Data organization and queries

2IAB1

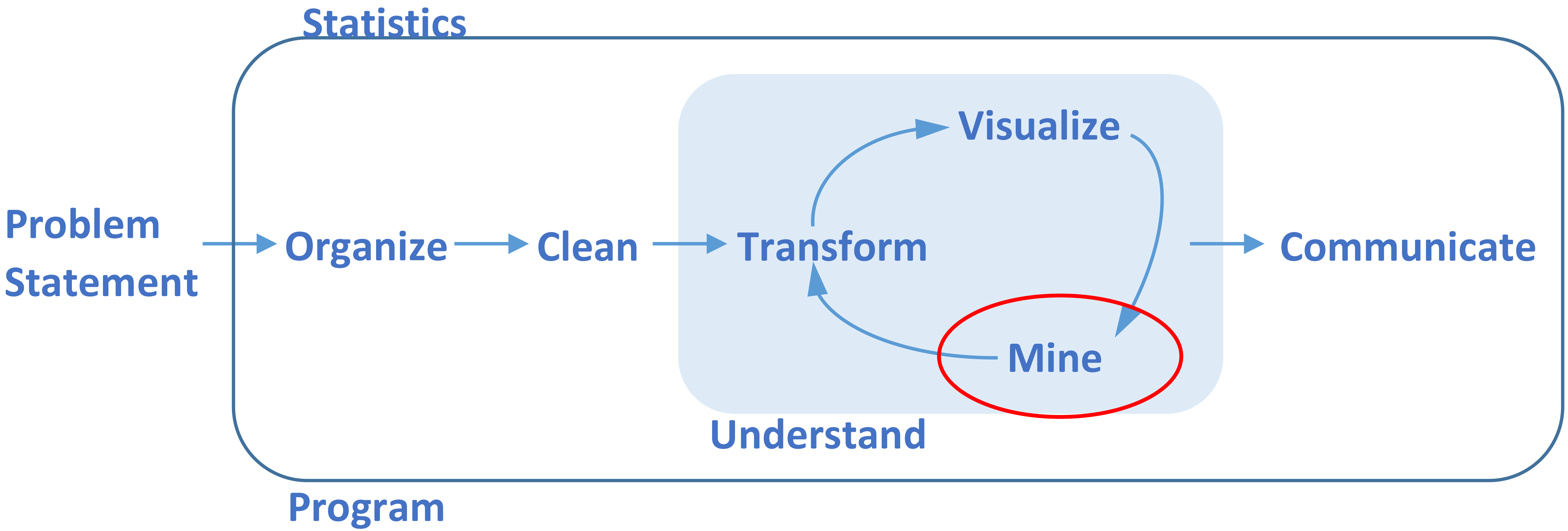
Week 4

Foundations of Data Analytics

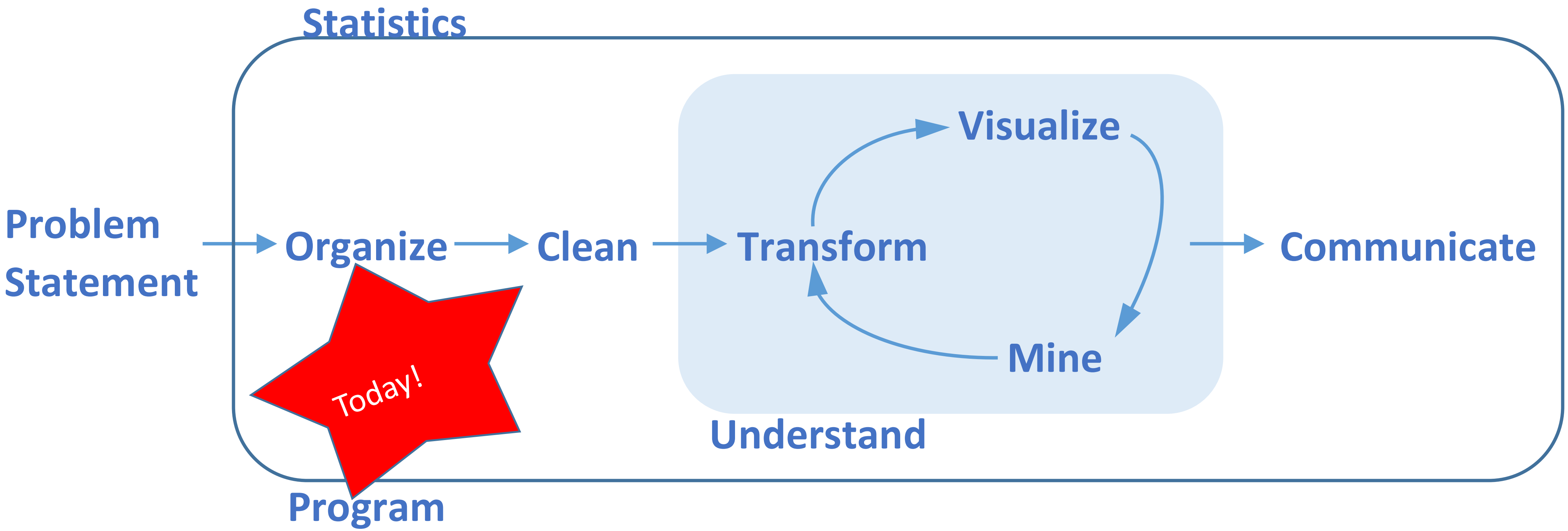
Academic year 2023-2024

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

What have we seen last week?



What will you learn this week?



Data everywhere! How to store? How to get?

Data organisation and queries

Data needs to be **stored and retrieved** in many types of applications:

- **scientific applications**: biology, chemistry, physics, social network analytics, …
- **technical/engineering applications:** automotive controls, embedded systems, air traffic control, climate control, power stations and grids, …
- **administrative applications:** banking, student administration, retail, manufacturing, logistics, human resources, …
- **document-oriented applications:** news sites, (digital) libraries, websites, search engines,…

Learning goals

1. Conceptual and Logical Data modelling:

capturing the conceptual model of a domain and implementing its logical structure in the relational database model

- understand the basics of conceptual data modeling in the Entity Relationship model
- understand the basics of the relational database model
- be able to translate a conceptual model into a relational database schema

2. Data retrieval (queries):

given a database, how to retrieve data of interest

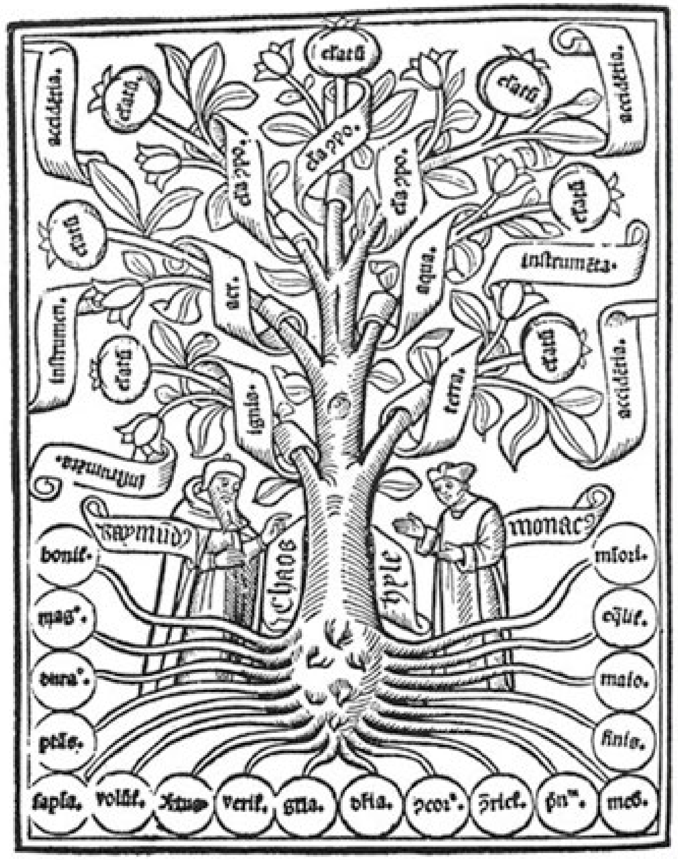
- be able to create simple SQL queries for retrieving relevant data, potentially spanning multiple database tables and involving grouping and aggregation of data

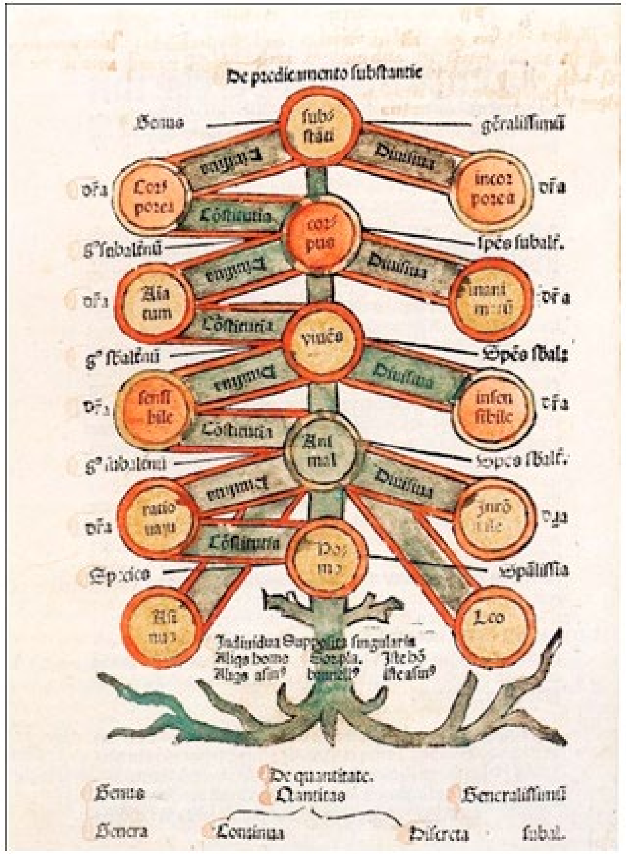
Using contemporary tools to support explainable, repeatable, durable, portable, efficient, and scalable data analytics

Object System vs. Information System

- ***object system***:
  the "real world" of a company, organization, or experiment, with people, machines, products, warehouses, chemical reactions, social relationships, …
- ***information system***:
  a *representation* of the real world in a computer system, using data (e.g., numbers) to *represent objects* such as people, machines, products, …
- ***example***: students are *people* in the real world, but they are *represented* by an identifying student number, name, address, list of enrolled courses, grades, etc. in the student administration database
- **the representation is always an *approximation***
  - e.g., your knowledge of a university course is represented by an integer number between 0 and 10

Modelling of Information





Ramon Llull’s “tree of knowledge” (1295)

“All models are wrong, but some are useful.”

