CS-A1153

Databases for Data Science

Unified Modeling Language (UML): Part I
Functional Dependencies & Normal Forms: Part I
Defining SQL Tables, Integrity Constraints & Views

April 27, 2021

Aalto University
School of Science

Instructors:
Prof. Nitin Sawhney
Prof. Barbara Keller
Dr. Sami El-Mahgary

Project topics

- There are two topics available for the group project:
  - dairy farm
  - vaccine distribution.
- The formal descriptions of the project topics will be released on MyCourses.
- Decide with your group which project you want to work on.
- Please glance through the topics before your exercise session.

Requirements: UML modelling

- You are first required to create an UML model for the database.
- Use the notation taught in the course, notational errors will affect your grade on the project.
- It is recommended to start working on the UML diagram already this week.

Requirements: Relational model

- Turn your UML model into a relational schema.
- Use the techniques taught in the course.
- Clearly list all the relations and underline key attributes.
- The relational model needs to be based on your UML model.
- It is recommended to do this task on Week 3 after the lecture on the topic.

Requirements: Functional dependencies, anomalies and BCNF

- List the non-trivial functional dependencies in your model.
- Analyse your model to find any anomalies or redundancies.
- Inspect whether or not the model is in BCNF.
- If it is not, make the appropriate changes to the relational model. Document the process of turning the model into BCNF. Submit both the original and the decomposed versions.

Documentation

The document should contain

- The names and student numbers of your group members,
- UML diagram for the model,
- Relational schema of the converted UML,
- Reasoning behind your design choices (recommended length 1-2 pages),
- Answers to the questions about functional dependencies, anomalies and BCNF.

Deadlines and grading

The project has three parts, with the following deadlines

- **Monday, May 10th** at 18:00, Part 1 (modelling)
- Monday, May 24th at 18:00, Part 2 (SQL implementation)
- Friday, June 4th at 18:00, Final (data analysis and corrections)

Instructions for the other parts will be released later.

Each part is worth 50 points, so in total you will get 150 points from the project. You will need to get **at least 50%** (75p.) of the total points to pass the course.

Unified Modeling Language (UML)

CS-A1153: Databases for Data Science

Prof. Nitin Sawhney

Aalto University
School of Science

Acknowledgements

These slides are based in part on presentation materials created by **Kerttu Pollari-Malmi** and **Juha Puustjärvi** in previous years and on the course text book: **A First Course in Database Systems**, Third Edition. Pearson by Jeffery D. Ullman and Jennifer Widom.

Thanks to **Etna Lindy** & **Ville Vuorenmaa** for translating prior lecture slides for this course.

Learning Goals

- Describing databases with the **Unified Modeling Language (UML)**
- Familiarity with fundamental concepts of UML-modeling:
  - class and attribute
  - association
  - association class
  - subclass
  - composition and aggregation

UML Modeling

- *Problem:* what relations should we define for our database?
- Often it is easier to start planning from a higher abstraction level than to dive into a relational model.
- Here we introduce one way of represent things on a higher abstraction level using UML diagrams.
- UML was originally created for designing object-oriented programs. For database design we only use a small portion of the properties and features of UML.
- Other commonly used methods to represent high-level database designs:
  - E/R diagrams (Entity-Relationship)
  - ODL (Object Description Language)

Differences with object-oriented UML modeling

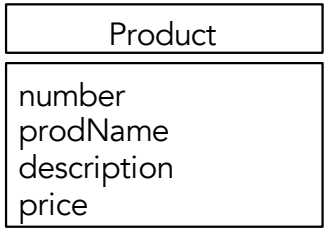
- Unlike object-oriented UML modeling, in database design
  - The classes have no methods
  - Key attributes need to be marked on the diagram
  - Those attributes, that can be retrieved through an association, and won't be written inside the class
  - Types of the attributes are atomic
  - We need to indicate the multiplicity of the association, unless it is *1..1*
  - Lines representing associations often don't have arrowheads

Class in UML

- A class in a UML diagram is roughly equivalent to a class in object-oriented programming – except here classes don't have methods
- Attributes describe the properties of the object

Example of a Class

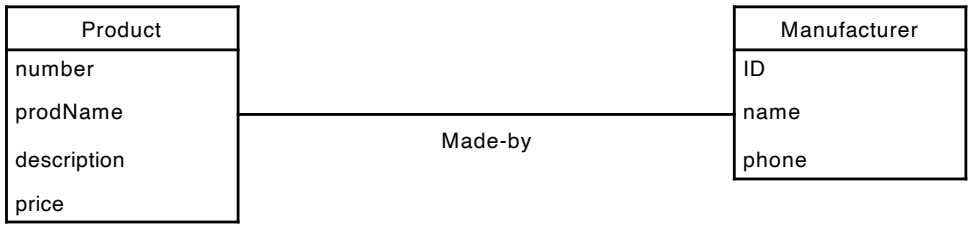
- The class *Product* describes products in an online store.



- For each product available in the store there exists one *Product*-object.
- The attributes of the class are listed under the name of the class. Types of the attributes need to be atomic. Each *Product*-object has its own values for attributes *number*, *prodName*, *description* and *price*.
- When modeling databases with UML diagrams, we don't include any possible actions (methods) for the objects of the class.

Associations

- Associations create a connection between two classes (in a UML diagram no more than two).
- For example between the classes *Product* and *Manufacturer* we might define an association *Made-by*. If a manufacturer *m* has manufactured the product *p*, the object *p* from the class *Product* will be connected to the object *m* of the class *Manufacturer* with association *Made-by*.
- The name of the association is usually written below the line representing the association.



Multiplicity of Association

- On both ends of the line representing the association, it's indicated, how many objects from one class can connect to some other class through an association.

- In this example, there can be 0 or 1 *Manufacturer*-objects for each *Product*-object.
- Each *Manufacturer*-object can connect to arbitrarily many *Product*-objects.
- **Pay attention to which ends the labels are on!**

Multiplicity of Association: Precise Notations

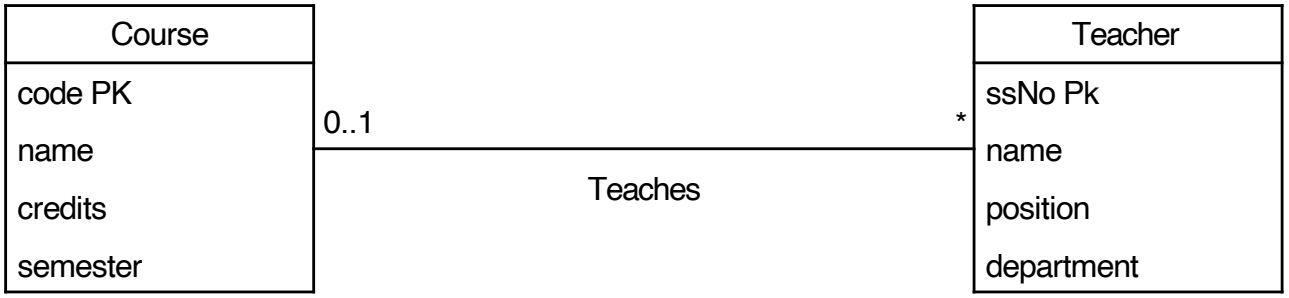
- In an association between classes *A* and *B*, above or below the line, the label m..n **at class *B*'s end** means, that for each object in the class *A* we have at least *m* and at most *n* objects of class *B* connected to it.
- Examples:
  - *0..1* at most 1, but possibly none
  - *0..5* at most 5, but possibly none
  - *1..2* at least 1, at most 2
  - *1..1* exactly 1 (can be labeled with *1*)
  - *1..** at least one 1, but otherwise arbitrarily many
  - *0..** arbitrary amount (can be labeled simply with *).
- When it comes to database modeling, a missing label is equivalent to 1..1.

Multiplicity of Association: Terminology

- *Many-many* association: each object from both classes can have arbitrarily many objects of the other class associated with it.
- *Many-one* association: each object from the first class has at most one object of the other class associated with it.
- *Many-exactly one* association: each object from the first class has exactly one object of the other class associated with it.

Exercise 1

- Which of the following claims hold for the UML diagram of teachers and courses below?



