UNIVERSITETET
I OSLO

Ekstramateriale: Eksempel på PostgreSQL 8.4 og SQL:1999 (ikke pensum 2011)

UDTer

Distinkt UDT i Postgres: **create domain**

```
create domain persnrtype as char(5)
check (value ~ '[0-9][0-9][0-9][0-9][0-9]');
```

Strukturert UDT: Postgres-terminologi er 'composite types'

```
create type fører as (
  fdato date,
  persnr persnrtype,
  navn varchar(50));
```

Typede relasjoner

For hver **create table** lages automatisk en tilhørende composite type. Alle relasjoner er derfor 'typede' i Postgres.

```
create table fører (
  fdato date,
  persnr persnrtype,
  navn varchar(50),
  primary key (fdato, persnr)
);
```

Bak kulissene skapes **type** fører med utseende som på forrige lysark. Merk at primær- og kandidatnøkler deklareres sammen med tabeller, ikke typer

Brukerdefinerte funksjoner I

Eksempel på brukerdefinert funksjon: Funksjon som tester om et fødselsnummer er lovlig. Tar som input et tuppel av type `fører` og returnerer en boolsk verdi. Innmaten er i dette tilfellet gitt ved SQL-setninger.

```
create function lovligfnr (in fører) returns boolean as $$
select
 (i4 = 11 - k1 or k1 = 0 and i4 = 0) and (i5 = 11 - k2 or k2 = 0 and i5 = 0) as lovlig
from
 (select i4, i5,
  (((3 * d1) + (7 * d2) + (6 * m1) + (1 * m2) + (8 * a1) + (9 * a2) + (4 * i1) + (5 * i2) + (2 * i3)) % 11) as k1,
  (((5 * d1) + (4 * d2) + (3 * m1) + (2 * m2) + (7 * a1) + (6 * a2) + (5 * i1) + (4 * i2) + (3 * i3) + (2 * i4)) % 11) as k2
  from (<plukk ut sifrene d1 d2 m1 m2 a1 a2 fra fører.fdato
          og i1 i2 i3 i4 i5 fra fører.persnr – se neste lysark for kode>) as testsiffer;
$$ language SQL;
```

Brukerdefinerte funksjoner II

Utplukking av sifrene i fødselsnummeret. $1 er funksjonsargumentet (av type `fører`). `n % m` er `n` modulo `m` (dvs. restverdien når `n` heltallsdivideres på `m`).

```
select
   (cast(extract(day from $1.fdato) as integer) / 10) as d1,
   (cast(extract(day from $1.fdato) as integer) % 10) as d2,
   (cast(extract(month from $1.fdato) as integer) / 10) as m1,
   (cast(extract(month from $1.fdato) as integer) % 10) as m2,
   ((cast(extract(year from $1.fdato) as integer) % 100) / 10) as a1,
   (cast(extract(year from $1.fdato) as integer) % 10) as a2,
   cast(substring($1.persnr, 1, 1) as integer) as i1,
   cast(substring($1.persnr, 2, 1) as integer) as i2,
   cast(substring($1.persnr, 3, 1) as integer) as i3,
   cast(substring($1.persnr, 4, 1) as integer) as i4,
   cast(substring($1.persnr, 5, 1) as integer) as i5
```

Brukerdefinerte funksjoner III

Eksempel på eksekvering av brukerdefinerte funksjoner:

```
select lovligfnr(f), f from fører f;
```

f i select-klausulen gjør at tuplene i fører vises som kommaseparerte lister under kolonnenavnet f. Alternativt kan man som vanlig hente ut de enkelte komponentene i fører til hver sin kolonne ved * eller ved å nevne attributtene eksplisitt:

```
select lovligfnr(f), * from fører f;
select lovligfnr(f), f.fdato, f.persnr, f.navn from fører f;
```

Subtyper og sen binding I

Postgres har ikke subtyper, ikke metoder og – som vi skal se – ingen måte å styre sen binding til rett metode/funksjonsinstans slik typehierarkier i objektorienterte systemer gir muligheten til. Vi kan benytte vanlige brukerdefinerte funksjoner og overlasting av disse til å oppnå noe av det samme, men ikke uten noen heftige triks.

Det nærmeste vi kommer subtyper i Postgres, er relasjonsarv. I stedet for typehierarkiet av typene `kjøretøy`, `a_kjøretøy` og `f_kjøretøy` (s. 36), kan vi danne et relasjonshierarki med tre relasjoner `kjøretøy`, `a_kjøretøy` og `f_kjøretøy`.