–George Box, 1987

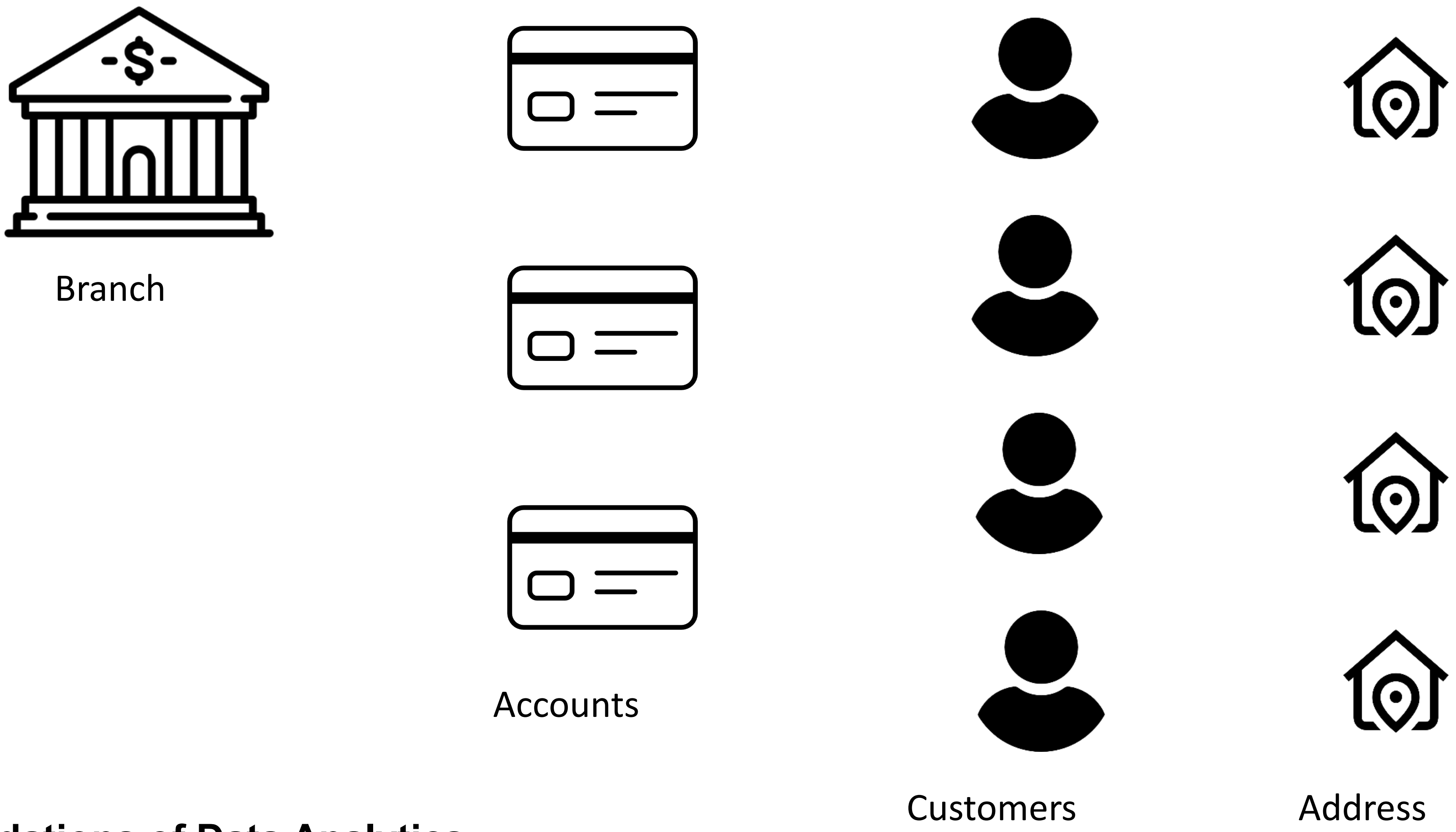
Relational Databases

Why do we use database systems?

What is wrong with storing all my data in one table?

- **Problems**
  - Duplication of information
  - Difficulty in keeping information consistent
  - Difficulty in accessing and sharing data
  - Hard or impossible to keep the data safe and secure
  - Hard or impossible to express (and efficiently execute) interesting (i.e., high-value) analytics over the data

Running example: bank



Data redundancy

- **Imagine** we store account information in **one single table** in a straightforward way.
- John Doe has several accounts → several rows in the table
- His address is the same in all rows → **redundancy!**
- What about joint accounts?

acct Number	bName	custName	custStreet	custCity	balance
001	Eindhoven Centrum	**John Doe**	**Kruisstraat**	**Eindhoven**	145.00
…	…	…	…	…	…
047	The Hague Centrum	**John Doe**	**Kruisstraat**	**Eindhoven**	100.00
047	The Hague Centrum	**Lotje Doe**	**Lindelaan**	**The Hague**	100.00
…	…	…	…	…	…

Data inconsistency

- John moves to Boschdijk and the record is corrected at the Eindhoven Centrum branch → **inconsistency!**

The address needs to be updated in all records of John!

acct Number	bName	custName	custStreet	custCity	balance
001	Eindhoven Centrum	**John Doe**	**Boschdijk**	**Eindhoven**	145.00
…	…	…	…	…	…
047	The Hague Centrum	**John Doe**	**Kruisstraat**	**Eindhoven**	100.00
047	The Hague Centrum	Lotje Doe	Lindelaan	The Hague	100.00
…	…	…	…	…	…

Structuring data to solve problems

Organize your data!
- Several tables:
  - a table with records about the customers
  - a table with records about accounts
  - a table with records of account ownership

Account

acctNumer	bName	balance
001	Eindhoven Centrum	145
…	…	…
046	Eindhoven Centrum	200
047	The Hague Centrum	100
…	…	…

Customer

custName	custStreet	custCity
John Doe	Boschdijk	Eindhoven
Lotje Doe	Lindelaan	The Hague
…		

Depositor

custName	acctNumber
John Doe	001
John Doe	047
Lotje Doe	047
…	…

Problems solved

- Redundancy and inconsistency are avoided
  - One record with John's address
  - Changing the address means changing one record

Account

acctNumer	bName	balance
001	Eindhoven Centrum	145
…	…	…
046	Eindhoven Centrum	200
047	The Hague Centrum	100
…	…	…

Customer

custName	custStreet	custCity
John Doe	Boschdijk	Eindhoven
Lotje Doe	Lindelaan	The Hague
…		

Depositor

custName	acctNumber
John Doe	001
John Doe	047
Lotje Doe	047
…	…

Primary key

- A **minimal set of attributes** of a table that **uniquely** defines each row of this table
  - custName defines the address → primary key
  - acctNumber defines the branch and the balance → primary key
  - custName does not define the account number, the account number does not define the custName → combination (custName, acctNumber) is a primary key

Account

acctNumer	bName	balance
001	Eindhoven Centrum	145
…	…	…
046	Eindhoven Centrum	200
047	The Hague Centrum	100
…	…	…

Customer

custName	custStreet	custCity
John Doe	Boschdijk	Eindhoven
Lotje Doe	Lindelaan	The Hague
…		

Depositor

custName	acctNumber
John Doe	001
John Doe	047
Lotje Doe	047
…	…

Primary key

- Another way to look at the "key" of a table:
  if we remove all "**non-key** columns" from the table, all rows are unique!

- Primary keys for the tables below?

Customer

custName	custStreet	custCity
John Doe	Boschdijk	Eindhoven
Lotje Doe	Lindelaan	The Hague
…		

- Every custName is unique,
  it is a primary key of this table

Depositor

custName	acctNumber
John Doe	001
John Doe	047
Lotje Doe	047
…	…

- The same custName can appear multiple times,
  custName is not a primary key of this table
- The same acctNumber can appear multiple times,
  acctNumber is not a primary key of this table
- The combination is always unique

Primary key

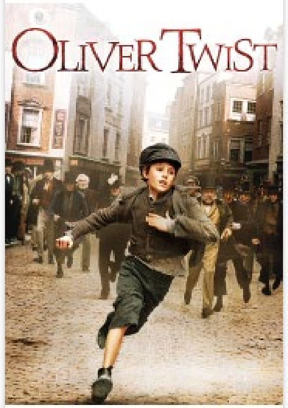
- A primary key is a **minimal** set of attributes (columns) that **uniquely identifies** each record (row) in the table
  - one single column or several columns
  - no subset of this set is a key
  - sometimes it is better to introduce an id

Movie

Movie title	Release date	Budget	Profit	
…	…	…	…	…

A primary key for the Movie table?

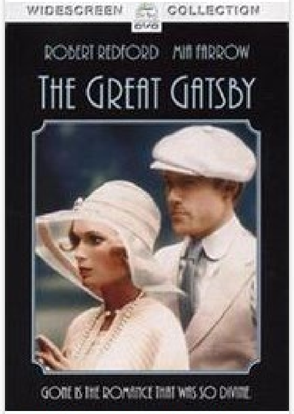
Oliver Twist (1948)



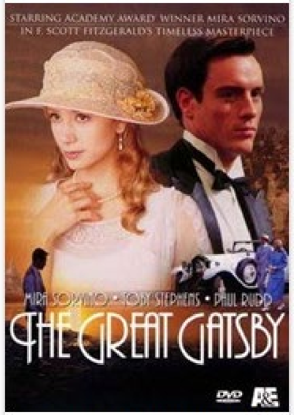
Oliver Twist (2005)

Oliver Twist (2007)

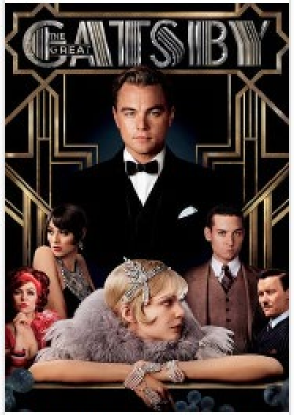
The Great Gatsby (1949)



The Great Gatsby (1974)



The Great Gatsby (2000)



The Great Gatsby (2013)

Summary: Why do we use database systems?

- A Database Management System (DBMS) is a very common type of information system.

- DBMS's are designed to support systematic principled solutions to these problems (and more):
  - **Data redundancy and inconsistency**
  - Data security
  - Expressive and efficient data analytics

- A **primary key** is a **minimal** set of attributes (columns) that **uniquely identifies** each record (row) in the table

Database Models

Database Models

- A **database model** is a collection of tools for describing:
  - Data
  - Data relationships
  - Data semantics (i.e., the meaning of the data)
  - Data constraints
- Historically, there have been many proposals:
  - Network and Hierarchical database models (1960-1970's)
  - Relational database model (1970-1980's)
  - Object-based database models (1980-1990's)
  - XML data model (1990-2000's)
  - RDF (graph) data model (2000-2010's)
  - …

- We study the **relational database model**, as it is the dominant practical model, and industry standard

Instances and Schemas

- Schemas are similar to variables in programming languages and instances are similar to variable values

- Logical Schema (data model) – logical structure of the database
  - Analogous to *name and type* of a variable in a program
  - A relational database schema consists of a collection of table schemas.

