

B0B36DBS, BD6B36DBS: **Database Systems**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/>

Lecture 1

# **Conceptual Modeling**

**Martin Svoboda**

[martin.svoboda@fel.cvut.cz](mailto:martin.svoboda@fel.cvut.cz)

18. 2. 2020

**Czech Technical University in Prague, Faculty of Electrical Engineering**

# Lecture Outline

- **Introduction to database systems**
  - What is a database?
  - Basic terminology
- **Conceptual database modeling**
  - ER – Entity-Relationship Model
  - UML – Unified Modeling Language

# Basic Terminology

- **Database (DB)**

- Logically organized collection of related data
  - Self-describing, metadata stored together with data
    - Data + schema + integrity constraints + ...

- **Database management system (DBMS)**

- Software system enabling access to a database
  - Provides mechanisms to ensure security, reliability, concurrency, integrity of stored data, ...

- **Database system**

- Information system

- Database, DBMS, hardware, people, processes, ...

*"Everything around DB!"*

# Motivation for Databases

- Why database systems?
  - **Data sharing and reusability**
    - Consistency, correctness, compactness...
    - Concurrency, isolation, transactions, ...
  - **Unified interface and languages**
    - Data definition and manipulation
  - **Information security**
    - User authentication, access authorization, ...
  - **Administration and maintenance**
    - Replication, backup, recovery, migration, tuning, ...

# Brief History

- Database models and systems
  - **Network** and **hierarchical** databases
  - **Relational** databases
  - **Object** and **object-relational** databases
  - **XML** databases
  - **NoSQL** databases
    - Key-value stores, document-oriented, graph databases, ...
  - Stream, active, deductive, spatial, temporal, probabilistic, real-time, in-memory, embedded, ...
- Still **evolving area** with plenty of challenges

# Brief History

- **Why so many different database systems?**
  - Different contexts
    - OLTP, OLAP, Cloud computing, Big data, ...
  - Different requirements
    - Performance, scalability, consistency, availability, ...
  - Different architectures
    - Centralized, distributed, federated, ...
  - **Different forms of data**
    - Relations, objects, graphs, ...
    - Semi-structured, unstructured data, texts, ...
    - Multimedia, web

# Database Modeling

- **Process of database design**

- One vague sentence at the beginning...
  - ... a fully working system at the end
- Understanding and **modeling** the reality
- **Organizing** the acquired information
- Balancing the identified requirements
- Creating a **suitable database schema**

- **Who are stakeholders?**

- **Stakeholder** is any person who is relevant for our application
    - E.g. **users, investors, owners, domain experts, etc.**

# Layers of Database Modeling

- **Conceptual layer**

- Models a part of the reality (problem domain) relevant for a database application, i.e. identifies and describes real-world entities and relationships between them
- Conceptual models such as ER or UML

- **Logical layer**

- Specifies how conceptual components are represented in database structures
- Logical models such as relational, object-relational, graph, ...

- **Physical layer**

- Specifies how logical database structures are implemented in a specific technical environment
- Data files, index structures (e.g. B<sup>+</sup> trees), etc.

# **Conceptual Database Modeling**

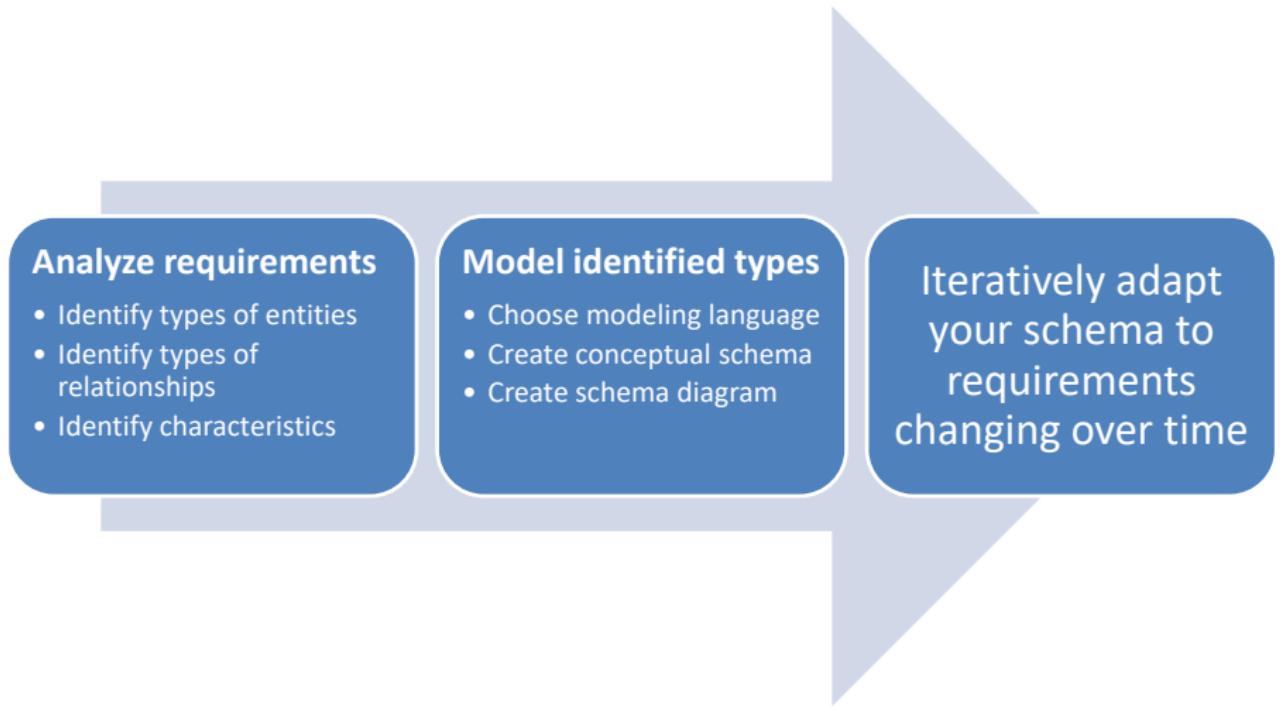
# Conceptual Database Modeling

- Conceptual modeling
  - **Process of creating a conceptual schema** of a given problem domain
    - In a selected **modeling language**
    - And on the basis of given requirements
  - **Multiple conceptual schemas** are often needed
    - Each schema describes a given database application (applications) from a **different point of view**
    - Even different conceptual models may be needed
  - We only focus on **conceptual data viewpoint**

# Basic Terminology

- **Model** = modeling language
  - Set of constructs you can use to express something
  - UML model = {class, attribute, association}
  - Relational model = {relational schema, attribute}
- **Schema** = modeling language expression
  - Instance of a model
  - Relational schema = {Person(name, email)}
- **Diagram** = schema visualization

# Conceptual Modeling Process



# Requirement Analysis (Step 1)

- Step 1 of conceptual modeling
  - Start with requirements of different stakeholders
    - Usually expressed in a natural language
    - Meetings, discussions, inquiries, ...
  - Identify important...
    - types of real-world entities,
    - their characteristics,
    - types of relationships between them, and
    - their characteristics
  - ... and deal with ambiguities

# Identification of Entities (Step 1.1)

- Example
  - Try to identify all types of entities:

Our environment consists of persons which may have other persons as their colleagues. A person can also be a member of several research teams. And, they can work on various research projects. A team consists of persons which mutually cooperate. Each team has a leader who must be an academic professor (assistant, associate or full). A team acts as an individual entity which can cooperate with other teams. Usually, it is formally part of an official institution, e.g., a university department. A project consists of persons working on a project but only as research team members.

# Identification of Entities (Step 1.1)

- Example

Our environment consists of persons which may have other persons as their colleagues. A person can also be a member of several research teams. And, they can work on various research projects. A team consists of persons which mutually cooperate. Each team has a leader who must be an academic professor (assistant, associate or full). A team acts as an individual entity which can cooperate with other teams. Usually, it is formally part of an official institution, e.g., a university department. A project consists of persons working on a project but only as research team members.

- Identified entity types

- Person
- Team
- Project
- Professor
  - Assistant Professor
  - Associate Professor
  - Full Professor
- Institution
- Department

# Identification of Relationships (Step 1.2)

- Example
  - Try to identify all types of relationships:

Our environment consists of persons which may have other persons as their colleagues. A person can also be a member of several research teams. And, they can work on various research projects. A team consists of persons which mutually cooperate. Each team has a leader who must be an academic professor (assistant, associate or full). A team acts as an individual entity which can cooperate with other teams. Usually, it is formally part of an official institution, e.g., a university department. A project consists of persons working on a project but only as research team members.

# Identification of Relationships (Step 1.2)

- Example

Our environment consists of persons which may have other persons as their colleagues. A person can also be a member of several research teams. And, they (person) can work on various research projects. A team consists of persons which mutually cooperate. Each team has a leader who must be an academic professor (assistant, associate or full). A team acts as an individual entity which can cooperate with other teams. Usually, it (team) is formally part of an official institution, e.g., a university department. A project consists of persons working on a project but only as research team members.

- Relationship types

- Person is colleague of Person
- Person is member of Team
- Person works on Project
- Team consists of Person
- Team has leader Professor
- Team cooperates with Team
- Team is part of Institution
- Project consists of Person who is a member of Team

# Identification of Characteristics (Step 1.3)

- Example
  - Try to identify characteristics of persons:

Each person has a name and is identified by a personal number. A person can be called to their phone numbers. We need to know at least one phone number. We also need to send them emails.

# Identification of Characteristics (Step 1.3)

- Example

Each person has a name and is identified by a personal number. A person can be called to their phone numbers. We need to know at least one phone number. We also need to send them emails.

- Person characteristics

- **Personal number**
- **Name**
- One or more **phone numbers**
- **Email**

# Identification of Characteristics (Step 1.3)

- Example
  - Try to identify characteristics of memberships:

We need to know when a person became a member of a project and when they finished their membership.

# Identification of Characteristics (Step 1.3)

- Example

We need to know when a person became a member of a project and when they finished their membership.

- Identified membership characteristics
  - From
  - To

# Schema Creation (Step 2)

- Step 2 of conceptual modeling
  - **Model the identified types and characteristics using a suitable conceptual data model** (i.e. create a conceptual data schema) **and visualize it as a diagram**
  - Various modeling tools (so-called **Case Tools**) can be used, e.g.,
    - Commercial: Enterprise Architect, IBM Rational Rose, ...
    - Academic: eXolutio

# Modeling Language Selection (Step 2.1)

- **Which model should we choose?**
  - There are several available languages, each associated with a well-established visualization in diagrams
  - We will focus on...
    - **Unified Modeling Language (UML)** class diagrams
    - **Entity-Relationship model (ER)**
  - There are also others...
    - **Object Constraints Language (OCL)**
    - **Object-Role Model (ORM)**
    - **Web Ontology Language (OWL)**
    - Predicate Logic, Description Logic (DL)

# Conceptual Schema Creation (Step 2.2)

- **How to create a schema in a given language?**
  - Express identified types of entities, relationships and their characteristics using constructs offered by the selected conceptual modeling language
    - UML: **classes, associations, attributes**
    - ER: **entity types, relationship types, attributes**

# **Entity-Relationship Model (ER)**

# **Unified Modeling Language (UML)**

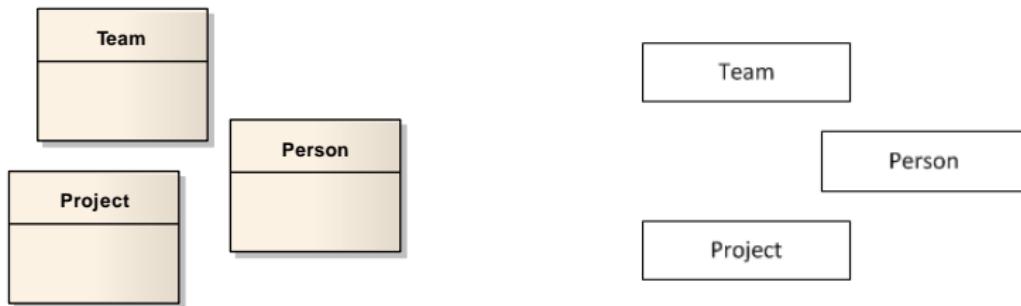
# ER and UML Modeling Languages

- ER
  - Not standardized, various notations and extensions (e.g. ISA hierarchy)
- UML
  - Family of models such as class diagrams, use case diagrams, state diagrams, ...
    - Standardized by the OMG (Object Management Group)
    - <http://www.omg.org/spec/UML/>
- Note that...
  - ER is more oriented to data design, UML to code design
  - Both ER and UML are used in practice, but UML has become more popular
  - ER constructs were incorporated to new versions of UML as well

# Types of Entities

Type of real-world entities

Persons, research teams and research projects.



UML

Class  
Name

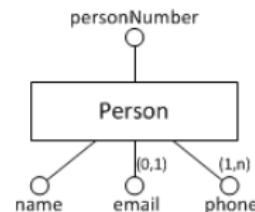
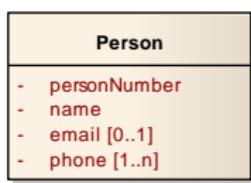
ER

Entity type  
Name

# Characteristics of Entities

## Attributes of a type of real-world entities

A person is characterized by their personal number, name, optional email address and one or more phone numbers.



UML

**Attribute of a class**  
Name and cardinality

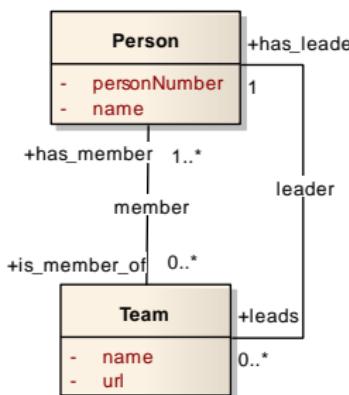
ER

**Attribute of an entity type**  
Name and cardinality

# Types of Relationships

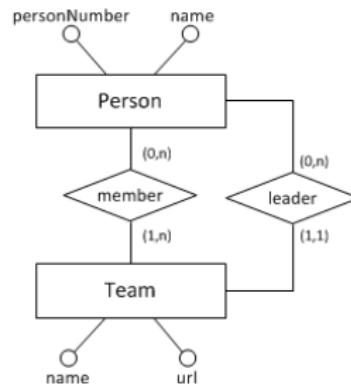
Type of a relationship between two real-world entities

A team has one or more members, a person can be a member of zero or more teams.  
A team has exactly one leader, a person can be a leader of zero or more teams.



UML

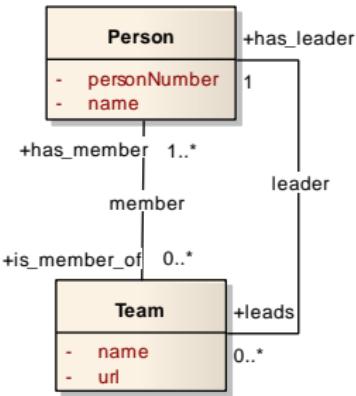
**Binary association:** name and two participants with names and cardinalities



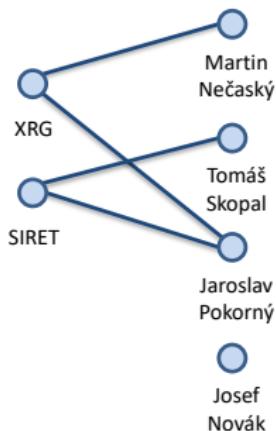
ER

**Binary relationship type:** name and two participants with cardinalities

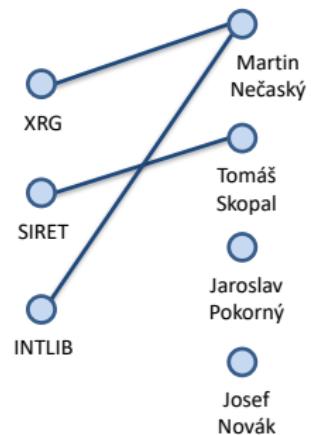
# Cardinalities in Relationships



Relationship  
**member**



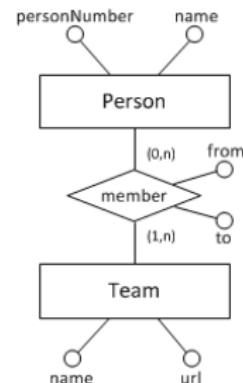
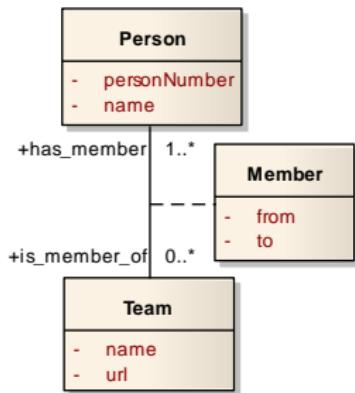
Relationship  
**leader**



# Characteristics of Relationships

Attributes of a type of relationship between real-world entities

A person is a team member within a given time interval



UML

**Attribute of a binary association class**  
Name and cardinality

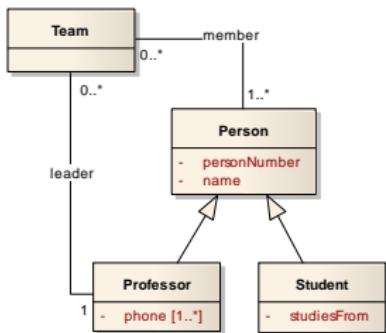
ER

**Attribute of a relationship type**  
Name and cardinality

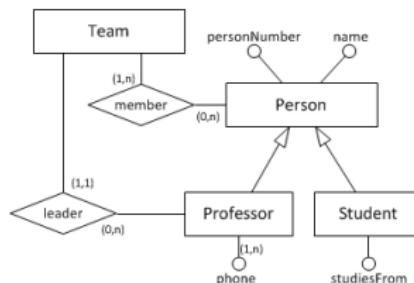
# Generalization / Specialization

Type of entities which is a specialization of another type

Each person has a personal number and name. A professor is a person which also has one or more phones and can lead teams. A student is a person which also has a date of study beginning.



UML



ER

**Generalization:** specific association with no name, roles and cardinalities

**ISA hierarchy:** specific relationship with no name and cardinalities

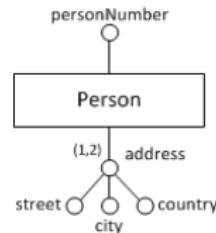
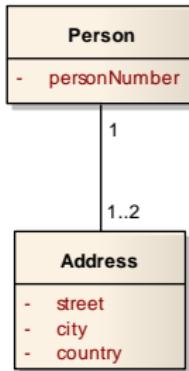
# Generalization / Specialization

- Note that...
  - Entity type can be a source for multiple hierarchies
  - Each entity type can have at most one generalization
- Additional constraints
  - **Covering constraint** (complete/partial)
    - Each entity must be of at least one specific type
      - I.e. each Person is a Professor or Student (or both)
  - **Disjointness constraint** (exclusive/overlapping)
    - Each entity must be of at most one specific type
      - I.e. there is no Student that would be a Professor at the same time

# Composite Attributes

Structured characteristics of real-world entity types

A person has one or two addresses comprising of a street, city and country.



UML

No specific construct  
Auxiliary **class**

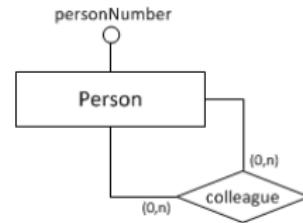
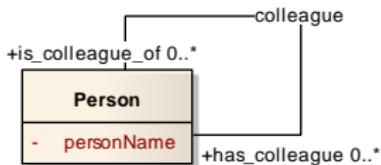
ER

**Composite attribute:** name, cardinality  
and sub-attributes

# Recursive Relationships

Type of a relationship between entities of the same type

A person has zero or more colleagues.



UML

Normal association  
with the same participants

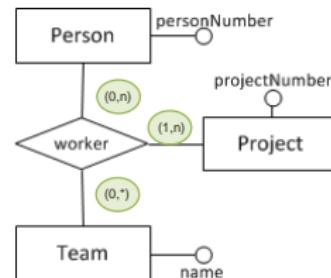
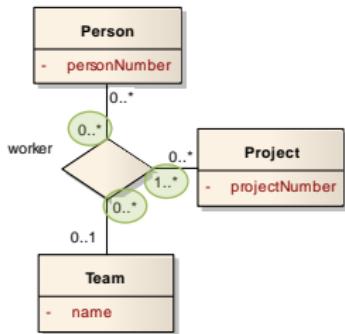
ER

Normal relationship type  
with the same participants

# N-ary Relationships

Type of a relationship between more than just two entities

A person works on a project but only as a team member.



UML

## N-ary association

Similar to a binary association but with three or more participants

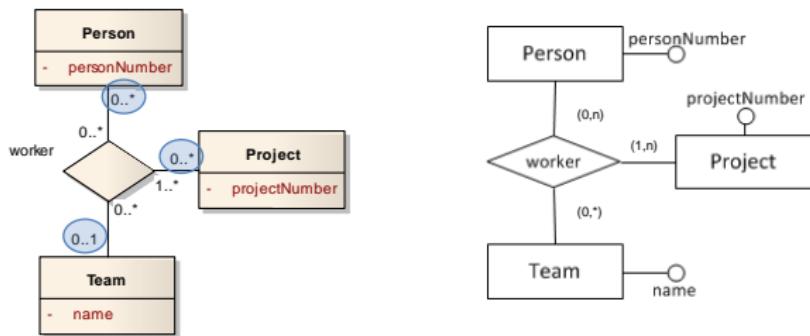
ER

## N-ary relationship type

Similar to a binary relationship type but with three or more participants

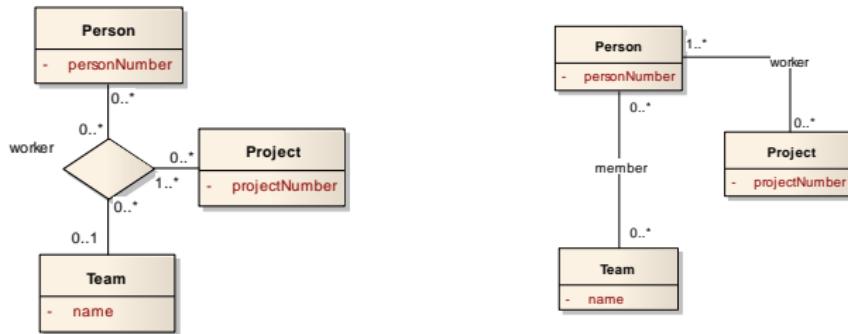
# N-ary Relationships

- Note that...
  - N-ary relationships can also have attributes
  - UML allows us to use **more expressive cardinalities**
    - E.g. a given combination of a particular person and project is related to zero or more teams through the given association
    - ...



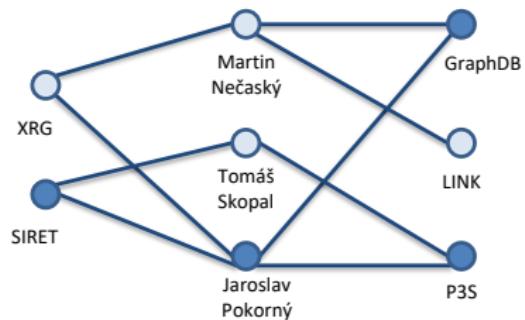
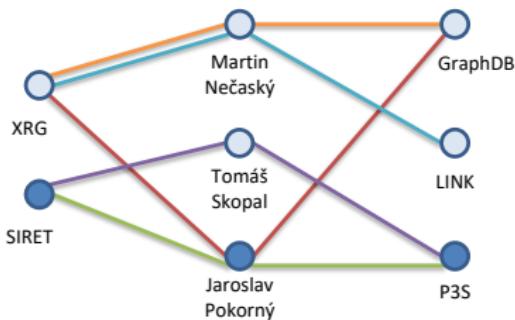
# N-ary Relationships

- Can n-ary relationships be replaced with binary?
- Which projects does *Jaroslav Pokorný* work on as a member of the *SIRET* research group?
- I.e. what is the difference between the following?



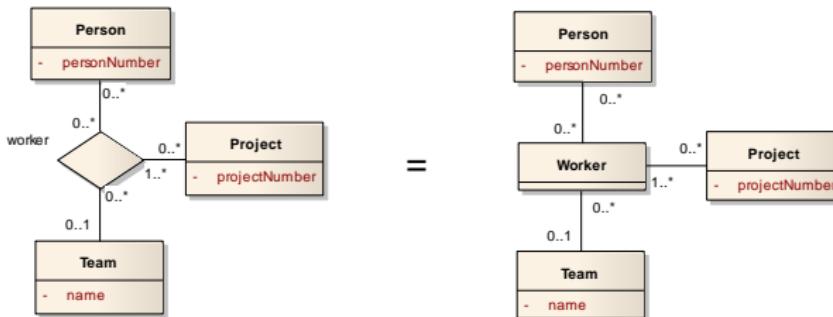
# N-ary Relationships

- Can n-ary relationships be replaced with binary?
- Which projects does *Jaroslav Pokorný* work on as a member of the *SIRET* research group?
- I.e. what is the difference between the following?



# N-ary Relationships

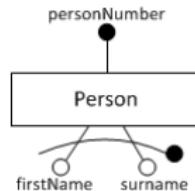
- Can n-ary relationships be replaced with binary?
- Yes, but in a different way...
- N-ary association = class + separate binary association for each of the original participants



# Identifiers

## Full identification of real-world entities

A person is identified either by their personal number or by a combination of their first name and surname.



UML

N/A

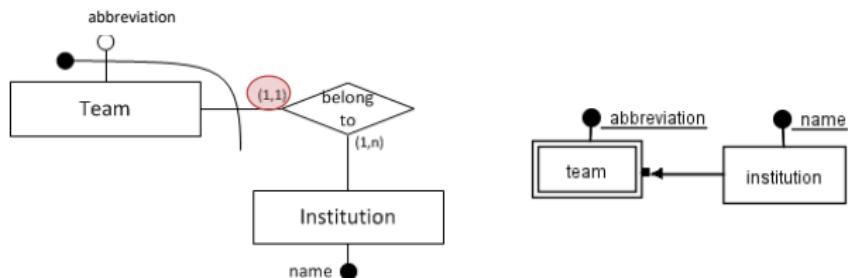
ER

Attribute or a group of attributes marked as an **identifier**

# Identifiers

Partial identification of real-world entities

A team is identified by a combination of its name and a name of its institution.



UML

N/A

ER

Attribute or a group of attributes marked as a partial **identifier**

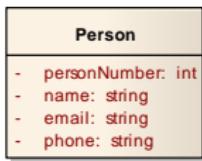
# Identifiers

- Note that...
  - **Each entity type must always be identifiable**
    - At least by a set of all its attributes if not specified explicitly
  - Partial identifiers create **identification dependencies**
    - **Only (1,1) cardinality is allowed** (makes a sense)!
- Entity types
  - **Strong entity type**
    - ... has **at least one (full) identifier**
  - **Weak entity type**
    - ... has **no (full) identifier**, and so at least one partial identifier
    - ... is both **existentially and identification dependent**

# Data Types

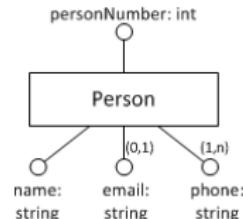
## Data type of attributes

A person has a personal number which is an integer and name, email and phone which are all strings.



UML

Attribute of a class may have a data type assigned



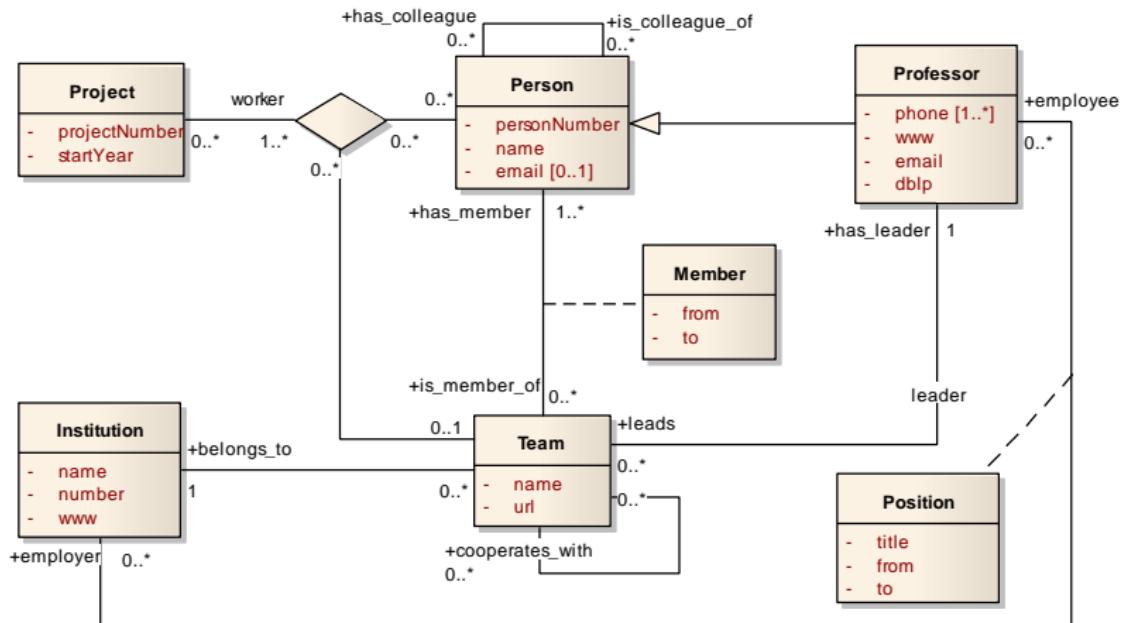
ER

Attribute of entity type may have a data type assigned

- Note that...

