, og `fil` er navnet på filen man vil skrive til.
- Andre databasesystemer har tilsvarende programmer
- Dette gjør det enkelt å duplisere eller dele databaser

SQL-scripts: Trygge kommandoer

- Dersom man forsøker å opprette en tabell som allerede finnes eller slette en tabell som ikke finnes så feiler kommandoen
- Dersom denne kommandoen er en del av en transaksjon, så feiler hele transaksjonen
- Dette kan hindres ved å bruke `If EXISTS` og `IF NOT EXISTS` i kommandoene
- For eksempel:

```
CREATE TABLE IF NOT EXISTS persons(name text, born date); -- Lager ny tabell
CREATE TABLE IF NOT EXISTS persons(name text, born date); -- Gir ingen error/lykkes
CREATE TABLE persons(name text, born date);               -- Gir ERROR og feiler
DROP TABLE IF EXISTS persons;                             -- Sletter tabellen
DROP TABLE IF EXISTS persons;                             -- Gir ingen error/lykkes
DROP TABLE persons;                                       -- Gir error, og feiler
```

- F.eks. nyttig dersom man oppdaterer scriptet som har generert en database
- Kan da kjøre scriptet for å kun få utført oppdateringene

SQL-scripts: Meta-kommandoer

- I et SQL-script har man også en del kommandoer som ikke er en del av SQL-språket
- F.eks. printe en beskjed, lage og gi verdier til variable, be om input fra en bruker, osv.
- Disse kommandoene har forskjellig syntaks fra RDBMS til RDBMS
- I PostgreSQL kan man printe en kommando ved å bruke `\echo`, f.eks.

```
\echo 'This is a message'
```

  og brukes for å gi informasjon mens scriptet kjører (progresjon ol.)
- Dersom en konstant verdi brukes mye i et script kan man gi den et navn med `\set`, f.eks.

```
\set val 42
INSERT INTO meaning_of_life VALUES (:val);
```

- Merk kolonet foran navnet når verdien brukes
- Disse kan også brukes i `psql` direkte

Transaksjoner

- Når man oppdaterer databasen og noe går galt underveis øsnker man ofte at ingen av oppdateringene skal ha skjedd
- F.eks. kan man få delvis lagde tabeller, delvis insatt data, osv.
- For eksempel, se for dere følgende bank-overføring:

```
UPDATE balances
SET balance = balance - 100
WHERE id = 1;

UPDATE balances
SET balance = balance + 100
WHERE id = 2;
```

- Dersom den første oppdateringen feiler (f.eks. fordi `balance < 100` men vi har en skranke `balances >= 0`) vil vi ikke at den andre skal utføres
- Det samme gjelder dersom vi får en feil mitt i et SQL-script
- Vi pakker derfor inn oppdateringer som skal utføres som en "enhet" i transaksjoner

Transaksjoner – Syntaks

Transaksjoner omsluttes av `BEGIN` og `COMMIT` slik:

```
BEGIN;

UPDATE balances
SET balance = balance - 100
WHERE id = 1;

UPDATE balances
SET balance = balance + 100
WHERE id = 2;

COMMIT;
```


ACID

For at transaksjoner skal fungere som forventet, tilfredstiller de fire kriterier:

- **A**tomicity – Alle kommandoene i en transaksjon ansees som en enhet, og enten skal alle kommandoer lykkes, eller så skal alle kommandoer feile (feiler én så feiler alle)
- **C**oncistency – Dersom en transaksjon lykkes skal databasen ende opp i en konsistent tilstand (altså ingen skranker skal være brutt)
- **I**solation – Transaksjoner skal kunne kjøres i parallell, men resultatet skal da være likt som om transaksjonene ble kjørt sekvensielt
- **D**urability – Etter at en transaksjon lykkes og har utført endringer på databasen, skal disse endringene alltid være utført (f.eks. dersom systemet restartes skal databasen fortsatt ha de samme endringene utført)