Schemas

Example: Suppose we have a bank **database schema** consisting of three tables:

- **customer(custName, custStreet, custCity)** table keeps track of each customer of the bank.
- **account(acctNumber, bName, balance)** table keeps track of each account of each branch of the bank.
- **depositor(custName, acctNumber)** table keeps track of which customer is associated to which account.

Instances

- **Instance – the actual content of the database at a particular point in time**
  - Analogous to the *value* of a variable

- *Example:* **(John Doe, Kruisstraat, Eindhoven) is an instance of the schema customer(custName, custStreet, custCity)**
  - Different terms used in the literature:
    an "**entry**", a "**tuple**", a "**row**"

Relation instance (table)

custName	custStreet	custCity
John Doe	Kruisstraat	Eindhoven
Mary Smith	Stratumsedijk	Eindhoven
Liesje Jansen	Veestraat	Helmond
Jantje Smit	Heuvelstraat	Tilburg
Klaas de Sint	Kasteelplein	Helmond
…	…	…

- a ***relation instance*** (also called a ***table***) with a schema consisting of three attributes, regarding **customers** with 6 rows, corresponding to 6 customers.
- The "**attributes**" or "**columns**" on which rows take values are *custName*, *custStreet,* and *custCity*.

A relational database instance (a collection of tables)

Customer

custName	custStreet	custCity
John Doe	Boschdijk	Eindhoven
Lotje Doe	Lindelaan	The Hague
…		

Account

acctNumer	bName	balance
001	Eindhoven Centrum	145
…	…	…
046	Eindhoven Centrum	200
047	The Hague Centrum	100
…	…	…

Depositor

custName	acctNumber
John Doe	001
…	…
John Doe	047
Lotje Doe	047
…	…

Summary database models

- A relational database model describes data and their relationship
- Database schemas define the organisation of tables
- *Instance* refers to the content of the database (at some moment)

Conceptual database design with E-R models

What is Database Design?

- A database represents the information of a particular domain (e.g, organization, experiment)
  - First and foremost: determine the ***information needs and the users***
  - Design a ***conceptual model*** for this information
  - Determine ***functional requirements*** for the system: which *operations* should be performed on the data?
- "Goodness" of *Conceptual* design:
  - Accurately reflect the semantics of use in the modeled domain

Modelling Entities in the ER model

- **A database can be modeled as** a collection of entities and relationships between entities.
- **An entity is an object that exists and is distinguishable from other objects.**
  - *Example*: a concrete person, e.g. Mary Johnson, a building, e.g. Auditorium at TU/e
- **Entities have attributes**
  - *Example*: people have names and addresses, buildings have addresses, height, etc.
- An **entity set** is a **set of entities** of the **same type** that share **the same properties.**
  - *Example*: set of all TU/e students, set of all TU/e buildings

Modeling Relationships in the ER Model

- A **relationship** is an **association among several entities**

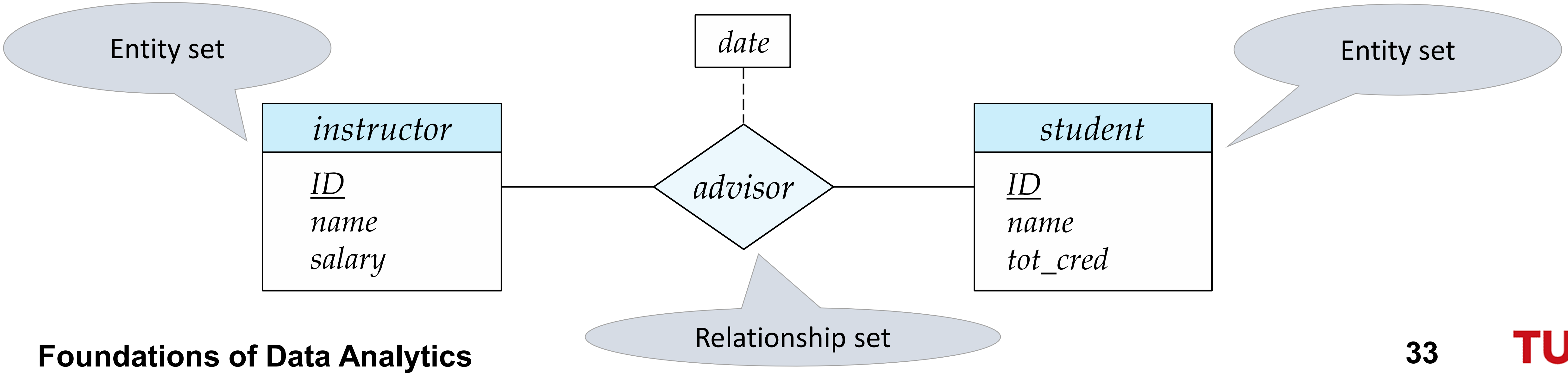
  Example:

<u>Crick</u>	**<u>advisor</u>**	**<u>Tanaka</u>**
instructor entity	***relationship set***	***student entity***

- **A relationship set is a collection of relationships among entity sets.**
  - **Example:**
- ***(Crick, Tanaka)*** **belongs to the relationship set** ***advisor***

The Entity-Relationship Model

- Models a real-world system as a collection of **entities** and **relationships**
  - *Entity*: a “thing” or “object” in the system that is distinguishable from other objects
    - Described by a set of attributes
  - *Relationship*: an **association among several entities**
- Represented diagrammatically by an **entity-relationship diagram**



Entity Sets instructor and student

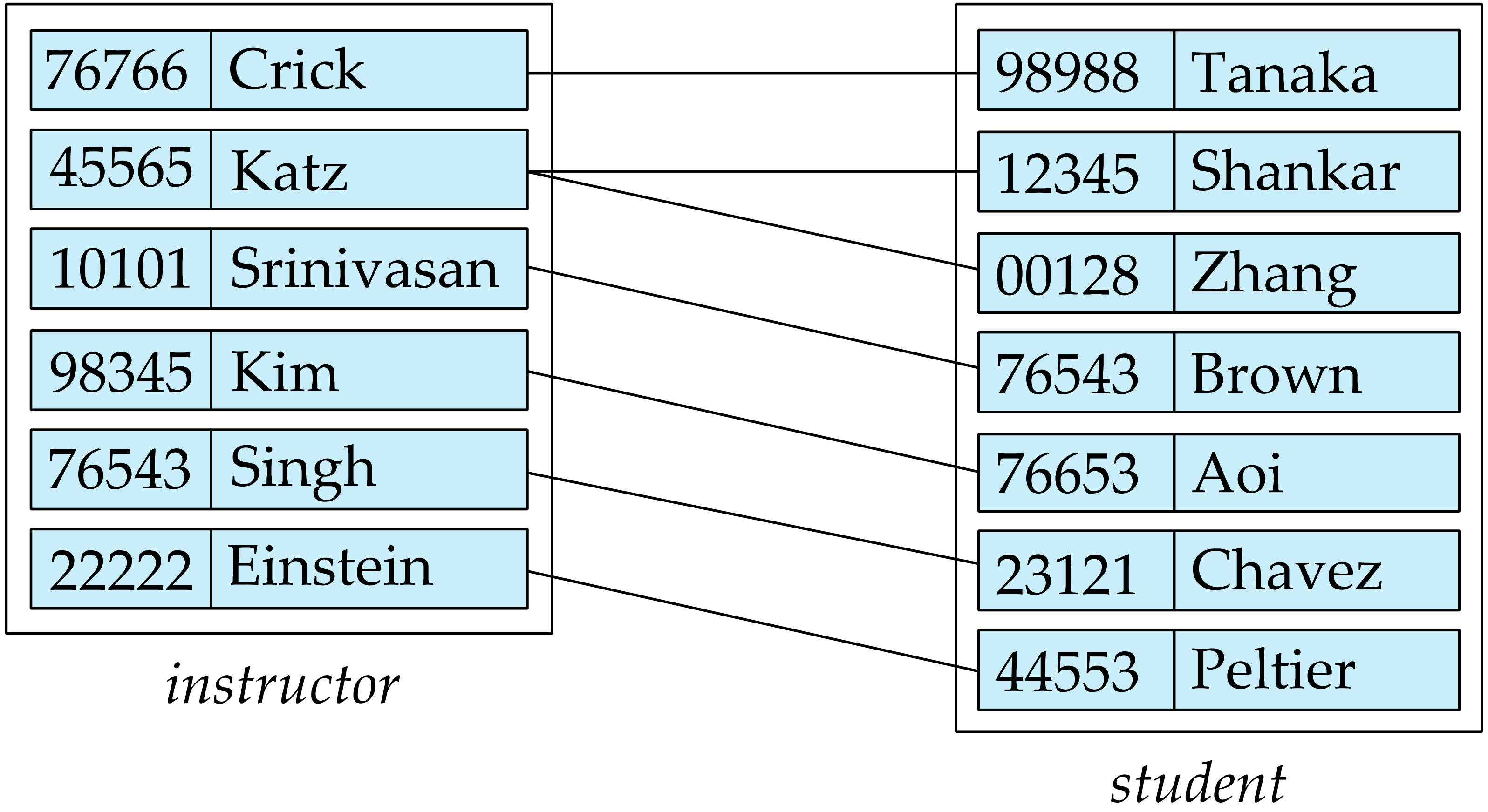
instructor_ID	instructor_name
76766	Crick
45565	Katz
10101	Srinivasan
98345	Kim
76543	Singh
22222	Einstein

instructor

student_ID	student_name
98988	Tanaka
12345	Shankar
00128	Zhang
76543	Brown
76653	Aoi
23121	Chavez
44553	Peltier

