SQL I
Select Statements and Data Types
Database Management and Data Visualization

Benjamin D. Liengaard
Department of Economics and Business Economics

Overview

Introduction

- Learn by doing!
- If time permits, there will be short "breaks" in the lectures where you can actively work with the SQL syntax
- Do not be afraid to make mistakes - experiment and "play" with your SQL syntax
- Don't stress if you don't understand it immediately - your skill is a function of the time you spend working with the subject

Tables

- Collection of related data
- Organized by rows and columns
- In a relational database these tables are related to one another
- Restrictions can be imposed on the tables (e.g. a column cannot have duplicated values or NULL values)
- Tables have different types of $keys$ - we will return to this later
- When the tables are depicted in the entity relation (ER) diagram (and in the book), only the columns are depicted

Schemas

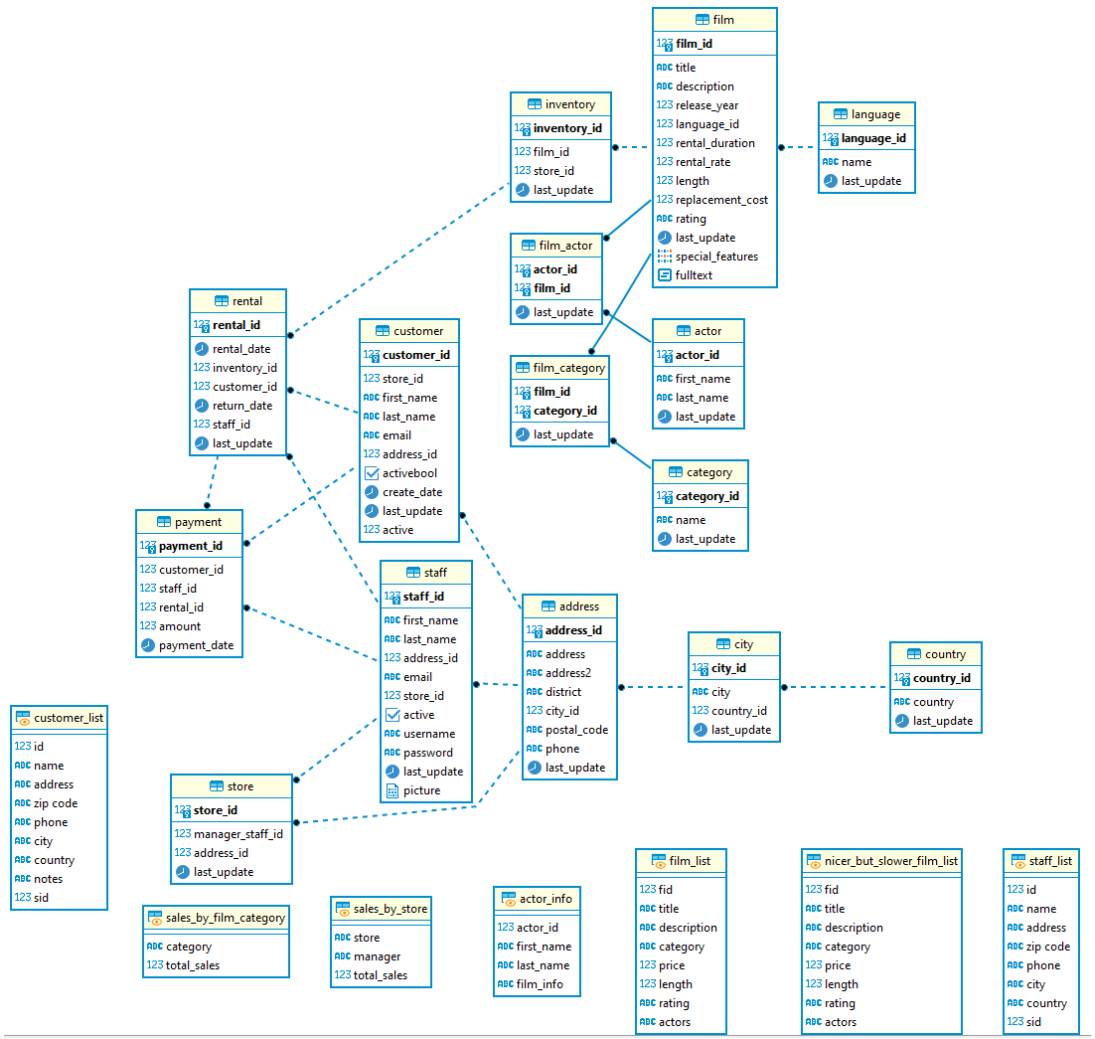
- Collection of tables
- Can also contain other database objects such as functions
- Why use schemas?
  - Organizing the components of the database
  - E.g. one schema for each business unit
  - We can grant users restricted access to specific tables or schemas
- Let us look at the structure of a database in DBeaver

Select statement

- The *select* statements allows you to
  - Return records from your database
  - It is one of the most common types of commands
- We will go through common types of select statements
  - I will illustrate with a small "toy-table" and the *dvdrental* database

Select statement

- The *dvd_rental* data (the business event of customers renting DVD's)



Manually creating a table

Create an empty table In your public schema

```sql
-- Create table in public schema
create table public.example_table (
    id serial primary key,
    first_name varchar(255),
    last_name varchar(255),
    height numeric(10),
    weight numeric(10),
    gender boolean,
    date_birth date
);
```

Insert values into the table

```sql
-- Insert values into the public schema (rowwise insertion)
insert into public.example_table(first_name, last_name, height, weight, gender, date_birth)
    values ('Anders', 'Madsen', 187, 90, '0', '1984-02-15')
           ,('Birgit', 'Hansen', 187, 78, '1', '1992-02-16')
           ,('Birgit', 'Kristiansen', 169, 50, '1', '1979-02-15')
           ,('Anders', 'Madsen', 175, 72, '0', '1995-08-25')
           ,('Anders', 'Madsen', 170, 55, '0', '2001-06-21')
           ,('Anne', 'Madsen', 170, 52, '1', '1997-06-21')
           ,('Pia', 'Henriksen', 168, 75, '1', '1989-06-21')
           ,('Pia', 'Henriksen', null, null, '1', '2021-06-21');
```

Structure of table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

Select the entire table

SQL query

```
select *
from public.example_table;
```

Result

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

Select specific columns

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select first_name, last_name
from public.example_table;
```

Result

	first_name	last_name
1	Anders	Madsen
2	Birgit	Hansen
3	Birgit	Kristiansen
4	Anders	Madsen
5	Anders	Madsen
6	Anne	Madsen
7	Pia	Henriksen
8	Pia	Henriksen

Select the top rows

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select *
from public.example_table
limit 3;
```

Result

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15

Distinct values

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select distinct first_name
from public.example_table;
```

Result

	first_name
1	Anne
2	Anders
3	Pia
4	Birgit

```
select distinct first_name, last_name
from public.example_table;
```

	first_name	last_name
1	Birgit	Hansen
2	Anne	Madsen
3	Birgit	Kristiansen
4	Pia	Henriksen
5	Anders	Madsen

Where values satisfy condition

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select first_name, last_name, weight
from public.example_table
where first_name = 'Anders';
```

Result

	first_name	last_name	weight
1	Anders	Madsen	90
2	Anders	Madsen	72
3	Anders	Madsen	55

```
select first_name, height, weight
from public.example_table
where height > 169;
```

	first_name	height	weight
1	Anders	187	90
2	Birgit	187	78
3	Anders	175	72
4	Anders	170	55
5	Anne	170	52

Documentation: Comparison operators

Comparison and Logical operators: and, or

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```sql
select first_name, height, weight
from public.example_table
where (height > 170) and (first_name = 'Anders');
```

Result

	first_name	height	weight
1	Anders	187	90
2	Anders	175	72

```sql
select first_name, height, weight
from public.example_table
where (height > 170) or (first_name = 'Anders');
```

	first_name	height	weight
1	Anders	187	90
2	Birgit	187	78
3	Anders	175	72
4	Anders	170	55

Documentation: Comparison operators, Logical operators

Comparison and Logical operators: between, not

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select first_name, height, date_birth
from public.example_table
where date_birth between '1989-06-21' and '2001-06-22';
```

Result

	first_name	height	date_birth
1	Birgit	187	1992-02-16
2	Anders	175	1995-08-25
3	Anders	170	2001-06-21
4	Anne	170	1997-06-21
5	Pia	168	1989-06-21

```
select first_name, height, date_birth
from public.example_table
where date_birth not between '1989-06-21' and '2001-06-22';
```

	first_name	height	date_birth
1	Anders	187	1984-02-15
2	Birgit	169	1979-02-15
3	Pia	[NULL]	2021-06-21

Documentation: Comparison operators

Order By values in column

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select first_name, date_birth
from public.example_table
order by date_birth;
```

Result

	first_name	date_birth
1	Birgit	1979-02-15
2	Anders	1984-02-15
3	Pia	1989-06-21
4	Birgit	1992-02-16
5	Anders	1995-08-25
6	Anne	1997-06-21
7	Anders	2001-06-21
8	Pia	2021-06-21

```
select first_name, date_birth
from public.example_table
order by date_birth desc;
```

	first_name	date_birth
1	Pia	2021-06-21
2	Anders	2001-06-21
3	Anne	1997-06-21
4	Anders	1995-08-25
5	Birgit	1992-02-16
6	Pia	1989-06-21
7	Anders	1984-02-15
8	Birgit	1979-02-15

Documentation: Order By

Order By values across two columns

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select first_name, height
from public.example_table
order by first_name, height;
```

Result

	first_name	height
1	Anders	170
2	Anders	175
3	Anders	187
4	Anne	170
5	Birgit	169
6	Birgit	187
7	Pia	168
8	Pia	[NULL]

```
select first_name, height
from public.example_table
order by height, first_name;
```

	first_name	height
1	Pia	168
2	Birgit	169
3	Anders	170
4	Anne	170
5	Anders	175
6	Anders	187
7	Birgit	187
8	Pia	[NULL]

Documentation: Order By

Group By and find Min/Max within groupings

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select first_name, max(height)
from public.example_table
group by first_name;
```

```
select first_name, min(height)
from public.example_table
group by first_name;
```

Result

	first_name	max
1	Anne	170
2	Anders	187
3	Pia	168
4	Birgit	187

	first_name	min
1	Anne	170
2	Anders	170
3	Pia	168
4	Birgit	169

Documentation: Group By, Aggregate functions

Group By and taking sum withing groupings

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select first_name, sum(height)
from public.example_table
group by first_name;
```

Result

	first_name	sum
1	Anne	170
2	Anders	532
3	Pia	168
4	Birgit	356

```
select first_name, last_name, sum(height)
from public.example_table
group by first_name, last_name;
```

	first_name	last_name	sum
1	Birgit	Hansen	187
2	Anne	Madsen	170
3	Birgit	Kristiansen	169
4	Pia	Henriksen	168
5	Anders	Madsen	532

Documentation: Group By, Aggregate functions

Group By and Count number of values within groupings

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select first_name, count(first_name)
from public.example_table
group by first_name;
```

```
select first_name, count(height)
from public.example_table
group by first_name;
```

Result

	first_name	count
1	Anne	1
2	Anders	3
3	Pia	2
4	Birgit	2

	first_name	count
1	Anne	1
2	Anders	3
3	Pia	1
4	Birgit	2

Documentation: Group by, Aggregate functions

Having with Group By

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select first_name, count(height)
from public.example_table
group by first_name
having count(height) > 1;
```

```
select first_name, sum(height)
from public.example_table
group by first_name
having sum(height) < 200;
```

Result

	first_name	count
1	Anders	3
2	Birgit	2

	first_name	sum
1	Anne	170
2	Pia	168

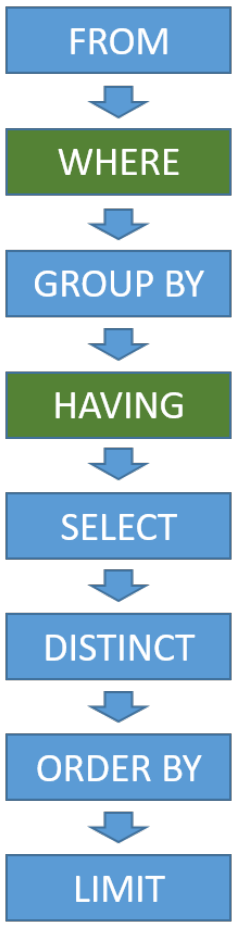
Documentation: Group by and Having,

Order of Evaluation

WHERE: To filter rows based on a specified condition before any aggregation or grouping takes place.

HAVING: filters groups of rows (or aggregated results) after the GROUP BY clause is proce

Order of Evaluation



Order of Syntax

Example: dvdrental, payment table

For each customer_id we want to see the sum of payments, but only considering payments above 1 dollar. In addition, we only want to see the customer_id with a total payment of more than 90 dollars, and we want the results to be sorted such that the lowest sum of payments is in the first row. Lastly, we only want to see the first three rows of this result set.

```
select sum(amount) sum_amount, customer_id
from public.payment
where amount > 1
group by customer_id
having sum(amount) > 90
order by sum_amount
limit 3;
```

sum_amount	customer_id
90.77	381
90.79	111
90.8	10

CASE WHEN, new values based on logical condition

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select first_name, gender,
    case gender
        when '0' then 'Male'
        when '1' then 'Female'
    end as gender_new
from public.example_table;
```

Result

	first_name	gender	gender_new
1	Anders	[ ]	Male
2	Birgit	[v]	Female
3	Birgit	[v]	Female
4	Anders	[ ]	Male
5	Anders	[ ]	Male
6	Anne	[v]	Female
7	Pia	[v]	Female
8	Pia	[v]	Female

```
select first_name, last_name, height,
    case
        when height > 175 then 'Higher'
        when height = 175 then 'Average'
        when height < 175 then 'Lower'
        else 'No data'
    end as in_relation_to_average
from public.example_table;
```

	first_name	last_name	height	in_relation_to_average
1	Anders	Madsen	187	Higher
2	Birgit	Hansen	187	Higher
3	Birgit	Kristiansen	169	Lower
4	Anders	Madsen	175	Average
5	Anders	Madsen	170	Lower
6	Anne	Madsen	170	Lower
7	Pia	Henriksen	168	Lower
8	Pia	Henriksen	[NULL]	No data

Documentation: CASE

Math across columns for each row

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```sql
select first_name, last_name, height,
height/100 as height_meters
from public.example_table;
```

Result

	first_name	last_name	height	height_meters
1	Anders	Madsen	187	1.87
2	Birgit	Hansen	187	1.87
3	Birgit	Kristiansen	169	1.69
4	Anders	Madsen	175	1.75
5	Anders	Madsen	170	1.7
6	Anne	Madsen	170	1.7
7	Pia	Henriksen	168	1.68
8	Pia	Henriksen	[NULL]	[NULL]

```sql
select first_name, last_name, height, weight,
(weight/(power(height/100, 2))) as bmi
from public.example_table;
```

	first_name	last_name	height	weight	bmi
1	Anders	Madsen	187	90	25.737081415
2	Birgit	Hansen	187	78	22.3054705596
3	Birgit	Kristiansen	169	50	17.5063898323
4	Anders	Madsen	175	72	23.5102040816
5	Anders	Madsen	170	55	19.0311418685
6	Anne	Madsen	170	52	17.9930795848
7	Pia	Henriksen	168	75	26.5731292517
8	Pia	Henriksen	[NULL]	[NULL]	[NULL]

