

# IN2090 NOTATER

Mohammad Fadel Al Khafaji

## SQL-spøringer

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)

## SELECT-spøringer

- ◆ (Enkle) **SELECT**-spøringer 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-spørninger 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>      ⇒       $\pi_{<kolonner>}(\sigma_{<uttrykk>}(<tabeller>))$   
     WHERE <uttrykk>
```

## 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
```



Alder
13
24
13
25
25

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

- ◆ 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
```

## 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

## 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

## 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 orden '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 tilfredsstiller 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`

## NULLs 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;
```

## 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 bruuke 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'`)

## 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%'
```

## 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

## 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

## Joins

---

- ◆ Spørninger 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) interesaert i å "joine" på nøkler

## Problemer med spørring over flere tabeller

---

Hvilken kunde har kjøpt hvilket produkt?

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

Resultat

products			orders		
ProductID	Name	Price	OrderID	ProductID	Customer
0	TV 50 inch	8999	0	1	John Mill
1	Laptop 2.5GHz	7499	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
- ◆ Feks. 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
- ◆ Feks. 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
```

## Eksempler 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>
```


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

## 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';
```

# Kombinere informasjon fra flere tabeller

---

- ◆ (Enkle) `SELECT`-spørninger har formen:

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

- ◆ Frem til nå har vi bare sett på spørninger 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

## Spørninger 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

## Delspøringer

---

- ◆ 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>
```

## Delspøringer 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

## Delspøringer 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 nøkkelordet 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')
```

## Ekempel-delspøringer

---

- ◆ 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 delspøringer som tabeller må gis et navn

## 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(11.Price - 12.Price) AS diff
  FROM (SELECT Price FROM products WHERE Name LIKE '%Laptop%') AS 11,
       (SELECT Price FROM products WHERE Name LIKE '%Laptop%') AS 12
```

- ◆ 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(11.Price - 12.Price) AS diff
  FROM laptops AS 11, laptops AS 12
```

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

## Typer SQL-spøringer

---

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)

## 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 skranner (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 eksempel:

```
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';
```

## Default-verdier

---

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

```
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!

## Eksempler på skrankeovertredelser (violations)

---

Som sagt tidlire, 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);
gir 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');
gir 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<sup>1</sup> for en oversikt

## 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

## 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'          ⇒
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'
```

## 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 spørninger mot som en vanlig tabell
- Lages slik:

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

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

```
REFRESH MATERIALIZED VIEW person_alder;
```

## SQL-scripts: 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

## 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 omsluttet 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:

- ◆ **Atomicity** – 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)
- ◆ **Consistency** – Dersom en transaksjon lykkes skal databasen ende opp i en konsistent tilstand (altså ingen skranner skal være brutt)
- ◆ **Isolation** – Transaksjoner skal kunne kjøres i parallel, men resultatet skal da være likt som om transaksjonene ble kjørt sekvensielt
- ◆ **Durability** – 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)

## Funksjonell avhengighet

---

- ◆ Et attributt  $A$  er **funksjonelt avhengig** av en 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  $\{\text{Brnavn}, \text{Kurskode}\}$  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

## 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 equivalent 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  $X^+ = X^+ \cup Z$

## 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 kandidatnøkkelen.
- ◆ 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.

## 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, 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:

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

- ◆ Brnavn → Navn, Etternavn, Adresse
  - ◆ Kurskode → Tittel, Beskrivelse, AntSP
  - ◆ Brnavn, Kurskode → Karakter
- 
- ◆ **Kandidatnøkkel:** Brnavn, Kurskode
  - ◆ **Navn** er avhengig av Brnavn og 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:

Ansatt(Id, Navn, Avdeling, AvdelingsKode)

Gitt FDene:

- ◆  $\text{Id} \rightarrow \text{Navn}, \text{AvdelingsKode}$
- ◆  $\text{AvdelingsKode} \rightarrow \text{Avdeling}$

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<sup>1</sup>:

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:

- ◆ Person, Type → Nærmeste
  - ◆ Nærmeste → Type

Her har vi:

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

<sup>1</sup>Inspiret av eksempel fra [https://en.wikipedia.org/wiki/Boyce%E2%80%93Codd\\_normal\\_form](https://en.wikipedia.org/wiki/Boyce%E2%80%93Codd_normal_form)

## 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  $X \rightarrow A$ :

1. Er  $X$  en supernøkkel?

Ja: BCNF så langt, gå til neste FD  
Nei: brudd på BCNF. Gå til 2.

2. Er  $A$  et nøkkelattributt?

Ja: 3NF så langt, gå til neste FD  
Nei: brudd på 3NF. Gå til 3.

3. Er  $X$  del av en kandidatnøkkel?

Nei: 2NF så langt, gå til neste FD  
Ja: brudd på 2NF og skjema er på 1NF, stopp.

### Finn normalformen:

S(Brnavn, Navn, Etternavn, Kurskode, KursTittel, Karakter)

FDer:

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

Kandidatnøkkel: {Brnavn, Kurskode}.

