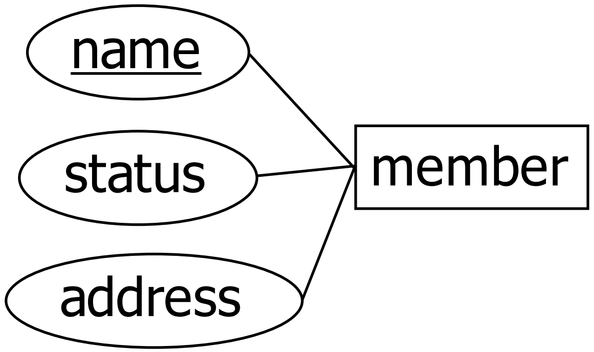
The Relational Database Model

- From E/R Model to Relational DB Model
  - Entities and Attributes:



member = (name:STRING, status:STRING, address:STRING)

member

name	status	address
...	...	...
...	...	...
...	...	...

  - Each entity is assigned to a relation
  - Each attribute of the entity becomes an attribute of the relation
  - The key of an entity becomes the primary key of the relation
  - Further attributes can be added to the relation (e.g. for relationships)

The Relational Database Model

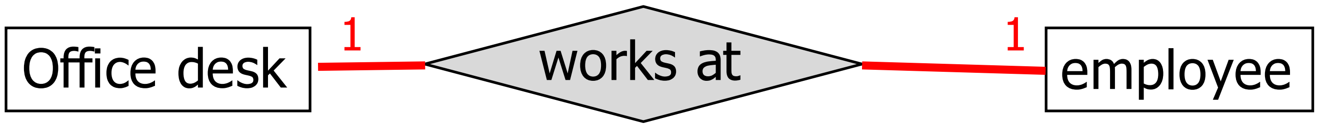
- From E/R Model to Relational DB Model
  - Relationships:
    - Realization of relationships in the relational model mainly depends on the functionality/cardinality of the relationships
    - General transformation rules (not strict but recommended for most cases):

functionality:	rule:
1:1	additional attributes in existing relations
1:n	additional attributes in existing relations
n:m	Generation of an additional relation

    - The first two functionalities are specific cases of the third one, so it is always possible to realize a relationship in a relation model by means of an additional relation, but due to performance issues, additional relations should be avoided if possible !!!

The Relational Database Model

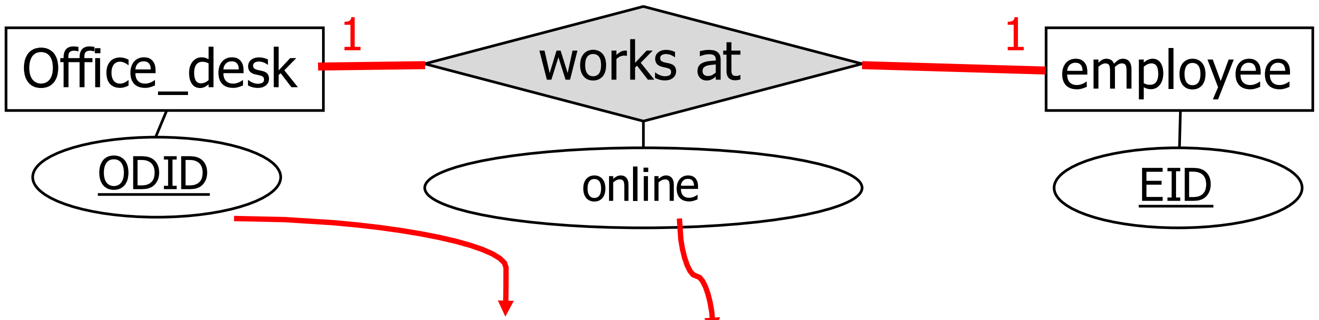
- From E/R Model to Relational DB Model
  - **One-to-One (1:1) Relationship:**

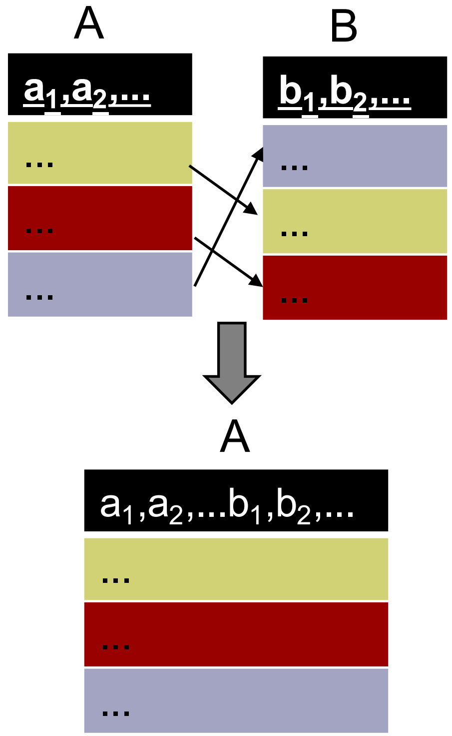


- No additional relation required !!!
- Both entities will be merged into one relation
- One of the primary keys of the entities is selected as primary key of the resulting relation

Note: Often, 1:1 relationships will be transformed in the same way as 1:n relationships (see later)

- From E/R Model to Relational DB Model
  - Example of One-to-One (1:1) Relationship transformation:



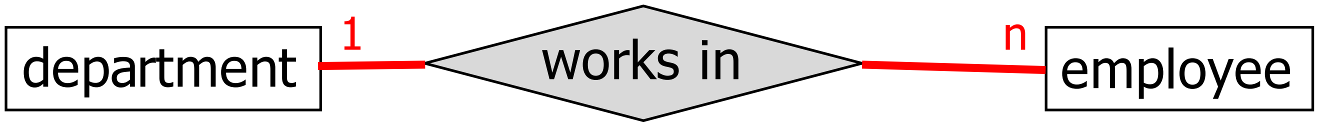


Employee (EID, name, …., ODID, ..., online, ...)

