IN2090 – Databaser og datamodellering

01 – Introduksjon og motivasjon: Databaser

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Hvorfor bruke databaser?



Motivasjon

Hvorfor bruke databaser?

Hvorfor ikke bare bruke f.eks. Python-lister eller Java Collections?

```
handlekurv = ["gulrot", "ost", "juice", "melk"]
```

1. Dataenes levetid
   - Variabler blir lagret i RAM (Random Access Memory)
   - Dette minnet blir slettet hver gang maskinen slås av
   - Vil ofte bevare data igjennom omstart
2. Skalerbar lagringsplass
   - 1 GB harddisk-minne mye billigere enn 1 GB med RAM
3. Separere data fra kode
   - Pythons data er kun tilgjengelig fra Pythons kjøretid
   - Vil ofte la dataene være tilgjengelig for flere programmer

Alle disse problemene er løst av filsystemet!

Motivasjon

Hvorfor bruke databaser?

Så hvorfor ikke bare bruke filer?

Python + Filer

```
import csv
import os

filea = "a.csv"
fileb = "b.csv"
temp = "temp.csv"
source1 = csv.reader(open(filea,"r"),delimiter=",")
source2 = csv.reader(open(fileb,"r"),delimiter=",")

source2_dict = {}
for row in source2:
    source2_dict[row[0]] = row[1]

with open(temp, "w") as fout:
    csvwriter = csv.writer(fout, delimiter=delim)
    for row in source1:
        if row[1] in source2_dict:
            row[3] = source2_dict[row[1]]
        csvwriter.writerow(row)
os.rename(temp, filea)
```

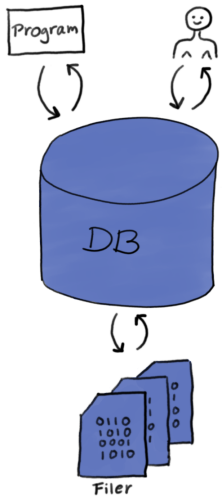
SQL + Database

```
UPDATE a
   SET c4=b.c2
  FROM b
 WHERE a.c2 = b.c1;
```

Motivasjon

Hvorfor bruke databaser?

Så hvorfor ikke bare bruke filer?



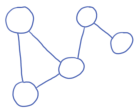
Databasen fungerer som et abstraksjonslag over filsystemet

- Enklere å søke i og manipulere data
- Enklere å spesifisere strukturen på dataene
- Effektivitet og skalerbarhet

Databaser

- En database er en samling data på et bestemt format
- Ulike typer databaser som fokuserer på lagring og håndtering av ulike typer data
- **Dokument-databaser:** Fokuserer på dokumenter og effektive søk i store mengder tekst
- **Key-value stores:** Fokuserer på nøkkel-verdi-par (JSON, Dictionaries, HashMaps, osv.)
- **Grafdatabaser:** Fokuserer på grafer og effektivt finne stier mellom noder i en graf
- **Relasjonelle databaser:** Fokuserer på data i tabell-form (mengder av tupler)
- I dette kurset skal vi kun jobbe med relasjonelle databaser







Relasjonelle databaser: Forenklet beskrivelse

- En relasjonell database er en samling tabeller
- En tabell er også kalt en relasjon
- En tabell har
  - et navn,
  - en samling kolonner
  - og en samling rader (som er dataene)
- En kolonne har
  - et navn,
  - og en type

Customers

CustomerID (int)	Name (text)	Birthdate (date)	NrProducts (int)
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

Relasjonelle databaser = Regneark?

Så, relasjonelle databaser er egentlig bare regneark?

Nei, i motsetning til regneark har relasjonelle databaser:

- en rigid struktur
- spørrespråk for uthenting og manipulering av data
- programmatiske grensesnitt for interaskjon med databasen (fra f.eks. Python eller Java)
- systemer for sikkerhet og kontroll av hvem som har tilgang til dataene
- systemer som sikrer integritet av dataene
- støtte for langt større og mer kompliserte datamengder

Databasesystemer



- Egentlig er en database bare en mengde data (ikke et system/program)
- Et databasesystem (DBMS) er
  et system som lar brukere definere, lage, vedlikeholde og kontrollere tillgangen til databasen.
- Et relasjonelt databasesystem (RDBMS) er
  et databasesystem over relasjonelle databaser.
- Bruker ofte ordet “database” for både dataene, programmet, og kombinasjonen

Hvorfor Relasjonelle Databaser?

- Relasjonelle databaser er den desidert mest brukte typen database
- Nesten all data kan (naturlig) representeres som tabeller
- Naturlig format å jobbe med
- Enkelt å presist spesifisere strukturen til dataene (metadata)
- Denne rigide strukturen tillater svært effektiv uthenting, manipulering og oppdatering av data
- Gir også mange ulike typer sikkerhet

Når bør man ikke bruke relasjonelle databaser?

Finnes også tilfeller når man ikke bør bruke relasjonelle databaser, f.eks. når:

- dataene ikke har noen klar struktur (f.eks. ren tekst, video, lyd)
  - Bruk dokument-databaser
- man er interessert i hvilke relasjoner gitte individer har (*"hva er relasjonen mellom Ola og Kari?"*) fremfor hvilke individer som har hvilke gitte relasjoner (*"hvem er mor til Ola?"*)
  - Bruk grafdatabaser
- man alltid må lese store, men kjente mengder data inn i minne ved hver bruk (f.eks. konfigurasjonsfiler, grafikk)
  - Bruk vanlige filer

Spørrespråk: SQL

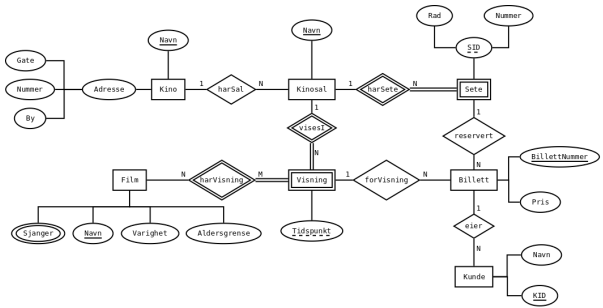
- Et spørrespråk et et språk for å formulere spørringer (spørsmål) til en database
- De fleste spørrespråk er *deklarative*, man uttrykker *hva* man vil finne fremfor *hvordan* det skal finnes
- SQL er det mest brukte spørrespråket og det vi skal lære i dette kurset
- SQL har også funksjonalitet for manipulering (oppdatere, sette inn, slette, osv.) av data og metadata
- SQL kan enten skrives og kjøres direkte av mennesker, eller bli generert og kjørt av programmer (f.eks. Java eller Python)

```
SELECT age
  FROM person
 WHERE name = 'Kari';
```

SQL-spørring som finner alderen til personen med navn Kari.

Takk for nå!

Neste video handler om datamodellering.



IN2090 – Databaser og datamodellering

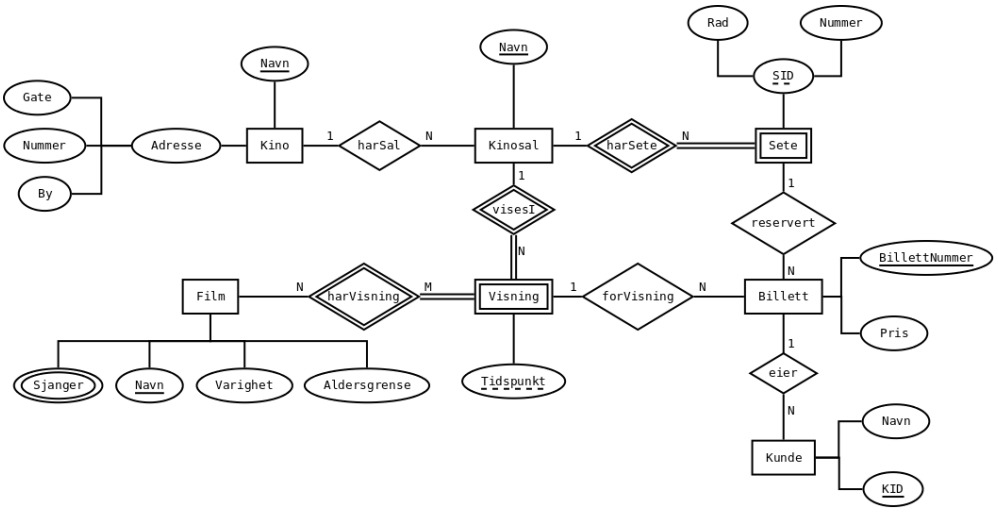
01 – Introduksjon og motivasjon: Modellering

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Hvorfor datamodellering?



Data vs. informasjon

- Innholdet i en database er *data*
- F.eks. i en kolonne som heter `Vekt` kan man ha verdien `142.3`
- Dette er en bit data, og sier oss egentlig ingenting!
- Må vite hvordan verdien skal tolkes for at det skal gi oss noe *informasjon*
- F.eks. hvilken måleenhet er brukt, hva er det vekten på, betegner det faktisk vekt eller maks vekt, osv.
- Informasjon er data pluss regler for hvordan data skal tolkes

Heis	Vekt	…
⋮	⋮	⋮
3	142.3	…
⋮	⋮	⋮

Heis nr. 3 veier 142.3kg?

3 heiser har maks kapasitet 142.3 kg?

Heis nr. 3 har maks kapasitet 142.3 tonn?

Heiser kan ta 3 mennesker som hver veier 142.3 kg?

Fra domene til data



- En database inneholder kun dataene fra et domene
- Så vi må oversette informasjon fra domene til data
- Mange måter å gjøre dette på, noen gode, andre dårlige
- Datamodellering hjelper oss med å finne de gode representasjonene

Hvorfor datamodellering?

Kompliserte domener: Entiteter

De fleste domener er veldig komplekse!

Universitet:	**Nettbutikk:**	**Sykehus:**	**Skatt:**
studenter	produkter	pasienter	personer
ansatte	kunder	ansatte	lønnstrinn
kurs	bestillinger	medisiner	relasjoner
foreninger	leverandører	sykdommer	frynsegoder
programmer	lagere	behandlinger	bedrifter
bygninger	kategorier	rutiner	organisasjoner
rom	kampanjer	krav	sykemeldinger
aktiviteter	regioner	rom	familiære forhold
forskningsgrupper	relatert	operasjoner	formue
karakterer	kvitteringer	inventar	arv
prosjekter	dokumenter	vakt	fagforeninger
⋮	⋮	⋮	⋮

Hvorfor datamodellering?

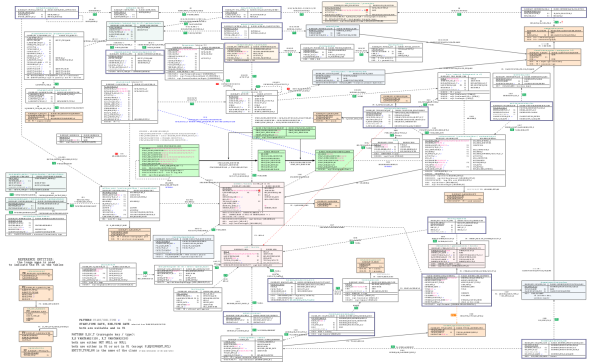
Kompliserte domener: Relasjoner

Disse *entitenene* har mange relasjoner mellom seg

Universitet:	**Nettbutikk:**	**Sykehus:**	**Skatt:**
ansatt ved	bestilt av	får behandling	er datter av
rom bestilt av	har kategori	har kometanse til	tjener
veileder	koster	har vakt	medlem av
har karakter i	er med i kampanje	har utstyr	ansatt i
ledes av	antall på lager	trinn i rutine	har lønnstrinn
⋮	⋮	⋮	⋮

Hvorfor datamodellering?
Kompliserte databaser

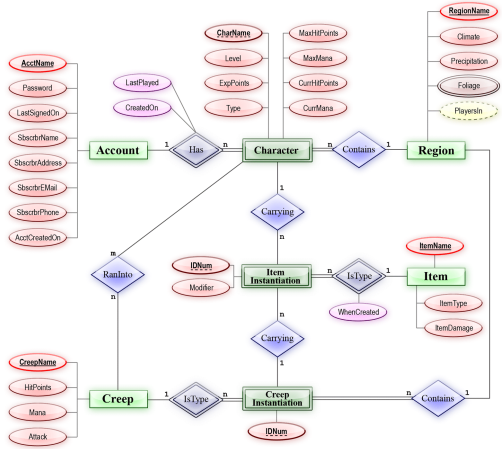
- Så databasen blir da veldig kompleks
  - Mange tabeller (f.eks. > 100)
  - Mange kolonner (f.eks. > 20)
  - Mange rader (millioner)
- Må da ha en god metode for å designe skjemaet til databasen
- Altså, hvilke tabeller og kolonner man skal ha, og hvordan tabellene er relatert



Komplisert databaseskjema

Datamodellering

- Et modelleringsspråk brukes for å lage en modell av et domene
- En modell er en forenkling av virkeligheten som beskriver kun de tingene vi er interessert i
- Denne modellen blir så oversatt til et godt databaseskjema
- Finnes mange modelleringsspråk:
  - UML
  - ORM
  - ER
  - OWL
- Vi skal bruke ER



ER-diagram av karakterer for et spill[1]

[1] https://commons.wikimedia.org/wiki/File:ER_Diagram_MMORPG.png

Takk for nå!

Neste video gir et eksempel på hvordan databaser og datamodellering brukes sammen.





IN2090 – Databaser og datamodellering

01 – Introduksjon og motivasjon: Eksempel

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Eksempelanvendelse

Beskrivelse

- La oss si at vi skal lage et system for kinoer
- Systemet skal bestå av en samling programmer som tilsammen skal kunne:
  - La ansatte legge inn info om saler, filmer, visninger, priser, osv.
  - La kunder bestille billetter, se program, osv.
  - Lage rapporter over billettsalg ol. for kinoledelsen
  - Tillate analyse over dataene



Interessedomene

Eksempelanvendelse

Fra domene til modell (1)

1. Bestemme hvilke ting fra domene vi er interessert i:

- Kino
- Filmer
- Kinosaler
- Billetter
- Visninger
- Priser
- Program
- …



2. Bestemme hva som er sant og interessant i vårt tilfelle

- Kan det være mer enn én sal per kino?
- Kan én billett brukes til å se mer enn én visning?
- Er det alltid én billett per person, eller finnes det gruppebilletter?
- Er det faste plasser på billetten?
- …

Eksempelanvendelse

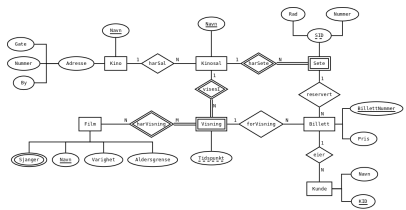
Fra domene til modell (2)

2. Bestemme hva som er sant og interessant i vårt tilfelle

- Kan det være mer enn én sal per kino?
- Kan én billett brukes til å se mer enn én visning?
- Er det alltid én billett per person, eller finnes det gruppebilletter?
- Er det faste plasser på billetten?
- …

⇒

3. Lage modell i henhold til disse faktaene

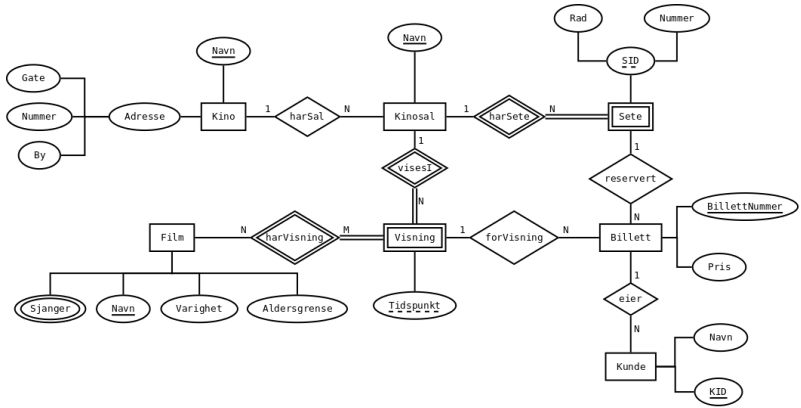


ER-diagram som modellerer domene vårt

Eksempelanvendelse

Fra domene til modell (3)

3. Lage modell i henhold til disse faktaene

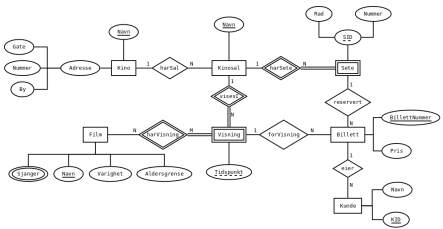


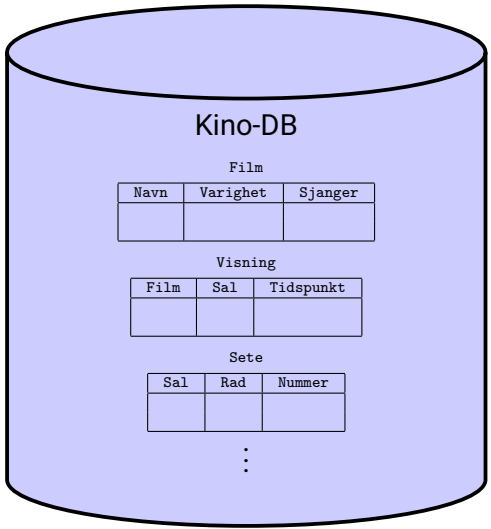
ER-diagram som modellerer domene vårt

Eksempelanvendelse

Fra modell til databaseskjema

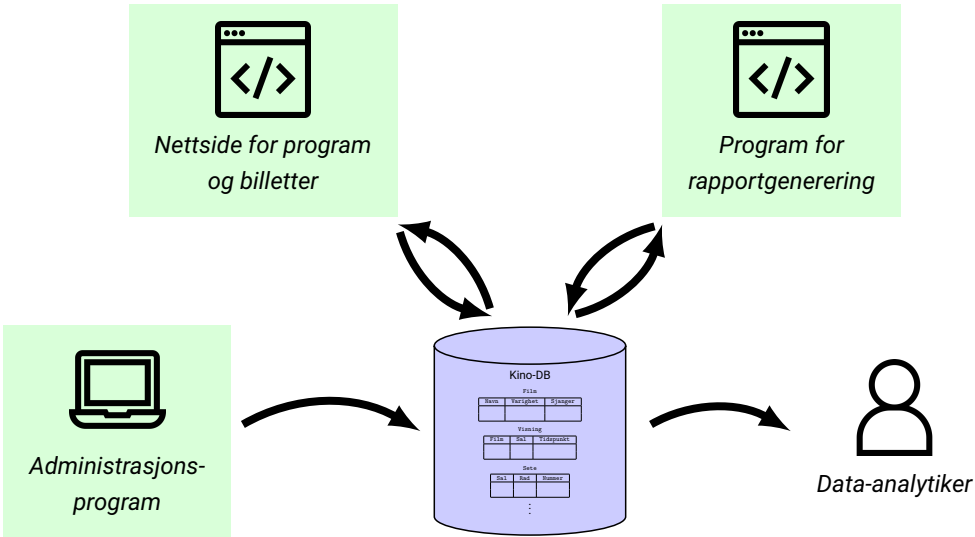
4. Lage databaseskjema i henhold til modell





Eksempelanvendelse

Fra databaseskjema til bruk



Takk for nå!

Neste uke vil vi lære om den relasjonelle modellen





IN2090 – Databaser og datamodellering

01 – Introduksjon og oversikt over emnet

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Oversikt over kurset

- To hovedkategorier: Modellering og SQL
- Tirsdager: Forhåndsinnspilte videoer (dekker pensum)
- Torsdager: Fysisk forelesning med gjennomgang av eksempler og spørsmål
- Forventes at dere ser/deltar på begge

Oversikt over kurset

Uke	Tema	Dato	Foreleser	Innleveringer
1	Introduksjon og motivasjon	Tors 22.08	Leif Harald	
2	*Modellering:* Relasjonsmodellen	Tors 29.08	Leif Harald	
3	*Modellering:* ER-modellering 1	Tors 05.09	Leif Harald	
4	*Modellering:* ER-modellering 2	Tors 12.09	Leif Harald	
5	*DB:* Grunnleggende SQL	Tors 19.09	Leif Harald	
6	*DB:* Datamanipulering, skranker og views	Tors 26.09	Leif Harald	1. Modellering
7	*Modellering:* Normalformer	Tors 03.09	Arild Waaler	
8	*Modellering:* Tapsfri dekomposisjon	Tors 10.10	Arild Waaler	2. Enkel SQL
9	*DB:* Aggregering og sortering	Tors 17.10	Arild Waaler	
10	*DB:* Ytre joins og mengde operatorer	Tors 24.10	Arild Waaler	3. Normalformer
11	*DB:* Programmering med databaser	Tors 31.10	Arild Waaler	
12	*DB:* Sikkerhet i databaser	Tors 07.11	Arild Waaler	4. Avansert SQL
13	*DB:* Indekser og spørreprosessering	Tors 14.11	Arild Waaler	
14	Repetisjon	Tors 21.11	Arild Waaler	5. Programmering med SQL

Mer om timeplan og obliger:

`https://www.uio.no/studier/emner/matnat/ifi/IN2090/h24/index.html`

Forelesere



Leif Harald Karlsen

leifhka@ifi.uio.no



Arild Waaler

arild@ifi.uio.no

Gruppelærere og rettere

Gruppelærere:

- Jamila Mehmandarova
- Julia Lundeby
- Ole-Sander Halla Johansen
- Lise Katarina Grønland
- Viljar Drevland Hardersen
- William Henrik Behn

Rettere:

- Mehmet Capa
- Nora Elise Haugen
- Sapnil Aditya Ayman

Andre innvolverte



Egor Kostylev

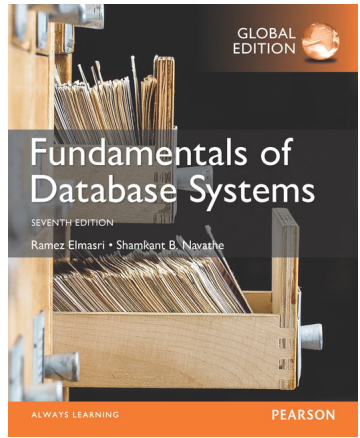
egork@ifi.uio.no

Gruppetimer

- Én-til-én hjelp med ukesoppgaver og innleveringer
- Gruppelærerne velger selv struktur på gruppetimene
- 6 grupper
- Det er lov å bytte gruppe, men spør gruppelæreren først
- Gruppetimenes tema er alltid forelesningstemaet for uken før!
- Første gruppetime er neste uke

Pensum

- Videoene og foilene til kurset
  - Publisert på timeplanen på semestersiden
  - Foiler/screencast fra torsdagsforelesningene blir også publisert
- Ukesoppgavene og innleveringsoppgavene
  - Gjennomgåes på gruppetimene
- Boken *Fundamentals of Database Systems*
  - Tykk bok, men skal bare bruke deler av den
  - Se timeplanen for hva som er pensum
  - Gjenbrukes i andre kurs (IN3020)



Ukesoppgaver

- Publiseres i slutten av uken[1]
- Gjennomgås på gruppetimen uken etter publisering
- Løsningsforslag publiseres i slutten av uken den gjennomgås

[1] `https://www.uio.no/studier/emner/matnat/ifi/IN2090/h24/ukesoppgaver/`

Obliger og innleveringsoppgaver

- Fem innleveringsoppgaver med tilbakmelding
- Første av disse er obligatorisk
  - Fokus er på læring, ikke på løsning
  - Sterkt anbefalt å gjøre alle!
  - Essensiell del av undervisningen
  - Frihet under ansvar
- Publiseres torsdager, frist på fredager[2]
- Leveres via Devilry
- To forsøk på obligen
- De andre kan man be om tilbakemelding på et andre forsøk
- Dersom du blir syk el., kan man be om utsatt frist

[2] `https://www.uio.no/studier/emner/matnat/ifi/IN2090/h24/innleveringer/index.html`

Eksamen

- Eksamen: 13. desember (4 timer, Silurveien)
- Bokstavkarakter (A-F)
- Alle skriftlige hjelpemidler er tillatt

Programvare

- RDBMS: PostgreSQL 16[2]
  - Avansert relasjonell databasesystem
  - Åpen kildekode og støtter de fleste platformer
  - Kan f.eks. bruke `psql`[4] eller Edbit[5]
- Modelleringsvertøy: Dia[3] eller `app.diagrams.net`
  - Både Dia og `app.diagrams.net` er generelle diagram-verktøy
  - Begge har åpen kildekode og støtter de fleste platformer (Linux, Mac, Windows)
  - Trenger IKKE bruke dette om man ikke ønsker, kan tegne på papir eller bruke andre programmer!







[2] `https://www.postgresql.org/`
[3] `http://dia-installer.de/download/index.html`
[4] `https://www.postgresql.org/docs/16/app-psql.html`
[5] `https://gitlab.com/leifhka/edbit`

Databaser

- IMDB: Internet Movie DataBase
  - Filmdatabasen
  - Alle filmer, serier, episoder, skuespillere, roller, osv. frem til 2011
  - Stor database (2.4G)
  - >1 mil. filmer/serier/episoder, >10 mil. roller, >1.7 mil. personer, osv.
- Northwind
  - Eksempel-database laget av Microsoft Research for opplæring i databaser
  - Database over produkter, bestillinger, kunder, osv.
  - Liten database, men realistisk oppsett
- Personlig database
  - Alle har hver sin egne database
  - Du har full tilgang (kan lage, slette, oppdatere data, osv.)
- Satt opp av USiT for dere
- Mer info om tilkobling osv. kommer senere

UiODiscourse

- Brukes for diskusjon, spørsmål og kommentarer rundt pensum, oppgaver, obliger, osv.
- Bruker UiO-brukernavn og passord for innlogging
- Støtter:
  - Publisering med bruker og anonymt
  - Markering av riktig svar
  - Math-mode og code-blocks (SQL)
  - Kategorier og tags
  - osv.
- Guide ligger her: `https://www.mn.uio.no/english/services/it/discourse-uio/astro-discourse-instructions-(students).pdf`

Vår UiODiscourse

`https://astro-discourse.uio.no/c/in2090-24h`

UiODiscourse: Råd og regler

- Gjerne hjelp hverandre, svar på hverandres spørsmål, osv.
- Vær hyggelig! :)
- Sjekk om spørsmålet du har allerede er postet

Motivasjon

Eksempler på hva dere skal lære

Databaser og SQL

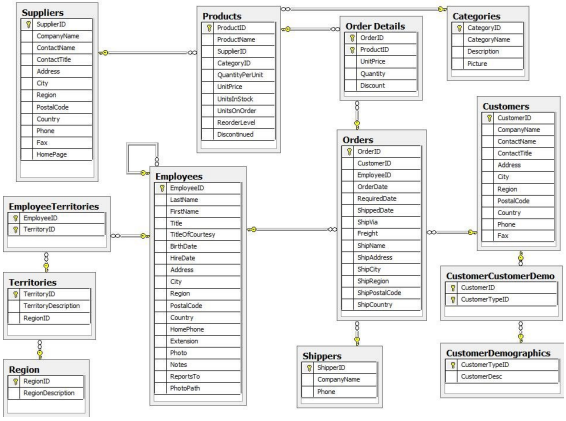
Finn topp 3 rangerte action-filmer med Harrison Ford som skuespiller.

```
SELECT f.title
FROM person AS p
  INNER JOIN filmparticipation AS fp USING (personid)
  INNER JOIN filmgenre AS fg USING (filmid)
  INNER JOIN filmrating AS fr USING (filmid)
  INNER JOIN film AS f USING (filmid)
WHERE p.firstname = 'Harrison' AND
  p.lastname = 'Ford' AND
  fp.parttype = 'cast' AND
  fg.genre = 'Action'
ORDER BY fr.rank DESC
LIMIT 3;
```

Eksempler på hva dere skal lære

Databaser og SQL

Northwind-databasen:



Eksempel-oppgave:

Finn navn og total regning for hver kunde som har kjøpt fler enn 1000 varer, sortert alfabetisk etter firmanavn.

```sql
SELECT c.company_name,
  sum(d.unit_price * d.quantity) AS bill
FROM customers AS c
  INNER JOIN orders AS o USING (customer_id)
  INNER JOIN order_details AS d USING (order_id)
GROUP BY c.customer_id, c.company_name
HAVING sum(d.quantity) > 1000
ORDER BY c.company_name;
```

Mer avansert eksempel

Vi vil lage en spørring som finner ut:

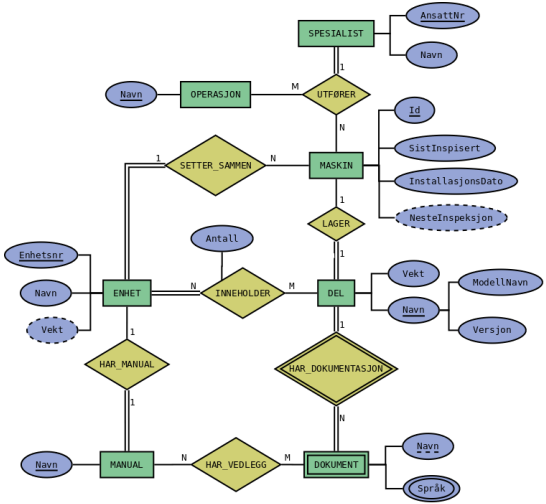
- hvilke produkter vi kan anbefale en kunde å kjøpe,
- basert på hva kunden har kjøpt,
- og hva andre kunder som har kjøpt det samme har kjøpt.

Mer avansert eksempel

```
WITH
  bought AS ( -- Relaterer kunde-IDer til produkt-IDene til det de har kjøpt
    SELECT DISTINCT c.customer_id, d.product_id -- Vil ikke ha duplikater!
    FROM customers AS c
         INNER JOIN orders USING (customer_id)
         INNER JOIN order_details AS d USING (order_id)
  ),
  correspondences AS ( -- Relaterer par av produkter til antallet ganger disse er kjøpt av samme kunde
    SELECT b1.product_id AS prod1, b2.product_id AS prod2, count(*) AS correspondence
    FROM bought AS b1
         INNER JOIN bought b2 USING (customer_id)
    WHERE b1.product_id != b2.product_id -- Fjern par hvor produktene er like
    GROUP BY b1.product_id, b2.product_id -- Gruppér på par av produkter
    HAVING count(*) > 18 -- Antall korrespondanser bør være litt høyt
  ),
  reccomend AS ( -- Relaterer kunde-IDer til anbefalte produkters IDer
    SELECT DISTINCT b.customer_id, c.prod2 AS product_id -- Vil ikke ha duplikater!
    FROM correspondences AS c
         INNER JOIN bought AS b
         ON (b.product_id = c.prod1)
    WHERE NOT c.prod2 IN       -- Fjern produkter som kunden allerede har kjøpt
            (SELECT product_id
             FROM bought AS bi
             WHERE b.customer_id = bi.customer_id)
   )
-- Til slutt finn navn på både kunde og produkt og aggreger produktnavnene
-- for hver kunde i ett array med aggregatfunksjonen array_agg
SELECT c.company_name, array_agg(p.product_name) AS reccomended_products
FROM customers AS c INNER JOIN reccomend AS r USING (customer_id)
     INNER JOIN products AS p USING (product_id)
GROUP BY c.customer_id, c.company_name;
```

Eksempler på hva dere skal lære

Modellering og ER



Eksempel ER-modell av informasjon om fabrikk som produserer enheter.

Litt mer motivasjon

Tall fra Stack Overflows *Developer Syrvey 2024*:

`https://survey.stackoverflow.co/2024/technology`

- SQL nest mest brukte språket blant profesjonelle utviklere!
- (PostgreSQL mest brukte databasesystemet)

Et par ord om læring

- Delta på undervisningen, gjør alle oppgaver, les tilbakemeldinger, osv.
- Husk: dere er er for å lære, ikke for å løse problemer!
- En av grunnene til endret oblig-opplegg
- Sjekk egen forståelse, og vær ærlige med dere selv
- Denne muligheten til å virkelig lære noe dypt får dere sjeldent igjen!

Forelesning vs. video/opptak

Hvorfor ha/møte til fysiske forelesninger?

- Gir noe ekstra å se noe i levende live (f.eks. teater)
- Tilskuere gir energi
- Tilskuere gir indirekte tilbakemelding (f.eks. blanke ansikter)
- Dere kan også gi direkte tilbakemelding (spørsmål, kommentarer)
- Dere kan diskutere med hverandre, snakke sammen
- Bygger et godt læringsmiljø

Så møt opp! :)

Teknologi, språkmodeller og læring

- Mye ny og gammel teknologi som kan løse problemer
- Store språkmodeller (som ChatGPT, Bing, osv.) er flotte verktøy!
- Vil nok klare mange av oppgavene i dette kurset
- Derfor to viktige spørsmål:
  - Er det noen grunn til å lære seg pensum?
  - Hvordan bruke store språkmodeller i studiene?

Er det noen grunn til å lære seg pensum? (1)

JA! :D

- Språkmodeller er langt flinkere på problemene i dette kurset enn pensum generelt!
  - Problemene dere får er “enkle” og vanlige
  - Modellene er derfor godt trent på disse, men ikke “ekte” problemer
  - Modellene sliter med å sette sammen mindre program-snutter til større løsninger
- Ikke alltid mulig å bruke språkmodeller
  - Sensitive domener
  - For store eller komplekse problemer
  - Dette kan endre seg, men vanskelig å spå fremtiden

Er det noen grunn til å lære seg pensum? (2)

JA! :D

- Må gi riktig input/prompt
  - Krever ofte dyp forståelse av domenet
  - Må vite hva som er mulig og ikke
- Språkmodellene gjør feil
  - Feil i selve løsningen
  - Løser feil problem
  - Svarene må derfor sjekkes
- Språkmodellene kan kun det den kan
  - Nye teknologier kommer og går
  - Kan ikke forskning og utvikling
- Det er gøy!

Hvordan bruke store språkmodeller i studiene?

- Ikke bruk språkmodeller til å løse oppgaver!
  - Læring krever at man faktisk tenker
  - Lærer lite av å bare se en løsning
- Bruk dem heller for samtaler om pensum!
  - Still generelle spørsmål og oppfølgingsspørsmål
  - Husk: De gjør fortsatt feil! Test ut svaret den gir
- Læring er noe du må gjøre selv og kan ikke automatiseres!

Nyttige lenker (Ikke pensum)

- Tidligere års forelesninger (opptak, slides, videoer, osv.)
  - https://www.uio.no/studier/emner/matnat/ifi/IN2090/h19/timeplan/index.html
  - https://www.uio.no/studier/emner/matnat/ifi/IN2090/h20/timeplan/index.html
  - https://www.uio.no/studier/emner/matnat/ifi/IN2090/h21/timeplan/index.html
  - https://www.uio.no/studier/emner/matnat/ifi/IN2090/h22/timeplan/index.html
  - https://www.uio.no/studier/emner/matnat/ifi/IN2090/h23/timeplan/index.html
- Mer om databaser:
  - Engelsk: `https://en.wikipedia.org/wiki/Database`
  - Norsk: `https://no.wikipedia.org/wiki/Database`
- Mer om relasjons-databaser:
  - Engelsk: `https://en.wikipedia.org/wiki/Relational_database`
  - Norsk: `https://no.wikipedia.org/wiki/Relasjonsdatabase`
- Mer om datamodellering:
  - Engelsk: `https://en.wikipedia.org/wiki/Data_modeling`
  - Engelsk: `https://en.wikipedia.org/wiki/Data_model`
- *What is the benefit of learning SQL?* (engelsk):
  `https://www.quora.com/What-is-the-benefit-of-learning-SQL`

IN2090 – Databaser og datamodellering

02 – Relasjonsmodellen

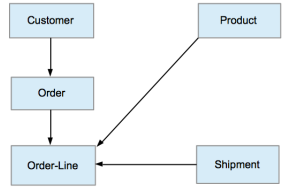
Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Litt historie

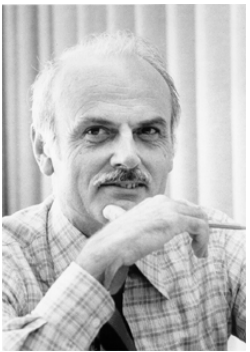
- Et av de første databasesystemet ble utformet av Bachman og Williams i 1964
- Ble solgt under navnet IDS (*Integrated Data Store*)
- IDS var en *nettverksdatabase* og var designet for bruk fra et programmeringsspråk
- I 1968 kom IBM med IMS (*Information Management System*), en *hierarkisk database* som forenkling av nettverksdatabaser
- IMS var nær enerådende for administrativ databehandling
- Fortsatt en del store firma som har legacy-systemer som bruker IMS



Eksempel nettverksdatabaseskjema

Litt mer historie

- I 1970 presenterte E. F. Codd sin *relasjonsmodell*
- Dette var den teoretiske beskrivelsen av den da nye typen databaser kalt *relasjonsdatabaser*
- Relasjonsdatabaser er enkle å beskrive og bruke, men vanskelig å implementere
- Først i 1977 klarte Oracle å lage et databasesystem som fortjener betegnelsen *relasjonell*
- Relasjonsdatabaser er den klart mest brukte typen database



Edgar Frank ”Ted” Codd

Hva er relasjonsmodellen?

