INF115 Lecture 6: *Advanced Queries*

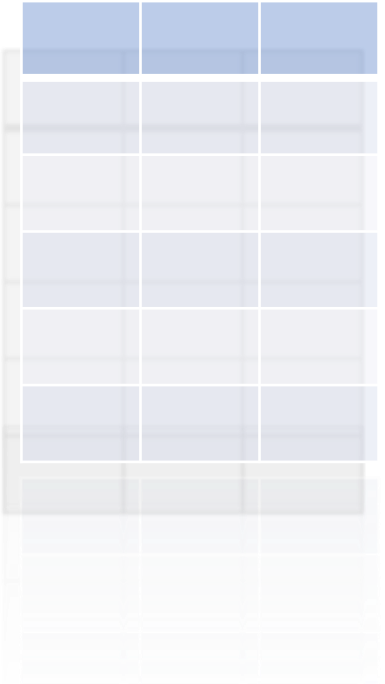
Adriaan Ludl
Department of Informatics
University of Bergen

Spring Semester 2021

Chapter 5: *Advanced Queries*

Learning Goals:

- **Use SQL to *solve complex problems***
- **SQL standard**
- Use **conditional** clauses and **subqueries**
- Set up and use **views**
- Advanced **aggregation** techniques
- *Understand which problems SQL can solve*



The SQL-standard

The SQL-standard has three parts:

- **Data Definition Language** (DDL): to define tables, indices and validation rules.
- **Data Manipulation Language** (DML): insert new data, update data, delete and query data.
- **Data Control Language** (DCL): user administration.

Versions: SQL:86, –:89, –:92, –:1999, –:2003, –:2008, –:2011, –:2016, –:2019, …

- *Not every problem can be solved with SQL.*
- Database applications are made using SQL and code written in a general programming language (C, Java, python ...)

Conditional clauses

Code:

```
SELECT VNr, Betegnelse, Pris,
  CASE
    WHEN Pris<100 THEN 'Billig'
    WHEN Pris<=500 THEN 'Middels'
    ELSE 'Dyr'
  END AS Prisklasse
FROM Vare
```

Output:

VNr	Betegnelse	Pris	Prisklasse
22054	Vannkanne, 5 ltr.	70.50	Billig
22179	Hafa gresklipper G9, elektrisk	1440.00	Dyr
25079	Trillebår	334.00	Middels
32067	Juwa Hagerive, 14 rette tinder	94.50	Billig

Three valued logic

- SQL is based on **three valued logic: TRUE, FALSE, NULL.**
  - The *special value* NULL is part of all datatypes and is not like ordinary values.
  - Null values can be propagated by expressions and logical operators.
- Truth tables with **NULL** for **AND** and **OR**:

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Views

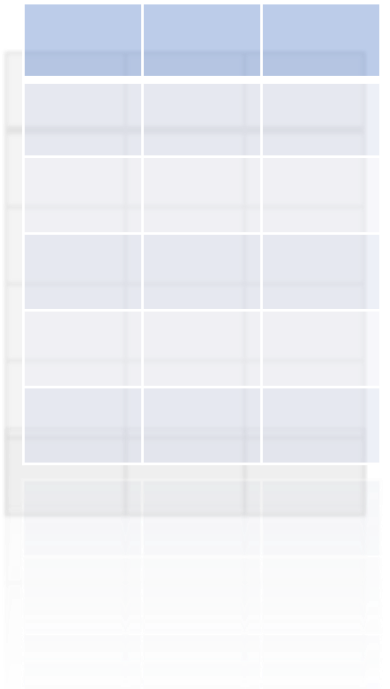
A view is a «virtual» table.