Alternative: Office_desk (ODID, name, …., EID, ..., online, ...)

The Relational Database Model

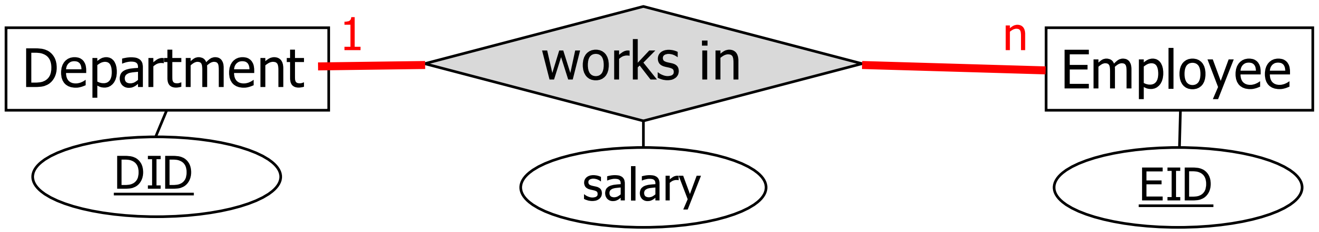
- From E/R Model to Relational DB Model
  - **One-to-Many (1:n) Relationship:**



- No additional relation required !!!
- Primary key of the relation (Entity) at the “one”-side (e.g. “department”)
  → additional attribute of the relation (Entity) at the “many”-side (e.g. employee) as foreign key
- No change of the primary keys of both relations
- Attributes of the relationship (if there are any) will be moved to the relation at the “many”-side as further additional attributes (but not as foreign keys).

The Relational Database Model

- From E/R Model to Relational DB Model
  - Example of One-to-Many (1:n) Relationship transformation:



Department (DID, name, ….)

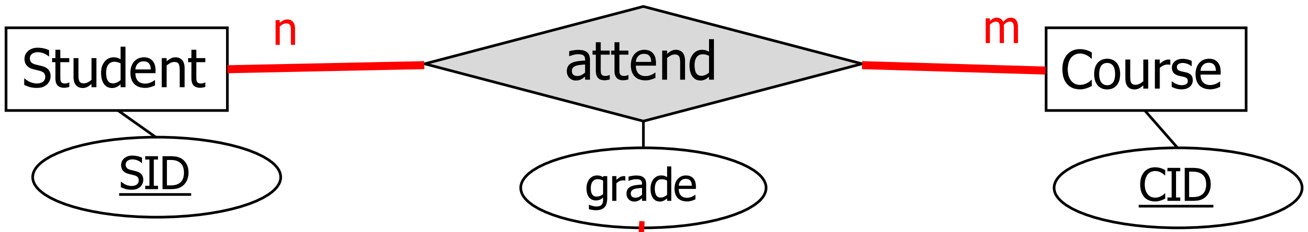
Employee (EID, name, age, …, DID, salary)

- From E/R Model to Relational DB Model
  - **Many-to-Many (n:m) Relationship:**

- Additional relation R required (with the name of the relationship)!!! Why?
- Attributes of the new relation R: At least, the primary keys of the two relations that are associated with the entities on both sides
- These attributes serve as foreign keys in R
- In addition, these attributes build the primary key of R
- Attributes of the relationship itself (if there are any) are added to the relation R as well (but neither with foreign key role, nor serving as primary key).

The Relational Database Model

- From E/R Model to Relational DB Model
  - Example of Many-to-Many (n:m) Relationship transformation:



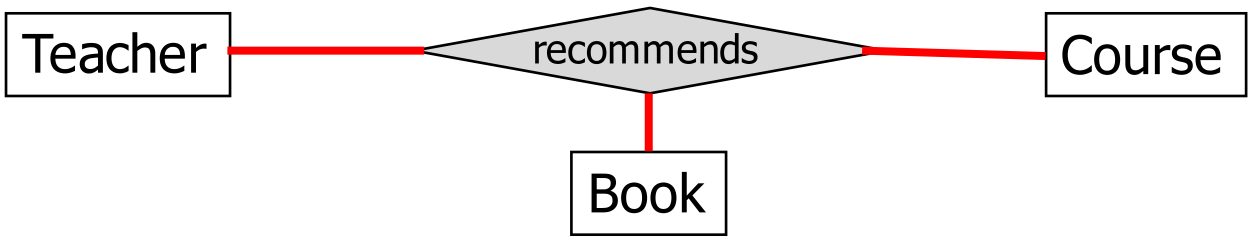
Student (SID, name, ...)

Course (CID, name, ...)

Attend (SID, CID, grade, ...)

The Relational Database Model

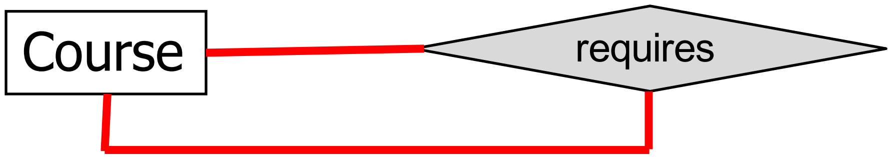
- From E/R Model to Relational DB Model
  - **Multi-Entity Relationships (e.g. ternary relationship):**



- If at least one functionality is “many” → Create own relation for the relationship (similar to n.m-relationship)
- Else, merge all entities into one relation (similar to 1:1 relationship)

The Relational Database Model

- From E/R Model to Relational DB Model
  - **Relationship of an entity to itself (unary relationship):**



- No special handling, similar rules as we have for the binary relationships, depending on the functionalities.

