

SQL VIEW + Normalisering

Modul 9 (uke 7) innhold

VIEW

Example, the world schema

- A **VIEW** is a pre-created query for one or more tables.

```
CREATE OR REPLACE VIEW EuropeCountry_view AS  
SELECT Code, Name, Population  
FROM country  
WHERE Continent = 'Europe';
```

```
SELECT *  
FROM EuropeCountry_view;
```



Code	Name	Population
---	-----	-----
ALB	Albania	3401200
AND	Andorra	78000
AUT	Austria	8091800
BEL	Belgium	10239000
BGR	Bulgaria	8190900
BIH	Bosnia and Herzegovina	3972000
BLR	Belarus	10236000
CHE	Switzerland	7160400
CZE	Czech Republic	10278100
DEU	Germany	82164700
DNK	Denmark	5330000
ESP	Spain	39441700
...		

- Views are treated the same way as tables are.

Why use views?

- Security: Restrict data access in the database.
- Reduce complexity when writing SQL queries later:
 - Can make complex queries easier.
 - Reduces the amount of data for the user.
- Customization: Achieve different views of the database.
 - user groups / applications

Disadvantages of views

- More complexity related to updates and changes:
 - There are restrictions on when you can change underlying data through a view.
 - The structure is determined when the VIEW is created, and will not automatically change later (VIEW based on `SELECT * FROM ...`)
- Performance:
 - A view can join many tables, and thus be a relatively heavy query.
 - This is not always easy to see for the user.

Create a view

```
CREATE [OR REPLACE] VIEW view_name AS  
subquery;  -- SELECT ... FROM ...
```

- OR REPLACE → overwrites a VIEW if it already exists
- subquery → the SELECT query we want to store

Example, world DB: (same as 1st slide)

```
CREATE OR REPLACE VIEW EuropeCountry_view AS  
SELECT Code, Name, Population  
FROM country  
WHERE Continent = 'Europe';
```

```
SELECT *  
FROM EuropeCountry_view;
```



Code	Name	Population
ALB	Albania	3401200
AND	Andorra	78000
AUT	Austria	8091800
BEL	Belgium	10239000
BGR	Bulgaria	8190900
BIH	Bosnia and Herzegovina	3972000
BLR	Belarus	10236000
CHE	Switzerland	7160400
CZE	Czech Republic	10278100
DEU	Germany	82164700
DNK	Denmark	5330000
ESP	Spain	39441700
...		

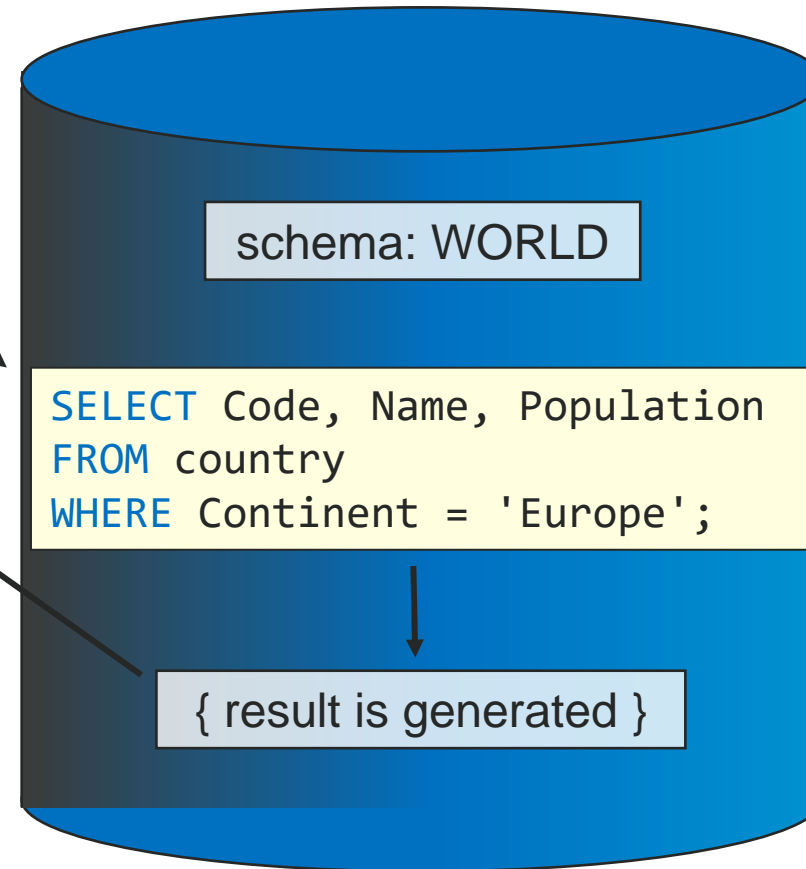
- Views are treated the same way as tables are.