Kjøretøydata, grunnlagstyper

Registreringsnummer er to bokstaver og fem sifre:

```
create domain regnrtype as char(7)
check (value ~ '[A-Z][A-Z][0-9][0-9][0-9][0-9][0-9]');
```

Chassisnummer begynner med tre bokstaver (korrekt oppbygging av chassisnummer utover dette sjekkes ikke):

```
create domain vintype as char(17)
check (value ~ '^[A-Z][A-Z][A-Z]');
```

Kjøretøydata, hovedrelasjon

```
create table kjøretøy (
  regnr regnrtype primary key,
  vin vintype unique not null,
  eier fører);
```

Postgres har ikke referansetyper. Å bruke en composite type (her: `fører`) som typen til et argument gir ikke samme virkning – men er her gjort for illustrasjonens skyld. (Sammenlikn med definisjonen av typen `kjøretøy` s. 34.) Tupler av typen `fører` kan nå legges inn som verdi til kolonnen `eier` (brudd på 1NF). Hvis denne informasjonen også skal inn i relasjonen `fører`, må den legges inn eksplisitt.

Relasjonsarv

```
create table a_kjøretøy (
  primary key (regnr),
  unique (vin)) inherits (kjøretøy);
create table f_kjøretøy (
  primary key (regnr),
  unique (vin)) inherits (kjøretøy);
```

a_kjøretøy og f_kjøretøy arver attributtene fra kjøretøy. I dette tilfellet har subrelasjonene ingen attributter utover de arvede. Tupler som legges inn i subrelasjonene, er synlige via select på kjøretøy.

I PostgreSQL 8.4 arves check og not null-skranker, men ikke primary key, unique, references etc. Derfor må slike skranker gjentas i subrelasjonene.

Composite types og insert

Tupler i brukerdefinerte typer bygges enklest med **row**:

```
insert into kjøretøy
values ('DK52477', 'WVWZZZ1GZVW012561',
        row('1990-01-31', '32771', 'Hans Aas'));

insert into a_kjøretøy
values ('ND44787', 'WVWZZZ1GZVW012562',
        row('1987-12-12', '32614', 'Nina Lie'));

insert into f_kjøretøy
values ('ND23995', 'VWVZZZ1HZVW012563',
        row('1948-11-30', '34374', 'Atle Mo'));
```

Composite types og select

Aksessering av innmaten til et tuppel skjer ved dotnotasjon. Legg merke til parentesene i select-setningen under, de er nødvendige for at kompilatoren skal skjønne at k er en relasjon og ikke det som heter 'skjema' i Postgres-sjargong.

```
select (k.eier).fdato from kjøretøy k;
```

Aksessering av innmaten til et tuppel returnert av en funksjon:

```
create function finneier (in kjøretøy) returns fører as
$$ select $1.eier;$$ language SQL;
select (finneier(k)).navn from kjøretøy k;
```

Overlasting av brukerdefinerte funksjoner

```
create function årsavgift (in kjøretøy) returns integer as
$$ select 0 as avgift;$$ language SQL;
```

Overlasting av funksjonen årsavgift:

```
create function årsavgift (in a_kjøretøy) returns integer as
$$ select 2790 as avgift;$$ language SQL;

create function årsavgift (in f_kjøretøy) returns integer as
$$ select 395 as avgift;$$ language SQL;
```

Rett instans velges ved runtime basert på argumentet.

Subtyper og sen binding II

Vi kan nå få tilgang til alle tuplene i kjøretøy og dens subrelasjoner via queries mot hovedrelasjonen kjøretøy, men i motsetning til relasjonen register s. 34, vil de tuplene vi får ut, alle tolkes som å være av typen kjøretøy – og denne typen er *ikke* relatert til typene a_kjøretøy og f_kjøretøy.

Problemet er da at det ikke er noen måte å gjennomløpe tuplene i kjøretøy og omdefinere dem "on-the-fly" til å tilhøre den typen de egentlig er tildelt ved sin subrelasjons-tilhørighet; tuplene som aksesseres via select på moderrelasjonen, er låst til moderrelasjonens type selv om de rent faktisk er lagret i en subrelasjon.

Subtyper og sen binding III

Eksempel: Anta at vi ønsker å beregne summen av årsavgiftene for de kjøretøyene hvor registreringsnummeret begynner med ND. I det tidligere eksempelet (se f.eks. lysark 34-36) kunne vi ha gjort som følger:

```
select sum(avgift) as sumavgift
from (select r.årsavgift() as avgift
      from register r
      where r.regnr like 'ND%') as a;
```

register inneholder kjøretøy av alle typer (i typehierarkiet kjøretøy definert s. 34-36). Systemet velger den instansen av årsavgift som passer best med subtypen til hvert enkelt utvalgt tuppel.