Documentation: Mathematical function

Using aliases

public.example_table

	id	first_name	last_name	height	weight	gender	date_birth
1	1	Anders	Madsen	187	90	[ ]	1984-02-15
2	2	Birgit	Hansen	187	78	[v]	1992-02-16
3	3	Birgit	Kristiansen	169	50	[v]	1979-02-15
4	4	Anders	Madsen	175	72	[ ]	1995-08-25
5	5	Anders	Madsen	170	55	[ ]	2001-06-21
6	6	Anne	Madsen	170	52	[v]	1997-06-21
7	7	Pia	Henriksen	168	75	[v]	1989-06-21
8	8	Pia	Henriksen	[NULL]	[NULL]	[v]	2021-06-21

SQL query

```
select first_name as fi, last_name as la
 from public.example_table;
```

Result

	fi	la
1	Anders	Madsen
2	Birgit	Hansen
3	Birgit	Kristiansen
4	Anders	Madsen
5	Anders	Madsen
6	Anne	Madsen
7	Pia	Henriksen
8	Pia	Henriksen

```
select *,
    case
        when p.bmi > 25 then 'Above 25'
        when p.bmi <= 25 and p.bmi >=20 then 'Normal'
        when p.bmi < 20 then 'Under 20'
        else 'No data'
    end bmi_classification
from (select first_name, last_name, height, weight, (weight/(power(height/100, 2))) as bmi
     from public.example_table) as p
```

the select table will be performed on this table we just created

	first_name	last_name	height	weight	bmi	bmi_classification
1	Anders	Madsen	187	90	25.737081415	Above 25
2	Birgit	Hansen	187	78	22.3054705596	Normal
3	Birgit	Kristiansen	169	50	17.5063898323	Under 20
4	Anders	Madsen	175	72	23.5102040816	Normal
5	Anders	Madsen	170	55	19.0311418685	Under 20
6	Anne	Madsen	170	52	17.9930795848	Under 20
7	Pia	Henriksen	168	75	26.5731292517	Above 25
8	Pia	Henriksen	[NULL]	[NULL]	[NULL]	No data

Data types

- We can specify the data format of each of the columns in our data tables
- Tradeoff between flexibility, data storage requirements and resilience to erroneous data
- Best to specify data type at table creation
- We can change the datatype later, but this might have unintended consequences if we already have values in the column (values that does not adhere to the specified datatype)

Data types

For each data type there is a range of subcategories. Correctly specifying the data type will use storage more efficiently

- Text
- Numeric
- Temporal
- Boolean
- Other data types and more detailed descriptions can be found **here**

Text

This data type gives the greatest flexibility

- TEXT
  - Strings of variable and (almost) unlimited length
  - E.g. written customer feedback and news articles
- VARCHAR(N)
  - Variable character length
  - *N*: Max length of string
- CHAR(N)
  - *N*: Exact number of characters
  - If we insert less than *N* characters, white space will be padded at the end

Integers

- smallint
  - Integers in the interval $[-32768; +32767]$
- integer
  - Integers in the interval $[-2147483648; +2147483647]$
- bigint
  - Integers in the interval $[-9223372036854775808; -9223372036854775807]$
- serial
  - Autoincrementing integer (1 to 2147483647)

Decimals

- decimal(precision, scale)
  - Precision: Total number of digits before and after the decimal point (up to 131072 digits)
  - Scale: Number of digits to the right of the decimal point (up to 16383 digits)
  - e.g. decimal(6,2) allows for 1001.32 but not for 10011.32 or 100.325
  - Numeric is synonyms with decimal and they can be used interchangeably

Temporal and Boolean

- timestamp
  - E.g. *2020-09-21 16:31:13*
- date
  - E.g. 2020-09-21
- boolean
  - "true" state: true, yes, on, 1
  - "false" state: false, no, off, 0
  - "unknown" state: null

Data types - Example

```sql
-- Create a schema called BI
create schema BI;

-- Data types
create table BI.books (
    ID serial primary key,
    Title char(250),
    Description varchar(250),
    price decimal(5,2),
    register_time timestamp,
    publish_date date,
    hardback boolean,
    own_ranking integer,
    own_notes text);
-- Insert into table
insert into BI.books
    values (default, 'SQL cookbook', 'Interesting read', 100.32, '2015-09-10 00:00:00', '2014-02-01', '0', 3, 'Possibly long note 1' )
    ,(default, 'SQL for beginners', 'Nice introduction',  13.475, '2001-02-04 00:00:00', '1999-12-01', '1', 5, null)
    ,(default, 'Advanced SQL', 'Good for the experienced user', 15, '2019-11-01 00:00:00', '2018-11-15', null, 8, 'Possibly long note 3');
-- View table
select *
from BI.books
```

No need to input specific values for the 'ID' column - use 'default' instead. This allows auto incrementing integers to be generated with no manual bookkeeping.

Exercises I

For exercise 1-14 you need to connect to the dvdrental database. For exercise 15, you need to connect to one of your own databases.

1. Return the first 10 rows of the $Customer$ table (include all columns)
2. Return the distinct postal codes from the $address$ table
3. Return the title of the films (from the $film$ table) with a length above 90 minutes.
4. Return the title, length, and rental rate of all films which has a length below 90 minutes and a rental rate above 4 dollars.
5. Return the payment id, the payment date, and the amount (from the $payment$ table) where the payment date falls in the interval [19 February 2007 at 7 p.m.; 20 February 2007 at 7 p.m.].
6. You can use brackets, $()$, to nest logical operators. Make the same query as in exercise 5, but only return rows where the amount is above 7 dollars.
7. Make two queries that return the rental table. The first query returns the rows sorted with respect to the rental_date column such that the first rental date is in the first row. The second query is similar to the first but returns the opposite sorting of the rental_date column.

Exercises II

8 Explain in words what the following query does:

```
select customer_id, amount, payment_id
from payment
order by customer_id, amount;
```

9 Using the *payment* table: Write a query that gets the number of payments for each customer. The returned table should have the customer id and the number of payments as columns and only show the 10 rows which have the highest number of payments.

10 Using the *payment* table: Write a query that gets the number of payments for each customer. The query should only return customer_id where the number of payments is between 15 and 17. Use the $<=$ and $>=$ comparison operators in your query.

11 Compute the average value of the amount column in the payment table.

12 Assume the average amount in the payment table is approximately 4.2. Write a query that returns a table with a column that indicates whether a payment is either 'At the average', 'Above average', 'Below average', or if there is 'no data available'.

Exercises III

13 The query given below can be written shorter (and easier to read). Rewrite the query below into a new and shorter query that returns the same result.

```
select sum(a.after_vat)
from (select amount*0.8 as after_vat
      from payment) as a;
```

14 The following query counts the number of distinct postal codes from the *address* table

```
select count(distinct postal_code)
from address;
```

write an alternative (and longer) query that returns the same output, but uses aliases. This code should first get a table of distinct values of postal codes, and secondly, use that table to count the number of rows. Use the previous exercise as inspiration.

15 Create your own table in one of your personal databases. This table should have an autoincrementing id column, an integer column, a text column where each cell can have a maximum of 200 characters, a numeric column that allows a total 10 digits and 3 digits after the decimal point, a column that contains dates and times, and a boolean column. After the table is created, insert two rows of data. Run the insert statement two times and check that your id column increments.

Constraints and Relationships

Database Management and Data Visualization

Benjamin D. Liengaard
Department of Economics and Business Economics

Overview

Data integrity - purpose

- The SQL database should ensure data is stored correctly: *Data integrity*
- Failure of data integrity: Inconsistent data and questioning the validity of the data
- To ensure data integrity we can provide *integrity constraints*
- We will consider the following constraints: **check**, **primary key**, **not null**, **unique**, and **foreign key** as well as the **serial** property related to primary keys

Click **here** for PostgreSQL documentation on constraints.

Check

- *Check*: Values must satisfy a Boolean (true/false) constraint

Create an empty sale table In the *bi* schema

Name of constraint

Boolean (true/false) constraint

```
create table bi.sale (
    sale_id integer,
    sale_person_id integer,
    price numeric constraint positive_price check (price >= 0)
);
```

Insert values into the sale table

```
insert into bi.sale
    values(1, 1, 100);
```

Possible

```
insert into bi.sale
    values(2, 1, 0);
```

Possible

```
insert into bi.sale
    values(1, 1, -100);
```

ERROR

Structure of table

sale_id	sale_person_id	price
1	1	100
2	1	0

Primary Key

- *Primary key*: Must be unique and cannot contain *null* values

Create an empty Student table In the *bi* schema

```
create table bi.Student (
    student_id serial primary key,
    student_name varchar(255),
    department_code int);
```

Insert values into the student table

```
insert into bi.Student
    values (1, 'Birgit', 2)
          ,(2, NULL, 1)
          ,(3, 'Pia', 3)
          ,(4, 'Henrik', 3);
```

Possible

```
insert into bi.Student
    values (NULL, 'Birgit', 2)
          ,(1, 'Anders', 1)
          ,(2, 'Pia', 3)
          ,(3, 'Henrik', 3);
```

ERROR

```
insert into bi.Student
    values (1, 'Birgit', 2)
          ,(1, 'Anders', 1)
          ,(3, 'Pia', 3)
          ,(4, 'Henrik', 3);
```

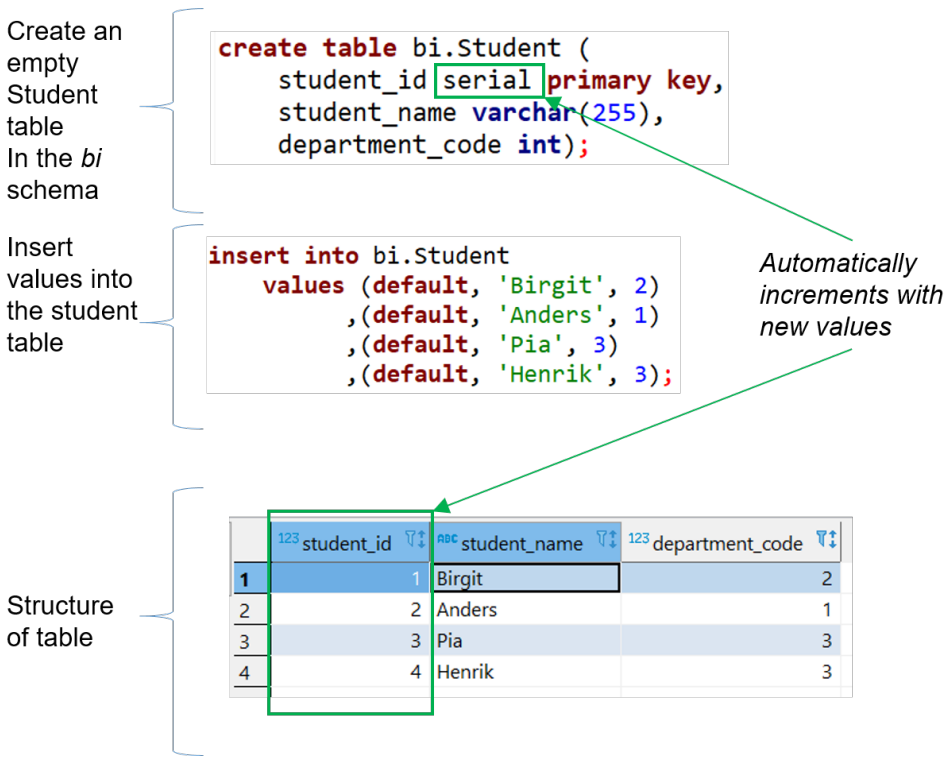
ERROR

Structure of table

	student_id	student_name	department_code
1	1	Birgit	2
2	2	[NULL]	1
3	3	Pia	3
4	4	Henrik	3

Serial

- *Serial*: When a new row is inserted, a new value is automatically inserted in the column specified with *serial* (typically increments with 1)
  - Uniqueness of values must be enforced with a *unique* or *primary key* constraint



Composite primary key

- A composite primary key consist of multiple columns, and rows are uniquely identified across the columns holding the primary key

```
create table bi.Student_course(
    student_number int,
    course_id int,
    accepted varchar(50),
    constraint PK_student_course primary key (student_number, course_id)
    );

insert into bi.student_course
    values (1, 2, 'Accepted')
          ,(1, 3, 'Not accepted')
          ,(2, 1, 'Accepted')
          ,(3, 4, 'Accepted')
          ,(3, 3, 'Awaiting')
          ,(4, 2, 'Not accepted');
```

Name of primary constraint

Primary key is held by two columns

The values across student_number and course_id is unique for each row

	student_number	course_id	accepted
1	1	2	Accepted
2	1	3	Not accepted
3	2	1	Accepted
4	3	3	Awaiting
5	3	4	Accepted
6	4	2	Not accepted

Not Null

- *Null* indicates that a value is absent (not the same as zero)
- By default columns accept *null* values
- By using a *not null* constraint, a column will not accept *null* values
- For example a table containing personal identification numbers and names might have *not null* constraints on both columns

Unique - Alternate key

- The column holding an alternate key uniquely identifies instances in an entity, but is not chosen as the primary key
- We can create an alternate key by creating a *unique* constraint

```
create table bi.app_logins (
    employee_ID serial primary key,
    username varchar(100) not null,
    password varchar(100) not null,
    constraint AK_password unique(password));
```

Name of constraint

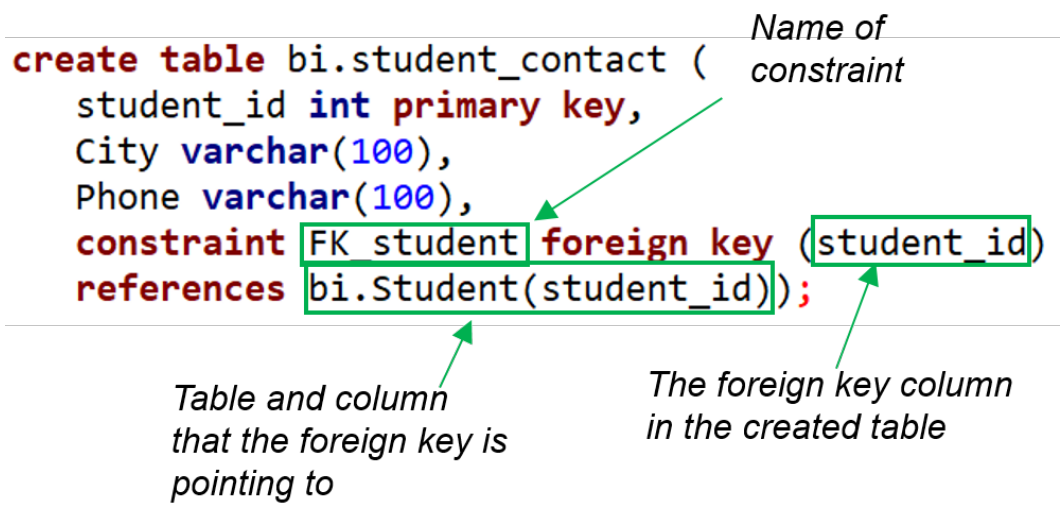
The column with the unique constraint

Foreign Key

- If we have *referential integrity*, then the relations between tables are meaningful
- Foreign keys are used to maintain *referential integrity*
- A Foreign key is a *referential constraint* - it is concerned with how data in one table relates to another table

Foreign Key

- *Foreign Key*: Any value in the column that holds the foreign key must exist in the column that the foreign key points to (references).
- The foreign key must point to (reference) either a primary or a candidate key



Foreign Key

Create an empty table In the bi schema

```
create table bi.student_contact (
   student_id int primary key,
   City varchar(100),
   Phone varchar(100),
   constraint FK_student foreign key (student_id)
   references bi.Student(student_id));
```

Insert values into the table

```
insert into bi.student_contact
   values (2,'Aarhus', '61818192')
          ,(3,'Vejle', '51729824')
          ,(1,'Aarhus N', '31838102')
          ,(4,'Aarhus V', '55018190');
```

Column that hold Foreign key

	student_id	City	Phone
1	1	Aarhus N	31838102
2	2	Aarhus	61818192
3	3	Vejle	51729824
4	4	Aarhus V	55018190