This is how querying a view works

```
SELECT *  
FROM EuropeCountry_view;
```

Code	Name	Population
---	-----	-----
ALB	Albania	3401200
AND	Andorra	78000
AUT	Austria	8091800
BEL	Belgium	10239000
BGR	Bulgaria	8190900
...		

- The definition of **the VIEW itself** (*not* the result) **is stored** in the database.



Updates through views

- Views can be used to update data in an underlying table.
- *Note:* There are ISO **restrictions** that must be followed before a **view** can be **used for updates**:
 - The view can only reference **one table**.
 - **DISTINCT cannot** be part of the view.
 - **All elements** in the select part of the view **must be columns** (not constants, summations, etc.)
 - **No GROUP BY** or **HAVING**.
 - Rows that are added **must follow the integrity rules** for the underlying table (not null, etc.).

Updates through views – cont.

- Note that a view updates the data in the table! (Not just the view itself.)


```
UPDATE EuropeCountry_view  
SET ID = 'A_Z'  
WHERE Country = 'Austria';
```

```
SELECT ID, Country  
FROM EuropeCountry_view;
```



ID	Country
----	-----
ALB	Albania
AND	Andorra
A_Z	Austria
BEL	Belgium
BGR	Bulgaria
...	

```
SELECT Code, Name, Population  
FROM country  
WHERE Name = 'Austria';
```



CODE	Name	Population
----	-----	-----
A_Z	Austria	8091800

Deleting views

- Syntax to delete a view is almost the same as for a table:
 - Just replace the word "table" with "view".

```
DROP VIEW EuropeCountry_view;
```

Normalization

The purpose of normalization

- The purpose of **normalization** is to find the most favourable relations ("table setup") for a given database.
- **The main criterion** we strive for in normalization:
 - **Minimal double storage (redundancy)**, so attributes are only stored in one place.
 - Exception to this is Foreign Keys: These columns necessarily need to be stored in two places, since they are to connect tables (entities/relations) through PK and FK.
- *Normalization usually means splitting a database into **more tables**.*

Problems with double storage

- The table employee_branch (with primary key enr) contains **double storage** (redundancy) in **several columns**.
 - BNR, BADDRESS and BZIP: branch number, branch address and branch postal code.

ENR	NAME	EADDRESS	EZIP	POSITION	SALARY	BNR	BADDRESS	BZIP
3	Jon Hvit	Bruveien 7	4000	Manager	30000	1	Utleieveien 15	4000
4	Anne Strand	Strandgaten	2000	Broker	12000	1	Utleieveien 15	4000
20	Olav Gautesen	Galmannsveien 4	3000	Broker	26000	1	Utleieveien 15	4000
5	David Opalsen	Gulerleveien 43	2000	Secretary	18000	1	Utleieveien 15	4000
2	Marie Hovland	Strilegaten 8	5000	Manager	13000	2	Smuglerstien 67	5000
23	Ole Ås	Mor Åseveien 56	4000	Broker	17000	2	Smuglerstien 67	5000
21	Per Pollesen	Podlestadveien 5	5000	Secretary	15000	2	Smuglerstien 67	5000
7	Karl Hansen	Olavsgt 7	2000	Manager	25000	3	Snusveien 7	7000

- *What problems does this table give us, in terms of inserting, updating and deleting data?*

INSERT problems

- Inserting a new employee for an existing branch, problems:
 - Must at the same time re-enter data for a branch.
(Branch number, address and postcode).
 - By typing these fields incorrectly, you get problems with the address being different for the same branch: you get an inconsistent database.