Customers

CustomerID (int)	Name (text)	Birthdate (date)	NrProducts (int)
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

- Forrige uke: Relasjonsdatabaser inneholder relasjoner/tabeller som består av kolonner og rader
- Relasjonsmodellen: formell/matematisk beskrivelse av denne måten å representere data
- Presis beskrivelse av:
  - hva en relasjon er (hva de består av, når er de like, osv.)
  - hvordan man beskriver data med relasjoner (struktur, identitet, referanser)
  - hvordan man kan hente ut og manipulere data representert med relasjoner
- Altså: komplett beskrivelse av hvordan bruke relasjoner for å beskrive data

Relasjoner og relasjonsdatabaser

Customers

CustomerID (int)	Name (text)	Birthdate (date)	NrProducts (int)
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

- *Relasjon:* en *signatur* og en *mengde tupler*
- Mengde: Samling av unike elementer (uordnet)
  - F.eks. $\{1, 2, 3\} = \{3, 1, 2\}$
- Tuppel: Liste med verdier (ordnet)
  - F.eks. $(1, 2, \text{'Ole'}) \neq (\text{'Ole'}, 1, 2)$
- Relasjonsdatabase: *mengde med relasjoner* med unikt navn

```
(Customers(CustomerID, Name, Birthdate, NrProducts),
 {(0, Anna Consuma,  1978-10-09, 19),
  (1, Peter Young,   2009-03-01, 1),
  (2, Carla Smith,   1986-06-14, 8),
  (3, Sam Penny,     1961-01-09, 14),
  (4, John Mill,     1989-11-16, 8),
  (5, Yvonne Potter, 1971-04-12, 6)})
```

Signatur og skjema

Customers

CustomerID (int)	Name (text)	Birthdate (date)	NrProducts (int)
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

- En *relasjonssignatur* består av et navn og en mengde attributter/kolonner
- Skrives ofte

  `Customers(CustomerID (int), Name (text), Birthdate (date), NrProducts (int))`

  eller

  `Customers(CustomerID, Name, Birthdate, NrProducts)`
- Med mer matematisk notasjon (hvor et par av `A` og `B` skriver `A: B`)

  `Customers: {CustomerID: int, Name: text, Birthdate: date, NrProducts: int}`
- Inneholder også nøkler (mer om dette i neste video)
- En signatur gir deg det du trenger å vite for å bruke relasjonen
- *Relasjonsskjema:* mengde med relasjonssignaturer

Attributt

Customers

CustomerID (int)	Name (text)	Birthdate (date)	NrProducts (int)
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

- Intuitivt: En kolonne
- Består av et navn og en type (av og til kalt domene)
- To attributter kan ikke ha samme navn (i samme relasjon)

Tuppel

Customers

CustomerID (int)	Name (text)	Birthdate (date)	NrProducts (int)
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

- Intuitivt: en rad
- Formelt: mengde med par av attributtnavn og verdier, f.eks.:

```
{CustomerID: '2', Name: 'Carla Smith', Birthdate: '1986-06-14', NrProducts: '8'}
```

- Alså ikke helt likt det vi vanligvis kaller tuppel, f.eks.:

```
('2', 'Carla Smith', '1986-06-14', '8')
```

- Vi kan bruke attributt-navn fremfor posisjon i tuplene! (F.eks. `t.Name`)
- Gitt en rekkefølge på attributtene kan vi droppe dem i tuplene
- Tuppel kalles av og til *instans*

Verdier/Data

Customers

CustomerID (int)	Name (text)	Birthdate (date)	NrProducts (int)
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

- Kun atomære verdier (med samme type som attributten/kolonnen)
- Atomær er et litt vagt begrep og avhenger av bruk
  - Kan brukes direkte, uten å måtte ”tolkes” eller ”pakkes ut” først
- `NULL`: Markerer manglende eller ukjent verdi

Full matematisk representasjon

Customers

CustomerID (int)	Name (text)	Birthdate (date)	NrProducts (int)
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

```
( Customers: { CustomerID: int, Name: text, Birthdate: date, NrProducts: int },
  { { CustomerId: '0', Name: 'Anna Consuma',  Birthdate: '1978-10-09', NrProducts: '19' },
    { CustomerId: '1', Name: 'Peter Young',   Birthdate: '2009-03-01', NrProducts: '1' },
    { CustomerId: '2', Name: 'Carla Smith',   Birthdate: '1986-06-14', NrProducts: '8' },
    { CustomerId: '3', Name: 'Sam Penny',     Birthdate: '1961-01-09', NrProducts: '14' },
    { CustomerId: '4', Name: 'John Mill',     Birthdate: '1989-11-16', NrProducts: '8' },
    { CustomerId: '5', Name: 'Yvonne Potter', Birthdate: '1971-04-12', NrProducts: '6' } } )
```

Eksempler: Ugyldig relasjon

Person

Epost (text)	(text)	Nummer (text)	By	Gate (text)	Nummer (int)	Postnummer (int)
ola@online.no	Ola Norman	01028912345	Oslo	Gateveien	2	ukjent
ali@hmail.net	Ali Nilsen	05109354321	Bergen	Nedre gate	første	5432
kari@epost.no	Kari Mo	vet ikke	Oslo	Veigata	7	2341

Eksempler: Like og ulike relasjoner

Student

Brukernavn (text)	Navn (text)	Program (text)
olano	Ola Norman	PROSA
alini	Ali Nilsen	DESIGN
karimo	Kari Mo	PROSA

?
=

Student

Brukernavn (text)	Program (text)	Navn (text)
alini	DESIGN	Ali Nilsen
olano	PROSA	Ola Norman
karimo	PROSA	Kari Mo

```
(
 (Student, {Brukernavn: text, Navn: text, Program: text}),
 {
  {Brukernavn: 'olano', Navn: 'Ola Norman', Program: 'PROSA'},
  {Brukernavn: 'alini', Navn: 'Ali Nilsen', Program: 'DESIGN'},
  {Brukernavn: 'karimo', Navn: 'Kari Mo', Program: 'PROSA'}
 }
)
```

?
=

```
(
 (Student, {Brukernavn: text, Program: text, Navn: text}),
 {
  {Brukernavn: 'alini', Program: 'DESIGN', Navn: 'Ali Nilsen'},
  {Brukernavn: 'olano', Program: 'PROSA', Navn: 'Ola Norman'},
  {Brukernavn: 'karimo', Program: 'PROSA', Navn: 'Kari Mo'}
 }
)
```

Takk for nå!

Neste video handler om nøkler.

IN2090 – Databaser og datamodellering

02 – Relasjonsmodellen: Nøkler

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Motiverende eksempel

La oss si vi skal lage en database for å lagre data om studenter, kurs, og karakterer.

- For studenter: Brukernavn, navn, etternavn, adresse...
- For kurs: Kurskode (fagkode + emnenr.), tittel, beskrivelse, studiepoeng...
- Hvem har fått hvilken karakter i hva.

Mulig løsning, alt i én relasjon:

Brnavn	Navn	Etternavn	Adresse	Fagkode	Emnenummer	Tittel	Beskrivelse	AntSP	Kara
evgenit	Evgenij	Thorstensen	Adr1	IN	2090	Databaser	EnBeskr...	10	B
peternl	Petter	Nilsen	Adr2	IN	2090	Databaser	EnBeskr...	10	A
evgenit	Evgenij	Thorstensen	Adr1	IN	2080	Beregn...	Descr...	10	A
leifhka	Leif H.	Karlsen	Adr3	IN	2090	Databaser	EnBeskr...	10	B
leifhka	Leif H.	Karlsen	Adr3	IN	3110	Program...	EnBeskr2...	5	C

Bedre løsning

Student

Brnavn	Navn	Etternavn	Adresse
evgenit	Evgenij	Thorstensen	Adr1
peternl	Petter	Nilsen	Adr2
leifhka	Leif H.	Karlsen	Adr3

Karakter

Brnavn	Fagkode	Emnenummer	Kara
evgenit	IN	2090	B
peternl	IN	2090	A
evgenit	IN	2080	B
leifhka	IN	2090	B
leifhka	IN	3110	C

Kurs

Fagkode	Emnenummer	Tittel	Beskrivelse	AntSP
IN	2090	Databaser	EnBeskr...	10
IN	2080	Beregn...	Descr...	10
IN	3110	Program...	EnBeskr2...	5

Merk: Samme attributtene og samme verdiene!

Design av databaser

- Kommer tilbake til god databasedesign senere i semesteret
- Tommelfingerregel:
  - Én relasjon per entitet (kurs, student, osv.)
  - Én relasjon per forhold (karakter, foreleser for, gift med, osv.)
  - Aldri repeter data

Identitet

- En database skal beskrive ting og deres forhold
- Essensielt å vite hva/hvem man beskriver
- Trenger noe som unikt peker ut en ting, f.eks.:
  - for kurs – kurskoder (fagkode + emnenummer)
  - for personer – personnummer
  - for biler – skiltnummer
- For en relasjon vil dette være én eller fler attributter med unik(e) verdier
- Altså, en *mengde med attributter*, f.eks.:
  - `{Brnavn}` for `Student`
  - `{Fagkode, Emnenummer}` for `Kurs`

`Karakter`

Student	Fagkode	Emnenummer	Kara
Petter	IN	2090	A
Leif H.	IN	2000	C
Petter	IN	3200	B

`Student`

Brnavn	Navn	Etternavn	Adresse
evgenit	Evgenij	Thorstensen	Adr1
peternl	Petter	Nilsen	Adr2
leifhka	Leif H.	Karlsen	Adr3
peterol	Petter	Olsen	Adr4

Supernøkler

- *Supernøkkel:* Mengde med attributter som alltid har unike verdier i en relasjon
- Kan bruke en supernøkkel for å unikt identifisere en rad (altså en ting)
- En relasjon kan ha mange supernøkler
  - Hvis vi har en supernøkkel, vil alle utvidelser også være en supernøkkel
  - Mengden av alle attributter for en relasjon er alltid en supernøkkel
- Viktig: En supernøkkel sier hva som *alltid* er unikt
  - Relasjoner kan endre seg over tid, og en nøkkel skal alltid være unik
  - Ikke bare hva som er unikt i relasjonen slik den ser ut nå
  - Beskriver altså virkeligheten (ikke bare dataene)

Supernøkler: Eksempel 1

Student

Brnavn	Navn	Etternavn	Adresse
evgenit	Evgenij	Thorstensen	Gateveien 1a
peternl	Petter	Nilsen	Stedplassen 23
leifhka	Leif H.	Karlsen	Bergfjellet 42
peterol	Petter	Olsen	Gateveien 3c

Supernøkler (eksempler):

- {Brnavn}
- {Brnavn, Navn}
- {Brnavn, Navn, Etternavn}
- {Brnavn, Etternavn}

Er følgende supernøkler?

- {Navn, Etternavn}? Nei!
- {Navn, Etternavn, Adresse}? Nei!
- {Adresse, Brnavn}? Ja!
- {Brnavn, Navn, Etternavn, Adresse}? Ja!

Supernøkler: Eksempel 2

Kurs

Fagkode	Emnenummer	Tittel	AntSP
IN	2090	Databaser og datamodellering	10
IN	2010	Algoritmer og datastrukturer	10
AST	3220	Kosmologi 1	10
MAT	2000	Prosjektarbeid i matematikk	10
BIOS	9312	Alpine Ecology	5
IN	2000	Software engineering	10

Supernøkler (eksempler):

- {Fagkode, Emnenummer}
- {Tittel}
- {Fagkode, Tittel}
- {Tittel, AntSP}
- {Fagkode, Emnenummer, Tittel, AntSP}

Er følgende supernøkler?

- {Fagkode, AntSP}? Nei!
- {Emnenummer}? Nei!
- {Emnenummer, Tittel}? Ja!

Supernøkler: Eksempel 3

Karakter

Brnavn	Fagkode	Emnenummer	Kara
evgenit	IN	2090	B
peternl	IN	2090	A
evgenit	IN	2080	B
leifhka	IN	2090	B
leifhka	IN	3110	C
evgenit	MAT	1100	B

Hvilke supernøkler har `Karakter`?

- Avhenger av om en student kan ha flere karakterer i samme kurs
- Hvis ja: kun {`Brnavn, Fagkode, Emnenummer, Kara`}
- Hvis nei: også {`Brnavn, Fagkode, Emnenummer`}

Kandidatnøkler

- Supernøkler gir oss det vi trenger, men kan inneholde mye unødvendig
- F.eks. var det nok med bare `Brnavn` for `Student`
- *Kandidatnøkkel:* En minimal supernøkkel
- Supernøkkel er en kandidatnøkkel dersom det ikke går an å fjerne en attributt og fortsatt ha en supernøkkel

Kandidatnøkler: Eksempler

Kurs

Fagkode	Emnenummer	Tittel	AntSP
IN	2090	Databaser og datamodellering	10
IN	2010	Algoritmer og datastrukturer	10
AST	3220	Kosmologi 1	10
MAT	2000	Prosjektarbeid i matematikk	10
BIOS	9312	Alpine Ecology	5
IN	2000	Software engineering	10

Kandidatnøkler:

- {Fagkode, Emnenummer}
- {Tittel}

Ikke kandidatnøkler:

- {Tittel, AntSP}

Er følgende kandidatnøkler?

- {Fagkode, Emnenummer, AntSP}? Nei!
- {Emnenummer}? Nei!
- {Emnenummer, Tittel}? Nei!

Primærnøkkel

- Gir mer mening å bruke kandidatnøklene for å identifisere ting
- En relasjon kan ha flere kandidatnøkler
  - F.eks. {Fagkode, Emnenummer} og {Tittel} for Kurs
- Vi velger derfor ut én som vi ønsker å bruke
- *Primærnøkkel:* En slik utvalgt kandidatnøkkel
- Merk: Hvis en relasjon bare har én kandidatnøkkel må denne bli primærnøkkel

Student

Brnavn	Navn	Etternavn	Adresse
evgenit	Evgenij	Thorstensen	Adr1
peternl	Petter	Nilsen	Adr2
leifhka	Leif H.	Karlsen	Adr3

- **Kandidatnøkler:** {Brnavn}
- **Valgt primærnøkkel:** {Brnavn}

Kurs

Fagkode	Emnenummer	Tittel	AntSP
IN	2090	Databaser og datamodellering	10
IN	2010	Algoritmer og datastrukturer	10
AST	3220	Kosmologi 1	10
MAT	2000	Prosjektarbeid i matematikk	10
BIOS	9312	Alpine Ecology	5
IN	2000	Software engineering	10

- **Kandidatnøkler:** {Fagkode, Emnenummer}, {Tittel}
- **Valgt primærnøkkel:** {Fagkode, Emnenummer}

Fremmednøkkel

- Alle ting som representeres i databasen har en primærnøkkel
- Kan så bruke verdiene fra denne primærnøkkelen i andre relasjoner
- Dette danner da en referanse fra en relasjon til en annen
- *Fremmednøkkel:* En slik referanse

Eksempel: Fremmednøkkel

Karakter

Student	Fagkode	Emnenummer	Kara
evgenit	IN	2090	B
peternl	AST	3220	A
evgenit	IN	2080	B
leifhka	IN	2090	B
leifhka	MAT	2000	C

Student

Brnavn	Navn	Etternavn	Adresse
evgenit	Evgenij	Thorstensen	Gateveien 1a
peternl	Petter	Nilsen	Stedplassen 23
leifhka	Leif H.	Karlsen	Bergfjellet 42

Kurs

Fagkode	Emnenummer	Tittel	AntSP
IN	2090	Databaser og datamodellering	10
IN	2010	Algoritmer og datastrukturer	10
AST	3220	Kosmologi 1	10
MAT	2000	Prosjektarbeid i matematikk	10
BIOS	9312	Alpine Ecology	5
IN	2000	Software engineering	10

- `Karakter(Student)` refererer til `Student(Brnavn)`
- `Karakter(Fagkode, Emnenummer)` refererer til `Kurs(Fagkode, Emnenummer)`

Mer om fremmednøkler

- Fremmednøkkel sier at verdier i kolonnen(e) må finnes i de(n) refererte kolonnen(e)
  - Alle brukernavn i `Karakter(Student)` må også finnes i `Student(Brnavn)`
- Skriver ofte fremmednøkler med en pil
  - F.eks.: `Karakter(Student) -> Student(Brnavn)`
- Merk: følgende er ikke ekvivalent!
  - `Karakter(Fagkode, Emnenummer) -> Kurs(Fagkode, Emnenummer)`
  - `Karakter(Fagkode) -> Kurs(Fagkode)`
    `Karakter(Emnenummer) -> Kurs(Emnenummer)`

Kurs

Fagkode	Emnenummer	Tittel	AntSP
IN	2090	Databaser og datamodellering	10
AST	3220	Kosmologi 1	10
BIOS	9312	Alpine Ecology	5

Karakter

Student	Fagkode	Emnenummer	Kara
evgenit	IN	2090	B
peternl	AST	3220	A
leifhka	IN	3220	C

Markere nøkler

- I forrige video lærte vi om signaturer, f.eks.:

  `Kurs(Fagkode, Emnenummer, Tittel, AntSP)`

- Bruker gjerne understreking for å markere kandidatnøkler, f.eks.:

  `Kurs(Fagkode, Emnenummer, Tittel, AntSP)`

- Hvis det er mer enn én kandidatnøkkel bruker vi gjerne dobbel understrek for primærnøkkel:

  `Kurs(Fagkode, Emnenummer, Tittel, AntSP)`

- Kan også bare skrive dem etter signaturen med vanlig tekst, f.eks:

  ```
  Kurs(Fagkode, Emnenummer, Tittel, AntSP)
  Kandidatnøkler: {Fagkode, Emnenummer}, {Tittel}
  Primærnøkkel: {Fagkode, Emnenummer}
  ```

  eller

  ```
  Kurs(Fagkode, Emnenummer, Tittel, AntSP)
  Kandidatnøkler: Kurs(Fagkode, Emnenummer), Kurs(Tittel)
  Primærnøkkel: Kurs(Fagkode, Emnenummer)
  ```

Takk for nå!

Neste video handler om relasjonsalgebra.

IN2090 – Databaser og datamodellering

02 – Relasjonsalgebra

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Hva er relasjonsalgebra?

- Vi har nå sett hvordan vi representerer data
- Ikke hvordan vi bruker dataene
- Med bruk mener vi:
  - Hente ut spesifikke data basert på én eller fler relasjoner
  - Kombinere og transformere relasjoner til nye relasjoner
- Relasjonsalgebra er et (presist/matematisk) språk for å bruke dataene beskrevet med relasjoner
- Danner fundamentet for databasespråket SQL

Relasjonsalgebra

- En algebra er en mengde med operasjoner/funksjoner
- Operasjonene/funksjonene har argumenter og resultat fra samme mengde/type
- Eksempel på en vanlig algebra: $\{+, -, \times, \div\}$ over reelle tall
  - F.eks. $(5 \times (3 + 4)) \div 4 = 8.75$
- Annet eksempel på en vanlig algebra: $\{\cap, \cup, \setminus\}$ over mengder
  - F.eks. $(\{1\} \cup (\{3, 4\} \setminus \{4, 5\})) \cap \{1, 2, 3\} = \{1, 3\}$
- *Relasjonsalgebra:* mengde med operasjoner over relasjoner
- En operasjon i relasjonsalgebra tar derfor én eller to relasjoner som argument, og returnerer en ny relasjon
- En slik algebra lar oss kombinere og hente ut data fra relasjonene

Projeksjon (π)

$$\pi_{\text{Brnavn,Etternavn}}(\texttt{Student}) =$$

Brnavn	Etternavn
evgenit	Thorstensen
peternl	Nilsen
leifhka	Karlsen

- π er unær (tar ett argument)
- Har en mengde med attributter som subskrift
- Returnerer ny relasjon med kun attributtene listet opp
- Den velger altså ut attributter
- Merk: Operasjonene *endrer* ikke de originale relasjonene i databasen, bare returnerer en ny relasjon
- Merk: Resultatet er en relasjon uten navn. Kan gi den et navn slik:

$$\texttt{StudentNavn} := \pi_{\text{Brnavn, Etternavn}}(\texttt{Student})$$

Seleksjon (σ)

$$\sigma_{\text{Emnenummer} \geq 3000 \wedge \text{AntSP} = 10}(\texttt{Kurs})$$

$=$

Fagkode	Emnernummer	Tittel	AntSP
AST	3220	Kosmologi 1	10
IN	4070	Logikk	10

- σ er også unær (tar ett argument)
- Har et uttrykk med attributt-navnene som variable som subskrift
  - Bruker symbolene $\wedge, \vee, \neg, \geq, \leq, >, <, =$ og konstanter (3000, 10, *IN*, osv.)
- Returnerer ny relasjon med kun de tuplene som tilfredstiller uttrykket
- Men samme attributter
- Den velger altså ut de radene vi er interessert i

Omdøping (ρ)

$$\rho_{\text{Tittel}\to\text{Navn},\text{AntSP}\to\text{Poeng}}(\texttt{Kurs})$$

$=$

Fagkode	Emnernummer	Navn	Poeng
IN	2090	Databaser og datamodellering	10
IN	2010	Algoritmer og datastrukturer	10
AST	3220	Kosmologi 1	10
MAT	2000	Prosjektarbeid	10
BIOS	9312	Alpine Ecology	5
IN	4070	Logikk	10

- ρ er også unær (tar ett argument)
- Har en mengde med attributt-pil-attributt som subskrift
- $A \to B$ sier at A skal omdøpes til B
- Returnerer ny relasjon med nye attributt-navn ihht. omdøpingene
- Endrer ellers ingenting
- Enkelte kilder bruker en litt annen syntaks

Eksempel: Unære operasjoner

Kurs

Fagkode	Emnernr	Tittel	AntSP
IN	2090	Databaser og datamodellering	10
IN	2010	Algoritmer og datastrukturer	10
AST	3220	Kosmologi 1	10
MAT	2000	Prosjektarbeid	10
IN	4070	Logikk	10
BIOS	9312	Alpine Ecology	5
IN	5800	Declarative data engineering	10
MAT	4500	Toplogi	10
IN	4230	Nettverk	10

Finn tittel på alle informatikk-masterkurs:

$$\rho_{\texttt{Tittel}\rightarrow\texttt{IfiMasterKurs}}\left(\pi_{\texttt{Tittel}}\left(\sigma_{\texttt{Fagkode}=\text{'IN'}\wedge\texttt{Emnenummer}\geq 4000}\left(\texttt{Kurs}\right)\right)\right)$$

$=$

Fagkode	Emnernr	Tittel	AntSP
IN	2090	Databaser og datamodellering	10
IN	2010	Algoritmer og datastrukturer	10
AST	3220	Kosmologi 1	10
MAT	2000	Prosjektarbeid	10
IN	4070	Logikk	10
BIOS	9312	Alpine Ecology	5
IN	5800	Declarative data engineering	10
MAT	4500	Toplogi	10
IN	4230	Nettverk	10

Kartesisk produkt ($\times$)

$$\texttt{Student} \times \texttt{Karakter} =$$

Brnavn	Navn	Etternavn	Adresse	Student	Fagkode	Emnenummer	Kara
evgenit	Evgenij	Thorstensen	Gateveien 1a	evgenit	IN	2090	B
evgenit	Evgenij	Thorstensen	Gateveien 1a	peternl	AST	3220	A
evgenit	Evgenij	Thorstensen	Gateveien 1a	evgenit	IN	2010	B
evgenit	Evgenij	Thorstensen	Gateveien 1a	leifhka	IN	2090	B
evgenit	Evgenij	Thorstensen	Gateveien 1a	leifhka	MAT	2000	C
peternl	Petter	Nilsen	Stedplassen 23	evgenit	IN	2090	B
peternl	Petter	Nilsen	Stedplassen 23	peternl	AST	3220	A
peternl	Petter	Nilsen	Stedplassen 23	evgenit	IN	2010	B
peternl	Petter	Nilsen	Stedplassen 23	leifhka	IN	2090	B
peternl	Petter	Nilsen	Stedplassen 23	leifhka	MAT	2000	C
leifhka	Leif H.	Karlsen	Bergfjellet 42	evgenit	IN	2090	B
leifhka	Leif H.	Karlsen	Bergfjellet 42	peternl	AST	3220	A
leifhka	Leif H.	Karlsen	Bergfjellet 42	evgenit	IN	2010	B
leifhka	Leif H.	Karlsen	Bergfjellet 42	leifhka	IN	2090	B
leifhka	Leif H.	Karlsen	Bergfjellet 42	leifhka	MAT	2000	C

- $\times$ er binær (tar to argumenter)
- Returnerer ny relasjon med:
  - Alle attributtene til begge relasjonene
  - Alle kombinasjoner av tupler fra de to relasjonene

Eksempel: Kartesisk produkt ($\times$)

Finn info om alle studenter og deres karakterer:

$\sigma_{\text{Brnavn}=\text{Student}}$ (

Brnavn	Navn	Etternavn	Adresse
evgenit	Evgenij	Thorstensen	Gateveien 1a
peternl	Petter	Nilsen	Stedplassen 23
leifhka	Leif H.	Karlsen	Bergfjellet 42

$\times$

Student	Fagkode	Emnenummer	Kara
evgenit	IN	2090	B
peternl	AST	3220	A
evgenit	IN	2010	B
leifhka	IN	2090	B
leifhka	MAT	2000	C

)

=

Brnavn	Navn	Etternavn	Adresse	Student	Fagkode	Emnenummer	Kara
evgenit	Evgenij	Thorstensen	Gateveien 1a	evgenit	IN	2090	B
evgenit	Evgenij	Thorstensen	Gateveien 1a	peternl	AST	3220	A
evgenit	Evgenij	Thorstensen	Gateveien 1a	evgenit	IN	2010	B
evgenit	Evgenij	Thorstensen	Gateveien 1a	leifhka	IN	2090	B
evgenit	Evgenij	Thorstensen	Gateveien 1a	leifhka	MAT	2000	C
peternl	Petter	Nilsen	Stedplassen 23	evgenit	IN	2090	B
peternl	Petter	Nilsen	Stedplassen 23	peternl	AST	3220	A
peternl	Petter	Nilsen	Stedplassen 23	evgenit	IN	2010	B
peternl	Petter	Nilsen	Stedplassen 23	leifhka	IN	2090	B
peternl	Petter	Nilsen	Stedplassen 23	leifhka	MAT	2000	C
leifhka	Leif H.	Karlsen	Bergfjellet 42	evgenit	IN	2090	B
leifhka	Leif H.	Karlsen	Bergfjellet 42	peternl	AST	3220	A
leifhka	Leif H.	Karlsen	Bergfjellet 42	evgenit	IN	2010	B
leifhka	Leif H.	Karlsen	Bergfjellet 42	leifhka	IN	2090	B
leifhka	Leif H.	Karlsen	Bergfjellet 42	leifhka	MAT	2000	C

Join ($\bowtie$)

$$\texttt{Student} \bowtie_{\text{Brnavn}=\text{Student}} \texttt{Karakter}$$

$=$

Brnavn	Navn	Etternavn	Adresse	Student	Fagkode	Emnenummer	Kara
evgenit	Evgenij	Thorstensen	Gateveien 1a	evgenit	IN	2090	B
evgenit	Evgenij	Thorstensen	Gateveien 1a	evgenit	IN	2010	B
peternl	Petter	Nilsen	Stedplassen 23	peternl	AST	3220	A
leifhka	Leif H.	Karlsen	Bergfjellet 42	leifhka	IN	2090	B
leifhka	Leif H.	Karlsen	Bergfjellet 42	leifhka	MAT	2000	C

- $\bowtie$ er binær (tar to argumenter)
- Har en mengde med attributt-$=$-attributt som subskrift
- $A = B$ sier at A skal være lik B i resultatet
- Kan også ha mer generelle uttrykk (slik som for σ)
- Returnerer ny relasjon med alle kombinasjoner av tupler fra de to relasjonene som tilfredstiller uttrykket (med alle attributtene fra begge relasjonene)
- Merk: $R \bowtie_e P$ er ekvivalent med $\sigma_e(R \times P)$.

Eksempel: Join ($\bowtie$)

Finn karakterer med all info om kurset:

$\pi_{\text{Student, Fagkode, Emnenummer, Karakter, Tittel, AntSP}}($

Student	Fagkode	Emnenummer	Kara
evgenit	IN	2090	B
peternl	AST	3220	A
evgenit	IN	2010	B
leifhka	IN	2090	B
leifhka	MAT	2000	C

$\bowtie_{\text{Fagkode} = \text{Fagkode} \wedge \text{Emnenummer} = \text{Emnenummer}}$???

$\rho_{\text{Fagkode} \to \text{Kode}, \text{Emnenummer} \to \text{Nummer}}$

Fagkode	Emnernummer	Tittel	AntSP
IN	2090	Databaser og datamodellering	10
IN	2010	Algoritmer og datastrukturer	10
AST	3220	Kosmologi 1	10
MAT	2000	Prosjektarbeid	10
BIOS	9312	Alpine Ecology	5
IN	4070	Logikk	10

$)$

$=$

Student	Fagkode	Emnenummer	Kara	Kode	Nummer	Tittel	AntSP
evgenit	IN	2090	B	IN	2090	Databaser og datamodellering	10
peternl	AST	3220	A	AST	3220	Kosmologi 1	10
evgenit	IN	2010	B	IN	2010	Algoritmer og datastrukturer	10
leifhka	IN	2090	B	IN	2090	Databaser og datamodellering	10
leifhka	MAT	2000	C	MAT	2000	Prosjektarbeid	10

Eksempel: Join ($\bowtie$)

Finn karakterer med all info om kurset:

$$\pi_{\text{Student, Fagkode, Emnenummer, Karakter, Tittel, AntSP}}\Big(\\ Karakterer \\ \bowtie_{\text{Fagkode} = \text{Kode} \wedge \text{Emnenummer} = \text{Nummer}} \\ \rho_{\text{Fagkode} \to \text{Kode}, \text{Emnenummer} \to \text{Nummer}}\ (Kurs)\\ \Big)$$

$=$

Student	Fagkode	Emnenummer	Kara	Tittel	AntSP
evgenit	IN	2090	B	Databaser og datamodellering	10
peternl	AST	3220	A	Kosmologi 1	10
evgenit	IN	2010	B	Algoritmer og datastrukturer	10
leifhka	IN	2090	B	Databaser og datamodellering	10
leifhka	MAT	2000	C	Prosjektarbeid	10

Naturlig join ($\star$)

Student	Fagkode	Emnenummer	Kara
evgenit	IN	2090	B
peternl	AST	3220	A
evgenit	IN	2010	B
leifhka	IN	2090	B
leifhka	MAT	2000	C

$\star$

Fagkode	Emnernummer	Tittel	AntSP
IN	2090	Databaser og datamodellering	10
IN	2010	Algoritmer og datastrukturer	10
AST	3220	Kosmologi 1	10
MAT	2000	Prosjektarbeid	10
BIOS	9312	Alpine Ecology	5
IN	4070	Logikk	10

$=$

Student	Fagkode	Emnenummer	Kara	Tittel	AntSP
evgenit	IN	2090	B	Databaser og datamodellering	10
peternl	AST	3220	A	Kosmologi 1	10
evgenit	IN	2010	B	Algoritmer og datastrukturer	10
leifhka	IN	2090	B	Databaser og datamodellering	10
leifhka	MAT	2000	C	Prosjektarbeid	10

- $\star$ er binær (tar to argumenter)
- Utfører en join på alle attributter med likt navn (og velger ut én kopi av hver)
- Dersom ingen attributter har likt navn får man det kartesiske produktet ($\times$)
- Må bruke ρ dersom enten:
  - Man ønsker å joine på attributter med ulikt navn (bruker ρ for å gjøre dem like)
  - Man ønsker ikke å joine på to attributter som har likt navn (bruker ρ for å gjøre dem ulike)

Snitt, union og differanse ($\cap, \cup, \setminus$)

Bachelorkurs

Fagkode	Emnernr	Tittel	AntSP
IN	2090	Databaser og datamodellering	10
IN	2010	Algoritmer og datastrukturer	10
AST	3220	Kosmologi 1	10
MAT	2000	Prosjektarbeid	10

Masterkurs

Fagkode	Emnernr	Tittel	AntSP
IN	4070	Logikk	10
BIOS	9312	Alpine Ecology	5
IN	5800	Declarative data engineering	10
MAT	4500	Toplogi	10

$\texttt{Bachelorkurs} \cup \texttt{Masterkurs} =$

Fagkode	Emnernr	Tittel	AntSP
IN	2090	Databaser og datamodellering	10
IN	2010	Algoritmer og datastrukturer	10
AST	3220	Kosmologi 1	10
MAT	2000	Prosjektarbeid	10
IN	4070	Logikk	10
BIOS	9312	Alpine Ecology	5
IN	5800	Declarative data engineering	10
MAT	4500	Toplogi	10

- $\cup, \cap$ og $\setminus$ er alle binær (tar to argumenter)
- Utfører vanlig union, snitt og differanse på mengden av tupler
- Krever at begge argument-relasjonene har samme signatur (attributt-navn og typer)

Større eksempel

Karakter

Student	Fagkode	Emnenummer	Kara
evgenit	IN	3220	A
evgenit	IN	2090	D
peternl	AST	3220	A
evgenit	IN	2010	B
leifhka	IN	2090	C
peternl	MAT	2000	A
leifhka	MAT	2000	E
erihan	IN	2090	A

Student

Brnavn	Navn	Etternavn	Adresse
evgenit	Evgenij	Thorstensen	Gateveien 1a
peternl	Petter	Nilsen	Stedplassen 23
leifhka	Leif H.	Karlsen	Bergfjellet 42
idamo	Ida	Mo	Knutekrysset 78b
erihan	Erik	Hansen	Bryggekaia 4

Finn navn på alle studenter som kun har fått A'er:

$$\pi_{\texttt{Navn},\texttt{Etternavn}}((\pi_{\texttt{Student}}(\sigma_{\texttt{Kara}=\texttt{'A'}}(\texttt{Karakter})) \setminus \pi_{\texttt{Student}}(\sigma_{\texttt{Kara}\neq\texttt{'A'}}(\texttt{Karakter}))) \bowtie_{\texttt{Brnavn}=\texttt{Student}} \texttt{Student})$$

$=$

Student	Fagkode	Emnenummer	Kara
evgenit	IN	3220	A
evgenit	IN	2090	B
peternl	AST	3220	A
evgenit	IN	2010	B
leifhka	IN	2090	B
peternl	MAT	2000	A
leifhka	MAT	2000	C
erihan	IN	2090	A

Takk for nå!

Neste uke handler om datamodellering.