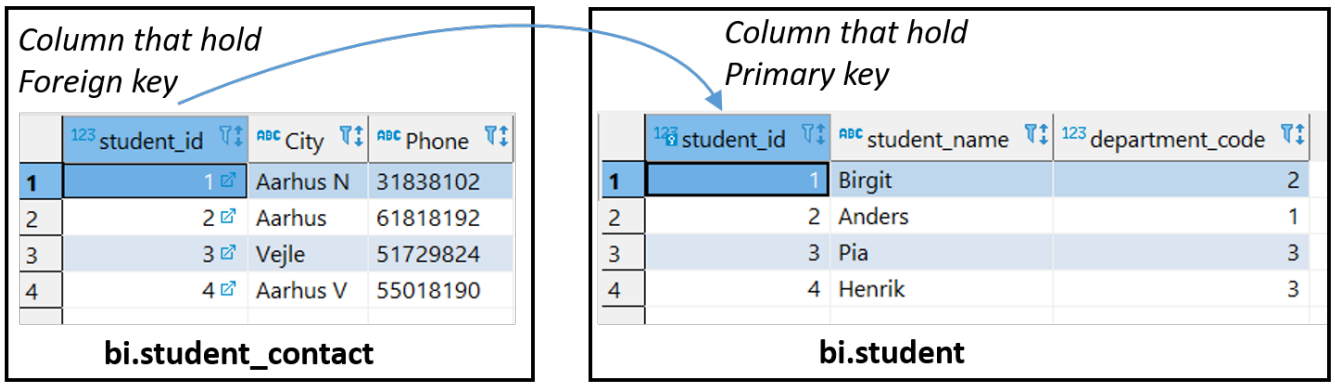
bi.student_contact

Column that hold Primary key

	student_id	student_name	department_code
1	1	Birgit	2
2	2	Anders	1
3	3	Pia	3
4	4	Henrik	3

bi.student

Foreign key and referential integrity



bi.student_contact

	student_id	City	Phone
1	1	Aarhus N	31838102
2	2	Aarhus	61818192
3	3	Vejle	51729824
4	4	Aarhus V	55018190

bi.student

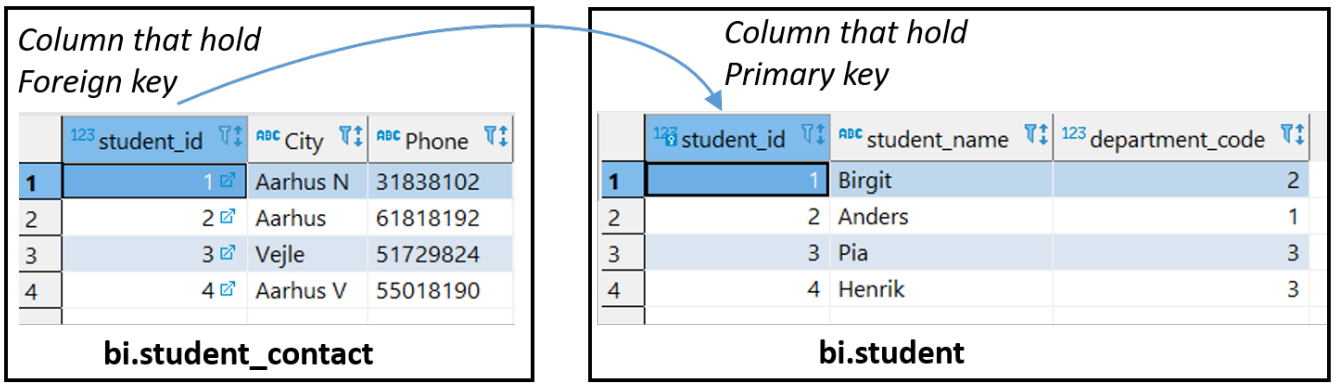
	student_id	student_name	department_code
1	1	Birgit	2
2	2	Anders	1
3	3	Pia	3
4	4	Henrik	3

```
insert into bi.student_contact
    values(5, 'Aarhus V', '55008190');
```

ERROR

We cannot insert the value of 5 in the student_id column of the student_contact table: All the values of the column holding the foreign key does not exists in the referenced column.

Foreign key and referential integrity



bi.student_contact

	student_id	City	Phone
1	1	Aarhus N	31838102
2	2	Aarhus	61818192
3	3	Vejle	51729824
4	4	Aarhus V	55018190

bi.student

	student_id	student_name	department_code
1	1	Birgit	2
2	2	Anders	1
3	3	Pia	3
4	4	Henrik	3

```
delete from bi.student_contact
where student_id = 1;
```

Possible

We can delete the row where student_id = 1 in the student_contact table: All the values of the column holding the foreign key still exists in the references column.

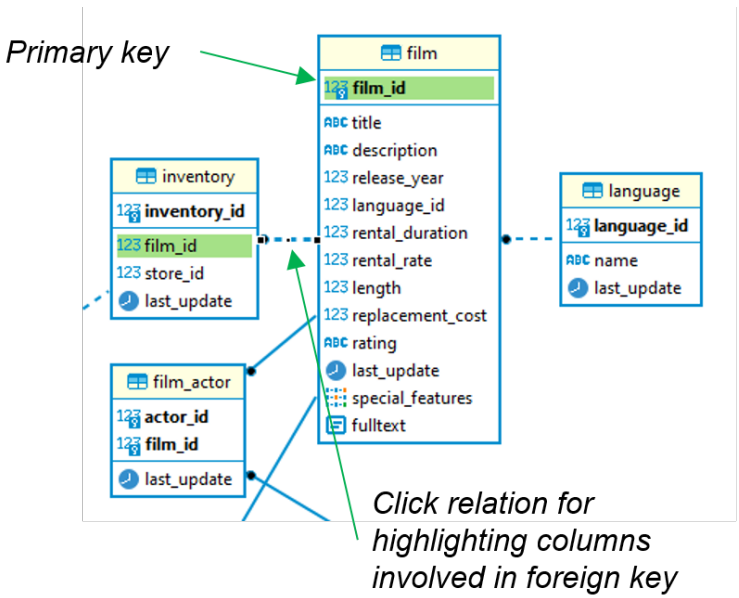
```
delete from bi.student
where student_name = 'Birgit';
```

ERROR

Deleting the row with 'Birgit' breaks referential integrity: All the values of the column holding the foreign key does not exists in the referenced column.

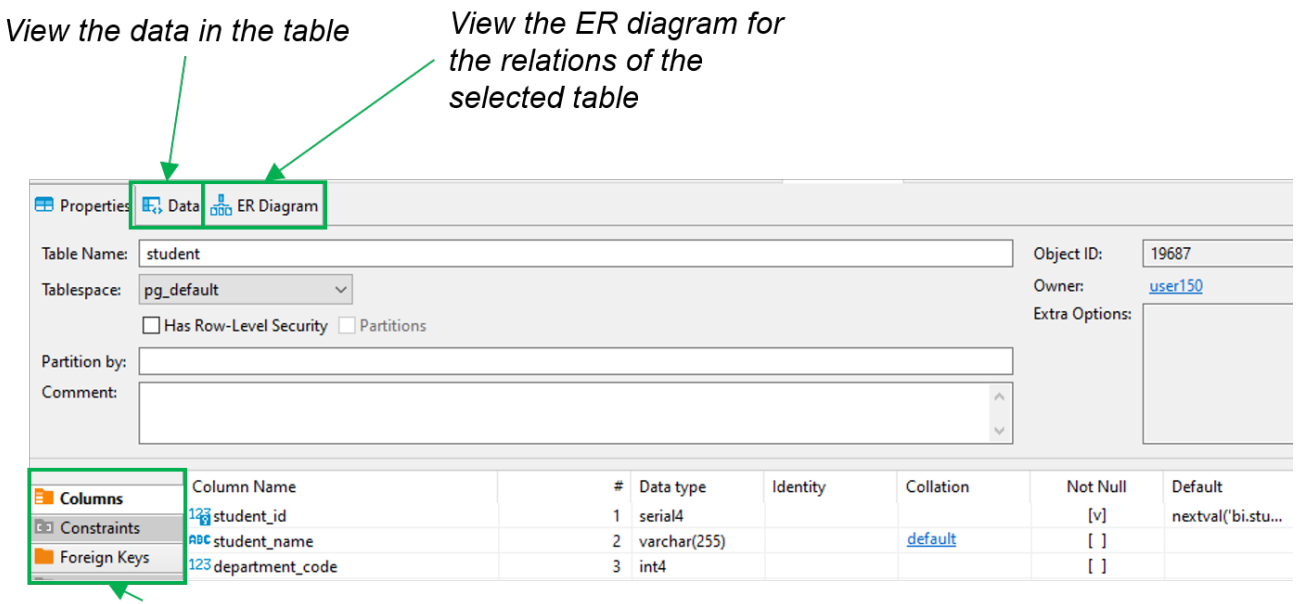
Inspecting ER diagram

- 1) Right click on schema $\longrightarrow$ "View Diagram"
- After you have made changes to the database (e.g. created a table or made a new foreign key), you have to right click on the schema and push *refresh* in order to see the changes in the ER diagram
- Location of the foreign key is denoted with a black dot.



Inspecting a table

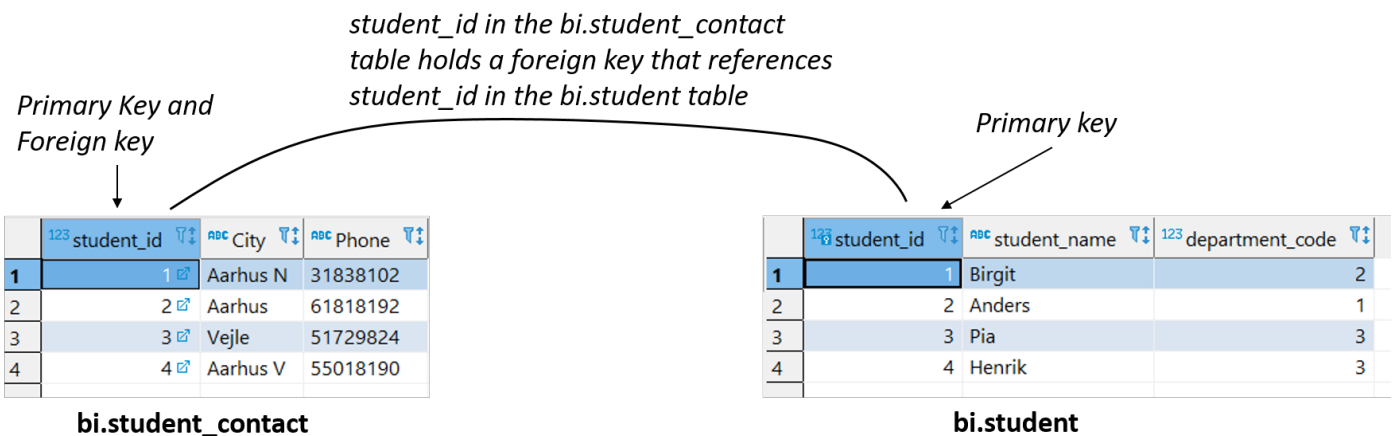
- Inspect a table by double-clicking on it (in the ER diagram or in the database navigator)



- Columns: Column data types, not null constraint, default values etc.
- Constraints: Such as primary keys and check constraints. The rows can be unfolded so you can see the columns involved
- Foreign Keys: Foreign keys in the table. The rows can be unfolded so you can see the columns involved

Identifying relationship

- The relation created on slide 12 is an identifying relationship because the primary key of the parent migrates to the child primary key. This relation is enforced by the foreign key in the *bi.student_contact* table.



	student_id	City	Phone
1	1	Aarhus N	31838102
2	2	Aarhus	61818192
3	3	Vejle	51729824
4	4	Aarhus V	55018190

bi.student_contact

	student_id	student_name	department_code
1	1	Birgit	2
2	2	Anders	1
3	3	Pia	3
4	4	Henrik	3

bi.student

Non-identifying mandatory relationship

```
create table bi.Student (
    student_id serial primary key,
    student_name varchar(255),
    department_code int);

insert into bi.Student
    values (default, 'Birgit', 2),
           (default, 'Anders', 1),
           (default, 'Pia', 3),
           (default, 'Henrik', 3);
```

	student_id	student_name	department_code
1	1	Birgit	2
2	2	Anders	1
3	3	Pia	3
4	4	Henrik	3

Parent entity

```
create table bi.book_sales_order (
    book_sales_id serial primary key,
    student_id int not null,
    constraint FK_book_student_id foreign key (student_id)
    references bi.Student(student_id));

insert into bi.book_sales_order
   values (default, 2),
          (default, 2),
          (default, 1),
          (default, 2),
          (default, 3),
          (default, 4),
          (default, 3);
```

	book_sales_id	student_id
1	1	2
2	2	2
3	3	1
4	4	2
5	5	3
6	6	4
7	7	3

The not null makes the relation mandatory. i.e. the student_id column in the book_sales_order table must have values (cannot be null) and at the same time the values must exist in the student_id column from the student table

Child entity

- Seen from the parent entity: one-to-many relationship
- Seen from the child entity: many-to-one relationship
- Removing the *not null* constraint $\longrightarrow$ non-identifying *optional* relationship.

Non-identifying optional relationship

```
create table bi.Department (
    department_code serial primary key,
    deparment_name varchar(255));

insert into BI.Department
    values (default, 'Computer Science'),
           (default, 'Economics and Business Economics'),
           (default, 'Law'),
           (default, 'Medicine');
```

department_code	deparment_name
1	Computer Science
2	Economics and Business Economics
3	Law
4	Medicine

Parent entity

```
create table bi.Student (
    student_id serial primary key,
    student_name varchar(255),
    department_code int,
    constraint FK_department foreign key (department_code)
    references bi.department(department_code));

insert into BI.Student
    values (default, 'Birgit', 2)
          ,(default, 'Anders', 1)
          ,(default, 'Pia', 3)
          ,(default, 'Henrik', null);
```

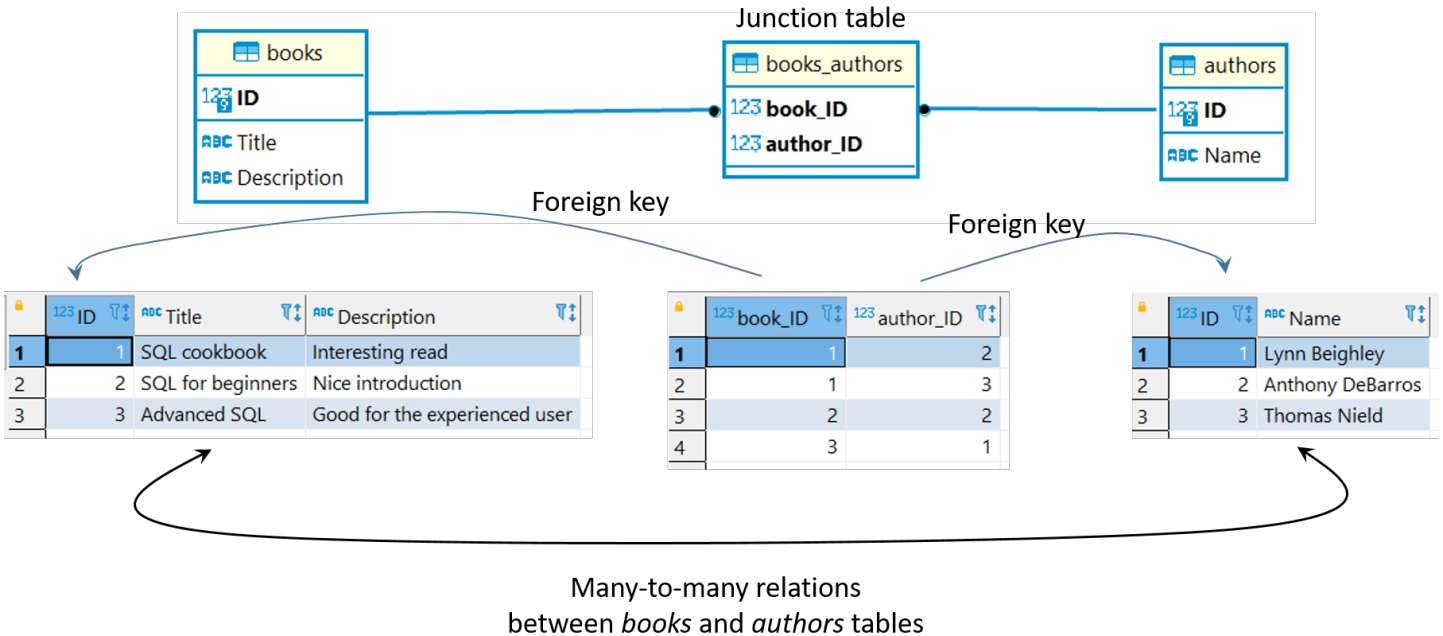
	student_id	student_name	department_code
1	1	Birgit	2
2	2	Anders	1
3	3	Pia	3
4	4	Henrik	[NULL]

Child entity

- The column holding the foreign key can have cells with null, but only values existing in referenced column (*department_code* in *bi.student* cannot take the value of e.g. 5)

Many-to-many relationship

Most difficult to implement: Cannot be implemented directly, so we need a junction table.