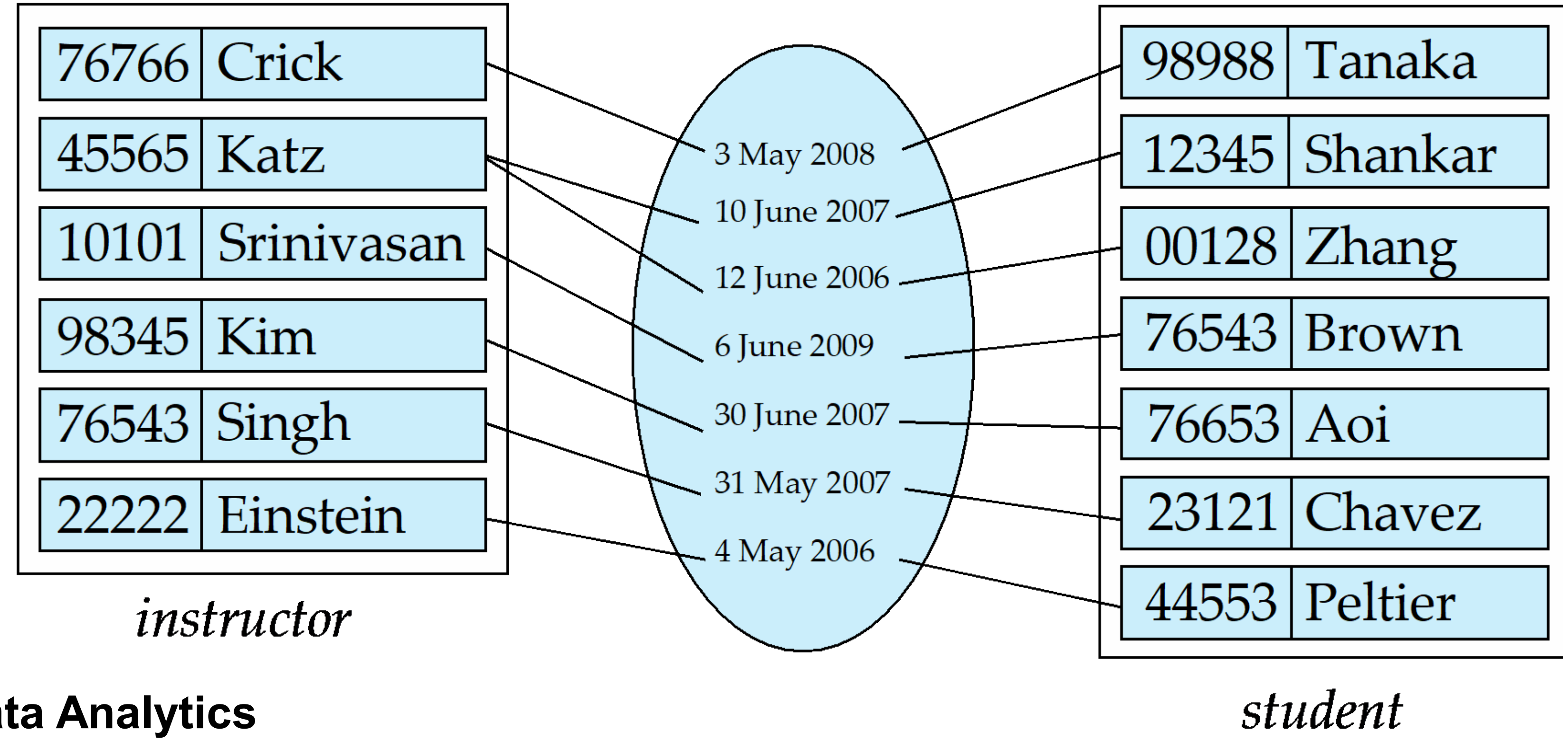
student

Relationship set advisor



Relationship Sets (cont.)

- An **attribute** can also be a property of a relationship set.
- For instance, the *advisor* relationship set between entity sets *instructor* and *student* may have an attribute *date*



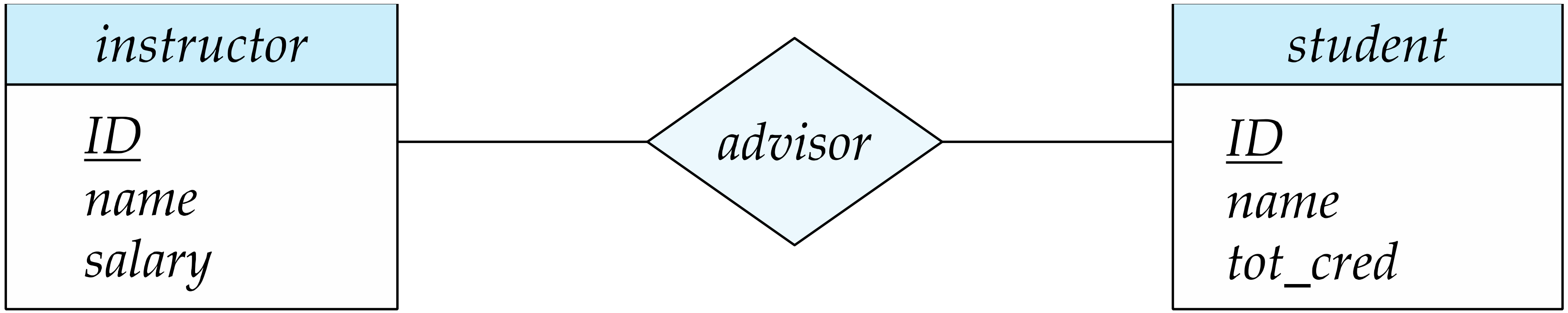
Attributes

- An entity is represented by a set of **attributes**, that is, descriptive properties possessed by all members of an entity set.
  - Example: entity sets

    instructor = (ID, name, street, city, salary)

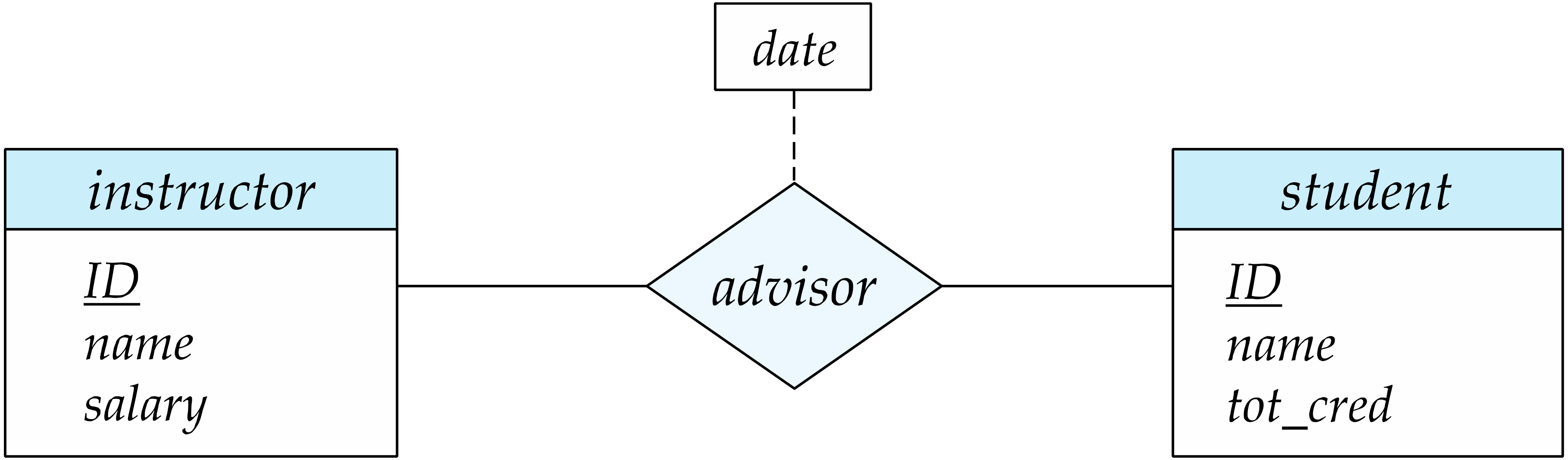
    course = (course_id, title, credits)
- **Domain** – the set of permitted values for each attribute
  - Example:
    - Instructor names are **strings** (e.g., “Jane Smith”)
    - Instructor salaries are **numbers** (e.g., 50000)

E-R Diagrams: syntax



- Rectangles represent entity sets.
- Diamonds represent relationship sets.
- Lines link entity sets to relationship sets.
- Attributes listed inside entity rectangles.
- **Underline** indicates "**primary key**" attributes
  - instructors have **unique** IDs and
  - students have **unique** IDs

Relationship Sets with Attributes



Types of Relationship Sets

An arrow means **ONE**!

One-to-one relationship

An instructor can be advisor of **only one** student, and a student can be advised by **only one** instructor.

One-to-many relationship

An instructor can be advisor of **many** students, but a student can be advised by **only one** instructor.

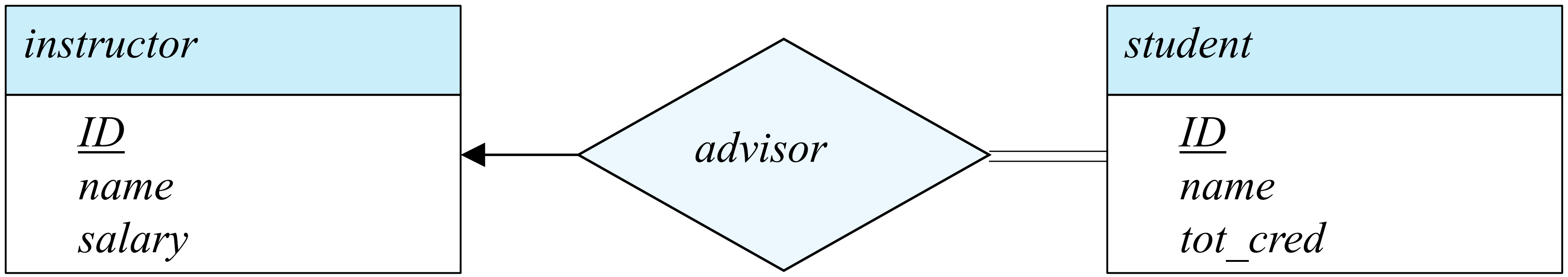
Many-to-many relationship

An instructor can be advisor of **many** students and a student can be advised by **many** instructors.

Pay attention to the word **can**. An instructor can be advisor of a student. But it also indicates that the instructor might not advise any student.

It indicates that **at most one** is related.

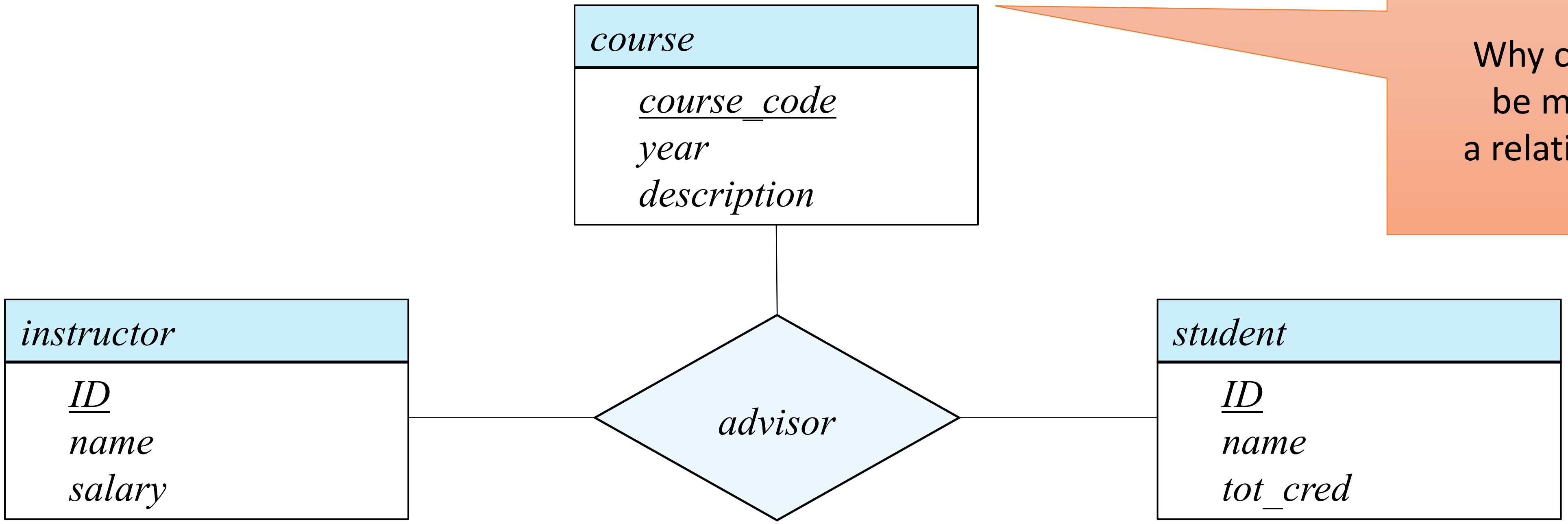
Types of Relationship Sets



A **double line** indicates **total** participation:
Every student **MUST be** related to an instructor via the advisor relationship.