Subtyper og sen binding IV

I Postgres skulle man derfor ønsket å kunne skrive noe åla

```
select sum(avgift) as sumavgift
from (select årsavgift(k) as avgift
      from kjøretøy k
      where k.regnr like 'ND%') as a;
```

Denne vil imidlertid tolke alle relevante tupler som å være av typen kjøretøy og beregne den tilhørende årsavgiften for hver til 0. Det er såvidt vi vet, ingen måte vi kan omdefinere k i årsavgift(k) til å tilhøre typen til den subrelasjonen der tuppelet er hjemmehørende.

Subtyper og sen binding V

Vi kunne selvfølgelig gjort dette eksplisitt slik:

```
select sum(avgift) as sumavgift
from ((select årsavgift(k) as avgift
        from only kjøretøy k where k.regnr like 'ND%')     union
       (select årsavgift(k) as avgift
        from only a_kjøretøy k where k.regnr like 'ND%') union
       (select årsavgift(k) as avgift
        from only f_kjøretøy k where k.regnr like 'ND%') as u;
```

men da må man vite på det tidspunktet når spørringen skrives, hvilke subrelasjoner som fins eller vil komme til å finnes en gang i fremtiden, og en viktig dimensjon av subtyping og sen binding slik dette er kjent fra den objektorienterte verdenen, går tapt. (**only** begrenser resultattuplene til de som tilhører nøyaktig den relasjonen som er navngitt, og utelukker derfor tupler fra dens subrelasjoner.)

Subtyper og sen binding VI

Vi kan få tak i navnet på den subrelasjonen et tuppel stammer fra, på følgende måte:

```
select p.relname as subrel, k.regnr as reg
from kjøretøy k, pg_class p
where k.regnr like 'ND%' and k.tableoid = p.oid;
```

Alle relasjoner har et "skjult" attributt tableoid. tableoid returnerer en id som indikerer hvilken subrelasjon et tuppel stammer fra. Vi slår opp subrelasjonens faktiske navn i systemrelasjonen pg_class.

Subtyper og sen binding VII

Det vi skulle ønsket oss, var etter å ha fått tak i de aktuelle subrelasjonsnavnene som på forrige side, å kunne benytte hvert av dem i ytterligere select-setninger av formen

```
"...
   select årsavgift(k) as avgift
   from only subrel k where k.regnr = reg
..."
```

Da ville rett variant av årsavgift blitt valgt.

Problemet er at subrel slik den fremkommer i queryen forrige side, returnerer en *verdi* (av typen name, som er en SQL-intern datatype), og ikke et *relasjonsnavn*. Setninger av formen over vil altså ikke parse.

Subtyper og sen binding VIII

Måten å likevel få til dette på, er å benytte mekanismer i Postgres embedded SQL eller noen andre av de prosedyrale mekanismene i Postgres.

Eksempelet viser i det videre hvordan dette kan gjøres ved hjelp av Postgres stored procedures, dvs. Postgres-språket PL/pgSQL. Ideen er som følger: Hent ut relasjonsnavnene som verdier slik det blir gjort side 69, og konstruer tekststrenger på bakgrunn av disse der tekststrengene beskriver SQL-setninger som minner om den på forrige side. Disse tekststrengene kan så eksekveres med kommandoen **execute** som er en del av PL/pgSQL-vokabularet.

Subtyper og sen binding IX

```
create language plpgsql; -- Bare hvis Postgres ikke finner plpgsql
create function sumårsavgiftND() returns integer as $$
declare
  totsum integer := 0;
  delsum integer := 0;
  subrel pg_class.relname%TYPE;
begin
  <innmaten av funksjonen; se neste lysark>
end;$$ language plpgsql;
```

pg_class.relname%TYPE returnerer typen til attributtet relname i pg_class.

Nå kan queryen skrives slik:

```
select sumårsavgiftND();
```

Subtyper og sen binding X

```
for subrel in
  select distinct p.relname
  from kjøretøy k, pg_class p
  where k.regnr like 'ND%' and k.tableoid = p.oid
loop
  execute 'select sum(avgift) as sumavgift '
          || 'from (select årsavgift(k) as avgift from only '
          ||          quote_ident(subrel)
          ||          ' k where k.regnr like '
          ||          quote_literal('ND%')
          ||          ') as sk'
  into delsum;
  totsum := totsum + delsum;
end loop;
return totsum;
```

quote_ident() og quote_literal() sikrer at henholdsvis verdien av subrel og tegnene "'ND%'" innlemmes på rett format i tekststrengen som er argument til **execute**.