Motivation:

- *Store queries* in the database,
- ***Break up complex queries,***
- Simplify or adapt the database to different users (groups)
- **Independence of representations (see ch. 6)**

Implementation of views:

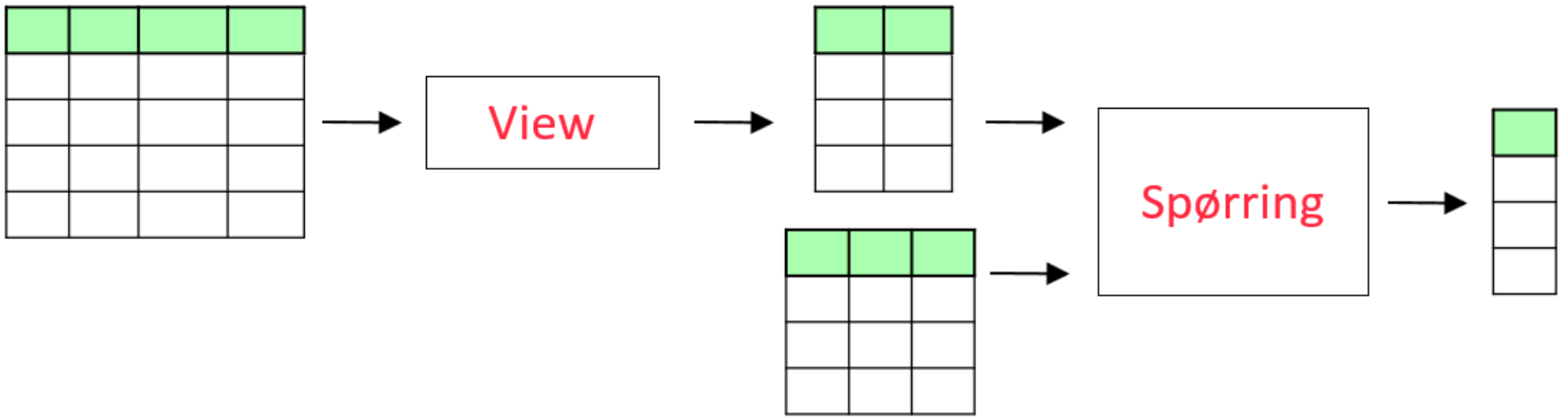
- Access has to be set up with ANSI SQL syntax.



Views and queries on queries

- Some tasks are difficult to solve with one query.
  - Break it up into several simpler problems.

  - Queries can be «stored» as views.
  - You can make queries on such views.



A complex problem

- Find the number of unique job descriptions.

1. Make a ***view*** that finds job descriptions:

```
CREATE VIEW StillingerIBruk AS
  SELECT DISTINCT Stilling
  FROM Ansatt
```

Ansatt(AnsattNr, Etternavn, Fornavn, AnsattDato, Stilling, Lønn)

2. Write a ***query*** using the ***view***:

```
SELECT COUNT(*) AS Antall
FROM StillingerIBruk
```

This can also be done directly with COUNT(DISTINCT Stilling)

Renaming columns

We can give columns new names:

```
CREATE VIEW Keramikk( Kode, Navn ) AS
   SELECT VNr, Betegnelse
   FROM Vare
   WHERE KatNr = 3
```

Query on the view:

```
SELECT *
FROM Keramikk
ORDER BY Navn
```

System tables and sorting

ViewNavn	ViewDefinisjon
Keramikk	SELECT …
StillingerIBruk	SELECT …
…	…

Views are «virtual» tables:

- A view is represented by its **defining query**.
- The DBMS stores definitions of views in a system table.

❖ *Views can therefore be seen as tables.*

- And tables are **not sorted**.

➢ **ORDER BY is normally not used to define views.**

- Rather use ORDER BY in queries on views.
- Although it is possible in some DBMS.

Views on multiple tables

- Views as a basis for sales reports:

```
CREATE VIEW Salg AS
  SELECT OL.*, V.Betegnelse, K.Navn,
         O.Ordredato, O.AnsNr, O.KNr
  FROM Ordre AS O, Ordrelinje AS OL,
       Vare AS V, Kategori AS K
  WHERE OL.OrdreNr = O.OrdreNr
    AND OL.VNr = V.VNr
    AND V.KatNr = K.KatNr
```

- The user of the view does not need to join tables !

- We present a «simplified database» to the user.