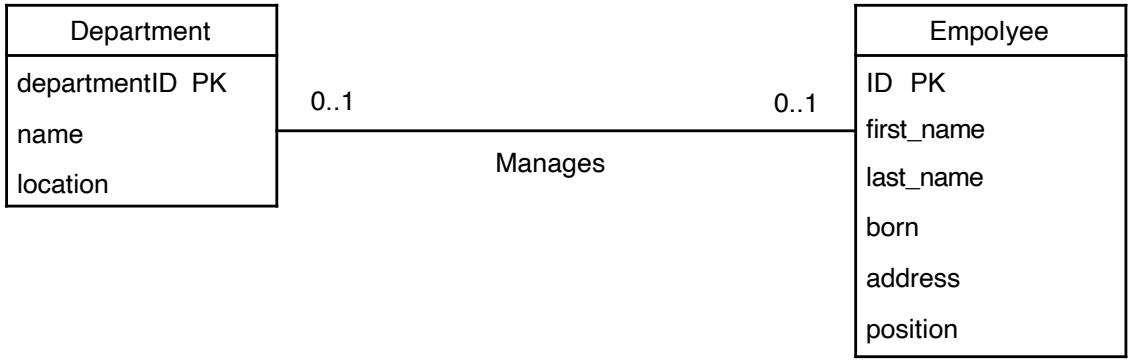
a. One teacher can teach arbitrarily many courses, and one course can have arbitrarily many teachers.
b. One teacher can teach at most one course, but one course can have arbitrarily many teachers.
c. One teacher can teach arbitrarily many courses, but one course can have at most one teacher.
d. One teacher can teach at most one course, and one course can have at most one teacher.

Answer

- In the diagram, at the end of the class *Teacher* we have a star. This means, that each *Course*-object can be associated with arbitrarily many *Teacher*-objects. In other words, each course can have arbitrarily many teachers.
- At the end of class *Course* we have the label 0..1. This means, that there can be 0 or 1 *Course*-objects for each *Teacher*-objects. In other words, each teacher can teach at most one course.
- Option b is correct.

Multiplicity of the association: more terminology

- If the association is many-one with respect to both classes, it's a *one-one* association.
- Example: in a company, each department has at most one manager, and each manager can lead at most one department.



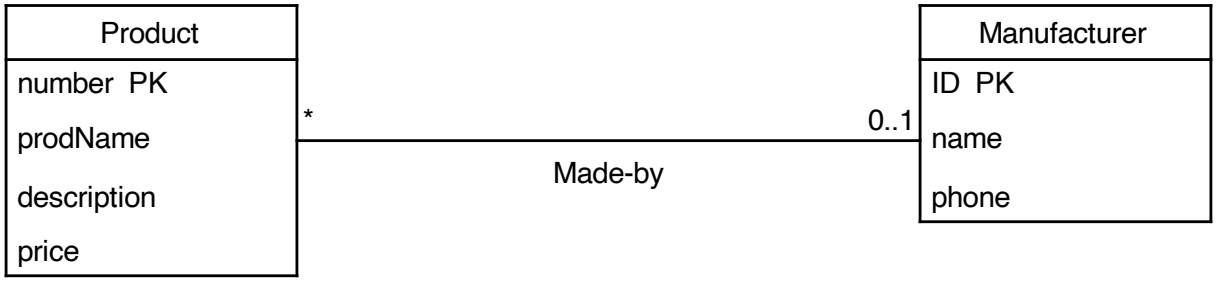
- One-one association is a special case of many-one association. Hence everything that holds for many-one associations, also holds for one-one associations.

Keys

- When designing object-oriented programs, there is no need for keys as in object-oriented programming each object has its own unique identity. However, in databases keys are essential to identify the rows (tuples) of the table. For this reason, the key attributes of the relation should be marked in the UML diagram.
- The key of the class *E* is such an attribute or a set of attributes, that no two distinct objects in class *E* share the values for the attribute, or the value combination for the set of attributes.
- Key attributes are marked with label PK.
- There can be multiple choices for a key, but only one is chosen and marked. If multiple attributes are marked with the PK-label, they together form the set of attributes that is the key of the class.

Example of Marking the Attributes

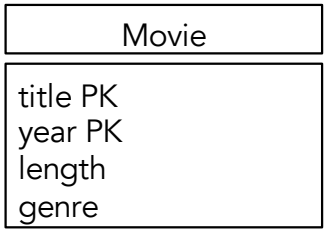
- A product is identified by its product number (attribute *number*), and a manufacturer by its ID (attribute *ID*).



Another Example of Keys

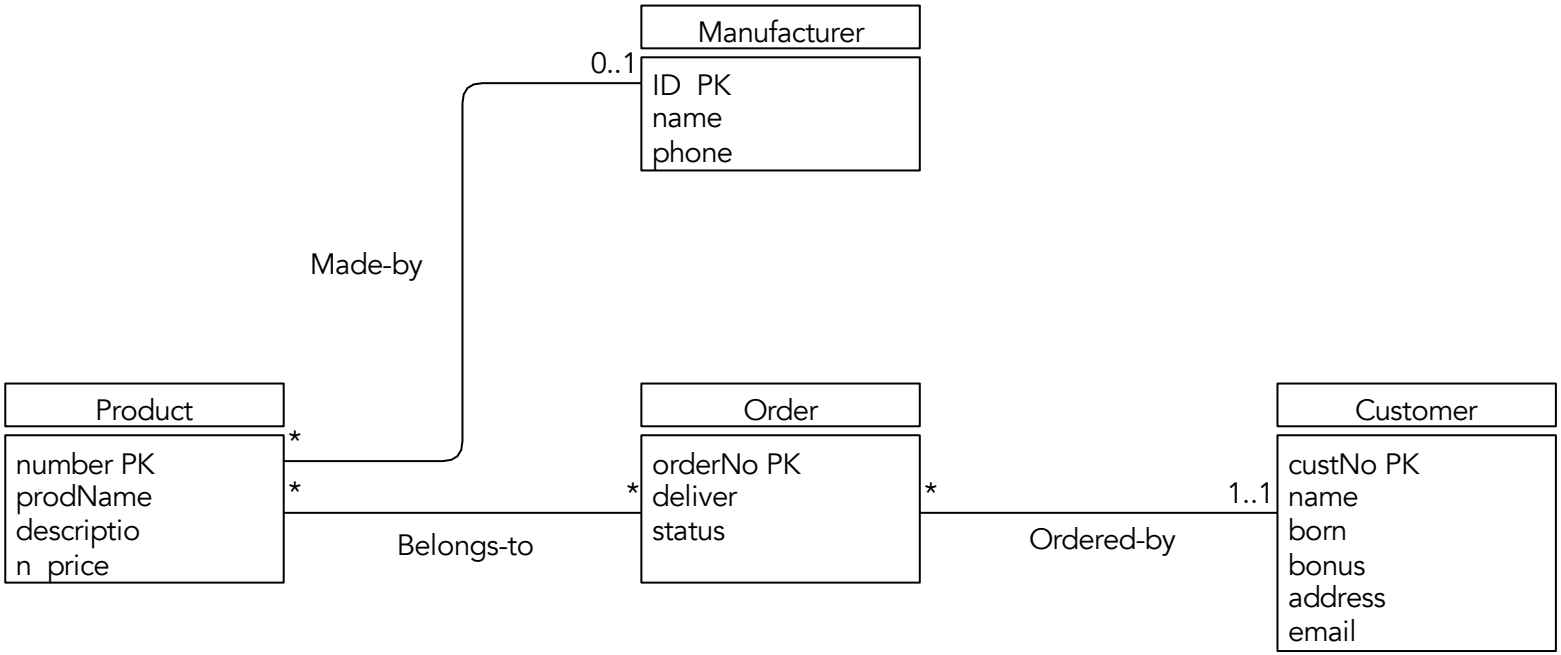
- In the textbook, the keys for the class *Movie* are the name of the movie and the year, together.

The idea is, that no film studio wants to produce a movie with the same name as another movie from a competing studio in the same year. However, it's possible that later someone wants to create a new version of the movie with the same name.



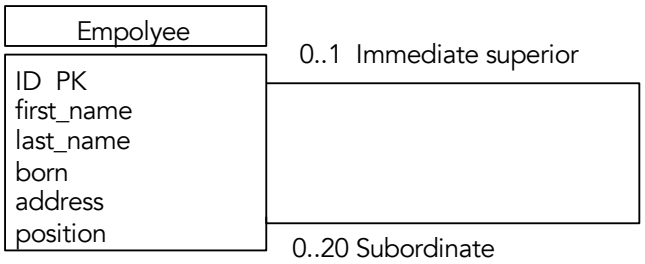
Example Diagram

- We have added the entities *Orders* and *Customers* and the associations *Ordered-by* and *Belongs-to* to our example database for the online store.
- This model allows us to add products with no manufacturer in the database. On the contrary we can't add orders with no customer.