- Set of available data types is not specified strictly
- Data types are actually **not very important at the conceptual layer**

# Sample UML Diagram





**B0B36DBS, BD6B36DBS: Database Systems**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/>

Lecture 2

# **Relational Model**

**Martin Svoboda**

[martin.svoboda@fel.cvut.cz](mailto:martin.svoboda@fel.cvut.cz)

25. 2. 2020

**Czech Technical University in Prague, Faculty of Electrical Engineering**

# Lecture Outline

- **Logical database models**
  - Basic overview
- Model-Driven Development
- **Relational model**
  - Description and features
  - Transformation of ER / UML conceptual schemas

# **Logical Database Models**

# Layers of Database Modeling

Abstraction

- 
- **Conceptual layer**
    - Models a part of the structured real world relevant for applications built on top of our database
  - **Logical layer**
    - Specifies how conceptual components (i.e. **entity types, relationship types, and their characteristics**) are represented in **logical data structures** that are interpretable by machines
  - **Physical layer**
    - Specifies how logical database structures are implemented in a specific technical environment

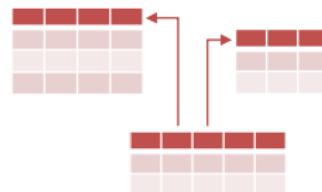
Implementation

# Logical Layer

- What are these logical structures?
  - Formally...
    - Sets, relations, functions, graphs, trees, ...
      - I.e. traditional and well-defined mathematical structures
  - Or in a more friendly way...
    - Tables, rows, columns, ...
    - Objects, pointers, ...
    - Collections, ...
    - ...

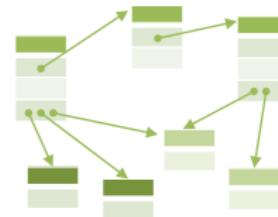
# Logical Models

- Models based on **tables**
  - Structure
    - **Rows for entities**
    - **Columns for attributes**
  - Operations
    - **Selection, projection, join, ...**
  - Examples
    - **Relational model**
    - ... and various derived **table models** introduced by:
      - **SQL** (as it is standardized)
      - and particular implementations like Oracle, MySQL, ...



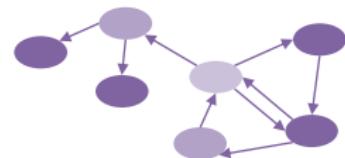
# Logical Models

- Models based on **objects**
  - Structure
    - Objects with **attributes**
    - Pointers between objects
  - Motivation
    - Object-oriented programming (OOP)
    - Encapsulation, inheritance, ...
  - Operations
    - **Navigation**



# Logical Models

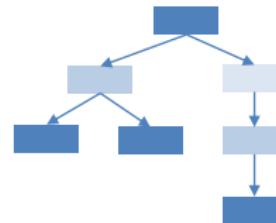
- Models based on **graphs**
  - Structure
    - Vertices, edges, attributes
  - Operations
    - **Traversals, pattern matching, graph algorithms**
  - Examples
    - Network model (one of the **very first database models**)
    - **Resource Description Framework (RDF)**
    - **Neo4j, InfiniteGraph, OrientDB, FlockDB, ...**



# Logical Models

- Models based on **trees**

- Structure
  - **Vertices** with attributes
  - **Edges** between vertices
- Motivation
  - Hierarchies, categorization, semi-structured data
- Examples
  - Hierarchical model (one of the very first database models)
  - **XML** documents
  - **JSON** documents

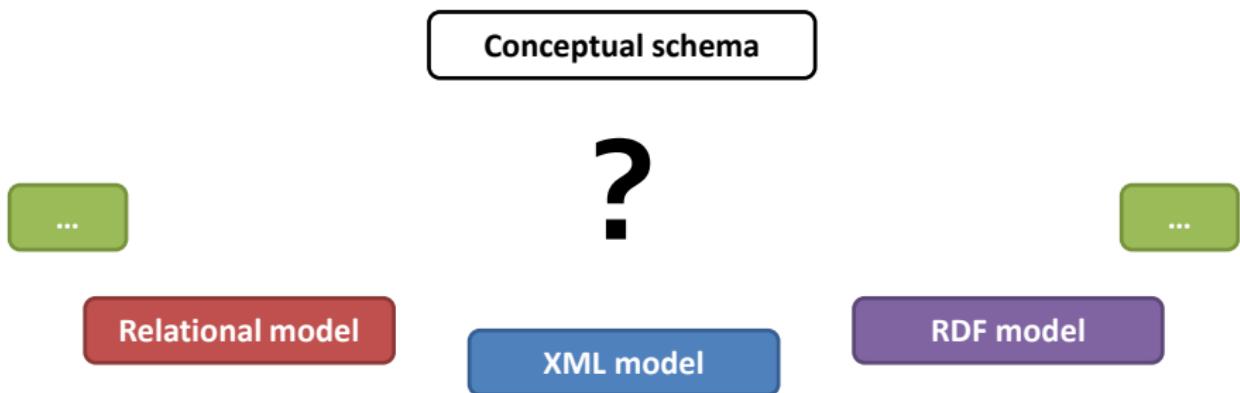


# Overview of Logical Models

- **There are plenty of (different / similar) models**
  - *The previous overview was intended just as an insight into some of the basic ideas and models*
  - Hierarchical, network, **relational**, **object**, **object-relational**, **XML**, **key-value**, **document-oriented**, **graph**, ...
- Why so many of them?
  - **Different models are suitable in different situations**
  - Not everything is (yet) standardized, proprietary approaches or extensions often exist

# Logical Modeling

- Step 1: Selection of the right logical model



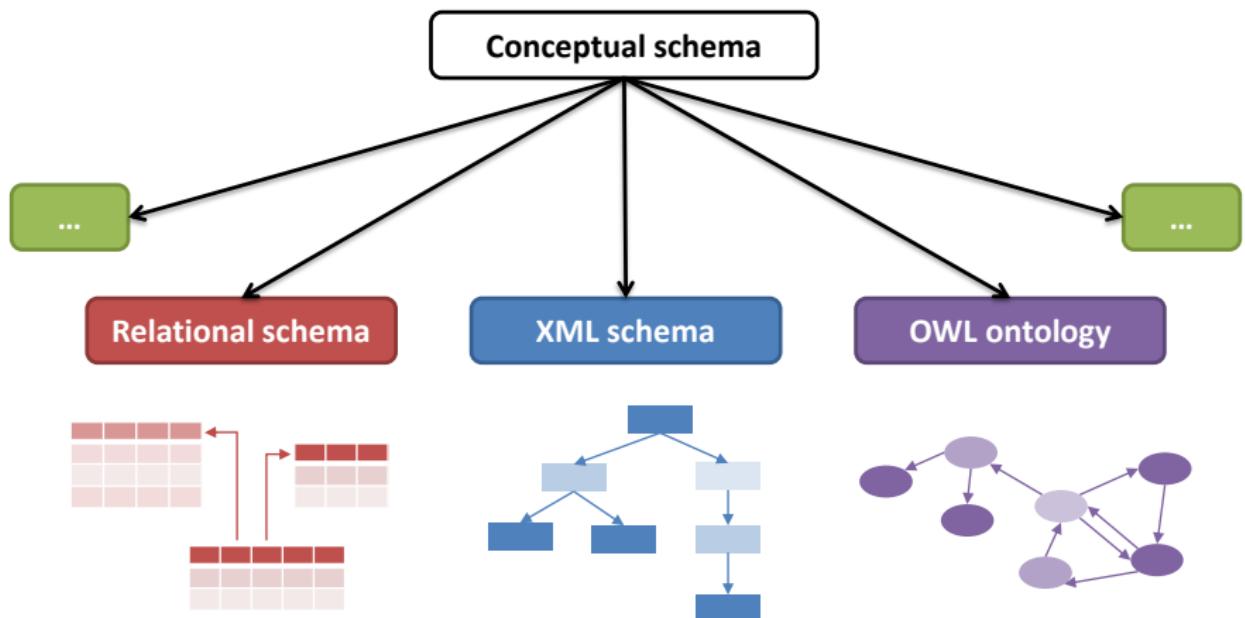
- Note that...
  - Relational model is not always the best solution

# Logical Modeling

- Step 1: **Selection of the right logical model**
  - According to...
    - **Data characteristics**
      - True nature of real-world entities and their relationships
    - **Query possibilities**
      - Available access patterns, expressive power, ...
    - **Intended usage**
      - Storage (JSON data in document-oriented databases, ...)
      - Exchange (XML documents sent by Web Service, ...)
      - Publication (RDF triples forming the Web of Data, ...)
      - ...
    - **Identified requirements**

# Logical Modeling

- Step 2: Creation of a logical schema



# Logical Modeling

- Step 2: **Creation of a logical schema**
  - Goal
    - Transformation of a conceptual schema to a logical one
  - Real-world applications often need **multiple schemas**
    - Focus on different parts of the real world
    - Serve different components of the system
    - Even expressed in different logical models
  - Challenge: **can this be achieved automatically?**
    - Or at least semi-automatically?
    - Answer: **Model-Driven Development**

# **Model-Driven Development (MDD)**

# Model-Driven Development

- MDD
  - Software development approach
    - **Executable schemas instead of executable code**
      - I.e. schemas that can be automatically (or at least semi-automatically) converted to executable code
    - **Unfortunately, just in theory... recent ideas, not yet fully applicable in practice today** (lack of suitable tools)
- MDD principles can be used for **database modeling** as well

# Terminology

- Levels of abstraction

- **Platform-Independent Level**

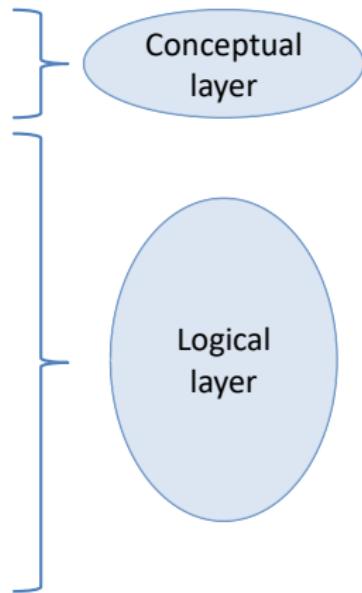
- Hides particular platform-specific details

- **Platform-Specific Level**

- Maps the conceptual schema (or its part) to a given logical model
    - Adds platform-specific details

- **Code Level**

- Expresses the schema in a selected machine-interpretable logical language
    - SQL, XML Schema, OWL, ...

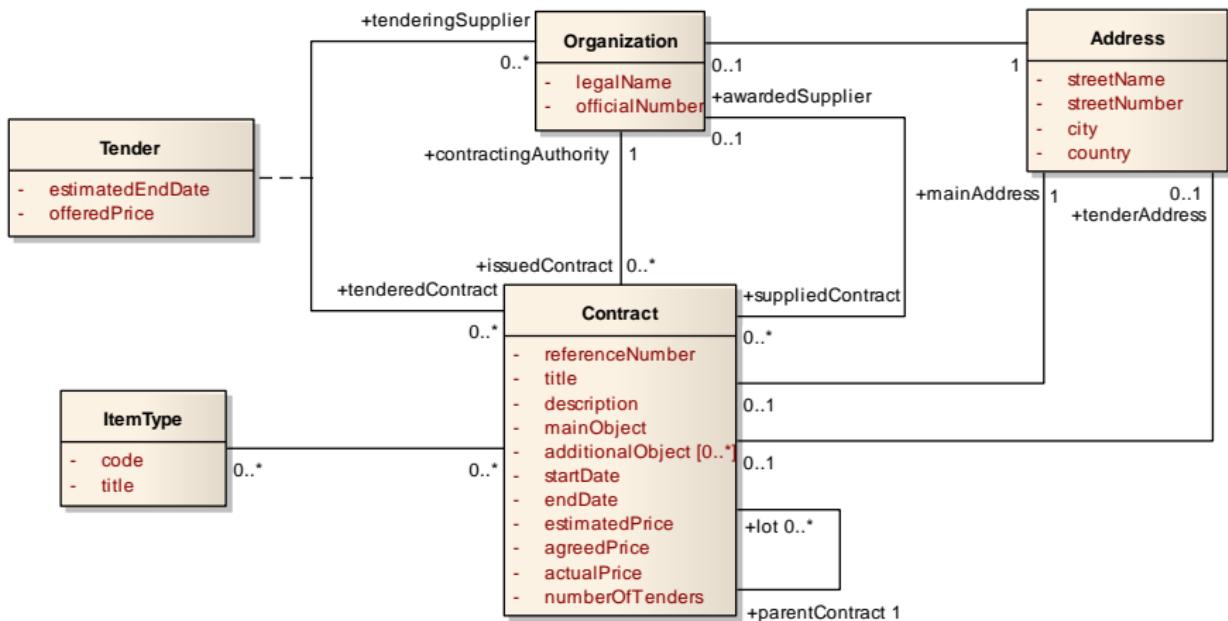


# Real-World Example

- Information System for Public Procurement
  - <http://www.isvzus.cz/>
  - There are several logical models used:
    - **Relational data model**
      - for data storage
    - **XML data model**
      - for exchanging data with information systems of public authorities which issue public contracts
    - **RDF data model**
      - for publishing data on the Web of Linked Data in a machine-readable form (at least this is a goal...)

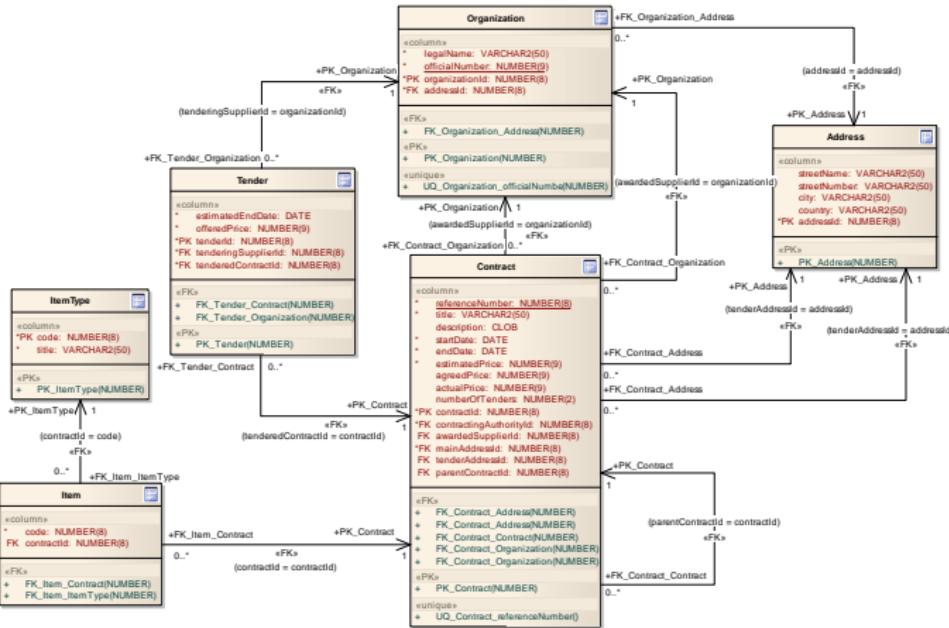
# Real-World Example

- Platform-independent schema



# Real-World Example

- Platform-specific schema: relational model



# Real-World Example

- Platform-specific schema: **relational model**
  - Notes to the previous UML diagram
    - It is a UML class diagram
      - But enhanced with features for modeling logical schemas in (object-)relational model
    - **Stereotypes** allow us to add **specific semantics** to basic constructs (class, attribute, association), e.g.,
      - <<table>> specifies that a class represents a table
      - <<PK>> specifies that an attribute models a primary key
      - <<FK>> specifies that an attribute/association models a foreign key
      - etc.

# Real-World Example

- Code level: **SQL** (snippet)

```
CREATE TABLE Contract (
    referenceNumber NUMBER(8) NOT NULL,
    title VARCHAR2(50) NOT NULL,
    description CLOB,
    startDate DATE NOT NULL,
    endDate DATE NOT NULL,
    estimatedPrice NUMBER(9) NOT NULL,
    ...
);

ALTER TABLE Contract ADD CONSTRAINT PK_Contract
    PRIMARY KEY (contractId);
ALTER TABLE Contract ADD CONSTRAINT FK_Contract_Address
    FOREIGN KEY (mainAddressId) REFERENCES Address (addressId);
    ...

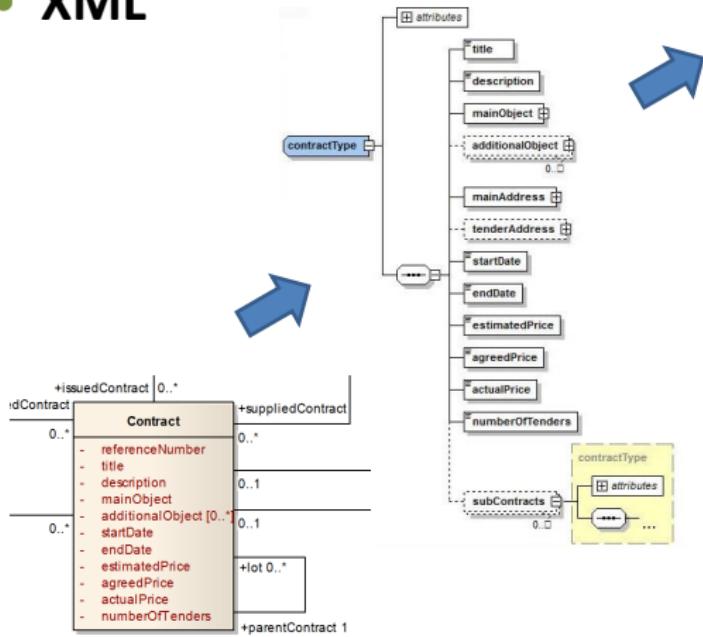
CREATE TABLE Organization(...);
    ...
```

# Real-World Example

- Code level: **SQL** (snippet)
  - The previous code was generated **fully automatically**
    - from a platform-specific diagram
      - It has to contain all the necessary information
    - using a **CASE tool** (Computer-Aided Software Engineering)
      - Which can detect errors and
      - helps with the specification

# Real-World Example

- XML

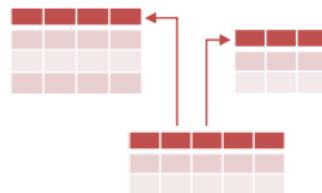


```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2012 sp1 (http://www.altova.com) by IM (Charles ! -->
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">
  <xss:complexType name="addressType">
    <xss:sequence>
      <xss:element name="streetName"/>
      <xss:element name="streetNumber"/>
      <xss:element name="city"/>
      <xss:element name="country"/>
    </xss:sequence>
  </xss:complexType>
  <xss:complexType name="objectType">
    <xss:sequence>
      <xss:element name="code" type="xs:int"/>
      <xss:element name="title" type="xs:string"/>
    </xss:sequence>
  </xss:complexType>
  <xss:complexType name="contractType">
    <xss:sequence>
      <xss:element name="title" type="xs:string"/>
      <xss:element name="description" type="xs:string"/>
      <xss:element name="mainAddress" type="addressType"/>
      <xss:element name="additionalObject" type="objectType"/>
      <xss:element name="tenderAddress" type="addressType"/>
      <xss:element name="startDate" type="xs:dateTime"/>
      <xss:element name="endDate" type="xs:dateTime"/>
      <xss:element name="estimatedPrice" type="xs:float"/>
      <xss:element name="agreedPrice" type="xs:float"/>
      <xss:element name="actualPrice" type="xs:float"/>
      <xss:element name="numberOftenders" type="xs:int"/>
      <xss:element name="subContracts" type="contractTyp
        <xss:sequence>
          <xss:attribute name="referenceNumber" type="xs:int"/>
        </xss:sequence>
      </xss:complexType>
    </xss:sequence>
  </xss:complexType>
</xss:schema>
```

# **Relational Model**

# Relational Model

- **Relational model**
  - Allows to store entities, relationships, and their attributes **in relations**
  - Founded by E. F. Codd in 1970
- Informally...
  - **Table** = collection of **rows**, each row represents one entity, values of **attributes** are stored in **columns**
  - Tables are more intuitive, but conceal important mathematical background



# Relational Model

- Definitions and terminology
  - **Schema of a relation**
    - Description of a relational structure (everything except data)
    - $S(A_1:T_1, A_2:T_2, \dots, A_n:T_n)$ 
      - $S$  is a schema name
      - $A_i$  are attribute names and  $T_i$  their types (attribute domains)
      - Specification of types is often omitted
    - Example:
      - Person(personalId, firstName, lastName)
  - **Schema of a relational database**
    - Set of relation schemas (+ integrity constraints, ...)

# Relational Model

- Definitions and terminology for **data**
  - **Relation**
    - Subset of the Cartesian product of attribute domains  $T_i$ 
      - I.e. relation is a set
    - Items are called **tuples**
  - **Relational database**
    - Set of relations

# Relational Model

- Basic requirements (or consequences?)
  - **Atomicity** of attributes
    - Only **simple types** can be used for domains of attributes
  - **Uniqueness** of tuples
    - Relation is a set, and so **two identical tuples cannot exist**
  - **Undefined order**
    - Relation is a set, and so **tuples are not mutually ordered**
  - **Completeness** of values
    - There are no *holes* in tuples, i.e. **all values are specified**
      - However, special **NULL** values (well-known from relational databases) can be added to attribute domains

# Integrity Constraints

- **Identification**
  - Every tuple is identified by one or more attributes
  - **Superkey** = set of such attributes
    - Trivial and special example: all the relation attributes
  - **Key** = superkey with a *minimal* number of attributes
    - I.e. no attribute can be removed so that the identification ability would still be preserved
    - Multiple keys may exist in one relation
      - They even do not need to have the same number of attributes
    - Notation: keys are underlined
      - Relation(Key, CompositeKeyPart1, CompositeKeyPart2, ...)
      - Note the difference between simple and composite keys

# Integrity Constraints

- Referential integrity
  - **Foreign key** = set of attributes of the referencing relation which corresponds to a (super)key of the referenced relation
    - It is usually not a (super)key in the referencing relation
    - Notation
      - `ReferencingTable.foreignKey ⊆ ReferencedTable.Key`
      - `foreignKey ⊆ ReferencedTable.Key`

# Sample Relational Database

- Schema

$\text{Course}(\underline{\text{Code}}, \text{Name}, \dots)$

$\text{Schedule}(\underline{\text{Id}}, \text{Event}, \text{Day}, \text{Time}, \dots)$ ,  $\text{Event} \sqsubseteq \text{Course.Code}$

- Data

<b>Id</b>	<b>Event</b>	<b>Day</b>	<b>Time</b>	<b>...</b>
1	A7B36DBS	THU	11:00	
2	A7B36DBS	THU	12:45	
3	A7B36DBS	THU	14:30	
4	A7B36XML	FRI	09:15	

<b>Code</b>	<b>Name</b>	<b>...</b>
A7B36DBS	Database systems	
A7B36XML	XML technologies	
A7B36PSI	Computer networks	

# Relations vs. Tables

- Tables
  - **Table header ~ relation schema**
  - **Row ~ tuple**
  - **Column ~ attribute**
- However...
  - Tables are not sets, and so...
    - there can be duplicate rows in tables
    - rows in tables can be ordered
  - I.e. SQL and existing RDBMS do not (always) follow the formal relational model strictly

# Object vs. (Object-)Relational Model

- **Relational model**
  - Data stored in flat tables
  - Suitable for data-intensive batch operations
- **Object model**
  - Data stored as graphs of objects
  - Suitable for individual navigational access to entities
- **Object-Relational model**
  - Relational model enriched by object elements
    - Attributes may be of complex data types
    - Methods can be defined on data types as well

# **Transformation of UML / ER to RM**

# Conceptual Schema Transformation

- **Basic idea**
  - What we have
    - ER: entity types, attributes, identifiers, relationship types, ISA hierarchies
    - UML: classes, attributes, associations
  - What we need
    - **Schemas of relations with attributes, keys, and foreign keys**
  - How to do it
    - **Classes with attributes** → relation schemas
    - **Associations** → separate relation schemas or together with classes (depending on cardinalities...)

# Classes

- **Class →**
  - Separate table
    - Person(personalNumber, address, age)
  - Artificial keys
    - Artificially added **integer identifiers**
      - with no correspondence in the real world
      - but with several efficiency and also design advantages
      - usually automatically generated and assigned
    - Person(personId, personalNumber, address, age)

Person
- personalNumber - address - age

# Attributes

- **Multivalued attribute →**

- Separate table

- Person(personalNumber)
  - Phone(personalNumber, phone)
  - Phone.personalNumber ⊆ Person.personalNumber

Person
- personalNumber - phone: String [1..*]

# Attributes

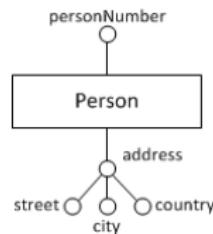
- **Composite attribute** →

- Separate table

- Person(personalNumber)  
Address(personalNumber, street, city, country)  
Address.personalNumber  $\subseteq$  Person.personalNumber

- Sub-attributes can also be inlined

- But only in case of (1,1) cardinality
  - Person(personalNumber, street, city, country)



# Binary Associations

- Multiplicity (1,1):(1,1) →



- Three tables (basic approach)
  - Person(personalNumber, address, age)  
Mobile(serialNumber, color)  
Ownership(personalNumber, serialNumber)  
Ownership.personalNumber ⊆ Person.personalNumber  
Ownership.serialNumber ⊆ Mobile.serialNumber

# Binary Associations

- Multiplicity (1,1):(1,1) →



- Single table
  - **Person(personalNumber, address, age, serialNumber, color)**

# Binary Associations

- Multiplicity (1,1):(0,1) →



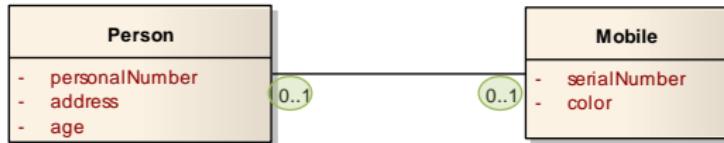
- Two tables
  - **Person(personalNumber, address, age, serialNumber)**  
Person.serialNumber  $\subseteq$  Mobile.serialNumber

**Mobile(serialNumber, color)**

- Why not just 1 table?
  - Because a mobile phone can exist independently of a person

# Binary Associations

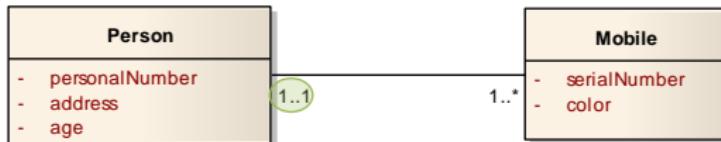
- Multiplicity (0,1):(0,1) →



- Three tables
  - Person(personalNumber, address, age)  
Mobile(serialNumber, color)  
Ownership(personalNumber, serialNumber)  
Ownership.personalNumber ⊆ Person.personalNumber  
Ownership.serialNumber ⊆ Mobile.serialNumber
  - Note that a personal number and serial number are both independent keys in the Ownership table

# Binary Associations

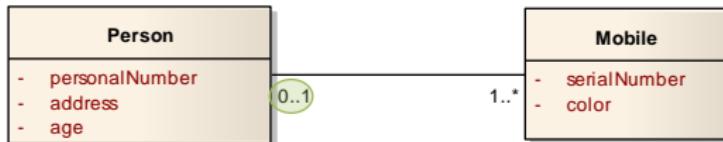
- Multiplicity  $(1,n)/(0,n):(1,1) \rightarrow$



- Two tables
  - Person(personalNumber, address, age)  
Mobile(serialNumber, color, personalNumber)  
 $\text{Mobile}.\text{personalNumber} \sqsubseteq \text{Person}.\text{personalNumber}$
  - Why a personal number is not a key in the Mobile table?
    - Because a person can own more mobile phones

# Binary Associations

- Multiplicity  $(1,n)/(0,n):(0,1) \rightarrow$



- Three tables
  - Person(personalNumber, address, age)  
Mobile(serialNumber, color)  
Ownership(personalNumber, serialNumber)  
Ownership.personalNumber  $\subseteq$  Person.personalNumber  
Ownership.serialNumber  $\subseteq$  Mobile.serialNumber
  - Why a personal number is not a key in the Ownership table?
    - Because a person can own more mobile phones

# Binary Associations

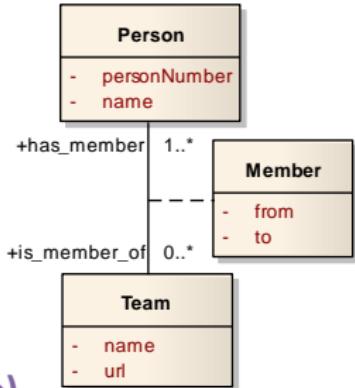
- Multiplicity  $(1,n)/(0,n):(1,n)/(0,n) \rightarrow$



- Three tables
  - Person(personalNumber, address, age)
  - Mobile(serialNumber, color)
  - Ownership(personalNumber, serialNumber)
- Ownership.personalNumber  $\subseteq$  Person.personalNumber
- Ownership.serialNumber  $\subseteq$  Mobile.serialNumber
- Note that there is a **composite key** in the Ownership table

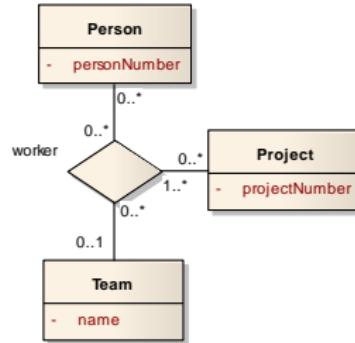
# Attributes of Associations

- Attribute of an association →
  - Stored together with a given association table
    - Person(personNumber, name)
    - Team(name, url)
    - Member(personNumber, name, from, to)
    - Member.personNumber  $\subseteq$  Person.personNumber
    - Member.name  $\subseteq$  Team.name
  - Multivalued and composite attributes are transformed analogously to attributes of ordinary classes



# General Associations

- N-ary association →
  - Universal solution:  
**N tables for classes + 1 association table**
    - Person(personNumber)
    - Project(projectNumber)
    - Team(name)
    - Worker(personNumber, projectNumber, name)**
    - Worker.personNumber  $\subseteq$  Person.personNumber
    - Worker.projectNumber  $\subseteq$  Project.projectNumber
    - Worker.name  $\subseteq$  Team.name
  - Less tables? Yes, in case of nice (1,1) cardinalities...