Outline

- Data, Metadata, Relationships and Ontologies
- Introduction to Data Modeling (E/R Diagram)
- The Relational Database Model
- Normalization
- Introduction to SQL (DDL, DML)

Normalization

- How to design a relational database schema?
  - Iterative design process:
    - Informal description: Requirements Specification
    - Conceptual design: E/R-Diagram
    - Relational DB-Design: Relational schema
  - In this chapter:
    Normalization theory as formal principle for the relational schema design
  - Central queries to be solved:
    - How can we represent objects and their relationships in the relational model?
    - How can we valuate the quality of a database schema, or in other words, how can we distinguish between a “good” and a “bad” database schema?

Normalization

- Motivation for Normalization

Supplier

SID	SNAME	SCity	State	Product	Price
103	Smith	NYC	NY	TV	$1200
103	Smith	NYC	NY	USB Cable	$15
103	Smith	NYC	NY	Projector	$800
…	…	…		…	…
762	Lee	Boston	MA	Projector	$750
762	Lee	Boston	MA	TV	$1300

- From the example we can observe the following Redundancies:
  - Tuples with same SID value also have same SNAME, SCITY and State value.
  - Tuples with same SCITY value have the same State value (even if they differ in SID)

Normalization

- What is the problem with redundancies?

Supplier

SID	SNAME	SCity	State	Product	Price
103	Smith	NYC	NY	TV	$1200
103	Smith	NYC	NY	USB Cable	$15
103	Smith	NYC	NY	Projector	$800
…	…	…		…	…
762	Lee	Boston	MA	Projector	$750
762	Lee	Boston	MA	TV	$1300

- One problem is that redundancies waste memory space.
- But the main problem are **anomalies.**

We know three types of anomalies:

- Update – anomaly: Inconsistent entries if we would update the address of a supplier in just one tuple. How can this happen?
- Insert – anomaly: Insertion of tuples that are inconsistent with existing ones, e.g. in Address. Insertion of a new supplier requires to specify a product.
- Delete – anomaly: If all products associated with a supplier are deleted, the address of the supplier will be deleted as well.

Normalization

- What can we do to avoid redundancies/anomalies?
  - Decomposition of the relation schemas into reasonable relations (entities)
  - For example:

    Supplier (<u>SID</u>, SNAME, SCity, State, <u>Product</u>, Price)

    ⇩

    SupplierAdr (<u>SID</u>, SNAME, SCity)
    City (<u>SCity</u>, State)
    Offer (<u>SID</u>, <u>Product</u>, Price)
  - Advantage: no redundancy and no anomalies anymore
  - Disadvantage: Searching the States of the suppliers of a specific product needs 2 join operations, which is expensive in terms of processing time
    => the user has to wait longer for any answer.

Normalization

- Definition: Functional Dependent

Given:

- A relation schema R
- A relation D (covering all possible instantiations) of schema R
- X, Y: Two sets of attributes of R ($X,Y \subseteq R$)

Definition:
Y is functional dependent of X denoted by "X → Y", iff $\forall$ t,r $\in$ D: t.X=r.X => t.Y=r.Y

In other words: For each set of values in X there exist exactly one set of values in Y.

Examples:	SID → SNAME	SID, Product → SNAME
	SID → SCity	SID, Product → SCity
	SID → State	SID, Product → Price

Normalization

- Normalization:

  General aim: Decompose relational schemas such that no attribute set Y is functional dependend from another attribute set, except from the complete key.

  - By stepwise decomposing the relational schemas in a database schema, we can bring the database schema in different states called normal forms.
  - This process is called normalization.

- Normal Forms:
  - 1. Normal form (1NF)
  - 2. Normal form (2NF)
  - 3. Normal form (3NF)
  - Boyce-Codd-Normal form
  - 4. Normal form

implies

implies

implies

implies

relation schema decomposition

3NF most important in praxis

Normalization

- 1. Normal Form (1NF):
  - No restriction concerning functional dependencies (FDs)
  - A relation scheme is in 1NF, if all attribute values are atomic (indivisible).

A	B	C	D
1	2	3 4	4 5
2	3	3	4
3	3	4 6	5 7

Non-atomar values, assuming dom(D) = INT

Note: Non-atomic values not allowed in (strict) relational databases anyway.

→ All relations in relational databases are in 1NF

Normalization

- 2. Normal Form (2NF):
  - Avoids that attributes are functional dependent on a part of the key.
  - Example:

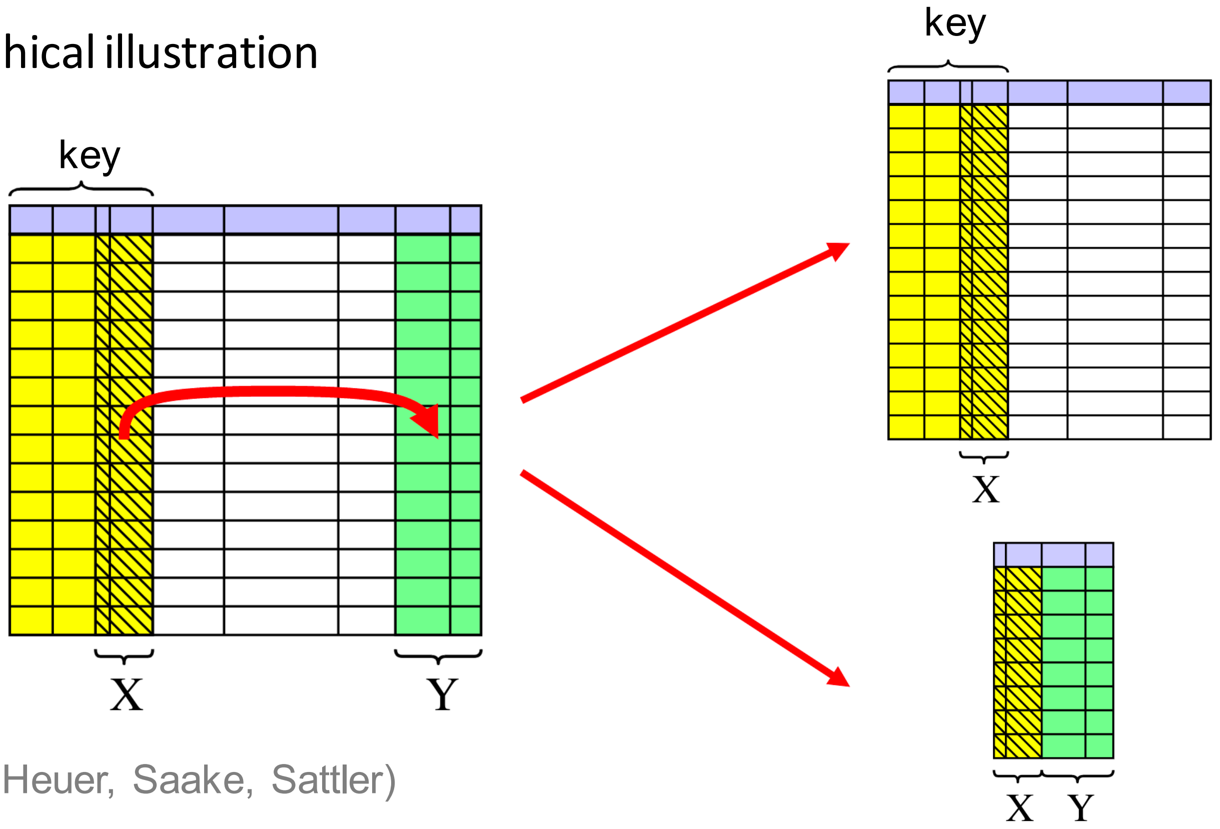
FDs

OrderID	CName	CCity	CState	Product	Price
103	Smith	NYC	NY	TV	$1200
103	Smith	NYC	NY	USB Cable	$15
103	Smith	NYC	NY	Projector	$800
…	…	…		…	…
762	Lee	Boston	MA	Projector	$750
762	Lee	Boston	MA	TV	$1300

- Key: {Order, Product}, FDs: Order → CName, Order → CCity, Order → CState, …
- Consequence: Information in dependent attributes (e.g. CName) have to be repeated multiple times → redundancy

Normalization

- 2. Normal Form (2NF): (cont.)
  - Graphical illustration



(Heuer, Saake, Sattler)

Normalization

- 3. Normal Form (3NF):
  - Avoids that attributes are functional dependent on non-key attributes.
  - Example:

FD

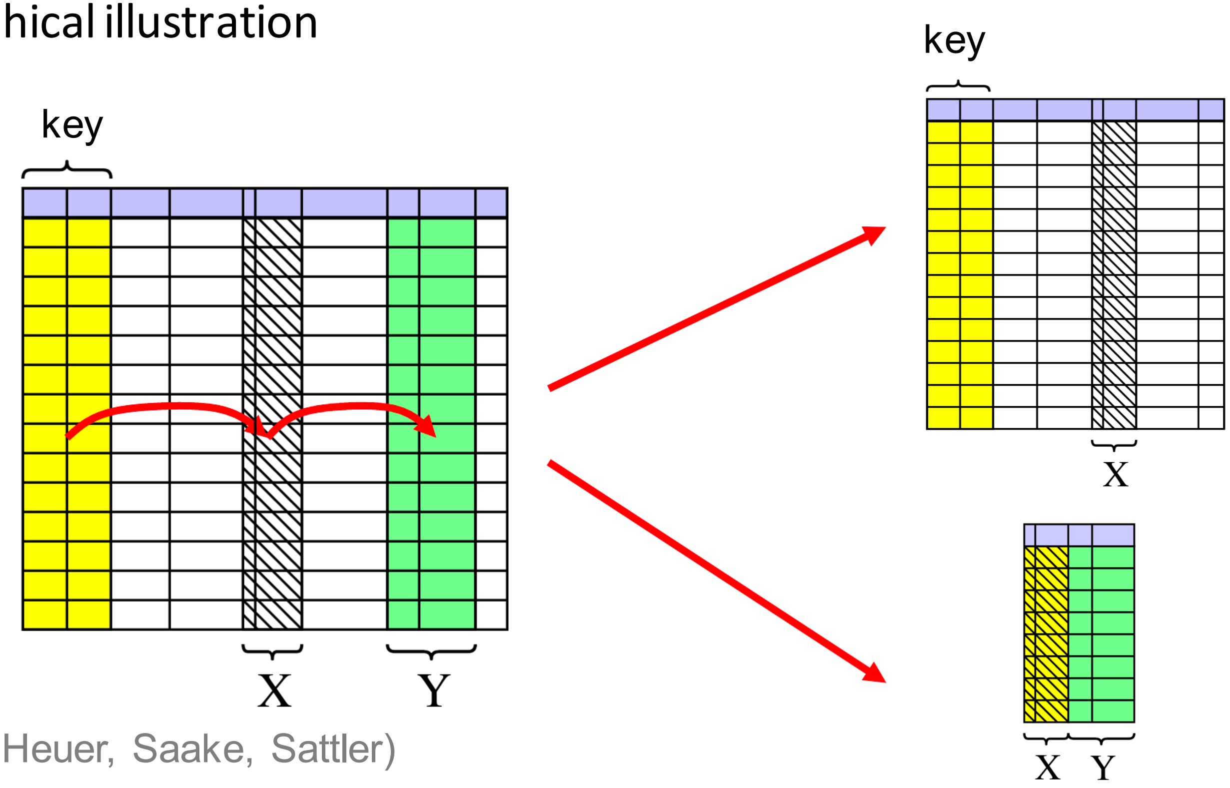
DeliveryAdr

OrderID	CName	CCity	CState
103	Smith	NYC	NY
104	Collins	LA	CA
105	Smith	SF	CA
…	…	…	
762	Lee	Boston	MA
763	Lee	Boston	MA

  - Redundancy: CState values are repeated multiple times
  - What kind of anomalies?

Normalization

- 3. Normal Form (3NF): (cont.)
  - Graphical illustration



(Heuer, Saake, Sattler)

Normalization

- 3. Normal Form (3NF): (cont.)
  - Example: final result of Normalization (in 3NF)

OrderID	CName	CCity	CState	Product	Price
103	Smith	NYC	NY	TV	$1200
103	Smith	NYC	NY	USB Cable	$15
103	Smith	NYC	NY	Projector	$800
…	…	…		…	…
762	Lee	Boston	MA	Projector	$750
762	Lee	Boston	MA	TV	$1300

OrderID	Product	Price
103	TV	$1200
103	USB Cable	$15
103	Projector	$800
…	…	…
762	Projector	$750
762	TV	$1300

OrderID	CName	CCity
103	Smith	NYC
104	Collins	LA
105	Smith	SF
…	…	…
762	Lee	Boston
763	Lee	Boston

CCity	CState
NYC	NY
LA	CA
SF	CA
…	
Boston	MA

- Most applications require a database schema that is in 3. Normal Form (3NF)

Normalization

- Normalization decomposes a relation into multiple relations
- Why can't we just decompose a relation arbitrarily?
- Relation decomposition has to be lossless:
  - The decomposed relations (together) need to represent the same information as the original relation.
  - We must be able to reconstruct the original relation from the decomposed relations.

  Note: Here we consider relations (i.e. tables with content) not only relation schemas!!!
- Relation decomposition has to be FD conservative:
  - Each functional dependencies in the original relation have to be conserved in at least one of the resulting relation after the decomposition. Why?

Outline

- Data, Metadata, Relationships and Ontologies
- Introduction to Data Modeling (E/R Diagram)
- The Relational Database Model
- Normalization
- Introduction to SQL (DDL, DML)

Introduction to SQL

- SQL = **S**tructured **Q**uery **L**anguage
  - Most important query language for relational databases.
  - Operations/Instructions are mainly based on relational algebra and relational calculus (theoretical query concepts for relational database models).
  - SQL is a declarative query language.
  - Users specify WHAT they want to extract from the database (DB).
  - Not HOW the information is extracted from DB (in contrast to procedural languages, e.g. Java, Python, C,…

  ➔ logical – physical data independency, i.e. independency from the physical layer of the data management.

  - Two main categories of SQL statements:
    - DDL (Data Definition Language) used to define the relational schema (generate/delete/update tables)
    - DML (Data Manipulation Language) used to query and manipulate the data in the relation (insert/delete/update data, query data, data aggregation, etc.)

Introduction to SQL

- History of SQL:
  - Beginning of 70th — IBM introduced DBMS-prototype “System R” with query language “SEQUEL” (Structured English Query Language), later called “SQL”
  - Beginning of 80th — 1st Commercial relational database management system SQL/DS
  - Later — Many other commercial RDBMS (Relational Database Management Systems), e.g. ORACLE, Informix, …

  Since RDBMS became popular, standandardization has become necassary

  - 1986 — First SQL-norm defined by ANSI (American National Standards Insitute)
  - 1989 — Revision of first SQL-norm
  - 1992 — SQL-92 standard (widely extended SQL standard), also called SQL 2
  - End 90th — SQL 3 standard
  - Since 2000 — Further standard revisions: SQL:2003, SQL:2006, SQL:2008, SQL:2011

  In the following: SQL concepts based on SQL-92 (and still complies with all standards up to SQL:2011)

Introduction to SQL

- Data Types in SQL
  - Three fundamental data types: Numbers, character sequences, and dates
  - **Character sequences**:
    - CHARACTER(n) (or CHAR(n)) — character sequence with fix number n of characters. Filled up with "space" on demand.
    - VARCHAR(n) — character sequence with variable number of characters up to n characters.
  - **Numbers**:
    - NUMERIC(p,s) — Number with precision p and scale s, e.g. numeric(5,2) is a number with 5 digits where 3 are before, and 2 after the decimal point (eamples: 437.32, 231.34, …) (SQL-Server: Bytes=5-17, Range: -10E38 – 10E38-1, depending on precision (scale))
    - INTEGER (or INT) — Number without decimal point. (SQL-Server: Bytes: 4, Range -2 147 483 648 – 2 147 483 647)
    - FLOAT — Number with float precision SQL Server: Bytes: 4, Range -3.40E38 to -1.18E-38 and 1.18E-38 to 3.40E38)