Self-Association

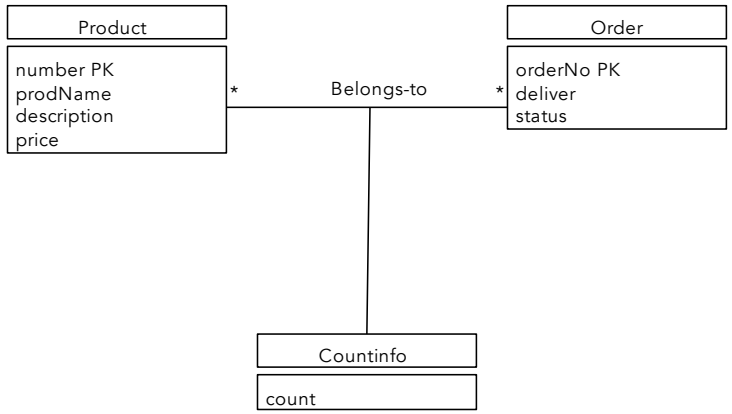
- *Problem:* How can we represent associations where the same class occurs twice?
- Example: let's define, that one employee can be an immediate superior for another employee.



- Both roles are written in the diagram. The multiplicity of the association is indicated at the end corresponding to the roles (here one employee can have at most one immediate superior, and 0–20 subordinates).

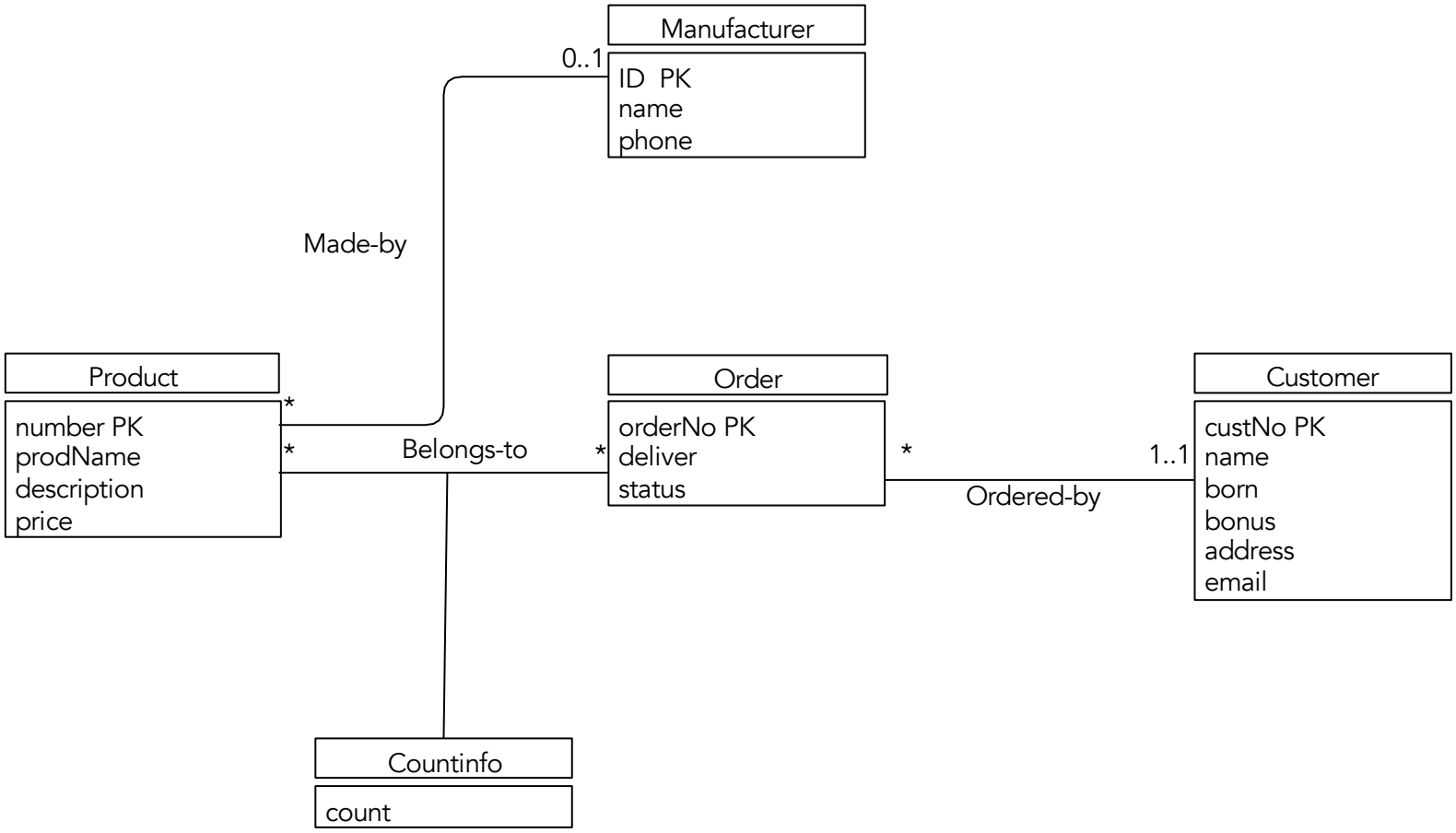
Association Class

- Example: in the web store example, let's specify that a customer can add many items of the same product to one order.
- Where should information about number of items be attached?
- Let's add to association a *Belongs-to* association class, and to it's attribute we'll add information about number of items.



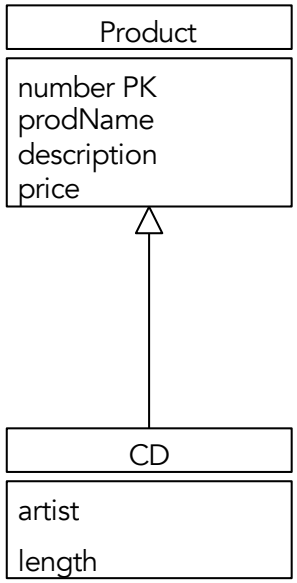
- An association class has no key attributes and multiplicity is never indicated on the line leading to the association class.

Web Store Example with Association Class



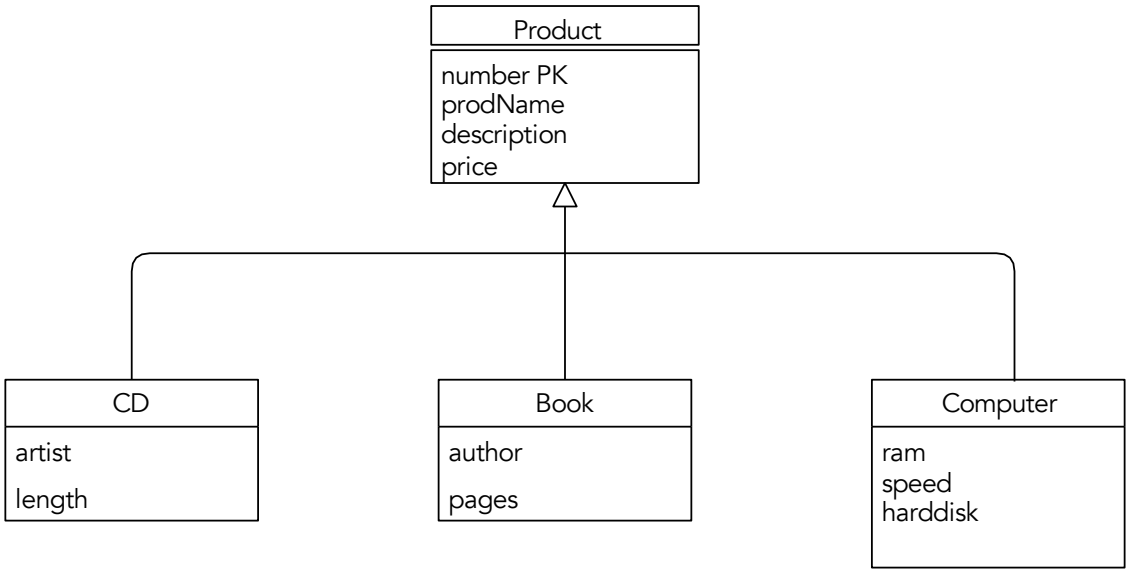
Inheritance and Subclasses

- Only some objects of the class have certain features. For these instances we can define a *subclass*.
- Object of subclass have all the attributes and associations of it's superclass and also the attributes and associations of the subclass.
- Relation from subclass to superclass is marked with a little triangle (pointing to the superclass)



Subclasses in UML-modeling: Example

- In this example the class *Product* has subclasses for different products: *CD*, *Book* and *Computer*.



- Subclasses may have their own associations that other classes of the inheritance hierarchy do not have. For example, class *CD* could have association to class *Track*, which describes one track of the album.

Inheritance and Keys

- Subclass never has it's own key attributes, but all the attributes needed for the key have to be attributes of the superclass.

When Attributes are not enough for the Key

- Let's assume that products of web store can have different versions, for example the same clothing can come in different colors and sizes.
- Let's define a class *Version* to describe different versions of product.
- Attributes of the class are size and color. Even together they are not enough to uniquely identify an object of *Version* (therefore they don't form a key).
- Solution: let's form a key of class *Version* by using attributes of class *Version* and the key attribute of class *Product*.
- The class *Version* is marked then with only it's own attributes.