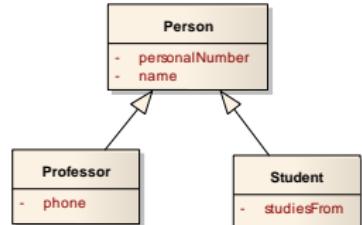


# Hierarchies

- ISA hierarchy →

- Universal solution:  
separate table for each type  
with specific attributes only

- Person(personalNumber, name)  
Professor(personalNumber, phone)  
Student(personalNumber, studiesFrom)  
Professor.personalNumber  $\subseteq$  Person.personalNumber  
Student.personalNumber  $\subseteq$  Person.personalNumber
    - Applicable in any case (w.r.t. **covering / overlap constraints**)
    - Pros: flexibility (when attributes are altered)
    - Cons: joins (when full data is reconstructed)



# Hierarchies

- ISA hierarchy →
  - Only one table for a hierarchy source
    - Person(personalNumber, name, phone, studiesFrom, type)
    - Universal once again, but **not always suitable**
      - Types of instances are distinguished by an artificial attribute
        - » Enumeration or event a set depending on the overlap constraint
    - Pros: no joins
    - Cons: NULL values required (and so it is not a nice solution)

# Hierarchies

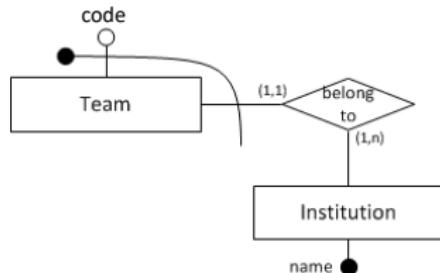
- ISA hierarchy →
  - Separate table for each leaf type
    - Professor(personalNumber, name, phone)
    - Student(personalNumber, name, studiesFrom)
    - This solution is **not always applicable**
      - In particular when the covering constraint is false
    - Pros: no joins
    - Cons:
      - Redundancies (when the overlap constraint is false)
      - Integrity considerations (uniqueness of a personal number)

# Weak Entity Types

- **Weak entity type →**

- Separate table

- Institution(name)  
Team(code, name)  
Team.name ⊆ Institution.name
- Recall that the cardinality must always be (1,1)
- Key of the weak entity type involves also a key (any when more available) from the entity type it depends on





B0B36DBS, BD6B36DBS: **Database Systems**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/>

Lecture 3

# **SQL: Data Definition**

**Martin Svoboda**

[martin.svoboda@fel.cvut.cz](mailto:martin.svoboda@fel.cvut.cz)

3. 3. 2020

**Czech Technical University in Prague, Faculty of Electrical Engineering**

# Outline

- **SQL**
  - **Data definition**
    - Definition of tables
    - Data types
    - Integrity constraints
    - Schema modification
  - **Data manipulation**
    - Insertion
    - Updates
    - Deletion

# **Structured Query Language (SQL)**

# Structured Query Language

- **SQL**
  - Standard language for accessing relational databases
    - **Data definition** (DDL)
      - Creation of table schemas and integrity constraints
    - **Data manipulation** (DML)
      - Querying
      - Data insertion, deletion, updates
    - **Transaction** management
    - Modules (programming language)
    - Database administration

# Structured Query Language

- **SQL standards**

- Backwards compatible
- ANSI/ISO
  - **SQL-86** – intersection of IBM SQL implementations
  - SQL-89 – small revision, integrity constraints
  - **SQL-92** – schema modification, transactions, set operators, new data types, cursors, referential integrity actions, ...
  - **SQL:1999** – recursive queries, triggers, object-relational features, regular expressions, types for full-text, images, spatial data, ...
  - **SQL:2003** – SQL/XML, sequence generators
  - SQL:2006 – other extensions of XML, integration of XQuery
  - SQL:2008
  - SQL:2011 – temporal databases

# Structured Query Language

- **Commercial systems**

- Current implementations at different standard levels
  - Most often SQL:1999, SQL:2003
- However (and unfortunately)...
  - **Some extra proprietary features supported**
  - **Some standard features not supported**
  - Even syntax may differ
    - And so data migration is usually not straightforward
- Specific extensions
  - Procedural, transactional and other functionality, e.g., TRANSACT-SQL (Microsoft SQL Server), PL/SQL (Oracle)

# SQL Syntax Diagrams

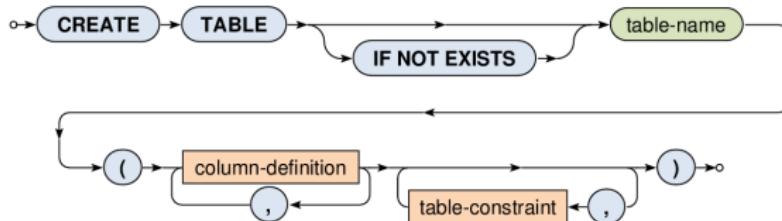
- **Syntax (railroad) diagrams**
  - Graphical representation of context-free grammars
    - I.e. a practical approach how to describe languages (such as SQL) in a graphical and user-friendly way
  - Technically...
    - Directed graph representing an automaton accepting SQL
    - Terms in diagrams:
      - Capital letters on blue – keywords
      - Small letters on green – literals
      - Small letters on orange – subexpressions



# **SQL: Schema Definition**

# Table Creation

- **CREATE TABLE**
  - Construction of a table schema (and an empty table)
    - **Table name**
    - Definition of **table columns**
      - Together with their column-scope integrity constraints
    - Definition of **table-scope integrity constraints**



# Table Creation

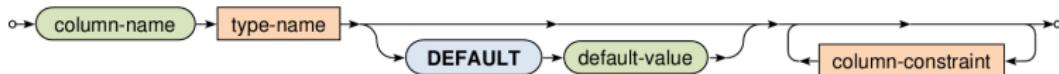
- **CREATE TABLE**

- **Definition of table columns**

- **Column name**
- **Data type**
- **Default value**

- When a new row is about to be inserted and not all its values are specified, then the default values are used (if defined)

- Definition of column-scope IC



# Table Creation

- Example
  - Simple table without integrity constraints

```
CREATE TABLE Product (
    Id INTEGER,
    Name VARCHAR(128),
    Price DECIMAL(6,2),
    Produced DATE,
    Available BOOLEAN DEFAULT TRUE,
    Weight FLOAT
);
```

# Data Types

- Available data types
  - Precise numeric types
    - **INTEGER**, INT, SMALLINT, BIGINT
    - **DECIMAL**(precision, scale)
      - Precision = number of all digits (including decimal digits)
      - Scale = number of decimal digits
  - Approximate numeric types
    - FLOAT, REAL, DOUBLE PRECISION – real numbers
  - Logical values
    - **BOOLEAN**

# Data Types

- Available data types
  - Character strings
    - **CHAR(length)**, **CHARACTER(length)** – fixed-length strings
      - Shorter strings are automatically right-padded with spaces
    - **VARCHAR(length)**, **CHARACTER VARYING(length)**
      - Strings of a variable length
  - Temporal types
    - **DATE**, **TIME**, **TIMESTAMP**
- Type conversions
  - Meaningful conversions are defined automatically
    - Otherwise see **CAST...**

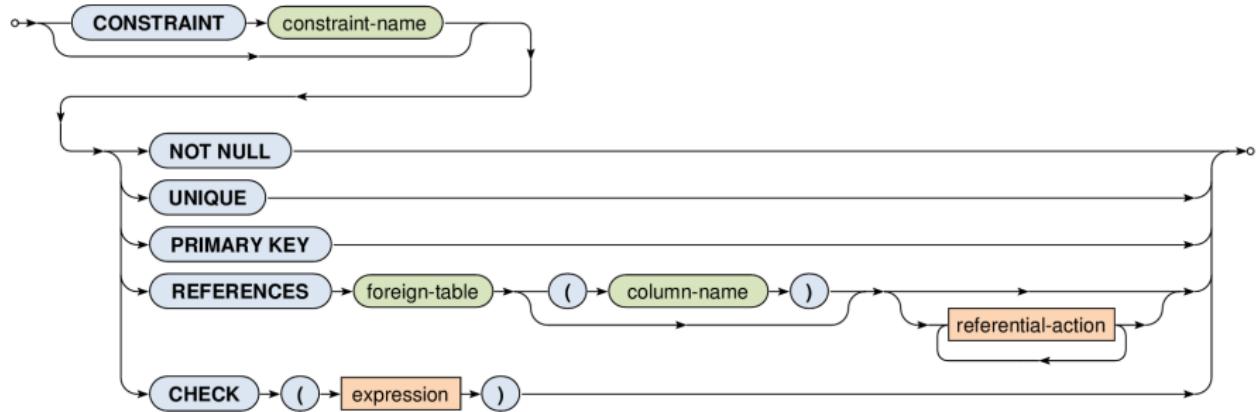
# Data Types

- Example
  - Simple table without integrity constraints

```
CREATE TABLE Product (
    Id INTEGER,
    Name VARCHAR(128),
    Price DECIMAL(6,2),
    Produced DATE,
    Available BOOLEAN DEFAULT TRUE,
    Weight FLOAT
);
```

# Integrity Constraints

- **Column integrity constraints**
  - Allow us to limit domains of the allowed values



# Integrity Constraints

- Column integrity constraints
  - **NOT NULL**
    - Values must not be NULL
  - **UNIQUE**
    - All values must be distinct
      - But can there be just one or multiple NULL values?
  - **PRIMARY KEY**
    - Only one primary key is allowed in a table!
    - Equivalent to NOT NULL + UNIQUE

# Integrity Constraints

- Column integrity constraints
  - **FOREIGN KEY**
    - Referential integrity
      - Values from the referencing table must also exist in the referenced table
      - NULL values are ignored
      - Only unique / primary keys can be referenced
  - **CHECK**
    - Generic condition that must be satisfied
      - However, only values within a given row may be tested

# Integrity Constraints: Example

```
CREATE TABLE Producer (
    Id INTEGER PRIMARY KEY,
    Name VARCHAR(128),
    Country VARCHAR(64)
);

CREATE TABLE Product (
    Id INTEGER CONSTRAINT IC_Product_PK PRIMARY KEY,
    Name VARCHAR(128) UNIQUE,
    Price DECIMAL(6,2) CONSTRAINT IC_Product_Price NOT NULL,
    Produced DATE CHECK (Produced >= '2015-01-01'),
    Available BOOLEAN DEFAULT TRUE NOT NULL,
    Weight FLOAT,
    Producer INTEGER REFERENCES Producer (Id)
);
```

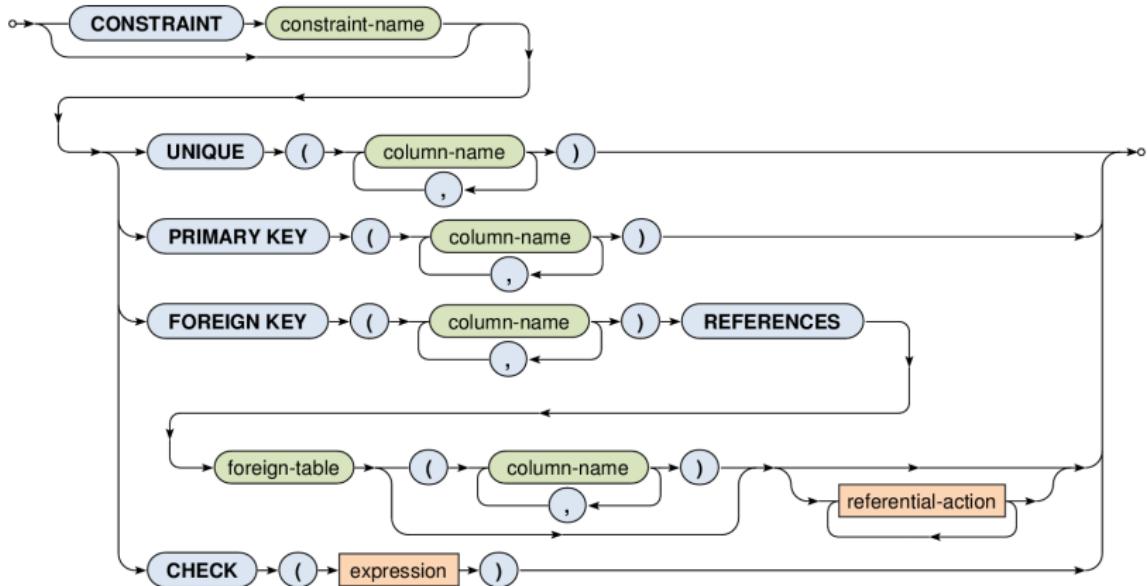
# Integrity Constraints: Example

- **Example**
  - Referential integrity within a single table

```
CREATE TABLE Employee (
    Id INTEGER PRIMARY KEY,
    Name VARCHAR(128),
    Boss INTEGER REFERENCES Employee (Id)
);
```

# Integrity Constraints

- Table integrity constraints



# Integrity Constraints

- **Table integrity constraints**

- Analogous to column IC, just for multiple columns,  
i.e. for tuples of values

- **UNIQUE**

- **PRIMARY KEY**

- **FOREIGN KEY**

- Tuples containing at least one `NULL` value are ignored

- **CHECK**

- Even with more complex conditions testing the entire tables
      - However, table integrity constraints are considered to be satisfied on empty tables (by definition, without evaluation)
      - See `CREATE ASSERTION...`

# Integrity Constraints: Example

```
CREATE TABLE Producer (
    Name VARCHAR(128),
    Country VARCHAR(3),
    CONSTRAINT IC_Producer_PK PRIMARY KEY (Name, Country)
);
```

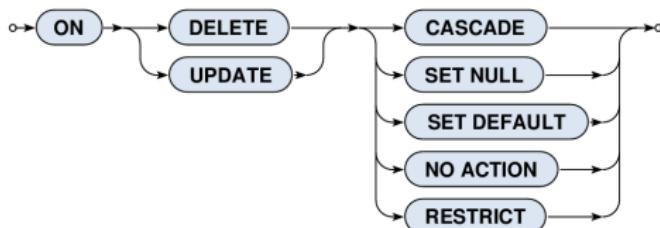
```
CREATE TABLE Product (
    Id INTEGER PRIMARY KEY,
    ...
    ProducerName VARCHAR(128),
    ProducerCountry VARCHAR(3),
    CONSTRAINT IC_Product_Producer_FK
        FOREIGN KEY (ProducerName, ProducerCountry)
        REFERENCES Producer (Name, Country)
);
```

# Referential Integrity

- **Referential actions**
  - When an operation on the referenced table would cause violation of the foreign key in the referencing table...
    - I.e. value of the foreign key of at least one row in the referencing table would become invalid as a result
  - ... then...
    - this operation is blocked and an error message is generated
    - but if a referential action is defined, it is triggered...

# Referential Integrity

- **Referential actions**



- Triggering situations
  - **ON UPDATE, ON DELETE**
    - When the action is triggered
    - Once again, these are considered to be operations on the referenced table

# Referential Integrity

- Referential actions
  - **CASCADE**
    - Row with the referencing value is updated / deleted as well
  - **SET NULL** – referencing value is set to **NULL**
  - **SET DEFAULT** – referencing value is **set to its default**
  - **NO ACTION** – default – **no action takes place**
    - i.e. as if no referential action would be defined at all
  - **RESTRICT** – no action takes place as well...
    - However, the integrity check is performed at the beginning, i.e. before the operation is even tried to be executed
      - ... and so triggers or the operation itself have no chance to remedy the situation even if they could be able to achieve such a state (and so RESTRICT is different to NO ACTION)

# Referential Integrity: Example

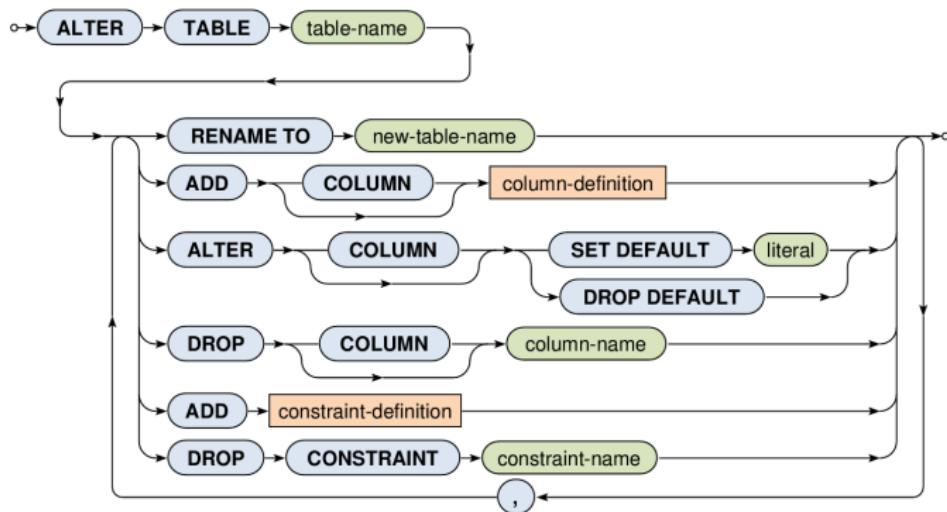
```
CREATE TABLE Producer (
    Id INTEGER PRIMARY KEY,
    Name VARCHAR(128),
    Country VARCHAR(64)
);
```

```
CREATE TABLE Product (
    Id INTEGER PRIMARY KEY,
    ...
    Producer INTEGER
        REFERENCES Producer (Id) ON DELETE CASCADE
);
```

When producer is deleted, Product is deleted too.

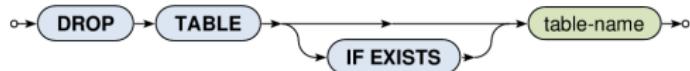
# Schema Modification

- **ALTER TABLE**
  - Addition/change/removal of table columns/IC



# Schema Modification

- **DROP TABLE**
  - Complementary to the table creation
    - I.e. table definition as well as table content are deleted



# **SQL: Data Manipulation**

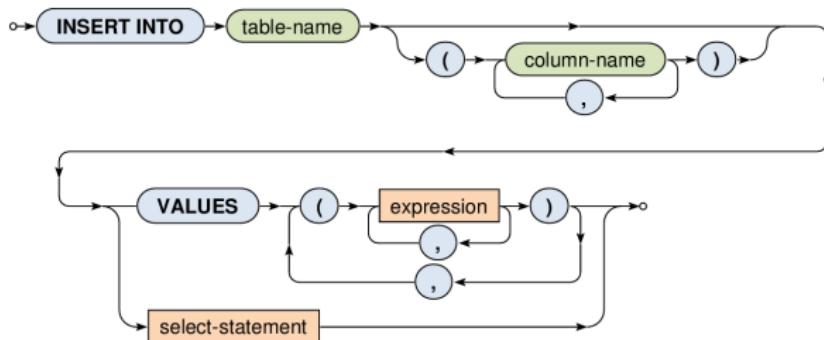
# SQL Data Manipulation

- Data manipulation language
  - Data modification
    - **INSERT INTO** – insertion of rows
    - **DELETE FROM** – deletion of rows
    - **UPDATE** – modification of rows
  - Data querying
    - **SELECT**... *the next lecture*

# Data Insertion

- **INSERT INTO**

- Insertion of new rows into a table
  - ...by an explicit enumeration / from a result of a selection
  - Default values are assumed for the omitted columns



# Data Insertion: Example

```
CREATE TABLE Product (
    Id INTEGER PRIMARY KEY,
    Name VARCHAR(128) UNIQUE,
    Price DECIMAL(6,2) NOT NULL,
    Produced DATE,
    Available BOOLEAN DEFAULT TRUE,
    Weight FLOAT,
    Producer INTEGER
);
```

```
INSERT INTO Product
VALUES (0, 'Chair1', 2000, '2015-05-06', TRUE, 3.5, 11);
```

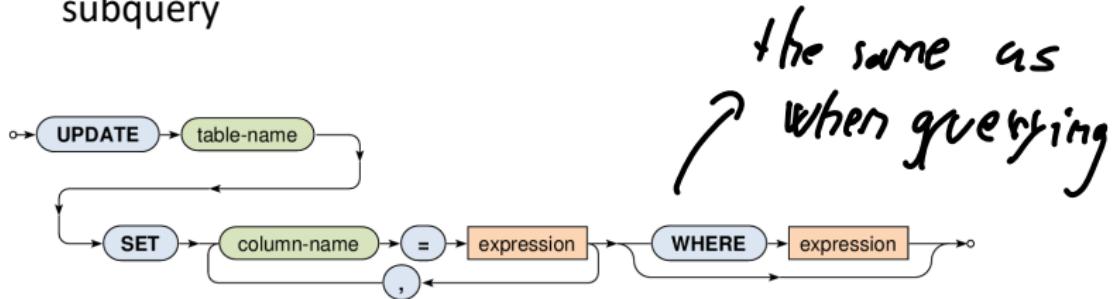
```
INSERT INTO Product
(Id, Name, Price, Produced, Weight, Producer)
VALUES (1, 'Chair2', 1500, '2015-05-06', 4.5, 11);
```

omitted column specification  
→ will have to insert everything.

# Data Updates

- **UPDATE**

- Modification of existing rows in a table
  - Only rows matching the given condition are considered
- Newly assigned values can be...
  - NULL, literal, value given by an expression, result of a scalar subquery



# Data Updates: Example

```
CREATE TABLE Product (
    Id INTEGER PRIMARY KEY,
    Name VARCHAR(128) UNIQUE,
    Price DECIMAL(6,2) NOT NULL,
    Produced DATE,
    Available BOOLEAN DEFAULT TRUE,
    Weight FLOAT,
    Producer INTEGER
);
```

**UPDATE Product**

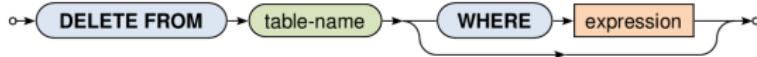
```
SET Name = 'Notebook'
WHERE (Name = 'Laptop');
```

**UPDATE Product**

```
SET Price = Price * 0.9
WHERE (Produced < '2015-01-01');
```

# Data Deletion

- **DELETE FROM**
  - Deletion of existing rows from a table
    - Only rows matching the given condition are considered



# Data Deletion: Example

```
CREATE TABLE Product (
    Id INTEGER PRIMARY KEY,
    Name VARCHAR(128) UNIQUE,
    Price DECIMAL(6,2) NOT NULL,
    Produced DATE,
    Available BOOLEAN DEFAULT TRUE,
    Weight FLOAT,
    Producer INTEGER
);
```

```
DELETE FROM Product
WHERE (Price > 2000);
```

```
DELETE FROM Product;
```



B0B36DBS, BD6B36DBS: **Database Systems**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/>

Lecture 4

# **SQL: Data Querying**

**Martin Svoboda**

[martin.svoboda@fel.cvut.cz](mailto:martin.svoboda@fel.cvut.cz)

10. 3. 2020

**Czech Technical University in Prague, Faculty of Electrical Engineering**

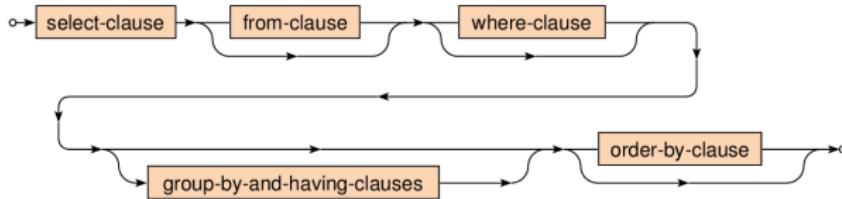
# Outline

- **SQL**
  - Data manipulation
    - **SELECT queries**

# **SQL: Select Queries**

# Select Queries

- **SELECT statements** in a nutshell
  - Consist of 1-5 clauses and optionally also ORDER BY clause
  - **SELECT** clause: which columns should be included in the result table
  - **FROM** clause: which source tables should provide data we want to query
  - **WHERE** clause: condition a row must satisfy to be included in the result
  - **GROUP BY** clause: which attributes should be used for the aggregation
  - **HAVING** clause: condition an aggregated row must satisfy to be in the result
  - **ORDER BY** clause: attributes that are used to sort rows of the final result



# Sample Tables

- Database of flights and aircrafts

Flights:



Flight	Company	Destination	Passengers
OK251	CSA	New York	276
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
OK321	CSA	London	156
AC906	Air Canada	Toronto	116
KL7621	KLM	Rotterdam	75
KL1245	KLM	Amsterdam	130

Aircrafts:

Aircraft	Company	Capacity
Boeing 717	CSA	106
Airbus A380	KLM	555
Airbus A350	KLM	253

# Select Queries: Example

- Which aircrafts can be used for the scheduled flights?
  - Only aircrafts of a given company and sufficient capacity can be used

```
SELECT Flights.* , Aircraft  
FROM Flights NATURAL JOIN Aircrafts  
WHERE (Passengers <= Capacity)  
ORDER BY Flight
```



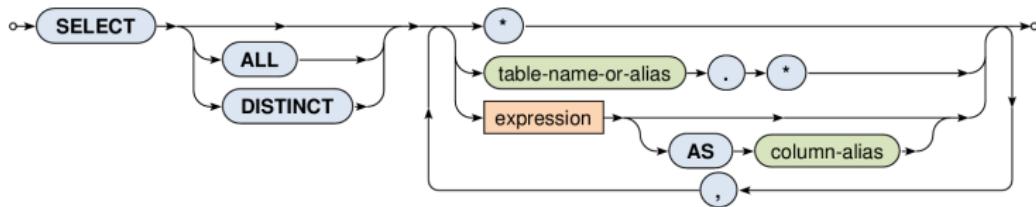
Aircraft	Company	Capacity
Boeing 717	CSA	106
Airbus A380	KLM	555
Airbus A350	KLM	253

Flight	Company	Destination	Passengers	Aircraft
KL1245	KLM	Amsterdam	130	Airbus A380
KL1245	KLM	Amsterdam	130	Airbus A350
KL7621	KLM	Rotterdam	75	Airbus A380
KL7621	KLM	Rotterdam	75	Airbus A350
OK012	CSA	Milano	37	Boeing 717

Flight	Company	Destination	Passengers
OK251	CSA	New York	276
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
OK321	CSA	London	156
AC906	Air Canada	Toronto	116
KL7621	KLM	Rotterdam	75
KL1245	KLM	Amsterdam	130

# Select Clause

- **SELECT ... FROM ... WHERE ... ORDER BY ...**
  - **List of columns** to be included in the result
    - Projection of input columns
      - **Column name**
      - **\*** (all columns), **table.\*** (all from a given table)
    - Definition of new, derived and aggregated columns
      - Using expressions based on literals, functions, subqueries, ...
    - Columns can also be assigned **(new) names** using **AS**



# Select Clause

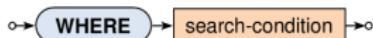
- **SELECT**
  - **Output modifiers**
    - **ALL** (default) – all the rows are included in the output
    - **DISTINCT** – **duplicities are removed**
  - **Examples**
    - **SELECT ALL \*** ...
    - **SELECT Flights.\***, Aircraft ...
    - **SELECT DISTINCT Company AS Carrier** ...
    - **SELECT ((3\*5) + 5) AS MyNumber**, 'Hello' AS MyString ...
    - **SELECT SUM(Capacity)** ...
    - **SELECT (SELECT COUNT(\*) FROM Table) AS Result** ...