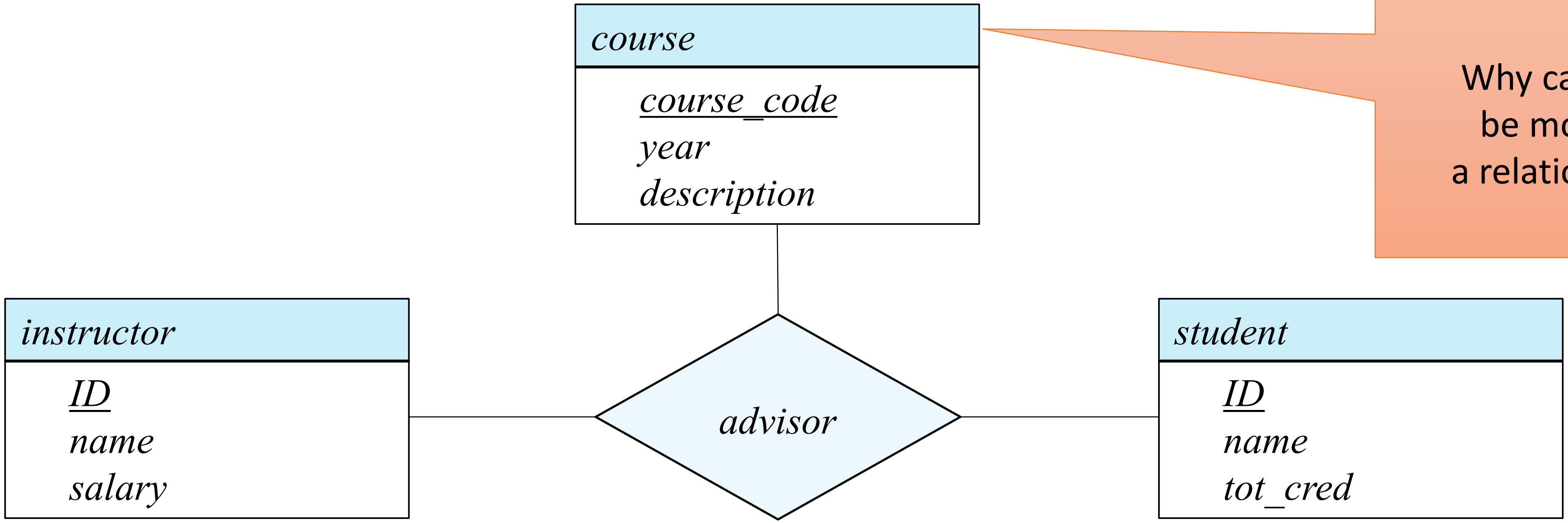
Relationship Sets (non-binary)

What if we want to keep records of which courses an instructor has advised a student in?



Why can't *course* be modeled as a relationship set?

Relationship Sets (non-binary)



Why can't *course* be modeled as a relationship set?

It cannot be modeled as a relationship set because it carries essential information that is not related to a student or to an instructor, like description.

A student can be advised by the same instructor in multiple courses. If course would be a relationship set with attributes, only the last record between an instructor and a student would be recorded in the database.

Summary Entity-Relationship models

- A good database requires designing a conceptual model of the data (how the data elements are related to each other)
- A database can be modeled as:
  - a collection of entity sets and
  - relationship sets linking the entity sets
- An Entity-Relationship model describes a database as entity sets and relationship sets

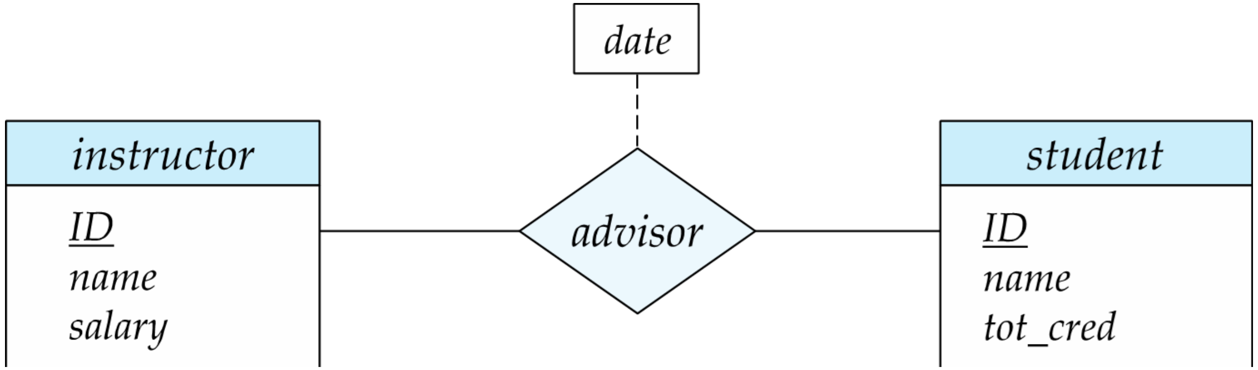
Translation to the relational model (i.e., implementing the ER conceptual model)

Reduction to Relation Schemas

- Both entity sets and relationship sets are expressed as **relation schemas** (tables) that represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of relation schemas.
- For each entity set and relationship set there is a unique schema that gets the name of the corresponding entity set or relationship set.
- Each schema has columns corresponding to attributes, which have unique names.

Reduction to Relation Schemas

- An entity set reduces to a schema with the same attributes
  - **primary key → primary key**
- A relationship set reduces to a schema whose **attributes** are the primary keys of the participating entity sets **and** attributes of the relationship set
  - The **primary key** is the **combination of the primary keys of the participating entities** (it **does not include** the attributes of the relationship set!)



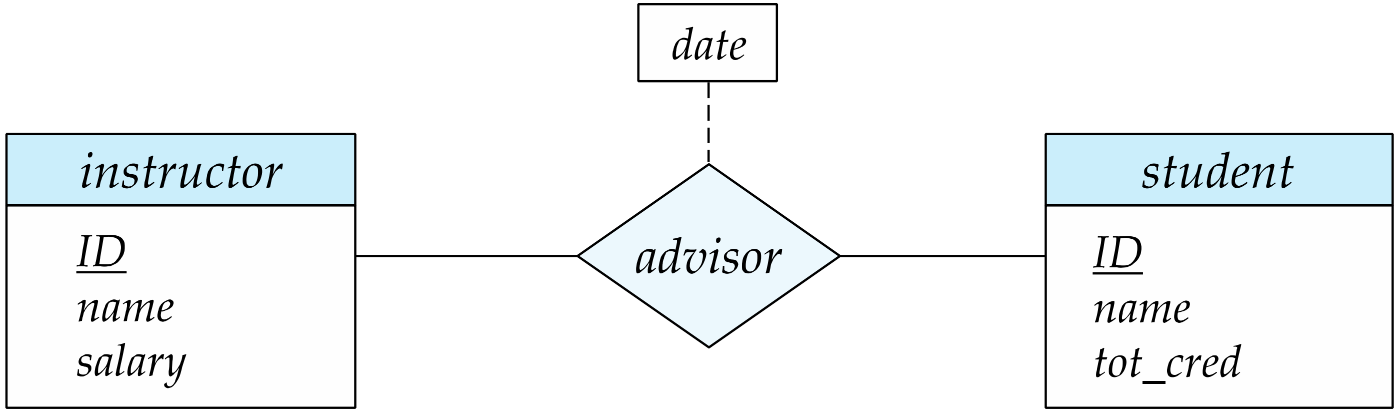
A relational database schema consisting of three tables:

instructor(instructor_id, name, salary)

student(student_id, name, tot_cred)

advisor (student_id, instructor_id, date)

Example



student

Student_ID	Name	tot_credit
s123	Luc	100
s234	Cindy	150
s456	Lue	80
…	…	

instructor

Instructor _ID	Name	Salary
inst123	Jane	1000
inst234	Sue	1000
inst456	Fred	800
inst789	Jane	2000

advisor

Instructor _ID	Student _ID	date
inst123	s234	1/1/2017
inst123	s456	8/7/2017
inst456	s123	9/11/2017
inst456	s234	6/4/2015
…	…	…

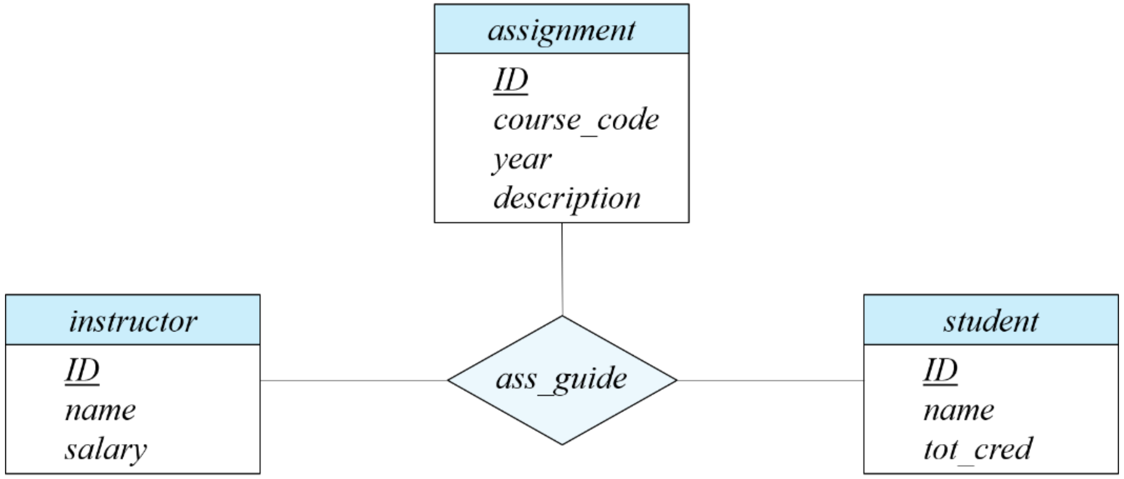
The database schema:

instructor(instructor_id, name, salary)

student(student_id, name, tot_cred)

advisor (student_id, instructor_id, date)