IN2090 – Databaser og datamodellering

03 – Datamodellering: Entiteter

Leif Harald Karlsen
leifhka@ifi.uio.no



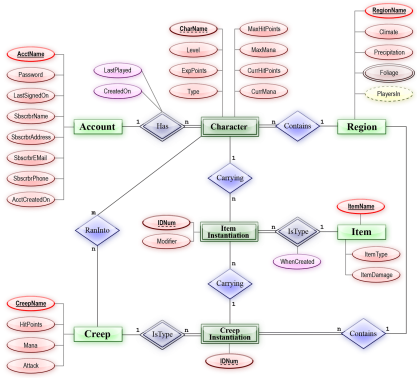
Universitetet i Oslo

Datamodellering

- Relasjonsmodellen: struktur og byggeklosser for databaser
- Vanskelig å lage gode relasjonsskjema (altså god struktur)
- Datamodellering hjelper til med dette
- Lager først en modell på et høyere nivå som så realiserer/oversettes til relasjoner
- Modellen uttrykkes i et eget språk: ER

ER

- ER (Entity-Relationship) er et visuelt modelleringsspråk
- Enkelt diagramspråk: bokser, ovaler, diamanter og streker
- Har 3 byggeklosser og streker mellom disse: Entiteter, attributter og relasjoner
- Kan uttrykke ulike egenskaper ved disse vha. visuelle markeringer
- Inneholder kun det som er relevant for relasjonsmodellen

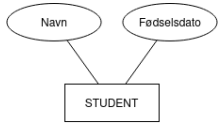


Entiteter



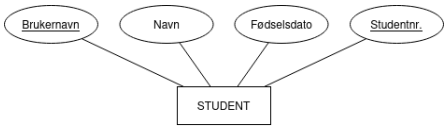
- Entitet: En konkret ting (Kari Eriksen, IN2090, Ole Johan Dahlshus, osv.)
- Entitetstype: Mengde entiteter med samme egenskaper/attributter
- Bruker ofte ordet “entitet” og “entitetstype” om hverandre
- Representeres i ER med et rektangel
- Har et unikt navn
- Har verdier knyttet til seg: attributter

Attributter



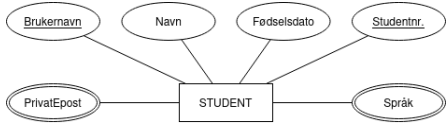
- Representeres ved ovaler i ER
- Knyttes til nøyaktig én entitetstype med en strek
- Har et unikt navn innenfor samme entitet
- Alle entiteter *kan* ha en verdi knyttet til attributten
- Alle entiteter *kan kun* ha verdier knyttet til en av entitetstypens attributter

Nøkkel-attributt



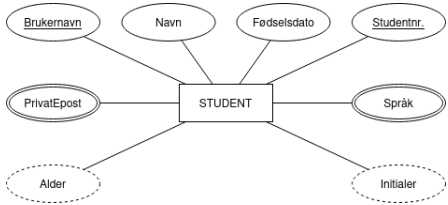
- Unike attributter kalles *nøkler*
- Markeres med en understrek under navnet
- Alle entiteter *må ha* en verdi for hver nøkkel

Flerverdi-attributt



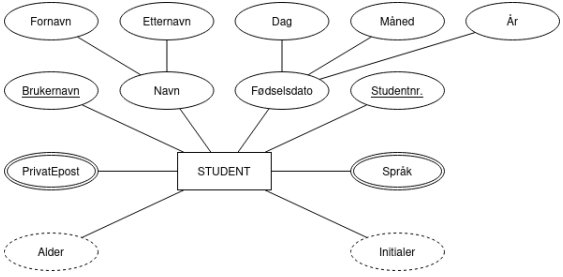
- Attributter hvor hver entitet kan ha flere verdier kalles *flerverdi*-attributter
- Markeres med dobbel-oval
- Kan ikke være nøkkel

Utledbar attributt



- Attributter hvis verdi kan utledes fra andre attributters verdi kalles *utledbar*
- Markeres med stiplet oval
- Merk: Sier ikke i ER-diagrammet *hvordan* den kan utledes!
- Litt vagt, men overordnet: Kan utledes vha.
  - Andre attributter i diagrammet
  - Vanlige operasjoner ($+, -, \cdot, \div$ på tall, `split`, `concat` på tekst, osv.)
  - Vanlige konstanter og bakgrunnskunnskap (tall og bokstaver, nå, dagens dato, dollarkurs, osv.)

Sammensatt attributt

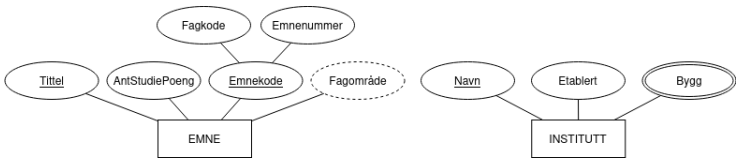


- En attributt kan bestå av flere attributter
- Kalles da *sammensatt*
- En sammensatt attributt kan bestå av sammensatte attributter
- Danner et tre
- Dersom en sammensatt attributt er nøkkel er det kombinasjonen av attributtene den består av som er unik

Eksempel

Lag en modell som modellerer følgende:

- Et emne har en unik tittel (f.eks. *Logikk*) og et antall studiepoeng (f.eks. *10*)
- Emner har også en unik emnekode (f.eks. *IN2090*)
- Emnekoden består igjen av en fagkode (f.eks. *IN*) og et emnenummer (f.eks. *2090*)
- Emner har et fagområde (f.eks. *informatikk*) som kan utledes fra fagkoden
- Et institutt har et unikt navn og et år det ble etablert
- Institutter har også en eller fler navn på bygg hvor de holder til



Takk for nå!

Neste video handler om relasjoner mellom entiteter.

IN2090 – Databaser og datamodellering

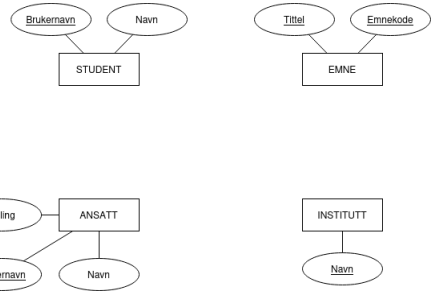
03 – Datamodellering: Relasjoner

Leif Harald Karlsen
leifhka@ifi.uio.no



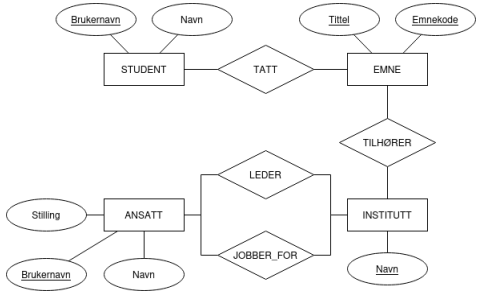
Universitetet i Oslo

Relasjoner



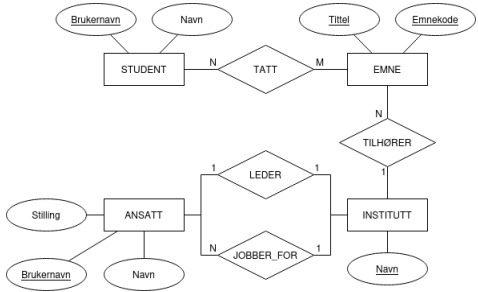
- Relasjoner er forhold som holder mellom entiteter
- Representeres med en diamant i ER
- Har et unikt navn
- Relaterer vanligvis to entiteter, men kan relatere flere (mer om dette neste uke)
- Strekene mellom relasjonen og entitetstypene kalles *deltakelser*

Øvre skranker



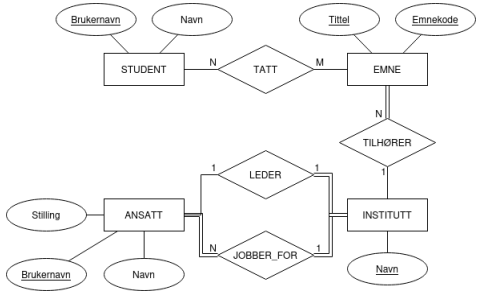
- Et viktig aspekt ved relasjoner er hvor mange entiteter en entitet kan være relatert til
- Vi skiller da kun på om det er *én* eller *mange* (altså potensielt flere enn 1)
- *Mange* representeres med en bokstav, slik som *N* eller *M*
- Tallet/bokstaven lengst vekk fra en entitet sier hvor mange den entiteten høyst kan være relatert til
- Disse kalles øvre skranker

Nedre skranker



- Tilsvarende, er et viktig aspekt ved relasjoner hvor mange entiteter en entitet må være relatert til
- Vi skiller da kun på om det er *minst én* eller *minst null*
- *Minst null* representeres med enkel linje, kalles *partsiell deltakelse*
- *Minst én* representeres med dobbel linje, kalles *total deltakelse*
- Streken nærmest en entitet sier antall den entiteten minst må være relatert til
- “Nøyaktig én” representeres med total deltakelse og 1

Attributter på relasjoner

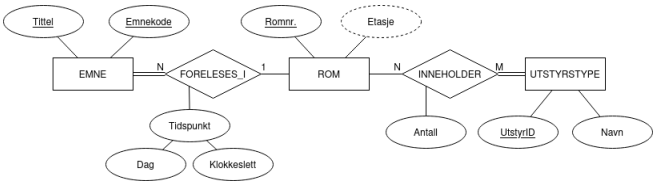


- I likhet med entitetstyper kan også relasjoner ha attributter
- Disse gir verdier som knyttes til forholdet mellom entitetene
- Kan her ikke ha nøkler

Eksempel

Lag en modell som modellerer følgende:

- Et rom har et unikt romnummer, samt en etasje (etasjen rommet befinner seg på), men dette kan utledes fra romnummeret
- En utstyrstype har en unik utstyrsID og et navn (slik som “tavle”, “prosjektor”, osv.)
- Et rom kan inneholde mange utstyr (altså flere typer utstyr) og en utstyrstype kan være inneholdt i mange rom, men vi lagrer kun de utstyrstypene som er inneholdt i minst ett rom
- Et rom kan inneholde flere ting av samme utstyrstype, og vi ønsker også å lagre dette antallet
- Et emne (likt som tidligere i denne videoen) foreleses i et rom på et fast ukentlig tidspunkt bestående av dag og klokkeslett
- Et emne foreleses i nøyaktig ett rom (altså minst og høyst ett), men et rom kan ha mange emner som foreleses i det rommet, og et rom trenger ikke brukes til forelesninger



Takk for nå!

Neste video handler om svake entiteter.

IN2090 – Databaser og datamodellering

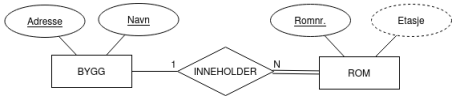
03 – Datamodellering: Svake entiteter

Leif Harald Karlsen
leifhka@ifi.uio.no



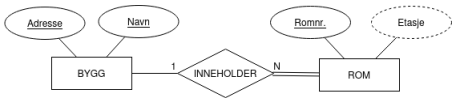
Universitetet i Oslo

Motiverende eksempel



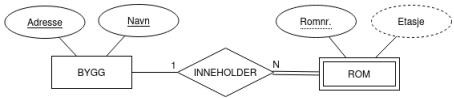
- Ønsker å utvide modellen til også å ha med bygg
- Men et rom har kun unikt romnummer innad i sitt bygg
- Kan forsøke å legge dette til nøkkelen til rom
- Ikke klart fra modellen at “Bygg”-attributten egentlig er bygget rommet er inneholdt i
- Veldig viktig å holde styr på identitet!

Svake entiteter



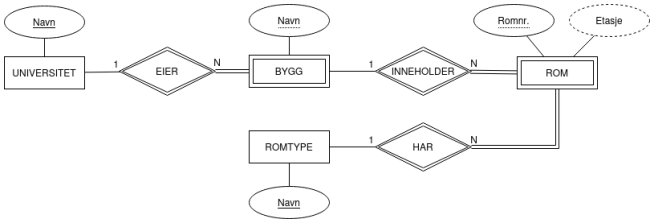
- En svak entitet er en entitet som har en nøkkel som avhenger av en annen entitets nøkkel
- En svak entitet har en nøkkel som kun er unik i en kontekst (gitt ved en relasjon til en annen entitet)
- Egen notasjon for dette i ER: Stiplet nøkkel-markering og dobbel boks

Identifiserende entitet og relasjon



- Men vi må også markere *hvilken* relasjon det er som angir denne konteksten
- Gjøres med dobbel diamant og kalles *identifiserende relasjon/entitet*
- Nødvendig fordi den svake entiteten kan være relatert til mange andre entiteter
- Merk: Svake entiteter må alltid være relatert til *nøyaktig én* via den identifiserende relasjonen

Flere identifiserende entiteter

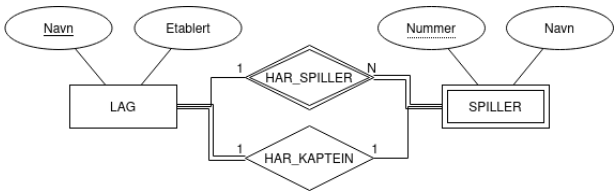


- En svak entitet kan også avhenge av flere enn én annen entitet
- Kan også ha flere nivåer med svake entiteter
- Her vil ROM ha en total nøkkel bestående av
  - UNIVERSITETets Navn
  - BYGGets Navn
  - ROMTYPEns Navn
  - ROMets Romnr.

Eksempel

Lag en modell som modellerer følgende:

- Et lag har et unikt navn og et år laget ble etablert
- En spiller spiller på nøyaktig ett lag, men et lag kan ha mange spillere (men trenger ikke ha noen)
- Et spiller har et navn, og et draktnummer som er unikt innad i laget spilleren spiller på
- Et lag kan også ha en kaptein, og et lag har nøyaktig én kaptein og en spiller kan høyst være kaptein for ett lag (men må ikke være kaptein for noe lag)



Takk for nå!

Neste uke handler om mer avansert ER og realisering.

IN2090 – Databaser og datamodellering

04 – Datamodellering: Ternære relasjoner

Leif Harald Karlsen
leifhka@ifi.uio.no

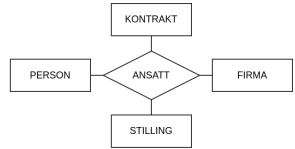


Universitetet i Oslo

Ternær og n-ær relasjon?

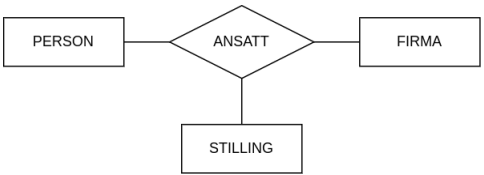






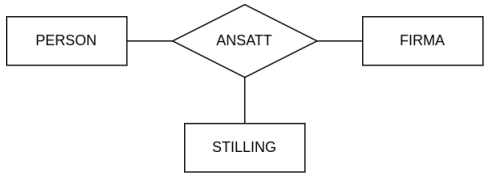
- Hittil har vi kun sett binære relasjoner
- En ternær relasjon er en relasjon som relaterer 3 entiteter av gangen
- N-ær relasjon relaterer N entiteter av gangen (for et tall N)
  - F.eks. $N = 4$/kvadr gir kvadrær-relasjon
- Uvanlige, men nyttige i noen sammenhenger
- Fokuserer på ternær, men prinsippene er like for $N > 3$

Ternære relasjoner



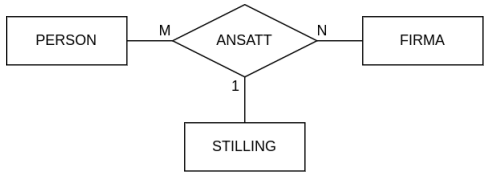
- Relaterer altså 3 entiteter
- Må alltid relatere 3 entiteter
- Ellers likt som binære
  - Kombinasjonen av de relaterte elementene er unike
  - Kan ha attributter på relasjonen

Ternære relasjoner: Øvre skranker



- Øvre skranker sier hvor mange entiteter vi kan ha av én entitetstype, gitt én kombinasjon av entiteter fra de andre entitetstypene
- F.eks. gitt én PERSON og en STILLING, hvor mange FIRMA kan vi da ha?
- La oss anta følgende:
  - Gitt én PERSON og ett FIRMA, så kan vi ha maks én STILLING
  - Gitt én PERSON og én STILLING, så kan vi ha mange FIRMAer
  - Gitt én STILLING og ett FIRMA, så kan vi ha mange PERSONer

Ternære relasjoner: Nedre skranker

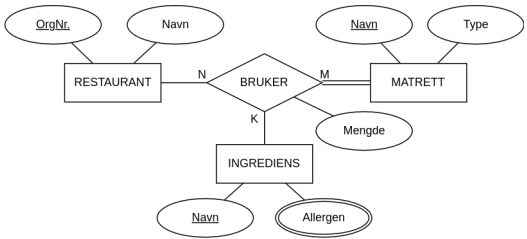


- Nedre skranker sier hvor mange entiteter vi må ha av én entitetstype i relasjonen
- F.eks. må alle personer være ansatt (med en stilling i et firma), eller ikke?
- La oss anta følgende:
  - PERSONer må ikke være ansatt (i noen firma i en stilling)
  - FIRMAER må derimot ha minst én ansatt i en stilling
  - En STILLING trenger derimot ikke ha noen ansatte (i noe firma)

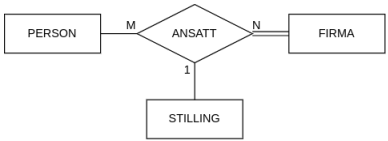
Eksempel

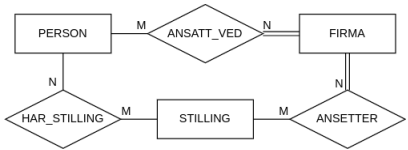
Lag en modell som modellerer følgende:

- En restaurant har et unikt organisasjonsnummer og et navn
- En matrett har et unikt navn og en type (f.eks. dessert, hovedrett, forrett, osv.)
- En ingrediens har et unikt navn, samt en mengde allergener (f.eks. gluten, soya, osv.)
- En restaurant bruker ingredienser i matretter, hvor hver matrett må inneholde minst én ingrediens på minst én restaurant, og:
  - Gitt en matrett og en ingrediens kan det være mange (N) restauranter;
  - gitt en ingrediens og en restaurant kan det være mange (M) matretter;
  - og gitt en restaurant og en matrett kan det være mange (K) ingredienser
- Vi ønsker å lagre mengden (gram) av en ingrediens en restaurant bruker i en matrett



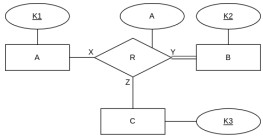
Binær vs. ternær

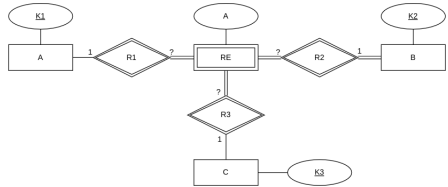


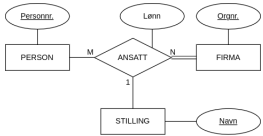


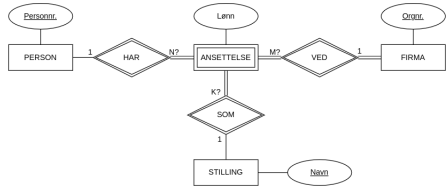
- Ternærrelasjoner er ikke ekvivalente med 3 binære
- De to modellene uttrykker forskjellige ting
- I modellen til høyre kan vi f.eks. ikke vite hvilken stilling en person har i et firma
- Også forskjellig hvilke skranker vi kan sette
- Må velge det som passer med informasjonen man ønsker å uttrykke

Representasjoner av relasjoner









- Merk at binærrelasjoner kan representeres med en svak entitet
- Kan ikke sette ekvivalente øvre skranker med entitets-representasjonen
- Kan kun representere N-M-K-relasjoner med svak entitet
- Nedre skranker kan derimot representeres riktig

Takk for nå!

Neste video handler om realisering av ER-modeller.

IN2090 – Databaser og datamodellering

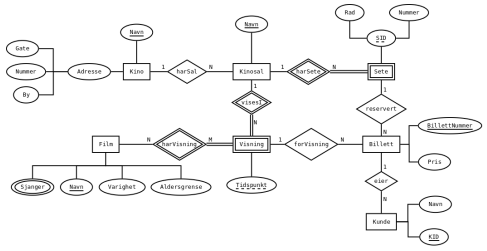
04 – Datamodellering: Realisering

Leif Harald Karlsen
leifhka@ifi.uio.no

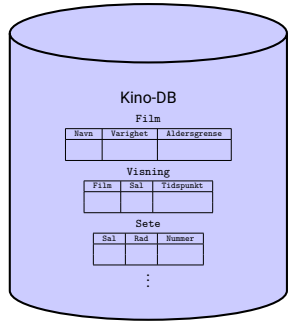


Universitetet i Oslo

Realisering

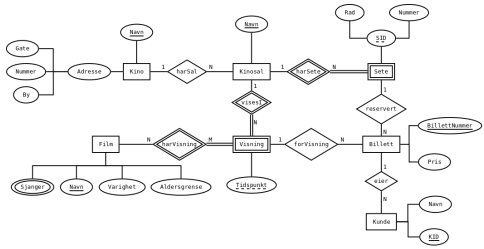




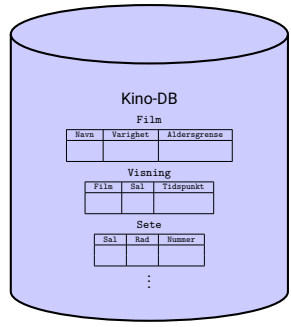


- Realisering er en prosess/algoritme som oversetter en ER-model til et relasjonsskjema
- Prosessesn sørger for at relasjonskjemaet:
  - fanger samme informasjon som ER-modellen
  - har god struktur
  - gjør det tydelig hvilke valg man kan ta uten å ødelegge overnevnte

Realiseringsalgoritmen

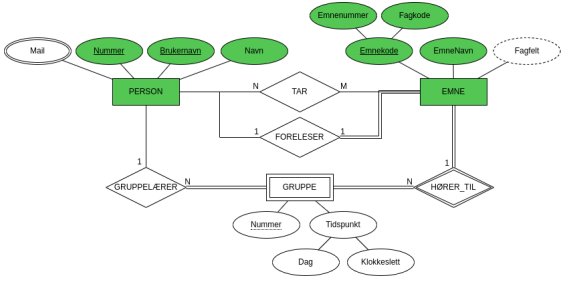






- Krever mange valg og ekstra input fra mennesker
- Vi gjør derfor prosessen manuelt
- I valgene ønsker vi å minimere antall relasjoner og antall `NULL`-verdier
- Prosessen oversetter de ulike delene av en ER-model i en bestemt rekkefølge
- Rekkefølge: Entiteter, svake entiteter, 1-1-relasjoner, 1-N-relasjoner, M-N-relasjoner, flerverdi-attributter, ternære/N-ære relasjoner

Realisering av (normale) entiteter

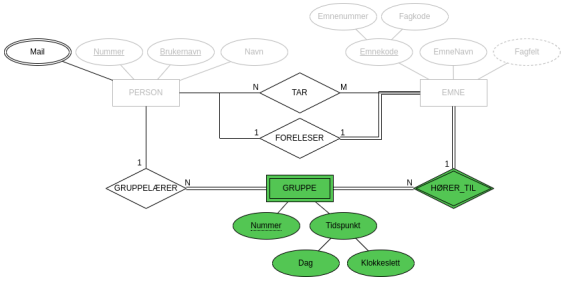



```
Person(nummer, brukernavn, navn)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}

Emne(emnenummer, fagkode, emnenavn)
- KN/PN: {emnenummer, fagkode}
```

- Entiteter blir relasjoner
- Ignorerer utledbare og venter med flerverdi
- Attributter blir relasjons-attributter (kun løv-noder for sammensatte)
- Nøkler blir kandidatnøkler (KN), må velge én primærnøkkel (PN)

Realisering av svake entiteter



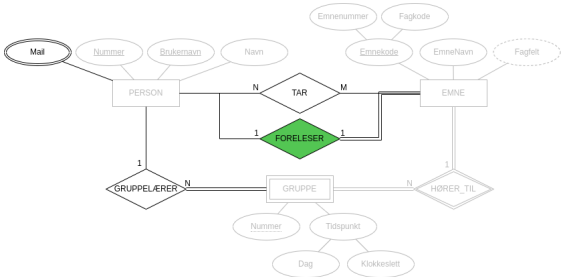

```
Person(nummer, brukernavn, navn)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}

Emne(emnenummer, fagkode, emnenavn)
- KN/PN: {emnenummer, fagkode}

Gruppe(nummer, dag, klokkeslett, emnenummer, fagkode)
- KN/PN: {nummer, emnenummer, fagkode}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
```

- Svake entiteter blir også relasjoner
- Får også PN til identifiserende entitet(er)s relasjon(er) som attributter
- PN blir identifiserendes PN + svak nøkkel
- Realiserer så den identifiserende relasjonen(e) via FN

Realisering av 1-1-relasjoner



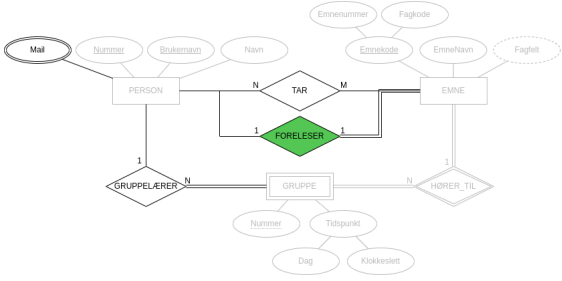

```
Person(nummer, brukernavn, navn)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}

Emne(emnenummer, fagkode, emnenavn)
- KN/PN: {emnenummer, fagkode}

Gruppe(nummer, dag, klokkeslett, emnenummer, fagkode)
- KN/PN: {nummer, emnenummer, fagkode}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
```

- Fire alternativer
  1. Sammensmeltning (merge)
  2. FN fra Person
  3. FN fra Emne
  4. Ny relasjon

Realisering av 1-1-relasjoner: Sammensmeltning

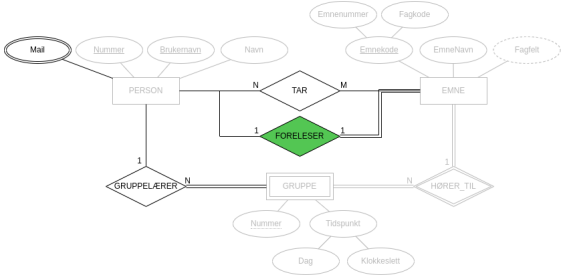



```
Gruppe(nummer, dag, klokkeslett, brukernavn)
- KN/PN: {nummer, brukernavn}
- FN: (brukernavn) -> Foreleser(brukernavn)

Foreleser(nummer, brukernavn, navn,
          emnenummer, fagkode, emnenavn)
- KN: {nummer}, {brukernavn}, {emnenummer, fagkode}
- PN: {brukernavn}
```

- Lager en ny relasjon som er sammensmeltningen av Person og Emne
- Får da alle deres attributter (må velge PN)
- Fjerner de gamle (og oppdaterer FNer)
- Kan kun brukes hvis begge deltakelsene er totale

Realisering av 1-1-relasjoner: FN fra Person



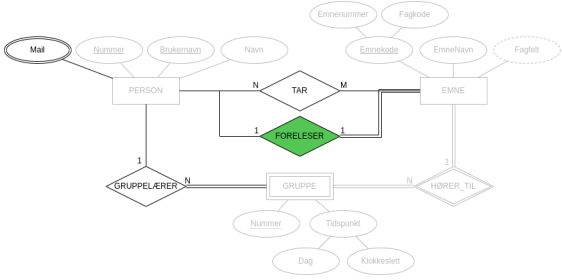

```
Person(nummer, brukernavn, navn, emnenummer, fagkode)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)

Emne(emnenummer, fagkode, emnenavn)
- KN/PN: {emnenummer, fagkode}

Gruppe(nummer, dag, klokkeslett, emnenummer, fagkode)
- KN/PN: {nummer, emnenummer, fagkode}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
```

- Legger til FN fra Person til Emnes PN
  1. Legger til attributter tilsvarende Emnes PN
  2. Markerer dem som FN
  3. Ville blitt KN hvis PERSON hadde total deltakelse

Realisering av 1-1-relasjoner: FN fra Emne



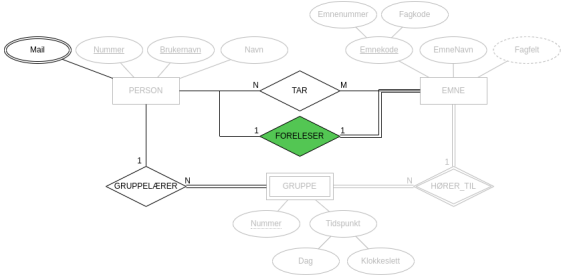

```
Person(nummer, brukernavn, navn)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}

Emne(emnenummer, fagkode, emnenavn, foreleser)
- KN/PN: {emnenummer, fagkode}, {foreleser}
- FN: (foreleser) -> Person(brukernavn)

Gruppe(nummer, dag, klokkeslett, emnenummer, fagkode)
- KN/PN: {nummer, emnenummer, fagkode}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
```

- Legger til FN fra Emne til Persons PN
  1. Legger til attributter tilsvarende Persons PN
  2. Markerer dem som FN
  3. Blir KN siden EMNE har total deltakelse

Realisering av 1-1-relasjoner: Ny relasjon




```
Person(nummer, brukernavn, navn)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}

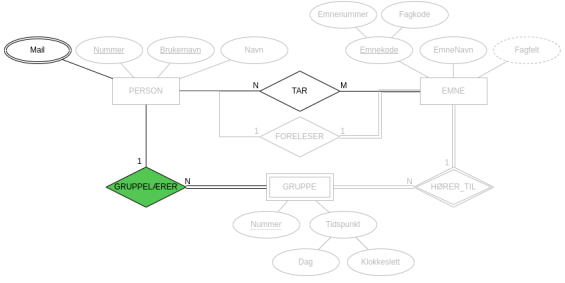
Emne(emnenummer, fagkode, emnenavn)
- KN/PN: {emnenummer, fagkode}

Gruppe(nummer, dag, klokkeslett, emnenummer, fagkode)
- KN/PN: {nummer, emnenummer, fagkode}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)

Foreleser(brukernavn, emnenummer, fagkode)
- KN: {brukernavn}, {emnenummer, fagkode}
- PN: {brukernavn}
- FN: (brukernavn) -> Person(brukeravn)
      (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
```

- Lager ny relasjon med referenser til entitetenes relasjoners PNer
- Får KN lik hver av disse PNene (og velger PN)
- Legger til FNer til disse PNene

Realisering av 1-N-relasjoner



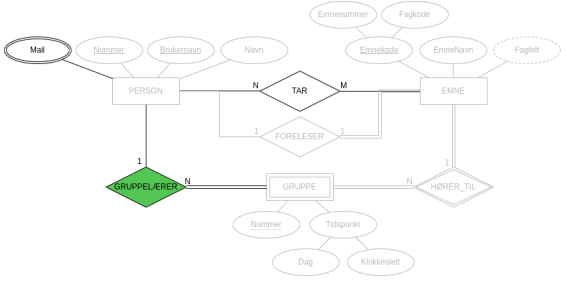

```
Person(nummer, brukernavn, navn)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}

Emne(emnenummer, fagkode, emnenavn, foreleser)
- KN/PN: {emnenummer, fagkode}, {foreleser}
- FN: (foreleser) -> Person(brukernavn)

Gruppe(nummer, dag, klokkeslett, emnenummer, fagkode)
- KN/PN: {nummer, emnenummer, fagkode}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
```

- To valg:
  1. Legge FN inn i N-siden (GRUPPE)
  2. Ny relasjon

Realisering av 1-N-relasjoner: FN i N-siden



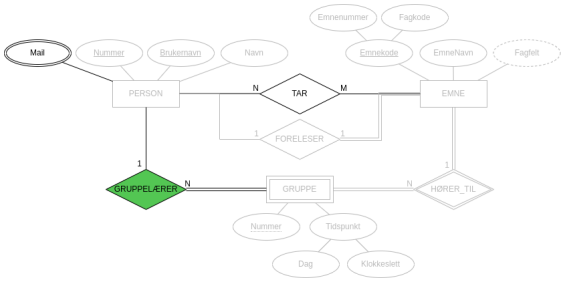

```
Person(nummer, brukernavn, navn)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}

Emne(emnenummer, fagkode, emnenavn, foreleser)
- KN/PN: {emnenummer, fagkode}, {foreleser}
- FN: (foreleser) -> Person(brukernavn)

Gruppe(nummer, dag, klokkeslett, emnenummer, fagkode, grlærer)
- KN/PN: {nummer, emnenummer, fagkode}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
      (grlærer) -> Person(brukernavn)
```

- Tilsvarende som FN for 1-1

Realisering av 1-N-relasjoner: Ny relasjon




```
Person(nummer, brukernavn, navn)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}

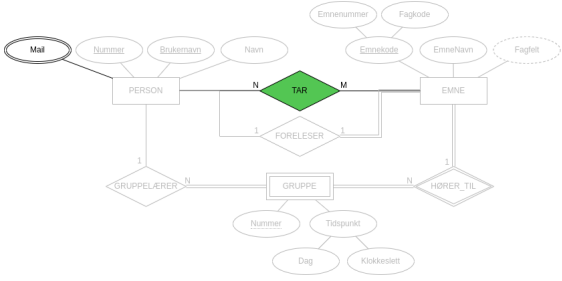
Emne(emnenummer, fagkode, emnenavn, foreleser)
- KN/PN: {emnenummer, fagkode}, {foreleser}
- FN: (foreleser) -> Person(brukernavn)

Gruppe(nummer, dag, klokkeslett, emnenummer, fagkode)
- KN/PN: {nummer, emnenummer, fagkode}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)

Gruppelærer(brukernavn, nummer, emnenummer, fagkode)
- KN/PN: {nummer, emnenummer, fagkode}
- FK: (brukernavn) -> Person(brukernavn)
      (nummer, emnenummer, fagkode) ->
          Gruppe(nummer, emnenummer, fagkode)
```

- Tilsvarende som for 1-1
- N-sidens atributter blir PN

Realisering av N-M-relasjoner




```
Person(nummer, brukernavn, navn)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}

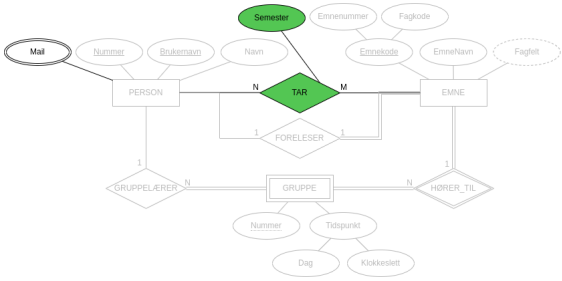
Emne(emnenummer, fagkode, emnenavn, foreleser)
- KN/PN: {emnenummer, fagkode}, {foreleser}
- FN: (foreleser) -> Person(brukernavn)

Gruppe(nummer, dag, klokkeslett, emnenummer, fagkode, grlærer)
- KN: {nummer, emnenummer, fagkode}, {grlærer}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
      (grlærer) -> Person(brukernavn)

Tar(brukernavn, emnenummer, fagkode)
- KN/PN: {brukernavn, emnenummer, fagkode}
- FN: (brukernavn) -> Person(brukernavn)
      (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
```

- Bare ett valg: Ny relasjon
- Tilsvarende 1-1 og 1-N
- PN lik alle attributter

Realisering av relasjoner med attributter




```
Person(nummer, brukernavn, navn)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}

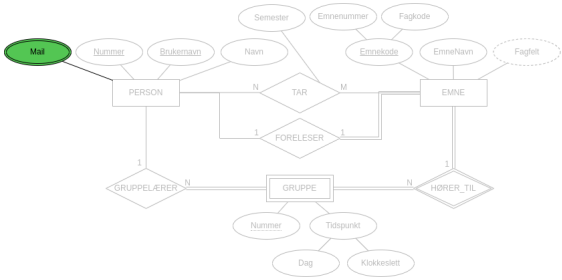
Emne(emnenummer, fagkode, emnenavn, foreleser)
- KN/PN: {emnenummer, fagkode}, {foreleser}
- FN: (foreleser) -> Person(brukernavn)

Gruppe(nummer, dag, klokkeslett, emnenummer, fagkode, grlærer)
- KN: {nummer, emnenummer, fagkode}, {grlærer}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
      (grlærer) -> Person(brukernavn)

Tar(brukernavn, emnenummer, fagkode, semester)
- KN/PN: {brukernavn, emnenummer, fagkode}
- FN: (brukernavn) -> Person(brukernavn)
      (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
```

- Attributter på relasjoner blir bare attributter i relasjonens realisering
- Tilsvarende som for entiteter
- Merk: Blir ikke del av KN/PN!