When Attributes are not enough for the Key: Example

Aggregation

- With aggregation we can tell that an object of some class is formed by objects of another class, for example in the web store there are product groups, that are formed by products.
- Aggregation is marked by an open diamond at the end of that class whose object is formed by objects of another class.

Composition

- Composition is like aggregation, but is a stricter requirement. In composition the object of one class must belong to an object of another class.

UML Modeling

- How can one start UML modeling?
- Simple, but often a good working approach:
  1. Write a description of the database being modeled.
  2. Underline all the nouns.
  3. From nouns, find candidates for classes and attributes.
  4. Some nouns won't become either.
  5. When the classes and attributes are done, think, what kind of relations there might be between the objects of the classes. Make them the associations.

Example: Web Store

- Create a database for a web store that has products and customers.
- Customers can make orders which can include multiple products.
- Products have product number, name, description, price and manufacturer.
- Manufacturers have ID, name and phone number.
- Customers have customer ID, name, year of birth, bonus points, address and email address.
- Every order has a unique order number. Orders also have shipping method, state, products included in the order and customers who made the order.

Example continues: underline attributes

- Create a database for a web store that has products and customers. Customers can make orders which can include multiple products. Products have product number, name, description, price and manufacturer. Manufacturers have ID, name and phone number. Customers have customer ID, name, year of birth, bonus points, address and email address. Every order has unique order number. Orders also have shipping method, state, products included in the order and the customer who made the order.
- "Database" is a common term, which is not related to the modeled object. Therefore it won't become a class or attribute. "Web store" describes the whole system that we are modeling (the whole UML model), so it won't either become a class or attribute either.

Design Principles

What should be considered when making a UML model of a real-life system?

- Faithfulness
- Avoiding Redundancy
- Simplicity Counts
- Choosing the Right Relationships
- Picking the Right Kind of Element

Faithfulness

- Classes and their attributes need to correspond with the real world they are describing.
- For example, multiplicity of association needs to be decided by the fact that the same object in real-life can be associated with one or more objects of other class.
- In a web store the same order can include several products, but the order must have exactly one customer. This is shown in the UML diagram.

Avoiding Redundancy

- One entity should be created only once.
- For example, class *Product* shouldn't have attribute of manufacturer ID, because association *Made-by* already indicates the manufacturer
- Why redundancy is harmful?
  - Repeating same information takes up unnecessary space.
  - Repeating the information may cause problems with database updates.
- Object-oriented programming works differently because from the UML diagram you want to directly access what attributes the classes of the object program have. When designing databases, the situation is different because the classes are not representing directly tables coming into the database.

Striving for Simplicity

- Model should not include additional elements.
- For example, class should not be separated into two classes and an association between them without a good reason.

Figure 11: A poor design with an unnecessary entity set

Choosing the Right Relationships

- Entities can be connected in various ways to relationships.
- Adding to our design every possible relationship is not a good idea.
- Doing so can lead to redundancies and anomalies.

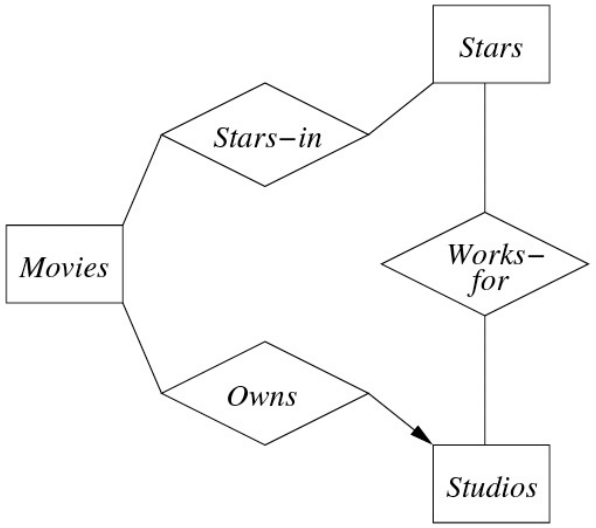


Figure 12: Adding a relationship between *Stars* and *Studios*

Using the Right Elements

- When should some element be described with a class or with an attribute?
- Let's examine the model below. Could class *Manufacturer* and association *Made-by* be replaced by adding additional attributes to the class *Products*?

Using the Right Elements, Continued

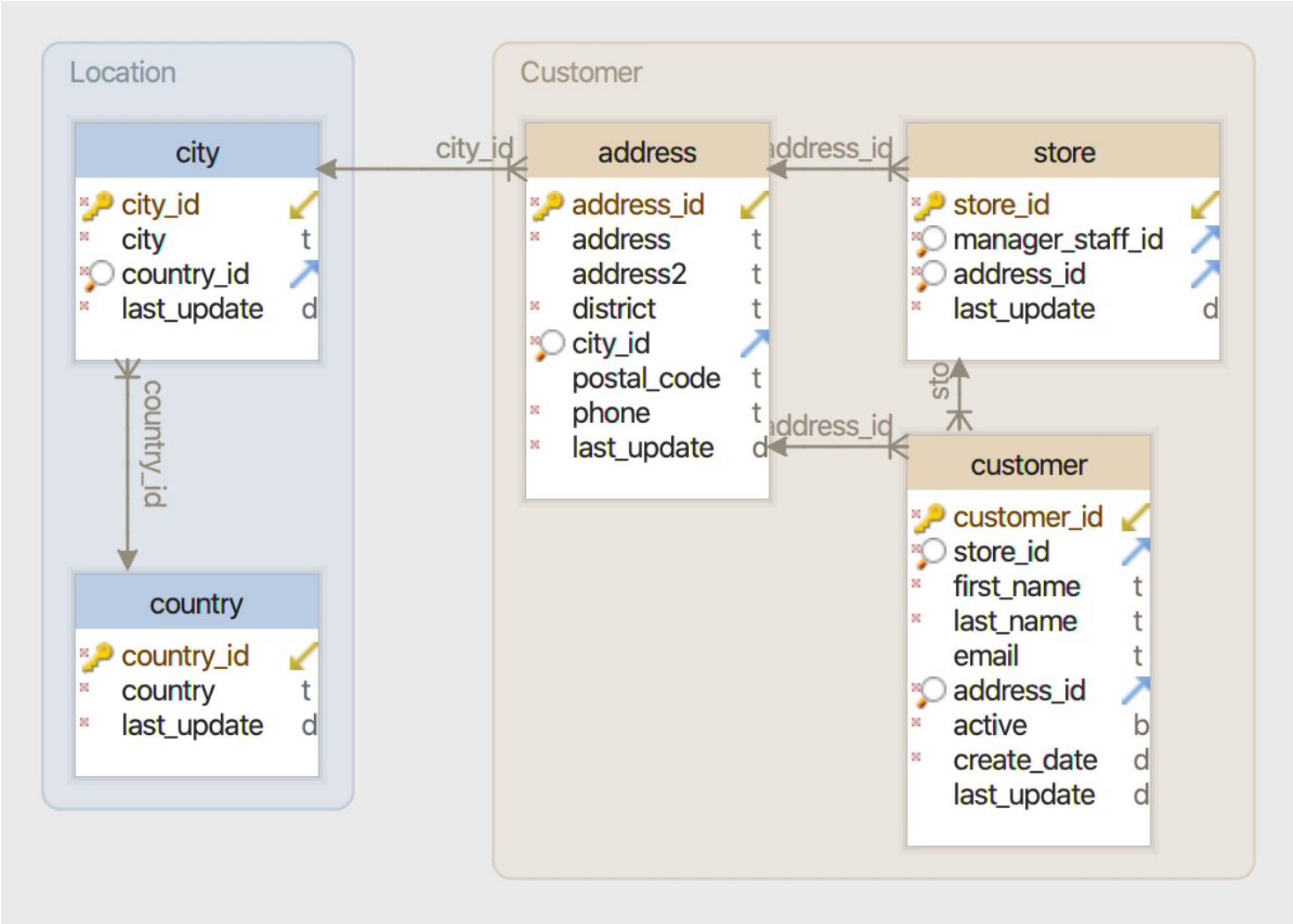
- In principle, the class *Products* could have attributes *manufacturerID*, *manufacturerName* and *phone*, which would replace the class *Manufacturer* and the association *Made-by*.
- But then the name and phone number of a certain manufacturer would be repeated in different products.
- *Common principle:* if something other than just name or numerical value of some real-world concept is being modeled, it should be modeled as class instead of attribute (however, class can be association class).
- When to use a regular class vs. an association class? Association class is good for situations where the information being described is related exactly to a pair formed by two objects and it does not have its own key.