	ID	Title	Description
1	1	SQL cookbook	Interesting read
2	2	SQL for beginners	Nice introduction
3	3	Advanced SQL	Good for the experienced user

	book_ID	author_ID
1	1	2
2	1	3
3	2	2
4	3	1

	ID	Name
1	1	Lynn Beighley
2	2	Anthony DeBarros
3	3	Thomas Nield

Exercises I

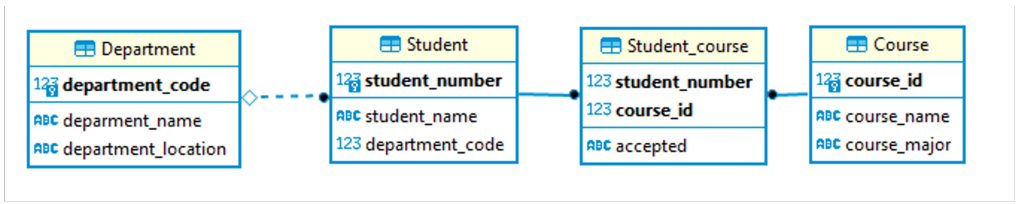
1. Create a table (in e.g. your *bi* schema) with the name *employee* containing the columns *id*, *name* and *business_unit*. Let the *id* hold the primary key, and ensure that a new value is automatically inserted in this column when values in *name* and *business_unit* is inserted. Insert 4 rows in this table (you can decide what the values should be).
2. Create a table with the name *employee_contact* containing the columns *employee_id*, *phone* and *email*. Let the *employee_id* hold the primary key. Also, specify a foreign key from *employee_id* that references the *id* column in the *employee* table.
3. Insert one row of values in the *employee_contact* table that maintain referential integrity with the employee table. Which values of *employee_id* in the *employee_contact* table would not be allowed in your table?

Exercises II

4. Which tables in the *dvdrental* schema have composite primary keys?
5. Find two identifying relations and one non-identifying mandatory relation in the *dvdrental* schema.
6. What kind of relationship would we have between *film.langauge_id* and *language.language_id* if *film.langauge_id* where allowed to have null values?
7. Consider the many-to-many relation depicted on slide 20. Write these tables in your bi schema, but also include the book "Database Systems for real" and assume it is authored by Ramez Elmasri and Thomas Nield.

Exercises III

8 Write SQL queries such that you reproduce the tables, constraints, and values given below. Hint: Consider the order in which you create tables and remember that a black dot in a relation in the ER diagram signifies the location of the foreign key.



BI.Student

	student_number	student_name	department_code
1	1	Birgit	2
2	2	Anders	1
3	3	Pia	3
4	4	Henrik	3

BI.Course

	course_id	course_name	course_major
1	1	Programming I	BSc in Computer Science
2	2	Principles of Economics	BSc in Economics
3	3	Distributed systems	BSc in Computer Science
4	4	Animal Law	Bsc in Law
5	5	Biochemistry	Bsc in Medicine

	department_code	deparment_name	department_location
1	1	Computer Science	Aarhus C
2	2	Economics and Business Economics	Aarhus V
3	3	Law	Aarhus C
4	4	Medicine	Aarhus C

BI.Department

	student_number	course_id	accepted
1	1	2	Accepted
2	1	3	Not accepted
3	2	1	Accepted
4	3	3	Awaiting
5	3	4	Accepted
6	4	2	Not accepted

BI.Student_course

Exercises IV

9 Note: This exercise is challenging but insightful, focusing on database normalization principles. Completing it is beneficial but not crucial. Please give it a try and do your best: A coworker has begun registration of employees, their units, and potential customers in a table. You can find the coworkers script for constructing the table at brightspace: "Learning Material" $\rightarrow$ "SQL in Practice II" $\rightarrow$ "initial_leads_table.sql". Currently, the only restriction on the table is that the "employee_id" column holds the primary key. The purpose is to be able to link employees in terms of their contact with a potential customer (i.e. the leads). For example, the initial table shows that the employee John Smith is within the 'Sales' unit (the sales unit is located in 'Aarhus'), and he is the lead responsible for the contact with 'Danske Commodities'. Another example is Jane Morgan and Vicka Adams who are both leads on the potential customer 'Copenhagen Economics', but Vicki Adams has the main responsibility for this contact. The relationship between the potential customers and the employees should be a many-to-many relationship. Your job is to bring the initial table into the Third Normal Form.

SQL III Joins and Views

Database Management and Data Visualization

Benjamin D. Liengaard
Department of Economics and Business Economics

Overview

Introduction I

- Up until now, tables has existed rather independently of each other
- Today you will learn how you can merge (join) these tables together to form new tables
- Joining tables is an essential skill when using SQL
- We will go through these types of joins
  - Inner joins
  - Left and right joins
  - Full joins

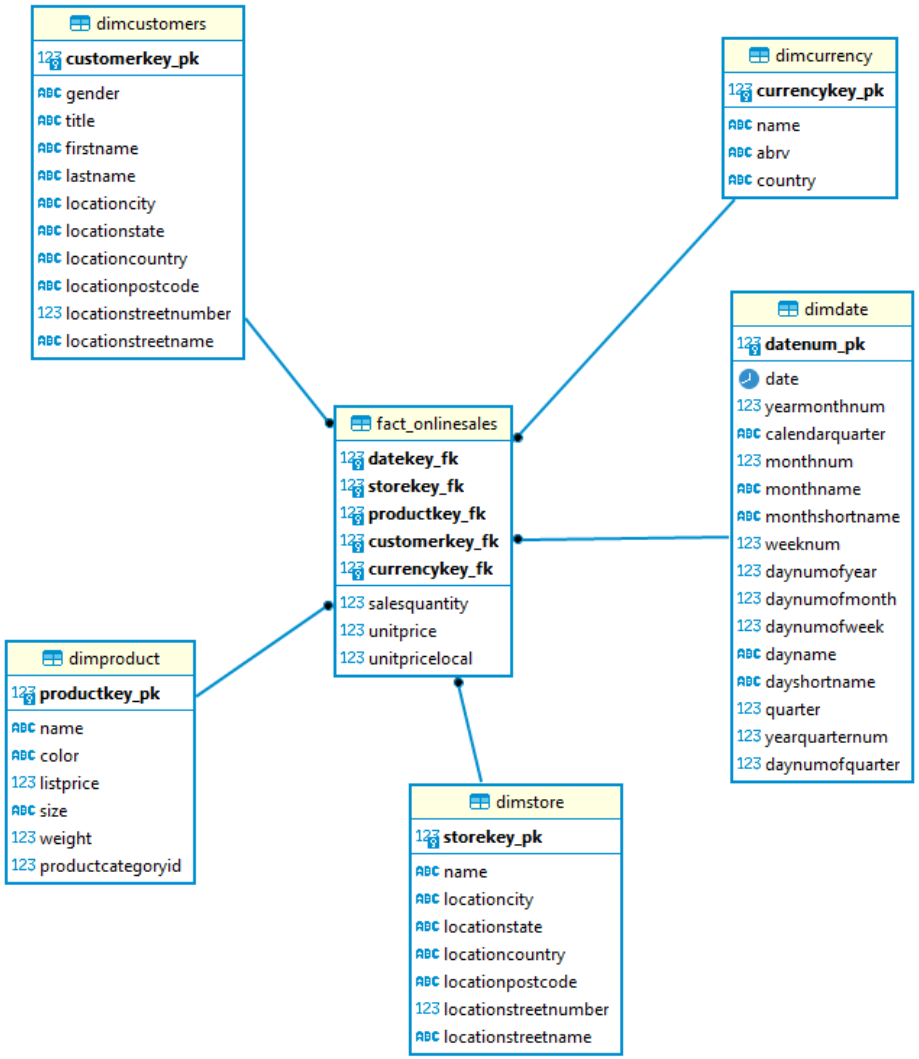
Introduction II

- In the slides covering *inner join* we will also see some generic features that can/should be used for the other joins:
  - The use of alias for tables
  - The use of alias for columns
  - The possibility to make multiple joins in one query
  - The possibility to only return a subset of the columns in the return table
- We can also supplement a join query with some of the clauses we learned in the SQL I lecture (e.g. *where* or *group by*)

The dimensional model *dim_mod*

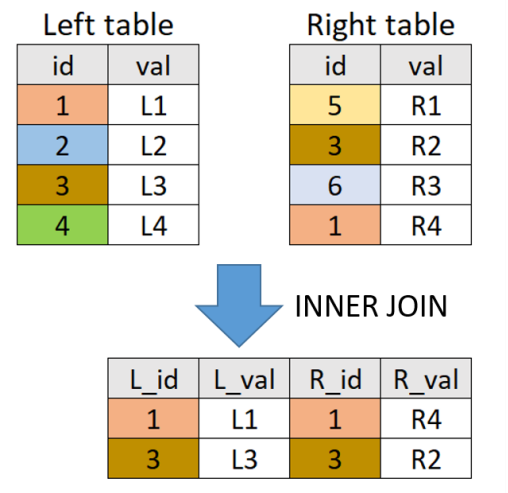
- I have created a dimensional model (as a star schema) in each of your databases
- You can find it in the new schema called *dim_mod*
- It will be used in the lectures and you will use it in relation to the exercises

The dimensional model dim_mod

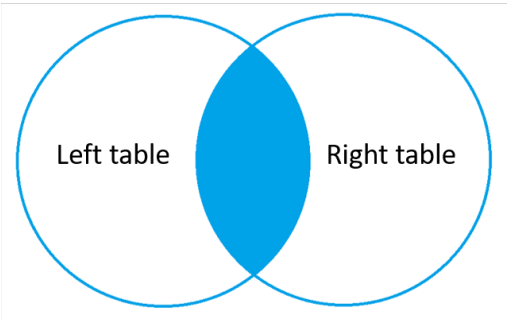


Inner Join - principle

- We join the tables based on the *id* columns
- Only rows that contain matching values in the id columns is retained




```sql
select *
from bi.left_table lt
inner join bi.right_table rt
on lt.ID = rt.ID;
```



Inner Join - query

	department_code	deparment_name	department_location
1	1	Computer Science	Aarhus C
2	2	Economics and Business Economics	Aarhus V
3	3	Law	Aarhus C
4	4	Medicine	Aarhus C

bi.Department

	student_number	student_name	department_code
1	1	Birgit	2
2	2	Anders	1
3	3	Pia	3
4	4	Henrik	3

bi.Student

```
select *
from bi.Student as s
inner join
bi.Department as d
on s.department_code = d.department_code;
```

Alias

Column *Department_code* from the *Student* table

Column *Department_code* from the *Department* table

	student_number	student_name	department_code	department_code	deparment_name	department_location
1	1	Birgit	2	2	Economics and Business Economics	Aarhus V
2	2	Anders	1	1	Computer Science	Aarhus C
3	3	Pia	3	3	Law	Aarhus C
4	4	Henrik	3	3	Law	Aarhus C

Inner Join - Alias I

- By using the alias from the tables we can specify which columns to select

Select the *student_name* column from the *student* table

Select the *department_location* column from the *department* table

```
select s.student_name, d.deparment_name, d.department_location
from bi.Student as s
inner join
bi.Department as d
on s.department_code = d.department_code;
```

	student_name	deparment_name	department_location
1	Birgit	Economics and Business Economics	Aarhus V
2	Anders	Computer Science	Aarhus C
3	Pia	Law	Aarhus C
4	Henrik	Law	Aarhus C

Inner Join - Alias II

- We can also use alias on the columns
- Notice how the result table has changed column names

```
select s.student_name as student,
       d.deparment_name as department,
       d.department_location as location
from bi.Student as s
inner join
bi.Department as d
on s.department_code = d.department_code;
```

Alias

	student	department	location
1	Birgit	Economics and Business Economics	Aarhus V
2	Anders	Computer Science	Aarhus C
3	Pia	Law	Aarhus C
4	Henrik	Law	Aarhus C

Inner Join - multiple tables

- We can join more than two tables in one query

	department_code	deparment_name	department_location
1	1	Computer Science	Aarhus C
2	2	Economics and Business Economics	Aarhus V
3	3	Law	Aarhus C
4	4	Medicine	Aarhus C

bi.Department

	student_number	student_name	department_code
1	1	Birgit	2
2	2	Anders	1
3	3	Pia	3
4	4	Henrik	3

bi.Student

	student_number	course_id	accepted
1	1	2	Accepted
2	1	3	Not accepted
3	2	1	Accepted
4	3	3	Awaiting
5	3	4	Accepted
6	4	2	Not accepted

bi.Student_course

```
select s.student_name as student,
       d.deparment_name as department,
       s.student_number,
       sc.course_id
from bi.Student as s
inner join
bi.Department as d
on s.department_code = d.department_code
inner join
bi.Student_course sc
on s.student_number = sc.student_number;
```

Now we are also joining the *student_course* table

	student	department	student_number	course_id
1	Birgit	Economics and Business Economics	1	2
2	Birgit	Economics and Business Economics	1	3
3	Anders	Computer Science	2	1
4	Pia	Law	3	3
5	Pia	Law	3	4
6	Henrik	Law	4	2

Left Join

- We can think of a left join as the left table is reaching out to the right table
- We keep all the information from the left table. Where we do not have matches on the id columns, *null* values will be inserted as the right tables values

Left table

id	val
1	L1
2	L2
3	L3
4	L4

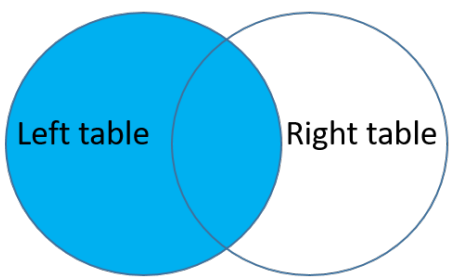
Right table

id	val
5	R1
3	R2
6	R3
1	R4

LEFT JOIN

L_id	L_val	R_id	R_val
1	L1	1	R4
2	L2		
3	L3	3	R2
4	L4		

```
select *
from bi.left_table lt
left join bi.right_table rt
on lt.ID = rt.ID;
```



Left Join - example

	department_code	deparment_name	department_location
1	1	Computer Science	Aarhus C
2	2	Economics and Business Economics	Aarhus V
3	3	Law	Aarhus C
4	4	Medicine	Aarhus C

bi.Department

	student_number	student_name	department_code
1	1	Birgit	2
2	2	Anders	1
3	3	Pia	3
4	4	Henrik	3

bi.Student

```
select d.department_code, d.deparment_name, s.student_name
from bi.Department d
left join bi.Student s
on d.department_code = s.department_code;
```