Realisering av flerverdi-attributter



- Blir egen relasjon som relaterer entitetens relasjons PN til attributten
- KN/PN lik alle attributtene i relasjonen
- FN til entitetens relasjons PN

```
Person(nummer, brukernavn, navn)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}

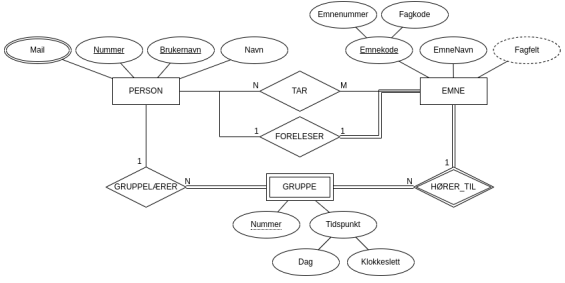
Emne(emnenummer, fagkode, emnenavn, foreleser)
- KN/PN: {emnenummer, fagkode}, {foreleser}
- FN: (foreleser) -> Person(brukernavn)

Gruppe(nummer, dag, klokkeslett, emnenummer, fagkode, grlærer)
- KN: {nummer, emnenummer, fagkode}, {grlærer}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
      (grlærer) -> Person(brukernavn)

Tar(brukernavn, emnenummer, fagkode, semester)
- KN/PN: {brukernavn, emnenummer, fagkode}
- FN: (brukernavn) -> Person(brukernavn)
      (emnenummer, fagkode) -> Emne(emnenummer, fagkode)

Mail(brukernavn, mail)
- KN/PN: {brukernavn, mail}
- FN: (brukernavn) -> Person(brukernavn)
```

Full realisering




```
Person(nummer, brukernavn, navn)
- KN: {nummer}, {brukernavn}
- PN: {brukernavn}

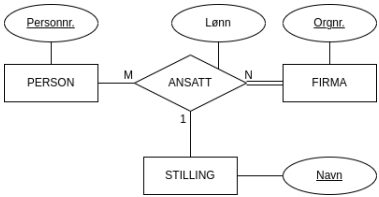
Emne(emnenummer, fagkode, emnenavn, foreleser)
- KN/PN: {emnenummer, fagkode}, {foreleser}
- FN: (foreleser) -> Person(brukernavn)

Gruppe(nummer, dag, klokkeslett, emnenummer, fagkode, grlærer)
- KN: {nummer, emnenummer, fagkode}, {grlærer}
- FN: (emnenummer, fagkode) -> Emne(emnenummer, fagkode)
      (grlærer) -> Person(brukernavn)

Tar(brukernavn, emnenummer, fagkode, semester)
- KN/PN: {brukernavn, emnenummer, fagkode}
- FN: (brukernavn) -> Person(brukernavn)
      (emnenummer, fagkode) -> Emne(emnenummer, fagkode)

Mail(brukernavn, mail)
- KN/PN: {brukernavn, mail}
- FN: (brukernavn) -> Person(brukernavn)
```

Realisering av ternære relasjoner



- Tilsvarende som for M-N-relasjoner (uavhengig av kardinaliteter)
- Altså, alltid egen relasjon
- Må sette KN/PN i henhold til kardinalitetene
- Helt likt for alle N-ære relasjoner

```
Person(personnr)
- KN/PN: {personnr}

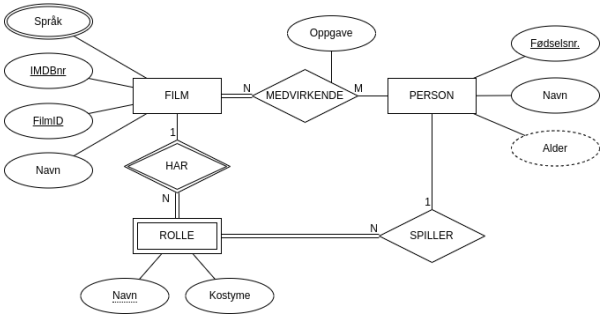
Firma(orgnnr)
- KN/PN: {orgnr}

Stilling(navn)
- KN/PN: {navn}

Ansatt(person, firma, stilling)
- KN/PN: {person, firma}
- FN: (person) -> Person(personnr)
      (firma) -> Firma(orgnr)
      (stilling) -> Stilling(navn)
```

Eksempel

Realiser modellen til et relasjonsskjema




```
Film(imdb_nr, film_id, navn)
- KN: {imdb_id}, {film_id}
- PN: {film_id}

Person(fødselsnr, navn)
- KN/PN: {fødselsnr}

Rolle(navn, film, kostyme, skuespiller)
- KN/PN: {navn, film}
- FN: (film) -> Film(film_id)
      (skuespiller) -> Person(fødselsnr)

Medvirkende(film, person, oppgave)
- KN/PN: {film, person}
- FN: (film) -> Film(film_id)
      (person) -> Person(fødselsnr)

Språk(film, språk)
- KN/PN: {film, språk}
- FN: (film) -> Film(film_id)
```

Ferdigstilling av databaseskjemaet

- Etter realiseringen har vi fått strukturen på relasjonene våre
- Må så legge til typer til attributtene
- Kan også legge til andre skranker (kommer til disse litt senere i semesteret)
- Kan så gjøre forbedringer, omstrukutering, osv. om man ser muligheter for forbedringer
- Til slutt kan man skrive SQL-kommandoer som oppretter skjemaet i et faktisk databasesystem (mer om dette om et par uker)

```
Film(imdb_id int, film_id int, navn text)
- KN: {imdb_id}, {film_id}
- PN: {film_id}

Person(fødselsnr text, navn text)
- KN/PN: {fødselsnr}

Rolle(navn text, film int, kostyme text, skuespiller text)
- KN/PN: {navn, film}
- FN: (film) -> Film(film_id)
      (skuespiller) -> Person(fødselsnr)

Medvirkende(film int, person text, oppgave text)
- KN/PN: {film, person}
- FN: (film) -> Film(film_id)
      (person) -> Person(fødselsnr)

Språk(film int, språk text)
- KN/PN: {film, språk}
- FN: (film) -> Film(fim_id)
```

Takk for nå!

Neste uke handler om grunnleggende SQL.

IN2090 – Databaser og datamodellering

05 – Intro til SQL

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

SQL: Structured Query Language

- SQL er et spørrespråk for relasjonelle databaser
- Det mest brukte spørrespråket for slike databaser
- Brukes for å formulere spørringer, altså spørsmål, til en database
- SQL kan også brukes for å manipulere en database
  - Lage tabeller
  - sette inn data
  - slette data
  - ...
- Ble laget i 1974, men ble først standardisert i 1986

Imperativ vs. deklarativ

La oss si at du er en tørst og din mor er i nærheten. To måter å få henne til å hente vann på:

- Imperativ:
  - “Hei mamma, kan du gå 2 meter til venstre, strekke ut armen din, trekke dørhåndtaket ned og mot deg. Så gå gjennom døren, snu deg til venstre, gå 4 meter frem, snu deg til høyre, ..., og sette glasset ned på bordet og slippe det.”
- Deklarativt:
  - “Hei mamma, vann er flytende H_2O og glass er smeltet sand formet på en slik måte at dets innhold ikke renner ut. Kan du hente meg et glass med vann, er du snill?

Python/Java vs. SQL

- Programmeringsspråk (f.eks. Python og Java) er imperative språk, altså presise språk for å utrrykke *sekvenser av instruksjoner for en datamaskin*
- F.eks.:
  - "Sett verdien av *x* til 2" (`x = 2`)
  - "Legg tallet 5 til listen *lst*" (`lst.add(5)`)
  - "For hvert element i listen *L* print verdien av elementet" (`for e in L: print(e)`)

- Et spørrespråk er et presist språk for å uttrykke *spørsmål til en database*
- Slike spørsmål kalles ofte en *spørring* (eng.: *query*)
- SQL er deklartivt, f.eks.:
  - "Finn alle elementer som har et navn som starter på 'P'?"
  - "La 'Forelder' være alle elementer som har en 'harBarn'-relasjon til et element"
  - "Finn antall ansatte som har en sjef som tjener mer enn 1000000 KR?"

Typer SQL-spørringer

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)

De første SQL-forelesningene omhandler kun `SELECT`.

SELECT-spørringer

- (Enkle) SELECT-spørringer har formen:

```
SELECT <kolonner>
  FROM <tabeller>
```

- hvor <kolonner> er en liste med kolonne-navn,
- og <tabeller> er en liste med tabell-navn

Resultatet av en SELECT-spørring er alltid en ny tabell, som består av

- kolonnene i <kolonner>
- basert på radene i tabellene i <tabeller>

Velge en enkelt kolonne

Spørring som henter ut alle navn i `Customer`-tabellen

```
SELECT Name
  FROM Customer
```

Resultat

CustomerID	Name	Birthdate	NrProducts
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

Velge flere kolonner

Spørring som henter alle navn -og fødselsdato-par i `Customer`-tabellen

```
SELECT Name, Birthdate
  FROM Customer
```

Resultat

CustomerID	Name	Birthdate	NrProducts
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

Velge alle kolonner

Spørring som henter alle kolonnene i `Customer`-tabellen

```
SELECT *
  FROM Customer
```

Resultat

CustomerID	Name	Birthdate	NrProducts
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

Legge inn filter via `WHERE`

- Ofte er vi kun interessert i spesifikke rader
- Vi kan da bruke en `WHERE`-klausul for å velge ut de radene vi ønsker
- SQL-spørringer har da formen

```
SELECT <kolonner>
  FROM <tabeller>
 WHERE <betingelse>
```

- `<betingelse>` er et uttrykk over kolonnenavnene fra tabellene
- For hver rad evalueres dette uttrykket til sant eller usant
- Resultatet er det samme som før, men begrenset til kun de radene som gjør `<betingelse>` sann

Velge ut spesifikke rader

Spørring som gir fødselsdatoen til kunden ved navn John Mill

```
SELECT Birthdate
  FROM Customer
 WHERE Name = 'John Mill'
```

Resultat

CustomerID	Name	Birthdate	NrProducts
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

SQL og relasjonsalgebra: Oversettelse

- En SQL spørring og relasjonsalgebraen har mye til felles
- En SQL-spørring kan oversettes til relasjonsalgebra
- For eksempel kan de enkle SQL-spørringene vi nå har sett oversettes slik:

```
SELECT <kolonner>
  FROM <tabeller>
 WHERE <uttrykk>
```

$\Longrightarrow$

$\pi_{\text{<kolonner>}}(\sigma_{\text{<uttrykk>}}(\text{<tabeller>}))$

SQL og relasjonsalgebra: Forskjeller

- Men i den relasjonsmodellen er relasjonene mengder av tupler
- I en mengde kan et element kun forekomme én gang, f.eks.:

Person

Navn	Alder
Per	13
Ola	24
Mari	13
Karl	25
Ida	25

$\pi_{\text{Alder}}(\text{Person})$

Alder
13
24
25

- I SQL har vi tabeller i stedet for relasjoner (multi-mengder av tupler):

```
SELECT Alder
  FROM Person
```

$\Rightarrow$

Alder
13
24
13
25
25

- Dette trenger vi for aggregering (sum, gjennomsnitt, osv.) av kolonner

SQL og syntaks

SQL bryr seg ikke om indent og linjeskift (slik som f.eks. Python), så

```
SELECT Birthdate
  FROM Customers
 WHERE NrProducts > 5

SELECT Birthdate FROM Customers
 WHERE NrProducts > 5

SELECT Birthdate
FROM Customers WHERE NrProducts > 5

SELECT Birthdate FROM Customers WHERE NrProducts > 5
```

er alle lov og representerer den samme spørringen.

SQL og bokstavering

- For SQL-nøkkelord og navn på tabeller og kolonner er SQL versalinsensitiv (eng.: *case-insensitive*)
- Altså, SQL skiller ikke mellom store og små bokstaver
- Så
  - `SELECT Name FROM Customers`
  - `select name from customers`

  er ekvivalente spørringer
- Men, SQL skiller på store og små bokstaver på verdier
  - så `'London'` og `'london'` er to forskjellige verdier
- Bruk `--` (to bindestreker) for kommentarer (blir ignorert av databasen), f.eks.

```
SELECT Name --Dette er en kommentar
  FROM Customers
```

SQL og skjema

- Tabellnavn kan i `FROM`-klausulen bli prefiksert med et skjemanavn, for eksempel:
- gitt et skjema `UiO` som inneholder tabell `Students`,
- så vil vi skrive `UiO.Students` i SQL

```
SELECT Name
  FROM UiO.Students
```

- Skjemaet `public` finnes automatisk i alle databaser og er standard skjemaet
- Om man ikke spesifiserer et skjema er det dette som brukes, så

```
SELECT Name FROM Person
```

blir

```
SELECT Name FROM public.Person
```

Takk for nå!

Neste video vil se på litt mer avanserte `WHERE`-klausuler.

IN2090 – Databaser og datamodellering

05 – `WHERE`-klausulen

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Velge over et intervall av verdier

Spørring som finner navnet på alle kunder som har kjøpt mer enn 10 produkter

```
SELECT Name
  FROM Customer
 WHERE NrProducts > 10
```

Resultat

CustomerID	Name	Birthdate	NrProducts
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

Kombinere betingelser

Spørring som finner fødselsdatoen og navnet til kunder som kjøpte mellom 4 og 10 produkter

```
SELECT Birthdate, Name
  FROM Customer
 WHERE NrProducts > 4 AND
       NrProducts < 10
```

Resultat

CustomerID	Name	Birthdate	NrProducts
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

Kombinere betingelser med OR

Spørring som finner navnet til kunder som har kjøpt færre enn 5 produkter eller fler enn 15 produkter

```
SELECT Name
  FROM Customer
 WHERE NrProducts < 5 OR
       NrProducts > 15
```

Resultat

CustomerID	Name	Birthdate	NrProducts
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

Bruke både AND og OR

Spørring som finner navn på kunder som har kjøpt mindre enn 5 eller mer enn 15 produkter og er født etter '2000-01-01'

```
SELECT Name FROM Customer
 WHERE (NrProducts < 5 OR
        NrProducts > 15) AND
       Birthdate > '2000-01-01'
```

Resultat

CustomerID	Name	Birthdate	NrProducts
0	Anna Consuma	1978-10-09	19
1	Peter Young	2009-03-01	1
2	Carla Smith	1986-06-14	8
3	Sam Penny	1961-01-09	14
4	John Mill	1989-11-16	8
5	Yvonne Potter	1971-04-12	6

Velge TVer

Spørring som henter navnet, merket og pris på 48 og 50 tommer TVer

```
SELECT Name, Brand, Price
  FROM Product
 WHERE Name = 'TV 50 inch' OR
       Name = 'TV 48 inch'
```

Resultat

ProductID	Name	Brand	Price	Stock
0	TV 50 inch	Sony	8999	29
1	Laptop 2.5GHz	Lenovo	7499	12
2	Laptop 8GB RAM	HP	6999	80
3	Speaker 500	Bose	4999	42
4	TV 48 inch	Panasonic	11999	31
5	Phone S6	IPhone	5195	65

Søke i tekst

- Med det vi har lært hittil har vi ingen måte å spørre etter alle TVer
  - (altså alle produkter som har navn som starter med `'TV'`)
- Vi kan kun bruke likhet, ingen måte å søke i tekst
- Dette kan gjøres med SQLs `LIKE`
- Kan så bruke `'%'` som “wildcard“ som matcher alt

LIKE

For eksempel:

- `Name LIKE 'TV%'`
  - Sant for alle `Name`-verdier som starter med `'TV'`
  - f.eks. `'TV 50 inch'` og `'TVSHOW'`
  - men ikke f.eks. `'hello'` eller `'MTV'`
- `Name LIKE '%TV'`
  - sant for alle `Name`-verdier som slutter med `'TV'`
  - f.eks. `'50 inch TV'` og `'MTV'`
  - men ikke f.eks. `'TV2'` eller `'Fun TV program'`
- `Name LIKE '%TV%'`
  - sant for alle `Name`-verdier som inneholder `'TV'` (hvor som helst)
  - f.eks. `'50 inch TV'` og `'Fun TV program'`
  - men ikke f.eks. `'T2V'` eller `'hello'`
- `Name LIKE '%TV%inch'`
  - sant for alle `Name`-verdier som inneholder `'TV'` og slutter med `'inch'`
  - f.eks. `'TV 50 inch'` og `'Fun TV program pinch'`
  - men ikke f.eks. `'TV 50 inches'` eller `'50 inch TV'`

Velge TVer med `LIKE`

Spørring som finner navn, pris og merke på alle TVer

```
SELECT Name, Brand, Price
  FROM Product
 WHERE Name LIKE 'TV%'
```

Resultat

ProductID	Name	Brand	Price	Stock
0	TV 50 inch	Sony	8999	29
1	Laptop 2.5GHz	Lenovo	7499	12
2	Laptop 8GB RAM	HP	6999	80
3	Speaker 500	Bose	4999	42
4	TV 48 inch	Panasonic	11999	31
5	Phone S6	IPhone	5195	65

Regulære uttrykk

- `LIKE` støtter kun `%` (og `_` for wildcard enkelt karakter)
- Ønsker man komplisert matching kan man bruke `SIMILAR TO` eller `~`
- `SIMILAR TO` bruker litt rar miks av `LIKE`-syntaks (`%`) og vanlige regulære uttrykk
- F.eks. er `Name = 'abc'` et mulig svar for

```
SELECT Name
FROM Products
WHERE Name SIMILAR TO '%(b|d)%'
```

- Man kan også bruke `~` for vanlige (POSIX) regulære uttrykk
- F.eks.

```
Name ~ '.*(b|d).*'
```

  er samme som over
- `LIKE` finnes fordi den er sikrere mhp. ytelse (kan alltid eksekveres raskt)

Negasjon

- Av og til vil vi bare ha svar som *ikke* tilfredstiller et uttrykk
- Bruker da `NOT`-nøkkelordet
- For eksempel:

```
SELECT Name
  FROM Products
 WHERE NOT Description LIKE '%simple%'
```

  er sant for alle rader som ikke har order `'simple'` i sin `Description`
- Merk at
  - `NOT (E1 AND E2)` er ekvivalent med `(NOT E1) OR (NOT E2)`
  - `NOT (E1 OR E2)` er ekvivalent med `(NOT E1) AND (NOT E2)`

Null

- Når vi setter inn data vil vi av og til mangle en verdi (f.eks. fordi den er ukjent eller ikke finnes)
- For eksempel, kan det være vi ikke vet fødselsdatoen til en bestemt student
- Likevel ønsker vi å legge studenten inn i databasen slik at vi kan lagre informasjon om studenten
- Men hva skal vi sette inn?
  - Den tomme teksten? Feil type!
  - År 0? Ikke korrekt!
- For ukjente og manglende verdier har SQL `NULL`
- Så, for å sette inn studenten `Sam Penny` med ukjent fødselsdato, bruker vi `NULL`

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	?

SQL og null

- Hvordan sjekker vi om en verdi er `NULL`?
- Dersom vi prøver

```
SELECT StdName
  FROM Students
 WHERE StdBirthdate = NULL
```

  får vi ingen svar!
- Faktisk så er `NULL = NULL` ikke sant
- og heller ikke `NOT (NULL = NULL)`!
- Grunnen til dette er at `NULL` representerer en manglende eller ukjent verdi
- Så `NULL` kan potensielt representere en hvilken som helst verdi
- Så `StdBirthdate = NULL` og `NULL = NULL` er begge ukjente, altså `NULL`
- Og `NULL` er ikke `TRUE` (*sant*) så det tilfredstiller ikke `WHERE`-klausulen

Sjekke for NULLs

- For å sjekke om en verdi er `NULL` må vi bruke `IS NULL`.
- For eksempel:

```
SELECT StdName
  FROM Students
 WHERE StdBirthdate IS NULL
```

  så får vi `Sam Penny` som svar
- Vi kan også bruke `IS NOT NULL` for å sjekke at en verdi ikke er `NULL`

`NULL`s oppførsel

- Merk at `NULL` oppfører seg som *ukjent*:
  - `NULL AND TRUE` resulterer i `NULL`
  - `NULL OR FALSE` resulterer i `NULL`
  - `NULL AND FALSE` resulterer i `FALSE`
  - `NULL OR TRUE` resulterer i `TRUE`
  - `10 + NULL` resulterer i `NULL`
  - (Prøv å lese hver setning over med *ukjent* i stedet for `NULL`)
- Så resultatet av et uttrykk med `NULL` er `NULL` dersom svaret avhenger av hva `NULL` kan være

Eksempel fra Northwind-databasen

Finn navnet og prisen på alle produkter som selges i flasker eller glass og som koster mer enn 30 dollar. [4 rader]

```
SELECT product_name, unit_price
  FROM products
 WHERE (quantity_per_unit LIKE '%bottles' OR
        quantity_per_unit LIKE '%jars')
       AND unit_price > 30;
```

Takk for nå!

Neste video vil se mer om `SELECT`-klausulen.

IN2090 – Databaser og datamodellering

05 – `SELECT`-klausulen

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Uttrykk i `SELECT`

- Hittil har vi bare hentet ut data direkte fra tabeller
- Ofte ønsker man å transformere dataene før vi returnerer svaret
- Dette kan gjøres med bruke uttrykk for å manipulere verdiene i `SELECT`-klausulen
- For eksempel, for å få alle priser i NOK fremfor USD (antar at 1 USD = 8 NOK) kan vi gjøre:

```
SELECT product_name, unit_price * 8
  FROM products
```

- Eller, for å få alle kunders kontaktpersoners navn med tittel først:

```
SELECT contact_title || ' ' || contact_name
  FROM customers
```

- `||` konkatenerer strenger (f.eks. `'hel'|| 'lo'= 'hello'`)

Gi navn til kolonner

- Når vi har et uttrykk i en `SELECT`-klausul får den resulterende kolonnen ingen navn
- Vi kan gi kolonner resultat-tabellen navn ved å bruke `AS`-nøkkelordet
- F.eks.:

```
SELECT product_name, unit_price * 8 AS unit_price_nok
  FROM products
```

```
SELECT contact_title || ' ' || contact_name AS contact_person
  FROM customers
```

Dupliserte svar

- Svarene fra en spørring kan inneholde duplikater
- F.eks. dersom vi kjører

```
SELECT contact_title
  FROM customers
 WHERE contact_title LIKE '%Manager%'
```

over `northwind`-databasen får vi 33 svar:

contacttitle
Marketing Manager
Accounting Manager
Marketing Manager
Sales Manager
Accounting Manager
Marketing Manager
Marketing Manager
⋮

Fjerning av duplikater

- Duplikater er av og til uønsket
- Vi kan fjerne duplikater med `DISTINCT`-nøkkelordet i `SELECT`-klausulen
- F.eks.:

```
SELECT DISTINCT contact_title
  FROM customers
 WHERE contact_title LIKE '%Manager%'
```

gir kun 3 svar:

contacttitle
Sales Manager
Marketing Manager
Accounting Manager

Aggregering

- En aggregeringsfunksjon er en funksjon som returnerer en enkel verdi fra en samling verdier
- I SQL har vi mange aggregeringsfunksjoner, slik som `sum`, `avg`, `count`, osv.
- Disse funksjonene kan enten bli anvendt på alle verdier i en kolonne (f.eks. summere alle priser)
- eller anvendes på grupper av rader (kommer tilbake til dette om noen uker)

Aggregering: Sum

- For å summere en hel kolonne, kan vi putte `sum(<column>)` i `SELECT`-klausulen
- For eksempel, for å finne det totale antallet varer på lager kan vi summere `units_in_stock`-kolonnen i `products`-tabellen slik:

```
SELECT sum(units_in_stock) AS total_nr_products
  FROM products
```

- Tilsvarende har vi:
  - `avg` – gjennomsnitt
  - `max` – maksimum
  - `min` – minimum
  - `count` – antall rader

Kombinere aggregering og andre kolonner

- En aggregeringsfunksjon returnerer én enkel verdi
- Altså gir det ikke mening å direkte kombinere denne med andre kolonner i samme `SELECT`-klausul
- F.eks. følgende gir ikke mening:

```
SELECT product_name,                                    -- ERROR!
       sum(units_in_stock) AS total_nr_products
  FROM products
```

- Merk at man derimot kan kombinere flere aggregater i samme `SELECT`-klausul, f.eks.:

```
SELECT max(unit_price) AS highest,
       min(unit_price) AS lowest,
       max(unit_price) - min(unit_price) AS difference,
  FROM products
```

Aggregering: `Count`

- For eksempel, for å finne antall produkter som koster mer enn 20 dollar kan vi kjøre:

```
SELECT count(*) AS nr_expensive_products
  FROM products
 WHERE unit_price > 20
```

- `count(*)` teller antall rader i resultatet
- `count(product_id)` teller antall ikke-`NULL` verdier i `product_id`-kolonnen
- Merk at det kan være duplikater i svaret, og disse blir telt med
- Skal senere se hvordan man kan telle kun unike svar

Takk for nå!

Neste video vil se mer om `FROM`-klausulen og joins.

IN2090 – Databaser og datamodellering

05 – `FROM`-klausulen og joins

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Kombinere informasjon fra flere tabeller

- (Enkle) `SELECT`-spørringer har formen:

```
SELECT <kolonner>
  FROM <tabeller>
 WHERE <uttrykk>
```

- Frem til nå har vi bare sett på spørringer over én og én tabell
- Ofte ønsker vi å kombinere informasjon fra ulike tabeller
- Dette kan gjøres ved å legge til flere tabeller i `FROM`-klausulen

Spørringer over flere tabeller

Hva skjer dersom vi putter flere tabeller i `FROM`?

To tabeller i `FROM`

```
SELECT *
  FROM products, orders
```

Resultat

products

ProductID	ProductName	Price
0	TV 50 inch	8999
1	Laptop 2.5GHz	7499

orders

OrderID	OrderedProduct	Customer
0	1	John Mill
1	1	Peter Smith
2	0	Anna Consuma
3	1	Yvonne Potter

Spørringer over flere tabeller

Hva skjer dersom vi putter flere tabeller i `FROM`?

To tabeller i `FROM`

```
SELECT *
  FROM products, orders
```

Resultat – Fargekodet

products

ProductID	ProductName	Price
0	TV 50 inch	8999
1	Laptop 2.5GHz	7499

orders

OrderID	OrderedProduct	Customer
0	1	John Mill
1	1	Peter Smith
2	0	Anna Consuma
3	1	Yvonne Potter

Kryssprodukt

- Med flere tabeller i `FROM`-klausulen får vi alle mulige kombinasjoner av radene fra hver tabell
- Dette kalles *kryssproduct* eller *Kartesisk produkt*
- Altså, det som var $\times$ i relasjonsalgebraen

Kryssprodukt av to tabeller

Med to tabeller:

T1

C1	C2	C3
x1	x2	x3
y1	y2	y3

T2

D1	D2
a1	a2
b1	b2
c1	c2
d1	d2



```
SELECT * FROM T1, T2
```

C1	C2	C3	D1	D2
x1	x2	x3	a1	a1
x1	x2	x3	b1	b2
x1	x2	x3	c1	c2
x1	x2	x3	d1	d2
y1	y2	y3	a1	a2
y1	y2	y3	b1	b2
y1	y2	y3	c1	c2
y1	y2	y3	d1	d2

Kryssproduktet av tre tabeller

Med tre tabeller:

T1

C1	C2	C3
x1	x2	x3
y1	y2	y3

T2

D1	D2
a1	a2
b1	b2
c1	c2
d1	d2

T3

E1	E2
n1	n2
m1	m2



SELECT * FROM T1, T2, T3

C1	C2	C3	D1	D2	E1	E2
x1	x2	x3	a1	a1	n1	n2
x1	x2	x3	a1	a1	m1	m2
x1	x2	x3	b1	b2	n1	n2
x1	x2	x3	b1	b2	m1	m2
x1	x2	x3	c1	c2	n1	n2
x1	x2	x3	c1	c2	m1	m2
x1	x2	x3	d1	d2	n1	n2
x1	x2	x3	d1	d2	m1	m2
y1	y2	y3	a1	a2	n1	n2
y1	y2	y3	a1	a2	m1	m2
y1	y2	y3	b1	b2	n1	n2
y1	y2	y3	b1	b2	m1	m2
y1	y2	y3	c1	c2	n1	n2
y1	y2	y3	c1	c2	m1	m2
y1	y2	y3	d1	d2	n1	n2
y1	y2	y3	d1	d2	m1	m2

Hvorfor er dette nyttig?

- Kryssproduktet lar oss relatere en hvilken som helst verdi i en kolonne i en tabell til en hvilken som helst verdi i en kolonne i en annen tabell
- Ved å bruke `WHERE` -og `SELECT`-klausulene kan vi velge ut hva vi ønsker fra denne tabellen av alle mulige kombinasjoner

Eksempel spørring med flere tabeller

Hvilken kunde har kjøpt hvilket produkt?

```
SELECT ProductName, Customer
  FROM products, orders
 WHERE ProductID = OrderedProduct
```

Resultat

products

ProductID	Name	Price
0	TV 50 inch	8999
1	Laptop 2.5GHz	7499

orders

OrderID	OrderedProduct	Customer
0	1	John Mill
1	1	Peter Smith
2	0	Anna Consuma
3	1	Yvonne Potter

Joins

- Spørringer over flere tabeller kalles *joins*,
- Mange måter å relatere tabeller på, altså mange mulige joins, f.eks.
  - equi-join
  - theta-join
  - inner join
  - self join
  - anti join
  - semi join
  - outer join
  - natural join
  - cross join
- De er alle bare forskjellige måter å kombinere informasjon fra to eller flere tabeller
- Oftest (men ikke alltid) interesert i å “joine” på nøkler

Navn på joins

- *Cross join* mellom `t1` og `t2`

```
SELECT * FROM t1, t2
```

- *Equi-join* mellom `t1` og `t2`

```
SELECT * FROM t1, t2
WHERE t1.a = t2.b
```

- *Theta-join* mellom `t1` og `t2`

```
SELECT * FROM t1, t2
WHERE <theta>(t1.a,t2.b)
```

  hvor `<theta>` er en eller annen relasjon (f.eks. `<`, `=`, `!=`, `LIKE`) eller mer komplisert uttrykk
- *Equi-join* er en spesiell type *Theta-join*
- Alle disse formene for join (og et par til vi skal se etterpå) kalles *indre joins* (eng.: *inner joins*)

Problemer med spørring over flere tabeller

Hvilken kunde har kjøpt hvilket produkt?

```
SELECT ProductName, Customer
  FROM products, orders
 WHERE ProductID = ProductID -- ERROR!
```

Resultat

products

ProductID	Name	Price
0	TV 50 inch	8999
1	Laptop 2.5GHz	7499

orders

OrderID	ProductID	Customer
0	1	John Mill
1	1	Peter Smith
2	0	Anna Consuma
3	1	Yvonne Potter

Like kolonnenavn

- Når vi har flere tabeller i samme spørring kan vi få flere kolonner med likt navn
- For å fikse dette kan vi bruke tabellnavnet som prefiks
- F.eks. `products.ProductID` og `orders.OrderID`

Hvilken kunde har kjøpt hvilket produkt?

```
SELECT ProductName, Customer
  FROM products, orders
 WHERE products.ProductID = orders.ProductID
```

Navngi tabeller

- Det er ofte nyttig å kunne gi en tabell et nytt navn
- F.eks. dersom tabellnavnet er langt og gjentas ofte i `WHERE`-klausulen
- Eller dersom vi ønsker å gjøre en self-join (mer om dette om litt)
- Tabeller kan navngis med `AS`-nøkkelordet

Eksempel: Navngi tabeller

Finn produktnavnet og prisen til hver bestilling (2155 rader)

```
SELECT p.product_name, o.unit_price
  FROM products AS p, order_details AS o
 WHERE p.product_id = o.product_id;
```

Kan også droppe `AS`-nøkkelordet, og f.eks. kun skrive

```
FROM products p, order_details o
```

Eksempeler på joins: `Northwind`-databasen

Finn alle unike par av (fulle) navn på kunde og ansatte som har inngått en handel med last (eng.: *freight*) over 500kg(13 rader)

```
SELECT DISTINCT
       c.company_name as kunde,
       e.first_name || ' ' || e.last_name AS ansatt
  FROM orders AS o, customers AS c, employees AS e
 WHERE o.customer_id = c.customer_id AND
       o.employee_id = e.employee_id AND
       o.freight > 500;
```

Relasjonell algebra og SQL

- SQL-spørringene med joins kan også oversettes til relasjonsalgebra
- For eksempel kan de enkle SQL-spørringene vi nå har sett oversettes slik:

```
SELECT <columns>
  FROM <t1>, <t2>, ..., <tN>
 WHERE <condition>
```



$$\pi_{\texttt{<columns>}}(\sigma_{\texttt{<condition>}}(\texttt{<t1>} \times \texttt{<t2>} \times \cdots \times \texttt{<tN>}))$$

Egen notasjon for joins

- SQL har en egen notasjon for joins
- For den typen joins vi har gjort hittil har man `INNER JOIN` -og `ON`-nøkkelordene
- Fremfor å skrive:

```
SELECT product_name
  FROM products AS p, orders AS o
 WHERE p.product_id = o.product_id AND
       o.unit_price > 7000
```

- kan man skrive

```
SELECT p.product_name
  FROM products AS p INNER JOIN order_details AS o
       ON (p.product_id = o.product_id)
 WHERE o.unit_price > 7000
```

- De to spørringene er ekvivalente
- Øverste kalles implisitt join, nederste kalles eksplisitt join
- Skal senere se at enkelte joins ikke kan skrives på den øverste formen
- Den nederste formen gjør det lettere å se hvordan tabellene er “joinet”

Flere join-eksempler (`Northwind`-DB)

Finn ut hvilke drikkevarer som er kjøpt og av hvem [404 rader]

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

Self-joins

- Av og til ønsker man å kombinere informasjon fra rader i samme tabell
- Dette kalles en *self-join*
- Dette gjøres ved å bruke den samme tabellen to eller flere ganger i `FROM`-klausulen
- Må da gi dem forskjellige navn

Self-join-eksempel

Finn navn og pris på alle produkter som er dyrere enn produktet Laptop 2.5GHz?

```
SELECT P2.Name, P2.Price
  FROM Product AS P1, Product AS P2
 WHERE P1.Name = 'Laptop 2.5GHz' AND P1.Price < P2.Price
```

Resultat

P1

ProductID)	Name	Brand	Price
0	TV 50 inch	Sony	8999
1	Laptop 2.5GHz	Lenovo	7499
2	Laptop 8GB RAM	HP	6999
3	Speaker 500	Bose	4999
4	TV 48 inch	Panasonic	11999
5	Phone S6	IPhone	5195

P2

ProductID	Name	Brand	Price
0	TV 50 inch	Sony	8999
1	Laptop 2.5GHz	Lenovo	7499
2	Laptop 8GB RAM	HP	6999
3	Speaker 500	Bose	4999
4	TV 48 inch	Panasonic	11999
5	Phone S6	IPhone	5195

Self-join-eksempel

Finn navn og pris på alle produkter som er dyrere enn produktet Laptop 2.5GHz?

```
SELECT P2.Name, P2.Price
  FROM Product AS P1, Product AS P2
 WHERE P1.Name = 'Laptop 2.5GHz' AND P1.Price < P2.Price
```

Resultat

P1.ProductID	P1.Name	P1.Brand	P1.Price	P2.ProductID	P2.Name	P2.Brand	P2.Price
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
0	TV 50 inch	Sony	8999	5	Phone S6	IPhone	5195
1	Laptop 2.5GHz	Lenovo	7499	0	TV 50 inch	Sony	8999
1	Laptop 2.5GHz	Lenovo	7499	1	Laptop 2.5GHz	Lenovo	7499
1	Laptop 2.5GHz	Lenovo	7499	2	Laptop 8GB RAM	HP	6999
1	Laptop 2.5GHz	Lenovo	7499	3	Speaker 500	Bose	4999
1	Laptop 2.5GHz	Lenovo	7499	4	TV 48 inch	Panasonic	11999
1	Laptop 2.5GHz	Lenovo	7499	5	Phone S6	IPhone	5195
2	Laptop 8GB RAM	HP	6999	0	TV 50 inch	Sony	8999
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.