Using the Right Elements, Continued

- In the web store example *Order* couldn't be an association class, but is has to be regular class, because:
  1. Order is not necessarily a relation between exactly one product and one customer, but one order can include several products.
  2. The class *Order* has it's own key attribute.

  Only one of these reasonings would be enough.

UML Modeling using *DbSchema*: Example



Functional Dependencies & Normal Forms: Part I

CS-A1153: Databases for Data Science

Prof. Barbara Keller

Aalto University
School of Science

Database – Theory

(aka the fun part)

Today: Functional Dependencies

Interactions Today:

Presemo:

http://presemo.aalto.fi/csa1153

Kahoot

https://kahoot.it

Intended Learning Outcomes

After this session you can:

- explain in your own words how redundancy can lead to anomalies
- find a (possible) functional dependency in a given table
- find closure (of a given set of attributes ;)).

Timeline

Quiz

socialSecurityNr → dateOfBirth

Functional Dependencies

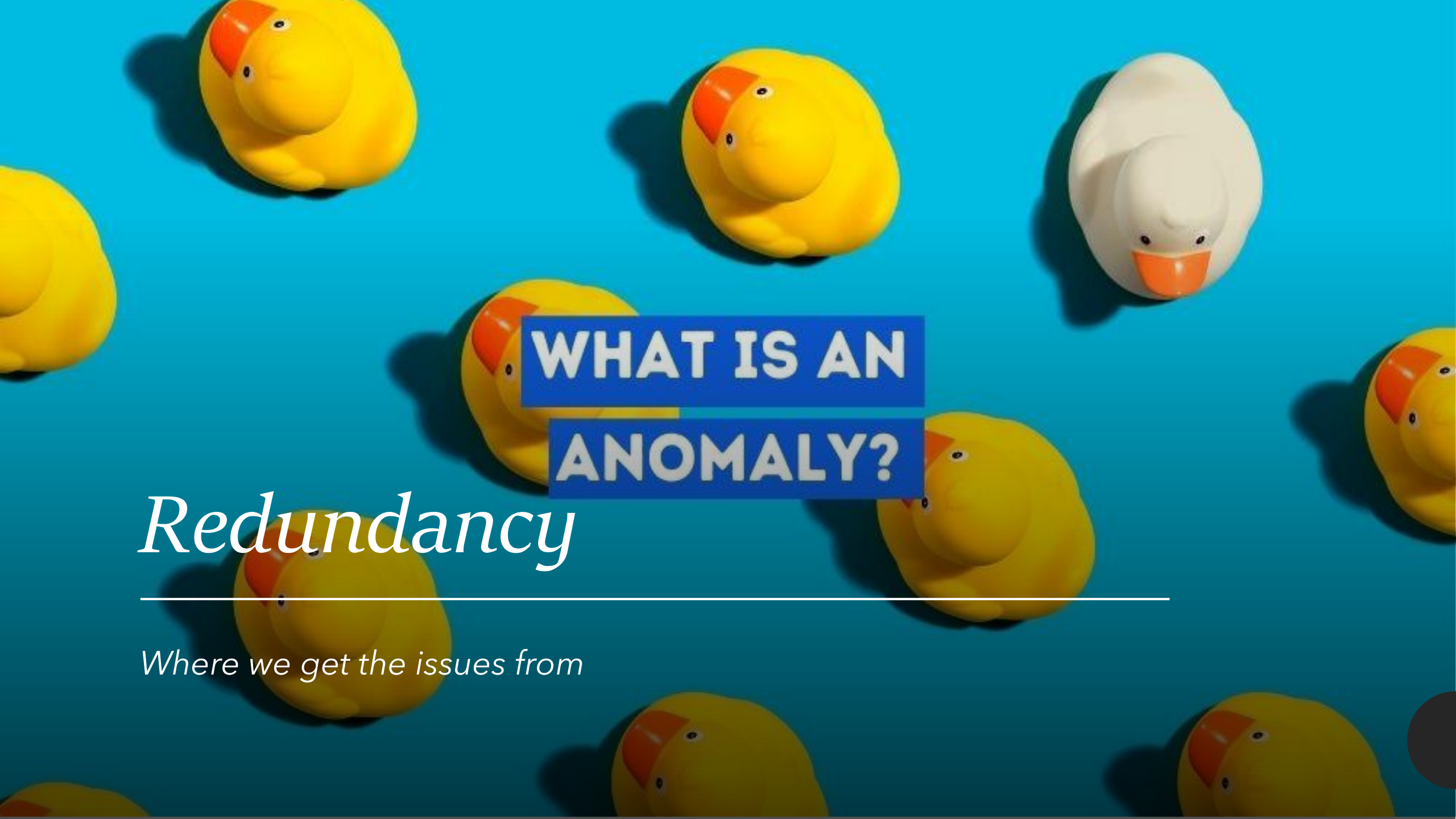
Anomalies

Functional Dependencies

To be able to design good database schemes

To be able to rate a database scheme

To learn how to formally talk about dependencies



Redundancy

Where we get the issues from

Why is Redundancy Bad (in Databases)?

- Space
- Update Anomaly
- Deletion Anomaly

Why is Redundancy Bad (in Databases)?

- **Space**
- Update Anomaly
- Deletion Anomaly

StudentCourse

student	field	course	credits	teacher	office
Anton	CS	Database	5	**Keller**	**B123**
Anton	CS	Programming	5	**Keller**	**B123**
Sandra	ELEC	**Network**	**5**	**Kivelä**	**A123**
Carl	DS	Database	5	**Keller**	**B123**
Carl	DS	**Network**	**5**	**Kivelä**	**A123**
Sandra	ELEC	Suomi	2	Järvinen	C123
Merlin	CS	Programming	5	**Keller**	**B123**
Harry	DS	Suomi	2	Järvinen	C123
Michael	CS	Business	5	Ungeheuer	D123

Why is Redundancy Bad (in Databases)?

- Space
- **Update Anomaly**
- Deletion Anomaly

StudentCourse

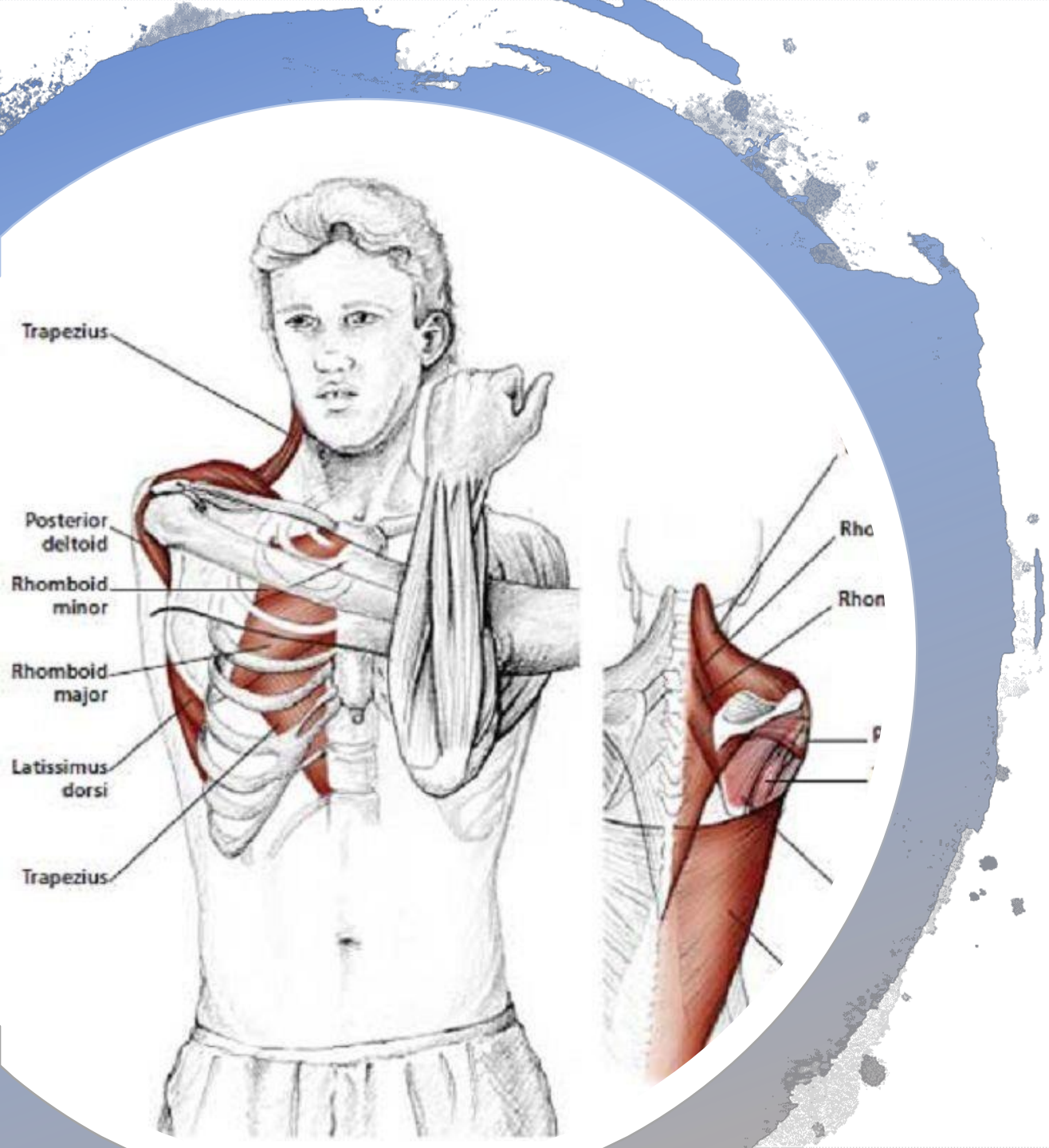
student	field	course	credits	teacher	office
Anton	CS	Database	5	Keller	B123
Anton	CS	Programming	5	Keller	B123
Sandra	ELEC	Network	5	Kivelä	A123
Carl	DS	Database	5	Keller	B123
Carl	DS	Network	5	Kivelä	A123
Sandra	ELEC	Suomi	**1**	Järvinen	C123
Merlin	CS	Programming	5	Keller	B123
Harry	DS	Suomi	2	Järvinen	C123
Michael	CS	Business	5	Ungeheuer	D123