The *Law* department appears twice because two students have a match on the *department_code*

	department_code	deparment_name	student_name
1	1	Computer Science	Anders
2	2	Economics and Business Economics	Birgit
3	3	Law	Pia
4	3	Law	Henrik
5	4	Medicine	[NULL]

No student associated with the medicine department

Right Join

- The principle is the same as for the Left Join
- We keep all the information from the right table. Where we do not have matches on the id columns, *null* values will be inserted as the left tables values

Left table

id	val
1	L1
2	L2
3	L3
4	L4

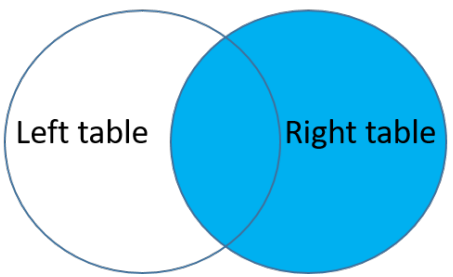
Right table

id	val
5	R1
3	R2
6	R3
1	R4

RIGHT JOIN

L_id	L_val	R_id	R_val
		5	R1
3	L3	3	R2
		6	R3
1	L1	1	R4

```
select *
from bi.left_table lt
right join bi.right_table rt
on lt.ID = rt.ID;
```



Right Join - example

	department_code	deparment_name	department_location
1	1	Computer Science	Aarhus C
2	2	Economics and Business Economics	Aarhus V
3	3	Law	Aarhus C
4	4	Medicine	Aarhus C

bi.Department

	student_number	student_name	department_code
1	1	Birgit	2
2	2	Anders	1
3	3	Pia	3
4	4	Henrik	3

bi.Student

```
select d.department_code, d.deparment_name, s.student_name
from bi.Department d
right join bi.Student s
on d.department_code = s.department_code;
```

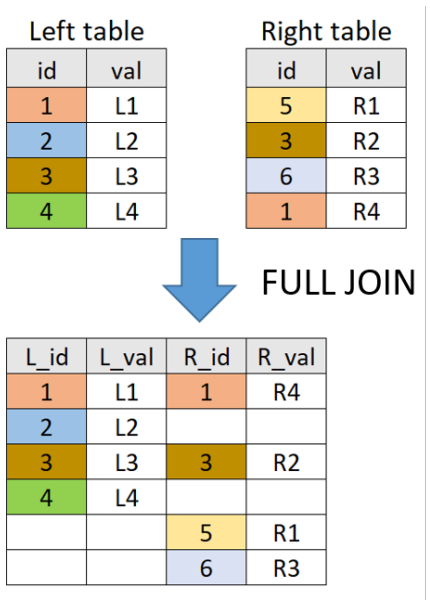
The *Law* department appears twice because two students have a match on the *department_code*

	department_code	deparment_name	student_name
1	2	Economics and Business Economics	Birgit
2	1	Computer Science	Anders
3	3	Law	Pia
4	3	Law	Henrik

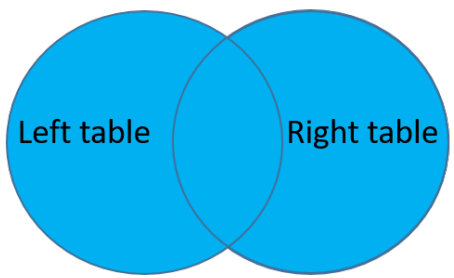
In contrast to the left join, the department *medicine* does not appear in the result table because no student is associated with this department (check the values in *department_code* in the Student table)

Full Join

- We join on matching pairs in the *id* columns, but still retain rows that does not have matches. Where we do not have matches on the id columns, *null* values will be inserted




```
SELECT *
FROM BI_DM.left_table lt
FULL JOIN BI_DM.right_table rt
on lt.ID = rt.ID;
```



Full Join - example

	department_code	deparment_name	department_location
1	1	Computer Science	Aarhus C
2	2	Economics and Business Economics	Aarhus V
3	3	Law	Aarhus C
4	4	Medicine	Aarhus C

bi.Department_1

	student_number	student_name	department_code
1	1	Birgit	2
2	2	Anders	1
3	3	Pia	3
4	4	Henrik	3
5	5	Per	[NULL]

bi.Student_1

We have a student without a *department_code*

```
select s.student_name, d.deparment_name, d.department_location
from bi.Department_1 d
full join bi.Student_1 s
on d.department_code = s.department_code;
```

We have a NULL value in the student name because no student is associated with the Medicine department

	student_name	deparment_name	department_location
1	Birgit	Economics and Business Economics	Aarhus V
2	Anders	Computer Science	Aarhus C
3	Pia	Law	Aarhus C
4	Henrik	Law	Aarhus C
5	Per	[NULL]	[NULL]
6	[NULL]	Medicine	Aarhus C

We have NULL values for *department_name* and *department_location*, because the student Per is not associated with any of the departments in the *department_1* table

Views

- Persistent base tables: Hold the actual SQL data
- Derived tables: The results you see after a query
- View: View definition is stored as a database object but no data is stored in association with the view
- Advantages of view
  - Can store complex queries
  - Instead of recreating queries - invoke the view
  - Handy way to restrict information available to users (e.g not displaying pay rate information to some users)
  - Hide complexity of database and only display user requested information
  - Views is automatically updated when persistent base tables is updated

Creating a View

Schema location and name of view

as keyword precedes the query expression

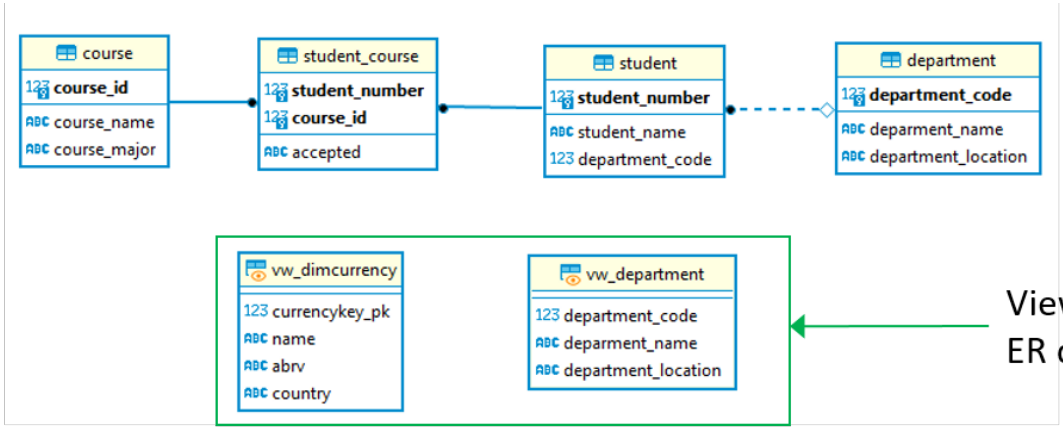
```
create view bi.vw_Department as
select *
from bi.Department;
```

Query expression (View definition)

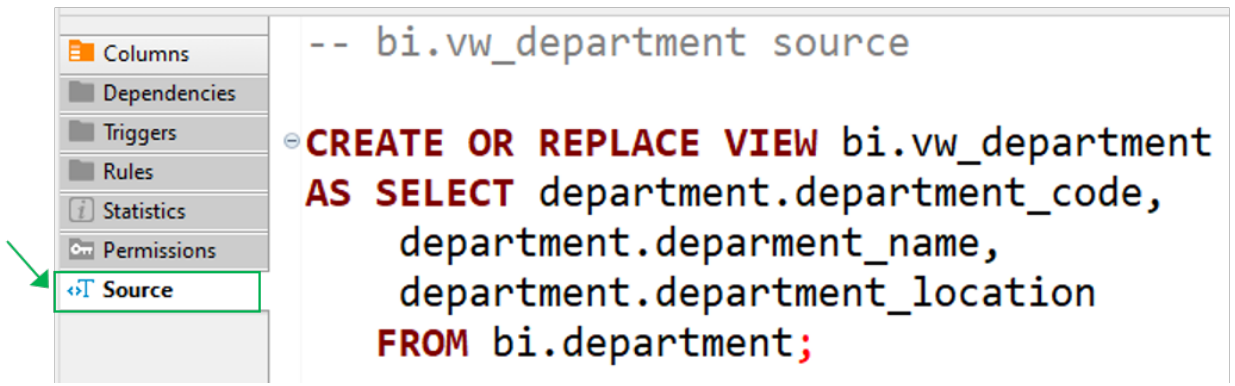
```
create view bi.vw_dimcurrency as
select *
from dim_mod.dimcurrency;
```

Create a view in the *bi* schema using a table from the *dim_mod* schema

View in schema



Can see how the View is created in the DBeaver user interface



Updating base table

	department_code	deparment_name	department_location
1	1	Computer Science	Aarhus C
2	2	Economics and Business Economics	Aarhus V
3	3	Law	Aarhus C
4	4	Medicine	Aarhus C

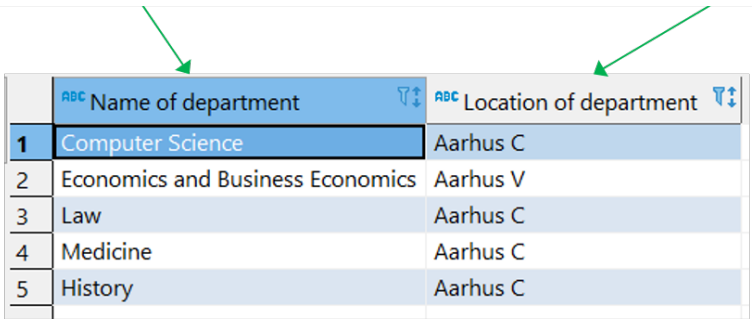
```
select * from bi.vw_department;
--- Insert new value into the base table
insert into bi.Department
    values (default, 'History', 'Aarhus C');
--- The view automatically runs the underlying query again
select * from bi.vw_department;
```

Inserting new value into base table
(notice the View itself is not changed)

	department_code	deparment_name	department_location
1	1	Computer Science	Aarhus C
2	2	Economics and Business Economics	Aarhus V
3	3	Law	Aarhus C
4	4	Medicine	Aarhus C
5	6	History	Aarhus C

Column names

```
--- Delete a View
drop view bi.vw_Department;
--- Give column names to view alternative I
create view bi.vw_Department as
select deparment_name as "Name of department" , department_location as "Location of department"
from bi.Department;
```

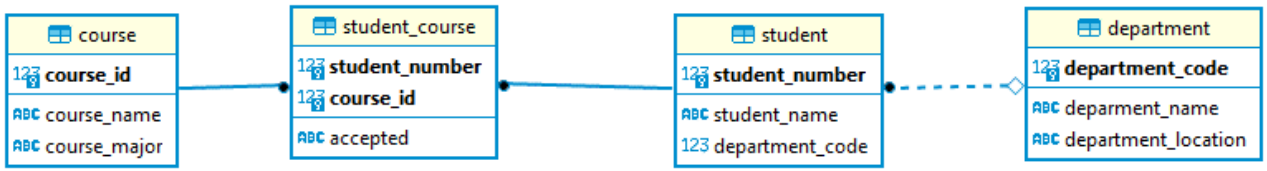


	Name of department	Location of department
1	Computer Science	Aarhus C
2	Economics and Business Economics	Aarhus V
3	Law	Aarhus C
4	Medicine	Aarhus C
5	History	Aarhus C

```
--- Give column names to view alternative II
create view bi.vw_Department ("Name of department", "Location of department") as
select deparment_name, department_location
from bi.Department;
```

View students applied I

- Remember the tables registering students and courses



- Assume you want to keep track of the number of students applying for different courses
- First: Build the query that gives the desired output
- Second: Save it as a view for easy access
- The desired output is

	student_applied	course_name
1	1	Animal Law
2	0	Biochemistry
3	2	Distributed systems
4	2	Principles of Economics
5	1	Programming I

View students applied II

	student_number	course_id	accepted
1	1	2	Accepted
2	1	3	Not accepted
3	2	1	Accepted
4	3	3	Awaiting
5	3	4	Accepted
6	4	2	Not accepted

bi.student_course

	course_id	course_name	course_major
1	1	Programming I	BSc in Computer Science
2	2	Principles of Economics	BSc in Economics
3	3	Distributed systems	BSc in Computer Science
4	4	Animal Law	Bsc in Law
5	5	Biochemistry	Bsc in Medicine

bi.course

Build query step 1

```
select *
from bi.Course c
left join bi.Student_course sc
on sc.course_id = c.course_id;
```

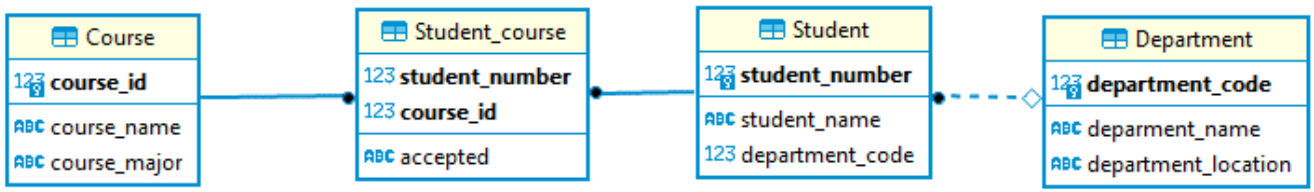
course_id	course_name	course_major	student_number	course_id	accepted
1	Programming I	BSc in Computer Science	2	1	Accepted
2	Principles of Economics	BSc in Economics	1	2	Accepted
2	Principles of Economics	BSc in Economics	4	2	Not accepted
3	Distributed systems	BSc in Computer Science	1	3	Not accepted
3	Distributed systems	BSc in Computer Science	3	3	Awaiting
4	Animal Law	Bsc in Law	3	4	Accepted
5	Biochemistry	Bsc in Medicine	[NULL]	[NULL]	[NULL]

Build query step 2 and create view

```
create view bi.vw_student_applied as
select count(sc.student_number) as student_applied, c.course_name
from bi.Course c
left join bi.Student_course sc
on sc.course_id = c.course_id
group by c.course_name;
```

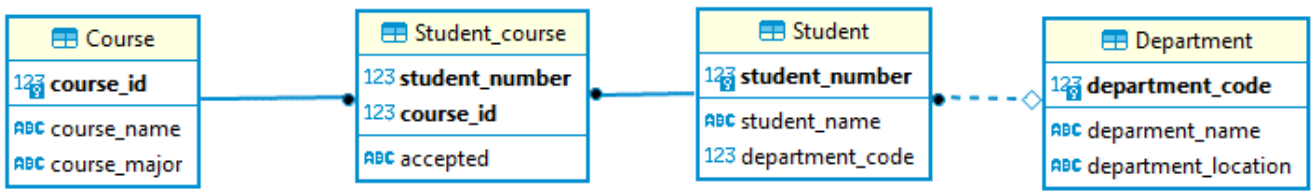
	student_applied	course_name
1	1	Animal Law
2	0	Biochemistry
3	2	Distributed systems
4	2	Principles of Economics
5	1	Programming I

Exercises I



1. Consider the tables given in the figure above. Make an inner join between *Student_course* and *Student*
2. Make the same inner join as in the previous exercise, but only select the columns *accepted* and *student_name*
3. Consider the tables given in the figure above. Make an inner join between *Student_course*, *Student* and *Course* but only select the columns *accepted*, *student_name* and *course_name*

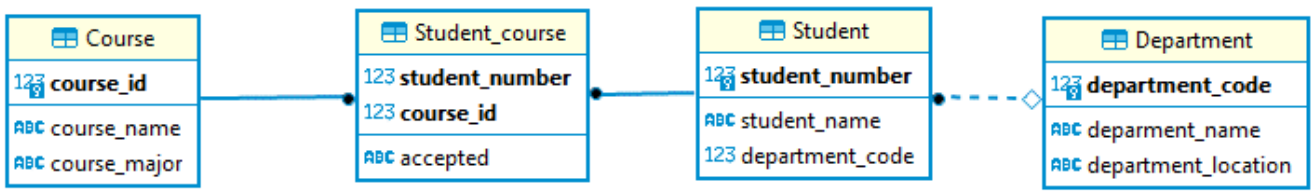
Exercises II



4 Consider the tables given in the figure above. Make a view that gives the number of students applied for the different course majors (i.e. replicate the table below)

	Students applied	course_major
1	3	BSc in Computer Science
2	1	Bsc in Law
3	0	Bsc in Medicine
4	2	BSc in Economics

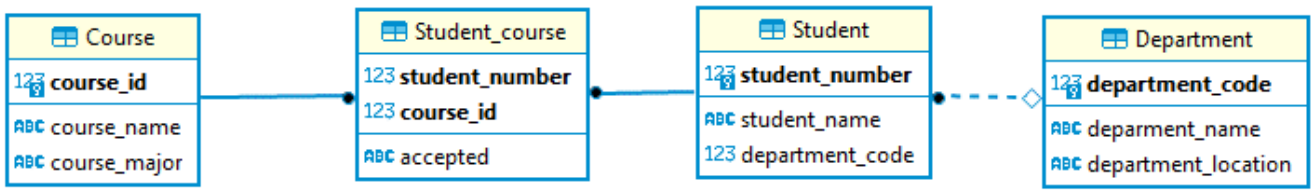
Exercises III



5 Consider the tables given in the figure above. Make a view (call the view *vw_students_all_info*) that gives the number of students who applied for each course with information on department name, and course major (i.e. replicate the table below).