Introduction to SQL

- Data Types in SQL
  - Three fundamental data types: Numbers, character sequences, and dates (cont.)
  - **dates**:
    - DATES — date values in the form YYYY-MM-DD (e.g. 2011-05-03)
    - TIME — time values in the form HH:MM:SS (e.g. 15:51:36). The granularity of the time value is usually a tick (100 nanoseconds).
    - TIMESTAMP — combination of date and time in the form YYYY-MM-DD HH:MM:SS
  - **Bit strings:**
    - BIT(n) — an array of *n* bits
    - BIT VARYING(n) — an array of up to *n* bits

Many further data types: see http://www.w3schools.com/sql/sql_datatypes_general.asp

Outline

- Introduction to SQL
  - DDL
  - DML

Data Definition Language (DDL)

- Data Definition Language (DDL) is part of SQL
- Composes commands to manipulate the relational schema (relations a.k.a. tables) of a relational database.
- Fundamental commands of the DDL:
  - CREATE X   Creates a new entity X (X∈{table (=relation), index, view, ... other types of entities})
  - ALTER X   Updates existing entities in a database
  - DROP X   Deletes an existing entity in a database
  - TRUNCATE X Deletes all entries of a table X and frees memory allocated for the table (relation)
  - RENAME X   Updates names (titles) of existing database entities

In the following we will concentrate on the two most important commands: CREATE TABLE … and ALTER TABLE.. that are used to build (realize) a relational schema in a relational dbms.

Data Definition Language (DDL)

- **Defining a new table (relation)**
- A new (initially empty) table (relation) will be created by the SQL command

```
CREATE TABLE TableName
(       attr.1    domain  [constraints (opt.)],
        attr. 2   domain  [constraints (opt.)],
        ...
);
```

- Example:
  Defining the relation student (SID:INT,SName:VARCHAR(10),SAge:INT)

```
CREATE TABLE student
(       SID       INT             not null,
        SName     VARCHAR(10)     not null,
        SAge      INT
);
```

constraint "not null" means that each entry (tuple) in the relation has to have a valid value for this attribute, i.e. not a "null" value which means a missing value. Usually used also as constraint for key attributes.

Data Definition Language (DDL)

- Constraints (a.k.a. integrity checks) associated with attributes:
  - Constraints for single attributes (directly after attribute type (domain)):
    - not null: attribute must not undefined, i.e. value ≠ null
    - primary key: attribute serves as primary key
      (only if primary key does not cover multiple attributes)
    - unique: attribute is a key candidate
    - references t1(a1): Reference to attribute a1 of another table t1 (foreign key)
    - default v1: if attribute value is not set, it is set to value v1
    - check f: Formular f will be evaluated as soon as a new entry is inserted in the table, e.g. check A <= 100.
    - Example:

```
CREATE TABLE Student
( SID      INT              primary key,
SName      VARCHAR(20)      not null,
SGDat      DATE,
SAdr       INT              references Adress(AdrID)
);
```

Data Definition Language (DDL)

- Constraints (a.k.a. integrity checks) associated with attributes:
  - Constraints for multiple attributes (as additional rows in the attribute specification block):
    - primary key (A1,A2,...,Ak): compound primary key
    - unique (A1,A2,...,Ak): compound key candidate
    - foreign key (A1,A2,...) references t1(B1,B2,...): Reference to compound key in relation t1.
      Note: if specification of (B1,B2,...) is missing, the system automatically uses (A1,A2,...) for t1 instead.
    - check f: Formular f will be evaluated, whereas f may involve multiple attributes, e.g. check A1 <= A2.

    - Example:

```
CREATE TABLE Student
(SName          VARCHAR(20)     not null,
SFirstName      VARCHAR(20)     not null,
SGDat           DATE            not null,
CourseTitel     VARCHAR(15),
Term            VARCHAR(10),
primary key (SName, SFirstName,SGDat),
foreign key (CourseTitle,Term) references Course on delete set null
);
```

Data Definition Language (DDL)

- **Update a table (relation)**
- The schema of a table (relation) will be updated by the SQL command
  ALTER TABLE *TableName*
  add/modify (attr domain [constr.(opt.)]);
- Examples:
  Updating the relation student (SID:INT,SName:VARCHAR(10),SAge:INT)
  - **ALTER TABLE** student
    add (SCourse VARCHAR(20)); — Adds a new attribute "SCourse" to student
  - **ALTER TABLE** student
    modify (SName VARCHAR(15)); — Updates the type of attribute SName in student

- **Delete a table (relation)**
  - **DROP TABLE** student;

Outline

- Introduction to SQL
  - DDL
  - DML

Data Definition Language (DML)

- Data Manipulation Language (DML) is part of SQL
- Composes commands to manipulate/retrieve the data in a relation (table) of a relational database.
- Fundamental commands of the DML:
  - SELECT Retrieves data from a database
  - INSERT Inserts data into a database
  - UPDATE Updates existing data within a table (relation)
  - DELETE Deletes all data (not schema information) in a table (relation)

In the following we will concentrate on the three most important commands: INSERT…, UPDATE, and SELECT...

Data Definition Language (DML)

- **Insert** data into a database:
  - Given database schema: student (SID:INT,SName:VARCHAR(10),SAge:INT)
  - INSERT INTO student
    values (123, Smith, 31);
  - or by SELECT statement (SELECT will be introduced later!)
    INSERT INTO Lecturer
    (SELECT DISTINCT FName, FAdr, 0 FROM Faculty)
- **Update** data in a database:
  - UPDATE Lecturer L
    SET L.salary = 6000
    WHERE LNr = 273
  - Result: For all tuples in table (relation) Lecturer having LNr = 273, the attribute salary will be updated to the value 6000.