Why is Redundancy Bad (in Databases)?

- Space
- Update Anomaly
- **Deletion Anomaly**

StudentCourse

student	field	course	credits	teacher	office
Anton	CS	Database	5	Keller	B123
Anton	CS	Programming	5	Keller	B123
Sandra	ELEC	Network	5	Kivelä	A123
Carl	DS	Database	5	Keller	B123
Carl	DS	Network	5	Kivelä	A123
Sandra	ELEC	Suomi	2	Järvinen	C123
Merlin	CS	Programming	5	Keller	B123
Harry	DS	Suomi	2	Järvinen	C123
~~Michael~~	~~CS~~	~~Business~~	~~5~~	~~Ungeheuer~~	~~D123~~



Break:
Move your Shoulders

If I give you the value in X – you know/can look up the entry in Y
Written as $X \rightarrow Y$
(Determinant → Dependent)

Functional Dependencies

student	field	course	year	credits	teacher	office
Anton	CS	Database	2021	5	Keller	B123
Anton	CS	Programming	2020	5	Keller	B123
Sandra	ELEC	Network	2020	5	Kivelä	A123
Carl	DS	Database	2020	5	Sawhney	A112
Carl	DS	Network	2021	5	Kivelä	A123
Sandra	ELEC	Suomi	2020	2	Järvinen	C123
Merlin	CS	Programming	2019	5	Pasi	B112
Harry	DS	Suomi	2020	2	Järvinen	C123
Michael	CS	Business	2020	5	Ungeheuer	D123

If I give you the value in X – you know/can look up the entry in Y
Written as $X \rightarrow Y$
(Determinant → Dependent)

Functional Dependencies – more abstract

A_1	B_1	C	D	E	F	G
Anton	CS	Database	2021	5	Keller	B123
Anton	CS	Programming	2020	5	Keller	B123
Sandra	ELEC	Network	2020	5	Kivelä	A123
Carl	DS	Database	2020	5	Sawhney	A112
Carl	DS	Network	2021	5	Kivelä	A123
Sandra	ELEC	Suomi	2020	2	Järvinen	C123
Merlin	CS	Programming	2019	5	Pasi	B112
Harry	DS	Suomi	2020	2	Järvinen	C123
Michael	CS	Business	2020	5	Ungeheuer	D123

$$A_1 \rightarrow B_1,\ C \rightarrow E,\ CD \rightarrow F,\ F \rightarrow G,\ G \rightarrow F$$

Keys

Functional Dependency is a generalization of Keys:

A set of one or more attributes of a relation is called a key of the relation if

1. *They functionally determine all other attributes of the same relation*
2. *the numbers or attributes is minimal (you cannot remove any attributes)*

A superkey is a set of attributes containing a key (from super set not  *)*
A relation can have more than one key

Splitting / Combining FDs and Trivial FDs

$$A \to B_1B_2 = A \to B_1, A \to B_2,$$
$$A_1A_2 \to B_1 \neq A_1 \to B_1\,, A_2 \to B_1$$

Trivial dependency: $A_1A_2A_3A_4 \to A_2A_4$
If all attributes on the right hand side are contained in the left hand side, the dependency is said to be trivial

Completely non-trivial dependency: $A_1A_2 \to B_1B_2$
If none of the attributes on the right ocurs on the left, the dependency is completely non trivial otherwise just non-trivial

Non-trivial dependency: $A_1A_2 \to A_1B_2$

If we have functional dependencies and given a set of attributes: What "can be concluded"?

$$R(A, B, C, D)$$
$$\boldsymbol{S} = \{A \rightarrow B, AC \rightarrow D\}$$
$$\{A, C\}^{+} = \{?\}$$

If we have functional dependencies and given a set of attributes: What "can be concluded"?

$$R(A, B, C, D)$$
$$\boldsymbol{S} = \{A \rightarrow B, AC \rightarrow D\}$$
$$\{A, C\}^{+} = \{?\}$$
$$\{A, C\}^{+} = \{A, C\}$$

If we have functional dependencies and given a set of attributes: What "can be concluded"?

$$R(A, B, C, D)$$
$$\boldsymbol{S} = \{A \rightarrow B, AC \rightarrow D\}$$
$$\{A, C\}^{+} = \{?\}$$
$$\{A, C\}^{+} = \{A, C, B\}$$

If we have functional dependencies and given a set of attributes: What "can be concluded"?

$$R(A, B, C, D)$$
$$\boldsymbol{S} = \{A \rightarrow B, AC \rightarrow D\}$$
$$\{A, C\}^{+} = \{?\}$$
$$\{A, C\}^{+} = \{A, C, B, D\}$$

If we have functional dependencies and given a set of attributes: What "can be concluded"?

$$R(A, B, C, D)$$
$$\boldsymbol{S} = \{A \rightarrow B, AC \rightarrow D\}$$
$$\{A, C\}^{+} = \{?\}$$
$$\{A, C\}^{+} = \{A, C, B, D\}$$

$\{A, C\} =$ superkey

If we have functional dependencies and given a set of attributes: What "can be concluded"?

$$R(A, B, C, D)$$
$$\boldsymbol{S} = \{A \rightarrow B, AC \rightarrow D\}$$
$$\{A, C\}^{+} = \{?\}$$
$$\{A, C\}^{+} = \{A, C, B, D\}$$

$\{A, C\} =$ superkey

$\{A\}/\{C\}$ a key?

Closure Algorithm

Input: set of Attributes $\mathcal{A} = \{A_1, A_2, \dots, A_n\}$ and a set $\mathcal{S}$ of FDs
Output: closure $\mathcal{A}^+$

1. If necessary, split the FDs so that there is a single Attribute to the right
2. Initialize $\mathcal{X} := \mathcal{A}$

---

3. Check in $\mathcal{S}$ for FD with $B_1 B_2 \dots B_n \rightarrow C$ so that all $B_k \in \mathcal{X}$ and $C \notin \mathcal{X}$
4. If found: $\mathcal{X} := \mathcal{X} \cup \{C\}$ repeat step 3
   Else: $\mathcal{A}^+ = \mathcal{X}$