# Where Clause

- SELECT ... FROM ... WHERE ... ORDER BY ...

- Selection condition

- I.e. condition that a row must satisfy to get into the result
    - Simple expressions may be combined using conjunctions
      - AND, OR, NOT



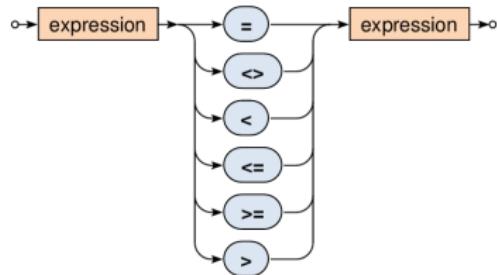
- Examples

- ... WHERE (Capacity > 200) AND (Aircraft LIKE 'Airbus%') ...
    - ... WHERE (Company IN ('KLM', 'Emirates')) ...
    - ... WHERE NOT (Passengers BETWEEN 100 AND 200) ...

# Search Conditions

- **Comparison predicates**

- Standard comparison
- Works even for tuples
  - Example:  $(1,2,3) \leq (1,2,5)$



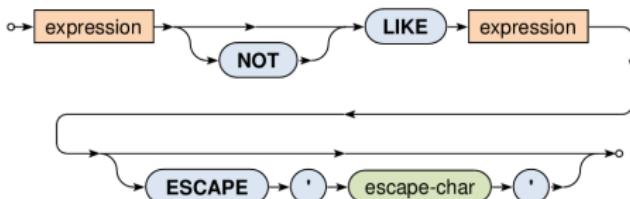
- **Interval predicate**

- Value BETWEEN Min AND Max  
is equivalent to  
 $(\text{Min} \leq \text{Value}) \text{ AND } (\text{Value} \leq \text{Max})$



# Search Conditions

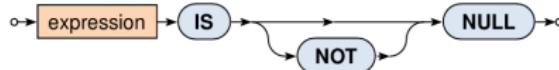
- **String matching predicate**
  - Tests whether a string value matches a given pattern
    - This pattern may contain special characters:
      - `%` matches an arbitrary substring (even empty)
      - `_` matches an arbitrary character
    - Optional escaping character can also be set



- Example
  - Company LIKE '%Airlines%'

# Search Conditions

- **NULL values detection predicate**
  - Tests whether a given value is / is not NULL
    - Note that, e.g., (expression = NULL) cannot be used!



# NULL Values

- **Impact of NULL values**
  - NULL values were introduced to handle **missing information**
  - But how such values should act in functions a predicates?
- **When a function (or operator) cannot be evaluated, NULL is returned**
  - For example:  $3 + \text{NULL}$  is evaluated as **NULL**
- **When a predicate cannot be evaluated**, special logical value **UNKNOWN** is returned
  - For example:  $3 < \text{NULL}$  is evaluated to **UNKNOWN**
  - This means we need to work with a **three-value logic**
    - TRUE, FALSE, UNKNOWN

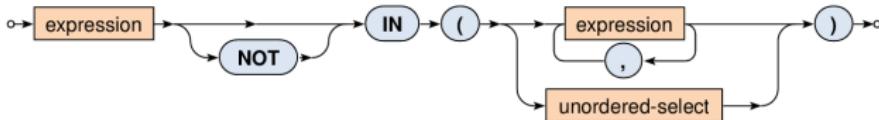
# Three-Value Logic

- Truth tables

p	q	p AND q	p OR q	NOT q
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE
TRUE	UNKNOWN	UNKNOWN	TRUE	UNKNOWN
FALSE	TRUE	FALSE	TRUE	
FALSE	FALSE	FALSE	FALSE	
FALSE	UNKNOWN	FALSE	UNKNOWN	
UNKNOWN	TRUE	UNKNOWN	TRUE	
UNKNOWN	FALSE	FALSE	UNKNOWN	
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	

# Search Conditions

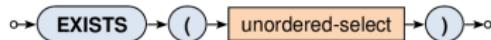
- **Set membership predicate**
  - Tests whether a value exists in a given set of values
    - Example: Company IN ('KLM', 'Emirates')



- Note that...
  - ... IN ( $\emptyset$ ) = FALSE
    - $\emptyset$  represents an empty table
  - ... IN ( $\aleph$ ) = UNKNOWN
    - $\aleph$  represents any table having rows with only NULL values

# Search Conditions

- **Existential quantifier predicate**
  - Tests whether a given set is not empty
  - Can be used to simulate the universal quantifier too
    - $\forall$  corresponds to  $\neg\exists\neg$



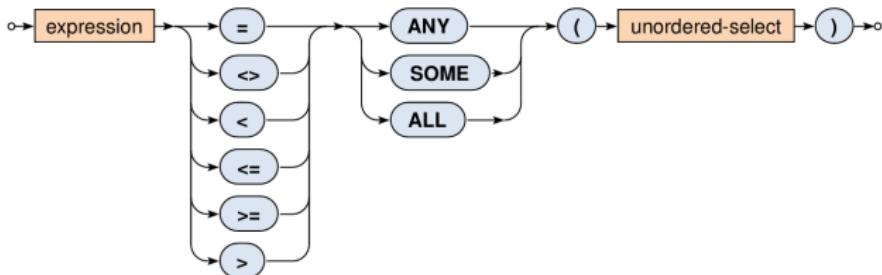
- Note that...
  - $\text{EXISTS } (\emptyset) = \text{FALSE}$
  - $\text{EXISTS } (X) = \text{TRUE}$

# Search Conditions

- Set comparison predicates

- ALL

- All the rows from the nested query must satisfy the operator
- $\text{ALL } (\emptyset) = \text{TRUE}$
- $\text{ALL } (\aleph) = \text{UNKNOWN}$



# Search Conditions

- Set comparison predicates
  - ANY and SOME (synonyms)
    - At least one row from the nested query must satisfy the given comparison operator
    - $\text{ANY } (\emptyset) = \text{FALSE}$
    - $\text{ANY } (\mathbb{N}) = \text{UNKNOWN}$

# From Clause

- SELECT ... **FROM** ... WHERE ... ORDER BY ...
  - **Description of tables to be queried**
    - Actually not only tables, but also **nested queries or views**
  - **Old way**
    - Comma separated **list of tables** (...)
    - **Cartesian product** of their rows is assumed
    - Required join conditions are specified in the WHERE clause
    - Example: **SELECT ... FROM Flights, Aircrafts WHERE ...**
  - **New way**
    - Usage of **join operators** with optional conditions
    - Example: **SELECT ... FROM Flights JOIN Aircrafts WHERE ...**

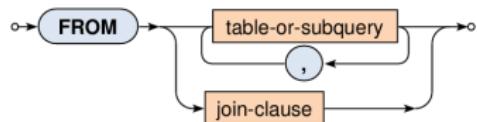
# From Clause

- SELECT ... **FROM** ... WHERE ... ORDER BY ...

- Description of tables to be queried

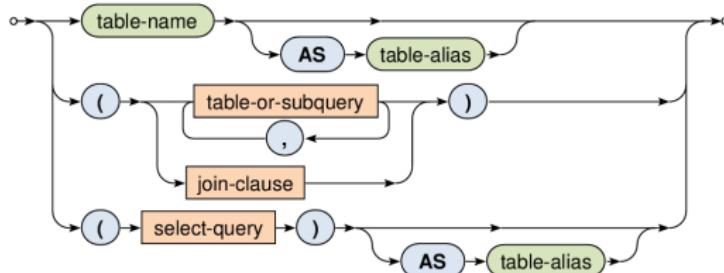
- Overall diagram

- Both old and new ways



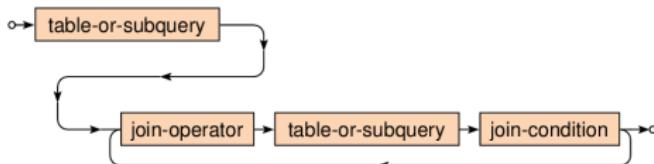
- Tables and subqueries

- Table name, auxiliary parentheses, direct select statement



# From Clause

- SELECT ... **FROM** ... WHERE ... ORDER BY ...
  - Description of tables to be queried
    - Basic structure of joins



- Examples
  - » Flights NATURAL JOIN Aircrafts
  - » Flights JOIN Aircrafts USING (Company)
  - » ...

- What types of joins are we provided?

# Table Joins

- **Cross join**

- Cartesian product of all the rows from both the tables



- **SELECT \* FROM T1 CROSS JOIN T2**

A	T1.*	A	T2.*	T1.A	T1.*	T2.A	T2.*
1	...	1	...	1	...	1	...
2	...	4	...	1	...	4	...
3	...			2	...	1	...
				2	...	4	...
				3	...	1	...
				3	...	4	...

# Table Joins

- **Natural join**
  - Pairs of rows are combined only when they have equal values in all the columns they share
    - I.e. columns of the same name



- **SELECT \* FROM T1 NATURAL JOIN T2**

The diagram illustrates a natural join between two tables, T1 and T2. On the left, there are two separate tables: T1 and T2. T1 has columns A and T1.\*. T2 has columns A and T2.\*. An arrow points from the joined tables to the result table. The result table has columns A, T1.\*, and T2.\*.

A	T1.*
1	...
2	...
3	...

A	T2.*
1	...
4	...

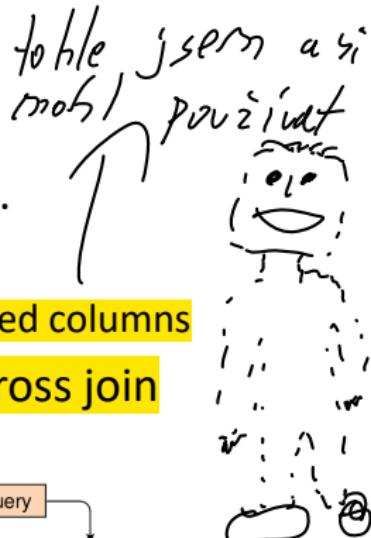
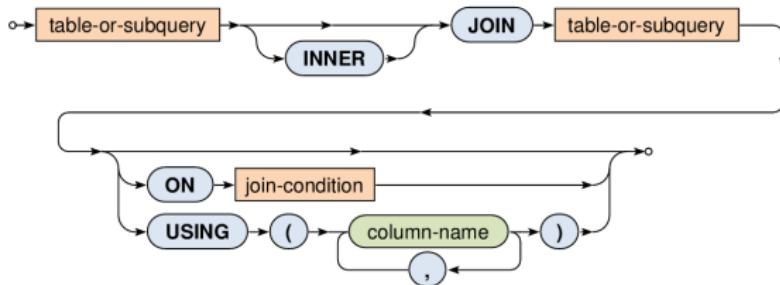


A	T1.*	T2.*
1	...	
		...
3	...	

# Table Joins

## • Inner join

- Pairs of rows are combined only when...
  - ON:** ... they satisfy the given join condition
  - USING:** ... they have equal values in the listed columns
- Note that inner join is a subset of the cross join



# Table Joins

- **Inner join**

- **SELECT \* FROM T1 JOIN T2 ON (T1.A <= T2.A)**

The diagram illustrates the execution of an inner join. On the left, two tables are shown: T1 (blue header) and T2 (purple header). T1 has columns A and T1.\* with rows 1, 2, and 3. T2 has columns A and T2.\* with rows 1 and 4. An arrow points from the resulting joined table on the right to the original tables on the left. The joined table has columns T1.A, T1.\*, T2.A, and T2.\*. It contains the following data:

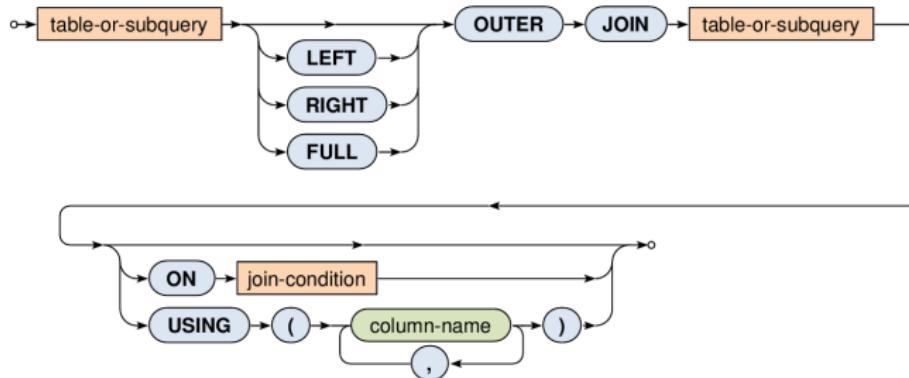
T1.A	T1.*	T2.A	T2.*
1	...	1	...
1	...	4	...
2	...		
3	...		
2	...	4	...
3	...	4	...

- **SELECT \* FROM T1 JOIN T2 USING (A)**
    - Equals to the corresponding natural join
  - **SELECT \* FROM T1 JOIN T2**
    - Equals to the corresponding cross join

# Table Joins

- Outer join

- Pairs of rows from the standard inner join + rows that cannot be combined, in particular, ...
  - **LEFT / RIGHT**: ... rows from the left / right table only
  - **FULL** (default): ... rows from both the tables



# Table Joins

- Outer join

- Note that...

- NULL values are used to fill missing information in rows that could not be combined

- SELECT \*

FROM T1 LEFT OUTER JOIN T2 ON (T1.A = T2.A)

The diagram illustrates a Left Outer Join operation. It shows three tables: T1, T2, and the resulting table after the join.

**T1:** A table with columns A and T1.\*. It contains three rows with values 1, 2, and 3 respectively, and ellipses for other columns.

**T2:** A table with columns A and T2.\*. It contains two rows with values 1 and 4 respectively, and ellipses for other columns.

**Resulting Table:** A table with columns T1.A, T1.\*, T2.A, and T2.\*. It has four rows. The first row (T1.A=1) has values 1, ..., 1, ... respectively. The second row (T1.A=2) has values 2, ..., NULL, NULL respectively. The third and fourth rows (T1.A=3) have values 3, ..., NULL, NULL respectively.

A	T1.*	A	T2.*	T1.A	T1.*	T2.A	T2.*
1	...	1	...	1	...	1	...
2	...	4	...	2	...	NULL	NULL
3	...			3	...	NULL	NULL

# Table Joins

- **Union join**

- Rows of both tables are integrated into one table,  
no pairs of rows are combined together at all

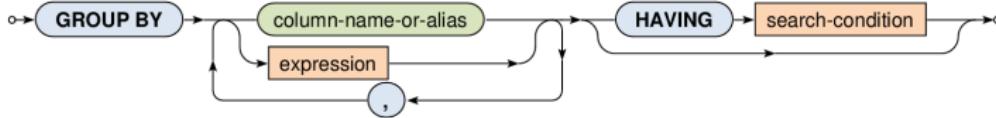


- **SELECT \* FROM T1 UNION JOIN T2**

A	T1.*	A	T2.*		T1.A	T1.*	T2.A	T2.*
1	...	1	...		1	...	NULL	NULL
2	...	4	...		2	...	NULL	NULL
3	...				3	...	NULL	NULL
					NULL	NULL	1	...
					NULL	NULL	4	...

# Aggregations

- Basic idea of table aggregation
  - First...
    - **FROM** and **WHERE** clauses are evaluated in a standard way
      - This results into an intermediate table
  - Then...
    - **GROUP BY**: rows of this table are divided into groups according to equal values over all the specified columns
    - **HAVING**: and, finally, these aggregated rows (superrows) can be filtered out using a provided search condition



# Aggregations: Example

- How many flights does each company have scheduled?
  - However, we are not interested in flights to Stuttgart and Munich
  - As well as we do not want companies with just one flight or less

```
SELECT Company, COUNT(*) AS Flights FROM Flights  
WHERE (Destination NOT IN ('Stuttgart', 'Munich'))  
GROUP BY Company HAVING (Flights > 1)
```

The diagram illustrates the flow of data through three stages:

- Source Table:** A table with columns Flight, Company, Destination, and Passengers. It contains the following data:

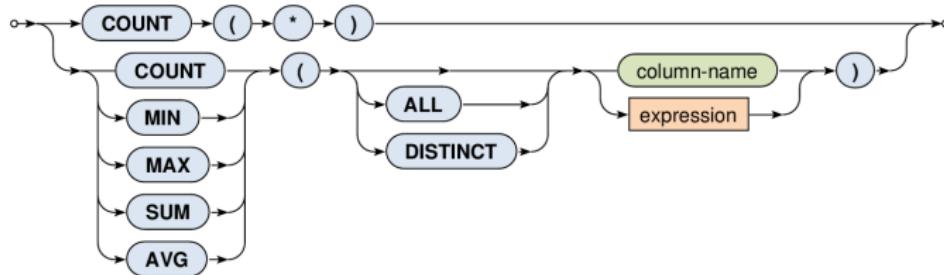
Flight	Company	Destination	Passengers
OK251	CSA	New York	276
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
OK321	CSA	London	156
AC906	Air Canada	Toronto	116
KL7621	KLM	Rotterdam	75
KL1245	KLM	Amsterdam	130
- Intermediate State:** A table where rows for 'Lufthansa' and 'KLM' are removed. The remaining data is grouped by Company.

Flight	Company	Destination	Passengers
OK251	CSA	New York	276
OK012	CSA	Milano	37
OK321	CSA	London	156
AC906	Air Canada	Toronto	116
KL7621	KLM	Rotterdam	75
KL1245	KLM	Amsterdam	130
- Result Table:** A final table showing the count of flights for each company, where companies with only one flight are excluded.

Company	Flights
CSA	3
Air Canada	1
KLM	2

# Aggregations

- What columns can be used...
  - in the SELECT clause as well as in the HAVING clause
  - ... when table aggregation takes place?
  - Answer (for both the cases): only...
    - Aggregating columns (i.e. those from the GROUP BY clause)
    - Columns newly derived using aggregation functions



# Aggregations

- Aggregate functions

- Allow to produce values from the rows within a group
  - **COUNT(\*)**
    - Number of all the rows including duplicates and `NULL` values
  - **COUNT / SUM / AVG / MIN / MAX**
    - Number of values / sum of values / average / min / max
    - `NULL` values are always and automatically ignored
    - Modifier **ALL** (default) includes duplicates, **DISTINCT** not
    - $\text{COUNT}(\emptyset) = 0$
    - $\text{SUM}(\emptyset) = \text{NULL}$  (which is strange!)
    - $\text{AVG}(\emptyset) = \text{NULL}$ ,  $\text{MIN}(\emptyset) = \text{NULL}$ ,  $\text{MAX}(\emptyset) = \text{NULL}$

# Aggregations: Example

- Find basic characteristics for all the scheduled flights
  - i.e. return the overall number of flights, the overall number of the involved companies, the sum of all the passengers, the average / minimal / maximal number of passengers

*specified in the select.*

**SELECT**

```
COUNT(*) AS Flights,  
COUNT(DISTINCT Company) AS Companies,  
SUM(Passengers) AS PSum,  
AVG(Passengers) AS PAvg,  
MIN(Passengers) AS PMin,  
MAX(Passengers) AS PMax
```

**FROM** Flights

Flight	Company	Destination	Passengers
OK251	CSA	New York	276
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
OK321	CSA	London	156
AC906	Air Canada	Toronto	116
KL7621	KLM	Rotterdam	75
KL1245	KLM	Amsterdam	130

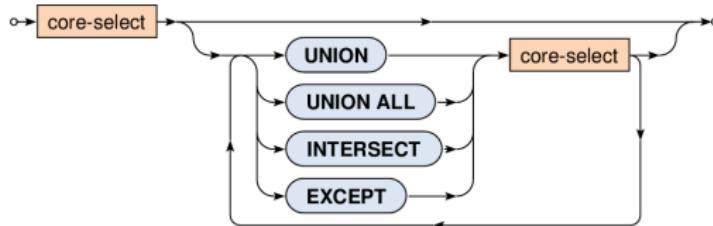


Flights	Companies	PSum	PAvg	PMin	PMax
7	4	858	123	37	276

# Set Operations

- Available **set operations**

- **UNION** – union of two tables (without duplicities)
- **UNION ALL** – union of two tables (with duplicities)
- **INTERSECT** – intersection of two tables
- **EXCEPT** – difference of two tables



# Set Operations: Example

- Merge available companies from tables of flights and aircrafts

```
SELECT Company FROM Flights
```

**UNION**

```
SELECT Company FROM Aircrafts
```

Company
CSA
Lufthansa
Air Canada
KLM

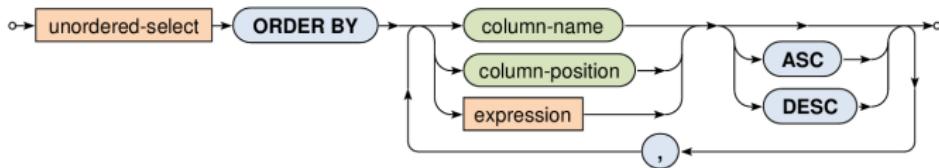
- Note that...

- Both the operands must be compatible
    - I.e. they have the same number of columns
    - And these columns must be of the same types

# Ordered Queries

- **ORDER BY**

- Note that **rows in the result have no defined order!**
  - ... unless this order is explicitly specified
- Multiple columns (...) can be used for such order
- NULL values precede any other values
- Directions
  - **ASC** (default) – ascending
  - **DESC** – descending



# Ordered Queries: Example

- Return an ordered list of all the scheduled destinations

```
SELECT DISTINCT Destination  
FROM Flights  
ORDER BY Destination ASC
```

Flight	Company	Destination	Passengers
OK251	CSA	New York	276
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
OK321	CSA	London	156
AC906	Air Canada	Toronto	116
KL7621	KLM	Rotterdam	75
KL1245	KLM	Amsterdam	130



Destination
Amsterdam
London
Milano
New York
Rotterdam
Stuttgart
Toronto

# Nested Queries

- **Where the nested queries can be used?**
  - In predicates...
    - ANY, SOME, ALL
    - IN
    - EXISTS
  - For definition of **tables** in the **FROM clause**
  - Almost in any expression if scalar values are produced

# Nested Queries: Example

- Find all the scheduled flights which have higher than average number of passengers.

```
SELECT *
FROM Flights
WHERE (Passengers > (SELECT AVG(Passenger) FROM Flights))
```

Flight	Company	Destination	Passenger
OK251	CSA	New York	276
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
OK321	CSA	London	156
AC906	Air Canada	Toronto	116
KL7621	KLM	Rotterdam	75
KL1245	KLM	Amsterdam	130



Flight	Company	Destination	Passenger
OK251	CSA	New York	276
OK321	CSA	London	156
KL1245	KLM	Amsterdam	130

# Nested Queries: Example

- Return the number of suitable aircrafts for each flight.
  - Only aircrafts of a given company and sufficient capacity can be used
  - Note how values from the outer query are bound with the inner one

**SELECT**

```
Flights.*,
(
    SELECT COUNT(*)
    FROM Aircrafts AS A
    WHERE
        (A.Company = F.Company) AND
        (A.Capacity >= F.Passengers)
) AS Aircrafts
FROM Flights AS F
```

Flight	Company	Destination	Passengers	Aircrafts
OK251	CSA	New York	276	0
LH438	Lufthansa	Stuttgart	68	0
OK012	CSA	Milano	37	1
OK321	CSA	London	156	0
AC906	Air Canada	Toronto	116	0
KL7621	KLM	Rotterdam	75	2
KL1245	KLM	Amsterdam	130	2

Aircraft	Company	Capacity
Boeing 717	CSA	106
Airbus A380	KLM	555
Airbus A350	KLM	253



B0B36DBS, BD6B36DBS: **Database Systems**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/>

Lecture 5

# **SQL: Advanced Constructs**

**Martin Svoboda**

[martin.svoboda@fel.cvut.cz](mailto:martin.svoboda@fel.cvut.cz)

17. 3. 2020

**Czech Technical University in Prague, Faculty of Electrical Engineering**

# Outline

- **SQL**
  - **Views**
  - Embedded SQL
    - **Functions** (stored procedures)
    - **Cursors**
    - **Triggers**
  - **SQL/XML**
    - Manipulation with XML data

# **Views**

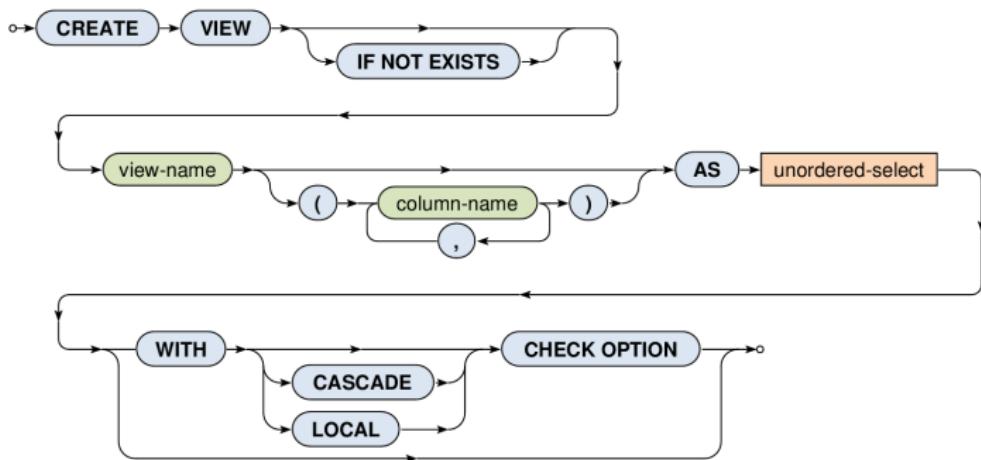
# Database Views

- What are views?
  - Named SELECT queries
    - They can be used similarly as tables
      - E.g. in the FROM clause of the SELECT statements
    - Evaluated dynamically
  - Motivation for views
    - Creation of virtual tables, security reasons (hiding tables and their content from particular users), repeated usage of the same complicated statements, ...
  - Content of views can be updatable
    - But only sometimes!

# Database Views

- **CREATE VIEW**

- View name and optionally names of its columns
- Select query and check option



# Database Views

- **View updatability**
  - I.e. can rows be inserted / updated in a view?
  - Yes, but only when...
    - It makes sense...
      - I.e. the given view is based on a **simple SELECT query** (without aggregations, subqueries, ...) with only projections (without derived values, ...) and selections **over right one table** (without joins, ...)
      - I.e. we are deterministically able to reconstruct the entire tuples to be inserted / updated in the original table(s)
    - And, moreover, ...