Example



student

Student_ID	Name	tot_credit
s123	Luc	100
s234	Cindy	150
s456	Lue	80
…	…	

instructor

Instructor_ID	Name	Salary
inst123	Jane	1000
inst234	Sue	1000
inst456	Fred	800
inst789	Jane	2000
…	…	

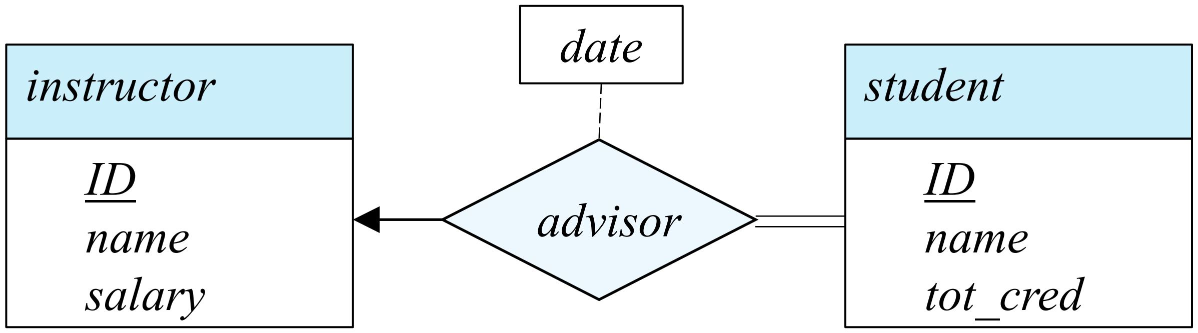
ass_guide

Instructor_ID	Student_ID	Ass_ID
inst123	s234	ass456
inst123	s456	ass456
inst456	s123	ass123
inst456	s234	ass234
…	…	…

assignment

Ass_ID	c_code	year	description
ass123	2IAB0	2020	…
ass234	2IAB0	2021	…
ass456	2WBB0	2021	…
…	…		

A total many to one relationship set: combination of tables



The standard reduction algorithm gives:

instructor(<u>instructor_id</u>, name, salary)

student(<u>student_id</u>, name, tot_cred)

advisor (<u>student_id</u>, <u>instructor_id</u>, date)

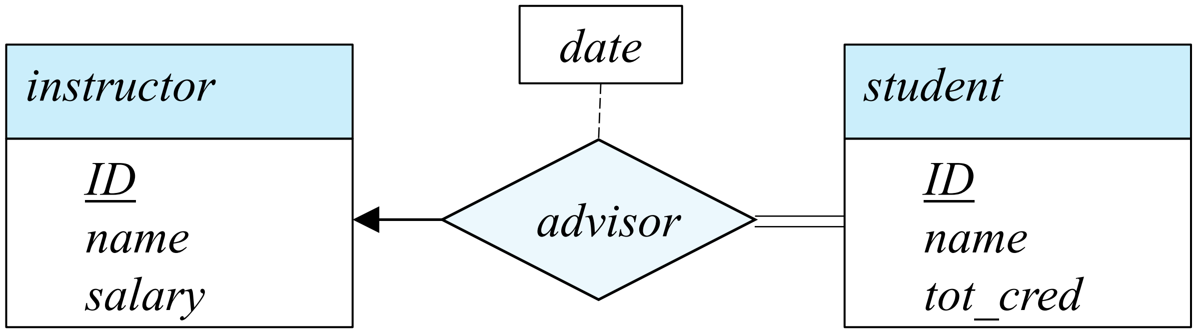
- Every student from the *student* table has exactly one instructor from the *instructor* table, so we can combine the tables student and advisor
- We only need two tables:

instructor(<u>instructor_id</u>, name, salary)

student(<u>student_id</u>, name, tot_cred, instructor_id, instructor_date)

Both solutions are correct but this one minimizes memory usage.

A total many to one relationship set: combination of tables



The standard reduction algorithm gives:

instructor(<u>instructor_id</u>, name, salary)

student(<u>student_id</u>, name, tot_cred)

advisor (<u>student_id</u>, <u>instructor_id</u>, date)

- Every student from the *student* table has exactly one instructor from the *instructor* table, so we can combine the tables student and advisor
- We only need two tables:

instructor(<u>instructor_id</u>, name, salary)

student(<u>student_id</u>, name, tot_cred, instructor_id, instructor_date)

Note that!
instructor_id is not part of the key of the student table, after joining these two tables.

Summary Relation Schemas

- An entity set reduces to a table with the same attributes.

- A relationship set is represented as a table with attributes for the primary keys of the participating entity sets (they form the primary key of this table), and attributes of the relationship set.

- When there is a total one-to-many relationship set between entity sets, then tables can be combined.

Data querying with SQL

Basic SQL

- Industry standard query language for RDBMS (relational database management systems)
- Query language: a language in which to express data analytics

What is SQL?

- Structured Query Language (SQL = "Sequel")
- Industry standard query language for RDBMS (relational database management systems)
- designed by IBM, now an ISO standard
  - available in most database systems (with local variations)
  - has *procedural* flavor but is *declarative* in principle
  - "reads like English"
- uses a relational database model where:
  - tuples in relation instance are ordered
  - query results may contain duplicate tuples

Why SQL?

- Lingua franca of data intensive systems
  - Relational databases
  - MapReduce systems, such as Hadoop
  - Streaming systems, such as Apache Spark and Apache Flink
  - …
- Stable mature language with 40+ years of international standardization and study
  - SQL is essentially the industry standard for first order logic, i.e., predicate logic, i.e., the relational DB model (see 2ID50 in Q2)
- Via information systems, you will continue to interact directly or indirectly with SQL or SQL-like languages for the coming decades …

Conventions in SQL

- Keywords are case-insensitive. They are often capitalized for better readability
- Table and column names can be case-sensitive (often configurable)
- The semicolon (;) at the end of an SQL statement is not mandatory for most database systems

Data Definition Language

- SQL DDL: Allows the specification of a set of relations (i.e., the relational database schema) and information about each relation, including:
  - The schema for each relation
  - The domain of values associated with each attribute
  - Integrity constraints
- An important constraint for attributes: is it allowed to have **missing values**? Missing attribute values get a **special value NULL**.

Data Definition Language

- *Example*

```
CREATE TABLE instructor (
        instructor_ID CHAR(5),          /*fixed-size string of 5 bites*/
        name VARCHAR(20),    /*variable-size string of at most 20 bites*/
        salary NUMERIC(8,2));
```

instructor

Instructor_ID	Name	Salary

Data Definition Language

- *Example*

```
INSERT INTO instructor
VALUES ('10211', 'Smith', 66000);
```

instructor

Instructor_ID	Name	Salary
10211	Smith	66000

Data Definition Language

- *Example*

```
INSERT INTO instructor
VALUES (‘10212’, ‘Nabokov’, 68000);
```

instructor

Instructor_ID	Name	Salary
10211	Smith	66000
10212	Nabokov	68000

Data Definition Language

- *Example*

```
UPDATE instructor
SET salary = salary *1.05
WHERE salary < 100000;
```

instructor

Instructor_ID	Name	Salary
10211	Smith	66000
10212	Nabokov	68000

After update:

instructor

Instructor_ID	Name	Salary
10211	Smith	69300
10212	Nabokov	71400

Summary SQL

- SQL is a query language for relational databases, i.e. it allows you to perform operations on databases

Query structure

Basic Query Structure

- The SQL *data-manipulation language (DML)* provides the ability to query information

- A typical SQL query has the form:

  SELECT *instructor.instructor_id, instructor.name*
  FROM *instructor, advisor, student*
  WHERE *instructor.instructor_id = advsor.instructor_id* and
  student.student_id = advisor.student_id and
  student.tot_credit < 100;

- *SELECT* lists attributes to retrieve
- *FROM* lists tables from which we query
- *WHERE* defines a predicate (i.e., a filter) over the values of attributes.

Example

instructor

Instructor_ID	Name	Salary
inst123	Jane	1000
inst234	Sue	1000
inst456	Fred	800
inst789	Jane	2000

student

Student_ID	Name	tot_credit
s123	Luc	100
s234	Cindy	150
s456	Lue	80

advisor

Instructor_ID	Student_ID	date
inst123	s234	1/1/2017
inst123	s456	8/7/2017
inst456	s123	9/11/2017
inst456	s234	6/4/2015

```
SELECT instructor.instructor_id, instructor.name
FROM   instructor, advisor, student
WHERE instructor.instructor_id = advisor.instructor_id
         AND student.student_id = advisor.student_id
         AND student.tot_credit <100;
```

Query result

Instructor_ID	Name
inst123	Jane

Banking example database schema

branch *(bName, bCity, assets)*

customer *(custName, custStreet, custCity)*

account *(acctNumber, bName, balance)*

loan *(loanNumber, bName, amount)*

depositor *(custName, acctNumber)*

borrower *(custName, loanNumber)*

The SELECT clause

- The **SELECT** clause lists the attributes desired in the result of a query, **it performs a "vertical" restriction of the table**
- *Example*: find the names of all branches in the loan relation

  SELECT *branch_name*
  FROM *loan;*

loan

loanNumber	branch_Name	amount
8844	Eindhoven east	1010
1765	Perryridge	2945
9977	Perryridge	98
6565	Tilburg south	12000
8768	Perryridge	15000
1234	Amsterdam east	453
…	…	…

The SELECT clause: duplicates

- **SQL allows duplicates** in relations as well as in query results
- To **eliminate duplicates**, insert the keyword **DISTINCT after SELECT**
- *Example*: Find the names of all branches in the loan relation, and remove duplicates

SELECT DISTINCT *branch_name*
FROM *loan*;

- The keyword **ALL** specifies that duplicates should not be removed (default option)

SELECT ALL *branch_name*
FROM *loan*;

ALL

Branch_Name
Eindhoven east
Perryridge
Perryridge
Perryridge
Perryridge
Amsterdam east
…

DISTINCT

Branch_Name
Eindhoven east
Perryridge
Tilburg south
Amsterdam east
…

The SELECT Clause: *

- An asterisk in the select clause denotes "all attributes"

SELECT *
FROM *loan*;

This will just result in the same table as loan

The SELECT Clause: calculations

- The SELECT clause can contain arithmetic expressions involving the operations +, –, $*$, and /, and operating on constants or attributes of tuples.
- The query:

  SELECT *loan_number, branch_name, amount* $*$ 100
  FROM *loan;*

  would return a relation that is the same as the *loan* relation, except that the value of the attribute *amount* is multiplied by 100 (euro to cents)