Time for Kahoot

Defining SQL Tables, Integrity Constraints & Views

CS-A1153: Databases for Data Science

Dr. Sami El-Mahgary

Aalto University
School of Science

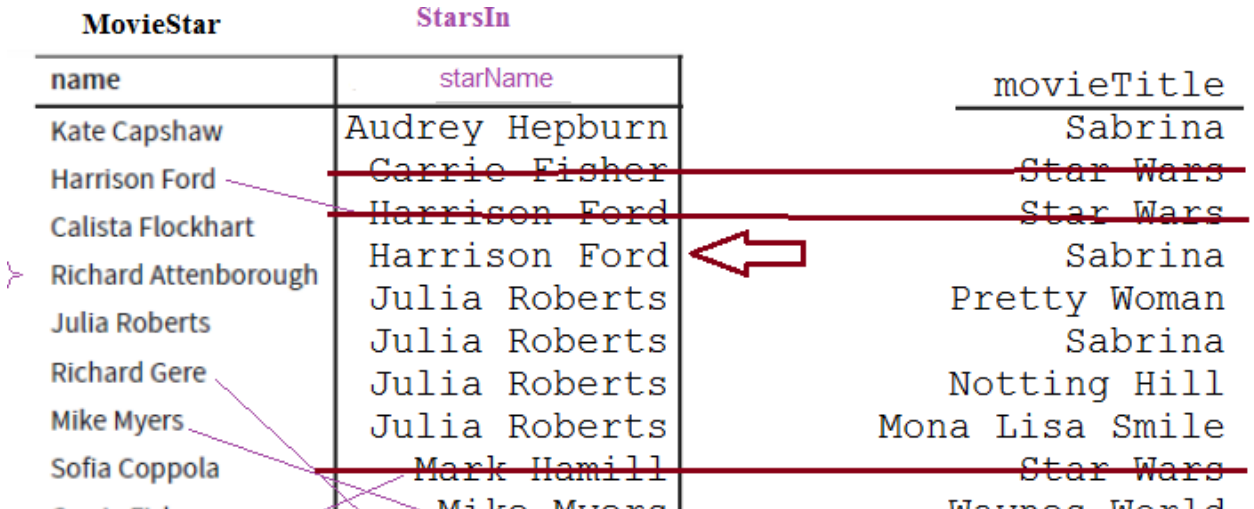
SQL Part II : Creating tables with constraints

SQL Part II: Introducing Negation: **Trying** to get actors who did not star in 'Star Wars' (N1/3)

```
1 SELECT  Distinct Name  FROM MovieStar
2 LEFT OUTER JOIN  StarsIn ON  name = starName
3          WHERE movieTitle NOT IN ('Star Wars')
```

The above query is not negation. It simply suppresses

displaying a row where the movie title is 'Star Wars': wrongly shows Harrison Ford (though he appeared in Star Wars), by displaying a row related to Harrison Ford starring in another movie.



SQL Part II: Introducing Negation (N2/3) Find movie stars who did *not* star in 'Star Wars'.

```
1  SELECT   Name  FROM MovieStar
2  EXCEPT
3  SELECT   Name  FROM MovieStar INNER JOIN  StarsIn
4  ON name = starName WHERE movieTitle  IN ('Star Wars')
```

A negation requires removing from set S_1 (all actors) the rows who belong to the given criteria of another set S_2 (actors who starred in 'Star Wars').

- This query works as it subtracts the actors who did star in 'Star Wars' from the set of all actors.
- Note how the INNER JOIN keyword can also be used for an inner equi-join.
- Remember the symmetry (also required when using INTERSECT or UNION). No **DISTINCT** needed $\rightarrow$ faster.

SQL Part II : **EXISTS** and Negation (N3/3) Another way to find actors who did *not* star in 'Star Wars'.

```
1  SELECT   Name  FROM MovieStar V
2   WHERE  NOT EXISTS
3      (SELECT 1 FROM  StarsIn S WHERE
4       V.name = S.starname
5      AND S.Movietitle = 'Star Wars');
```

- An EXISTS query return True if its subquery returns at least one row. Otherwise, a False is returned.
- Note the use of the **correlated** subquery: `starName` in inner query must match `name` in the outer query. (For large tables this might be slow).
- The use of 'SELECT 1' is for speed, since we only care whether the subquery returns any rows or not.
- Removing AND S.Movietitle = 'Star Wars' $\rightarrow$ actors who didn't star in any movie.

The Movies database schema and the four tables:

- **Movies**(*title*, *yearReleased*, *lengthMin*, *genre*, *studioName*, *prodno*)
- **MovieStar**(*name*, *address*, *gender*)
- **StarsIn**(*movieTitle*, *movieYear*, *starName*)
- **MovieExec**(*certno*, *name*, *address*)

Slight modifications in naming

- Year $\rightarrow$ YearReleased
- Length $\rightarrow$ LengthMin

Columns 'Year' and 'Length' have been renamed to avoid any error with reserved words.

SQL II (Schema)

The MovieStar table. In the book:

```
1 CREATE TABLE MovieStar (name CHAR(30) PRIMARY KEY,
2         address VARCHAR(255), gender CHAR(1),
3         DateOfBirth DATE);
```

We're using:

```
1 CREATE TABLE MovieStar (name VARCHAR(30) PRIMARY KEY,
2         address TEXT, gender CHAR(1),
3         DateofBirth DATE);
```

- For better performance, use VARCHAR over CHAR unless strings are known to be of same fixed size (e.g. postal code).
- TEXT is for columns with long strings that are typically not used as query criteria. References in a query to a TEXT-type column may be a bit slower than VARCHAR(x).

SQL II Table creating basics (Schema)

- Recap: for table StarsIn, all three attributes are needed for a composite (unique) key. The StarsIn table in the book:

```
CREATE TABLE StarsIn (movieTitle VARCHAR(100) ,
       movieYear INTEGER, starName VARCHAR(100),
       PRIMARY KEY (movieTitle, movieYear, starName));
```

```
CREATE TABLE Movies (Title VARCHAR(100) ,
       yearReleased INTEGER,  lengthMin INTEGER
       genre VARCHAR(10), studioName VARCHAR(30),
       prodno INTEGER,
       PRIMARY KEY (title, year));
```

- For Movies, the primary key is also composite, define at the end:

SQL II Table creating basics (0/7) (MovieExec)

- MovieExec has a single column `certno` as a primary key, it is a surrogate key (artificial).

```
1 CREATE TABLE MovieExec (certno INTEGER PRIMARY KEY, name TEXT,
2 address TEXT);
```

- A producer listed in (child) table Movies must also be listed in (parent) table MovieExec.
- So the foreign key `prodno` (Movies) gets joined to the primary key `certno` (MovieExec).
- How to make sure that the values used in table Movies maintain **referential integrity** so that each value for `prodno` also appears in table MovieExec under column `certno` ?

SQL From Constraints to Assertions (1A/7).

```
1  ...prodno INTEGER,
2  FOREIGN KEY (prodno) REFERENCES MovieExec (certno)...
```

The above referential-integrity constraint guarantees that

for each non-null value of `prodno` in child table (Movies) there is a corresponding row in the primary key of parent table MovieExec. By default, SQL rejects the following operations:

- (1) Inserting a new row into a child table (Movies) with a foreign key `prodno` that does not match the referenced primary key `certno`.
- (2) Updating a parent's table's (Movies) primary key with a value that no longer matches the foreign key in the child table. (3) Or vice versa!
- (4) Deleting a row from the parent table (MovieExec) when the child table (Movies) references that same primary key.

SQL From Constraints to Assertions (1B/7).

```
1  PRAGMA foreign_keys=ON;
2  CREATE TABLE MovieExec (certno INTEGER PRIMARY KEY,
3  name TEXT,address TEXT);
4
5  CREATE TABLE Movies (title VARCHAR(100), yearReleased INTEGER,
6  lengthMin INTEGER, genre VARCHAR(10), studioName  VARCHAR(30),
7  prodno INTEGER, FOREIGN KEY (prodno) REFERENCES
8  MovieExec (certno), PRIMARY KEY (title, yearReleased));
```

- The PRAGMA is an SQLite-specific used to modify default behaviour: *foreign_keys=ON* makes sure foreign key constraints will not be ignored.
- Table MovieExec is created before table Movies since it acts as a foreign key reference.
- In the parent table MovieExec, `certno` in has to be a primary key (or at least declared as unique).

SQL II From Constraints to Assertions: reducing foreign key constraint failures (1C/7).

```
1 FOREIGN KEY (prodno) REFERENCES MovieExec (certno)
2 ON DELETE SET NULL ON UPDATE CASCADE
```

Instead of having SQL gives an error violation, With ON DELETE SET NULL and UPDATE CASCADE keywords, SQL can **allow** rows with a foreign key constraint to be deleted/updated by automatically maintaining referential integrity:

- When deleting a key from the parent table MovieExec, the corresponding key `prodno` in child table Movies will be set to null, preventing a dangling tuple.
- with UPDATE CASCADE, a change in the primary key of the parent table is propagated to `prodno` in child table (Movies), the foreign key that references it.

SQL II From Constraints to Assertions: use of ON UPDATE CASCADE (1D/7).

```
1 UPDATE MovieExec SET certno = 209 WHERE certno = 290
```

- In the above, a fix using the UPDATE keyword is done for a particular key: `certno` 290 $\rightarrow$ 209
- with ON UPDATE CASCADE, (1) an update on the primary key `certno` is now **allowed** (2) and the value(s) for foreign key `prodno` in the child table will be changed accordingly from 290 to 209.
- Since a primary key's values are not often modified, there is often less use for ON UPDATE CASCADE. Due to this, Oracle RDBMS doesn't support UPDATE CASCADE.
- An example might be using a barcode as a primary key and switching from 12-digit to a 13-digit barcode.