# Database Views

- **View updatability**
  - ...
  - When **WITH CHECK OPTION** is specified
    - Then the **newly inserted / updated tuples must be visible...**
      - **LOCAL** – in the **given view**
      - **CASCADE** (default) – in the given view as well **as all the other views this given one is derived from (depends on)**

# Database Views



- **Examples**

- View creation

```
CREATE VIEW BigPlanes AS  
SELECT * FROM Aircrafts WHERE (Capacity > 200)  
WITH LOCAL CHECK OPTION
```

- Successful insertion

```
INSERT INTO BigPlanes  
VALUES ('Boeing 737', 'CSA', 201);
```

- Denied insertion

```
INSERT INTO BigPlanes  
VALUES ('Boeing 727', 'CSA', 100);
```

- This aircraft is only too small (will not be visible in the view)

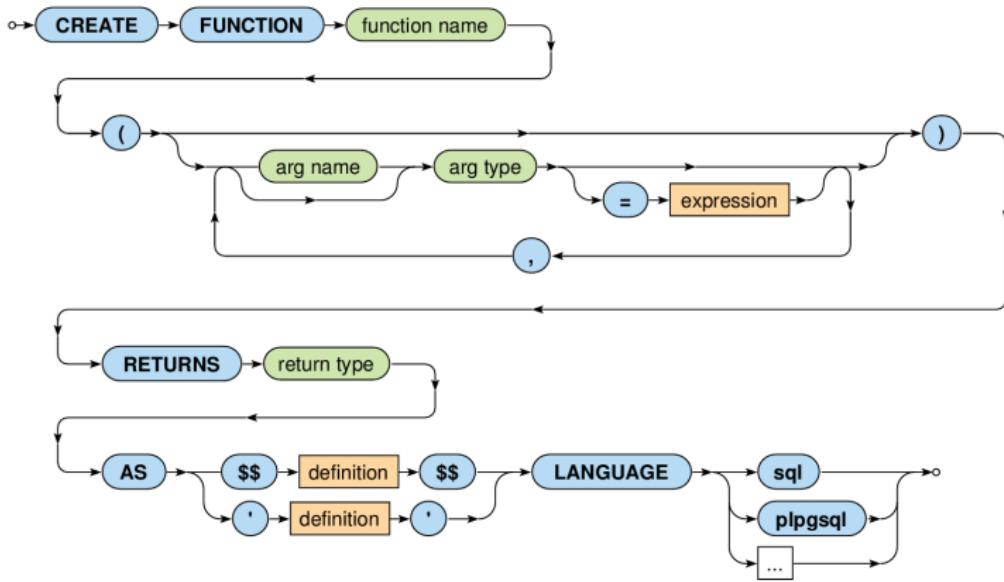
# **Embedded SQL**

# Embedded SQL

- **Internal database applications** = jíklož aplikace v rámci  
data base.
  - Proprietary procedural extensions of SQL
    - Transact SQL (T-SQL) – Microsoft SQL Server
    - PL/SQL – Oracle Database
    - **PL/pgSQL – PostgreSQL** ↗ PostgreSQL je me povídá
  - Available constructs
    - Control statements: **if then else, for, while, switch**
    - **Stored procedures**
    - **Cursors** – iterative scanning of tables
    - **Triggers** – general integrity constraints
    - ...

# Stored Procedures

- **CREATE FUNCTION** – defines a new function



# Stored Procedures: Example

```
CREATE FUNCTION inc(x INT)
RETURNS INT
AS
$$
BEGIN
    RETURN x + 1;
END;
$$
LANGUAGE plpgsql;

SELECT inc(5);
```

# Cursors

- **Cursor declaration**

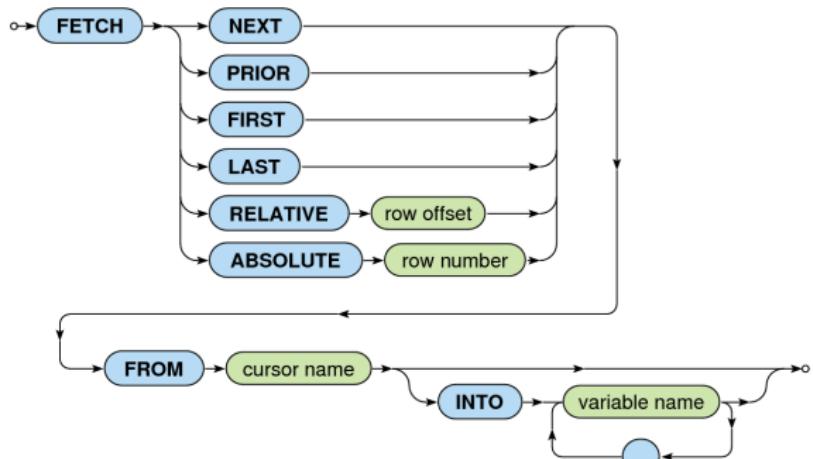
- Database cursor is a **control structure** that allows us to **traverse the rows of a selected table**



- **SCROLL option**
  - When specified, even **backward fetches** are permitted
  - Otherwise only **forward fetches** are allowed

# Cursors

- Data retrieval



- **INTO** clause

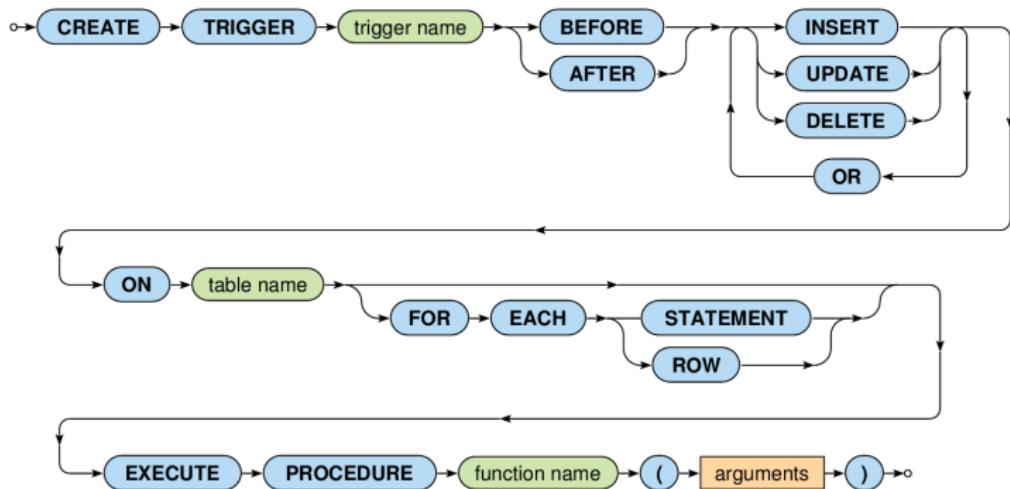
- Local variables into which a given row will be stored
- NULL values are returned when there is no additional row

# Triggers

- **CREATE TRIGGER** → něco jako Event Listener.
  - Trigger is a procedure that is automatically executed as a response to certain events (`INSERT, UPDATE, DELETE`)
    - Used for maintaining complex integrity constraints
  - Modes
    - FOR EACH STATEMENT (default mode)
      - Trigger will be invoked only once for all the rows involved in a given query
    - FOR EACH ROW
  - Procedure
    - Return type must be TRIGGER

# Triggers

- **CREATE TRIGGER**



# **SQL/XML**

# XML Documents: Example

```
<?xml version="1.0"?>
<library>
  <book id="1" catalogue="c1" language="en">
    <title>Red</title>
    <author>John</author>
    <author>Peter</author>
  </book>
  <book id="2" catalogue="c1">
    <title>Green</title>
    <price>25</price>
  </book>
  <book id="3" catalogue="c2" language="en">
    <title>Blue</title>
    <author>John</author>
  </book>
</library>
```

# Introduction

- **SQL/XML**
  - **Extension to SQL for XML data**
    - XML Datatype
    - Constructs
      - Functions, constructors, mappings, XQuery embedding, ...
- Standards
  - **SQL:2011-14 (ISO/IEC 9075-14:2011)**
    - Older versions 2003, 2006, 2008

# Example

- **Table:** books

<b>id</b>	<b>catalogue</b>	<b>title</b>	<b>details</b>	<b>language</b>
1	c1	Red	<author>John</author> <author>Peter</author>	en
2	c1	Green	<price>25</price>	NULL
3	c2	Blue	<author>John</author>	en

- **Table:** languages

<b>code</b>	<b>name</b>
en	English
cs	Czech

# Example

- **Query**

```
SELECT
    id,
    XMLEMENT (
        NAME "book",
        XMLEMENT (NAME "title", title),
        details
    ) AS book
FROM books
WHERE (language = "en")
ORDER BY title DESC
```

# Example

- Result

id	book
3	<book> <title>Blue</title> <author>John</author> </book>
1	<book> <title>Red</title> <author>John</author> <author>Peter</author> </book>

# XML Datatype

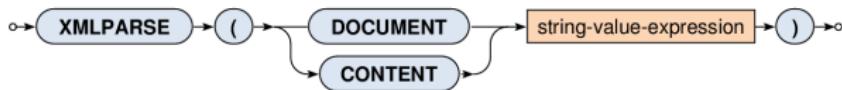
- Traditional types
  - BLOB, CLOB, VARCHAR, ...
- **Native XML type**
  - Collection of information items
    - Based on XML Information Set (**XML InfoSet**)
      - Elements, attributes, processing instructions, ...
      - But we also allow fragments without exactly one root element
        - » This means that XML values may not be XML documents
    - NULL

# Parsing XML Values

- **XMLPARSE**

- **Creates an XML value from a string**

- DOCUMENT – well-formed document with exactly one root
    - CONTENT – well-formed fragment



```
SELECT XMLPARSE (
```

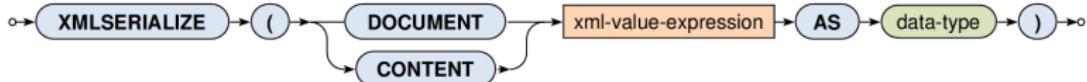
```
  DOCUMENT "<book><title>Red</title></book>"
```

```
) AS result
```

result
<book> <title>Red</title> </book>

# Serializing XML Values

- XMLSERIALIZE
  - Exports an XML value to a string



```
SELECT
    id, title,
    XMLSERIALIZE(CONTENT details AS VARCHAR(100)) AS export
FROM books
```

id	title	export
1	Red	<author>John</author><author>Peter</author>
...	...	...

# Well-Formedness Predicate

- IS DOCUMENT
  - **Tests whether an XML value is an XML document**
    - Returns TRUE if there is right one root element
    - Otherwise FALSE



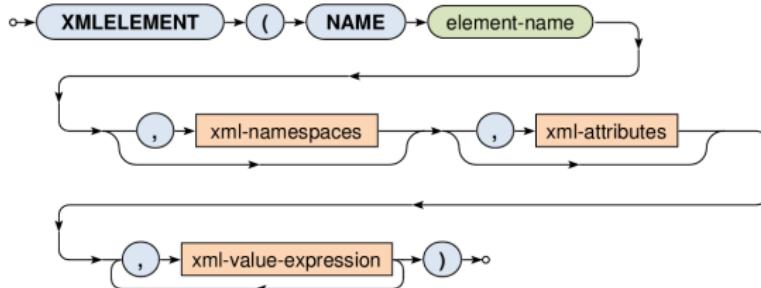
# Constructors

- Functions for construction of XML values...
  - **XMLEMENT** – elements
  - **XMLNAMESPACES** – namespace declarations
  - **XMLATTRIBUTES** – attributes
  - **XMLCOMMENT** – comments
  - **XMLPI** – processing instructions
  - **XMLFOREST** – sequences of elements
  - **XMLCONCAT** – concatenations of values
  - **XMLAGG** – aggregates

# Elements

- XMLEMENT

- Creates an XML element with a given name and...
  - optional **namespace declarations**
  - optional **attributes**
  - optional **element content**



# Elements: Example 1

```
SELECT
    id,
    XMLEMENT (NAME "book", title) AS result
FROM books
ORDER BY id
```

id	result
1	<book>Red</book>
2	<book>Green</book>
3	<book>Blue</book>

# Elements: Example 2: Subelements

```
SELECT
```

```
    id,  
    XMLELEMENT (  
        NAME "book",  
        XMLELEMENT (NAME "title", title),  
        XMLELEMENT (NAME "language", language)  
    ) AS records
```

```
FROM books
```

id	records
1	<book> <title>Red</title> <language>en</language> </book>
...	...

# Elements: Example 3: Mixed Content

```
SELECT
    id,
    XMLELEMENT(
        NAME "info",
        "Book ", XMLELEMENT(NAME "title", title),
        " with identifier equal to", id, "."
    ) AS description
FROM books
```

id	description
1	<info> Book <title>Red</title> with identifier equal to 1. </info>
...	...

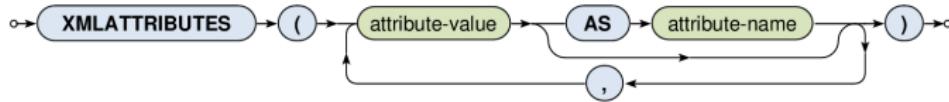
# Elements: Example 4: Subqueries

```
SELECT
    id,
    XMLELEMENT(NAME "title", title) AS book,
    XMLELEMENT(
        NAME "language",
        (SELECT name FROM languages WHERE (code = language))
    ) AS description
FROM books
```

<b>id</b>	<b>book</b>	<b>description</b>
1	<title>Red</title>	<language>English</language>
...	...	...

# Attributes

- XMLATTRIBUTES
  - Creates a set of attributes
  - Input: list of values
    - Each value must have an **explicit / implicit name**
      - It is used as a name for the given attribute
      - Implicit names can be derived, e.g., from column names
  - Output: XML value with a set of attributes



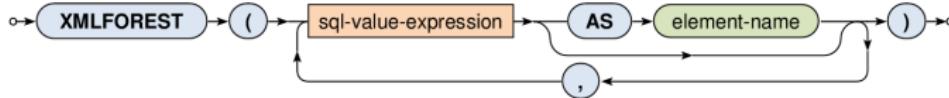
# Attributes: Example

```
SELECT
    id,
    XMLELEMENT(NAME "book",
        XMLATTRIBUTES (
            language, catalogue AS "location"
        ),
        XMLELEMENT(NAME "title", title)
    ) AS book
FROM books
```

id	book
1	<book language="en" location="c1">             <title>Red</title>         </book>
...	...

# Element Sequences

- XMLFOREST
  - Creates a sequence of XML elements
  - Input: list of SQL values
    - Individual content expressions evaluated to NULL are ignored
    - If all the expressions are evaluated to NULL, then NULL is returned
    - Each content value must have an **explicit / implicit name**
      - It is used as a name for the given element
  - Output: XML value with a sequence elements



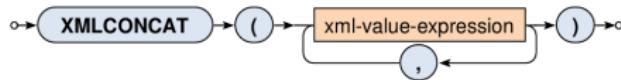
# Element Sequences: Example

```
SELECT
    id,
    XMLFOREST(
        title, language, catalogue AS location
    ) AS book
FROM books
```

id	book
1	<title>Red</title> <language>en</language> <location>c1</location>
2	<title>Green</title> <location>c1</location>
***	***

# Concatenation

- XMLCONCAT
  - **Creates a sequence from a list of values**
  - **Input:** list of XML values
    - Individual content expressions evaluated to `NULL` are ignored
    - If all the expressions are evaluated to `NULL`, then `NULL` is returned
  - **Output:** XML value with a sequence of values



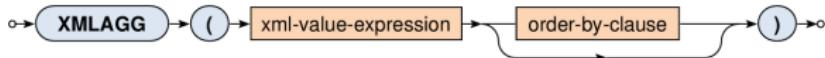
# Concatenation: Example

```
SELECT
    id,
    XMLCONCAT(
        XMLELEMENT(NAME "book", title),
        details
    ) AS description
FROM books
```

id	description
1	<book>Red</book> <author>John</author> <author>Peter</author>
...	...

# XML Aggregation

- XMLEGG
  - **Aggregates rows within a given super row**
    - I.e. acts as a standard aggregate function (like SUM, AVG, ...)
  - **Input: rows within a given super row**
    - These rows can first be optionally sorted (**ORDER BY**)
    - For each row an XML value is generated as described
      - Individual rows evaluated to **NULL** values are ignored
    - All the generated XML values are then concatenated
      - If all the rows are evaluated to **NULL**, then **NULL** is returned
  - **Output: XML value with a sequence of items**



# XML Aggregation: Example

```
SELECT
    catalogue,
    XMLAGG (
        XMLELEMENT (NAME "book", XMLATTRIBUTES (id),
                    title)
        ORDER BY id
    ) AS list
FROM books
GROUP BY catalogue
```

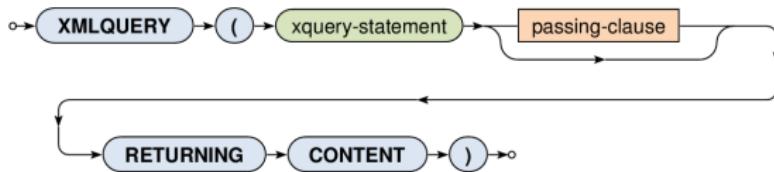
catalogue	list
c1	<book id="1">Red</book> <book id="2">Green</book>
c2	<book id="3">Blue</book>

# Querying

- Query constructs
  - Based on XQuery language
  - **XMLQUERY** – returns query result
    - Usually in SELECT clauses
  - **XMLTABLE** – decomposes query result into a table
    - Usually in FROM clauses
  - **XMLEXISTS** – tests query result non-emptiness
    - Usually in WHERE clauses

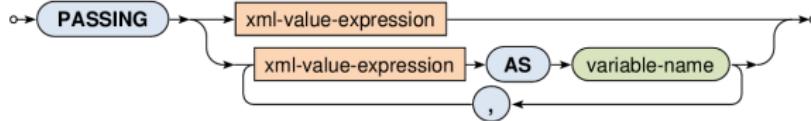
# XQuery Statements

- XMLQUERY
  - Evaluates an XQuery statement and returns its result
  - Input:
    - XML values declared in an optional PASSING clause
  - Output: XML value



# XQuery Statements

- XMLQUERY
  - Input data
    - When **only one input value** is specified...
      - its content is accessible via / inside the XQuery statement
    - When **one or more named variables** are specified...
      - their content is accessible via \$variable-name/



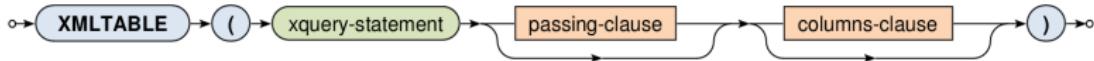
# XQuery Statements: Example

```
SELECT
    id, title,
    XMLQUERY (
        "<authors>{ count($data/author) }</authors>"
        PASSING details AS data
        RETURNING CONTENT
    ) AS description
FROM books
```

id	title	description
1	Red	<authors>2</authors>
***	***	***

# XML Tables

- XMLTABLE
  - Decomposes an XQuery result into a virtual table
  - Output:
    - When **COLUMNS** clause is specified...
      - Table containing the XQuery result being shredded into individual rows and columns according to the description
    - Otherwise...
      - Table with one row and one column with the XQuery result represented as an XML value



# XML Tables: Example 1

```
SELECT
    id, title, result.*
FROM
    books,
XMLTABLE (
    "<authors>{ count($data/author) }</authors>"
    PASSING books.details AS data
) AS result
```

<b>id</b>	<b>title</b>	<b>result</b>
1	Red	<authors>2</authors>
...	...	...

# XML Tables: Example 2

```
SELECT
    id, title, result.count
FROM
    books,
XMLTABLE (
    "<authors>{ count($data/author) }</authors>"
    PASSING books.details AS data
    COLUMNS
        count INTEGER PATH "authors/text()"
) AS result
```

<b>id</b>	<b>title</b>	<b>count</b>
1	Red	2
...	...	...

# Exists Predicate

- XMLEXISTS
  - Tests an XQuery statement result for non-emptiness
  - Output: Boolean value
    - Returns TRUE for result sequences that are not empty
    - Otherwise FALSE



# Exists Predicate: Example

```
SELECT books.*  
FROM books  
WHERE  
XMLEXISTS (  
    "/author"  
    PASSING details  
)
```

<b>id</b>	<b>catalogue</b>	<b>title</b>	<b>details</b>	<b>language</b>
1	c1	Red	<author>John</author> <author>Peter</author>	en
3	c2	Blue	<author>John</author>	en



# Accessing Database System

Martin Řimnáč

extension of slides  
by Zdeněk Kouba and Petr Křemen

# Accessing Database System

- client - server architecture
- heterogeneous data types and data structure
- applications can use many data sources
- need to standardize
  - ODBC (Open DataBase Connectivity)
    - an interface for managing connection, authorization, query and result delivery
    - in java: JDBC
  - ORM (Object Relational Mapping)
    - direct usage relational data in object oriented programming
    - In java: JPA (Java Persistence API)

# ODBC (Open Database Connection)

- makes an application independent on
  - Database Management System and its version
  - operating system (enable ports to various platforms)
- introduced in MS Windows in early 1990s
- steps:
  - create connection
  - create (prepared) statement
  - execute statement
  - result browsing
  - closing connection

# ODBC – create a connection

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class SlonDataTutorial {
    protected Connection connection;
    protected function createConnection (connectionString, userName, userPassword)
    {
        Class.forName("org.postgresql.Driver");
        this.connection = DriverManager.getConnection(connectionString,userName,userPassword);
    }
    public function execute ()
    {
        try {
            this.createConnection ("jdbc:postgresql://slon.felk.cvut.cz:5432/tutoralexample",
                                  "tutoruser", "tutorpass");
        } catch (ClassNotFoundException e) {
            System.out.println("PostgreSQL JDBC driver not found."); e.printStackTrace();
        } catch (SQLException e) {
            System.out.println("Connection failure."); e.printStackTrace();
        }
    }
}
```

# ODBC – create a connection

- The ODBC driver is dynamically loaded
  - `Class.forName("org.postgresql.Driver");`
  - provides `DriverManager`
    - creates `Connection`
- The connection parameters are concerned into the connection string
  - `"jdbc:postgresql://slon.felk.cvut.cz:5432/tutoralexample"`,
    - Means: *Postgresql* DBMS, server *slon.felk.cvut.cz*, port *5432*, database *tutoralexample*
- Authorization
  - Database user *tutoruser* authorized by password *tutorpass*

# ODBC – execute a query

- The ODBC connection creates a Statement
  - the statement is executed by the query
  - returns ResultSet
- The ResultSet is navigated by next() method
  - Fields are accessed by the field names or position

```
protected function getAllItems ()  
{  
    Statement statement = this.connection.createStatement();  
    ResultSet resultSet = statement.executeQuery("SELECT model, price FROM item");  
    while (resultSet.next()) {  
        this.printItem (resultSet);  
    }  
}  
protected function printItem (ResultSet resultSet)  
{  
    System.out.printf("%-30.30s %-30.30s%n", resultSet.getString("model"), resultSet.getFloat("price"));  
}
```

# ODBC – execute query

- The queries can be parametrized
  - SQL Query is constructed as the string concatenation

```
protected function getItemById (int id)
{
    Statement st = this.connection.createStatement();
    ResultSet rs = st.executeQuery("SELECT model, price FROM item WHERE id_item = " + id);
    while (rs.next()) {
        this.printItem(rs);
    }
    rs.close();
    st.close();
}
```

# ODBC – execute query

- The queries can be parametrized
  - SQL Query is constructed as the string concatenation

```
protected function getItemById (int id)
{
    Statement st = this.connection.createStatement();
    ResultSet rs = st.executeQuery("SELECT model, price FROM item WHERE id_item = " + id);
    while (rs.next()) {
        this.printItem(rs);
    }
    rs.close();
    st.close();
}
```

- Never use this query construction – SQL Injection

# ODBC – SQL injection

```
protected function authorizeEndUser (String userName, String userPassword)
{
    Statement st = this.connection.createStatement();
    ResultSet rs = st.executeQuery("SELECT * FROM authorizeduser
        WHERE username=“ + userName + „ AND password = “ + userPassword + “““
    if (rs.next()) {
        this.setAuthorizedUser(rs);
    }
    rs.close();
    st.close();
}
```

- Authorization example

- Works perfect for userName=“user“ and userPassword=“heslo“

# ODBC – SQL injection

```
protected function authorizeEndUser (String userName, String userPassword)
{
    Statement st = this.connection.createStatement();
    ResultSet rs = st.executeQuery("SELECT * FROM authorizeduser
        WHERE username=“ + userName + „ AND password = “ + userPassword + “““
    if (rs.next()) {
        this.setAuthorizedUser(rs);
    }
    rs.close();
    st.close();
}
```

- ## Authorization example

- Works perfect for `userName="user"` and `userPassword="heslo"`
- Fails for `userPassword=" OR "="` because the value changes the statement
  - `Username = 'user' AND password=" OR "="`

- ## Required to use the prepared statement

# ODBC – prepared statement

- The query is initialised as a pattern
  - Query parametrization is provided by ?
  - setInt, setString, ... method for value substitution
  - Functionality: the parameter does not change the query