Naturlig Join

- Vi joiner ofte på de kolonnene som har likt navn
- F.eks. `categories.category_id` med `products.category_id`
- Dette kan gjøres enklere med *naturlig join*
- Naturlig join joiner (med likhet) automatisk på alle kolonner med likt navn
- I tillegg projiserer den vekk de dupliserte kolonnene
- Trenger derfor aldri gi tabellene navn (i resultatet av en naturlig join vil det aldri finnes kolonner med likt navn)
- Merk: Må være sikker på at vi ønsker å joine på ALLE kolonnene med likt navn!

Naturlig Join: Eksempel

Finn navnet på alle drikkevarer [12 rader]

```
SELECT product_name
  FROM categories NATURAL JOIN products
 WHERE category_name = 'Beverages';
```

Enkle `SELECT`-spørringer i et nøtteskall

- `FROM`-klausulen sier hvilke tabell(er) som skal brukes for å besvare spørringen og joiner dem sammen
- `WHERE`-klausulen velger ut hvilke rader som skal være med i svaret
  - Kolonnenavn brukes som variable som instansieres med radenes verdier
  - Kan sammenlikne kolonner og verdier med f.eks. `=`, `!=`, `<`, `<=`, `LIKE`
  - Bruker `AND`, `OR` og `NOT` på uttrykk
  - Evaluerer til enten `TRUE`, `FALSE` eller `NULL` for hver rad
  - Kun de som evaluerer til `TRUE` blir med i svaret
- `SELECT`-klausulen velger hvilke verdier/kolonner som skal være med i svaret
  - Kan også endre rekkefølgen på kolonner, bruke dem i uttrykk, osv.
  - Bruk `*` for å velge alle kolonnene
- SQL bryr seg ikke om mellomrom og linjeskift, eller store og små bokstaver

Mye mer kan gjøres i hver klausul og det finnes flere klausuler, mer om dette senere i kurset!

Takk for nå!

Neste video vil se på nestede spørringer.

IN2090 – Databaser og datamodellering

05 – Nestede spørringer

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Delspørringer

- Husk at tingene i en `FROM`-klausul er tabeller
- Husk også at resultatet av en `SELECT`-spørring er en tabell
- Så, vi kan putte en `SELECT`-spørring i `FROM`-klausulen som en tabell!
- Altså

```
SELECT <columns>
  FROM (SELECT <columns>
          FROM <tables>
         WHERE <condition>
        ) AS subquery
 WHERE <condition>
```

Ekempel-delspørringer

- F.eks., for å finne antall unike kombinasjoner av land og by for alle kunder:

```
SELECT count(*)
  FROM (SELECT DISTINCT country, city FROM customers) AS d
```

- Følgende spørring finner antall solgte drikkevarer med delspørring

```
SELECT sum(d.quantity)
FROM (
      SELECT p.product_id
      FROM products AS p INNER JOIN categories AS c
           ON (p.category_id = c.category_id)
      WHERE c.category_name = 'Beverages'
     ) AS beverages
     INNER JOIN
     order_details AS d
     ON (beverages.product_id = d.product_id)
```

- Merk: Alle delspørringer som tabeller må gis et navn

Delspørringer som verdier

- En aggregatfunksjon over en kolonne returnerer én enkelt verdi
- Vi kan derfor bruke den som en verdi i `WHERE`-klausulen
- Så for å finne alle produkter som koster mer enn gjennomsnittet kan vi skrive:

```
SELECT product_name
  FROM products
 WHERE unit_price > (SELECT avg(unit_price)
                     FROM products)
```

- Merk at én enkel verdi og en tabell med kun én verdi behandles likt av SQL

Delspørringer som mengder

- Dersom vi ønsker å begrense én verdi (eller et tuppel av verdier) til svarene av en annen spørring i `WHERE`-klausulen, kan vi bruke nøkkelorder `IN`
- Kan ofte brukes i stedet for joins
- F.eks. for å finne navnet på alle produkter med en “supplier” fra Tyskland:

```
SELECT product_name
  FROM products
 WHERE supplier_id IN (SELECT supplier_id
                         FROM suppliers
                         WHERE country = 'Germany')
```

Eksempel: Finn navn og pris på alle produktet med lavest pris (1)

Ved `min`-aggregering og delspørring som tabell

```
SELECT p.product_name, p.unit_price
  FROM (
        SELECT min(unit_price) AS minprice
        FROM products
       ) AS h
       INNER JOIN products AS p
       ON (p.unit_price = h.minprice)
```

Eksempel: Finn navn og pris på alle produktet med lavest pris (2)

Ved `min`-aggregering og delspørring som verdi

```
SELECT product_name, unit_price
  FROM products
 WHERE unit_price = (SELECT min(unit_price)
                     FROM products)
```

WITH

Hva er den største differansen mellom prisen på laptopper?

```
SELECT max(l1.Price - l2.Price) AS diff
FROM (SELECT Price FROM products WHERE Name LIKE '%Laptop%') AS l1,
     (SELECT Price FROM products WHERE Name LIKE '%Laptop%') AS l2
```

- Dersom vi ønsker å bruke den samme delspørringen om igjen kan man navngi den først med `WITH`, f.eks.:

```
WITH
  laptops AS (SELECT Price FROM products WHERE Name LIKE '%Laptop%')
SELECT max(l1.Price - l2.Price) AS diff
FROM laptops AS l1, laptops AS l2
```

- Dette er både enklere å lese, lettere å vedlikeholde, og mer effektivt (slipper å kjøre `laptops`-spørringen to ganger)
- `WITH` er også nytting for lesbarhet dersom man har mange delspørringer

Takk for nå!

Neste uke vil vi lære hvordan SQL kan brukes for datamanipulering.

IN2090 – Databaser og datamodellering
06 – Datamanipulering med SQL

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

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 Students-tabellen kan vi kjøre

```
CREATE TABLE Students (
    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 Students (
    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 Students (
    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 Students (
    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 Students (
    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

Alternativ syntaks for skranker

- Man kan også skrive skrankene til slutt, slik:

```
CREATE TABLE Students (
    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 Students (
    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 Students (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 Students (
    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`

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 `Students`-tabellen og alle tabeller som avhenger av denne (slik som `TakesCourse`):

```
DROP TABLE Students 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 products
   SET unit_price = unit_price * 1.1
 WHERE quantity_per_unit LIKE '%bottles%'
```

  øker prisen med 10% på alle produkter som selges i flasker i `products`-tabellen

Takk for nå!

Neste video vil se på typesystemet i SQL og PostgreSQL.

IN2090 – Databaser og datamodellering

06 – Typer og skranker

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Databasers typesystem

- De fleste relasjonelle databaser har et strengt typesystem
- Alle kolonner må ha en tilhørende type, og kun verdier av denne typen kan legges i kolonnen
- I tillegg har man som regel mange funksjoner man kan bruke for å manipulere verdier
- Også disse har en type-signatur som angir lovlig type til argumentene og typen til returnerte verdier

Hvilke datatyper har vi

- PostgreSQL har mange innebygde typer, se:

  `https://www.postgresql.org/docs/current/datatype.html`

- De mest brukte er:
  - Numeriske typer
  - Strengtyper
  - Dato- og tidstyper
  - Boolean
  - Array
- Merk at man også kan lage egne typer (utenfor pensum)
- Man kan også lage egne funksjoner (også utenfor pensum)

SQL-standarden vs. RDBMSer

- Noen typer er en del av SQL-standarden, men mange vanlig brukte typer er ikke
- Også mange funksjoner på SQL-standard-typer som ikke selv er en del av standarden
- Likevel stor likhet mellom ulike RDBMS, ofte kun syntaktiske forskjeller

Casting

- Dersom vi skriver `'1'` er dette en utypet literal for databasen
- Databasen forsøker så å *caste* denne verdien til det den brukes som
- Så dersom vi har tabellene `person(id int)` og `notes(note text)` vil

```
INSERT INTO person VALUES ('1');
INSERT INTO notes VALUES ('1');
```

  vil den første bli en `int` mens den andre `text`
- Vi kan også *caste* ekplisitt, og det er flere måter å gjøre det på
  - Med `::` – `'1'::int`
  - Med `cast`-nøkkelordet – `cast('1' AS int)`
  - Eller å sette typen først – `int '1'`
- Merk: Vi skriver alltid inn utypede litteraler til databasen, enten caster den eller så caster vi

Numeriske typer

Vi har følgende numeriske typer[1]:

Name	Storage Size	Description	Range
`smallint`	2 bytes	small-range integer	-32768 to +32767
`integer`	4 bytes	typical choice for integer	-2147483648 to +2147483647
`bigint`	8 bytes	large-range integer	-9223372036854775808 to +9223372036854775807
`decimal`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`numeric`	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
`real`	4 bytes	variable-precision, inexact	6 decimal digits precision
`double precision`	8 bytes	variable-precision, inexact	15 decimal digits precision
`smallserial`	2 bytes	small autoincrementing integer	1 to 32767
`serial`	4 bytes	autoincrementing integer	1 to 2147483647
`bigserial`	8 bytes	large autoincrementing integer	1 to 9223372036854775807

Større presisjon fører til høyere forbruk av lagringsplass og bergninger/spørringer tar lengre tid.

[1] `https://www.postgresql.org/docs/current/datatype-numeric.html`

Numeriske typer – Funksjoner og operatorer

Har de fleste vanlige matematiske funksjoner og operatorer:

- `ceil`, `floor` – runder opp/ned
- `sqrt`, `power` – kvaderatrot og potens-funksjon
- `abs` – absoluttverdi
- `sin`, `cos`, `tan` – trigonometriske funksjoner
- `random()` – tilfeldig tall
- ...

Se følgende for alle:

`https://www.postgresql.org/docs/11/functions-math.html`

Numeriske typer: eksempel-spørring

Ønsker å pakke bestilte varer i kubeformede bokser for hvert produkt, så vil finne 3. roten av antall bestilte varer for de som er bestilt, samt ca. pris avrundet til nærmeste heltall på det som er bestilt.

```
SELECT product_name,
       ceil(power(units_on_order, 1.0/3)) AS box_size,
       round(unit_price * units_on_order) AS ca_price
FROM products
WHERE units_on_order > 0;
```

Streng-typer

Vi har følgende strengtyper[2]:

Name	Description
`character varying(n)`, `varchar(n)`	variable-length with limit
`character(n)`, `char(n)`	fixed-length, blank padded
`text`	variable unlimited length

- Insetting av strenger som er lengre enn `n` i kolonner med type `varchar(n)` eller `char(n)` gir error
- `text` er ikke en del av SQL-standarden, men nesten alle RDBMS implementerer en slik type
- I PostgreSQL er det ingen fordel å bruke `varchar(n)` eller `char(n)` med hensyn på effektivitet eller minne
- Bruk `varchar(n)` eller `char(n)` kun som skranker (begrense lovlig lengde)

[2] `https://www.postgresql.org/docs/current/datatype-character.html`

Streng-typer – Funksjoner og operatorer

- SQL-standarden har noe rar syntaks for en del streng-manipulering
- For eksempel:
  - `position(sub in str)` finner posisjonen til `sub` i `str`
  - `substring(str from s for e)` finner substrengen i `str` som starter på indeks `s` og har lengde `e`
- Postgres støtter disse, men har mer naturlige varianter, f.eks.:
  - `strpos(str, sub)`, samme som `position`-uttrykket over
  - `substr(str, s, e)`, samme som `substring`-uttrykket over
- Ellers, se følgende for vanlige strengmanipulerings-funksjoner

  `https://www.postgresql.org/docs/11/functions-string.html`

Strengtyper: eksempel-spørring

Lag en pen melding på formen ”Product [name] costs [price] per [quantity]” men hvor ”glasses” er byttet ut med ”jars”.

```
SELECT replace(prod_desc, 'glasses', 'jars') AS prod_desc
FROM (
  SELECT format('Product %s costs %s per %s',
                product_name,
                unit_price,
                quantity_per_unit) AS prod_desc
  FROM products
) AS t;
```

Tidstyper

Vi har følgende tidstyper[3]:

Name	Storage Size	Description	Low Value	High Value	Resolution
`timestamp [ (p) ] [ without time zone ]`	8 bytes	both date and time (no time zone)	4713 BC	294276 AD	1 microsecond
`timestamp [ (p) ] with time zone`	8 bytes	both date and time, with time zone	4713 BC	294276 AD	1 microsecond
`date`	4 bytes	date (no time of day)	4713 BC	5874897 AD	1 day
`time [ (p) ] [ without time zone ]`	8 bytes	time of day (no date)	00:00:00	24:00:00	1 microsecond
`time [ (p) ] with time zone`	12 bytes	time of day (no date), with time zone	00:00:00+1459	24:00:00-1459	1 microsecond
`interval [ fields ] [ (p) ]`	16 bytes	time interval	-178000000 years	178000000 years	1 microsecond

hvor `p` er antall desimaler for sekunder.

[3] `https://www.postgresql.org/docs/current/datatype-datetime.html`

Formater for datoer

Datoer

Example	Description
1999-01-08	ISO 8601; January 8 in any mode (recommended format)
January 8, 1999	unambiguous in any `datestyle` input mode
1/8/1999	January 8 in `MDY` mode; August 1 in `DMY` mode
1/18/1999	January 18 in `MDY` mode; rejected in other modes
01/02/03	January 2, 2003 in `MDY` mode; February 1, 2003 in `DMY` mode; February 3, 2001 in `YMD` mode
1999-Jan-08	January 8 in any mode
Jan-08-1999	January 8 in any mode
08-Jan-1999	January 8 in any mode
99-Jan-08	January 8 in `YMD` mode, else error
08-Jan-99	January 8, except error in `YMD` mode
Jan-08-99	January 8, except error in `YMD` mode
19990108	ISO 8601; January 8, 1999 in any mode
990108	ISO 8601; January 8, 1999 in any mode
1999.008	year and day of year
J2451187	Julian date
January 8, 99 BC	year 99 BC

Formater for tid

Tid

Example	Description
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	same as 04:05; AM does not affect value
04:05 PM	same as 16:05; input hour must be <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	time zone specified by abbreviation
2003-04-12 04:05:06 America/New_York	time zone specified by full name

Tidstyper – Funksjoner og operatorer

- Man kan bruke + og − mellom tidstyper, f.eks.:

```
date '2001-09-28' + interval '1 hour' = timestamp '2001-09-28 01:00:00'
date '2001-09-28' - interval '1 hour' = timestamp '2001-09-27 23:00:00'
date '2001-09-28' + 7                 = date '2001-10-05'
```

- Tilsvarende som for strengtyper har SQL noe rar syntaks for å manipulere tidstyper
- Man bruker `extract(part from value)` for å hente ut `part`-delen av `value`
- For eksempel:

```
extract(hour from timestamp '2001-02-16 20:38:40') = 20
extract(year from timestamp '2001-02-16 20:38:40') = 2001
extract(week from timestamp '2001-02-16 20:38:40') = 7
```

- Man kan også bruke `date_part(part, timestamp)`, f.eks.:

```
date_part('hour', timestamp '2001-02-16 20:38:40') = 20
date_part('year', timestamp '2001-02-16 20:38:40') = 2001
date_part('week', timestamp '2001-02-16 20:38:40') = 7
```

Tidstyper – Andre funksjoner og operatorer

- Man har også funksjoner som gir dagens dato, ol.
- For eksempel vil `now()` gi et `timestamp` med nåværende tidspunkt
- Mens `current_date` er en konstant som inneholder dagens dato
- For å lage en dato kan man også bruke
  `make_date(year int, month int, day int)`
- For tidstyper kan man også bruke `OVERLAPS` mellom par, f.eks.:

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: true
```

- Ellers, se

  `https://www.postgresql.org/docs/11/functions-datetime.html`

Tidstyper: eksempel-spørring

Finn gjennomsnittlig tid fra bestilling av en ordre til den må være levert, i perioden fra 1997 frem til nå.

```
SELECT avg(required_date::timestamp - order_date::timestamp) AS ship_to_req
  FROM orders
 WHERE (order_date, required_date) OVERLAPS (make_date(1997, 1, 1), now());
```

Andre nyttige typer

PostgreSQL støtter også

- Array
- XML
- JSON
- Bitstrenger
- Nettverks-adress-typer
- Geometriske typer
- ...

`https://www.postgresql.org/docs/11/datatype.html`

Check-skranker

- Hittil har vi sett på følgende varianter av skranker:
  - `UNIQUE`
  - `NOT NULL`
  - `PRIMARY KEY`
  - `REFERENCES`
  - typer
- `CHECK` er en annen generell og nyttig skranke
- `CHECK` lar oss bruke et generelt uttrykk for å avgjøre om verdier kan settes inn i kolonnen eller ikke
- For eksempel, med

```
CREATE TABLE students (
  sid int PRIMARY KEY,
  name text NOT NULL,
  birthdate date CHECK (birthdate < date '2010-01-01')
);
```

  kan man kun legge inn studenter med fødselsdato før år 2010.
- Merk: Man kan fortsatt sette inn `NULL`

Takk for nå!

Neste video vil se på views.

IN2090 – Databaser og datamodellering
06 – Views

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

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 s.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

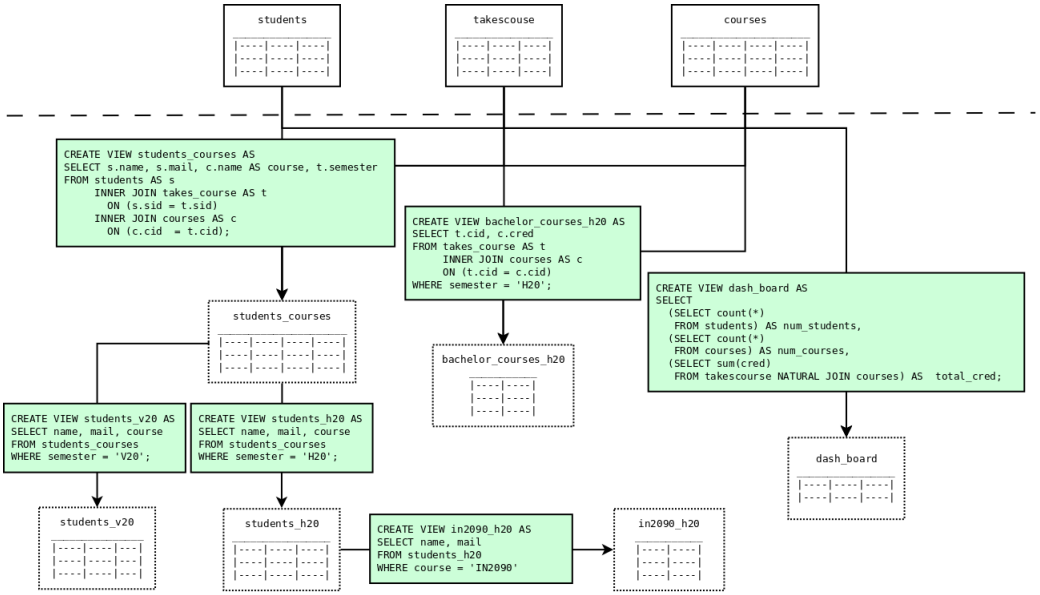
takescourses

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

courses

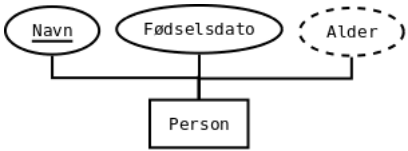
cid	name	cred	lvl
1	Databases	10	B
2	Programming 101	5	B
3	Advanced SQL	10	M

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;
```

Takk for nå!

Neste video vil se på SQL-scripts og transaksjoner.