ENR	NAME	EADDRESS	EZIP	POSITION	SALARY	BNR	BADDRESS	BZIP
3	Jon Hvit	Bruveien 7	4000	Manager	30000	1	Utleieveien 15	4000
4	Anne Strand	Strandgaten	2000	Broker	12000	1	Utleieveien 15	4000

- Inserting a new branch without employees, problems:
 - Must enter NULL in the fields that apply to employees. But ENR is the PK and NULL is not allowed there.
 - Thus, must add a dummy employee.

DELETE and UPDATE problems

- Deleting last employee for a branch, problems:
 - At the same time loses all information about the branch.

ENR	NAME	EADDRESS	EZIP	POSITION	SALARY	BNR	BADDRESS	BZIP
3	Jon Hvit	Bruveien 7	4000	Manager	30000	1	Utleieveien 15	4000

- Updating zip or address of a branch, problems:
 - Must make the same changes in all rows to all employees for this branch.
 - By typing incorrectly, you get problems with the address being different on the same branch: you get an inconsistent database.

ENR	NAME	EADDRESS	EZIP	POSITION	SALARY	BNR	BADDRESS	BZIP
3	Jon Hvit	Bruveien 7	4000	Manager	30000	1	Utleieveien 15	4000
4	Anne Strand	Strandgaten	2000	Broker	12000	1	Utleieveien 15	4000

Better table structure

- *Solution:* Split employee_branch into two tables – employee and branch.

ENR	NAME	EADDRESS	EZIP	POSITION	SALARY	BNR
3	Jon Hvit	Bruveien 7	4000	Manager	30000	1
4	Anne Strand	Strandgaten	2000	Broker	12000	1
20	Olav Gautesen	Galmannsveien 4	3000	Broker	26000	1
5	David Opalsen	Gulerleveien 43	2000	Secretary	18000	1
2	Marie Hovland	Strilegaten 8	5000	Manager	13000	2
23	Ole Ås	Mor Åseveien 56	4000	Broker	17000	2
21	Per Pollesen	Podlestadveien 5	5000	Secretary	15000	2
7	Karl Hansen	Olavsgt 7	2000	Manager	25000	3

BNR	BADDRESS	BZIP
1	Utleieveien 15	4000
2	Smuglerstien 67	5000
3	Snusveien 7	7000