```
protected function getItemByIdSave (int idItem)
{
    PreparedStatement st = this.connection.prepareStatement
        ("SELECT model, price FROM item WHERE id_item = ?");
    st.setInt(1, idItem);
    ResultSet rs = st.executeQuery();
    while (rs.next()) {
        this.printItem (rs)
    }
    rs.close();
    st.close();
}
```

# Best Practice

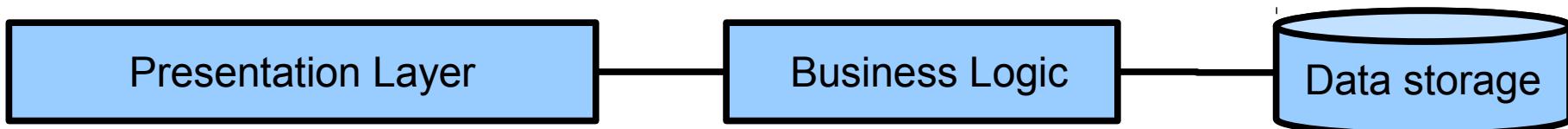
- Do not use \* in SELECTs, if not necessary
  - Eliminates transfer of unused attributes
- Use data paging (LIMIT/OFFSET)
  - Nobody wants to see all the items
  - Required also the result to be sorted
- Do not export artificial keys (id\_...)
  - Use the key value corresponding to reality in all external APIs (interoperability)
- Do not use INSERTs without attributes
  - Application is not resistant to data schema changes

# ORM and JPA 2.0

Zdeněk Kouba, Petr Křemen

# What is Object-relational mapping ?

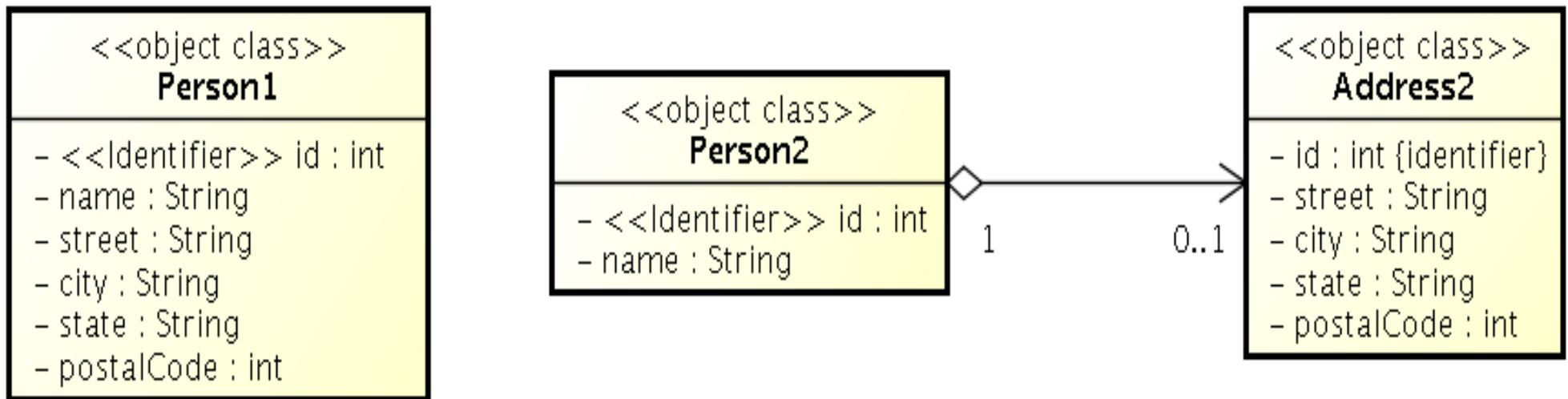
- a typical information system architecture:



- How to avoid data format transformations when interchanging data from the (OO-based) presentation layer to the data storage (RDBMS) and back ?
- How to ensure persistence in the (OO-based) business logic ?

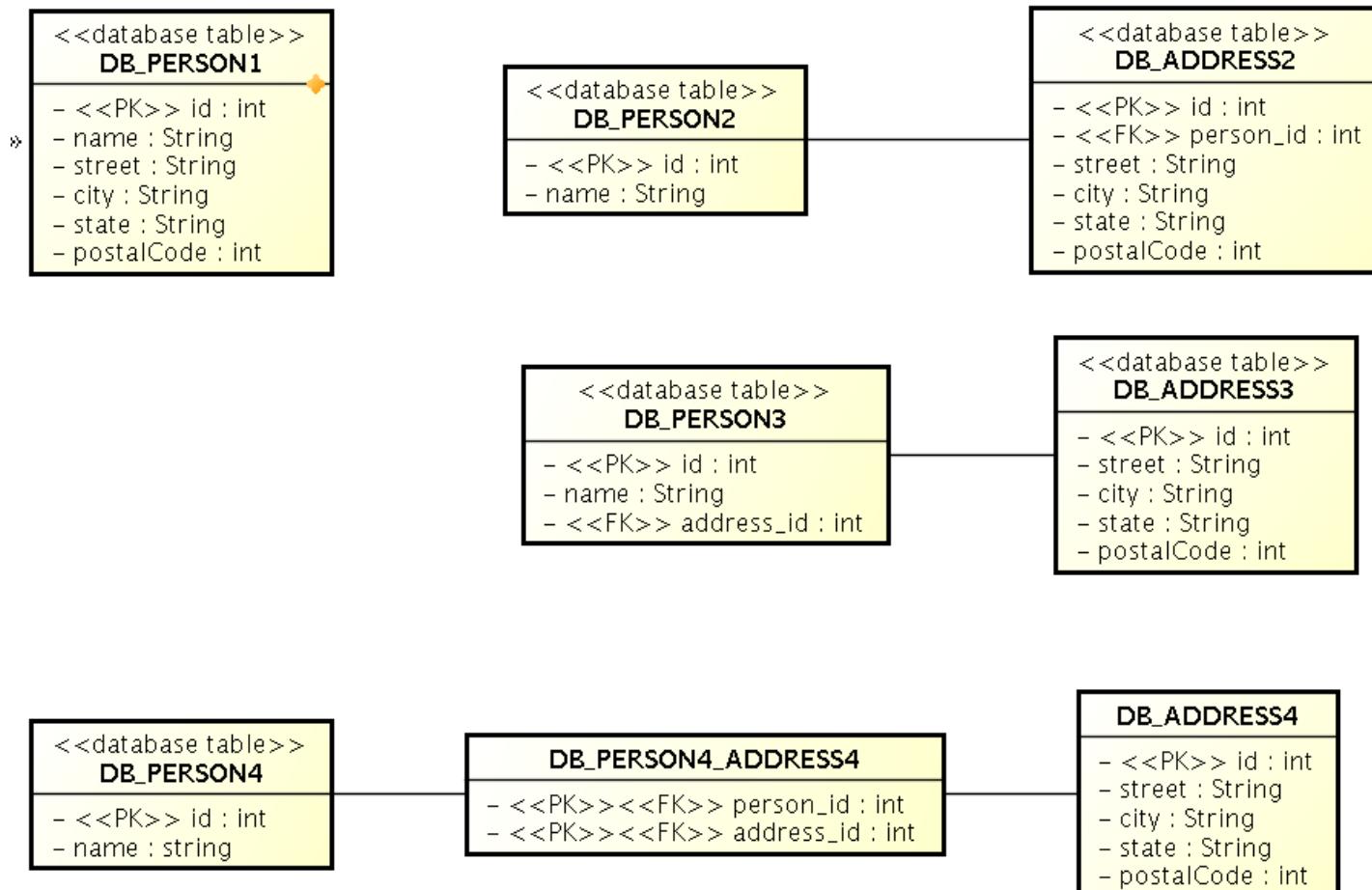
# Example – object model

- When would You stick to one of these options ?



# Example – database

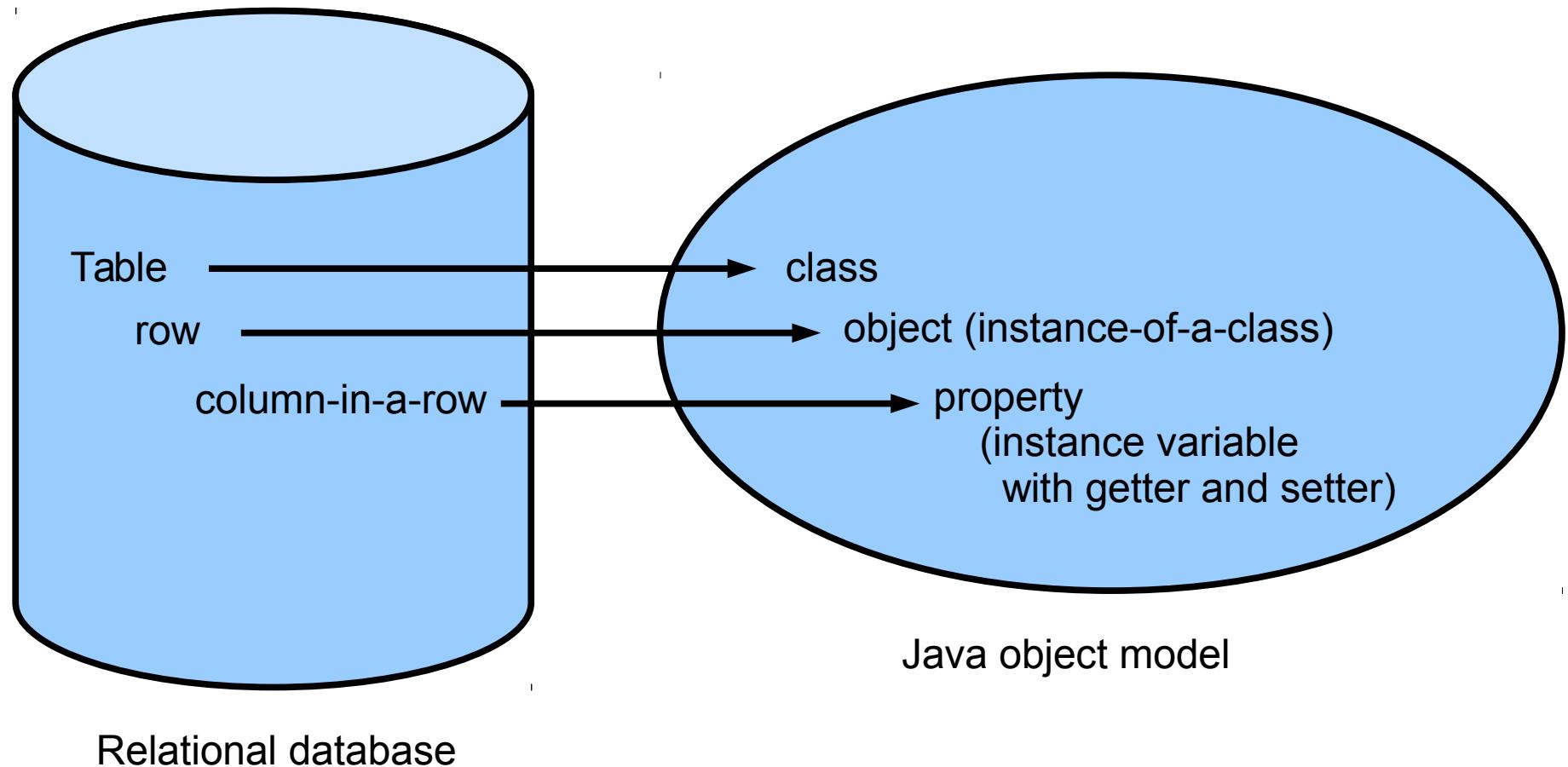
- ... and how to model it in SQL ?



# Object-relational mapping

- Mapping between the database (declarative) schema and the data structures in the object-oriented language.
- Let's take a look at JPA 2.0

# Object-relational mapping



# JPA 2.0

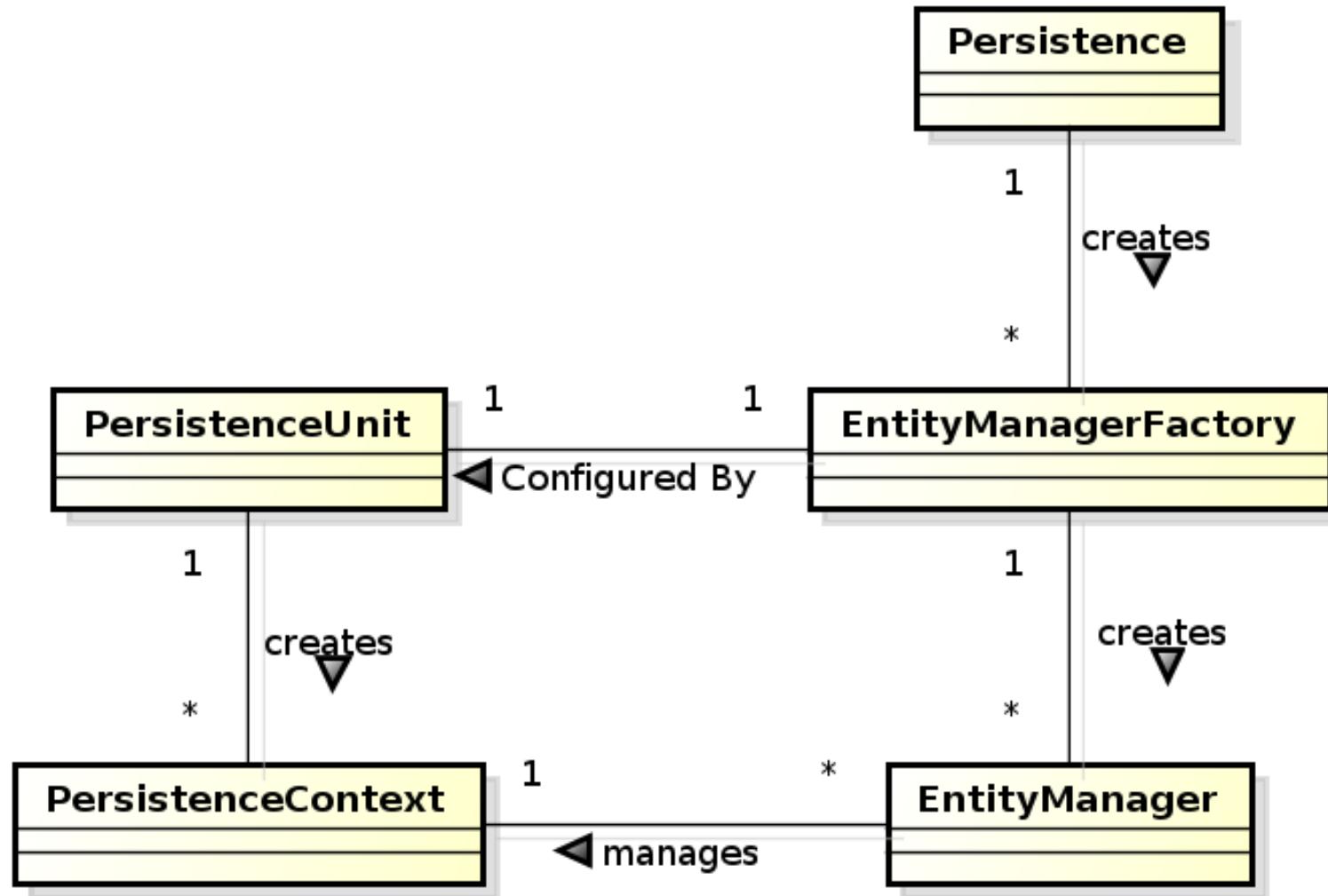
- Java Persistence API 2.0 (JSR-317)
- Although part of Java EE 6 specifications, JPA 2.0 can be used both in EE and SE applications.
- Main topics covered:
  - Basic scenarios
  - Controller logic – EntityManager interface
  - ORM strategies
  - JPQL + Criteria API

# JPA 2.0 – Entity Example

- Minimal example (configuration by exception):

```
@Entity  
  
public class Person {  
  
    @Id  
  
    @GeneratedValue  
  
    private Integer id;  
  
    private String name;  
  
    // setters + getters  
  
}
```

# JPA2.0 – Basic concepts

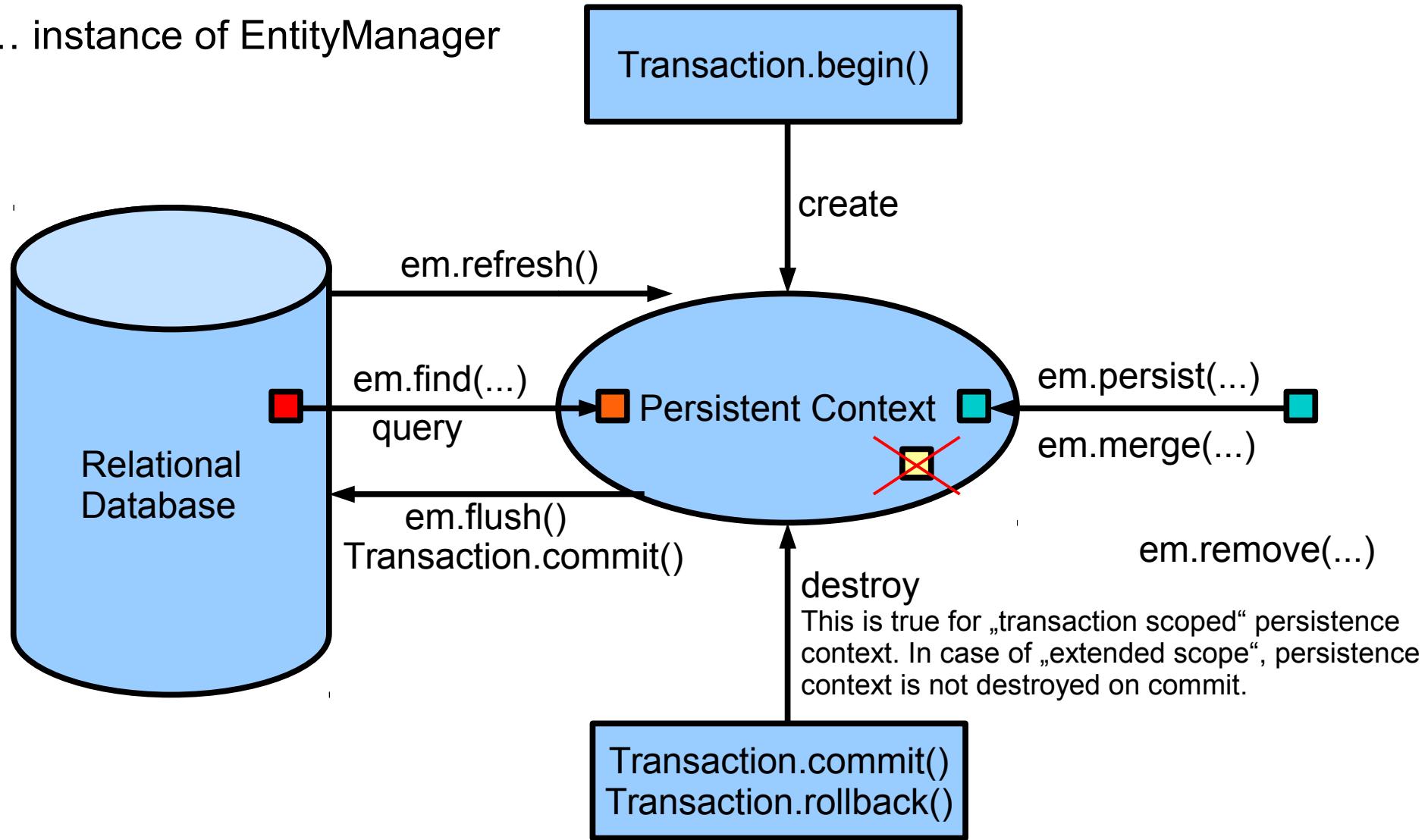


# JPA 2.0 - Basics

- Let's have a set of „suitably annotated“ POJOs, called **entities**, describing your domain model.
- A set of entities is logically grouped into a **persistence unit**.
- JPA 2.0 providers :
  - generate persistence unit from existing database,
  - generate database schema from existing persistence unit.
  - TopLink (Oracle) ... JPA
  - EclipseLink (Eclipse) ... JPA 2.0
- What is the benefit of the keeping Your domain model in the persistence unit entities (OO) instead of the database schema (SQL)

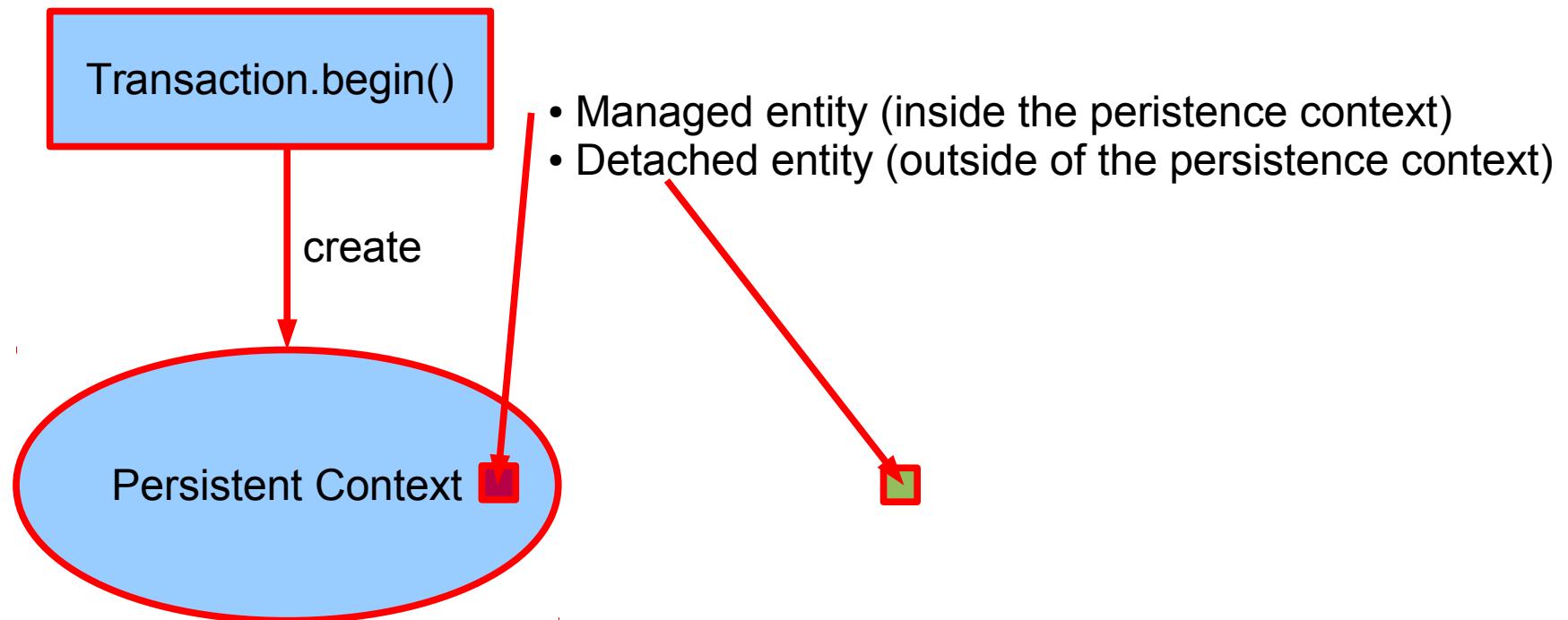
# JPA 2.0 – Persistence Context

em ... instance of EntityManager



# JPA 2.0 – Persistence Context

em ... instance of EntityManager



- `em.persist(entity)` ... persistence context must not contain an entity with the same id
- `em.merge(entity)` ... merging the state of an entity existing inside the persistence context and its other incarnation outside

# JPA 2.0 – Persistence Context

- In runtime, the application accesses the object counterpart (represented by entity instances ) of the database data. These (*managed*) entities comprise a ***persistence context (PC)***.
  - PC is synchronized with the database on demand (refresh, flush) or at transaction commit.
  - PC is accessed by an EntityManager instance and can be shared by several EntityManager instances.

# JPA 2.0 – EntityManager

- **EntityManager (EM)** instance is in fact a generic DAO, while entities can be understood as DPO (managed) or DTO (detached).
- Selected operations on EM (CRUD) :
  - Create : em.persist(Object o)
  - Read : em.find(Object id), em.refresh(Object o)
  - Update : em.merge(Object o)
  - Delete : em.remove(Object o)
  - native/JPQL queries: createNativeQuery, createQuery, etc.
  - Resource-local transactions: getTransaction().  
[begin(),commit(),rollback()]

# ORM - Basics

- Simple View
  - Object classes = entities = SQL tables
  - Object properties (fields/Accessor methods) = entity properties = SQL columns
- The ORM is realized by means of Java annotations/XML.
- Physical Schema annotations
  - @Table, @Column, @JoinColumn, @JoinTable, etc.
- Logical Schema annotations
  - @Entity, @OneToOne, @ManyToOne, etc.
- Each property can be fetched lazily/eagerly.

# Access types – Field access

```
@Entity  
public class Employee {  
    @Id  
    private int id;  
    ...  
    public int getId() {return id;}  
    public void setId(int id) {this.id=id;}  
    ...  
}
```

**The provider will get and set the fields of the entity using reflection (not using getters and setters).**

# Access types – Property access

```
@Entity  
public class Employee {  
    private int id;  
    ...  
    @Id  
    public int getId() {return id;}  
    public void setId(int id) {this.id=id;}  
    ...  
}
```

**Annotation is placed in front of getter.  
(Annotation in front of setter omitted)**

**The provider will get and set the fields of the entity by invoking  
getters and setters.**

# Access types – Mixed access

- Field access with property access combined within the same entity hierarchy (or even within the same entity).
- `@Access` – defines the default access mode (may be overridden for the entity subclass)
- An example on the next slide

# Access types – Mixed access

```
@Entity @Access(AccessType.FIELD)
public class Employee {
    public static final String LOCAL_AREA_CODE = "613";
    @Id private int id;
    @Transient private String phoneNum;
    ...
    public int getId() {return Id;}
    public void setId(int id) {this.id = id;}

    public String getPhoneNumber() {return phoneNum;}
    public void setPhoneNumber(Strung num) {this.phoneNum=num;}

    @Access(AccessType.PROPERTY) @Column(name="PHONE")
    protected String getPhoneNumberForDb() {
        if (phoneNum.length()==10) return phoneNum;
        else return LOCAL_AREA_CODE + phoneNum;
    }
    protected void setPhoneNumberForDb(String num) {
        if (num.startsWith(LOCAL_AREA_CODE))
            phoneNum = num.substring(3);
        else phoneNum = num;
    }
}
```

# ORM – Basic data types

- Primitive Java types: String → varchar/text, Integer → int, Date → TimeStamp/Time/Date, etc.
- Wrapper classes, basic type arrays, Strings, temporal types
- `@Column` – physical schema properties of the particular column (nullable, insertable, updatable, precise data type, defaults, etc.)
- `@Lob` – large objects
- Default EAGER fetching (except Lobs)

# ORM – Enums, dates

- `@Enumerated(value=EnumType.String)`

```
private EnumPersonType type;
```

- Stored either in text column, or in int column

- `@Temporal(TemporalType.Date)`

```
private java.util.Date datum;
```

- Stored in respective column type according to the TemporalType.

# ORM – Identifiers

- Single-attribute: `@Id`,
- Multiple-attribute – an identifier class must exist
  - Id. class: `@IdClass`, entity ids: `@Id`
  - Id. class: `@Embeddable`, entity id: `@EmbeddedId`
- How to write `hashCode`, `equals` for entities ?
- `@Id`  
`@GeneratedValue(strategy=GenerationType.SEQUENCE)`  
`private int id;`

# Generated Identifiers

## Strategies

- AUTO - the provider picks its own strategy
- TABLE – special table keeps the last generated values
- SEQUENCE – using the database native SEQUENCE functionality (PostgreSQL)
- IDENTITY – some DBMSs implement autonumber column

# Generated Identifiers

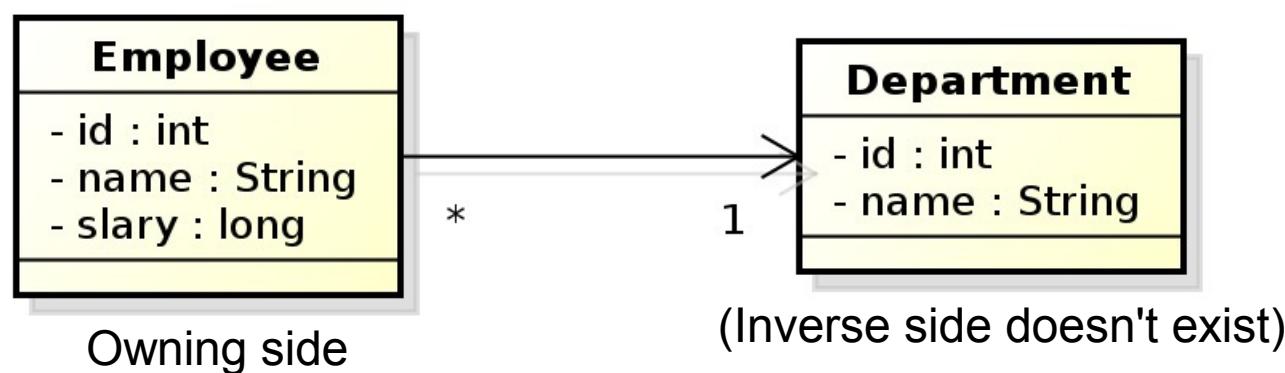
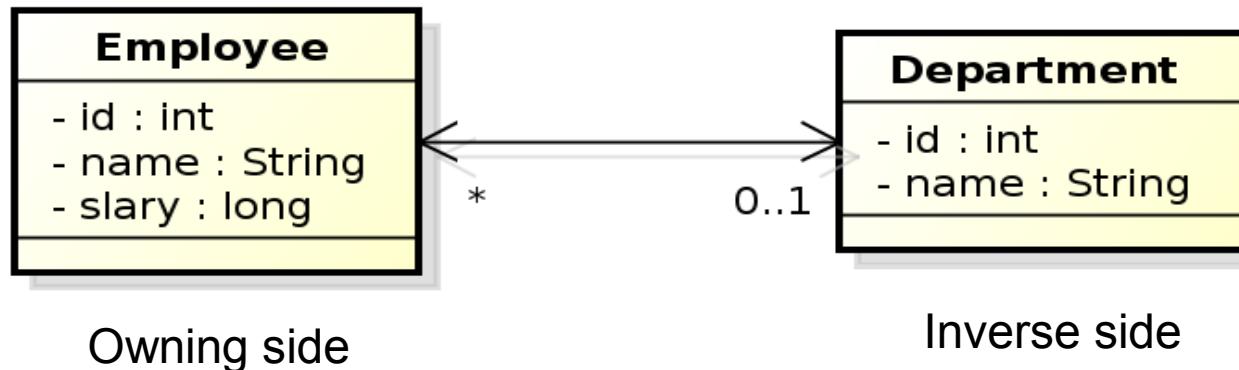
## TABLE strategy

```
@TableGenerator(  
    name="AddressGen",  
    table="ID_GEN",  
    pkColumnName="GEN_NAME",  
    valueColumnName="GEN_VAL",  
    pkColumnValue="ADDR_ID",  
    initialValue=10000,  
    allocationSize=100)
```

```
@Id @GeneratedValue(generator="AddressGen")
```

```
private int id;
```

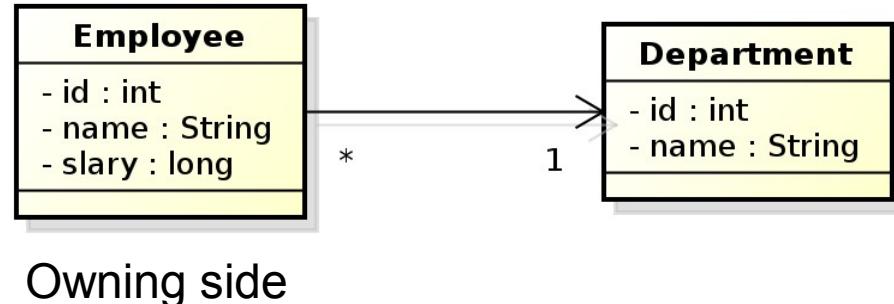
# ORM – Relationships



# ORM – Relationships

		unidirectional	bidirectional
many-to-one	owning	<code>@ManyToOne [@JoinColumn]</code>	<code>@ManyToOne [@JoinColumn]</code>
	inverse	X	<code>@OneToMany(mappedBy)</code>
one-to-many	owning	<code>@OneToMany [@JoinTable]</code>	<code>@OneToMany [@JoinColumn]</code>
	inverse	X	<code>@ManyToOne(mappedBy)</code>
one-to-one	owning (any)	<code>@OneToOne [@JoinColumn]</code>	<code>@OneToOne [@JoinColumn]</code>
	inverse (the other)	X	<code>@OneToOne(mappedBy)</code>
many-to-many	owning (any)	<code>@ManyToMany [@JoinTable]</code>	<code>@ManyToMany [@JoinTable]</code>
	inverse (the other)	X	<code>@ManyToMany(mappedBy)</code>

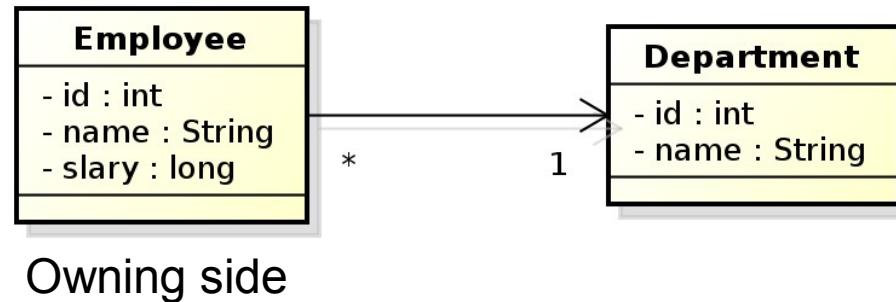
# Unidirectional many-to-one relationship