- ◆ FD 1: Brnavn, Kurskode er en supernøkkel, så BCNF så langt
- ◆ FD 2: Brnavn ikke supernøkkel, brudd på BCNF
- ◆ FD 2: Navn ikke nøkkelattributt, brudd på 3NF
- ◆ FD 2: Brnavn del av en kandidatnøkkel, brudd på 2NF

Relasjonen S er derfor på 1NF.

## 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.  $A, B \rightarrow C, D$  blir  $A, B \rightarrow C$  og  $A, B \rightarrow D$ )
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  $Y \rightarrow A \in F$  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  $Y \rightarrow A \in F$  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  $F = \{AB \rightarrow C, C \rightarrow A\}$ .

- ◆ Kanidatnøkkelen:  $B$  forekommer ikke på høyresider, så  $B$  er med i alle nøkler.  $\{A, B\}$  og  $\{B, C\}$  er nøklene
- ◆  $AB \rightarrow C$  er ikke brudd på BCNF, siden  $AB$  er en supernøkkel
- ◆  $C \rightarrow A$  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  $S_1$  ( $C \rightarrow A$ ) 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  $S_2(C, B)$

## 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 spørninger 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;
```

## 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: Regn ut aggregatet for hver gruppe og ferdigstill

avg(Price)	Category
10499	Televisions
6731	Computers
4999	Speakers

## 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;
```

## 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-spørninger nå slik ut:

```
WITH <navngitte-spøringer>
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**

## Enklere syntaks for joins

---

- ◆ Man kan bruke `USING (<kolonne>)` fremfor  
`ON (a.<kolonne> = b.<kolonne>)`
- ◆ For eksempel:

```
SELECT p.product_name, c.category_name
  FROM products AS p
    INNER JOIN categories AS c USING (category_id);
```

- ◆ Merk: Må fortsatt bruke `ON` dersom kolonnene har ulikt navn

## Utlede informasjon om entiteter

---

- ◆ Aggregering i grupper, sortering og å begrense svaret er svært nyttig når man har store mengder data
- ◆ Når vi grupperer kan vi enten utlede implisitt informasjon om allerede eksisterende entiteter
- ◆ Eller lage nye entiteter fra attributter
- ◆ Sortering og begrensning lar oss hente ut de mest interessante objektene
- ◆ Dette gjør også at vi kan lage langt mer interessante views

## Eksempel 2: Implisitt informasjon om land

### Finn de tre mest kjøpte produktene for hvert land

```
WITH
    bought_by_country AS (
        SELECT c.country, p.product_name, count(*) AS nr_bought
        FROM products AS p
            INNER JOIN order_details AS d USING (product_id)
            INNER JOIN orders AS o USING (order_id)
            INNER JOIN customers AS c USING (customer_id)
        GROUP BY c.country, p.product_name
        ORDER BY nr_bought DESC
    ),
    countries AS (
        SELECT DISTINCT country FROM customers
    )
SELECT c.country,
    (SELECT s.product_name FROM bought_by_country AS s
        WHERE s.country = c.country
        LIMIT 1) AS first_place,
    (SELECT s.product_name FROM bought_by_country AS s
        WHERE s.country = c.country
        OFFSET 1
        LIMIT 1) AS second_place,
    (SELECT s.product_name FROM bought_by_country AS s
        WHERE s.country = c.country
        OFFSET 2
        LIMIT 1) AS third_place
FROM countries AS c;
```

## 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
```

## 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 bs kolonner

## Andre nyttige bruksområder for ytre joins

---

- ◆ Som vi ser er ytre joins nyttige når vi aggererer, 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:

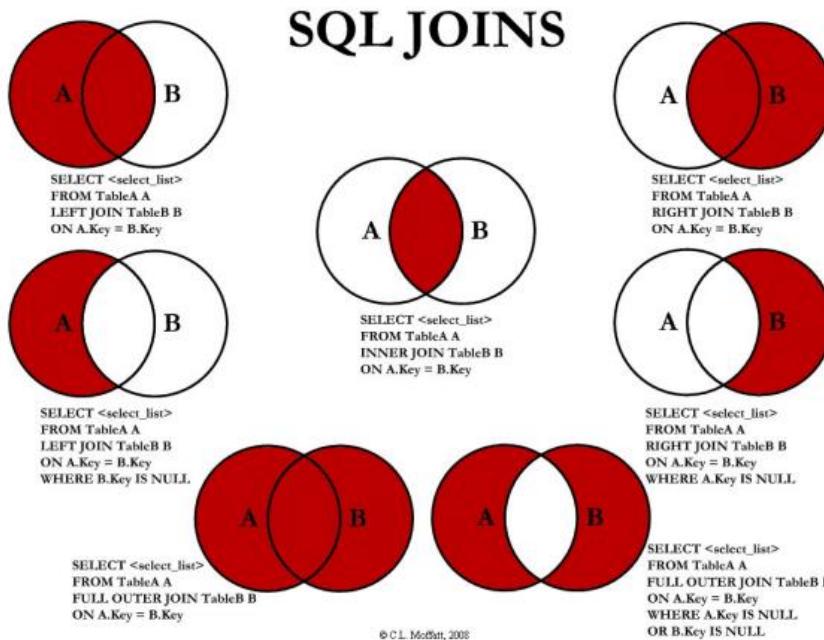
Persons	
ID	Name
1	Per
2	Mari

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;
```

## 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