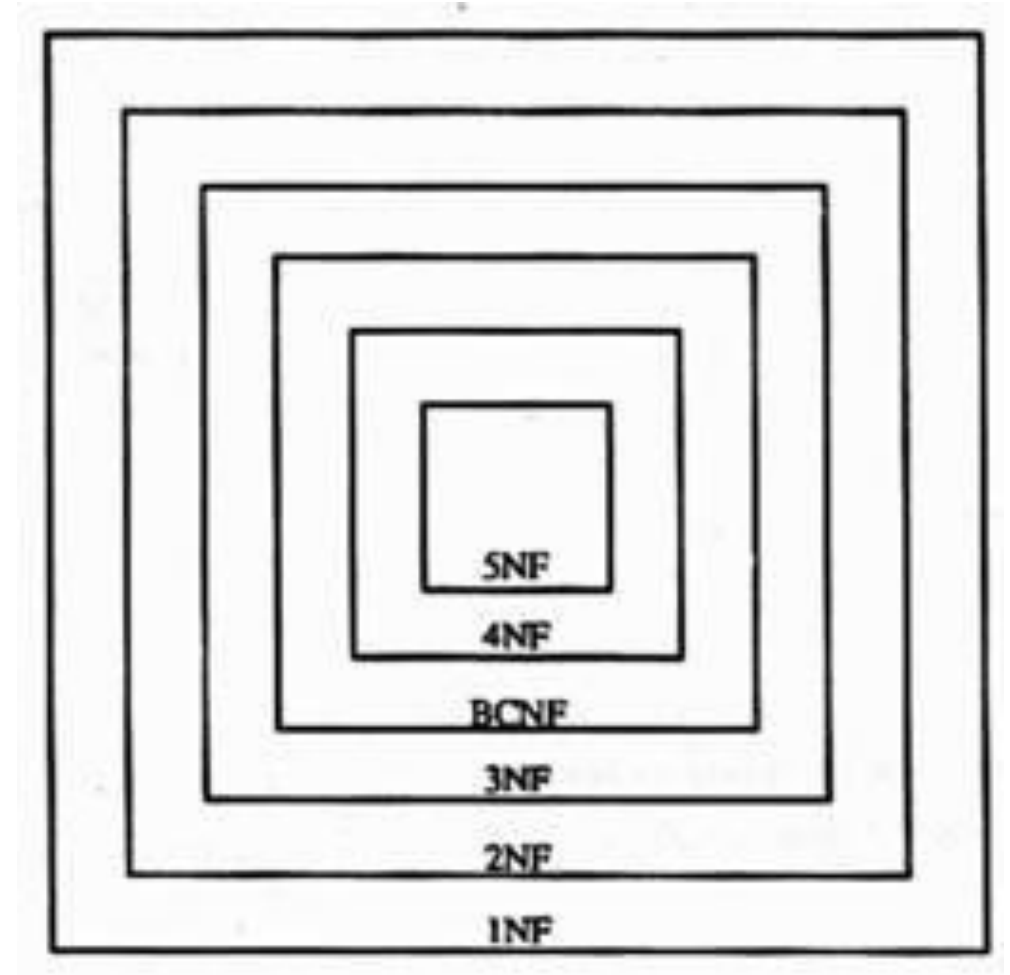
- Are we now having problems with insert / update / delete?
 - Nope! :-) (More details on coming slide.)

Better table structure – cont.

- Inserting a new employee
 - We only enter personal information, plus the correct branch number (no other branch data).
 - The branch address cannot be different for one and the same branch, as the branch information is only stored in one row in the database. (Has a consistent database).
- Inserting a new branch
 - It is perfectly okay that there is no information about employees in the branch table.
- Change of postcode or address of a branch
 - Only necessary to change the information for one row in the database. (Has a consistent database.)
- Delete last employee for a branch
 - The branch still exists in the branch table.

The normalization stages

- Normalization are done in stages (levels), based on certain rules.
- Each increasing stage builds on the previous stage.
 - (BCNF can in this regard be seen as "3.5NF".)
- NOTE: Databases in *the real world* are usually normalized to 3NF or BCNF.
 - *NOT* all the way to 5NF.



Denormalization

- We can also go the opposite way:
 - Denormalization
- Denormalization means:
 - Altering tables so that normalization is reduced by one degree or more.
 - *Example:* Going from BCNF or 3NF to 2NF.
- The reason for denormalization is usually that our JOINS take time:
 - We can make our SELECTs faster if they do not contain JOINS.

Denormalization – cont.

- **Benefits of denormalization:**
 - You avoid linking tables (fewer joins), making queries easier to write(?).
 - Less linking of tables can increase the query speed when looking up large amounts of data.
- **Disadvantages:**
 - Implementation of the structure might become more difficult.
 - Double storage of data (redundancy).
 - Slower storage / updating.
 - Flexibility decreases.
- *NOTE:* Denormalization is the exception! **We usually want a normalized DB.** 😊