```
@Entity  
public class Employee {  
    // ...  
    @ManyToOne  
    private Department department;  
    // ...  
}
```

In database, the N:1 relationship is implemented as a foreign key placed in the Employee table. In this case, the foreign key has a default name.

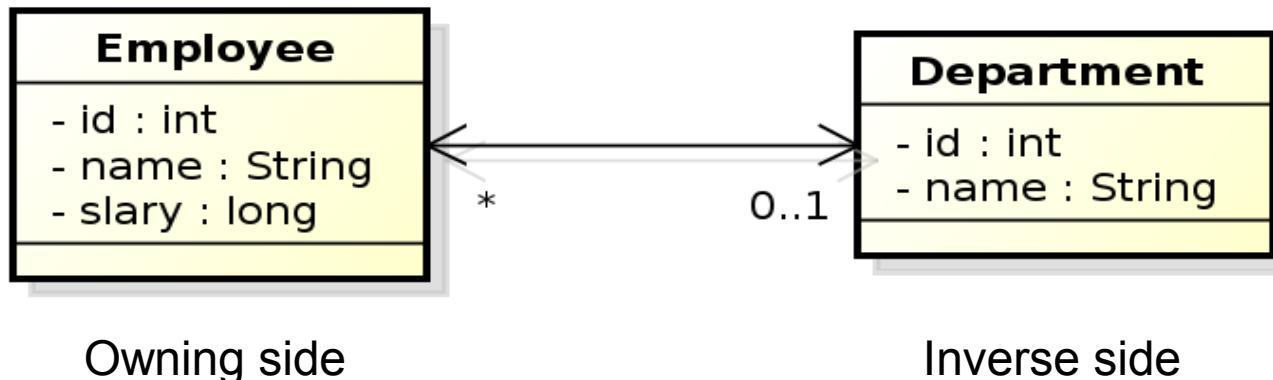
# Unidirectional many-to-one relationship



```
@Entity  
public class Employee {  
  
    @Id private int id;  
    private String name;  
    @ManyToOne  
    @JoinColumn(name="DEPT_ID")  
    private Department department;  
  
}
```

In this case, the foreign key is defined as the `@JoinColumn` annotation.

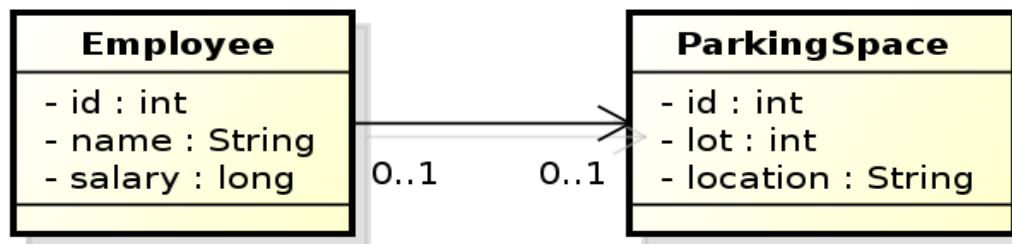
# Bidirectional many-to-one relationship



```
@Entity  
public class Employee {  
  
    @Id private int id;  
    private String name;  
    @ManyToOne  
    @JoinColumn(name="DEPT_ID")  
    private Department department;  
  
}
```

```
@Entity  
public class Department {  
  
    @Id private int id;  
    private String name;  
    @OneToMany(mappedBy="department")  
    private Collection<Employee> employees;  
}
```

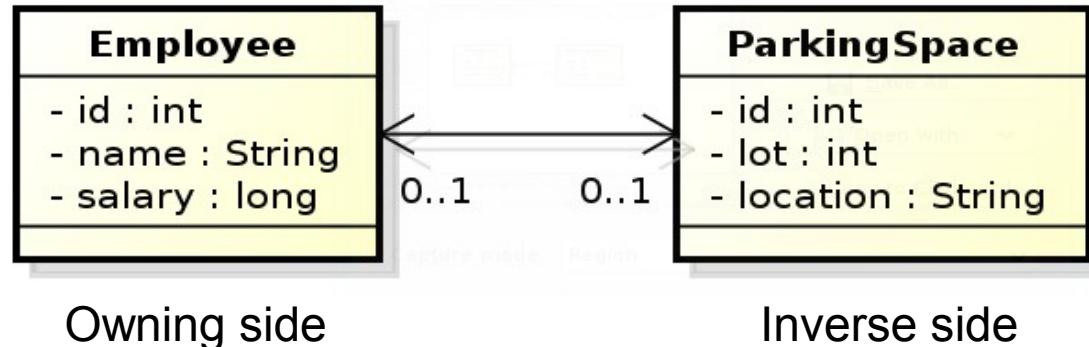
# Unidirectional one-to-one relationship



Owning side

```
@Entity  
public class Employee {  
  
    @Id private int id;  
    private String Name;  
    @OneToOne  
    @JoinColumn(name="PSPACE_ID")  
    private ParkingSpace parkingSpace;  
  
}
```

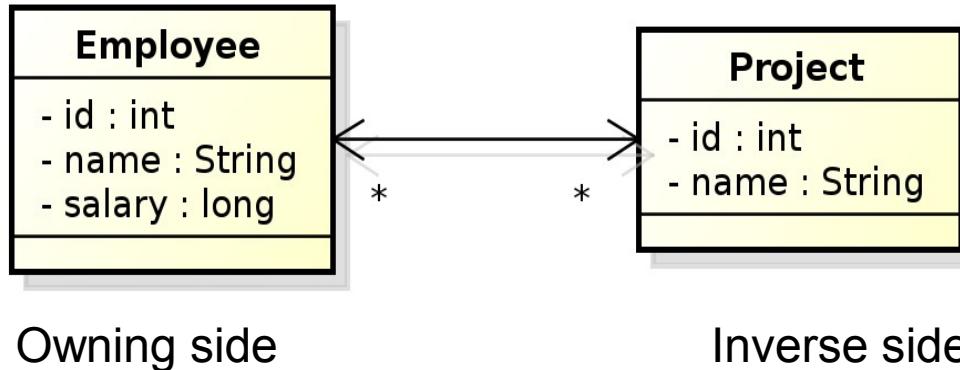
# Bidirectional one-to-one relationship



```
@Entity  
public class Employee {  
  
    @Id private int id;  
    private String Name;  
    @OneToOne  
    @JoinColumn(name="PSPACE_ID")  
    private ParkingSpace parkingSpace;  
  
}
```

```
@Entity  
public class ParkingSpace {  
  
    @Id private int id;  
    private int lot;  
    private String location;  
    @OneToOne(mappedBy="parkingSpace");  
    private Employee employee;  
  
}
```

# Bidirectional many-to-many relationship



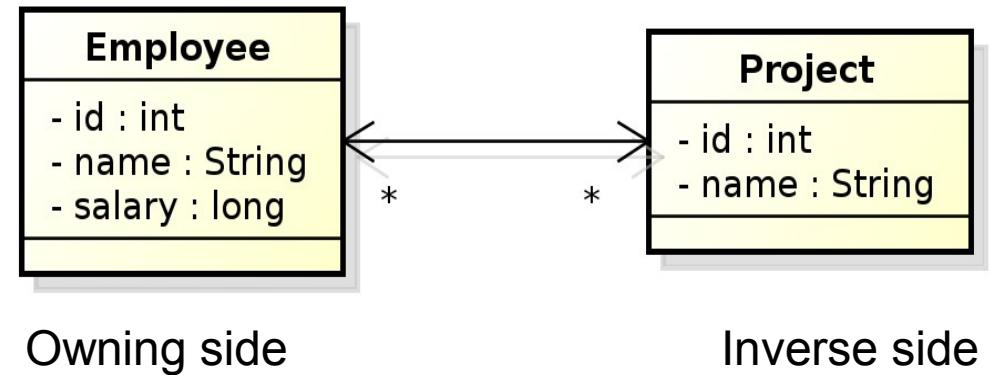
```
@Entity  
public class Employee {  
  
    @Id private int id;  
    private String Name;  
    @ManyToMany  
    private Collection<Project> projects;
```

```
@Entity  
public class Project {  
  
    @Id private int id;  
    private String name;  
    @ManyToMany(mappedBy="projects")  
    private Collection<Employee> employees;
```

In database, N:M relationship must be implemented by means of a table with two foreign keys.  
In this case, both the table and its columns have default names.

# Bidirectional many-to-many relationship

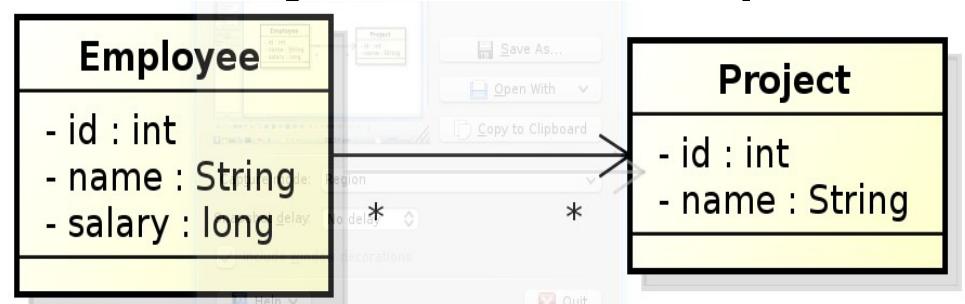
```
@Entity  
public class Employee {  
  
    @Id private int id;  
    private String Name;  
    @ManyToMany  
    @JoinTable(name="EMP_PROJ",  
        joinColumns=@JoinColumn(name="EMP_ID"),  
        inverseJoinColumns=@JoinColumn(name="PROJ_ID"))  
    private Collection<Project> project;  
  
}
```



```
@Entity  
public class Project {  
  
    @Id private int id;  
    private String name;  
    @ManyToMany(mappedBy="projects")  
    private Collection<Employee> employees;  
  
}
```

# Unidirectional many-to-many relationship

```
@Entity  
public class Employee {  
  
    @Id private int id;  
    private String Name;  
    @ManyToMany  
    @JoinTable(name="EMP_PROJ",  
        joinColumns=@JoinColumn(name="EMP_ID"),  
        inverseJoinColumns=@JoinColumn(name="PROJ_ID"))  
    private Collection<Project> project;  
  
}
```

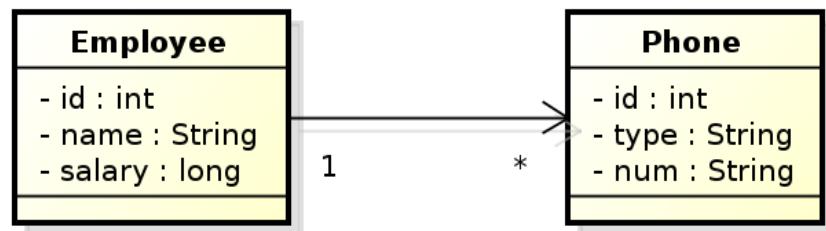


Owning side

```
@Entity  
public class Project {  
  
    @Id private int id;  
    private String name;  
  
}
```

# Unidirectional one-to-many relationship

JPA 2.0 spec: *The default mapping for unidirectional one-to-many relationships uses a join table. Unidirectional one-to-many relationship may be implemented using one-to many foreign key mappings, using the JoinColumn and JoinColumns annotations.*



Owning side

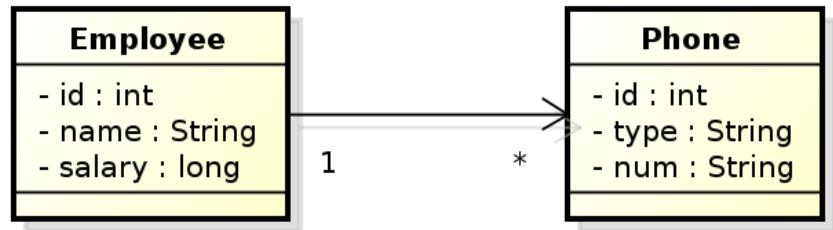
```
@Entity  
public class Employee {  
  
    @Id private int id;  
    private String name;  
    private float salary;  
    @OneToMany  
    @JoinColumn(name="EMP_ID")  
    private Collection<Phone> phones;
```

Not available prior to JPA 2.0

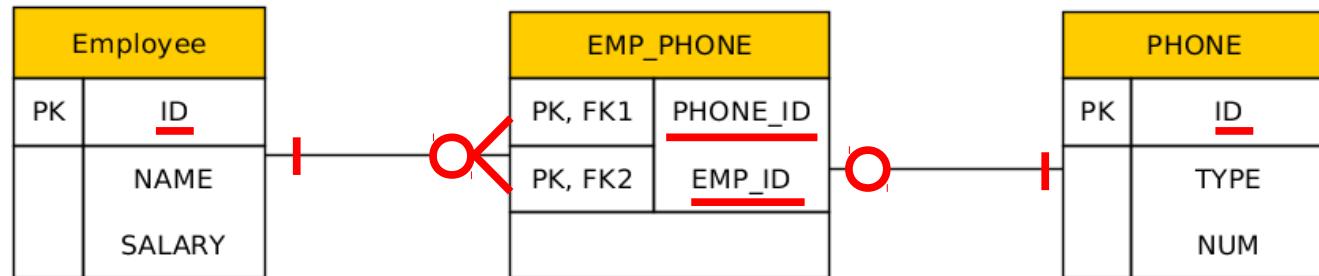
// join column is in the table for Phone

}

# Unidirectional one-to-many relationship



Owning side



Logical database schema

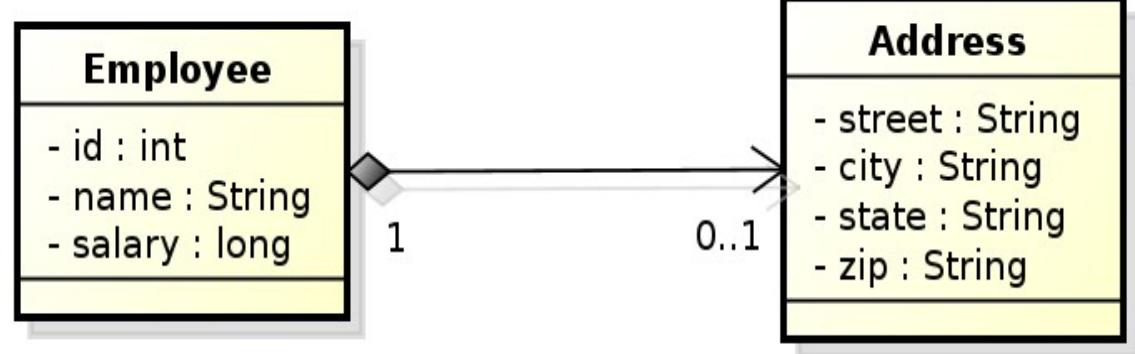
```
@Entity  
public class Employee {  
  
    @Id private int id;  
    private String name;  
    private float salary;  
    @OneToMany  
    @JoinTable(name="EMP_PHONE",  
        joinColumns=@JoinColumn(name="EMP_ID"),  
        inverseJoinColumns=@JoinColumn(name="PHONE_ID"))  
    private Collection<Phone> phones;  
  
}
```

# Lazy Relationships

```
@Entity  
public class Employee {  
  
    @Id private int id;  
    private String name;  
    @OneToOne(fetch=FetchType.LAZY)  
    private ParkingSpace parkingSpace;  
  
}
```

# Embedded Objects

EMPLOYEE	
PK	ID
	NAME
	SALARY
	STREET
	CITY
	STATE
	ZIP_CODE

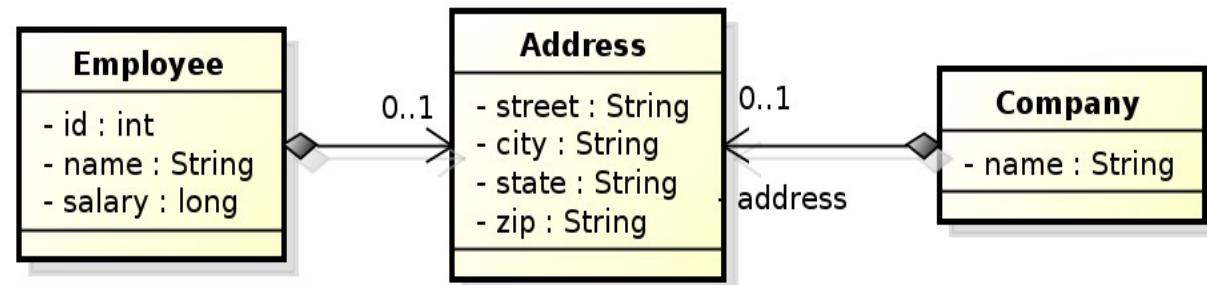


```
@Embeddable
@Access(AccessType.FIELD)
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
}
```

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded private Address
    address;
}
```

# Embedded Objects

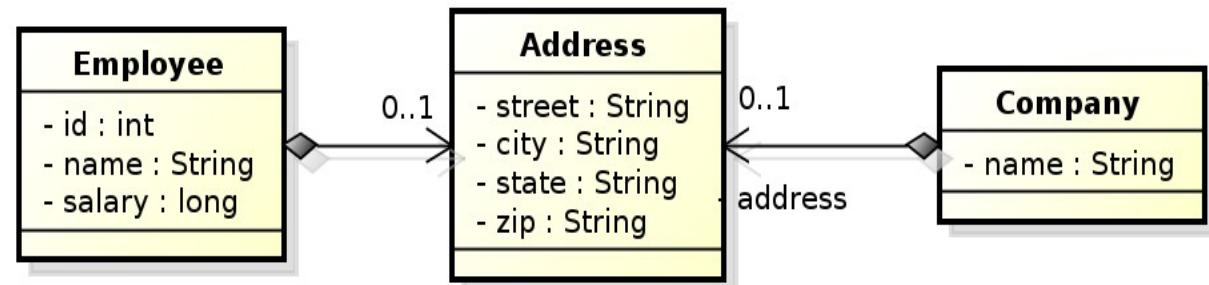
EMPLOYEE		COMPANY	
PK	ID	PK	NAME
	NAME		
	SALARY		
	STREET		STREET
	CITY		CITY
	PROVINCE		STATE
	POSTAL_CODE		ZIP_CODE



```
@Embeddable
@Access(AccessType.FIELD)
public class Address {
    private String street;
    private String city;
    private String state;
    @Column(name="ZIP_CODE")
    private String zip;
}
```

# Embedded Objects

EMPLOYEE		COMPANY	
PK	ID	PK	NAME
	NAME		
	SALARY		
	STREET		STREET
	CITY		CITY
	PROVINCE		STATE
	POSTAL_CODE		ZIP_CODE

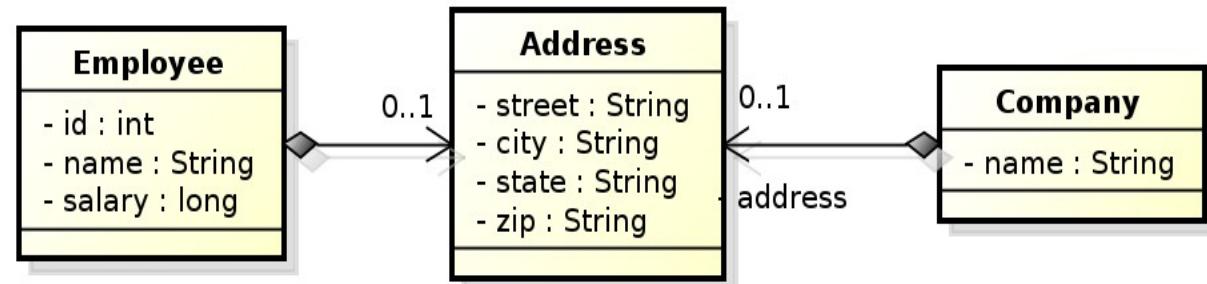


**@Entity**

```
public class Company {  
    @Id private String name;  
    @Embedded  
    private Address address;  
}
```

# Embedded Objects

EMPLOYEE		COMPANY	
PK	ID	PK	NAME
	NAME		
	SALARY		
	STREET		STREET
	CITY		CITY
	PROVINCE		STATE
	POSTAL_CODE		ZIP_CODE



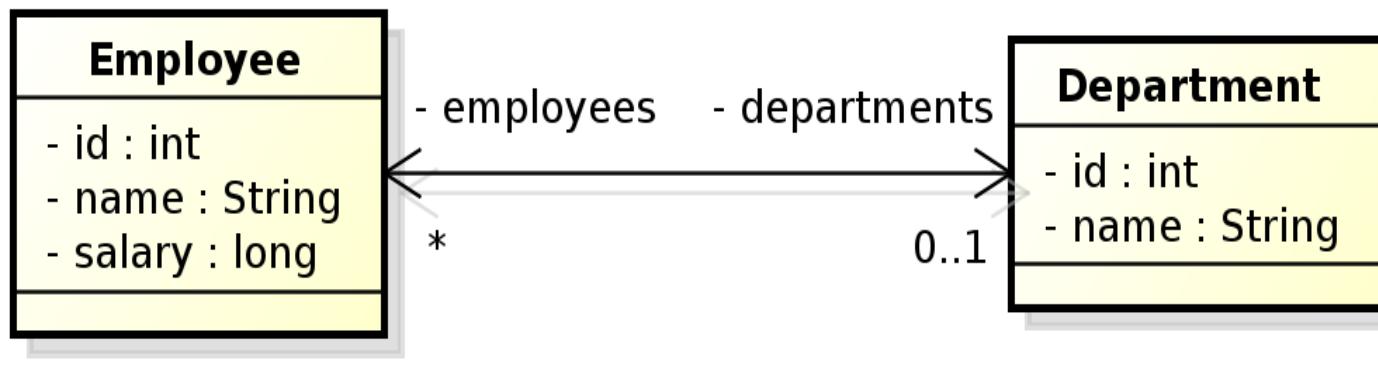
```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name="state", column=@Column(name="PROVINCE")),
        @AttributeOverride(name="zip", column=@Column(name="POSTAL_CODE"))
    })
    private Address address;
}
```

# Cascade Persist

```
@Entity  
public class Employee {  
    // ...  
    @ManyToOne(cascade=cascadeType.PERSIST)  
    Address address;  
    // ...  
}
```

```
Employee emp = new Employee();  
emp.setId(2);  
emp.setName("Rob");  
Address addr = new Address();  
addr.setStreet("164 Brown Deer Road");  
addr.setCity("Milwaukee");  
addr.setState("WI");  
emp.setAddress(addr);  
emp.persist(addr);  
emp.persist(emp);
```

# Persisting bidirectional relationship



...

```
Employee emp = new Employee();
emp.setId(2);
emp.setName("Rob");
emp.setSalary(25000);
Department dept = em.find(Department.class, 101);
dept.employees.add(emp); // @ManyToOne(cascade=cascadeType.PERSIST)
emp.persist(emp);
```

!!! emp.departments still doesn't contain dept !!!

```
emp.refresh(dept);
```

!!! emp.departments does contain dept now !!!

# Cascade

List of operations supporting cascading:

- cascadeType.ALL
- cascadeType.DETACH
- cascadeType.MERGE
- cascadeType.PERSIST
- cascadeType.REFRESH
- cascadeType.REMOVE

# ORM and JPA 2.0

Zdeněk Kouba, Petr Křemen

# Collection Mapping

- Collection-valued relationship (above)
  - @OneToMany
  - @ManyToMany
- Element collections
  - @ElementCollection
  - Collections of Embeddable (new in JPA 2.0)
  - Collections of basic types (new in JPA 2.0)
- Specific types of Collections are supported
  - Set
  - List
  - Map

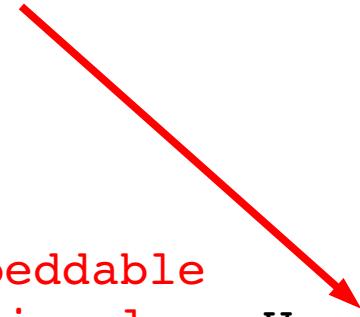
# Collection Mapping

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    // ...
    @ElementCollection(targetClass=VacationEntry.class);
    private Collection vacationBookings;

    @ElementCollection
    private Set<String> nickNames;
    // ...
}

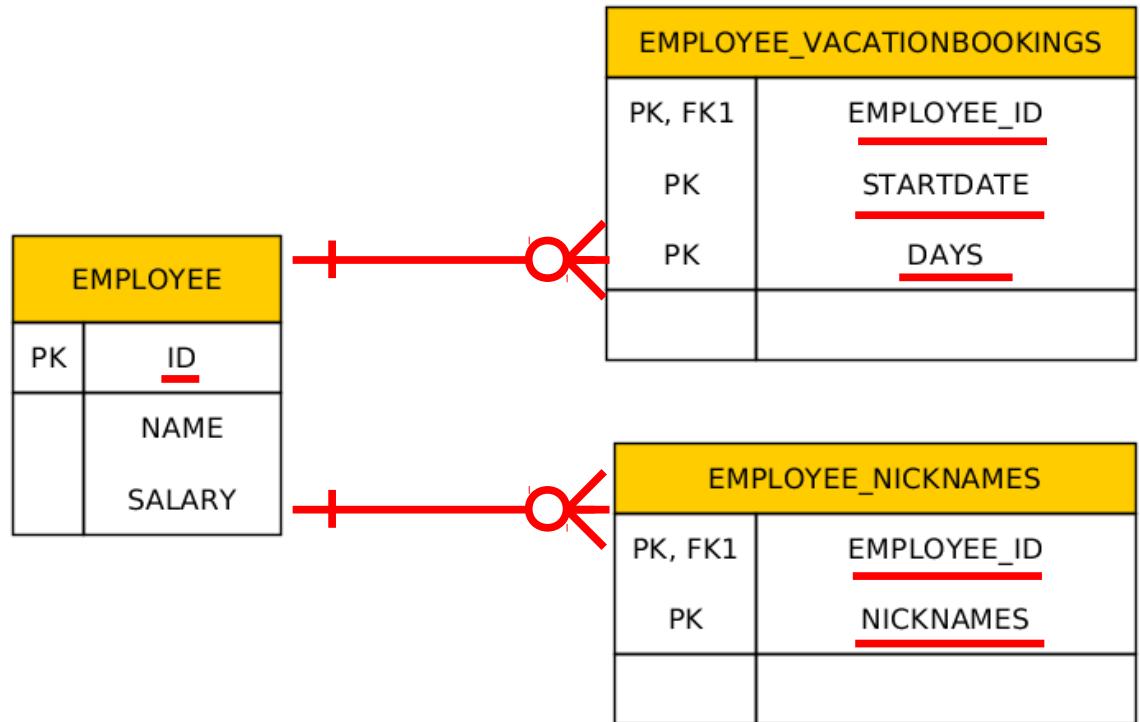
@Embeddable
public class VacationEntry {
    @Temporal(TemporalType.DATE)
    private Calendar startDate;

    @Column(name="DAYS")
    private int daysTaken;
    // ...
}
```



# Collection Mapping

```
@Entity  
public class Employee {  
    @Id private int id;  
    private String name;  
    private long salary;  
    // ...  
    @ElementCollection(targetClass=VacationEntry.class);  
    private Collection vacationBookings;  
  