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Basic scheme:

    SELECT … <list of attribute names, or „*“ for all attributes> — What information to extract (query)

    FROM … <list of relation names> — From which relations (tables)

    WHERE … <set of constraints (logically connected)> — Constraints on the information to be extracted

  - Example: Given DB schema:
    Student (SID:INT, SName:VARCHAR(10), SAge:INT, SCourse:VARCHAR(20))
    Course (CourseName:VARCHAR(20), Lecturer:VARCHAR(15), Department:INT)

    - SELECT SName, SAge
      FROM Student
      WHERE SCourse = „SDB“

SName	SAge
Smith	31
Collin	28
…	…

    - SELECT s.SName, c.Lecturer
      FROM Student s, Course c
      WHERE s.SCourse = c.CourseName and Lecturer = „Renz“

Data Definition Language (DML)

- **Retrieve** data from a database:
  - SELECT clause: For the list of attributes A1, A2, … one can use:
    - An attribute name of a relation (but the relation has to be specified in the FROM statement).
    - A skalar expression concatenating attributes and constant scalars by means of arithmetic operations.
    - Aggregation functions (will be introduced later)
    - Expression of the form A_1 as A_2 to rename attribute names (renaming only affects the result relation, not the input relations!)
  - Example:

SELECT pname, price*126.24 as JYen, price*EURUSDExchangeRate as USD, „US$“ as currency

FROM products, currencies ….

pname	JYen	USD	Currency
Pen	230,46	2.08	US$
Paper	40,13	0.36	US$
Nail	23,47	0.21	US$
…	…	…	…

Data Definition Language (DML)

- **Retrieve** data from a database:
  - **FROM** clause:
    - At least contains one relation in the form of R_1
    - If the „FROM“ clause includes more entries
      FROM R_1, R_2, …
      the cartesian product
      $R_1 \times R_2 \times \ldots$
      will be built.
    - If two or more relations share the same attribute name, the attributes used in the SELECT and WHERE terms are becoming ambigues.
      → this can be solved by explicite attribute-relation assignment, e.g. SELECT Students.name, Lecturer.name
    - If relation names are quite long it is helpful to rename the relations within the query expression (alias names):

```
SELECT s.name, l.name, …
FROM Students s, Lecturer l
WHERE …
```

alias names also useful for self-joins!

  - This can be combined also with the „*“ term: (Note: „*“ means the list of all attributes of a relation)

```
SELECT     s.*, l.name AS Lecturername, …
FROM       Students s, Lecturer l
WHERE      …
```

Data Definition Language (DML)

- **Retrieve** data from a database:
  - **WHERE** clause:
    - Contains exactly one logical predicate Φ (where Φ is a function that returns a boolean value *true* or *false*)
    - The boolean predicate consists of:
      - Comparison between attribute values and constant
      - Comparison between different attributes
      - Comparison operators: = , < , <= , > , >= , <>
      - Test of undefined value: e.g. A_1 IS NULL / IS NOT NULL
      - Inexact string comparison: e.g. A_1 LIKE „Database%“
        The "%" sign is used to define wildcards (missing letters) both before and after the string pattern.
      - Set inclusion terms: e.g. A_1 IN (2, 3, 5, 7, 11, 13)
    - Within a predicate: scalar expressions
      - Concatenating numerical values/attributes with +, -, *, /.
      - String expressions: e.g.
        str1 || str2 → concatenation of str1 and str2
      - Common usage of parentheses (..).
    - Single predicates can be formed into more complex ones with AND, OR, NOT.

Data Definition Language (DML)

- **Retrieve** data from a database:
  - **Joins**:
    - Usually by means of selection over cartesian product, e.g.:
      - SQL: SELECT * FROM A, B WHERE A.x = B.y
    - Example:
      SELECT *
      FROM Customer C, Order O
      WHERE C.name = O.name

Customer

name	x	y
Collins	M	10
Smith	M	20
Lewis	L	0

Order

name	z
Collins	B
Collins	R
Smith	B

Customer ⋈ Order

name	x	y	z
Collins	M	10	B
Collins	M	10	R
Smith	M	20	B

Data Definition Language (DML)

- **Retrieve** data from a database:
  - UNION, INTERSECT, EXCEPT:
    - Set operations usually combines the result of two SQL queries, e.g.

      SELECT * FROM A WHERE name = „Ali“
      UNION
      SELECT * FROM B WHERE name = „Arie“

    - Modern DBMS also allow to use set operations within the „FROM“-clause:

      SELECT * FROM A union B WHERE ...

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Nested SELECT queries:
    - **Nested query in SELECT clause:**

```
SELECT PID, PName,      (SELECT sum(paystub)
                         FROM Salary s
                         WHERE s.PID = p.PID)
FROM Professor p;
```

    - **Nested query in FROM clause:**

```
SELECT PID, PName
FROM (SELECT * FROM Professor p WHERE p.paystub<3000) prof
WHERE prof.PAge > 45
```

    - **Nested query in WHERE clause:**

```
SELECT *
FROM Student s
WHERE
   s.GebDat<(SELECT min(p.GebDat) FROM Professor p)
```

```
SELECT *
FROM Student s
WHERE
   exists(SELECT * FROM Prof p WHERE s.Age>p.Age)
```

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Sorting and Aggregation:
  - SQL supports data analysis by providing sorting and aggregation operations
  - Extend SQL-query statements
  - Syntax:
    SELECT ... ← Aggregation, e.g. sum, avg, etc.
    FROM ...
    [WHERE ...]
    [GROUP BY A_1, A_2, ...] ← Grouping tuples for aggregation
    [HAVING ...] ← Constraints for grouping
    [ORDER BY ...] ← Sorting tuples

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Sorting:
    - In SQL with ORDER BY A_1, A_2, …
    - Is specified a t the end of the SQL query statement
    - The result of the SQL statement will be ordered by the attributes A_1, A_2, … (for multiple attributes → lexicographical order)

A	B
1	1
3	1
2	2
4	1
3	3

ORDER BY A, B

A	B
1	1
2	2
3	1
3	3
4	1

ORDER BY B, A

A	B
1	1
3	1
4	1
2	2
3	3

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Sorting:
    - Sorting in ascending order (default) or descending order.