	Students applied	deparment_name	course_name	course_major
1	1	Law	Distributed systems	BSc in Computer Science
2	1	Economics and Business Economics	Distributed systems	BSc in Computer Science
3	0	Medicine	[NULL]	[NULL]
4	1	Law	Principles of Economics	BSc in Economics
5	1	Economics and Business Economics	Principles of Economics	BSc in Economics
6	1	Law	Animal Law	Bsc in Law
7	1	Computer Science	Programming I	BSc in Computer Science

Exercises IV



6 A new student, *Magnus*, is accepted to take the course *Biochemistry* at *Bsc in Medicine* in the *Medicine* department. Update the basetable(s) accordingly. Run the view *vw_students_all_info*) and confirm it is updated correctly (i.e. the view should look like the table below)

	Students applied	deparment_name	course_name	course_major
1	1	Law	Distributed systems	BSc in Computer Science
2	1	Economics and Business Economics	Distributed systems	BSc in Computer Science
3	1	Law	Principles of Economics	BSc in Economics
4	1	Medicine	Biochemistry	Bsc in Medicine
5	1	Economics and Business Economics	Principles of Economics	BSc in Economics
6	1	Law	Animal Law	Bsc in Law
7	1	Computer Science	Programming I	BSc in Computer Science

Exercises V

- For the exercises on this slide, you should use the *dvdrental* database

7 What is the *customer_id* and *first_name* of the customer with most dvd-rentals?

8 How many dvdrentals has each staff member made to the customer with *customer_id* = 148?

9 Which customer (indicated by *customer_id*, *first_name*, and *last_name*) has the highest average film length on their rented dvds?

10 Inspect the view definition of view *public.sales_by_film_category* and describe in words what this view shows. Hint: Inspect *Source* in the *Properties* pane of the view.

Exercises VI

- Consider the dim_mod Schema for the exercises on this slide

11 What is the first and the last date of a unit sold

12 Define a sale as $salesquantity \cdot unitprice$. Make a query that gives the total sale per week day over the entire period. Sort the results such that the weekday with the highest total sale is listed first in the returned table.

SQL IV, Like and Index

Database management and Data Visualization

Benjamin D. Liengaard
Department of Economics and Business Economics

Overview

Like

- Finding text matches
- Can use various flavors of regular expression for text matching
- See documentation **here**
- When used in a *where* clause it evaluates to true/false for each row

Match beginning of string

- In contrast to the PostgreSQL syntax, the textmatching is case sensitive
- % placed last, indicate we are matching on the beginning of the string

```
select *
from bi_five.training t
where t.text like 'Whi%';
```

text
While they were at breakfast the letters were brought in. Among the
Whitwell till you return."
While the family were in this confusion, Charlotte Lucas came to spend
While they were dressing, he came two or three times to their different
While settling this point, she was suddenly roused by the sound of the

```
select *
from bi_five.training t
where t.text like 'whi%';
```

text
which overpowered them at first, was voluntarily renewed, was sought
while she lives, rather than for them--something of the annuity kind I
which, in her opinion, could alone be called taste. Yet, though
which, if it did not denote indifference, spoke of something almost as
which recollection called forth as they entered the house were soon
which her husband's wanted. But they would have been improved by some

Match end or anywhere

```
select *
from bi_five.training t
where t.text like '%Dashwood';
```

text
She concluded with a very kind invitation to Mr. and Mrs. John Dashwood
the most suitable period for its accomplishment. But Mrs. Dashwood
amazement. For a few moments every one was silent. Mrs. Dashwood
all sides. Sir John was ready to like anybody, and though Mr. Dashwood
appearance to think his acquaintance worth having; and Mr. Dashwood

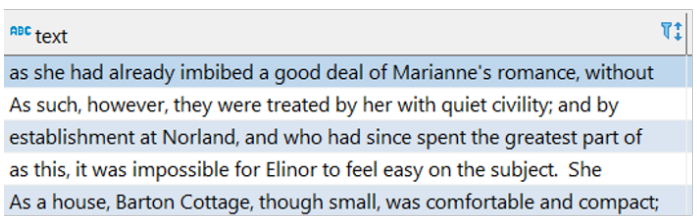
```
select *
from bi_five.training t
where t.text like '%Dashwood%';
```

text
The family of Dashwood had long been settled in Sussex. Their estate
into his house the family of his nephew Mr. Henry Dashwood, the legal
Mrs. Henry Dashwood to his wishes, which proceeded not merely from
By a former marriage, Mr. Henry Dashwood had one son: by his present
bequest. Mr. Dashwood had wished for it more for the sake of his wife

Figure: Query to the left match on the end of a string, and query to the right match anywhere in the string

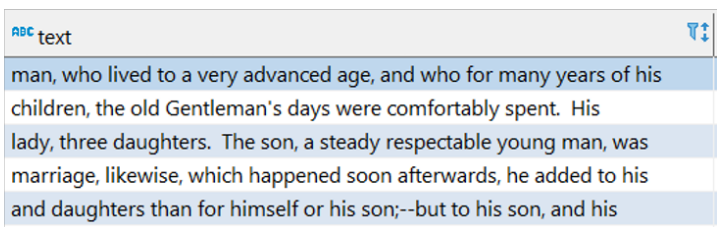
Using the _ wildcard

```
select *
from bi_five.training t
where t.text like '_s%';
```



text
as she had already imbibed a good deal of Marianne's romance, without
As such, however, they were treated by her with quiet civility; and by
establishment at Norland, and who had since spent the greatest part of
as this, it was impossible for Elinor to feel easy on the subject. She
As a house, Barton Cottage, though small, was comfortable and compact;

```
select *
from bi_five.training t
where t.text like '%_s';
```



text
man, who lived to a very advanced age, and who for many years of his
children, the old Gentleman's days were comfortably spent. His
lady, three daughters. The son, a steady respectable young man, was
marriage, likewise, which happened soon afterwards, he added to his
and daughters than for himself or his son;--but to his son, and his

Figure:

Query to the left: First character in the string can be any character, and the second character should be an 's'.

Query to the right: Second last character can be any character, and last character should be an 's'

Using the escape character \

```
select *
from bi_five.training t
where t.text like '%\%';
```

text
not the entreaty of her eldest girl induced her first to reflect on the %

```
select *
from bi_five.training t
where t.text like '\_%';
```

text
_ unaccountably follow me out of the room. And Elinor, in quitting
may fall in love with one of them, and therefore you must visit him as
did--I heard something about it--but I hardly know what--something
his pride, if he had not mortified _mine_."
begin freely--a slight preference is natural enough; but there are

Figure:
Query to the left: Last character should be '%'.
Query to the right: First character should be '_'

Matching on the escape character \

```
select *
from bi_five.training t
where t.text like '%\\%';
```

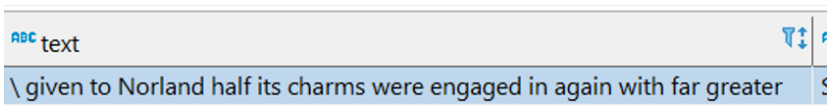


Figure:Matches if '\' exists anywhere in the string.

Indexes overview

- We can use indexes to improve the data retrieval time
- They are specified on one (or more) column(s) in a table
- Indexes use more storage and require more write operations in the background
- It takes longer to update or insert into tables with indexes
- PostgresSQL automatically indexes primary keys
- You should not index every column
- Focus on columns you often filter on
- In general, indexing is more beneficial for larger tables compared to smaller tables

Types of indexes

- There is a range of different indextypes
- By default PostgreSQL uses the B-tree type
- Other types can be seen in the documentation **here**

Creating an index

```sql
create table bi.department (
    department_code serial primary key,
    department_name varchar(255),
    department_location varchar(255)
);
-- Insert into table
insert into bi.Department
    values (default, 'Computer Science', 'Aarhus C')
          ,(default, 'Economics and Business Economics', 'Aarhus V')
          ,(default, 'Law', 'Aarhus C');
-- Create index
create index idx_department_name
on bi.department(department_name);
-- Drop index
drop index bi.idx_department_name;
```

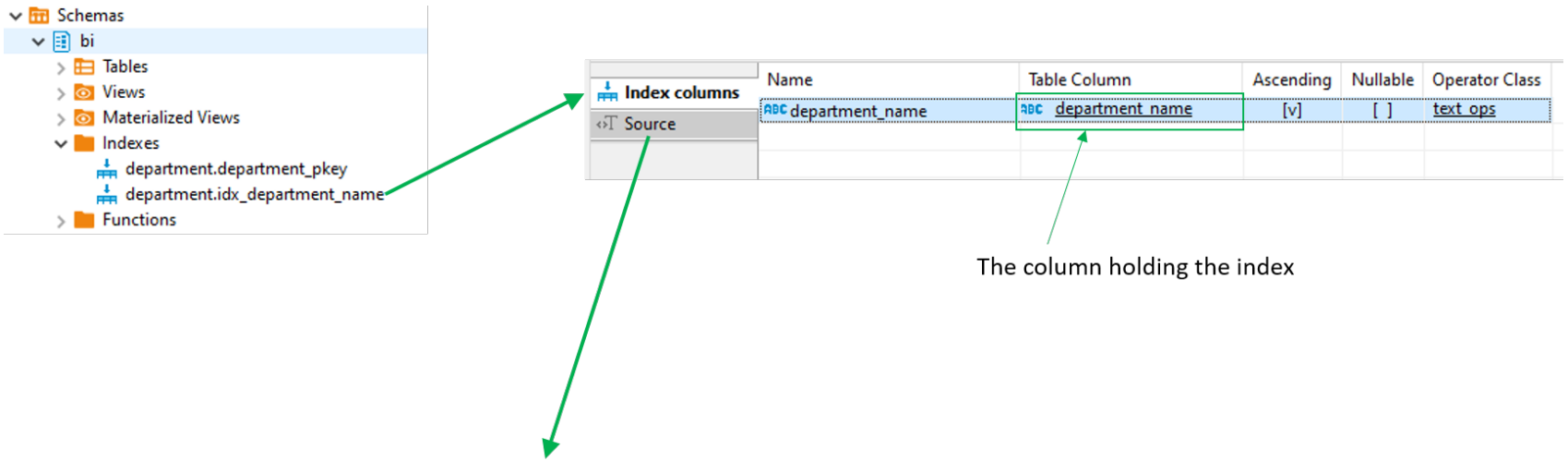
Creating a table (nothing new)

Index name

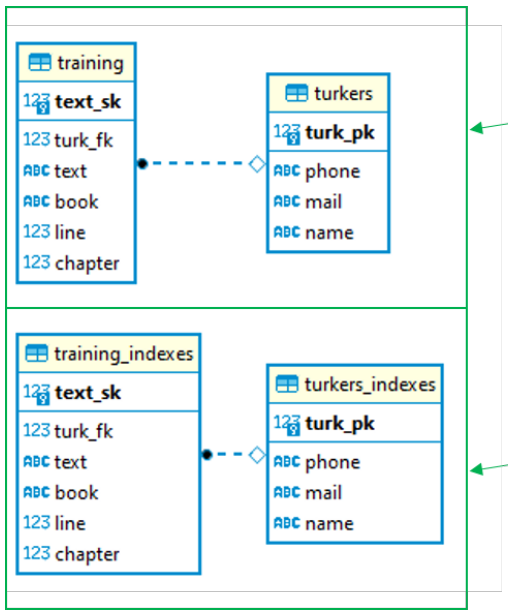
Specify schema, table and column to hold the index.
We can index multiple columns, by separating
column names with a comma.

Deleting an index

Inspecting indexes in DBeaver

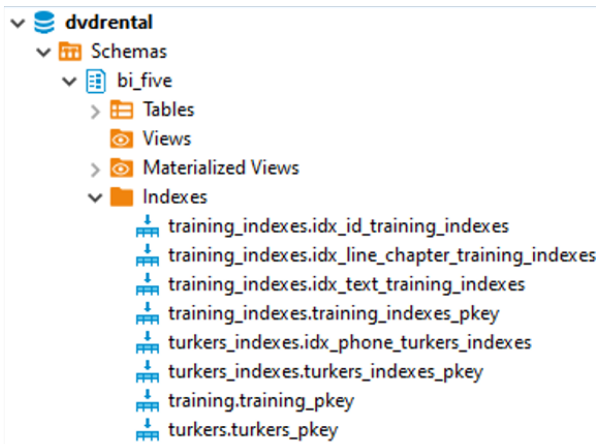


Sample data I

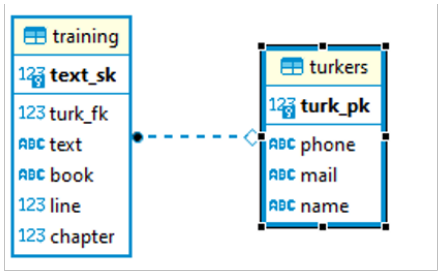


Only indexes on primary keys

Additional indexes specified



Sample data II



text_sk	turk_fk	text	book	line	chapter
23,788	2,053	SENSE AND SENSIBILITY	Sense & Sensibility	1	0
47,576	5,299	The family of Dashwood had long been settled in Sussex. Their estate	Sense & Sensibility	13	1
51,947	4,981	was large, and their residence was at Norland Park, in the centre of	Sense & Sensibility	14	1
19,834	9,493	their property, where, for many generations, they had lived in so	Sense & Sensibility	15	1
17,374	2,812	respectable a manner as to engage the general good opinion of their	Sense & Sensibility	16	1
28,892	2,351	surrounding acquaintance. The late owner of this estate was a single	Sense & Sensibility	17	1
48,307	5,263	man, who lived to a very advanced age, and who for many years of his	Sense & Sensibility	18	1

bi_five.training

turk_pk	phone	mail	name
1	(159)-924-2914	dowxpzi@jkbv.zrb	Dwana Marquardt
2	(956)-428-3189	tve@qbs.mvn	Agnus Mante
3	(457)-542-8321	datq@bcxgwpuf.bj	Zane Sporer
4	(863)-624-4625	abpexwty@dbnaeq	Geraldo Langosh
5	(542)-823-8715	cboiswxdum@hysc	Kristie Strosin
6	(268)-243-6529	xncligbzyu@hwdp	Arica Cassin

bi_five.turkers

Benchmarking I

Get information about our query

```
explain analyze select *
from bi_five.training t
where t.text_sk > 132 and t.text_sk < 52000
order by t.text_sk;
```

QUERY PLAN
Index Scan using training_pkey on training t (cost=0.29
Index Cond: ((text_sk > 132) AND (text_sk < 52000))
Planning Time: 0.529 ms
Execution Time: 29.592 ms

```
explain analyze select *
from bi_five.training_indexes ti
where ti.text_sk > 132 and ti.text_sk < 52000
order by ti.text_sk;
```

QUERY PLAN
Index Scan using idx_id_training_indexes on training_indexes ti (cos
Index Cond: ((text_sk > 132) AND (text_sk < 52000))
Planning Time: 0.532 ms
Execution Time: 31.417 ms

Benchmarking II

```
explain analyze select *
from bi_five.training t
order by t.text;
```

QUERY PLAN
Sort (cost=9305.79..9454.50 rows=59482 width=95) (actua
Sort Key: text
Sort Method: external merge Disk: 6320kB
-> Seq Scan on training t (cost=0.00..1536.82 rows=5948
Planning Time: 0.068 ms
Execution Time: 116.097 ms

```
explain analyze select *
from bi_five.training_indexes ti
order by ti.text;
```

QUERY PLAN
Index Scan using idx_text_training_indexes on training_indexes ti (cos
Planning Time: 0.071 ms
Execution Time: 32.092 ms

The index is used

Exercises I

- For these exercises connect to one of your own databases

1. Create tables according to the *create_tables_for_update_delete_triggers.sql* file. Update the department table such that *Computer Science* is changed to *Computer and Data Science*.
2. Update the department code of the student *Pia* such that she is associated with the *Computer and Data Science* department.
3. Delete the student *Pia* from the *student* table.
4. Create a history table, of the SCD2 type, which is related to the *course* table. Then create a trigger such that this history table automatically gets updated whenever the *course_name* in the *course* table gets updated. Test that the trigger works correctly (i.e. check that the history table gets updated).