IN2090 – Databaser og datamodellering
06 – SQL script og transaksjoner

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

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, stdbrthdate 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 uio.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');
```

- Et script `script.sql` kjøres ved `psql <flag> -f script.sql` eller fra psql-shellet ved `\i script.sql`

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 beskjed 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 ønsker 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 midt 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)

Takk for nå!

Neste uke skal vi se på normalformer.

IN2090 – Databaser og datamodellering

07 – Introduksjon til databasedesign

Leif Harald Karlsen (Evgenij Thorstensen)
leifhka@ifi.uio.no



Universitetet i Oslo

Motiverende eksempel

La oss si vi skal lage en database for å lagre data om studenter, kurs, og karakterer.

- For studenter: Brukernavn, navn, etternavn, adresse...
- For kurs: Kurskode, tittel, beskrivelse, ant. studiepoeng...
- Hvem har fått hvilken karakter i hva.

Naiv løsning: Alt i en tabell.

Skjema

Brnavn	Navn	Etternavn	Adresse	Kurskode	Tittel	Beskrivelse	AntSP	Kara
evgenit	Evgenij	Thorstensen	Addr1	IN2090	Databaser	EnBeskr...	10	B
peternl	Petter	Nilsen	Addr2	IN2090	Databaser	EnBeskr...	10	A
evgenit	Evgenij	Thorstensen	Addr1	IN2080	Beregn...	Descr...	10	A
leifhka	Leif H.	Karlsen	Addr3	IN2090	Databaser	EnBeskr...	10	B
leifhka	Leif H.	Karlsen	Addr3	IN3110	Program...	EnBeskr2...	5	C

- Hvorfor er dette et dårlig design, annet enn at det er rotete?
- Hvilke praktiske problemer har vi her?

Anomalier: Dataduplisering

Brnavn	Navn	Etternavn	Adresse	Kurskode	Tittel	Beskrivelse	AntSP	Kara
evgenit	Evgenij	Thorstensen	Addr1	IN2090	Databaser	EnBeskr...	10	B
peternl	Petter	Nilsen	Addr2	IN2090	Databaser	EnBeskr...	10	A
evgenit	Evgenij	Thorstensen	Addr1	IN2080	Beregn...	Descr...	10	A
leifhka	Leif H.	Karlsen	Addr3	IN2090	Databaser	EnBeskr...	10	B
leifhka	Leif H.	Karlsen	Addr3	IN3110	Program...	EnBeskr2...	5	C

I tabellen dupliseres data:

- Tar unødvendig med plass
- Gjør spørringer mindre effektive

Anomalier: Insetting og oppdatering

Brnavn	Navn	Etternavn	Adresse	Kurskode	Tittel	Beskrivelse	AntSP	Kara
evgenit	Evgenij	Thorstensen	Addr1	IN2090	Databaser	EnBeskr...	10	B
peternl	Petter	Nilsen	Addr2	IN2090	Databaser	EnBeskr...	10	A
evgenit	Evgenij	Thorstensen	Addr1	IN2080	Beregn...	Descr...	10	A
leifhka	Leif H.	Karlsen	Addr3	IN2090	Databaser	EnBeskr...	10	B
leifhka	Leif H.	Karlsen	Addr3	IN3110	Program...	EnBeskr2...	5	C
				IN9999	Quantum	Beskr3	10	
abcdef	Aber C.	Deflan	Addr4					

Dupliseringen gjør det vanskeligere å sette inn og oppdatere data:

- Må skrive inn all informasjon om en student, selvom vi bare ønsker å legge til et nytt resultat til en allerede eksisterende student
- Umulig å sette inn en student uten kurs eller et kurs uten en student (uten masse `NULL`s)
- Ved oppdateringer må vi være sikre på å oppdatere alle duplikatene

Anomalier: Sletting

Brnavn	Navn	Etternavn	Adresse	Kurskode	Tittel	Beskrivelse	AntSP	Kara
evgenit	Evgenij	Thorstensen	Addr1	IN2090	Databaser	EnBeskr...	10	B
peternl	Petter	Nilsen	Addr2	IN2090	Databaser	EnBeskr...	10	A
evgenit	Evgenij	Thorstensen	Addr1	IN2080	Beregn...	Descr...	10	A
leifhka	Leif H.	Karlsen	Addr3	IN2090	Databaser	EnBeskr...	10	B
leifhka	Leif H.	Karlsen	Addr3	IN3110	Program...	EnBeskr2...	5	C

- Sletting av et kurs kan føre til sletting av en student
- Sletting av en student kan føre til sletting av et kurs
- Vanskelig å komme unna

Et bra skjema

Brnavn	Navn	Etternavn	Adresse	Kurskode	Tittel	Beskrivelse	AntSP	Kara
evgenit	Evgenij	Thorstensen	Addr1	IN2090	Databaser	EnBeskr...	10	B
peternl	Petter	Nilsen	Addr2	IN2090	Databaser	EnBeskr...	10	A
evgenit	Evgenij	Thorstensen	Addr1	IN2080	Beregn...	Descr...	10	A
leifhka	Leif H.	Karlsen	Addr3	IN2090	Databaser	EnBeskr...	10	B
leifhka	Leif H.	Karlsen	Addr3	IN3110	Program...	EnBeskr2...	5	C

Student

Brnavn	Navn	Etternavn	Adresse
evgenit	Evgenij	Thorstensen	Addr1
peternl	Petter	Nilsen	Addr2
leifhka	Leif H.	Karlsen	Addr3

Karakter

Brnavn	Kurskode	Kara
evgenit	IN2090	B
peternl	IN2090	A
evgenit	IN2080	B
leifhka	IN2090	B
leifhka	IN3110	C

Kurs

Kurskode	Tittel	Beskrivelse	AntSP
IN2090	Databaser	EnBeskr...	10
IN2080	Beregn...	Descr...	10
IN3110	Program...	EnBeskr2...	5

Merk: Samme attributtene og samme verdiene!

Godt og dårlig skjema: Hva er forskjellen?

Brnavn	Navn	Etternavn	Adresse	Kurskode	Tittel	Beskrivelse	AntSP	Kara
evgenit	Evgenij	Thorstensen	Addr1	IN2090	Databaser	EnBeskr...	10	B
peternl	Petter	Nilsen	Addr2	IN2090	Databaser	EnBeskr...	10	A
evgenit	Evgenij	Thorstensen	Addr1	IN2080	Beregn...	Descr...	10	A
leifhka	Leif H.	Karlsen	Addr3	IN2090	Databaser	EnBeskr...	10	B
leifhka	Leif H.	Karlsen	Addr3	IN3110	Program...	EnBeskr2...	5	C

Student

Brnavn	Navn	Etternavn	Adresse
evgenit	Evgenij	Thorstensen	Addr1
peternl	Petter	Nilsen	Addr2
leifhka	Leif H.	Karlsen	Addr3

Karakter

Brnavn	Kurskode	Kara
evgenit	IN2090	B
peternl	IN2090	A
evgenit	IN2080	B
leifhka	IN2090	B
leifhka	IN3110	C

Kurs

Kurskode	Tittel	Beskrivelse	AntSP
IN2090	Databaser	EnBeskr...	10
IN2080	Beregn...	Descr...	10
IN3110	Program...	EnBeskr2...	5

- Godt skjema: `Brnavn` bestemmer alt om en student, `Kurskode` alt om et kurs!
- Dårlig skjema: Ingen (minste mengde) attributter bestemmer resten!

Prinsipper for databasedesign

Altså handler god database design om:

- Data om entiteter i hver sin tabell.
- Relasjoner mellom entiteter via referanser.

Kan dette gjøres mer systematisk?

Normalformer

- Det går an å formelt definere kriterier som gjør at anomalier ikke forekommer.
- Et skjema som oppfyller kriteriene, sies å være på en normalform.
- Vi skal se på normalformene 1NF, 2NF, 3NF, og BCNF.
- For å komme dit, trenger vi litt teori.

Takk for nå!

Neste video vil handle om funksjonelle avhengigheter.

IN2090 – Databaser og datamodellering

07 – Funksjonelle avhengigheter

Leif Harald Karlsen (Evgenij Thorstensen)
leifhka@ifi.uio.no



Universitetet i Oslo

Funksjonell avhengighet

- Et attributt A er **funksjonelt avhengig** av an mengde attributter X hvis det bare kan finnes en verdi av A for hver mengde verdier av attributtene i X.
- Det skrives $X \rightarrow A$, og en slik formel kalles en funksjonell avhengighet (FD).
- For eksempel er `Karakter` funksjonelt avhengig av $\{\texttt{Brnavn}, \texttt{Kurskode}\}$ i `Karakter`-tabellen:

Karakter

Brnavn	Kurskode	Kara
evgenit	IN2090	B
peternl	IN2090	A
evgenit	IN2080	B
leifhka	IN2090	B
leifhka	IN3110	C

- Og både `Navn`, `Etternavn` og `Adresse` er funksjonelt avhengig av `Brnavn` i `Student`-tabellen:

Student

Brnavn	Navn	Etternavn	Adresse
evgenit	Evgenij	Thorstensen	Addr1
peternl	Petter	Nilsen	Addr2
leifhka	Leif H.	Karlsen	Addr3

FDer, data og virkeligheten

- FDer uttrykker det vi mener er sant i virkeligheten som dataene våre beskriver
- F.eks. er brukernavnet til en student faktisk unikt for hver student, mens adressen kanskje ikke trenger å være det
- Kan fort bli et komplisert spørmål om verdens tilstand
- FDer forteller oss hvilke data hører sammen, og hva de hører til

Syntaks for FDer

- Jeg leser ofte pilen som "bestemmer", så

$$X \to Y$$

  leses enten "X bestemmer Y" eller "Y er funksjonelt avhengig av X"
- Vi dropper ofte mengde-tegnene i FDer, så skriver f.eks. i stedet for

$$\{\texttt{Brnavn}, \texttt{Kurskode}\} \to \{\texttt{Karakter}\}$$

  skriver vi ofte

$$\texttt{Brnavn}, \texttt{Kurskode} \to \texttt{Karakter}$$

- Dersom attributtene er enkle bokstaver (A, B, osv.) dropper vi ofte også komma og skriver f.eks. i stedet for:

$$A, B \to X, Y, Z$$

  skriver vi ofte

$$AB \to XYZ$$

FDers oppførsel

- Vi kan samle opp høyresider i FDer, og skrive

$$X \to A, B$$

  dersom vi både har $X \to A$ og $X \to B$.
- FDer er transitive: Hvis $X \to Y$ og $Y \to Z$, så har vi at $X \to Z$.
- En FD $X \to Y$ hvor $Y \subseteq X$ kalles triviell, f.eks.:

$$\texttt{Brnavn}, \texttt{navn} \to \texttt{navn}$$

- Vi ignorerer slike trivielle FDer, fordi de alltid er sanne og dermed ikke gir oss noe informasjon

Eksempel, FDer

$\texttt{R}(\texttt{Brnavn}, \texttt{Navn}, \texttt{Etternavn}, \texttt{Adresse}, \texttt{Kurskode}, \texttt{Tittel}, \texttt{Beskrivelse}, \texttt{AntSP}, \texttt{Karakter})$

Jeg foreslår følgende FDer:

- $\texttt{Brnavn} \rightarrow \texttt{Navn}, \texttt{Etternavn}, \texttt{Adresse}$
- $\texttt{Kurskode} \rightarrow \texttt{Tittel}, \texttt{Beskrivelse}, \texttt{AntSP}$
- $\texttt{Brnavn}, \texttt{Kurskode} \rightarrow \texttt{Karakter}$

Nøkler

- En **supernøkkel** for en relasjon er jo enhver mengde attributter som sammen er unike for relasjonen
- En **kandidatnøkkel** er en $\subseteq$-minimal supernøkkel
- Dersom en mengde attributter er unike forekommer jo hver kombinasjon av disse kun i et tuppel, og bestemmer derfor de andre verdiene i tuplet
- Med andre ord, en nøkkel (enten super eller kandidat) er en mengde attributter som bestemmer de andre attributtene i relasjonen
- FDer sier jo hvilke attributter som bestemmer hvilke andre attributter
- Altså, FDene sier hvilke supernøkler og kandidatnøkler vi har!

FDer og nøkler

- Dersom R er en relasjon med attributter X, så vil:
  - $Y \subseteq X$ være en supernøkkel for R hvis $Y \rightarrow X \setminus Y$, som er evivalent med $Y \rightarrow X$
  - $Y \subseteq X$ er en kandidatnøkkel for R hvis Y er en minimal supernøkkel
- For å sjekke om X er en supernøkkel, sjekk om alt er avhengig av X
- Altså, bruk FDene og finn alle attributter som er avhengige av X, de som er avhengige av disse igjen, osv.

Tillukning

- **Tillukningen** X^+ av X på en mengde FDer er mengden attributter som er funksjonelt avhengige av X
- Hvis $X \rightarrow A$, så er $A \in X^+$ sant
- Hvis $A \notin X^+$, så er ikke $X \rightarrow A$ sant
- Tillukningen kan regnes ut ved å bruke FDene om og om igjen:
  - sett $X^+ = X$
  - sålenge X^+ forandres:
    - finn en FD $Y \rightarrow Z$ med $Y \subseteq X^+$
    - sett $X^+ = X^+ \cup Z$

Eksempel tillukning

Gitt følgende FDer:

- $\texttt{Brnavn} \rightarrow \texttt{Navn}, \texttt{Adresse}$
- $\texttt{Kurskode} \rightarrow \texttt{Grad}$
- $\texttt{Brnavn}, \texttt{Kurskode} \rightarrow \texttt{Karakter}$
- $\texttt{Grad}, \texttt{Karakter} \rightarrow \texttt{Bestått}$

Så har vi følgende tillukninger:

- $\texttt{Brnavn}^+ = \texttt{Brnavn}, \texttt{Navn}, \texttt{Adresse}$
- $\{\texttt{Brnavn}, \texttt{Kurskode}\}^+ = \texttt{Brnavn}, \texttt{Kurskode}, \texttt{Navn}, \texttt{Adresse}, \texttt{Grad}, \texttt{Karakter}, \texttt{Bestått}$
- $\texttt{Navn}^+ = \texttt{Navn}$
- $\texttt{Grad}^+ = \texttt{Grad}$

Finne kandidatnøkler

- Vi må sjekke alle delmengder av attributter, nedenfra. Men, følgende to regler hjelper oss:
  - Hvis *A* ikke forekommer i noen høyreside, er *A* med i **alle** kandidatnøkler.
  - Hvis *A* forekommer i minst en høyreside, men ingen venstresider, er *A* **ikke del** av noen kandidatnøkkel.
- Så begynn med alle attributter som ikke forekommer på høyre side. Beregn tillukningen.
- Hvis alle attributter er med, sjekk minimalitet. Hvis ikke, utvid i tur og orden med ett og ett nytt attributt.

Eksempel (lett)

$\texttt{R}(\texttt{Brnavn}, \texttt{Navn}, \texttt{Etternavn}, \texttt{Adresse}, \texttt{Kurskode}, \texttt{Tittel}, \texttt{Beskrivelse}, \texttt{AntSP}, \texttt{Karakter})$

- $\texttt{Brnavn} \rightarrow \texttt{Navn}, \texttt{Etternavn}, \texttt{Adresse}$
- $\texttt{Kurskode} \rightarrow \texttt{Tittel}, \texttt{Beskrivelse}, \texttt{AntSP}$
- $\texttt{Brnavn}, \texttt{Kurskode} \rightarrow \texttt{Karakter}$

Attributter som ikke er på høyresider: $\texttt{Brnavn}, \texttt{Kurskode}$

Attributter som er i høyresider, men ikke venstre: Alle andre!

Ergo er $\{\texttt{Brnavn}, \texttt{Kurskode}\}$ eneste kandidatnøkkel.

Eksempel (litt mer komplisert)

R(Brnavn, Navn, Adresse, Kurskode, Tittel, Beskrivelse, AntSP, Karakter, Bestått)

- Brnavn $\rightarrow$ Navn, Adresse
- Kurskode $\rightarrow$ Tittel, Beskrivelse, AntSP
- Tittel $\rightarrow$ Kurskode, Beskrivelse, AntSP
- Brnavn, Kurskode $\rightarrow$ Karakter
- Karakter $\rightarrow$ Bestått

Ikke på høyresider: Brnavn
Kun på høyresider: Navn, Adresse, Beskrivelse, AntSP, Bestått
Forsøke å utvide med: Kurskode, Tittel, Karakter

$X =$ Brnavn $\qquad X^+ =$ Brnavn, Navn, Adresse

Eksempel (litt mer komplisert)

$\texttt{R}(\texttt{Brnavn}, \texttt{Navn}, \texttt{Adresse}, \texttt{Kurskode}, \texttt{Tittel}, \texttt{Beskrivelse}, \texttt{AntSP}, \texttt{Karakter}, \texttt{Bestått})$

- $\texttt{Brnavn} \to \texttt{Navn}, \texttt{Adresse}$
- $\texttt{Kurskode} \to \texttt{Tittel}, \texttt{Beskrivelse}, \texttt{AntSP}$
- $\texttt{Tittel} \to \texttt{Kurskode}, \texttt{Beskrivelse}, \texttt{AntSP}$
- $\texttt{Brnavn}, \texttt{Kurskode} \to \texttt{Karakter}$
- $\texttt{Karakter} \to \texttt{Bestått}$

Ikke på høyresider: $\texttt{Brnavn}$
Kun på høyresider: $\texttt{Navn}, \texttt{Adresse}, \texttt{Beskrivelse}, \texttt{AntSP}, \texttt{Bestått}$
Forsøke å utvide med: $\texttt{Kurskode}, \texttt{Tittel}, \texttt{Karakter}$

$$X = \texttt{Brnavn}, \texttt{Kurskode} \qquad X^+ = \texttt{Brnavn}, \texttt{Kurskode}, \texttt{Navn}, \texttt{Adresse}, \texttt{Tittel}, \texttt{Beskrivelse}, \texttt{AntSP}, \texttt{Karakter}, \texttt{Bestått}$$

Kandidatnøkler: $\{\texttt{Brnavn}, \texttt{Kurskode}\}$

Eksempel (litt mer komplisert)

$\texttt{R}(\texttt{Brnavn}, \texttt{Navn}, \texttt{Adresse}, \texttt{Kurskode}, \texttt{Tittel}, \texttt{Beskrivelse}, \texttt{AntSP}, \texttt{Karakter}, \texttt{Bestått})$

- $\texttt{Brnavn} \rightarrow \texttt{Navn}, \texttt{Adresse}$
- $\texttt{Kurskode} \rightarrow \texttt{Tittel}, \texttt{Beskrivelse}, \texttt{AntSP}$
- $\texttt{Tittel} \rightarrow \texttt{Kurskode}, \texttt{Beskrivelse}, \texttt{AntSP}$
- $\texttt{Brnavn}, \texttt{Kurskode} \rightarrow \texttt{Karakter}$
- $\texttt{Karakter} \rightarrow \texttt{Bestått}$

Ikke på høyresider: $\texttt{Brnavn}$
Kun på høyresider: $\texttt{Navn}, \texttt{Adresse}, \texttt{Beskrivelse}, \texttt{AntSP}, \texttt{Bestått}$
Forsøke å utvide med: $\texttt{Kurskode}, \texttt{Tittel}, \texttt{Karakter}$

$$X = \texttt{Brnavn}, \texttt{Tittel} \qquad X^+ = \texttt{Brnavn}, \texttt{Tittel}, \texttt{Navn}, \texttt{Adresse}, \texttt{Kurskode}, \texttt{Beskrivelse}, \texttt{AntSP}, \texttt{Karakter}, \texttt{Bestått}$$

Kandidatnøkler: $\{\texttt{Brnavn}, \texttt{Kurskode}\}$, $\{\texttt{Brnavn}, \texttt{Tittel}\}$

Eksempel (litt mer komplisert)

R(Brnavn, Navn, Adresse, Kurskode, Tittel, Beskrivelse, AntSP, Karakter, Bestått)

- Brnavn → Navn, Adresse
- Kurskode → Tittel, Beskrivelse, AntSP
- Tittel → Kurskode, Beskrivelse, AntSP
- Brnavn, Kurskode → Karakter
- Karakter → Bestått

Ikke på høyresider: Brnavn
Kun på høyresider: Navn, Adresse, Beskrivelse, AntSP, Bestått
Forsøke å utvide med: Kurskode, Tittel, Karakter

$X =$ Brnavn, Karakter $X^+ =$ Brnavn, Karakter, Navn, Adresse, Bestått

Kandidatnøkler: {Brnavn, Kurskode}, {Brnavn, Tittel} ← alle kandidatnøklene for R

Oppsummering så langt

- Skjemaer som er dårlig designet inneholder anomalier
- Som regel skyldes dette at ikke-relatert informasjon er i samme tabell
- FDer sier hvilken informasjon som henger sammen, samt hvilke nøkler tabeller har
- FDer og nøkler gir oss dermed det vi trenger for å spesifisere kriterier for når vi får anomalier og ikke
- Disse kriteriene definerer ulike normalformer

Takk for nå!

Neste video vil handle om normalformer.

IN2090 – Databaser og datamodellering

07 – Normalformer

Leif Harald Karlsen (Evgenij Thorstensen)
leifhka@ifi.uio.no



Universitetet i Oslo

Normalformene 1NF-BCNF

- Normalformene vi skal se på danner et hierarki:

$$BCNF \subseteq 3NF \subseteq 2NF \subseteq 1NF$$

- Det vil si: Hvis et skjema oppfyller 3NF, oppfyller det også 2NF og 1NF
- Høyere NF gir færre anomalier, men flere tabeller og flere joins
- Gitt et skjema, så finnes det en algoritme som lager et ekvivalent skjema på hvilken NF man ønsker
- For **alle** NF er det slik at et skjema oppfyller en gitt NF hvis alle tabeller oppfyller kravene

1NF

- En tabell er på 1NF hvis alle attributter er **atomære**
- Altså ikke attributter med arrays/JSON/osv.
- Heller ikke strenger som inneholder flere verdier
- Av og til ullent begrep (PostgreSQL støtter f.eks. arrays og JSON som datatyper)
- Poenget er at verdiene man ønsker å bruke i joins, sammenlikninger, ol. skal være i egne kolonner
- Dette slik at man skal slippe kompleks manipulering av datastrukturer eller strenger for enkle operasjoner
- Vi antar derfor at 1NF alltid er oppfylt

1NF, eksempler på brudd

Student

Brnavn	...	Veiledere	Adresse
evgenit	...	[arild, martingi]	Gateveien 1b, 0123 Oslo
peternl	...	[abc]	Stedplassen 2, 1234 Bergen

Ansatt

Brnavn	...	Studenter
arild	...	[evgenit]
abc	...	[peternl]

Nesten alltid lurere å

- lage en egen tabell `Veiledning(Student, Veileder)`
- splitte strengen opp i `Student(..., Gate, Postnummer, Poststed)`

2NF

- En tabell oppfyller 2NF hvis
  - den oppfyller 1NF og
  - alle attributter *A* som ikke er nøkkelattributter, **ikke** er funksjonelt avhengige av en delmengde av en kandidatnøkkel
- Nøkkelattributt: Attributt som er med i en kandidatnøkkel.
- Alternativt: En tabell **bryter** 2NF hvis det finnes et ikke-nøkkelattributt *A* som **er avhengig** av en delmengde av en kandidatnøkkel.

2NF, eksempel

Følgende tabell er på 1NF, men ikke 2NF:

$\texttt{R}(\texttt{Brnavn}, \texttt{Navn}, \texttt{Etternavn}, \texttt{Adresse}, \texttt{Kurskode}, \texttt{Tittel}, \texttt{Beskrivelse}, \texttt{AntSP}, \texttt{Karakter})$

- $\texttt{Brnavn} \rightarrow \texttt{Navn}, \texttt{Etternavn}, \texttt{Adresse}$
- $\texttt{Kurskode} \rightarrow \texttt{Tittel}, \texttt{Beskrivelse}, \texttt{AntSP}$
- $\texttt{Brnavn}, \texttt{Kurskode} \rightarrow \texttt{Karakter}$

- Kandidatnøkkel: $\texttt{Brnavn}, \texttt{Kurskode}$
- $\texttt{Navn}$ er avhengig av $\texttt{Brnavn}$ og $\texttt{Brnavn}$ er en del av nøkkelen. Brudd på 2NF.

3NF

- En tabell oppfyller 3NF hvis
  - den oppfyller 2NF og
  - alle ikke-nøkkelattributter **kun** er avhengige av kandidatnøkler
- Alternativt: En tabell bryter 3NF hvis det finnes et ikke-nøkkelattributt som **er avhengig** av noe som **ikke** er en kandidatnøkkel.
- En tabell på 2NF og ikke 3NF kan kun skje dersom en ikke-nøkkelattributt A er avhengig av en (eller fler) ikke-nøkkelattributt(er) X som selv er avhengig av en kandidatnøkkel
- Altså, om K er en kandidatnøkkel kunne vi over ha FDene $K \rightarrow X$ og $X \rightarrow A$.
- Altså, A avhenger ikke direkte av kandidatnøkkelen, men transitivt

3NF, eksempel

Følgende tabell er på 2NF, men ikke 3NF:

$$\texttt{Ansatt}(\texttt{Id}, \texttt{Navn}, \texttt{Avdeling}, \texttt{AvdelingsKode})$$

Gitt FDene:

- $\texttt{Id} \rightarrow \texttt{Navn}, \texttt{AvdelingsKode}$
- $\texttt{AvdelingsKode} \rightarrow \texttt{Avdeling}$

Her har vi:

- `Id` er en kandidatnøkkel (transitivitet)
- Alle attributter avhenger av hele kandidatnøkkelen, altså `Id` (2NF)
- men `Avdeling` er avhengig av `AvdelingsKode`, som ikke er en nøkkelattributt.
- `Avdeling` dobbeltlagres for hver ansatt (burde skilles ut i egen tabell)

BCNF

- Kort for Boyce-Codd normalform
- En tabell oppfyller BCNF hvis alle attributter kun er avhengige av en kandidatnøkkel
- Samme som 3NF, men unntaket for nøkkelattributter er **fjernet**
- Unntaket blir sjeldent brukt, og som regel er tabeller på 3NF også på BCNF
- Huskeregel for BCNF: *The key, the whole key, and nothing but the key, (so help me Codd)*

BCNF, eksempel

Følgende tabell er på 3NF, men ikke BCNF[1]:

Nærmeste butikk

Person	Type	Nærmeste
David	Optiker	Ørneøyet
David	Frisør	Kutt og krøll
Kari	Bokhandler	Merlins bøker
Erik	Bakeri	Deiglig
Erik	Frisør	Klipperud

Med følgende FDer:

- $\texttt{Person}, \texttt{Type} \rightarrow \texttt{Nærmeste}$
- $\texttt{Nærmeste} \rightarrow \texttt{Type}$

Her har vi:

- Kandidatnøkler: $\{\texttt{Person}, \texttt{Type}\}$ og $\{\texttt{Person}, \texttt{Nærmeste}\}$
- Alle ikke-nøkkelattributter (ingen slike) avhenger kun av kandidatnøkler (altså 3NF)
- Men, nøkkel-attributtet $\texttt{Type}$ avhenger av $\texttt{Nærmeste}$ som ikke er en kandidatnøkkel

[1] Inspirert av eksempel fra `https://en.wikipedia.org/wiki/Boyce%E2%80%93Codd_normal_form`

Normalformer i praksis

- Jo høyere normalform, jo færre anomalier
- Men jo høyere normalform, jo fler relasjoner i skjemaet
- BCNF fjerner (nesten) alle anomalier og er et godt kompromiss
- De fleste ordentlige skjemaene (`Northwind`, `filmdb`, osv.) er på BCNF

Takk for nå!

Neste uke vil omhandle algoritmer for å avgjøre normalformer og tapsfri dekomposisjon.

IN2090 – Databaser og datamodellering

08 – Bestemme normalform

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Hvordan sjekke normalformer

- For å sjekke hvilken NF et skjema oppfyller, kan vi sjekke alle tabeller opp mot alle FDer
- Og deretter sjekke om FDen bryter med BCNF, 3NF og 2NF
- Vi skal nå se en algoritme som gjør dette systematisk
- Det er derimot to ting vi må gjøre med FDene før vi kan bruke algoritmen: Skrive om FDer og fjerne redundans

Skrive om FDer

- FDene må skrives på formen $X \rightarrow A$ (splitt høyresidene).
- F.eks. fremfor

$$\texttt{ProduktID} \rightarrow \texttt{Navn}, \texttt{Pris}$$

må man skrive

$$\texttt{ProduktID} \rightarrow \texttt{Navn}$$
$$\texttt{ProduktID} \rightarrow \texttt{Pris}$$

- Algoritmen antar altså at det kun er én attributt på høyresidene

Fjerne redundans

- Hvis $X \to B$, så har vi også $X \cup Y \to B$ for alle Y
- F.eks. dersom

  $$\texttt{ProduktID} \to \texttt{Navn}$$

  så har vi jo også

  $$\texttt{ProduktID}, \texttt{Pris} \to \texttt{Navn}$$

  men ønsker kun å bruke den øverste
- Vi må fjerne alle redundante FDer, algoritmen antar ingen redundans
- Må gjøres når man bestemmer FDene

Algoritme for å sjekke NF

- Først, finn alle kandidatnøkler med algoritmen fra forige uke
- For hver tabell og hver FD $X \rightarrow A$:
  1. Er X en supernøkkel?
     - Ja: BCNF sålangt, gå til neste FD
     - Nei: brudd på BCNF. Gå til 2.
  2. Er A et nøkkelattributt?
     - Ja: 3NF sålangt, gå til neste FD
     - Nei: brudd på 3NF. Gå til 3.
  3. Er X del av en kandidatnøkkel?
     - Nei: 2NF sålangt, gå til neste FD
     - Ja: brudd på 2NF og skjema er på 1NF, stopp.
- Tabellen er så på den laveste normalformen vi får ut av denne algoritmen
- Skjemaet er på den laveste normalformen av tabellenes
- Med andre ord: Hvis jeg har en tabell og en FD som bryter 2NF, er skjemaet på 1NF.

Eksempel 1

Algoritme:

Finn alle kandidatnøkler.
For hver tabell og hver FD $X \to A$:

1. Er X en supernøkkel?
   - Ja: BCNF sålangt, gå til neste FD
   - Nei: brudd på BCNF. Gå til 2.
2. Er A et nøkkelattributt?
   - Ja: 3NF sålangt, gå til neste FD
   - Nei: brudd på 3NF. Gå til 3.
3. Er X del av en kandidatnøkkel?
   - Nei: 2NF sålangt, gå til neste FD
   - Ja: brudd på 2NF og skjema er på 1NF, stopp.

Finn normalformen:

$\texttt{S}(\texttt{Brnavn}, \texttt{Navn}, \texttt{Etternavn}, \texttt{Kurskode}, \texttt{KursTittel}, \texttt{Karakter})$

FDer:

1. $\texttt{Brnavn}, \texttt{Kurskode} \to \texttt{Karakter}$
2. $\texttt{Brnavn} \to \texttt{Navn}$
3. $\texttt{Brnavn} \to \texttt{Etternavn}$
4. $\texttt{Kurskode} \to \texttt{KursTittel}$

Kandidatnøkkel: $\{\texttt{Brnavn}, \texttt{Kurskode}\}$.

- FD 1: $\texttt{Brnavn}, \texttt{Kurskode}$ er en supernøkkel, så BCNF sålangt
- FD 2: $\texttt{Brnavn}$ ikke supernøkkel, brudd på BCNF
- FD 2: $\texttt{Navn}$ ikke nøkkelattributt, brudd på 3NF
- FD 2: $\texttt{Brnavn}$ del av en kandidatnøkkel, brudd på 2NF

Relasjonen $\texttt{S}$ er derfor på 1NF.

Eksempel 2

Algoritme:

Finn alle kandidatnøkler.
For hver tabell og hver FD $X \to A$:

1. Er X en supernøkkel?
   - Ja: BCNF sålangt, gå til neste FD
   - Nei: brudd på BCNF. Gå til 2.
2. Er A et nøkkelattributt?
   - Ja: 3NF sålangt, gå til neste FD
   - Nei: brudd på 3NF. Gå til 3.
3. Er X del av en kandidatnøkkel?
   - Nei: 2NF sålangt, gå til neste FD
   - Ja: brudd på 2NF og skjema er på 1NF, stopp.

Finn normalformen:

$\texttt{Produkt}(\texttt{pid}, \texttt{navn}, \texttt{kategori}, \texttt{pris}, \texttt{butikk})$

FDer:

1. $\texttt{navn}, \texttt{kategori} \to \texttt{pid}$
2. $\texttt{pid}, \texttt{butikk} \to \texttt{pris}$
3. $\texttt{pid} \to \texttt{kategori}$
4. $\texttt{pid} \to \texttt{navn}$

Kandidatnøkler: $\{\texttt{pid}, \texttt{butikk}\}$, $\{\texttt{navn}, \texttt{kategori}, \texttt{butikk}\}$.
FD 1:

1. $\{\texttt{navn}, \texttt{kategori}\}$ er ikke en supernøkkel, så brudd på BCNF.
2. $\texttt{pid}$ er nøkkelattributt, så 3NF så langt.

Eksempel 2

Algoritme:

Finn alle kandidatnøkler.
For hver tabell og hver FD $X \to A$:

1. Er X en supernøkkel?
   - Ja: BCNF sålangt, gå til neste FD
   - Nei: brudd på BCNF. Gå til 2.
2. Er A et nøkkelattributt?
   - Ja: 3NF sålangt, gå til neste FD
   - Nei: brudd på 3NF. Gå til 3.
3. Er X del av en kandidatnøkkel?
   - Nei: 2NF sålangt, gå til neste FD
   - Ja: brudd på 2NF og skjema er på 1NF, stopp.

Finn normalformen:

$\texttt{Produkt}(\texttt{pid}, \texttt{navn}, \texttt{kategori}, \texttt{pris}, \texttt{butikk})$

FDer:

1. $\texttt{navn}, \texttt{kategori} \to \texttt{pid}$
2. $\texttt{pid}, \texttt{butikk} \to \texttt{pris}$
3. $\texttt{pid} \to \texttt{kategori}$
4. $\texttt{pid} \to \texttt{navn}$

Kandidatnøkler: $\{\texttt{pid}, \texttt{butikk}\}$, $\{\texttt{navn}, \texttt{kategori}, \texttt{butikk}\}$.
FD 2:

1. $\{\texttt{pid}, \texttt{butikk}\}$ er en supernøkkel, så BCNF så langt.

Eksempel 2

Algoritme:

Finn alle kandidatnøkler.
For hver tabell og hver FD $X \to A$:

1. Er X en supernøkkel?
   - Ja: BCNF sålangt, gå til neste FD
   - Nei: brudd på BCNF. Gå til 2.
2. Er A et nøkkelattributt?
   - Ja: 3NF sålangt, gå til neste FD
   - Nei: brudd på 3NF. Gå til 3.
3. Er X del av en kandidatnøkkel?
   - Nei: 2NF sålangt, gå til neste FD
   - Ja: brudd på 2NF og skjema er på 1NF, stopp.

Finn normalformen:

$\texttt{Produkt}(\texttt{pid}, \texttt{navn}, \texttt{kategori}, \texttt{pris}, \texttt{butikk})$

FDer:

1. $\texttt{navn}, \texttt{kategori} \to \texttt{pid}$
2. $\texttt{pid}, \texttt{butikk} \to \texttt{pris}$
3. $\texttt{pid} \to \texttt{kategori}$
4. $\texttt{pid} \to \texttt{navn}$

Kandidatnøkler: $\{\texttt{pid}, \texttt{butikk}\}$, $\{\texttt{navn}, \texttt{kategori}, \texttt{butikk}\}$.
FD 3:

1. $\texttt{pid}$ er ikke en supernøkkel, så brudd på BCNF.
2. $\texttt{kategori}$ er en nøkkelattributt, så 3NF så langt.

Eksempel 2

Algoritme:

Finn alle kandidatnøkler.
For hver tabell og hver FD $X \to A$:

1. Er X en supernøkkel?
   - Ja: BCNF sålangt, gå til neste FD
   - Nei: brudd på BCNF. Gå til 2.
2. Er A et nøkkelattributt?
   - Ja: 3NF sålangt, gå til neste FD
   - Nei: brudd på 3NF. Gå til 3.
3. Er X del av en kandidatnøkkel?
   - Nei: 2NF sålangt, gå til neste FD
   - Ja: brudd på 2NF og skjema er på 1NF, stopp.

Finn normalformen:

$\texttt{Produkt}(\texttt{pid}, \texttt{navn}, \texttt{kategori}, \texttt{pris}, \texttt{butikk})$

FDer:

1. $\texttt{navn}, \texttt{kategori} \to \texttt{pid}$
2. $\texttt{pid}, \texttt{butikk} \to \texttt{pris}$
3. $\texttt{pid} \to \texttt{kategori}$
4. $\texttt{pid} \to \texttt{navn}$

Kandidatnøkler: $\{\texttt{pid}, \texttt{butikk}\}$, $\{\texttt{navn}, \texttt{kategori}, \texttt{butikk}\}$.
FD 4:

1. `pid` er ikke en supernøkkel, så brudd på BCNF.
2. `navn` er et nøkkelattributt, så ikke brudd på 3NF

Altså er `Produkt` på 3NF.

Takk for nå!

Neste video vil handle om tapsfri dekomposisjon.

IN2090 – Databaser og datamodellering

08 – Tapsfri dekomposisjon

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Hvordan oppnå BCNF?

- Hvordan lage et nytt skjema på BCNF som inneholder den samme informasjonen?
- Problemet er, grovt sett, at data som ikke hører sammen er i samme tabell.
- Kan løses ved å **dekomponere** tabellen til mindre tabeller.
- Kan ikke dekomponere som vi vil — dekomposisjonen må være **tapsfri**.

Tapsfri dekomponering

La $R(X)$ være en relasjon. En dekomponering av R er en mengde nye relasjoner $\{S_1(Y_1), \ldots, S_n(Y_n)\}$ slik at

1. $Y_i \subseteq X$
2. $\bigcup_{i=1}^{n} Y_i = X$

En dekomponering er tapsfri hvis vi alltid kan ta en instans IR av R, projisere ned til instanser IS_i av S_i, og så rekonstruere IR via naturlig join, altså:

$$\pi_{Y_1}(IR) \bowtie \pi_{Y_2}(IR) \bowtie \ldots \bowtie \pi_{Y_n}(IR) = IR$$

Ikke tapsfri dekomponering

Opprinnelig tabell: `Ansatte(AvdID, AvdNavn, AnsattID, Navn, Etternavn)`

Dekomponert til:

- `Avdeling(AvdID, AvdNavn)`
- `Ansatt(AnsattId, Navn, Etternavn)`

Alle atttributter er med, men vi har mistet noe viktig, nemlig **forholdet mellom** ansatt og avdeling.

Tapsfri dekomponering, eksempel

Brnavn	Navn	Etternavn	Adresse	Kurskode	Tittel	Beskrivelse	AntSP	Kara
evgenit	Evgenij	Thorstensen	Addr1	IN2090	Databaser	EnBeskr...	10	B
peternl	Petter	Nilsen	Addr2	IN2090	Databaser	EnBeskr...	10	A
evgenit	Evgenij	Thorstensen	Addr1	IN2080	Beregn...	Descr...	10	A
leifhka	Leif H.	Karlsen	Addr3	IN2090	Databaser	EnBeskr...	10	B
leifhka	Leif H.	Karlsen	Addr3	IN3110	Program...	EnBeskr2...	5	C

Student

Brnavn	Navn	Etternavn	Adresse
evgenit	Evgenij	Thorstensen	Addr1
peternl	Petter	Nilsen	Addr2
leifhka	Leif H.	Karlsen	Addr3

Karakter

Brnavn	Kurskode	Kara
evgenit	IN2090	B
peternl	IN2090	A
evgenit	IN2080	B
leifhka	IN2090	B
leifhka	IN3110	C

Kurs

Kurskode	Tittel	Beskrivelse	AntSP
IN2090	Databaser	EnBeskr...	10
IN2080	Beregn...	Descr...	10
IN3110	Program...	EnBeskr2...	5

Alle attributter er med, og naturlig join gir opprinnelig tabell.

Hvordan garantere tapsfri dekomponering?

- Fagins teorem: En dekomponering av $R(X, Y, Z)$ til $S_1(X, Y), S_2(X, Z)$ er tapsfri hvis og bare hvis $X \rightarrow Y$
- Med andre ord, vi kan skille ut noen attributter og det de alle er avhengige av
- Dette gir opphav til dekomponeringsalgoritmen for BCNF

Tapsfri dekomponering til BCNF

Tapsfri dekomponering av $R(X)$ med FDer F:

1. Beregn nøklene til R (fra F)
2. Split alle FDer i F slik at det kun er ett attributt på høyresiden av hver FD (f.eks. $A, B \to C, D$ blir $A, B \to C$ og $A, B \to D$)
3. Sjekk om R bryter med BCNF.
   - 3.1 Hvis R *ikke* bryter med BCNF (altså er på BCNF), stopp og returner R
   - 3.2 Hvis R bryter med BCNF:
     - 3.2.1 Finn én FD $Y \to A \in F$ som bryter med BCNF
     - 3.2.2 Beregn Y^+ med hensyn på FDene i F
     - 3.2.3 Dekomponer R til $S_1(Y^+)$ og $S_2(Y, X/Y^+)$
     - 3.2.4 Fortsett rekursivt over S_1
       (med FDene som kun inneholder attributter fra S_1 (altså Y^+))
     - 3.2.5 Fortsett rekursivt over S_2
       (med FDene som kun inneholder attributter fra S_2 (altså $Y, X/Y^+$))

Eksempel 1

Tapsfri dekomponering av $R(X)$ med FDer F:

1. Beregn nøklene til R (fra F)
2. Hvis R *ikke* bryter med BCNF, stopp og returner R
3. Hvis R bryter med BCNF:
   - 3.1 Finn FD $Y \to A \in F$ som bryter med BCNF
   - 3.2 Beregn Y^+ med hensyn på F
   - 3.3 Dekomponer R til $S_1(Y^+)$ og $S_2(Y, X/Y^+)$
   - 3.4 Fortsett rekursivt over S_1 (med FDene med kun attributter fra S_1)
   - 3.5 Fortsett rekursivt over S_2 (med FDene med kun attributter fra S_2)

La $R(A, B, C)$ ha FDer $F = \{AB \to C, C \to A\}$.

- Kanidatnøkkel: B forekomer ikke på høyresider, så B er med i alle nøkler. $\{A, B\}$ og $\{B, C\}$ er nøklene
- $AB \to C$ er ikke brudd på BCNF, siden AB er en supernøkkel
- $C \to A$ er brudd på BCNF, men ikke på 3NF, siden A er et nøkkelattributt
- Beregner $C^+ = CA$
- Dekomponerer R til $S_1(C, A)$ og $S_2(C, B)$
- Kun én FD som holder for S_1 ($C \to A$) og bryter ikke med BCNF og ingen FDer for S_2, altså begge på BCNF
- $R(A, B, C)$ dekomponeres dermed til $S_1(C, A)$ og $S_2(C, B)$

Eksempel 2

Tapsfri dekomponering av $R(X)$ med FDer F:

1. Beregn nøklene til R (fra F)
2. Hvis R *ikke* bryter med BCNF, stopp og returner R
3. Hvis R bryter med BCNF:
   - 3.1 Finn FD $Y \to A \in F$ som bryter med BCNF
   - 3.2 Beregn Y^+ med hensyn på F
   - 3.3 Dekomponer R til $S_1(Y^+)$ og $S_2(Y, X/Y^+)$
   - 3.4 Fortsett rekursivt over S_1 (med FDene med kun attributter fra S_1)
   - 3.5 Fortsett rekursivt over S_2 (med FDene med kun attributter fra S_2)

$\texttt{S}(\texttt{Brnavn}, \texttt{Navn}, \texttt{Etternavn}, \texttt{Kurskode}, \texttt{KursTittel}, \texttt{Karakter})$

FDer:

1. $\texttt{Brnavn}, \texttt{Kurskode} \to \texttt{Karakter}$
2. $\texttt{Brnavn} \to \texttt{Navn}$
3. $\texttt{Brnavn} \to \texttt{Etternavn}$
4. $\texttt{Kurskode} \to \texttt{KursTittel}$

- Kandidatnøkkel: $\{\texttt{Brnavn}, \texttt{Kurskode}\}$
- $\texttt{Brnavn} \to \texttt{Navn}$ bryter med BCNF
- Beregner $\texttt{Brnavn}^+ = \{\texttt{Brnavn}, \texttt{Navn}, \texttt{Etternavn}\}$
- Får da $\texttt{S}_1(\texttt{Brnavn}, \texttt{Navn}, \texttt{Etternavn})$ og $\texttt{S}_2(\texttt{Brnavn}, \texttt{Kurskode}, \texttt{Kurstittel}, \texttt{Karakter})$
- S_1 har FDene 2. og 3. , men ingen av disse bryter BCNF
- S_2 har FDene 1. og 4.

Eksempel 2 (forts.)

Tapsfri dekomponering av $R(X)$ med FDer F:

1. Beregn nøklene til R (fra F)
2. Hvis R *ikke* bryter med BCNF, stopp og returner R
3. Hvis R bryter med BCNF:
   - 3.1 Finn FD $Y \to A \in F$ som bryter med BCNF
   - 3.2 Beregn Y^+ med hensyn på F
   - 3.3 Dekomponer R til $S_1(Y^+)$ og $S_2(Y, X/Y^+)$
   - 3.4 Fortsett rekursivt over S_1 (med FDene med kun attributter fra S_1)
   - 3.5 Fortsett rekursivt over S_2 (med FDene med kun attributter fra S_2)

$\mathtt{S}_2(\mathtt{Brnavn}, \mathtt{Kurskode}, \mathtt{Kurstittel}, \mathtt{Karakter})$

FDer:

1. $\mathtt{Kurskode} \to \mathtt{Kurstittel}$
2. $\mathtt{Brnavn}, \mathtt{Kurskode} \to \mathtt{Karakter}$

- Kandidatnøkkel: $\{\mathtt{Brnavn}, \mathtt{Kurskode}\}$
- $\mathtt{Kurskode} \to \mathtt{Kurstittel}$ bryter med BCNF
- $\mathtt{Kurskode}^+ = \mathtt{Kurskode}, \mathtt{Kurstittel}$
- Får $\mathtt{S}_{21}(\mathtt{Kurskode}, \mathtt{Kurstittel})$ og $\mathtt{S}_{22}(\mathtt{Kurskode}, \mathtt{Brnavn}, \mathtt{Karakter})$
- $\mathtt{S}_{21}$ har kun første FD som ikke bryter med BCNF
- $\mathtt{S}_{22}$ har kun andre FD som ikke bryter med BCNF

Eksempel 2 (forts.)

$\texttt{S}(\texttt{Brnavn}, \texttt{Navn}, \texttt{Etternavn}, \texttt{Kurskode}, \texttt{KursTittel}, \texttt{Karakter})$

FDer:

1. $\texttt{Brnavn}, \texttt{Kurskode} \rightarrow \texttt{Karakter}$
2. $\texttt{Brnavn} \rightarrow \texttt{Navn}$
3. $\texttt{Brnavn} \rightarrow \texttt{Etternavn}$
4. $\texttt{Kurskode} \rightarrow \texttt{KursTittel}$

Dekomponeres altså til:

- $\texttt{S}_1(\texttt{Brnavn}, \texttt{Navn}, \texttt{Etternavn})$
- $\texttt{S}_{21}(\texttt{Kurskode}, \texttt{Kurstittel})$
- $\texttt{S}_{22}(\texttt{Kurskode}, \texttt{Brnavn}, \texttt{Karakterer})$

Dekomponering i praksis

- Algoritmen gir oss den riktige strukturen på tabellene
- Må etterpå gi tabellene meningsfulle navn og sette skranker på kolonner
- Dersom man startet med en skjema som inneholdt data må disse flyttes over
- Gitt en tabell $R(X)$ som dekomponeres til $S_1(Y_1), \ldots, S_n(Y_n)$ kan dette gjøres ved å kjøre følgende for hver S_i:

```
INSERT INTO S_i
SELECT DISTINCT Y_i
FROM R;
```

Takk for nå!

Neste video vil handle om design i praksis.

IN2090 – Databaser og datamodellering

08 – Design i praksis

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Hvordan designe til ca. BCNF?

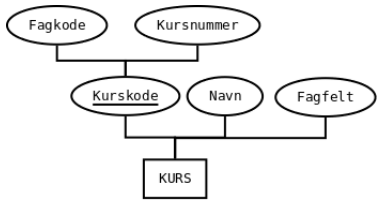
- Én tabell per entitetstype, altså ett tuppel = en entitet
- Relasjoner mellom entiteter representeres enten
  - via fremmenøkler fra en entitets-tabell til en annen (en-til-en og en-til-mange)
  - eller via egne tabeller (mange-til-mange)

ER-realisering og normalformer

- Prinsippene fra forige slide minner veldig om realiseringsalgoritmen for ER
- Den algoritmen er laget slik at den (så langt det lar seg gjøre) gir et BCNF-skjema som resultat
- F.eks.:
  - Hver entitet blir én tabell hvor nøklene bestemmer alle andre attributter
  - Relasjoner blir kun del av en entitets tabell dersom kolonnen er bestemt av nøkkelen
  - Flerverdi-attributter blir egne tabeller
  - Utledbare attributter blir ikke del av realiseringen
- Men, merk at vi ikke er garantert BCNF, eller noe annet høyere enn 1NF!
- ER er ikke uttrykningskraftig nok til å uttrykke alle FDer

ER-realisering til 1NF – Eksempel

- ER-modell:



- FDer:
  - $Fagkode, Kursnummer \rightarrow Navn$
  - $Fagkode \rightarrow Fagfelt$ ← Kan ikke uttrykkes i ER!
- Realiseres til:

$$\texttt{Kurs}(\underline{\texttt{Fagkode}, \texttt{Kursnummer}}, \texttt{Navn}, \texttt{Fagfelt})$$

- $Fagkode \rightarrow Fagfelt$ bryter med 2NF fordi $Fagkode$ kun er en del av kandidatnøkkelen.

Når må man dekomponere?

- Etter ER-realisering i noen tilfeller
- Overtar dårlig designet database
- Databaser utvilker seg over tid
- Virkeligheten kan endre seg
- Migrere data fra f.eks. regneark til relasjonsskjema



sektor	art	år	statistikkvariabel	10721: Offentlig forvaltning. Inntekter og utgifter (mill. kr)
6100 Statsforvaltningen	A Totale inntekter	2000	Inntekter og utgifter	716095
6100 Statsforvaltningen	A Totale inntekter	2001	Inntekter og utgifter	728298
6100 Statsforvaltningen	A Totale inntekter	2002	Inntekter og utgifter	732725
6100 Statsforvaltningen	A Totale inntekter	2003	Inntekter og utgifter	745081
6100 Statsforvaltningen	A Totale inntekter	2004	Inntekter og utgifter	842956
6100 Statsforvaltningen	A Totale inntekter	2005	Inntekter og utgifter	965980
6100 Statsforvaltningen	A Totale inntekter	2006	Inntekter og utgifter	1125908
6100 Statsforvaltningen	A Totale inntekter	2007	Inntekter og utgifter	1182816
6100 Statsforvaltningen	A Totale inntekter	2008	Inntekter og utgifter	1336544
6100 Statsforvaltningen	A Totale inntekter	2009	Inntekter og utgifter	1157152
6100 Statsforvaltningen	A Totale inntekter	2010	Inntekter og utgifter	1223062
6100 Statsforvaltningen	A Totale inntekter	2011	Inntekter og utgifter	1367004
6100 Statsforvaltningen	A Totale inntekter	2012	Inntekter og utgifter	1444387
6100 Statsforvaltningen	A Totale inntekter	2013	Inntekter og utgifter	1430247
6100 Statsforvaltningen	A Totale inntekter	2014	Inntekter og utgifter	1450038
6100 Statsforvaltningen	A Totale inntekter	2015	Inntekter og utgifter	1420699
6100 Statsforvaltningen	A Totale inntekter	2016	Inntekter og utgifter	1407886
6100 Statsforvaltningen	A Totale inntekter	2017	Inntekter og utgifter	1500859
6100 Statsforvaltningen	A Totale inntekter	2018	Inntekter og utgifter	1666401
6100 Statsforvaltningen	A Totale inntekter	2019	Inntekter og utgifter	1695598
6100 Statsforvaltningen	A1 Skatter	2000	Inntekter og utgifter	208477
6100 Statsforvaltningen	A1 Skatter	2001	Inntekter og utgifter	209395

Data om offentlig forvaltning fra SSB

(`https://data.ssb.no/api/v0/dataset/928194?lang=no`)

BCNF = godt design?

$$\texttt{R}(\texttt{Brukernavn}, \texttt{Navn}, \texttt{Etternavn}, \texttt{KursKode}, \texttt{KursNavn})$$

FDer:

1. $\texttt{Brukernavn} \rightarrow \texttt{Navn}$
2. $\texttt{Brukernavn} \rightarrow \texttt{Etternavn}$
3. $\texttt{KursKode} \rightarrow \texttt{KursNavn}$

- Kan dekomponeres til:
  - $\texttt{S}(\texttt{Brukernavn}, \texttt{Navn})$
  - $\texttt{T}(\texttt{Brukernavn}, \texttt{Etternavn})$
  - $\texttt{U}(\texttt{Brukernavn}, \texttt{KursKode})$
  - $\texttt{V}(\texttt{KursKode}, \texttt{KursNavn})$
- Har her brukt algoritmen, men fjernet tillukningen av venstresiden
- Tapsfri dekomponering og BCNF, men vi har to tabeller som burde vært en
- Godt databasedesign er altså mer enn bare normalformer

Takk for nå!

Neste uke vil handle om aggregering og sortering i SQL.

IN2090 – Databaser og datamodellering

09 – Sortering

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Sortering

- For å sortere radene i resultatet fra en `SELECT`-spørring, kan vi bare legge `ORDER BY <kolonner>` på slutten av spørringen
- hvor `<kolonner>` er en liste med kolonner
- For eksempel, for å sortere alle produkter etter pris:

```
SELECT product_name, unit_price
  FROM products
ORDER BY unit_price;
```

- Sorteringen er gjort i henhold til typens naturlige ordning
  - Tall: verdi
  - Tekst: alfabetisk
  - Tidspunkter: kronologisk
  - osv.
- `ORDER BY`-klausulen kommer alltid etter `WHERE`-klausulen

Sortere på flere kolonner og reversering

- Standard-ordningen er fra minst til størst
- For å reversere ordningen trenger man bare legge til `DESC` (kort for "descending") etter kolonnenavnet
- Med flere kolonner i `ORDER BY` vil radene ordnes først ihht. til den første kolonnen, så ihht. den andre kolonnen for de med like verdier i den første, osv.
- For eksempel, for å sortere drikkevarer først på pris, og så på antall på lager, begge i nedadgående rekkefølge:

```
  SELECT product_name, unit_price, units_in_stock
    FROM products
ORDER BY unit_price DESC,
         units_in_stock DESC;
```

Begrense antall rader i resultatet

- Når vi gjør spørringer mot store tabeller får vi ofte mange svar
- Av og til er vi ikke interessert i alle svarene
- For å begrense antall rader kan vi bruke `LIMIT`
- For eksempel, for å velge ut de dyreste 5 produktene:

```
SELECT product_name, unit_price
  FROM products
ORDER BY unit_price DESC
 LIMIT 5;
```

- `LIMIT`-klausulen kommer alltid til sist

Eksempel: Finn navn og pris på produktet med lavest pris

Ved `min`-aggregering og tabell-delspørring

```
SELECT p.product_name, p.unit_price
  FROM products AS p INNER JOIN
       (SELECT min(unit_price) AS minprice FROM products) AS h
       ON p.unit_price = h.minprice;
```

Ved `min`-aggregering og verdi-delspørring

```
SELECT product_name, unit_price
  FROM products
 WHERE unit_price = (SELECT min(unit_price) FROM products);
```

Ved `ORDER BY` og `LIMIT 1`

```
SELECT product_name, unit_price
  FROM products
ORDER BY unit_price
 LIMIT 1;
```

Hoppe over rader

- Av og til ønsker man å hoppe over rader
- Dette er nyttig dersom man ønsker å presentere resultater i grupper
- F.eks. slik som søkeresultater fra en søkemotor, man får presentert én og én side med resultater
- Dette gjøres med `OFFSET`-klausulen
- Dersom vi ønsker å vise 10 og 10 produkter av gangen, sortert etter pris, kan man kjøre:

```
  SELECT product_name, unit_price
    FROM products
ORDER BY unit_price DESC
   LIMIT 10
  OFFSET <sidetall*10>; -- Først 0, så 10, så 20, osv.
```

Takk for nå!

Neste video handler om aggregering i grupper.

IN2090 – Databaser og datamodellering

09 – Aggregering i grupper

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Aggregere i grupper

- Vi har sett hvordan vi kan aggregere over hele kolonner
- Det finner derimot en egen klausul for å gruppere radene før man aggregerer
- Nemlig `GROUP BY <kolonner>`
- `GROUP BY` tar en liste med kolonner, og grupperer dem i henhold til likhet på verdiene i disse kolonnene
- Vi kan så bruke aggregeringsfunksjoner på hver gruppe i `SELECT`-klausulen
- Vi kan da også ha de grupperende kolonnene sammen med aggregatet i `SELECT`-klausulen
- Kun de grupperte kolonnene gir mening å ha utenfor et aggregat i `SELECT`

Aggregere i grupper: Eksempel

Finn gjennomsnittsprisen for hver kategori

```
  SELECT Category, avg(Price) AS Averageprice
    FROM Products
GROUP BY Category
```

Resultat

Products

ProductID (int)	Name (text)	Brand (text)	Price (float)	Category (text)
0	TV 50 inch	Sony	8999	Televisions
1	Laptop 2.5GHz	Lenovo	7499	Computers
2	Laptop 8GB RAM	HP	6999	Computers
3	Speaker 500	Bose	4999	Speakers
4	TV 48 inch	Panasonic	11999	Televions
5	Laptop 1.5GHz	IPhone	5195	Computers

Aggregere i grupper: Eksempel

Finn gjennomsnittsprisen for hver kategori

```
SELECT Category, avg(Price) AS Averageprice
  FROM Products
GROUP BY Category
```

Resultat: Velg ut kolonner og grupper ihht. `Categories`

Price	Category
8999	Televisions
7499	Computers
6999	Computers
4999	Speakers
11999	Televisions
5195	Computers

Aggregere i grupper: Eksempel

Finn gjennomsnittsprisen for hver kategori

```
SELECT Category, avg(Price) AS Averageprice
  FROM Products
GROUP BY Category
```

Resultat: Regn ut aggregatet for hver gruppe og ferdigstill

avg(Price)	Category
10499	Televisions
6731	Computers
4999	Speakers

Aggregering i grupper: Eksempel 1

Finn antall produkter per bestilling

```
  SELECT order_id, sum(quantity) AS nr_products
    FROM order_details
GROUP BY order_id;
```

Aggregering i grupper: Eksempel 2

Finn gjennomsnittspris for hver kategori (i Northwind)

```
  SELECT c.category_name, avg(p.unit_price) AS Averageprice
    FROM categories AS c INNER JOIN products AS p
         ON (c.category_id = p.category_id)
GROUP BY c.category_name;
```

Gruppere på flere kolonner

- Vi kan også gruppere på flere kolonner
- Da vil hver gruppe bestå av de radene med like verdier på alle kolonnene vi grupperer på

Finn antall produkter for hver kombinasjon av kategori og hvorvidt produktet fortsatt selges

```
  SELECT c.category_name, p.discontinued, count(*) AS nr_products
    FROM categories AS c INNER JOIN products AS p
         ON (c.category_id = p.category_id)
GROUP BY c.category_name, p.discontinued;
```

Aggregering i grupper: Eksempel 3

Finn navn på ansatte og antall bestillinger den ansatte har håndtert, sortert etter antall bestillinger fra høyest til lavest

```
  SELECT format('%s %s', e.first_name, e.last_name) AS emp_name,
         count(*) AS num_orders
    FROM orders AS o INNER JOIN employees AS e
         ON (o.employee_id = e.employee_id)
GROUP BY e.first_name, e.last_name
ORDER BY num_orders DESC;
```

Filtrere på aggregat-resultat

- I enkelte tilfeller er vi kun interessert i grupper hvor et aggregat har en bestemt verdi
- F.eks. dersom man vil vite kategorinavn og antall produkter på de kategoriene som har flere enn 10 produkter
- Nå kan vi gjøre dette med en delspørring:

```
SELECT category_name, nr_products
FROM (
  SELECT c.category_name, count(*) AS nr_products
  FROM categories AS c
       INNER JOIN products AS p ON (c.category_id = p.category_id)
  GROUP BY c.category_name) AS t
WHERE nr_products > 10;
```

- Men det finnes en egen klausul for å velge ut grupper

HAVING-klausulen

- Denne klausulen heter `HAVING` og kommer rett etter `GROUP BY`, slik:

```
  SELECT c.category_name, count(*) AS nr_products
    FROM categories AS c
         INNER JOIN products AS p ON (c.category_id = p.category_id)
GROUP BY c.category_name
  HAVING count(*) > 10;
```

- Merk: Kan ikke bruke navnene vi gir i `SELECT`
- `HAVING` blir altså evaluert på hver gruppe
- Fungerer altså som en slags `WHERE` for grupper

Oversikt over SQLs `SELECT`

- Vi har nå sett mange nye klausuler
- Generelt ser våre SQL-spørringer nå slik ut:

```
    WITH <navngitte-spørringer>
  SELECT <kolonner>
    FROM <tabeller>
   WHERE <uttrykk>
GROUP BY <kolonner>
  HAVING <uttrykk>
ORDER BY <kolonner> [DESC]
   LIMIT <N>
  OFFSET <M>
```

- I denne rekkefølgen (`LIMIT` og `OFFSET` kan bytte plass)
- Kan selvfølgelig droppe klausuler, men må ha `GROUP BY` for å ha `HAVING`

Hvor kan ulike navn brukes

- Navnene vi lager med `AS` i `WITH`-klausulen kan brukes i alle de etterfølgende spørringene
- Navnene fra `SELECT` kan brukes i `ORDER BY`-klausulen og alle ytre spørringer
- Navnene fra `FROM` kan brukes i alle klausuler utenom samme `FROM`-klausul

Takk for nå!

Neste video gjennomgår avanserte eksempler.

IN2090 – Databaser og datamodellering

10 – Ytre joins

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Aggregering og NULL

- Aggregering med `sum`, `min`, `max` og `avg` ignorerer `NULL`-verdier
- Det betyr også at dersom det kun er `NULL`-verdier i en kolonne blir resultatet av disse `NULL`
- `count(*)` teller med `NULL`-verdier
- Men dersom vi oppgir en konkret kolonne, f.eks. `count(product_name)` vil den kun telle verdiene som ikke er `NULL`
- For eksempel:

Person

Name	Age
Per	2
Kari	4
Mari	NULL

```
SELECT min(Age) FROM Person;    --> 2
SELECT avg(Age) FROM Person;    --> 3
SELECT count(Age) FROM Person; --> 2
SELECT count(*) FROM Person;    --> 3

SELECT sum(Age) FROM Person
WHERE Name = 'Mari';            --> NULL

SELECT count(Age) FROM Person
WHERE Name = 'Mari';            --> 0
```

Repetisjon: Inner joins

Hvilken kunde har kjøpt hvilket produkt?

```
SELECT ProductName, Customer
FROM products AS p INNER JOIN orders AS o
     ON p.ProductID = o.ProductID
```

Resultat

products

ProductID	Name	Price
0	TV 50 inch	8999
1	Laptop 2.5GHz	7499

orders

OrderID	ProductID	Customer
0	1	John Mill
1	1	Peter Smith
2	0	Anna Consuma
3	1	Yvonne Potter

Inner joins og manglende verdier

Hvilken kunde har kjøpt hvilket produkt?

```
SELECT ProductName, Customer
FROM products AS p INNER JOIN orders AS o
     ON p.ProductID = o.ProductID
```

Resultat

products

ProductID	Name	Price
0	TV 50 inch	8999
1	Laptop 2.5GHz	7499
2	Noise-amplifying Headphones	9999

orders

OrderID	ProductID	Customer
0	1	John Mill
1	1	Peter Smith
2	0	Anna Consuma
3	1	Yvonne Potter

Inner joins og manglende verdier med aggregater

Hvor mange har kjøpt hvert produkt?

```
SELECT p.ProductName, count(o.Customer) AS num
FROM products AS p INNER JOIN orders AS o
     ON p.ProductID = o.ProductID
GROUP BY p.ProductName
```

Resultat

products

ProductID	Name	Price
0	TV 50 inch	8999
1	Laptop 2.5GHz	7499
2	Noise-amplifying Headphones	9999

orders

OrderID	ProductID	Customer
0	1	John Mill
1	1	Peter Smith
2	0	Anna Consuma
3	1	Yvonne Potter

Problemer med Indre joins

- I forige spørring fikk vi ikke opp at 0 kunder har kjøpt Noise-amplifying Headset
- Årsaken er at den ikke joiner med noe, og derfor forsvinner fra svaret
- For å få ønsket resultat trenger vi altså en ny type join
- De nye joinene som løser problemet vårt heter ytre joins, eller *outer join* på engelsk

Outer Joins

- Vi har flere varianter av ytre joins, nemlig
  - `left outer join`
  - `right outer join`
  - `full outer join`
- Brukes ved å bytte ut `INNER JOIN` med f.eks. `LEFT OUTER JOIN`
- Hovedidéen bak denne typen join er å bevare alle rader fra en eller begge tabellene i joinen
- Og så fylle inn med `NULL` hvor vi ikke har noen match

Left Outer Join

- I en *left outer join* vil alle rader i den venstre tabellen bli med i svaret
- Resultatet av `a LEFT OUTER JOIN b ON (a.c1 = b.c2)` blir
  - samme som `a INNER JOIN b ON (a.c1 = b.c2)`,
  - men hvor alle rader fra `a` som ikke matcher noen i `b`
  - (altså hvor `a.c1` ikke er lik noen `b.c2`)
  - blir lagt til resultatet, med `NULL` for alle `b`s kolonner

Eksempel: Left Outer Join

Left outer join mellom `products` og `orders`

```
SELECT *
FROM products AS p LEFT OUTER JOIN orders AS o
     ON p.ProductID = o.ProductID;
```

Resultat

products

ProductID	Name	Price
0	TV 50 inch	8999
1	Laptop 2.5GHz	7499
2	Noise-amplifying Headphones	9999

orders

OrderID	ProductID	Customer
0	1	John Mill
1	1	Peter Smith
2	0	Anna Consuma
3	1	Yvonne Potter

Eksempel: Left Outer Join

Hvor mange har kjøpt hvert produkt?

```
    SELECT p.ProductName, count(o.Customer) AS num
    FROM products AS p LEFT OUTER JOIN orders AS o
        ON p.ProductID = o.ProductID
GROUP BY p.ProductName
```

Resultat

products

ProductID	Name	Price
0	TV 50 inch	8999
1	Laptop 2.5GHz	7499
2	Noise-amplifying Headphones	9999

orders

OrderID	ProductID	Customer
0	1	John Mill
1	1	Peter Smith
2	0	Anna Consuma
3	1	Yvonne Potter

Andre nyttige bruksområder for ytre joins

- Som vi ser er ytre joins nyttige når vi aggregerer, for å ikke miste resultater underveis
- Ytre joins kan også være nyttige for å kombinere ufullstendig informasjon fra flere tabeller
- For eksempel:

Persons

ID	Name
1	Per
2	Mari
3	Ida

Numbers

ID	Phone
1	48123456
3	98765432

Emails

ID	Email
1	per@mail.no
2	mari@umail.net

```
SELECT p.Name, n.Phone, e.Email
FROM Persons AS p
     LEFT OUTER JOIN Numbers AS n
       ON (p.ID = n.ID)
     LEFT OUTER JOIN Emails AS e
       ON (p.ID = e.ID);
```

p.Name	n.Phone	e.Email
Per	48123456	per@mail.no
Mari	NULL	mari@umail.net
Ida	98765432	NULL

Andre ytre joins

- `a RIGHT OUTER JOIN b ON (a.c1 = b.c2)` er akkurat det samme som `b LEFT OUTER JOIN a ON (b.c2 = a.c1)`
- Altså, i en *right outer join* vil alle radene i den høyre tabellen være med i resultatet
- Vi har også en `FULL OUTER JOIN` som er en slags kombinasjon, her vil ALLE rader være med i svaret
- For eksempel:

Persons

ID	Name
1	Per
2	Mari

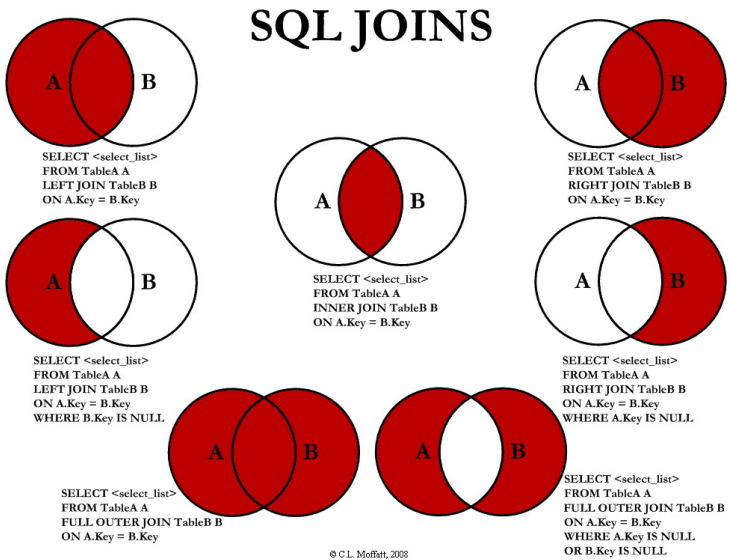
Numbers

ID	Phone
1	48123456
3	98765432

```
SELECT p.Name, n.Phone
FROM Persons AS p
     FULL OUTER JOIN Numbers AS n
       ON (p.ID = n.ID);
```

p.Name	n.Phone
Per	48123456
Mari	NULL
NULL	98765432

Oversikt over joins



Ytre join-eksempel (1)

Finn navn på alle kunder som har gjort 2 eller færre bestillinger

```
  SELECT c.company_name, count(o.order_id) AS num_orders
    FROM customers AS c
         LEFT OUTER JOIN orders AS o USING (customer_id)
GROUP BY c.company_name
  HAVING count(o.order_id) <= 2;
```

Ytre join-eksempel (2)

Finn ut for hvor mange produkter i hver kategori firmaet Leka Trading supplier

```
WITH
  supplies AS (
    SELECT category_id
    FROM suppliers INNER JOIN products USING (supplier_id)
    WHERE company_name = 'Leka Trading'
  )
SELECT c.category_name, count(s.category_id) AS nr_products
FROM categories AS c
     LEFT OUTER JOIN supplies AS s USING (category_id)
GROUP BY c.category_name;
```

Syntaks for joins

I stedet for

- `LEFT OUTER JOIN` kan man skrive `LEFT JOIN`
- `RIGHT OUTER JOIN` kan man skrive `RIGHT JOIN`
- `FULL OUTER JOIN` kan man skrive `FULL JOIN`
- `INNER JOIN` kan man skrive `JOIN`

Takk for nå!

Neste video handler om mengdeoperatorer.

IN2090 – Databaser og datamodellering

10 – Mengdeoperatorer

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Mengdeoperatorer

- Vi har nå et relativt uttrykningskraftig språk for å hente ut informasjon fra en database
- Men det er noen elementære ting vi fortsatt ikke kan gjøre
- F.eks. kombinere svar fra to spørringer til én tabell
- Eller trekke svarene fra en spørring fra en annen
- Husk at vi kan se på svarene fra `SELECT` som en (multi-)mengde
- SQL tillater oss å bruke vanlige mengdeoperatorer (snitt, union, osv.)
- Ettersom SQLs tabeller er multimengder har vi to versjoner av hver operator:
  - én versjon som behandler resultatene som mengder (f.eks. `UNION`)
  - én versjon som behandler dem som multimengder (f.eks. `UNION ALL`)
- Disse mengdeoperatorene puttes *mellom to spørringer*

Mengdeoperatorene

- Vi har følgende mengdeoperatorer:
  - Union – `UNION`
  - Snitt – `INTERSECT`
  - Differanse – `EXCEPT`
- For alle disse har vi i tillegg en variant med `ALL` etter seg som behandler resultatene som multimengder
- Antall ganger en rad er med i resultatet av:
  - `q1 UNION ALL q2` er summen av antall ganger raden er med i `q1` og `q2`
  - `q1 INTERSECT ALL q2` er det minste antall ganger raden er med i `q1` og `q2`
  - `q1 EXCEPT ALL q2` er antall ganger raden er med i `q1` minus antallet ganger den er med i `q2`

Union-operatoren

persons

ID	Name	Phone	Email
1	Per	48123456	per@mail.no
2	Mari	NULL	mari@umail.net
3	Ola	NULL	NULL
4	Ida	98765432	NULL

```
(SELECT *
 FROM persons
 WHERE Phone IS NOT NULL)
UNION
(SELECT *
 FROM persons
 WHERE Email IS NOT NULL)
```

Resultat:

ID	Name	Phone	Email
1	Per	48123456	per@mail.no
4	Ida	98765432	NULL
2	Mari	NULL	mari@umail.net

```
(SELECT *
 FROM persons
 WHERE Phone IS NOT NULL)
UNION ALL
(SELECT *
 FROM persons
 WHERE Email IS NOT NULL)
```

Resultat:

ID	Name	Phone	Email
1	Per	48123456	per@mail.no
4	Ida	98765432	NULL
1	Per	48123456	per@mail.no
2	Mari	NULL	mari@umail.net

Union-kompatibilitet

- Hva skjer om vi tar unionen av to spørringer som returnerer forskjellig antall kolonner?

```
(SELECT Name, Phone
 FROM person
 WHERE Phone IS NOT NULL)
UNION
(SELECT Name, Phone, Email
 FROM person
 WHERE Email IS NOT NULL)
```

- Vi får en error! Spørringen gir ikke mening.
- For å ta unionen av to spørringer må de returnere like mange kolonner
- Kolonnene må også ha kompatible typer
- Kan f.eks. ta unionen av en kolonne med `integer` og `decimal`, får da en kolonne av typen `numeric`
- Alle mengdeoperatorer må ha union-kompatibilitet mellom delspørringene

Eksempel: Union

Finn navn og by på alle leverandør- (eng.: supplier) og kundefirmaer fra Tyskland

```
(SELECT company_name, city
 FROM customers
 WHERE country = 'Germany')
UNION
(SELECT company_name, city
 FROM suppliers
 WHERE country = 'Germany');
```

Snitt-operatoren

persons

ID	Name	Country
1	Per	UK
2	Mari	Norway
3	Ola	Norway
4	Ida	Italy
5	Carl	USA

companies

ID	Name	Country
1	Per's company	Germany
2	Fish'n trolls	Norway
3	Matpakke AS	Norway
4	Big Burgers	USA
5	Ysteriet	Norway

```
(SELECT Country
 FROM persons)
INTERSECT
(SELECT Country
 FROM companies)
```

Resultat:

Country
Norway
USA

```
(SELECT Country
 FROM person)
INTERSECT ALL
(SELECT Country
 FROM companies)
```

Resultat:

Country
Norway
Norway
USA

Differanse-operatoren

persons

ID	Name	Country
1	Per	UK
2	Mari	Norway
3	Ola	Norway
4	Ida	Italy
5	Carl	USA

companies

ID	Name	Country
1	Per's company	Germany
2	Fish'n trolls	Norway
3	Matpakke AS	Norway
4	Big Burgers	USA
5	Ysteriet	Norway

```
(SELECT Country
 FROM companies)
EXCEPT
(SELECT Country
 FROM persons)
```

Resultat:

Country
Germany

```
(SELECT Country
 FROM companies)
EXCEPT ALL
(SELECT Country
 FROM persons)
```

Resultat:

Country
Germnay
Norway

EXISTS

- Av og til er vi kun interessert i om en del spørring *har et svar*, og ikke svaret i seg selv
- Typisk er dette når vi er interessert i å hente ut objekter med en bestemt egenskap, men hvor egenskapen kan avgjøres med en delspørring
- I slike tilfeller kan vi bruke `EXISTS` før en delspørring i `WHERE`-klausulen
- `EXISTS q` er sann for en spørring `q` dersom `q` har minst ett svar
- Kan også bruke `NOT EXISTS q` for å finne ut om `q` ikke har noen svar

EXISTS-nøkkelordet

companies

ID	Name	Country
1	Per's company	Germany
2	Fish'n trolls	Norway
3	Matpakke AS	Norway
4	Big Burgers	USA
5	Ysteriet	Norway

persons

ID	Name	Country
1	Per	UK
2	Mari	Norway
3	Ola	Norway
4	Ida	Italy
5	Carl	USA

```sql
SELECT p.Name
FROM persons AS p
WHERE NOT EXISTS (
        SELECT * -- Kan bruke hva som helst her
        FROM companies AS c
        WHERE c.country = p.country
      );
```

Resultat:

p.Name
Per
Ida

Mange måter å gjøre det samme på

Finn ID på alle kunder som ikke har bestilt noe:

Med `EXCEPT`

```
(SELECT customer_id
 FROM customers)
EXCEPT
(SELECT customer_id
 FROM orders);
```

Med `NOT IN`

```
SELECT customer_id
FROM customers
WHERE customer_id NOT IN (
  SELECT customer_id
  FROM orders);
```

Med `NOT EXISTS`

```
SELECT c.customer_id
FROM customers AS c
WHERE NOT EXISTS (
 SELECT * FROM orders AS o
 WHERE o.customer_id = c.customer_id
);
```

Med `LEFT OUTER JOIN`

```
SELECT c.customer_id
FROM customers AS c
     LEFT OUTER JOIN orders AS o
     USING (customer_id)
WHERE o.customer_id IS NULL;
```

Mye vi ikke har sett på

Følgende er nyttige ting vi ikke har sett på (og ikke del av pensum):

- Viduspørringer
  `https://www.postgresql.org/docs/current/tutorial-window.html`
- Rekursive spørringer
  `https://www.postgresql.org/docs/current/queries-with.html`
- Lateral join
  Sek. 7.2.1.5 i
  `https://www.postgresql.org/docs/current/queries-table-expressions.html`
- Triggere
  `https://www.postgresql.org/docs/current/plpgsql-trigger.html`
- Lage egne SQL-funksjoner og typer
  `https://www.postgresql.org/docs/current/xfunc.html`
  `https://www.postgresql.org/docs/current/xtypes.html`

Takk for nå!

Neste uke handler om programmering med SQL.

IN2090 – Databaser og datamodellering

11 – Programmering med SQL

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Programmering med databaser

- Som oftest er det ikke mennesker som manuelt skriver SQL
- Men programmer som genererer spørringer som de sender til databasen
- Spørringene kan da genereres basert på bruker-input, hendelser, el.
- Naturlig indeling av frontend og backend:
  - Frontend håndterer input fra bruker, visualiserer resultater, osv.
  - Backend svarer på spørringer, utfører kompliserte beregninger, osv.

Eksempel

Går inn på `http://finn.no`'s "Bolig til salgs" og setter:

- Sted: Oslo eller Akershus
- Makspris: 5,000,000,-
- Minste pris: 3,000,000,-
- Antall rom: 3

og klikker "Søk"

Generert (mulig) SQL-spørring:

```
SELECT *
  FROM boliger
 WHERE (sted = 'Oslo'
        OR sted = 'Akershus')
       AND pris <= 5000000
       AND pris >= 3000000
       AND ant_rom >= 3;
```

Generelle prinsipper

- Programmer håndterer SQL-spørringer som strenger
- Kan dermed manipulere SQL-spørringer akkurat som strenger
- For å kunne sende en spørring til en database trenger man to ting:
  - En tilkobling – `Connection`
  - En eller flere spørrings-eksekverere – `Cursor/Statement`

`Connection`

- `Connection`-objekter er ansvarlige for å lage en tilkobling til databasen
- Input til disse er databasenavn, brukernavn, passord, host, osv.
- Når tilkobligen er vellykket kan man begynne å lage spørrings-eksekverere fra en `Connection`

Spørrings-eksekverere

- Lages fra en `Connection`
- Gis en spørring som en streng
- Kan så eksekvere spørringen via et metode-kall (typisk `execute()`)
- Kan så hente ut svarene fra spørringen

Python og Psycopg2

- Biblioteket for intraksjon med PostgreSQL fra Python heter `psycopg`[1]
- Man starter med å lage et `Connection`-objekt[2] ved å kalle `psycopg2.connect(connection)`
- Fra dette lager man så `Cursor`-objekter[3] (via `Connections` `cursor()`-metode)
- `Cursor`-objetket kan så ekvekvere spørringer via `execute(query)` hvor `query` er en streng som inneholder en SQL-spørring
- Spørringene kan så hentes ut som vanlige Python-lister av tupler ved å kalle `cursor.fetchall()`

[1] `http://psycopg.org/docs/`
[2] `http://psycopg.org/docs/connection.html`
[3] `http://psycopg.org/docs/cursor.html`

Java og JDBC

- Biblioteket for interaksjon med databaser fra Java heter JDBC
- Egen driver for PostgreSQL som lastes inn med `Class.forName("org.postgresql.Driver")`
- Kan lage `Connection`-objekt[4]-objekt ved å kalle `DriverManager.getConnection(<conStr>)` hvor `<conStr>` er en streng som inneholder en URI med tilkoblingsdetaljer
- Kan så lage `Statement`[5]/`PreparedStatement`[6]-objekter ved å kalle `connection.createStatement()` eller `connection.prepareStatement()`
- En spørring eksekveres ved å kalle `statement.execute()`
- Resultatene fra en spørring kommer i form av et `ResultSet`[7]

[4] `https://docs.oracle.com/javase/8/docs/api/java/sql/Connection.html`
[5] `https://docs.oracle.com/javase/8/docs/api/java/sql/Statement.html`
[6] `https://docs.oracle.com/javase/8/docs/api/java/sql/PreparedStatement.html`
[7] `https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html`

ResultSet

- Et `ResultSet` holder alltid en peker til én rad i resultatet
- Man kan hoppe videre til neste rad ved å kalle metoden `next()`
- Denne metoden returnerer en boolsk verdi som er usann dersom det ikke finnes flere rader i resultatet
- For hver mulige type har man en egen `get`-metode (f.eks. `getString()`, `getInt()`) som tar en `int` som argument som er kolonne-nummeret
- Så `result.getString(2)` henter ut verdien i kolonne 2 i den nåværende raden, som en streng

Takk for nå!

Neste video introduserer eksempelet vi skal bruke videre.

IN2090 – Databaser og datamodellering

11 – Webshop-eksempel

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Webshop

- Vi skal lage programmer for en nettbutikk
- Skal i videoene lage program som lar brukere
  - Registrere ny bruker
  - Logge inn
  - Søke etter produkter
  - Bestille produkter
- I obligen skal dere lage to programmer
  - Ett som lar ansatte legge inn nye kategorier og produkter
  - Ett som lager regninger for kunder
- Viser både Python og Java
- For obligen velger dere enten Python eller Java

Antagelser og forenklinger

- Enkle tekst-baserte programmer (ingen fancy GUI)
- Kommer til å bruke deres personlige databaser
- Putter vi brukernavn og passord rett inn i kildekoden
- Ønsker ikke å lime inne passord for deres UiO-bruker!
- Bruk derfor de genererte brukerne tilsendt på mail (som har brukernavn som slutter på `_priv`)

Webshop – Databasen: Users

```
CREATE SCHEMA ws;

CREATE TABLE ws.users (
    uid SERIAL PRIMARY KEY,
    username varchar(20) UNIQUE NOT NULL CHECK (username ~ '[^\s]+'),
    password text NOT NULL, -- NEVER store passwords in plain text in real applications!!
    name text NOT NULL,
    address text NOT NULL
);
```

Webshop – Databasen: Categories

```
CREATE TABLE ws.categories (
    cid SERIAL PRIMARY KEY,
    name text UNIQUE NOT NULL
);
```

Webshop – Databasen: Products

```
CREATE TABLE ws.products (
    pid SERIAL PRIMARY KEY,
    name text NOT NULL,
    price float NOT NULL CHECK (price >= 0),
    cid int REFERENCES ws.categories(cid),
    description text
);
```

Webshop – Databasen: Orders