      ORDER BY A_1, A_2, … [ASC/DESC]

    - Sorting can be applied only to attributes that are specified in the SELECT-clause !!!
    - Example:

SELECT * FROM MagicNumbers ORDER BY Value DESC

or

SELECT Value FROM MagicNumbers ORDER BY Value DESC

Not Possible !!!:

SELECT Name FROM MagicNumbers ORDER BY Value

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Aggregation
    - Aggregation over (sub-)sets of tuples
    - Aggregation operations within SELECT-clause
    - Aggregationfunctions in SQL:
      - COUNT number of tuples (values)
      - SUM sum of values of an attribute
      - AVG Average of the values of an attribute
      - MAX Maximum value of an attribute
      - MIN Minimum value of an attribute
    - Aggregation over ...
      - complete result relation
      - partitions of result relations (Partitions defined by the „GROUP BY“-clause

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Aggregation
    - Example

```
SELECT SUM(salary), AVG(salary)
FROM Lecturer
WHERE LAge < 40
```

    - Result of an aggrgation over a complete result relation is always just one tuple.
    - „NULL“ values will be ignored, even with the „count“ operator.

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Aggregation
    - Example

```
SELECT SUM(salary), AVG(salary)
FROM Lecturer
WHERE LAge < 40

SELECT *
FROM Product
WHERE Price < (SELECT avg(Price) FROM Product)
```

- Result of an aggrgation over a complete result relation is always just one tuple.
- „NULL“ values will be ignored, even with the „count“ operator.

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Grouping
    - Decomposing the result into partitions of result tuples
    - Goal: Aggregate information within groups (partitions)
    - Example: Overall salary and number of lecturers within a department

Lecturer | Aggregation

LNr	Name	Department	Salary	∑ Salary	Count
001	Collins	046	2000	6300	3
002	Smith	046	2500		
003	Lewis	046	1800		
004	Li	398	2500	4200	2
005	Jones	398	1700		

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Grouping
    - Decomposing the result into partitions of result tuples
    - Goal: Aggregate information within groups (partitions)
    - Example: Overall salary and number of lecturers within a department

Not possible in SQL …

..the result is not a relation.

Lecturer

LNr	Name	De			
001	Collins				
002	Smith	046	2500		
003	Lewis	046	1800		
004	Li	398	2500	4200	2
005	Jones	398	1700		

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Grouping
    - Decomposing the result into partitions of result tuples
    - Goal: Aggregate information within groups (partitions)
    - Example: Overall salary and number of lecturers within a department

LNr	Name	Department	Salary
001	Collins	046	2000
002	Smith	046	2500
003	Lewis	046	1800
004	Li	398	2500
005	Jones	398	1700

SELECT Department, sum(Salary), count(*)
FROM Lecturer
GROUP BY Department

Department	sum(Salary)	count(*)
046	6300	3
398	4200	2

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Grouping
    - The SELECT clause only may have attributes that
      - appear in the GROUP BY – clause (arithmetic expressions included) or
      - aggregation functions (also among other attributes, e.g. *)

Example: not allowed!!!

SELECT LNr, Department, sum(Salary)
FROM Lecturer
GROUP BY Department

LNr	Name	Department	Salary
001	Collins	046	2000
002	Smith	046	2500
003	Lewis	046	1800
004	Li	398	2500
005	Jones	398	1700

LNr	Department	sum(Salary)
~~001, 002, 003~~	046	6300
~~004, 005~~	398	4200

Data Definition Language (DML)

- **Retrieve** data from a database:
  - Grouping
    - The SELECT clause only may have attributes that
      - appear in the GROUP BY – clause (arithmetic expressions included) or
      - aggregation functions (also among other attributes, e.g. *)

Example:

```
SELECT Year, Department, sum(Salary)
FROM Lecturer
GROUP BY Year, Department
```

LNr	Name	Year	Department	Salary
001	Collins	2006	046	2000
002	Smith	2008	046	2500
003	Lewis	2006	046	1800
004	Li	2006	398	2500
005	Jones	2006	398	1700

Year	Department	sum(Salary)
2006	046	3800
2006	398	4200
2008	046	2500

Data Definition Language (DML)

- **Retrieve** data from a database:
  - HAVING
    - Specifies constraints among aggregates built by GROUP BY
    - Motivative Example:
      Retrieve overall salary of each department that has at least 5 Lecturers

```
SELECT Department, sum(salary)
FROM Lecturer
WHERE count(*)>=5          not possible !!!!
GROUP BY Department
```

Data Definition Language (DML)

- **Retrieve** data from a database:
  - HAVING
    - Specifies constraints among aggregates built by GROUP BY
    - Motivative Example:
      Retrieve overall salary of each department that has at least 5 Lecturers

```
SELECT Department, sum(salary)
FROM Lecturer
GROUP BY Department
HAVING count(*) >= 5
```

correct

    - Reason: Grouping will be processed after the SELECT .. FROM...WHERE operation.