The WHERE clause

- **The WHERE clause performs a "horizontal" restriction of the table, it specifies conditions on the rows (records)**
- *Example*: find all loan number for loans made at the Perryridge branch with loan amounts greater than \$1200.

```
SELECT loan_number
FROM   loan
WHERE branch_name = 'Perryridge'  AND
          amount > 1200
```

loan

loanNumber	branch_Name	amount
8844	Eindhoven east	1010
1765	Perryridge	2945
9977	Perryridge	98
6565	Tilburg south	12000
8768	Perryridge	15000
1234	Amsterdam east	453
…	…	…

The WHERE clause: operators

- Conditions can use the logical operators **AND**, **OR**, and **NOT**, and parentheses **( )** for grouping
- Comparisons =, <>, <, <=, >, >=, can be applied to results of arithmetic expressions.
- String comparison: operator **LIKE** and
  - %   the percent sign represents zero, one, or multiple characters
  - Write strings between ‘ ‘

(amount <=1000 OR amount >1000000) AND (branch_name LIKE ‘%Eindhoven%’)

loan amounts not exceeding 1000 or greater than 1000000 and
branch names containing “Eindhoven” (e.g. Eindhoven-Woensel, or BigEindhoven1)

Summary Queries

- A typical SQL query uses the following clauses
  - SELECT (“vertical selection” from table)
  - FROM (list the tables from which we query)
  - WHERE (“horizontal selection” from table)
- The order select – from – where is fixed

Joining tables

The FROM clause

- The **FROM** clause lists the tables involved in the query

- Find all pairs of *instructor* and *teaches* entries

  SELECT *
  FROM *instructor, teaches*

- this generates **every possible pair** (instructor, teaches), with all attributes from both relations

SELECT * FROM instructor, teaches

instructor

ID	name	dept_name	salary
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
[illegible]	[illegible]	[illegible]	[illegible]

teaches

ID	course_id	sec_id	semester	year
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

SELECT * FROM instructor, teaches

inst.ID	name	dept_name	salary	teaches.ID	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Multiple tables in the FROM Clause

- Generating all pairs of entries is not very useful directly, but is useful combined with where-clause conditions

Example

instructor

Instructor_ID	Name	Salary
inst123	Jane	1000
inst234	Sue	1000
inst456	Fred	800
inst789	Jane	2000

student

Student_ID	Name	tot_credit
s123	Luc	100
s234	Cindy	150
s456	Lue	80

advisor

Instructor_ID	Student_ID	date
inst123	s234	1/1/2017
inst123	s456	8/7/2017
inst456	s123	9/11/2017
inst456	s234	6/4/2015

```
SELECT instructor.instructor_id, instructor.name
FROM   instructor, advisor, student
WHERE  instructor.instructor_id = advisor.instructor_id
       AND student.student_id = advisor.student_id
       AND student.tot_credit < 100;
```

Query result

Instructor_ID	Name
inst123	Jane

Example

instructor

Instructor_ID	Name	Salary
inst123	Jane	1000
inst234	Sue	1000
inst456	Fred	800
inst789	Jane	2000

student

Student_ID	Name	tot_credit
s123	Luc	100
s234	Cindy	150
s456	Lue	80

advisor

Instructor_ID	Student_ID	date
inst123	s234	1/1/2017
inst123	s456	8/7/2017
inst456	s123	9/11/2017
inst456	s234	6/4/2015

SELECT *instructor.instructor_id, instructor.name*
FROM *instructor, advisor, student*
WHERE ***instructor.instructor_id = advisor.instructor_id***
AND student.student_id = advisor.student_id
AND student.tot_credit <100;

Query result

Instructor_ID	Name
inst123	Jane

The from clause: example

Linking data from the borrower and loan tables:

Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch:

branch *(bName, bCity, assets)*
customer *(custName, custStreet, custCity)*
account *(acctNumber, bName, balance)*
loan *(loanNumber, bName, amount)*
depositor *(custName, acctNumber)*
borrower *(custName, loanNumber)*

```
SELECT customer_name, borrower.loan_number, amount
FROM   borrower, loan
WHERE  borrower.loan_number = loan.loan_number
       AND
       branch_name = 'Perryridge'
```

The rename operation AS

- SQL allows renaming relations and attributes using the **AS** clause:
  old-name AS *new-name*
- *Example*: find the name, loan number and loan amount of all customers; rename the column name *loan_number* to *loan_id.*

SELECT *customer_name, borrower.loan_number* **AS** *loan_id, amount*
FROM *borrower, loan*
WHERE *borrower.loan_number = loan.loan_number*

Tuple variables

- Tuple variables are defined in the FROM clause using the AS clause
- Keyword AS is optional and may be omitted,
  i.e. you can write ***borrower b*** instead of ***borrower* AS *b***

Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch:

```
SELECT customer_name, borrower.loan_number, amount
FROM   borrower AS b, loan AS l
WHERE  b.loan_number = l.loan_number
       AND
       branch_name = ‘Perryridge’
```

Tuple variables

- Find the names of all branches that have greater assets than some branch located in Brooklyn.

branch *(<u>bName</u>, bCity, assets)*
customer *(<u>custName</u>, custStreet, custCity)*
account *(<u>acctNumber</u>, bName, balance)*
loan *(<u>loanNumber</u>, bName, amount)*
depositor *(<u>custName</u>, <u>acctNumber</u>)*
borrower *(<u>custName</u>, <u>loanNumber</u>)*

```
SELECT DISTINCT b2.branch_name
FROM  branch b1, branch b2
WHERE b2.assets > b1.assets AND b1.branch_city = ‘Brooklyn’
```

Summary joining tables

- Use the WHERE clause in order to define the way you join the tables!

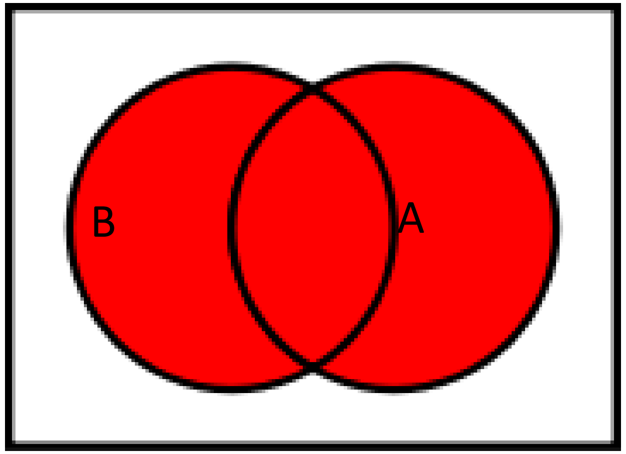
Set operations

Set operations

- The set operations **UNION**, **INTERSECT**, and **EXCEPT** operate on relations.
  - A “set” is a collection of objects without repetition of elements and without any particular order.
  - E.g., each of us has a set of hobbies, which might be empty.

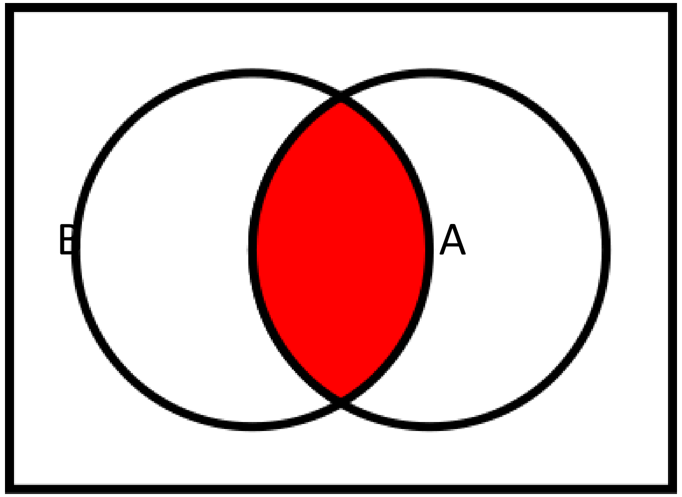
Set Operations: union

- Given two sets A and B, the **union** of A and B is the set containing all elements of both A and B.



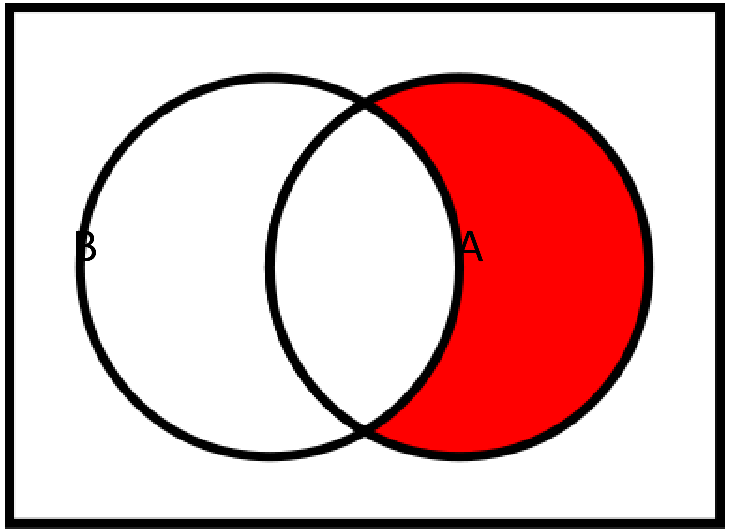
Set Operations: intersection

- Given two sets A and B, the **intersection** of A and B is the set containing all elements appearing in both A and B.



Set Operations: set difference - except

- Given two sets A and B, the **difference** of A and B is the set containing all elements appearing in A but not in B. This is denoted by **EXCEPT** in SQL.



Set Operations

- The set operations **UNION**, **INTERSECT**, and **EXCEPT** operate on relations (tables).
- Each of these operations automatically eliminates duplicates; to retain all duplicates, use the corresponding multiset versions **UNION ALL**, **INTERSECT ALL** and **EXCEPT ALL**.