Queries on views

- Sales of the previous month by product:

```
SELECT VNr, SUM(Antall*PrisPrEnhet) AS Totalt
FROM Salg
WHERE YEAR( Ordredato )= Year( CURDATE() )
  AND MONTH( Ordredato ) = Month( CURDATE() )
GROUP BY VNr
```

- As if the query was executed so that the output of **Salg** is the input to the query.
- Or by **substituting** the definition of the view **Salg** in the query.

If we query the tables directly:

```
SELECT OL.VNr,
   SUM(OL.Antall*OL.PrisPrEnhet) AS Totalt
FROM Ordre AS O, Ordrelinje AS OL,
      Vare AS V, Kategori AS K
WHERE OL.OrdreNr = O.OrdreNr
   AND OL.VNr = V.VNr
   AND V.KatNr = K.KatNr
   AND YEAR(Ordredato) = YEAR(CURDATE())
   AND MONTH(Ordredato) = MONTH(CURDATE())
GROUP BY OL.VNr
```

Updates and view conditions

❖ We can **prevent updates** that break **with view conditions**:

```
CREATE VIEW DyreVarer AS
   SELECT *
   FROM Vare
   WHERE Pris > 1000
   WITH CHECK OPTION
```

➢ The following will now fail:

```
UPDATE DyreVarer
SET Pris = 999
WHERE Pris BETWEEN 1000 and 1050
```

Updatability

KatNr	Antall
1	28
2	2
3	22
4	18

The following view cannot be updated:

```
CREATE VIEW AntallVarerPrKategori AS
SELECT KatNr, COUNT(*) AS Antallvarer
FROM Vare
GROUP BY KatNr
```

- What happens if we change the number of products in the category dairy products (KatNr = 3, meierivarer) ?
- What is the effect on the underlying table ?
- SQL has **rules** defining which views are **updatable**:
  - Cannot use grouping and set functions (aggregating)
  - Primary keys must be selected.

Quizz on Multiple Tables (part 1)

Please answer the practice quizz on mitt.uib now ☺

(you can take it again later if you want)

Link:

- https://mitt.uib.no/courses/27455/quizzes

Break:
Lecture resumes in 15 minutes !

Subqueries – Motivation

- Find all who earn more than the average.
- *So far* the DBMS only had to go through the table **once** to answer our queries**.**
- Now we must:
  - first compute the average salary and
  - thereafter find those with salary above average.

ansnr	etternavn	stilling	lønn
1	Hansen	Selger	500.000
2	Mo	Programmerer	**600.000**
3	Jensen	Selger	500.000
4	Karlsen	Sekretær	400.000
5	Bø	Direktør	**800.000**

snitt
560.000

Subqueries in SQL

➢ *Find all who earn more than the average – **with SQL :***

```
SELECT *
FROM Ansatt
WHERE Lønn>(SELECT AVG(Lønn) FROM Ansatt)
```

- Subquery must return a table with one row and one column (to be used on the right sie of « > »).

➢ The result of the subquery is substituted into the main query:

- If the average is 560.000 ,
- We get: Lønn>560.000 .

Executing subqueries

- *Find products that are cheaper than average.*
- The average is computed first (subquery) and the result is substituted into the main query.

VNr		Pris	
90693		57.00	
44939		115.00	
...		...	

delspørring

AVG(Pris)
403.00

SELECT VNr, Pris FROM Vare WHERE Pris < (**SELECT AVG(Pris) FROM Vare**)

Interacting subqueries

- Find all who earn more than the **average of their job category:**

```
SELECT A1.*
FROM Ansatt AS A1
WHERE Lønn >
      ( SELECT AVG(Lønn)
        FROM Ansatt AS A2
        WHERE A1.Stilling = A2.Stilling )
```

- Compare this to the query: «Find all who earn more than the average».

- How many times does the DBMS have to go through the table this time ?

Exercise: Interacting subqueries (5 minutes)

- Find the cheapest products in each category: An interacting subquery on the *Product* table (Vare).
- Note that the main query and the subquery each work on «their own» copy of the table.