SQL II From Constraints to Assertions: Preventing dangling references (1E/7).

```
1  DELETE FROM MovieExec WHERE certno = 123
```

- Since we're using **ON DELETE SET NULL**, executive George Lucas (`Prodno`= 123) can be deleted from MovieExec.
- The foreign key `Prodno`= 123 in child-table Movies gets set to Null to avoid having a dangling reference affecting the inner-joins (and prevent data corruption).

	title	yearReleased	lengthMin	genre	studioName	prodno
1	Gone with the Wind	1939	231	drama	Warner	456
2	Star Wars	1977	124	SciFi	Fox	123
3	Waynes World	1992	95	comedy	Paramount	965
4	Pretty Woman	1990	119	comedy	Disney	834
5	Mary Poppins	1964	139	musical		223

123 is set to Null.

(MovieExec)

	certno	name	address		
					Disney 218
1	114	Sydney Pollack	Pacific Palisades		Paramount 587
2	123	~~George Lucas~~	~~85 Palm St., San Fransisco~~	e	Paramount 585
3	146	Jean-Jacques Annaud	Paris, France		Paramount 378

SQL II From **Constraints** to Assertions (2/7). Attribute constraints: any single column

```
1 CHECK (logical_check_for_single_column)
```

An attribute-based CHECK constraint enforces the data integrity for a given **single column** associated with a single table. Use it to specify:

- an allowable numerical/date range
- allowable values for a categorical variable such as gender or student status ('absent','attending','exchange')
- an attribute-based constraint can be written as part of the column definition when creating the table.

SQL II From **Constraints** to Assertions (3A/7).

```
1  ...,yearReleased INTEGER,
2  lengthMin INTEGER CHECK (lengthMin BETWEEN 1 AND 999),
3  genre VARCHAR(10)...
```

The attribute-based check constraint above requires

`lengthMin` to be within the range [1..999]

- The above column constraint fires whenever a **new value** gets assigned to the specified column in any row of the specified table.
- Constraints are always checked before a change is attempted: inserting or updating data (no check when deleting).
- Attribute-based constraints are best kept simple: use a logical expression referring to a single column and avoid SQL queries.

SQL II From Constraints to Assertions (3B/7): must prevent **Nulls** explicitly

```
1  ...,yearReleased INTEGER,
2  lengthMin INTEGER NOT NULL CHECK (lengthMin BETWEEN 1 AND 999),
3  genre VARCHAR(10)...
```

Recall that a Null cannot be compared to a value

(the result is Unknown)

- The range [1..999] as such does **not** prevent Nulls.
- Nulls are prevented by a separate **NOT NULL** constraint.
- Careful when setting a foreign key to not Null: will prevent the use of the **ON DELETE SET NULL** option.

SQL II From **Constraints** to Assertions: Catching typos (3C/7).

```
1 ....
2 gender CHAR(1) CHECK (gender IN ('F','M','O')),
3 ...
```

- The above attribute constraint prevents inputting any value besides 'F', 'M' or 'O' for the gender column.

```
1 INSERT INTO MovieStar VALUES ('Russell Crowe',
2 'Coffs Harbour, NSW 2450, Australia','MM', '1964-04-07');
```

- The above insert would fail, catching the typing error for the gender. A check constraint is satisfied only when the specified condition is True or evaluates to *Unknown* due to a NULL value.

SQL II From **Constraints** to Assertions (3D/7).

```
....
gender CHAR(1) CHECK (gender IN ('F','M','O')),
...
```

```
INSERT INTO MovieStar VALUES (
'J.J. Smyth', 'c/o Eire AB Studios, Cork T23, Ireland',
null,'1997-12-05');
```

- The above insert bypasses the constraint due to the **Null** and is thus allowed.
- What if there are prior rows in the table for which the column in the CHECK constraint is not met? $\rightarrow$ old rows are checked for validity when a new check constraint is applied.

SQL II From **Constraints** to Assertions (4A/7): tuple-based checks: can use multiple columns

```
1  CREATE TABLE Movies (Title VARCHAR(100) ,
2         yearReleased INTEGER,  lengthMin INTEGER
3         genre VARCHAR(10), studioName VARCHAR(30),
4         producer INTEGER,
5         PRIMARY KEY (title, year)),
6         CHECK (lengthMin <= 180 OR genre IN ('drama','war'));
```

A tuple-based check constraint (above) enforces data integrity at row level and can refer to any column in the table.

- Unlike the attribute-based CHECK, the tuple-based CHECK is written *separately* after the data type declarations.
- The above check constraint for table Movies fires whenever a new row gets inserted or an existing gets updated.
- The constraint: only war and drama movies can exceed 3 hrs.

SQL II From **Constraints** to Assertions (4B/7): **tuple-based** checks on an existing table

```
1 ALTER TABLE Movies ADD CONSTRAINT
2 ConstMovieLength CHECK (
3 lengthMin <= 180 OR genre IN ('drama','war'));
```

If the table already exists, the constraint is added by modifying the table structure using the keyword ALTER.

- Note that when the constraint is added later through the ALTER keyword, it has to be given a name (here *ConstMovieLength*).
- This is so that it may be later revoked using DROP `constraint_name`;

SQL II From Constraints to Assertions: **Deferred constraints** (5/7).

Enforcing a constraint in a large DB may slow down transactions too much

Solution: allow constraints to be **temporarily** violated: use the **DEFERRABLE** constraint, comes with two options:

- **INITIALLY IMMEDIATE**: check at statement level $\rightarrow$ check constraints after each statement (same as without DEFERRABLE option).
- **INITIALLY DEFERRED**: check at transaction level $\rightarrow$ check constraints after transaction commits.
- with **DEFERRABLE INITIALLY IMMEDIATE** constraints can be deferred as needed: on many RDBMSs, done via SET CONSTRAINTS ConstraintName = DEFERRED.
- With NOT DEFERRABLE checks can never be deferred until commit time.

SQL II From Constraints to **Assertions** (6A/7).

```
1 CREATE ASSERTION CheckMoviesLenForAll CHECK (
2 NOT EXISTS (SELECT * FROM Movies WHERE  genre = 'thriller'
3 GROUP BY genre HAVING COUNT(title) > 3))
```

An assertion (above) enforces data integrity at global (schema) level: an SQL stmt with several tables/columns touches many rows.

- The GROUP BY-clause groups the films according to their genres, computes the no of titles (even entries where movie title is same but yearReleased differs).
- The subquery returns True to the outer NOT EXISTS predicate only if 4 or more war-related movies are found.
- The assertion: a max. of 3 thriller-related movies allowed.

SQL II From Constraints to Assertions (6B/7).

Summing it up...

```
1 CREATE TABLE MovieStar (name CHAR(30) PRIMARY KEY,
2  address VARCHAR(255) DEFAULT 'N.A.',
3  gender CHAR(1),
4  DateofBirth DATE);
```

- Nulls can be problematic in data-analysis → Use NOT NULL when it makes sense.
- If it is known that data for a particular column is very hard to come by, consider marking it by default with a short string (see above).
- The advantage of having 'N.A' over NULL is that the former indicates that the value has not been omitted by mistake.
- Pay special attention to maintaining referential integrity for foreign keys.

SQL II From Constraints to Assertions (6C/7).

...Summing it up cont.

- attribute-based checks are very handy when kept simple.
- tuple-based checks are more flexible: can refer to any column in table. But also checked on INS/UPDATE/ op., so keep them simple (might slow down performance)
- An assertion should be checked after every INS/UPDATE/DEL op. But how does the RDBMS figure this out? $\rightarrow$ In practise very few RDBMSs support assertions, and instead support **triggers**.
- A primary key is unique (no nulls allowed). So when to use UNIQUE as in: ...,name **TEXT UNIQUE**...?
- UNIQUE guarantees uniqueness in a column and creates an additional index $\rightarrow$ great for speeding queries on the column.
- Does UNIQUE allow multiples NUlls? The ANSI SQL standard says yes. But not all RDBMS allow multiple nulls for a UNIQUE constraint.

SQL Part II (Views) 7/7

```
1 CREATE VIEW ActiveMovieStars
2 AS SELECT name, address, gender FROM MovieStar
3 WHERE name IN (SELECT StarName FROM StarsIN);
```

- When a complex query is used frequently, we can store it as a view: the above defines those actors who actually appeared in a movie.
- A view is a virtual table: only the query is stored separately.
- The data is generated from the underlying tables **on-the-fly** each time it is accessed.
- Query a view just like you query a table: **SELECT * FROM** ActiveMovieStars. Very handy with grouped data.
- Some RDBMSs (not SQLite) support updating data through VIEWS just like updating a table.

SQL Part II

End of Part II