Exercises II

- For these exercises connect to the *dvdrental* database

5 Return all rows in the bi_five.training table where rows in the *text* column contains "Henry".

6 Return all rows in the bi_five.training table where rows in the *text* column has the second last character being "p".

Exercises III

- For these exercises connect to the *dvdrental* database

7 Compare the query time between bi_five.turkers and bi_five.turkers_indexes when ordering on the phone *column*.

8 Compare the query time between bi_five.training and bi_five.training_indexes when ordering on the *line* and *chapter* columns simultaneously. Use point-and-click in DBeaver to figure out which columns hold the index.

9 Do a left join with bi_five.training being the left table and bi_five.turkers being the right table. Also, do a left join with bi_five.training_indexes being the left table and bi_five.turkers_indexes being the right table. In both cases, after the join is completed you should sort on the phone column. Can you generally say that one operation is faster than another? Do we still use an index after the join operation?

Exercises IV

- For this exercise connect to one of your own databases

10 Create a small table with an id column and a text column with some random text. Create an index on the text column. Verify that the index is created by point-and-click in DBeaver.

SQL IV Update, Delete and Triggers

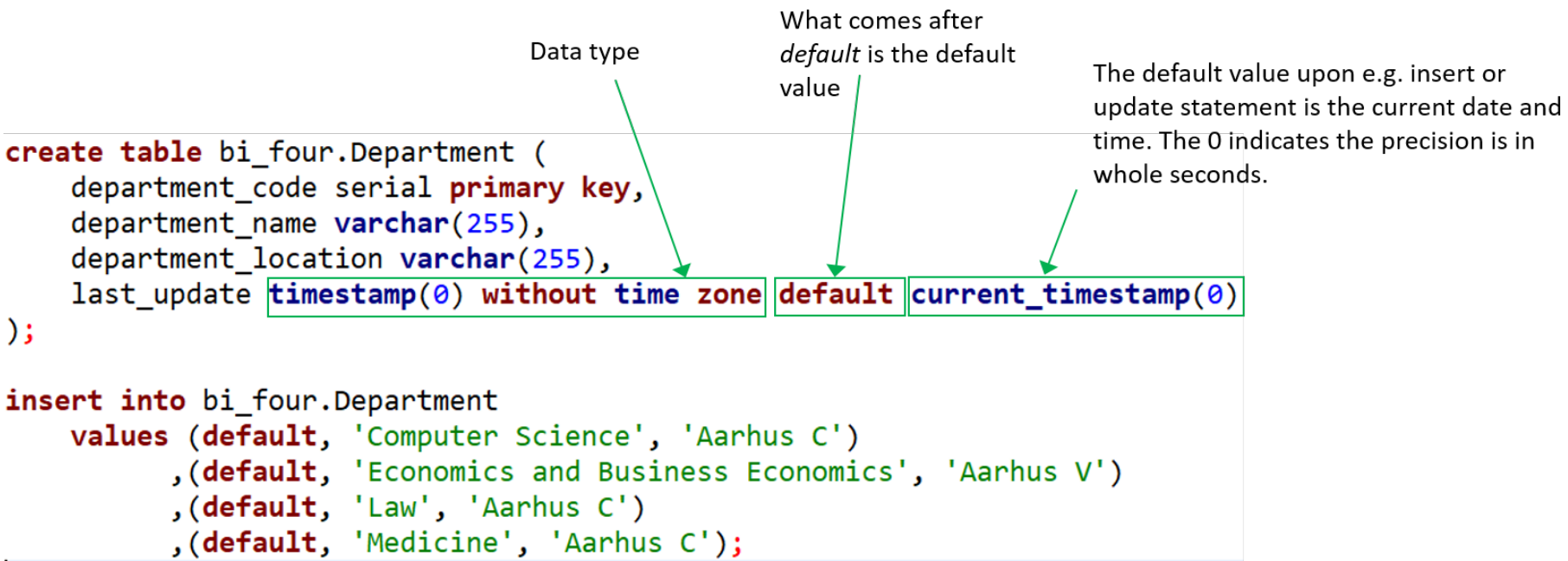
Database Management and Data Visualization

Benjamin D. Liengaard
Department of Economics and Business Economics

Overview

Default values

- Default values can be called when we use e.g. an insert statement or update statement
- The *serial* data type automatically sets a default value on the column (autoincrementing integers)




```sql
create table bi_four.Department (
    department_code serial primary key,
    department_name varchar(255),
    department_location varchar(255),
    last_update timestamp(0) without time zone default current_timestamp(0)
);

insert into bi_four.Department
    values (default, 'Computer Science', 'Aarhus C')
          ,(default, 'Economics and Business Economics', 'Aarhus V')
          ,(default, 'Law', 'Aarhus C')
          ,(default, 'Medicine', 'Aarhus C');
```

Update row

course_id	course_name	course_major	last_update
1	Programming I	BSc in Computer Science	2022-10-31 00:06:12.000
2	Principles of Econom	BSc in Economics	2022-10-31 00:06:12.000
3	Distributed systems	BSc in Computer Science	2022-10-31 00:06:12.000
4	Animal Law	Bsc in Law	2022-10-31 00:06:12.000
5	Biochemistry	Bsc in Medicine	2022-10-31 00:06:12.000

bi_four.course

Columns to be updated

```
update four.course
set course_name = 'Programming 1.01',
last_update = default
where course_id = 1;
```

New values

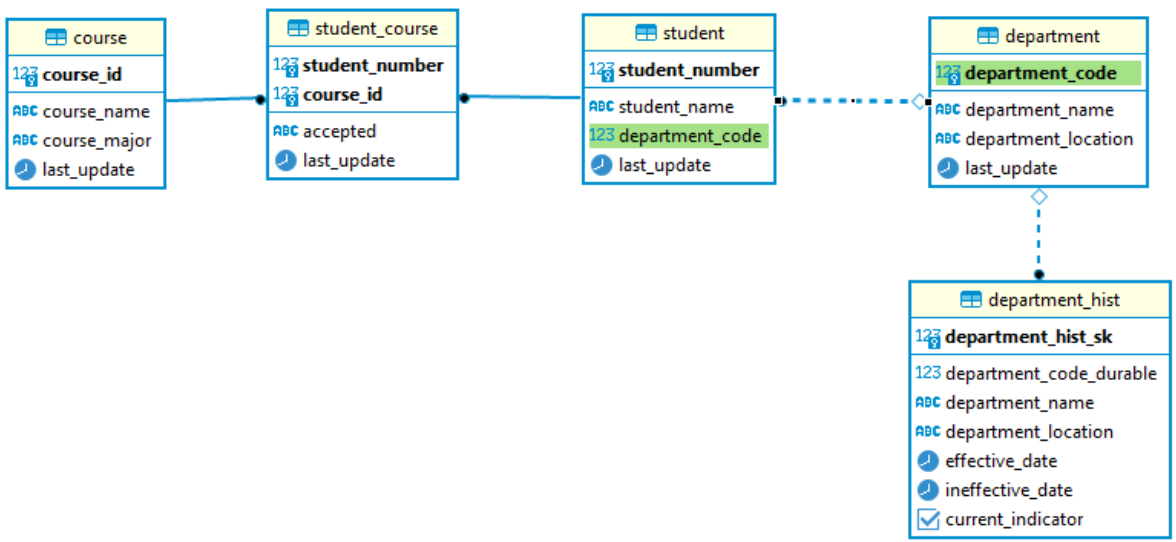
Rows satisfying this condition will be updated

course_id	course_name	course_major	last_update
1	Programming 1.01	BSc in Computer Science	2022-10-31 00:10:56.000
2	Principles of Economics	BSc in Economics	2022-10-31 00:06:12.000
3	Distributed systems	BSc in Computer Science	2022-10-31 00:06:12.000
4	Animal Law	Bsc in Law	2022-10-31 00:06:12.000
5	Biochemistry	Bsc in Medicine	2022-10-31 00:06:12.000

bi_four.course

The bi_four schema

- You can load the schema below into your database by running the script *create_tables_for_update_delete_triggers.sql*
- We have a student who has changed department. We will focus on updating the *student* table accordingly.
- In the update statement, we would like to only give the student name and the name of the new department.



Update row using another table

	student_number	student_name	department_code	last_update
1	1	Birgit	2	2022-10-31 10:41:10.000
2	2	Anders	1	2022-10-31 10:41:10.000
3	3	Pia	3	2022-10-31 10:42:34.000
4	4	Henrik	3	2022-10-31 10:41:10.000

bi_four.student

	department_code	department_name	department_location	last_update
1	1	Computer Science	Aarhus C	2022-10-31 10:41:10.000
2	2	Economics and Business Econom	Aarhus V	2022-10-31 10:41:10.000
3	3	Law	Aarhus C	2022-10-31 10:41:10.000
4	4	Medicine	Aarhus C	2022-10-31 10:41:10.000

bi_four.department

Allows us to use values from the department table

```
update bi_four.student
set department_code = d.department_code,
    last_update = default
from
bi_four.department d
where
(d.department_name = 'Medicine') and (student_name  = 'Henrik');
```

Using value(s) from the department table to insert in the student table.

Notice we have restrictions on both tables

	student_number	student_name	department_code	last_update
1	1	Birgit	2	2022-10-31 10:41:10.000
2	2	Anders	1	2022-10-31 10:41:10.000
3	3	Pia	3	2022-10-31 10:42:34.000
4	4	Henrik	4	2022-10-31 10:48:23.000

bi_four.student

Update row using another table - alternative

	student_number	student_name	department_code	last_update
1	1	Birgit	2	2022-10-31 10:41:10.000
2	2	Anders	1	2022-10-31 10:41:10.000
3	3	Pia	3	2022-10-31 10:42:34.000
4	4	Henrik	3	2022-10-31 10:41:10.000

bi_four.student

	department_code	department_name	department_location	last_update
1	1	Computer Science	Aarhus C	2022-10-31 10:41:10.000
2	2	Economics and Business Econom	Aarhus V	2022-10-31 10:41:10.000
3	3	Law	Aarhus C	2022-10-31 10:41:10.000
4	4	Medicine	Aarhus C	2022-10-31 10:41:10.000

bi_four.department

```
update bi_four.student
set department_code = (select d.department_code from bi_four.department d
                       where d.department_name  = 'Medicine'),
    last_update = default
where
student_name  = 'Henrik';
```

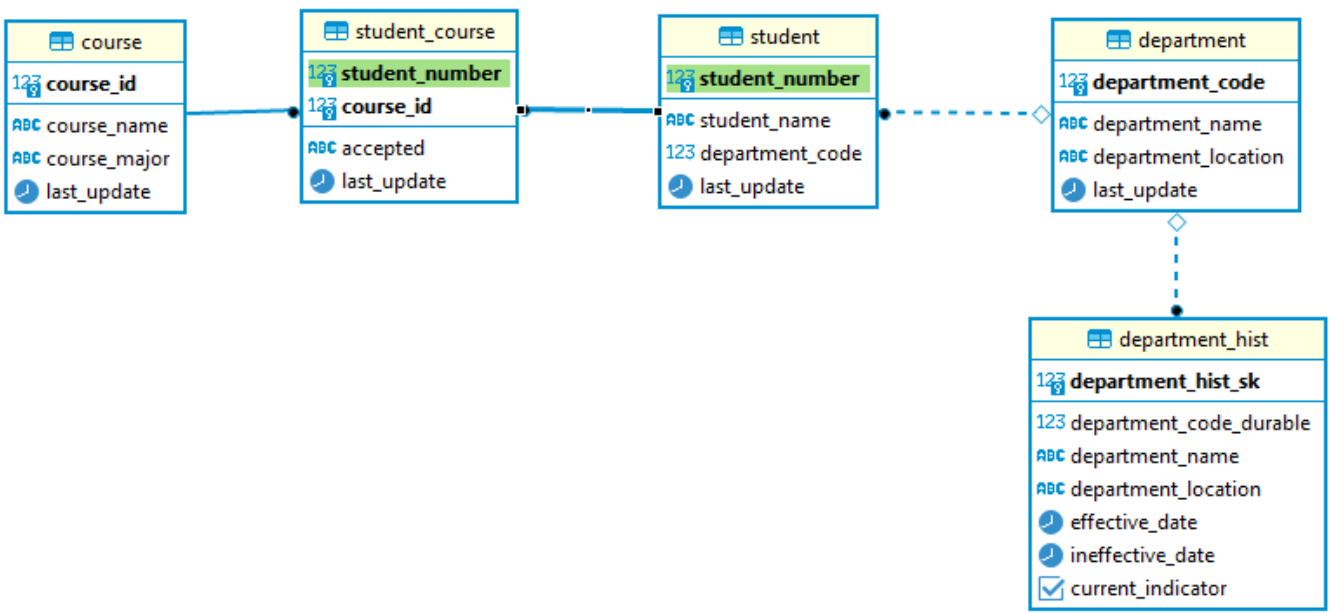
Explicitly specifying which values to get from the department table

	student_number	student_name	department_code	last_update
1	1	Birgit	2	2022-10-31 10:41:10.000
2	2	Anders	1	2022-10-31 10:41:10.000
3	3	Pia	3	2022-10-31 10:42:34.000
4	4	Henrik	4	2022-10-31 10:48:23.000

bi_four.student

Delete student

- We need to remove a student who has dropped out. The student is registered in the *student_course* and *student* tables.
- There is a foreign key from the *student_course* pointing to *student*, so we need to delete in *student_course* before we can delete in *student*