Vare **AS** V1

VNr	Betegnelse	Pris	KatNr
21580	Sponflis, natur	13.50	17
33044	Blandet blomsterfrø	14.50	15
35911	Meksikansk solsikke	11.50	15
35912	Stemorsblomst	11.50	15
46741	Lyskrukke, 4cm	10.00	17

Vare **AS** V2

VNr	Betegnelse	Pris	KatNr
21580	Sponflis, natur	13.50	17
33044	Blandet blomsterfrø	14.50	15
35911	Meksikansk solsikke	11.50	15
35912	Stemorsblomst	11.50	15
46741	Lyskrukke, 4cm	10.00	17

Solution

Find the cheapest products in each category:

```
SELECT V1.VNr, V1.Pris
FROM Vare AS V1
WHERE V1.Pris =
      ( SELECT Min(V2.Pris)
            FROM Vare AS V2
            WHERE V1.KatNr = V2.KatNr )
```

Subqueries in FROM-sections

➢ Find the number of unique job descriptions:

```
SELECT COUNT(*) AS AntallStillinger
FROM
   (
      SELECT DISTINCT Stilling
      FROM Ansatt
   ) AS StillingerIBruk
```

- As if we had a table StillingerIBruk.
- Such table expressions can be joined again with other tables.

❖ **Subqueries can also be used in WHERE, UPDATE, INSERT, DELETE clauses.**

IN

- **IN** denotes whether a value is in a set.
- IN corresponds to ∈ in set-theory.
- We want to show the clients with one or several orders:

```
SELECT *
FROM Kunde
WHERE KNr IN (SELECT KNr FROM Ordre)
```

- If we add NOT in front of the condition, we get the number of clients that do not have any orders.
- We can also use IN to simplify OR-conditions:

```
SELECT *
FROM Vare
WHERE KatNr IN (2, 4, 7)
```

Quantifiers ALL and SOME

- The quantifiers **ALL** and **SOME** can be placed before a subquery.
- Find those who earn more than all secretaries:

```
SELECT *
FROM Ansatt
WHERE Lønn > ALL
     (SELECT Lønn
      FROM Ansatt
      WHERE Stilling='Sekretær')
```

- Can we do this without ALL ? (Try using MAX).
  - This subquery must return one column.
- What happens if we replace **ALL** with **SOME**?

Quantifier EXISTS

- **EXISTS** checks whether the subquery gives a non-empty result.
- Find employees who have not participated in any projects:

```
SELECT A.*
FROM Ansatt AS A
WHERE NOT EXISTS
    (SELECT *
     FROM ProsjektDeltakelse AS PD
     WHERE PD.AnsNr = A.AnsNr)
```

- **EXISTS** can sometimes be replaced by **IN**.
  - Let the subquery return a list with employee numbers (AnsNr).
- Find employees who have participated in all projects: for an employee X there should not be any projects that X has not participated in. (A solution with SQL is given on the next slide.)

Quantifier EXISTS (part 2)

- Find employees who have participated in <u>all</u> projects: *for an employee X there should not be any projects that X has not participated in.*

```
SELECT A.*
FROM Ansatt AS A
WHERE NOT EXISTS
    (SELECT *
     FROM Prosjekt as P
     WHERE NOT EXISTS
         (SELECT *
          FROM ProsjektDeltakelse AS PD
          WHERE PD.AnsNr = A.AnsNr
          AND P.PNr = PD.PNr) )
```

List projects where the employee X has not participated:

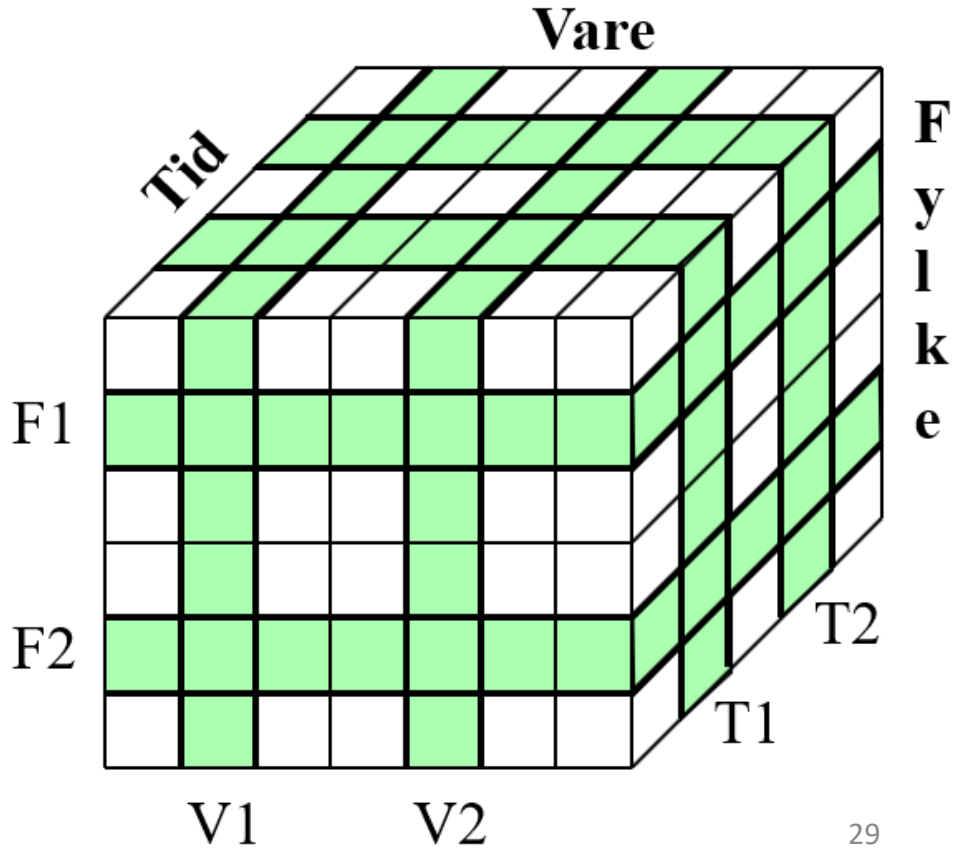
List participation of employee X in a given project.

- **Note**: this is *not easily readable* and it would be a case where it can be **easier to do this with a general programming language**.

Revisiting Aggregation

- **OLTP** (On-Line Transactional Processing) for daily operations.
- **OLAP** (On-Line Analytical Processing)

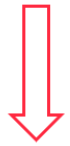
  for strategic decisions build on aggregated data – frequently stored in *data warehouses*.
- Aggregated data can be presented as **data cubes (GROUP BY CUBE).**



Window functions

- Window functions can be used to show single rows together with aggregated data.
- Such a windows can show several rows, with one row in focus.
- The window can be moved across the table, one row at a time.

VNr	Betegnelse	Pris
25136	Juwa Anleggspade	180.00
32069	Stikkspade	154.00
32067	Juwa Hagerive, 14 rette tinder	94.50
32055	Juwa Barkespade	130.50
25154	Ljå	276.00
25138	Grensaks med sideskjær	166.50
25137	Juwa Snøskuffe, standard	228.00



Accumulation

Cumulative statistics, e.g. accumulated daily precipitation (in PostgreSQL):

```
SELECT Dato, Nedbør, SUM(Nedbør) OVER
(ORDER BY Dato ROWS BETWEEN UNBOUNDED PRECEDING
AND CURRENT ROW) AS Akkumulert
FROM DagligNedbør
```

DagligNedbør(Dato, Nedbør)

Dato	Nedbør	Akkumulert
02.12.2019	7	7
03.12.2019	12	19
04.12.2019	0	19
05.12.2019	5	24
06.12.2019	22	46

Moving Average

- The window function can be used to «**smoothen**» out random variations in a data set.
  - Various data such as coronavirus infection rates or rainfall can vary substantially from day to day.
  - However, there are often longer term trends in the data.