Set Operations, examples

- **Find all customers who have a loan, an account, or both:**

branch *(bName, bCity, assets)*
customer *(custName, custStreet, custCity)*
account *(acctNumber, bName, balance)*
loan *(loanNumber, bName, amount)*
depositor *(custName, acctNumber)*
borrower *(custName, loanNumber)*

(**SELECT** *customer_name* **FROM** *depositor*)
UNION
(**SELECT** *customer_name* **FROM** *borrower*)

Set Operations, examples

- Find all customers who have a loan, an account**, or both:**

  (SELECT *customer_name* **FROM** *depositor*)
  UNION
  (SELECT *customer_name* **FROM** *borrower*)

- Find all customers who have **both** a loan **and** an account**:**

  (SELECT *customer_name* **FROM** *depositor*)
  INTERSECT
  (SELECT *customer_name* **FROM** *borrower*)

- Find all customers who have an account **but no** loan**:**

  (SELECT *customer_name* **FROM** *depositor*)
  EXCEPT
  (SELECT *customer_name* **FROM** *borrower*)

Summary set operations

- Basic set operations on databases are:
  - union (SQL: UNION, UNION ALL)
  - intersection (SQL: INTERSECT, INTERSECT ALL)
  - set difference (SQL: EXCEPT ,EXCEPT ALL)

Aggregate functions in SQL

Aggregate Functions

These functions operate on all values of a column (including duplicate values, by default), and return a value

COUNT: number of values

MIN: minimum value
MAX: maximum value
AVG: average value (on numbers)
SUM: sum of values (on numbers)

Aggregate functions: examples

- **Find the average account balance at the Perryridge branch.**

  SELECT AVG *(balance)* **AS** *avgBalance*

  FROM *account*
  WHERE *branch_name* = 'Perryridge'

- **Find the number of *customers* in the bank.**

  SELECT COUNT (*) **AS** custCnt
  FROM *customer*

- **Find the number of depositors in the bank.**

  SELECT COUNT **(DISTINCT** *customer_name)*
  FROM *depositor*

branch *(<u>bName</u>, bCity, assets)*
customer *(<u>custName</u>, custStreet, custCity)*
account *(<u>acctNumber</u>, bName, balance)*
loan *(<u>loanNumber</u>, bName, amount)*
depositor *(<u>custName</u>, <u>acctNumber</u>)*
borrower *(<u>custName</u>, <u>loanNumber</u>)*

Aggregate Functions: Group By

Find the total assets in each city where the bank has a branch.

```
SELECT bCity, SUM(assets) AS totalAssets
FROM   branch
GROUP BY bCity
```

branch

bName	bCity	assets
Perryridge	Mainville	102000
Eindhoven east	Eindhoven	100000
Eindhoven west	Eindhoven	200000
Tilburg south	**Tilburg**	**21000**
Amsterdam east	Amsterdam	10000
Amsterdam south	Amsterdam	40000

Query result

bCity	totalAssets
Mainville	102000
Eindhoven	300000
Tilburg	**21000**
Amsterdam	50000

Aggregate functions: GROUP BY example

- Find the **number of depositors** for each **branch**.

branch *(<u>bName</u>, bCity, assets)*
customer *(<u>custName</u>, custStreet, custCity)*
account *(<u>acctNumber</u>, bName, balance)*
loan *(<u>loanNumber</u>, bName, amount)*
depositor *(<u>custName</u>, <u>acctNumber</u>)*
borrower *(<u>custName</u>, <u>loanNumber</u>)*

```
SELECT      branch_name, COUNT (DISTINCT customer_name)
FROM        depositor, account
WHERE       depositor.account_number = account.account_number
GROUP BY    branch_name
```

Aggregate functions: HAVING clause

Find the total assets in each city where the bank has a branch with the total assets of at least 100k.

SELECT ***bCity***, **SUM(***assets***) AS** *totalAssets*
FROM *branch*
GROUP BY ***bCity***
HAVING SUM(*assets*) >= 100000

branch

bName	bCity	assets
Perryridge	Mainville	102000
Eindhoven east	Eindhoven	100000
Eindhoven west	Eindhoven	200000
Tilburg south	**Tilburg**	**21000**
Amsterdam east	Amsterdam	10000
Amsterdam south	Amsterdam	40000

Note: predicates in the **HAVING** clause are applied **after grouping**, whereas predicates in the **WHERE** clause are applied **before grouping**

Old query result, without having:

bCity	Assets
Mainville	102000
Eindhoven	300000
Tilburg	**21000**
Amsterdam	50000

New query result, with having:

bCity	assets
Mainville	102000
Eindhoven	300000

Aggregate Functions – Having Clause

Find the names and average account balances of all **branches in Eindhoven** where the **average account balance is more than 1200 euros**.

```
SELECT branch_name, AVG (balance)
FROM    account as acc, branch AS b
WHERE acc.bName = b.bName AND
            bCity = 'Eindhoven'
GROUP BY branch_name
HAVING AVG(balance) > 1200
```

branch *(bName, bCity, assets)*
customer *(custName, custStreet, custCity)*
account *(acctNumber, bName, balance)*
loan *(loanNumber, bName, amount)*
depositor *(custName, acctNumber)*
borrower *(custName, loanNumber)*

Remember: predicates in the **HAVING** clause are applied **after grouping**, whereas predicates in the **WHERE** clause are applied **before grouping**

Aggregate Functions – More examples

- Find the names of customers and the number of their accounts for customers who have more than one account and the total balance of their accounts is higher than €1200

SELECT *customer.custName,* **COUNT** (account.*acctNumber*)
FROM *customer, account, depositor*
WHERE *customer.custName = depositor.custName* **AND** *depositor.acctNumber = account.acctNumber*
GROUP BY *customer.custName*
HAVING COUNT (account.*acctNumber*) > *1* **AND SUM** *(account.balance) > 1200*

- Find the names of branches for which the total of loan amounts is higher than the assets of that branch

SELECT *branch.bName*
FROM *branch, loan*
WHERE *branch.bName = loan.bName*
GROUP BY *branch.bName*
HAVING SUM *(loan.amount) > branch.assets*

Summary: Aggregate Functions

- *Conceptual evaluation of a query with aggregation*

1. The **FROM** clause is first evaluated to get a relation
2. If a **WHERE** clause is present, then the predicate in the clause is applied on this result
3. Tuples from the resulting relation are then placed into groups by the **GROUP BY** clause
4. The **HAVING** clause is applied to each group, with groups not satisfying the clause removed
5. The **SELECT** clause results in **one tuple per group**, after applying the aggregate function(s).
    - The **SELECT** clause has to contain the attribute(s) used in **GROUP BY** and possibly the aggregate functions
    - The aggregate functions can appear in the **SELECT** clause and in the **HAVING** clause, but **NOT in the WHERE clause**

Conclusion

What you need to know

- Why we ought to organize data
- The notion of primary key
- The basics of conceptual modeling, using the **Entity-Relationship model**
- The basics of **Relational Database Models**
- How to **translate** Entity-Relationship model into a relational database schema
- Basic of **SQL querying language** including
  - the basics of grouping and aggregation (GROUP BY, HAVING)
  - Set operations: union, intersection and except

What you should be able to do

- Read and interpret Entity-Relationship models
- Translate a simple Entity-Relationship models to relational database models
- Understand what simple SQL queries retrieve from a data set
- Write simple SQL queries, including
  - the basics of grouping and aggregation (GROUP BY, HAVING)
  - Set operations: UNION, INTERSECTION and EXCEPT

Follow-up course

- 2ID50 Datamodelling and databases

Old exam problems

Problem 1

- Which of the following queries reflects the question:
  "Which branches have lower assets than the total balance of all accounts of that branch?"

a.
```
SELECT    branch.bName
FROM      account, branch
WHERE     branch.bName = account.bName
GROUP BY  branch.bName
HAVING    assets < SUM(balance);
```

b.
```
SELECT    bName
FROM      account
GROUP BY  bName
HAVING    assets < SUM(balance);
```

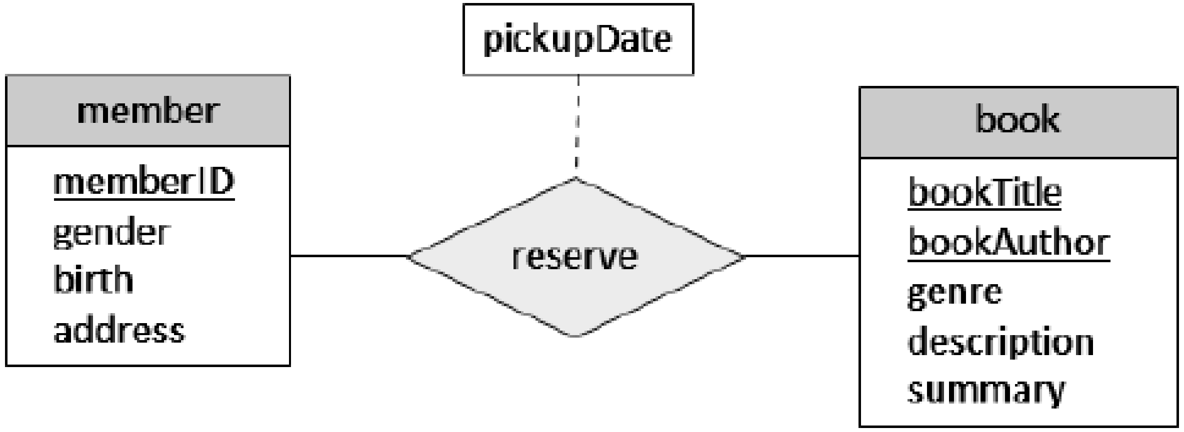
c.
```
SELECT    branch.bName
FROM      account, branch
WHERE     branch.bName = account.bName
          AND assets < SUM(balance)
GROUP BY  assets;
```

d.
```
SELECT    branch.bName
FROM      account, branch
WHERE     branch.bName = account.bName
GROUP BY  assets
HAVING    assets < SUM(balance);
```

Problem 2

Description of a library reservation system

A library uses a simple database for registering the reservations that members make to be able to pick up books they want to borrow. Its E-R diagram is shown below. The underlined attributes indicate the primary keys of the entity sets. Note that whenever a book has been picked up or the pick up date has passed, the reservation is removed from the database.



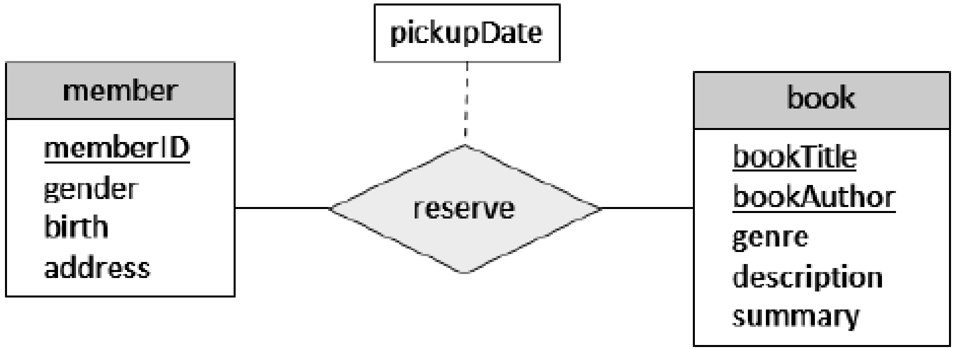
When translating the E-R diagram to a relational database schema, which tables would be part of the relational model?

What is the primary key of each table?

Problem 2

Description of a library reservation system

A library uses a simple database for registering the reservations that members make to be able to pick up books they want to borrow. Its E-R diagram is shown below. The underlined attributes indicate the primary keys of the entity sets. Note that whenever a book has been picked up or the pick up date has passed, the reservation is removed from the database.



Consider the following statements:

Statement A: Introducing a new *relationship set* **borrowing(between member and book)** is an appropriate modification of the ER-diagram if the library also wants keep a record of every time that a member borrows a book. Adding attribute **date** to relationship set **borrowing** allows to model situations when a member borrows the same book multiple times on different dates.

Statement B: The database represented by the E-R diagram contains the data allowing to write and answer a query getting the list of the library books written by a certain author.

What can be said about these statements?