Delete in child table holding foreign key

	student_number	course_id	accepted	last_update
1	1	2	Accepted	2022-10-31 11:18:48.000
2	1	3	Not accepted	2022-10-31 11:18:48.000
3	2	1	Accepted	2022-10-31 11:18:48.000
4	3	4	Accepted	2022-10-31 11:18:48.000
5	3	3	Awaiting	2022-10-31 11:18:48.000
6	4	2	Not accepted	2022-10-31 11:18:48.000

bi_four.student_course

student_number	student_name	department_code	last_update
1	Birgit	2	2022-10-31 11:18:48.000
2	Anders	1	2022-10-31 11:18:48.000
3	Pia	3	2022-10-31 11:18:58.000
4	Henrik	4	2022-10-31 11:18:58.000

bi_four.student

```
delete
from bi_four.student_course sc
where
sc.student_number  =
        (select s.student_number
        from bi_four.student s
        where s.student_name = 'Henrik');
```

Getting the student_number value from the student table

	student_number	course_id	accepted	last_update
1	1	2	Accepted	2022-10-31 11:18:48.000
2	1	3	Not accepted	2022-10-31 11:18:48.000
3	2	1	Accepted	2022-10-31 11:18:48.000
4	3	4	Accepted	2022-10-31 11:18:48.000
5	3	3	Awaiting	2022-10-31 11:18:48.000

Delete in parent table

- Now student_number 4 does not appear in the *student_course* table and we can delete the student from the student table without violating any foreign key constraint

student_number	student_name	department_code	last_update
1	Birgit	2	2022-10-31 11:18:48.000
2	Anders	1	2022-10-31 11:18:48.000
3	Pia	3	2022-10-31 11:18:58.000
4	Henrik	4	2022-10-31 11:18:58.000

bi_four.student

```
delete
from bi_four.student s
where s.student_name = 'Henrik'
```

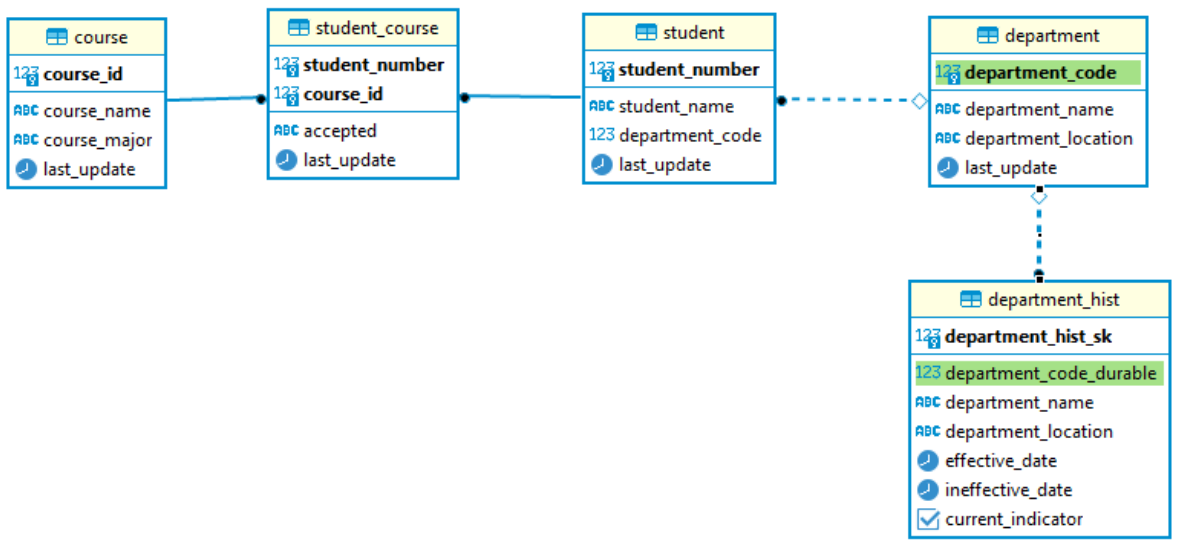
student_number	student_name	department_code	last_update
1	Birgit	2	2022-10-31 11:18:48.000
2	Anders	1	2022-10-31 11:18:48.000
3	Pia	3	2022-10-31 11:18:58.000

Triggers overview

- Triggers are user-defined functions related to a table
- They are automatically called into action when a specified event (as e.g. "update" or "delete") occurs on the table
- Two types of triggers
  - Row-level trigger: Trigger function will be called as many times as the number of rows affected
  - Statement-level trigger: Will be called as many times as a statement is given
- Two steps in creating a trigger
  - Creating the trigger function
  - Creating the trigger
- We will look at a row-level trigger to show how we can mimic the behavior of a Slowly Changing Dimension type 7 (our simple example is not a dimensional model, but you will encounter one in the exercise).

Tracking department name changes

- We will setup a trigger that stores historical values of a department name whenever the department name is changed in the *department* table



Creating the trigger function

Command specifying that we are creating a user-defined function

Name of the function

We do not provide any arguments to the trigger function

Specifies that we are dealing with a trigger function.

Which language we will write the function in. Here we use Procedural Language (can also be others like Python or R)

Here we write the actual function

```
create or replace function log_department_changes()
  returns trigger
  language PLPGSQL
  as
$$
begin
    if new.department_name <> old.department_name then
         --- Update ineffective date and current indicator on old row
         update bi_four.department_hist
         set ineffective_date = now(),
             current_indicator = '0'
         where department_hist_sk = (select max(department_hist_sk)
                                     from bi_four.department_hist  dh
                                     where (dh.department_code_durable = old.department_code));
         --- Insert new values in new row
         insert into bi_four.department_hist(department_hist_sk, department_code_durable,
                                     department_name, department_location,
                                     effective_date, ineffective_date, current_indicator)
         values(default, old.department_code, new.department_name,
                old.department_location, now(), '3022-10-30 23:19:23', '1');
    end if;
return new;
end;
$$
```

Creating the trigger function II

The function is wrapped in an if-statement. It only runs if the new department_name is different from the old department_name

new: Refers to the new row from the update/insert statement

Different from

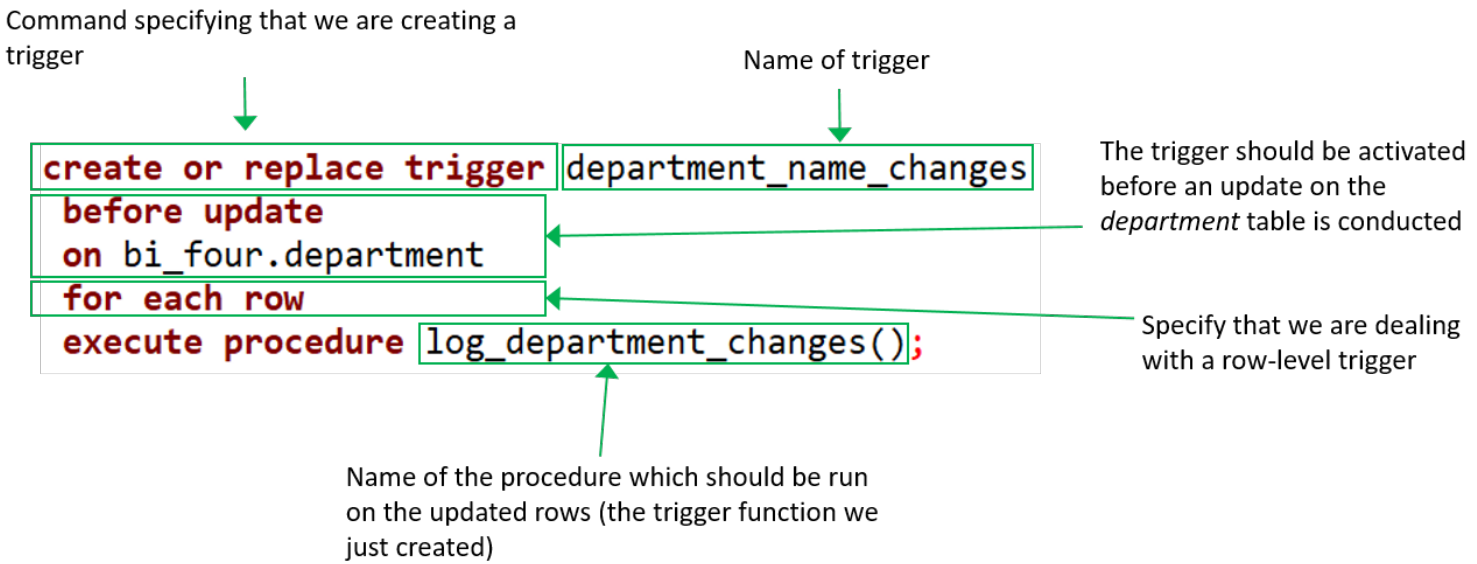
old: Refers to the old row. I.e. the row before the update/insert statement is run

```
    if new.department_name <> old.department_name then
        --- Update ineffective date and current indicator on old row
        update bi_four.department_hist
        set ineffective_date = now(),
            current_indicator = '0'
        where department_hist_sk = (select max(department_hist_sk)
                                    from bi_four.department_hist  dh
                                    where (dh.department_code_durable = old.department_code));
        --- Insert new values in new row
        insert into bi_four.department_hist(department_hist_sk, department_code_durable,
                                    department_name, department_location,
                                    effective_date, ineffective_date, current_indicator)
        values(default, old.department_code, new.department_name,
               old.department_location, now(), '3022-10-30 23:19:23', '1');
    end if;
return new;
```

First, we update the history table by changing the ineffective_date column and the current_indicator column for the old record.

Second, we insert a new row in the history table

Creating the trigger



Running an update with the trigger

department_code	department_name	department_location	last_update
1	Computer Science	Aarhus C	2022-10-31 13:27:09.000
2	Economics and Business Econom	Aarhus V	2022-10-31 13:27:09.000
3	Law	Aarhus C	2022-10-31 13:27:09.000
4	Medicine	Aarhus C	2022-10-31 13:27:09.000

bi_four.department

	department_hist_sk	department_code_durable	department_name	department_location	effective_date	ineffective_date	current_indicator
1	1	1	Computer Science	Aarhus C	2022-10-31 13:23:20.000	3022-10-30 23:19:23.000	[v]
2	2	2	Economics and Business Economics	Aarhus V	2022-10-31 13:23:20.000	3022-10-30 23:19:23.000	[v]
3	3	3	Law	Aarhus C	2022-10-31 13:23:20.000	3022-10-30 23:19:23.000	[v]
4	4	4	Medicine	Aarhus C	2022-10-31 13:23:20.000	3022-10-30 23:19:23.000	[v]

bi_four.department_hist

```
update bi_four.department
set department_name = 'Clinical Medicine',
    last_update = default
where department_code = 4;

update bi_four.department
set department_name = 'Law and Justice',
    last_update = default
where department_code = 3;
```

	department_code	department_name	department_location	last_update
1	1	Computer Science	Aarhus C	2022-10-31 13:23:20.000
2	2	Economics and Business Econom	Aarhus V	2022-10-31 13:23:20.000
3	4	Clinical Medicine	Aarhus C	2022-10-31 13:25:32.000
4	3	Law and Justice	Aarhus C	2022-10-31 13:25:37.000

bi_four.department

department_hist_sk	department_code_durable	department_name	department_location	effective_date	ineffective_date	current_indicator
1	1	Computer Science	Aarhus C	2022-10-31 13:27:09.000	3022-10-30 23:19:23.000	[v]
2	2	Economics and Business Economics	Aarhus V	2022-10-31 13:27:09.000	3022-10-30 23:19:23.000	[v]
4	4	Medicine	Aarhus C	2022-10-31 13:27:09.000	2022-10-31 13:32:01.197	[ ]
5	4	Clinical Medicine	Aarhus C	2022-10-31 13:32:01.197	3022-10-30 23:19:23.000	[v]
3	3	Law	Aarhus C	2022-10-31 13:27:09.000	2022-10-31 13:32:02.688	[ ]
6	3	Law and Justice	Aarhus C	2022-10-31 13:32:02.688	3022-10-30 23:19:23.000	[v]

bi_four.department_hist

Exercise - Explanation 1

- In this exercise, you will implement an SCD type 7 schema
- This exercise is comprehensive and you will have to use many of the things you have learned in the SQL in practice lectures
- The exercise can be broken up into three phases

1 Create the schema given on slide 19 (either in your own containerized PostgreSQL, or in the personal database I have made available to you) and populate it with the values given on the slide.
   - Remember to use appropriate data types when creating tables, and also specify the appropriate constraints, e.g., foreign keys. The SQL script 'create_tables_for_update_delete_trigger' contains many useful commands.
   - Hint: the SQL function 'current_date' will return the current date, and behaves similarly to the functions 'current_timestamp(0)' or 'now()', when setting default column values, or when inserting values.

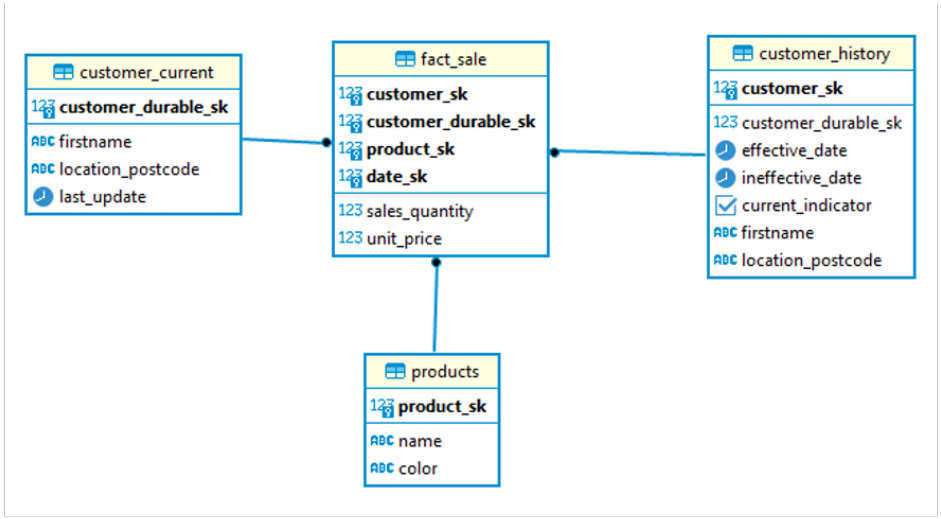
Exercise - Explanation 2

2 Create a trigger function and the trigger
   - Create a trigger function such that updates to the 'customer_current' table trigger a new row to be written in 'customer_history'.
   - Hint: You can use parts of the SQL script 'Update, Delete and Triggers' and modify it for this exercise.

3 Make updates to the 'location_postcode' column in the 'customer_current' table:
   - 3.1 Update 'Donalds' Postcode to 32584
   - 3.2 Update 'Victorias' Postcode to 73611

- Check that the updates is implemented in the 'customer_current' table, and that the 'customer_history' table has been updated appropriately.

Exercise



customer_sk	customer_durable_sk	effective_date	ineffective_date	current_indicator	firstname	location_postcode
1	1	2023-02-01	2023-10-09	[ ]	Donald	98052
2	2	2023-10-10	9999-10-10	[v]	Victoria	46908
3	1	2023-11-09	9999-11-09	[v]	Donald	67133

Rows in '*customer_history*' table

customer_sk	customer_durable_sk	product_sk	date_sk	sales_quantity	unit_price
1	1	2	20,230,202	1	349
2	2	3	20,231,810	1	20
3	1	4	20,231,509	1	503

Rows in '*fact_sale*' table

customer_durable_sk	firstname	location_postcode	last_update
1	Donald	67133	2023-11-09
2	Victoria	46908	2023-10-10

Rows in '*customer_current*' table

product_sk	name	color
1	HL Road Frame - Black, 58	Black
2	AWC Logo Cap	Multi
3	Long-Sleeve Logo Jersey, L	Multi
4	Road-150, 52	Red

Rows in '*products*' table

Figure: Schema *bi_trigger*