- For example, compute the average rainfall on a given day as the average over the three preceeding and three following days:

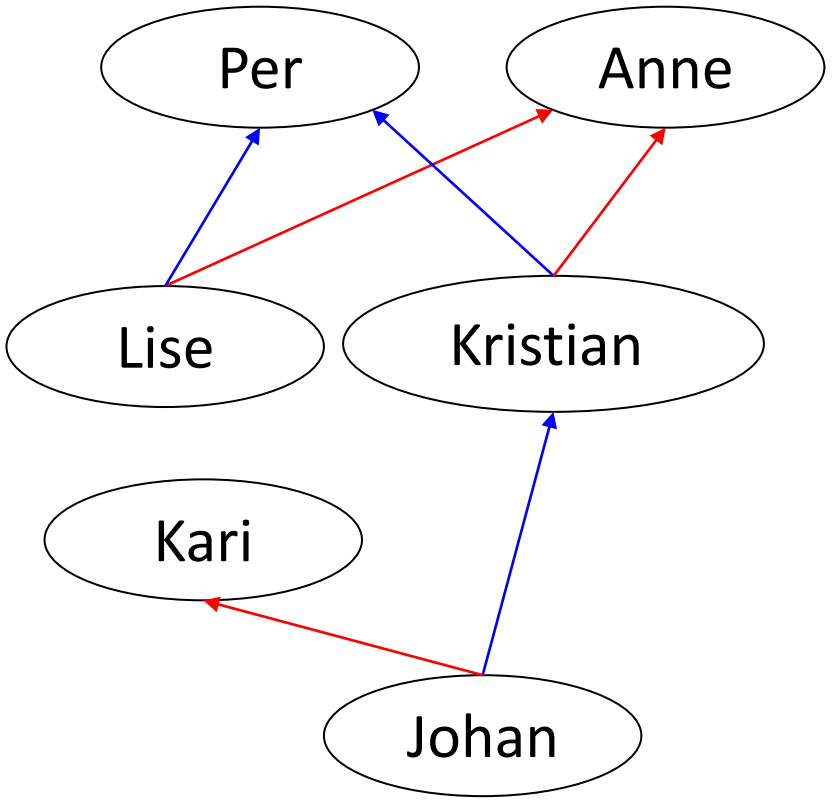
```
SELECT Dato, Nedbør, AVG(Nedbør) OVER
(ORDER BY Dato ROWS BETWEEN 3 PRECEDING
AND 3 FOLLOWING) AS UkeSnitt
FROM DagligNedbør
```

A hierarchical structure: A genealogical tree

The columns mother and father contain ID-numbers.

- These *foreign keys* refer to **PNr** (in the same table).

PNr	Fornavn	Mor	Far
1	Per		
2	Anne		
3	Lise	2	1
4	Kristian	2	1
5	Kari		
6	Johan	5	4



Limitations of SQL

- Using the **genealogy** (slekt) table on the previous slide:
  - Write a query that finds mother and father for a given person.
  - Same with grand parents (bestemødre, bestefedre)
  - **Get all ancestors of a given person !**
  - How many times do we have to go through the table to carry out these queries ? (assume that the ID-number is not ordered in a systematic way)
  - It is difficult (or nearly impossible) to solve this general problem with SQL (depending on whether the DBMS allows **recursion**).

❖ We must **understand the limitations of SQL** to *avoid trying to solve impossible problems*. In these cases it is better to combine SQL with general programming languages (C, Java, Python, …).

Quizz on Multiple Tables (part 2)

Please answer the practice quizz on mitt.uib now ☺

(you can take it again later if you want)

Link:

- https://mitt.uib.no/courses/27455/quizzes

Summary: *Advanced Queries*

- Use SQL to *solve complex problems*
- Use **subqueries** and **conditional** statements (CASE …)
- Set up and use **views** (`CREATE VIEW DyreVarer AS ...`)
- Advanced **aggregation** techniques (GROUP BY CUBE, windows, moving average …)
- *Understand which problems SQL can solve.*
- **SQL standard**
- Examples of problems that are **difficult to solve with SQL**
- Keywords: AS, IN, EXISTS, ALL, SOME