```
CREATE TABLE ws.orders (
    oid SERIAL PRIMARY KEY,
    uid int REFERENCES ws.users(uid),
    pid int REFERENCES ws.products(pid),
    num int NOT NULL,
    date date NOT NULL,
    payed int NOT NULL CHECK (payed = 0 OR payed = 1)
);
```

Takk for nå!

De neste to videoene implementerer Webshop-programmene i Python og Java.

IN2090 – Databaser og datamodellering

12 – Sikkerhet i databaser

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Hovedmål med databasesikkerhet

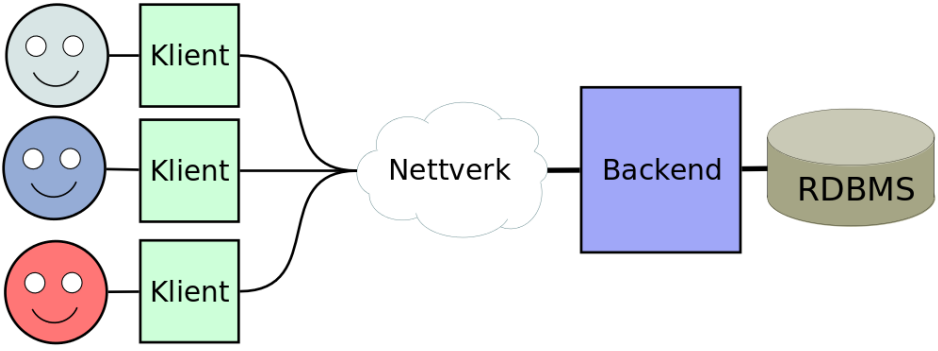
- Konfidensialitet
  - Uvedkommende må ikke kunne se data de ikke skal ha tilgang til
- Intergritet
  - Data må være korrekte og pålitelige. Derfor må data beskyttes mot endringer fra uautoriserte brukere
- Tilgjengelighet
  - Brukere må kunne se eller modifiesere data de har fått tilgang til

Sikkerhet: Ikke bare i databasesystemet

- Programmer inneholder ofte mye mer enn bare databasen
- Databasesikkerhet kan derfor ikke kun fokusere på databasen
- Sikkerhetshull kan forekomme i alle ledd (frontend, backend, nettverket, osv.)
- Sikkerhet er derfor alltid en *helhetlig* oppgave
- Må derfor sikre hver enkelt komponent og interaksjonen mellom dem
- Må være tydelige på hvilke antagelser om andre komponenter hver del av systemet gjør

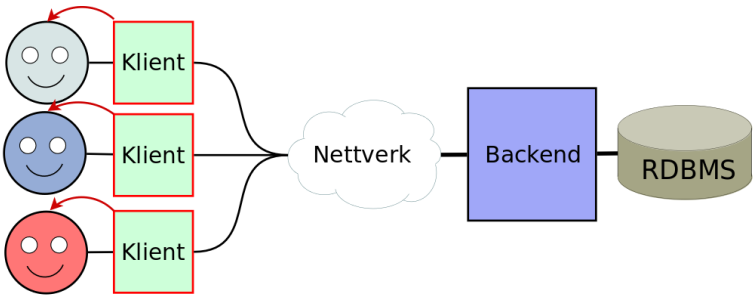


Oversikt over systemer med databaser



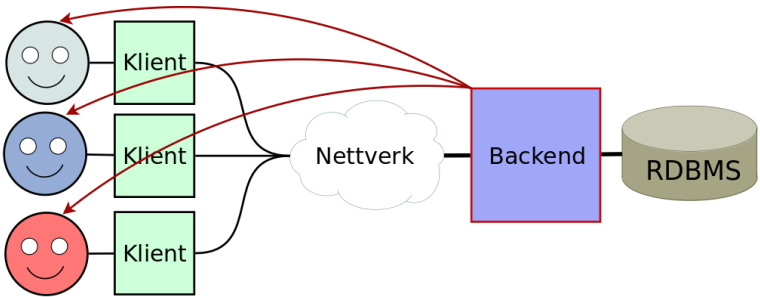
Tilgangskontroll: Klient/Frontend

- Klienten autentiserer brukerne
- Klienten sjekker hva brukeren har lov til
- Backend og RDBMS må stole på at klienten gjør dette riktig
- Trenger sikring slik at bare klienten kan få tilgang til backend/RDBMS



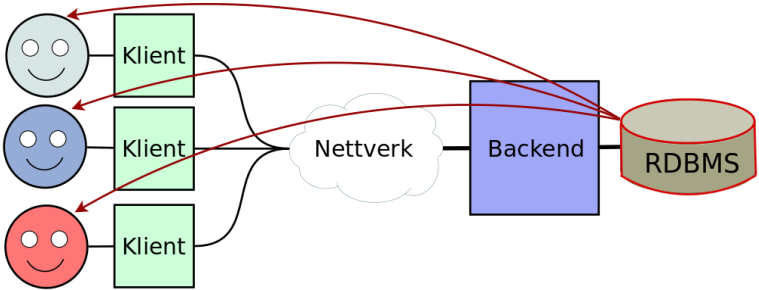
Tilgangskontroll: Backend

- Backend autentiserer brukerne
- Backend sjekker hva brukeren har lov til
- Backend har ofte én bruker til databasen
- RDBMS må stole på at backend gjør dette riktig
- Backend og RDBMS ofte bak samme brannmur



Tilgangskontroll: Database

- Databasesystemet autentiserer brukerne direkte
- Hver bruker av programmet får da hver sin databasebruker
- Klienten kan forhåndssjekke (f.eks. for å tilpasse brukergrensesnittet)



Tilgangskontroll i databaser

- Tilgang til databasen kontrolleres gjennom tre ting:
  - brukere
  - roller
  - rettigheter
- For eksempel:
  - Bruker `leifhka` har rollen `kundeadmin`
  - Bruker `leifhka` har rollen `produktansvarlig`
  - Bruker `klient` har rollen `kunde`
  - Brukere med rollen `kundeadmin` kan opprette og oppdatere kunder (rader i `customer`-tabellen)
  - Brukere med rollen `produktansvarlig` kan opprette, oppdatere og slette produkter (rader i `products`-tabellen)
  - Brukere med rollen `kunde` kan se på produkter (rader i `products`-tabellen) samt legge inn ordre (rader i `orders`-tabellen)

Brukere vs. roller

- Mulig å gi hver bruker de rettighetene de skal ha
- Men vanskelig å holde rede på at hver bruker har de riktige rettighetene
- Spesielt om det er mange brukere og mange rettigheter
- Typisk vil mange brukere trenge samme rettigheter: Vanskelig å vedlikeholde
- Vi lager derfor roller som fanger en mengde med rettigheter som hører sammen
- Og gir deretter brukere de passende rollene
- Dette heter *Role-based Access Control*

Databasebrukere

- F.eks. når dere logger dere inn i databasen med:

  ```
  $ psql -h dbpg-ifi-kurs01 -U leifhka -d fdb
  ```

  er `leifhka` brukeren
- Autentisering skjer typisk via passord, SSH public keys, el.
- Gydige brukernavn og (krypterte) passord lagres av RDBMS
- Autentisering kan delegeres til andre systemer
- Alle databaser, skjema, tabeller, views, osv. eies av en bruker

Lage brukere og roller med SQL

- For å lage en ny bruker `leifhka` med passord `hemmelig` og rollene `kundeadmin` og `produktansvarlig` kan man kjøre følgende SQL-kommando[1]

```
CREATE USER leifhka WITH PASSWORD 'hemmelig'
    ROLE kundeadmin, produktansvarlig;
```

- Roller lages nesten helt likt[2]:

```
CREATE ROLE produktansvarlig;
```

- I PostgreSQL er `CREATE USER` bare et alias for `CREATE ROLE` med `LOGIN`-adgang (mao. brukere er bare en spesiell type rolle)
- Roller og brukre slettes med `DROP`
- Merk: Som oftest bare superbrukere som kan lage brukere/roller

[1] se `https://www.postgresql.org/docs/12/sql-createuser.html`
[2] se `https://www.postgresql.org/docs/12/sql-createrole.html`

Begrense bruk

- Kan bergrense hvor lenge en bruker eller rolle skal være gyldig ved å sette `VALID UNTIL '2021-01-01'` i kommandoene over
- Kan begrense antall tilkoblinger en bruker/rolle kan ha ved å sette `CONNECTION LIMIT 5` i kommandoene over
- Dette er det som gjør at noen av dere har fått feilmeldingen:

```
psql: FATAL:  too many connections for role "user_name"
```

- For å gi en bruker/rolle (generelle) rettigheter til å lage databaser, roller, osv. kan man legge til `CREATEDB`, `CREATEROLE`, osv.

Gi og fjerne rettigheter

- Man kan gi roller/brukere mer detaljerte rettigheter via `GRANT`-kommandoen[3]
- `GRANT`-kommandoen har følgende form:

```
GRANT <privileges> ON <object> TO <role>;
```

- hvor `<privileges>` f.eks.:

```
SELECT, UPDATE, INSERT, DELETE, CREATE, CONNECT, USAGE, ALL
```

- og `<object>` er f.eks. en database, en tabell, et skjema, el.
- Gir man rettigheter til en rolle, vil alle dens medlemmer også få disse
- Fjerning av rettigheter kan gjøres tilsvarende med `REVOKE`

[3] `https://www.postgresql.org/docs/12/sql-grant.html`

GRANT-eksempler

- For å gi rollen `kundeadmin` rettighetene til å oprette og oppdatere `ws.users`-tabellen kan vi kjøre følgende kommando:

```
GRANT INSERT, UPDATE ON TABLE ws.users TO kundeadmin;
```

- For å gi rollen `webshopadmin` alle rettigheter innenfor skjemaet `ws`:

```
GRANT ALL ON SCHEMA ws TO webshopadmin;
```

- Kan også gi en bruker en ny rolle med `GRANT`:

```
GRANT kundeadmin TO leifhka;
```

- Kan til og med gi tillatelser på kolonnenivå:

```
GRANT UPDATE (price) ON ws.products TO prisansvarlig;
```

- For å fjerne `kunde`-rollens tilgang til `categories` kan vi kjøre

```
REVOKE USAGE ON ws.categories FROM kunde;
```

Tilgang og views

- I enkelte tilfeller ønsker vi ikke gi tilgang til tabellene direkte
- Men f.eks. kun aggregerte eller utvalgte verdier
- F.eks. vil ikke gi tilgang til pasientjournalen til hver enkelt person, men heller antall med ulike sykdommer per kommune
- Kan da lage views, og så gi tilgang til disse

Takk for nå!

Neste video handler om sikkerhet i programmer som bruker databaser.

IN2090 – Databaser og datamodellering

12 – Sikkerhet i programmer som bruker databaser

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

Databasetilkoblinger

- `Connection`-objektene (og de tilhørende `cursor` eller `Statement` og `ResultSet`) er ressurser som kan føre til minnelekasjer
- Alle disse har en egen metode som heter `close()` som stenger ressursen og frigjør den
- For de enkle programmene vi så sist uke var dette ikke veldig viktig, ettersom ressursene blir frigjort når programmet avsluttes
- Men for større programmer og tjenester som skal kjøre over lengre tid bør man alltid sørge for at tilkoblingene blir stengt med en gang man er ferdige med dem

Stenge tilkoblinger

- I Python har både `Connection` og `Cursor` en `close()` metode
- I Java har både `Connection`, `Statement`/`PreparedStatement` og `ResultSet` egne `close()` metoder
- Generelt blir alle objekter stengt når objektet det er laget fra stenges
- F.eks. vil en `PreparedStatement` stenges dersom dens `Connection` stenges

Automatisk stenge ressurser

- Java kan bruke `try-with`-blokker (fom. Java 7) for automatisk å stenge tilkoblinger
- F.eks.:

```
try (Connection con = DriverManager.getConnection(conStr);
    PreparedStatement stmt = con.prepareStatement(query)) {
    try (ResultSet res = stmt.executeQuery()) {
        // Do stuff with the result set.
    }
} catch (...) {
    // errors
} // con, stmt, res closed here
```

- Har tilsvarende i Python:

```
with psycopg2.connect(dsn) as conn:
    with conn.cursor() as cur:
        cur.execute(sql)
```

men dette stenger ikke tilkoblingen (kun eventuelle åpne transaksjoner)

SQL injections

- SQL injection er en type angrep mot en klient/backend hvor en (ondsinnet) bruker får kjørt egen SQL-kode på databasen
- Brukeren kan da forbigå autentisering eller hente ut, endre eller slette data brukeren egentlig ikke skal ha tilgang til
- SQL injection utnytter følgende faktorer:
  - At vi har et program som kjører SQL-spørringer (f.eks. Java eller Python)
  - Programmet tar input fra brukeren som skal være en del av spørringene
  - Programmet ikke skiller mellom data og kommandoer i spørringen

SQL injections: Grunnleggende prinsipp

- La oss si at en nettbutikk lar brukere søke etter varer på følgende måte:

```
product = input("Product: ");
cur.execute("SELECT * FROM products WHERE navn = '" + product + "';");
```

- Med input `Socks` blir spørringen:

```
SELECT * FROM products WHERE navn = 'Socks';
```

- Med input `O'Boy` får vi error:

```
SELECT * FROM products WHERE navn = 'O'Boy';
```

- Med input `Socks'; DROP TABLE products;--` får vi

```
SELECT * FROM products WHERE navn = 'Socks'; DROP TABLE products; --';
```

SQL injections: Webshop-eksempel

Live-kode-eksempel!

Hvorfor er dette viktig?

- En av de vanligste formene for hackerangrep
- Dersom angrepet lykkes vil det gi den ondsinnede brukeren veldig mye makt:
  - Ødelegge/slette data
  - Endre data
  - Hente ut konfidensiell informasjon
  - Hindre tjenesten i å fungere (DOS)
- Veldig enkelt å forhindre med parametriserte spørringer!

Helhetlig sikkerthet

- Merk at selvom klienten skulle være sårbar for SQL injection-angrep kan databasen likevel forhindre mye skade om man har satt riktige rettigheter
- F.eks. dersom databasebrukeren som klienten benytter ikke har tilgang til å slette eller endre tabeller eller spørre mot vilkårlige tabeller
- Dette viser hvor viktig det er at alle ledd er sikret og at man ikke bør anta for mye av andre komponenter i et system

Takk for nå!

Neste uke handler om indekser og spørreprossesering.

IN2090 – Databaser og datamodellering

13 – Indekser og Spørreprosessering

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

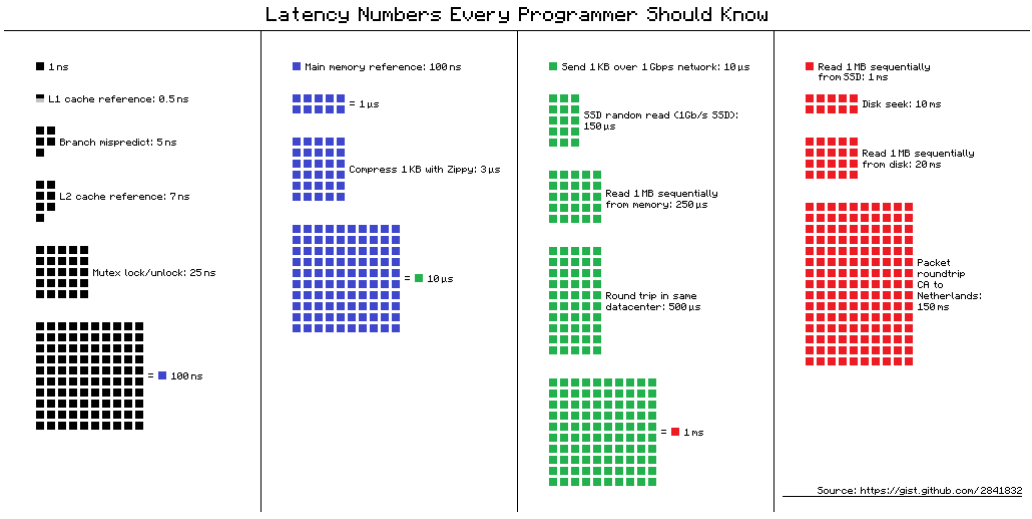
Spørringer og kompleksitet

- Hvordan utfører en database et oppslag på en bestem verdi?
- Eller en join mellom to tabeller?
- Begge disse problemene er egentlig et søk etter bestemte verdier i en kolonne
- For at databasen skal kunne utføre disse operasjonene effektivt (spesielt over veldig store tabeller) trenger vi datastrukturer som gjør søket mer effektivt
- Slike datastrukturer heter indeksstrukturer

Indeksstrukturer

- En indeksstruktur er en datastruktur som lar databasen hurtig finne bestemte rader i en tabell, basert på verdiene i en (eller flere) kolonner
- Husk at databaser lagrer data på disk, ikke i minne (RAM)
- Databaseindekser skiller seg litt fra andre datastrukturer fordi de ikke lagres i minne, men på disk

Lese fra harddisk

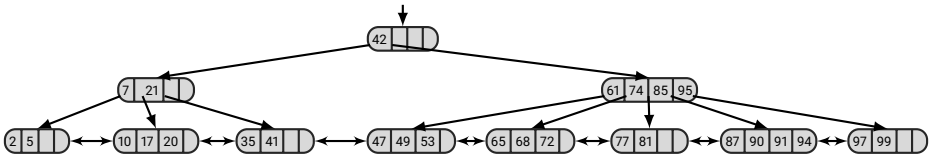


Hentet fra `https://gist.github.com/hellerbarde/2843375`

Lese fra harddisk

- Å lese fra disk tar ca. 10,000 ganger lengre tid enn fra RAM (avhenig av disktype og minnetype)
- Indeksstrukturer i databaser er derfor optimert for å utføre så få diskoppslag som mulig
- Har to hovedtyper indekser: Hash-baserte og tre-baserte

B-tre-indekser



- Trestruktur hvor hver node kan ha mange barn
- Nodene har samme størrelse som en disk-blokk
- Minimerer antall oppslag på disk
- Hver verdi i løvnodene har pekere til dens tilhørende rad i den tilhørende tabellen
- Kan utføre effektive oppslag på konkrete verdier
- Samt effektive intervall søk

Hash-indekser

- En hash-indeks bruker en hash-funksjon for å oversette en verdi til en minneadresse
- På minneadressen ligger så en liste med pekere til rader som har denne verdien
- Hash-indekser er mer effektive på oppslag på konkrete verdier
- Men kan ikke brukes for intervaller (må da gjøre ett oppslag for hver mulige verdi i intervallet)

Andre indeksstrukturer

- Det finnes mange andre indeksstrukturer
- Ulike strukturer er tilpassed ulike datatyper
- Egne strukturer for f.eks.:
  - Søk i tekst
  - Romlige data og koordinater i høyere dimensjoner
  - Sammensatte strukturer (JSON, XML, osv.)

Nøkler og indekser

- Når man markerer en kolonne med `PRIMARY KEY` blir det automatisk opprettet en B-tre-indeks på denne kolonnen
- Joins over primærnøkler er derfor alltid relativt effektive
- Men, det kan hende man ønsker å gjøre søk, oppslag eller joins over kolonner som ikke er primærnøkler
- Vi må da lage indeksene selv

Lage indekser med SQL

- For å lade en indeks på en kolonne trenger man bare å skrive

  ```
  CREATE INDEX <index_name> ON <table>(<columns>);
  ```

  hvor `<index_name>` er navnet på indeksen, `<table>` er et tabellnavn og `<columns>` er en liste med kolonner man ønsker å indeksere
- Databasen lager da en passende indeks (typisk B-tre)
- F.eks.:

  ```
  CREATE INDEX price_index ON products(unit_price);
  ```

- Merk: Dersom man lister opp flere kolonner blir indeksen over alle kolonnene samtidig
- Databasen finner selv ut når indeksen bør brukes

Indeks-demonstrasjon

```
DROP TABLE IF EXISTS t1;
DROP TABLE IF EXISTS t2;

CREATE TABLE t1(id int);
INSERT INTO t1
SELECT n*random() -- Genererer 10 mill. tilfeldige tall
FROM generate_series(1, 10000000) AS x(n);

CREATE TABLE t2 AS (SELECT * FROM t1); -- t2 inneholder samme data som t1

CREATE INDEX t1_ind ON t1(id); -- Lager index på t1

-- Følgende gjør at psql printer ut hvor lang tid spørringer tar
\timing on

SELECT * FROM t1 WHERE id = 100; --> Time:   0.792 ms
SELECT * FROM t2 WHERE id = 100; --> Time: 303.022 ms
```

Takk for nå!

Neste video handler om spørreprosessering.

IN2090 – Databaser og datamodellering

13 – Spørreprosessering

Leif Harald Karlsen
leifhka@ifi.uio.no



Universitetet i Oslo

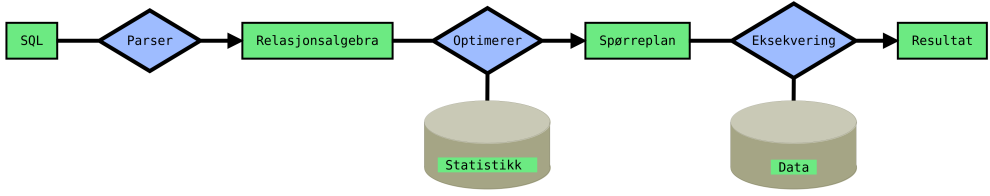
Spørreprosessering: Oversikt

```
SELECT p.name, c.name
  FROM products p INNER JOIN
       categories c USING (cid);
```

$$\pi_{p.name,c.name}\left(\rho_{p}\left(products\right) \bowtie_{p.cid=c.cid} \rho_{c}\left(categories\right)\right)$$

```
                                 QUERY PLAN
-------------------------------------------------------------------------------
Hash Join  (cost=1.18..3.26 rows=77 width=65)
  Hash Cond: (p.category_id = c.category_id)
  ->  Seq Scan on products p  (cost=0.00..1.77 rows=77 width=19)
  ->  Hash  (cost=1.08..1.08 rows=8 width=50)
        ->  Seq Scan on categories c  (cost=0.00..1.08 rows=8 width=50)
(5 rows)
```

```
      name      |      name
----------------+----------------
 Chai           | Beverages
 Chang          | Beverages
 Aniseed Syrup  | Condiments
 Chef Anton's Cajun Seasoning | Condiments
 Chef Anton's Gumbo Mix | Condiments
 Grandma's Boysenberry Spread | Condiments
 Uncle Bob's Organic Dried Pears | Produce
```



- En spørring går gjennom flere steg før den til slutt blir evaluert over dataene
- Disse stegene sørger for at spørringen blir besvart så effektivt som mulig

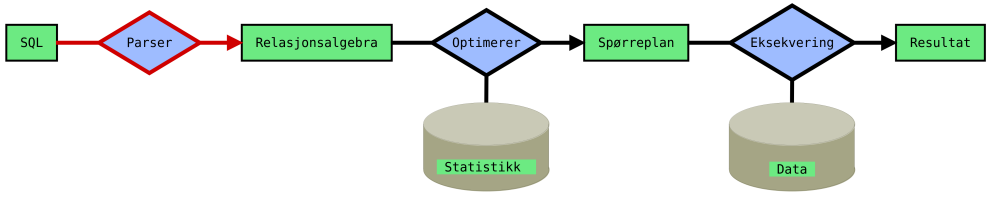
Fra SQL til relasjonell algebra

```
SELECT p.name, c.name
  FROM products p INNER JOIN
       categories c USING (cid);
```

$$\pi_{p.name,c.name}\left(\rho_{p}\left(products\right) \bowtie_{p.cid=c.cid} \rho_{c}\left(categories\right)\right)$$

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=1.18..3.26 rows=77 width=65)
   Hash Cond: (p.category_id = c.category_id)
   ->  Seq Scan on products p  (cost=0.00..1.77 rows=77 width=19)
   ->  Hash  (cost=1.08..1.08 rows=8 width=50)
         ->  Seq Scan on categories c  (cost=0.00..1.08 rows=8 width=50)
(5 rows)
```

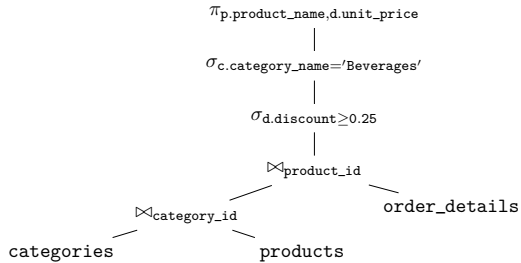
```
          name          |    name
------------------------+------------
 Chai                   | Beverages
 Chang                  | Beverages
 Aniseed Syrup          | Condiments
 Chef Anton's Cajun Seasoning | Condiments
 Chef Anton's Gumbo Mix | Condiments
 Grandma's Boysenberry Spread | Condiments
 Uncle Bob's Organic Dried Pears | Produce
```



- Det første som skjer er at spørringen sjekkes syntaktisk (f.eks. tabellene og kolonnene finnes i databasen, typene er riktige, osv.)
- Deretter oversettes spørringen til et spørre-tre over relasjons algebraen

Oversettelse: Eksempel

```
SELECT p.product_name, d.unit_price
FROM categories AS c JOIN
  products AS p USING (category_id) JOIN
  order_details AS d USING (product_id)
WHERE c.category_name = 'Beverages'
  AND d.discount >= 0.25;
```



$$\pi_{\texttt{p.product_name,d.unit_price}}\Big(\sigma_{\texttt{c.category_name}='\texttt{Beverages}'}\Big(\sigma_{\texttt{d.discount}\geq 0.25}\big(\texttt{categories} \bowtie_{\texttt{category_id}} \texttt{products} \bowtie_{\texttt{product_id}} \texttt{order_details}\big)\Big)\Big)$$

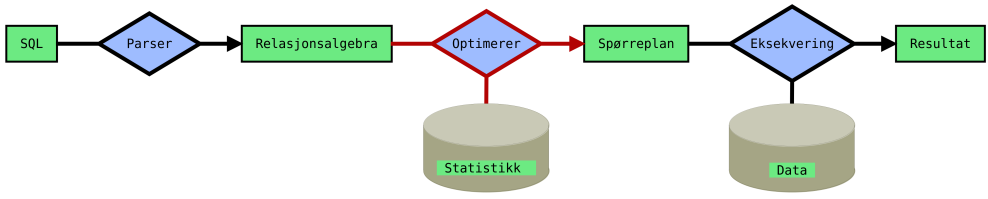
Ulike spørringer – Likt resultat

```
SELECT p.name, c.name
  FROM products p INNER JOIN
       categories c USING (cid);
```

$$\pi_{p.name,c.name}\left(\rho_{p}\left(products\right) \bowtie_{p.cid=c.cid} \rho_{c}\left(categories\right)\right)$$

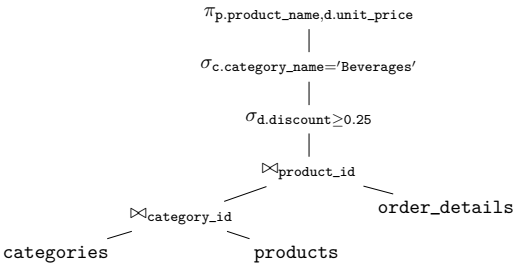
```
                                  QUERY PLAN
------------------------------------------------------------------------------
Hash Join  (cost=1.18..3.26 rows=77 width=65)
  Hash Cond: (p.category_id = c.category_id)
  ->  Seq Scan on products p  (cost=0.00..1.77 rows=77 width=19)
  ->  Hash  (cost=1.08..1.08 rows=8 width=50)
        ->  Seq Scan on categories c  (cost=0.00..1.08 rows=8 width=50)
(5 rows)
```

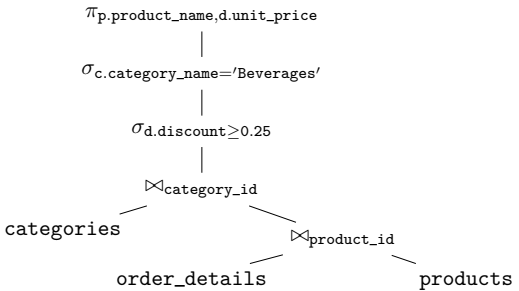
```
          name          |     name
------------------------+--------------
 Chai                   | Beverages
 Chang                  | Beverages
 Aniseed Syrup          | Condiments
 Chef Anton's Cajun Seasoning | Condiments
 Chef Anton's Gumbo Mix | Condiments
 Grandma's Boysenberry Spread | Condiments
 Uncle Bob's Organic Dried Pears | Produce
```

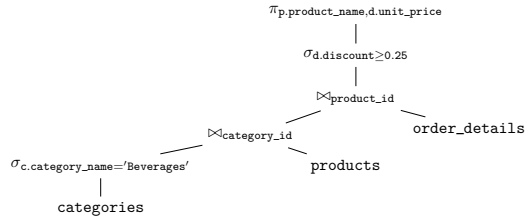


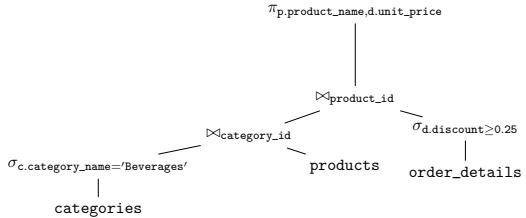
- Spørringen uttrykt i relasjonell algebra kan manipuleres algebraisk
- Dette brukes for å genere forskjellige men ekvivalente spørringer
- Altså, spørringer som gir samme svar, men ser forskjellige ut
- Forskjellige spørringer kan ha ulik kompleksitet
- De ulike spørringene skal så (i neste steg) bli tilordnet en ca. kostnad
- Vi vil så velge den spørringen som er billigst å eksekvere

Ulike spørringer: Eksempel









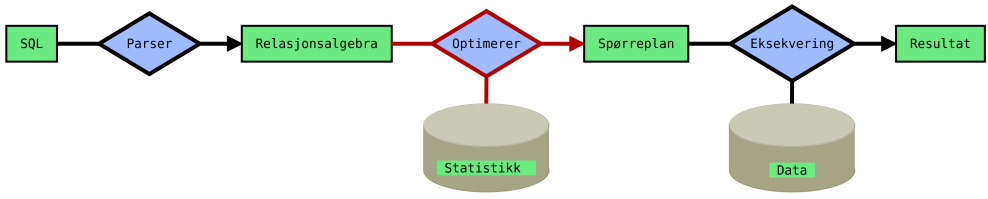
Fra spørring til kostnad

```
SELECT p.name, c.name
  FROM products p INNER JOIN
       categories c USING (cid);
```

$$\pi_{p.name,c.name}\left(\rho_{p}\left(products\right) \bowtie_{p.cid=c.cid} \rho_{c}\left(categories\right)\right)$$

```
                                  QUERY PLAN
-------------------------------------------------------------------------------
Hash Join  (cost=1.18..3.26 rows=77 width=65)
  Hash Cond: (p.category_id = c.category_id)
  ->  Seq Scan on products p  (cost=0.00..1.77 rows=77 width=19)
  ->  Hash  (cost=1.08..1.08 rows=8 width=50)
        ->  Seq Scan on categories c  (cost=0.00..1.08 rows=8 width=50)
(5 rows)
```

```
          name          |     name
------------------------+--------------
 Chai                   | Beverages
 Chang                  | Beverages
 Aniseed Syrup          | Condiments
 Chef Anton's Cajun Seasoning | Condiments
 Chef Anton's Gumbo Mix | Condiments
 Grandma's Boysenberry Spread | Condiments
 Uncle Bob's Organic Dried Pears | Produce
```



- De ulike spørringene blir så tilordnet en kostnad
- Kostnadsevalueringen bruker statistikk over databasen
- F.eks. antall rader i hver tabell, antall ulike verdier i hver kolonne, osv.
- Bruker her også skranker (f.eks. `UNIQUE`, `CHECK`) og indeksstrukturer
- Høyere kostnad betyr lengre eksekveringstid
- Databasen velger så den spørringen med lavest kostnad

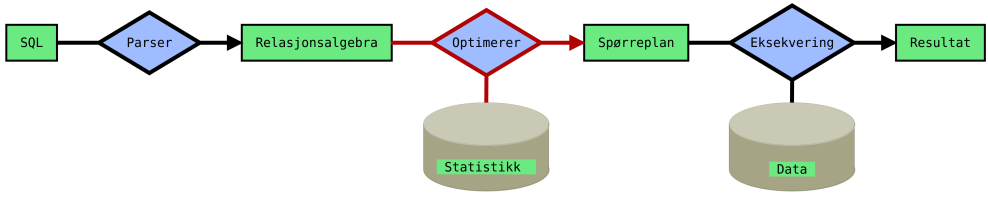
Spørreplaner

```
SELECT p.name, c.name
  FROM products p INNER JOIN
       categories c USING (cid);
```

$$\pi_{p.name,c.name}\left(\rho_{p}\left(products\right) \bowtie_{p.cid=c.cid} \rho_{c}\left(categories\right)\right)$$

```
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=1.18..3.26 rows=77 width=65)
   Hash Cond: (p.category_id = c.category_id)
   ->  Seq Scan on products p  (cost=0.00..1.77 rows=77 width=19)
   ->  Hash  (cost=1.08..1.08 rows=8 width=50)
         ->  Seq Scan on categories c  (cost=0.00..1.08 rows=8 width=50)
(5 rows)
```

```
          name          |    name
------------------------+------------
 Chai                   | Beverages
 Chang                  | Beverages
 Aniseed Syrup          | Condiments
 Chef Anton's Cajun Seasoning | Condiments
 Chef Anton's Gumbo Mix | Condiments
 Grandma's Boysenberry Spread | Condiments
 Uncle Bob's Organic Dried Pears | Produce
```



- Det siste som skjer i dette trinnet er at det blir laget en spørreplan for den valgte spørringen
- Dette er en mer detaljert plan for hvordan spørringen skal eksekveres

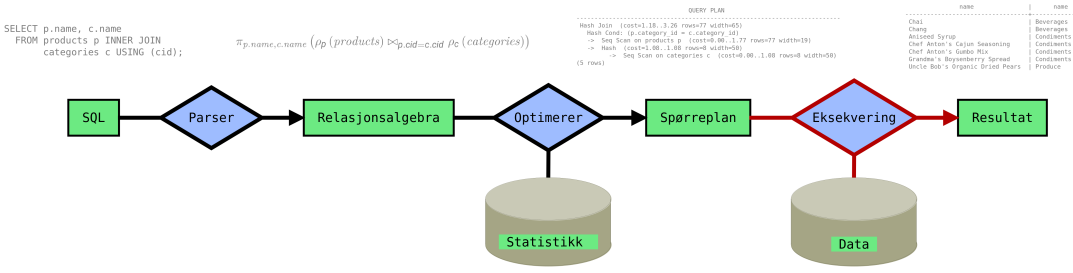
EXPLAIN

- Av og til kan det være nyttig å få se denne spørreplanen
- F.eks. dersom man lurer på hvordan spørringen vil bli eksekvert
- Eller dersom man ønsker et ca. estimat på hvor komplisert spørringen blir å eksekvere
- Dette kan gjøres ved å skrive `EXPLAIN` foran spørringen
- Spørringen blir da ikke eksekvert
- (Merk: Ikke pensum å kunne forstå spørreplaner!)

EXPLAIN: Eksempel

```
psql=> EXPLAIN SELECT p.product_name, d.unit_price
FROM categories AS c JOIN
  products AS p USING (category_id) JOIN
  order_details AS d USING (product_id)
WHERE c.category_name = 'Beverages'
  AND d.discount >= 0.25;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 Hash Join  (cost=3.32..43.04 rows=20 width=21)
   Hash Cond: (d.product_id = p.product_id)
   ->  Seq Scan on order_details d  (cost=0.00..38.94 rows=154 width=6)
         Filter: (discount >= '0.25'::double precision)
   ->  Hash  (cost=3.20..3.20 rows=10 width=19)
         ->  Hash Join  (cost=1.11..3.20 rows=10 width=19)
               Hash Cond: (p.category_id = c.category_id)
               ->  Seq Scan on products p  (cost=0.00..1.77 rows=77 width=21)
               ->  Hash  (cost=1.10..1.10 rows=1 width=2)
                     ->  Seq Scan on categories c  (cost=0.00..1.10 rows=1 width=2)
                           Filter: ((category_name)::text = 'Beverages'::text)
(11 rows)
```

Evaluering



- Til slutt evalueres spørringen over databasen
- Databasen har så svært effektive algoritmer for joins, oppslag, sorteing, osv.
- Merk: Databasen trenger kun én algoritme per operator i den (utvidede) relasjonelle algebraen

ANALYZE

- Dersom vi ønsker å vite hvor lang tid en spørring faktisk tar å eksekvere, samt detaljert analyse av hver del av spørreplanen kan vi bruke `EXPLAIN ANALYZE`
- Får da også informasjon om minnebruk
- Da vil spørringen bli eksekvert, og databasen samler så nøyaktig informasjon om eksekveringen
- Dersom en spørring tar lang tid kan dette bruker for å finne ut hvilken del av spørringen som er komplisert
- Kan også brukes for å finne manglende indeksstrukturer

ANALYZE: Eksempel

```
psql=> EXPLAIN ANALYZE SELECT p.product_name, d.unit_price
FROM categories AS c JOIN
  products AS p USING (category_id) JOIN
  order_details AS d USING (product_id)
WHERE c.category_name = 'Beverages'
  AND d.discount >= 0.25;
                                                              QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=3.32..43.04 rows=20 width=21) (actual time=0.130..1.066 rows=32 loops=1)
   Hash Cond: (d.product_id = p.product_id)
   ->  Seq Scan on order_details d  (cost=0.00..38.94 rows=154 width=6) (actual time=0.031..0.887 rows=154 loops=1)
         Filter: (discount >= '0.25'::double precision)
         Rows Removed by Filter: 2001
   ->  Hash  (cost=3.20..3.20 rows=10 width=19) (actual time=0.085..0.085 rows=12 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 9kB
         ->  Hash Join  (cost=1.11..3.20 rows=10 width=19) (actual time=0.034..0.077 rows=12 loops=1)
               Hash Cond: (p.category_id = c.category_id)
               ->  Seq Scan on products p  (cost=0.00..1.77 rows=77 width=21) (actual time=0.008..0.022 rows=77 loops=1)
               ->  Hash  (cost=1.10..1.10 rows=1 width=2) (actual time=0.016..0.016 rows=1 loops=1)
                     Buckets: 1024  Batches: 1  Memory Usage: 9kB
                     ->  Seq Scan on categories c  (cost=0.00..1.10 rows=1 width=2) (actual time=0.008..0.012 rows=1 loops=1)
                           Filter: ((category_name)::text = 'Beverages'::text)
                           Rows Removed by Filter: 7
 Planning Time: 0.567 ms
 Execution Time: 1.146 ms
(17 rows)
```

Takk for nå!

Lykke til med forberedelsene til eksamen!