    @ElementCollection  
    private Set<String> nickNames;  
    // ...  
}
```



# Collection Mapping

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    // ...
    @ElementCollection(targetClass=VacationEntry.class);
    @CollectionTable(
        name="VACATION",
        joinColumn=@JoinColumn(name="EMP_ID"));
    @AttributeOverride(name="daysTaken", column="DAYS_ABS"))
    private Collection vacationBookings;
```

```
@ElementCollection
@Column(name="NICKNAME")
private Set<String> nickName;
// ...
}
```

```
@Embeddable
public class VacationEntry {
    @Temporal(TemporalType.DATE)
    private Calendar startDate;

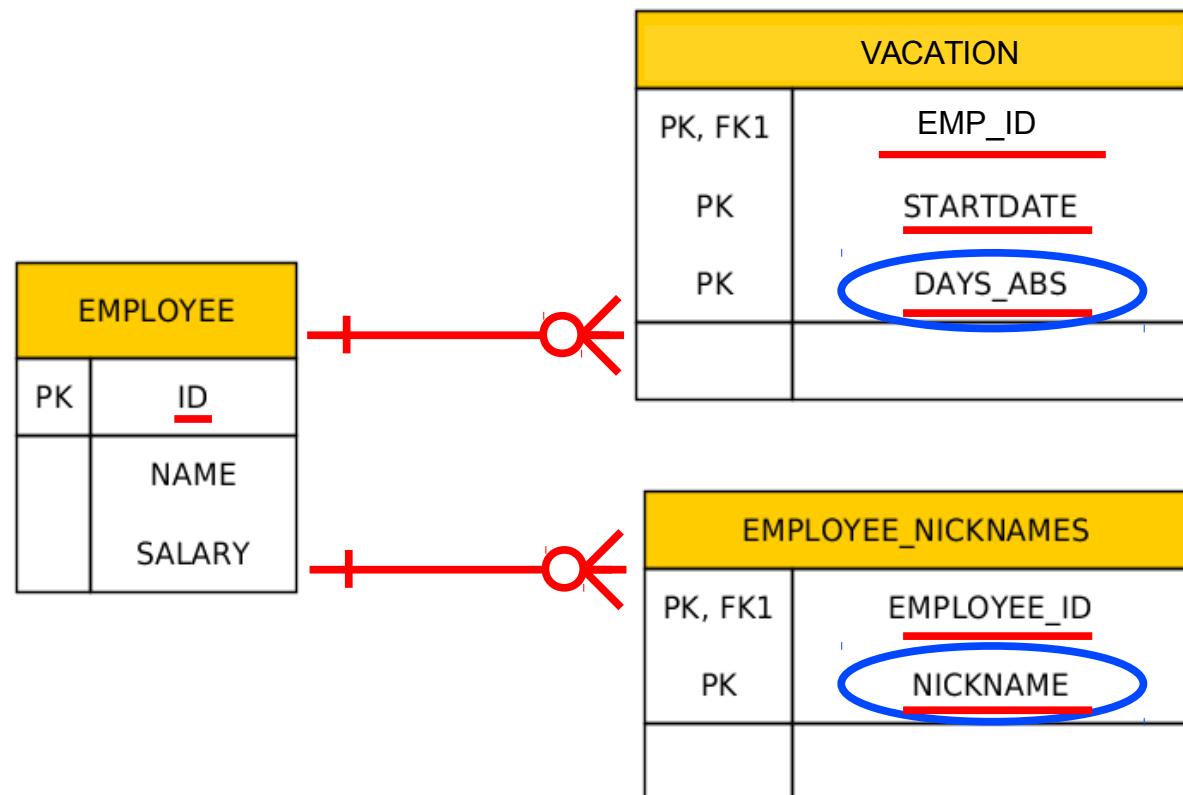
    @Column(name="DAYS")
    private int daysTaken;
    // ...
}
```

# Collection Mapping

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
    // ...
    @ElementCollection(targetClass=VacationEntry.class);
    @CollectionTable(
        name="VACATION",
        joinColumn=@JoinColumn(name="EMP_ID"));
    @AttributeOverride(name="daysTaken", column=@Column(name="DAYS_ABS"))
    private Collection vacationBookings;

    @ElementCollection
    @Column(name="NICKNAME")
    private Set<String> nickName;
    // ...
}

@Embeddable
public class VacationEntry {
    @Temporal(TemporalType.DATE)
    private Calendar startDate;
    @Column(name="DAYS")
    private int daysTaken;
    // ...
}
```



# Collection Mapping

Interfaces:	<ul style="list-style-type: none"><li>• Collection</li><li>• Set</li><li>• List</li><li>• Map</li></ul>	may be used for mapping purposes.
-------------	---	-----------------------------------

An instance of an appropriate implementation class (HashSet,有序列表, etc.) will be used to implement the respective property initially (the entity will be **unmanaged**).

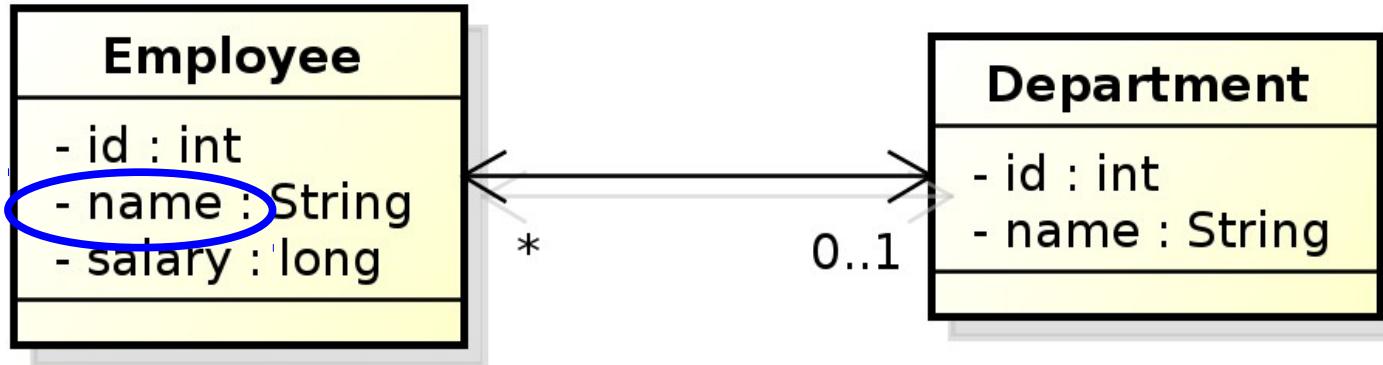
As soon as such an Entity becomes **managed** (by calling em.persist(...)), we can expect to get an instance of the respective interface, not an instance of that particular implementation class when we get it back (em.find(..)) to the persistence context. The reason is that the JPA provider may replace the initial concrete instance with an alternate instance of the respective interface (Collection, Set, List, Map).

# Collection Mapping – ordered List

- Ordering by Entity or Element Attribute
  - ordering according to the state that exists in each entity or element in the List
- Persistently ordered lists
  - the ordering is persisted by means of an additional database column(s)
  - typical example – ordering = the order in which the entities were persisted

# Collection Mapping – ordered List

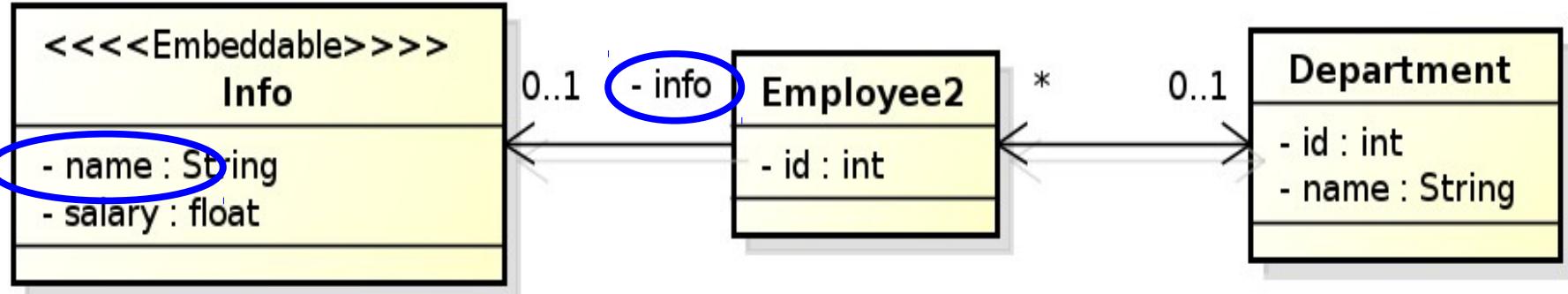
(Ordering by Entity or Element Attribute)



```
@Entity
public class Department {
    // ...
    @OneToOne(mappedBy="department")
    @OrderBy("name ASC")
    private List<Employee> employees;
    // ...
}
```

# Collection Mapping – ordered List

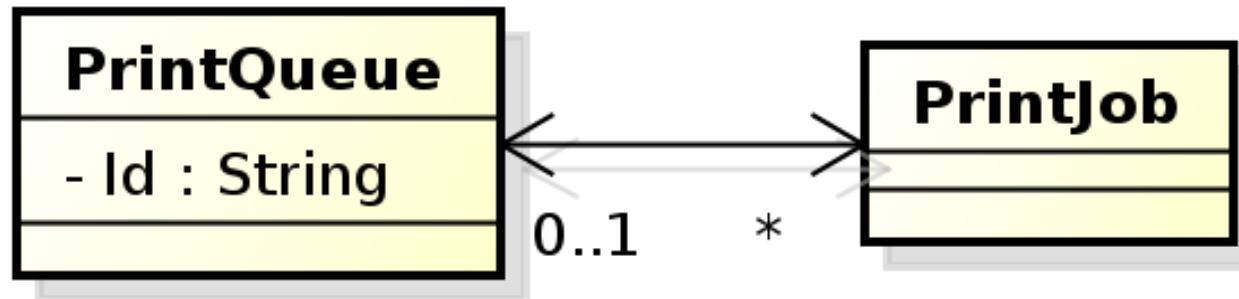
(Ordering by Entity or Element Attribute)



```
@Entity
public class Department {
    // ...
    @OneToMany(mappedBy="department")
    @OrderBy("info.name ASC")
    private List<Employee2> employees;
    // ...
}
```

# Collection Mapping – ordered List

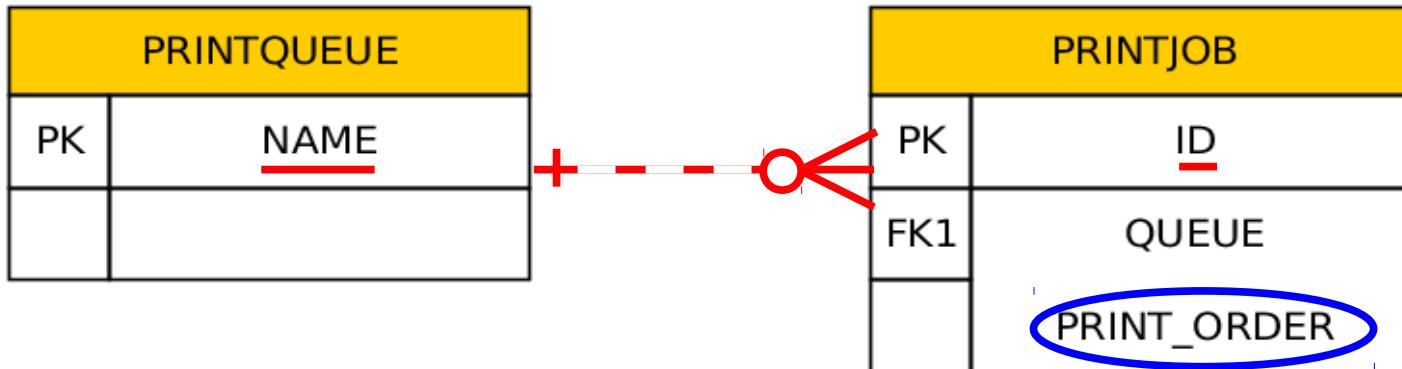
(Persistently ordered lists)



```
@Entity
public class PrintQueue {
    @Id private String name;
    // ...
    @OneToMany(mappedBy="queue")
    @OrderColumn(name="PRINT_ORDER")
    private List<PrintJob> jobs;
    // ...
}
```

# Collection Mapping – ordered List

(Persistently ordered lists)



```
@Entity  
public class PrintQueue {  
    @Id private String name;  
    // ...  
    @OneToMany(mappedBy="queue")  
    @OrderColumn(name="PRINT_ORDER")  
    private List<PrintJob> jobs;  
    // ...  
}
```

This annotation need not be necessarily on the owning side

# Collection Mapping – Maps

Map is an object that maps keys to values.

A map cannot contain duplicate keys;  
each key can map to at most one value.

## Keys:

- Basic types (stored directly in the table being referred to)
  - Target entity table
  - Join table
  - Collection table
- Embeddable types ( - “ - )
- Entities (only foreign key is stored in the table)

## Values:

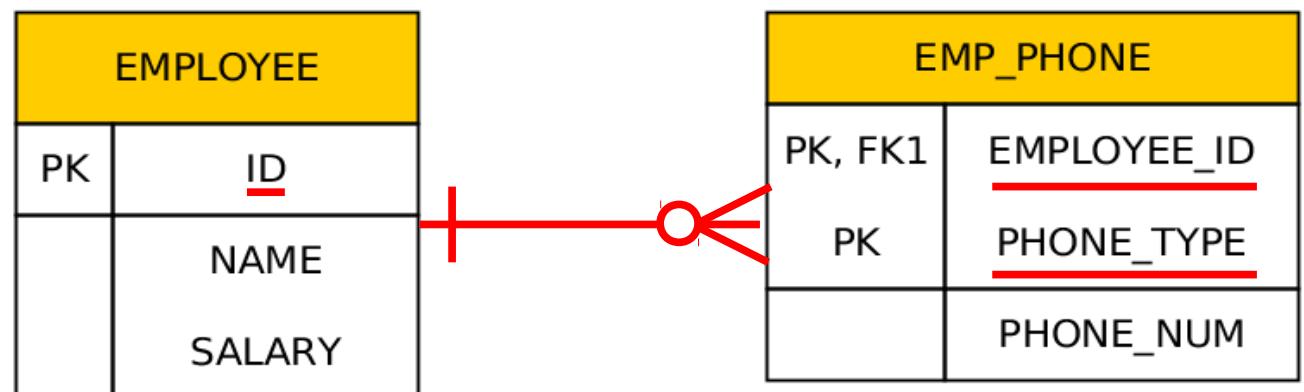
- Values are entities => Map must be mapped as a one-to-many or many-to-many relationship
- Values are basic types or embeddable types => Map is mapped as an element collection

# Collection Mapping – Maps

(keying by basic type – key is String)

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;

    @ElementCollection
    @CollectionTable(name="EMP_PHONE")
    @MapKeyColumn(name="PHONE_TYPE")
    @Column(name="PHONE_NUM")
    private Map<String, String> phoneNumbers;
    // ...
}
```



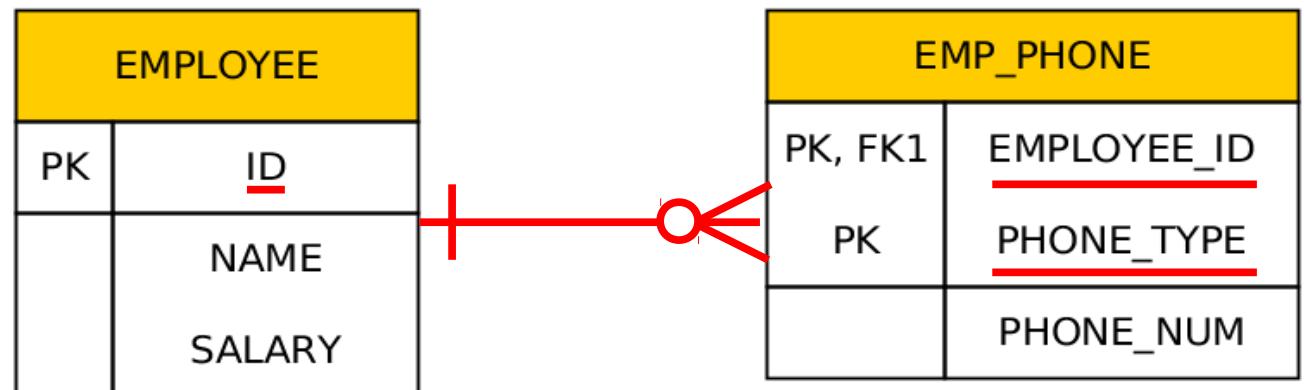
# Collection Mapping – Maps

(keying by basic type – key is an enumeration)

```
@Entity
public class Employee {
    @Id private int id;
    private String name;
    private long salary;
}

@ElementCollection
@CollectionTable(name="EMP_PHONE")
@MapKeyEnumerated(EnumType.String)
@MapKeyColumn(name="PHONE_TYPE")
@Column(name="PHONE_NUM")
private Map<PhoneType, String> phoneNumbers;
// ...
}
```

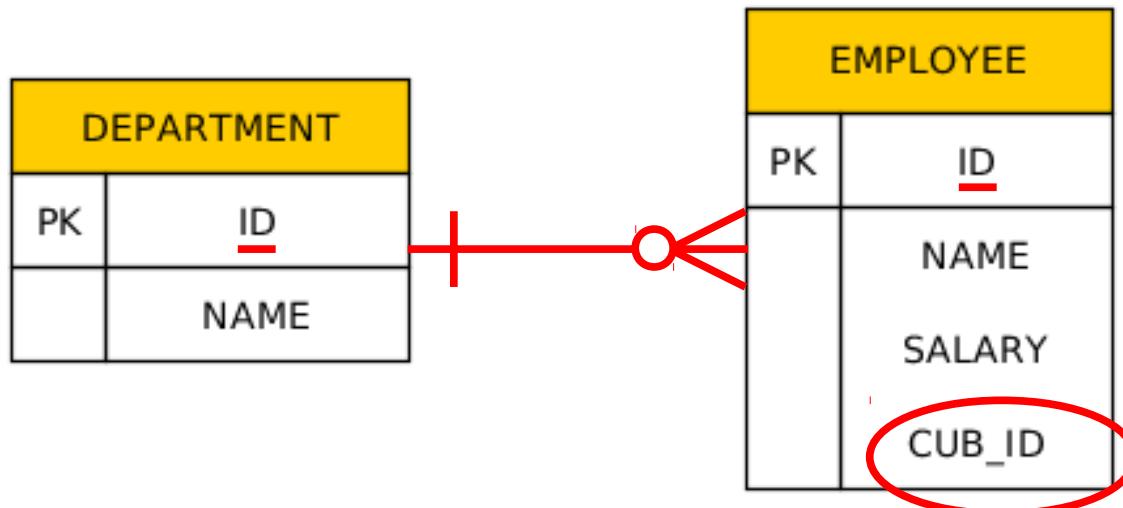
```
Public enum PhoneType {
    Home,
    Mobile,
    Work
}
```



# Collection Mapping – Maps

(keying by basic type – 1:N relationship using a Map with String key)

```
@Entity  
public class Department {  
    @Id private int id;  
    private String name;  
  
    @OneToMany(mappedBy="department")  
    @MapKeyColumn(name="CUB_ID")  
    private Map<String, Employee> employeesByCubicle;  
    // ...  
}
```

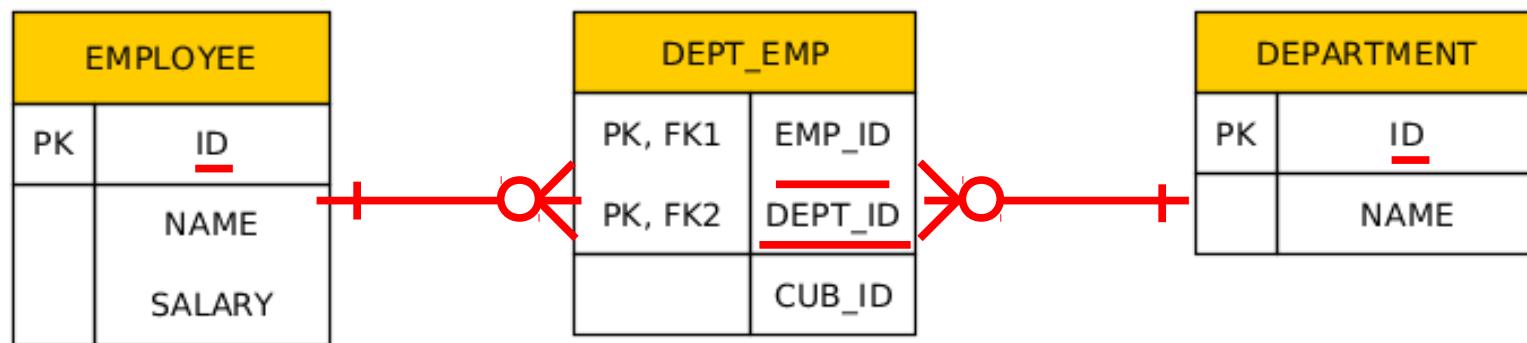


# Collection Mapping – Maps

(keying by basic type – N:M relationship using a Map with String key)

```
@Entity
public class Department {
    @Id private int id;
    private String name;

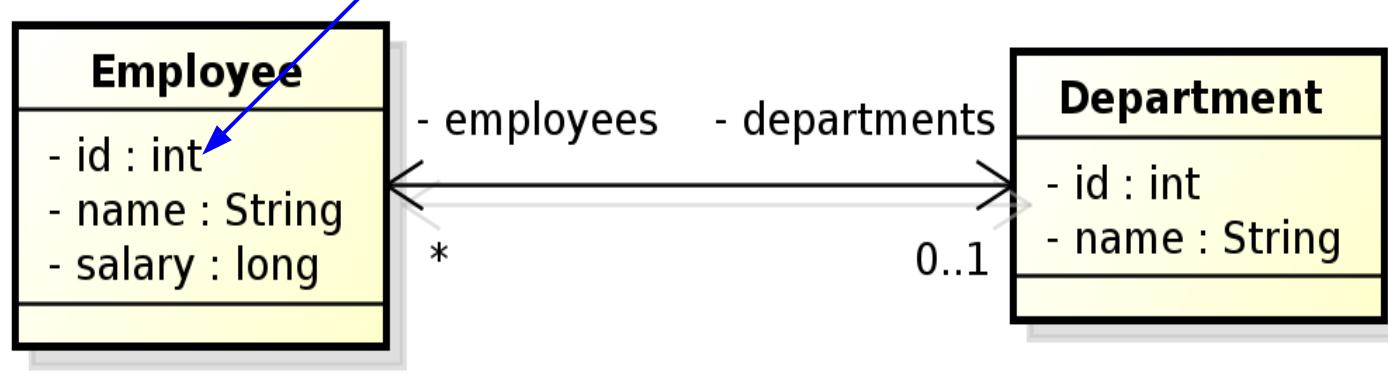
    @ManyToMany
    @JoinTable(name="DEPT_EMP",
        joinColumns=@JoinColumn(name="DEPT_ID"),
        inverseJoinColumns=@JoinColumn(name="EMP_ID"))
    @MapKeyColumn(name="CUB_ID")
    private Map<String, Employee> employeesByCubicle;
    // ...
}
```



# Collection Mapping – Maps

(keying by entity attribute)

```
@Entity  
public class Department {  
    // ...  
    @OneToMany(mappedBy="department")  
    @MapKey(name="id")  
    private Map<Integer, Employee> employees;  
    // ...  
}
```



# Read-only mappings

The constraints are checked on commit!  
Hence, the constrained properties can be  
Modified in memory.

```
@Entity
public class Employee
    @Id
    @Column(insertable=false)
    private int id;

    @Column(insertable=false, updatable=false)
    private String name;

    @Column(insertable=false, updatable=false)
    private long salary;

    @ManyToOne
    @JoinColumn(name="DEPT_ID", insertable=false, updatable=false)
    private Department department;
    // ...
}
```

# ORM and JPA 2.0

Zdeněk Kouba, Petr Křemen

# Compound primary keys

## Id Class

EMPLOYEE	
PK	COUNTRY
PK	EMP_ID
	NAME
	SALARY

No setters. Once created, can not be changed.

```
@Entity  
@IdClass(EmployeeId.class)  
public class Employee {  
    @Id private String country;  
    @Id  
    @Column(name="EMP_ID")  
    private int id;  
    private String name;  
    private long salary;  
    // ...  
}
```

```
public class EmployeeId  
    implements Serializable {  
    private String country;  
    private int id;  
  
    public EmployeeId() {}  
    public EmployeeId(String country,  
                      int id) {  
        this.country = country;  
        this.id = id;  
    }  
}
```

```
public String getCountry() {...};  
public int getId() {...}  
  
public boolean equals(Object o) {...}  
  
public int hashCode() {  
    Return country.hashCode() + id;  
}
```

```
EmployeeId id = new EmployeeId(country, id);  
Employee emp = em.find(Employee.class, id);
```

# Compound primary keys

## Embedded Id Class

EMPLOYEE	
PK	COUNTRY
PK	EMP_ID
	NAME
	SALARY

```
@Embeddable
public class EmployeeId {
    private String country;
    @Column(name="EMP_ID")
    private int id;

    public EmployeeId() {}
    public EmployeeId(String country,
                      int id) {
        this.country = country;
        this.id = id;
    }
    // ...
}
```

```
@Entity
public class Employee {
    @EmbeddedId private EmployeeId id;
    private String name;
    private long salary;
    // ...
    public String getCountry() {return id.getCountry();}
    Public int getId() {return id.getId();}
    // ...
}
```

# Compound primary keys

## Embedded Id Class

EMPLOYEE	
PK	COUNTRY
PK	EMP_ID
	NAME
	SALARY

```
@Embeddable
public class EmployeeId {
    private String country;
    @Column(name="EMP_ID")
    private int id;

    public EmployeeId() {}
    public EmployeeId(String country,
                      int id) {
        this.country = country;
        this.id = id;
    }
    // ...
}
```

Referencing an embedded IdClass in a query:

```
em.createQuery("SELECT e FROM Employee e " +
              "WHERE e.id.country = ?1 AND e.id.id = @2")
    .setParameter(1, country)
    .setParameter(2, id)
    .getSingleResult();
```

# Optionality

```
@Entity
public class Employee
    // ...

    @ManyToOne(optional=false)
    @JoinColumn(name="DEPT_ID", insertable=false, updatable=false)
    private Department department;
    // ...
}
```

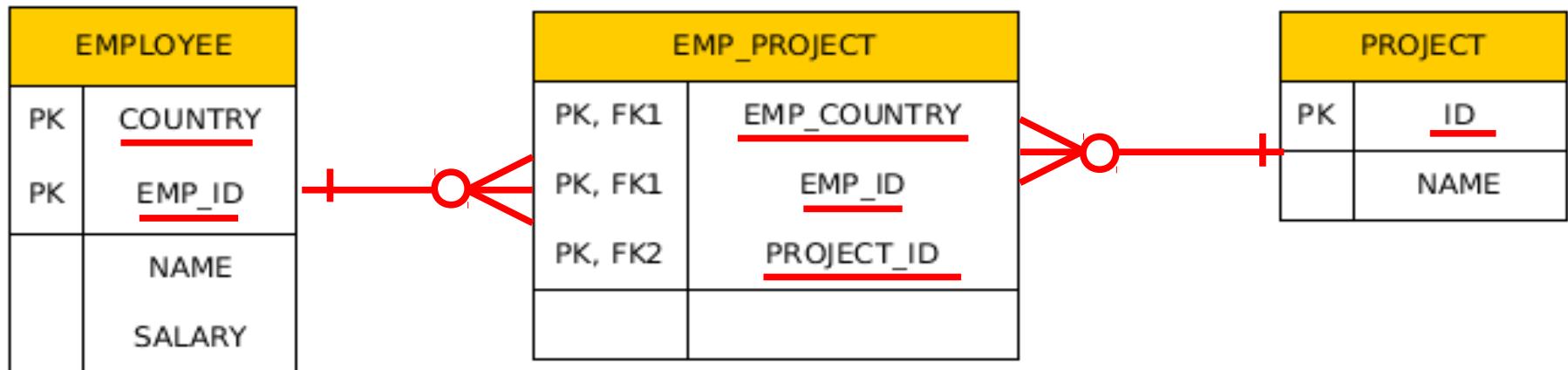
Optionality (parciality) can be used only for **@ManyToOne** and **@OneToOne** relations making the „1“ side of the cardinality „0...1.“

# Compound Join Columns

EMPLOYEE	
PK	<u>COUNTRY</u>
PK	<u>EMP_ID</u>
	NAME
	SALARY
FK1	MGR_COUNTRY
FK1	MGR_ID

```
@Entity  
@IdClass(EmployeeId.class)  
public class Employee {  
    @Id private String country;  
    @Id  
    @Column name="EMP_ID")  
    private int id;  
  
    @ManyToOne  
    @JoinColumns({  
        @JoinColumn(name="MGR_COUNTRY",  
                  referencedColumnName="COUNTRY"),  
        @JoinColumn(name="MGR_ID",  
                  referencedColumnName="EMP_ID")  
    })  
    private Employee manager;  
  
    @OneToMany(mappedBy="manager")  
    private Collection<Employee> directs;  
    // ...  
}
```

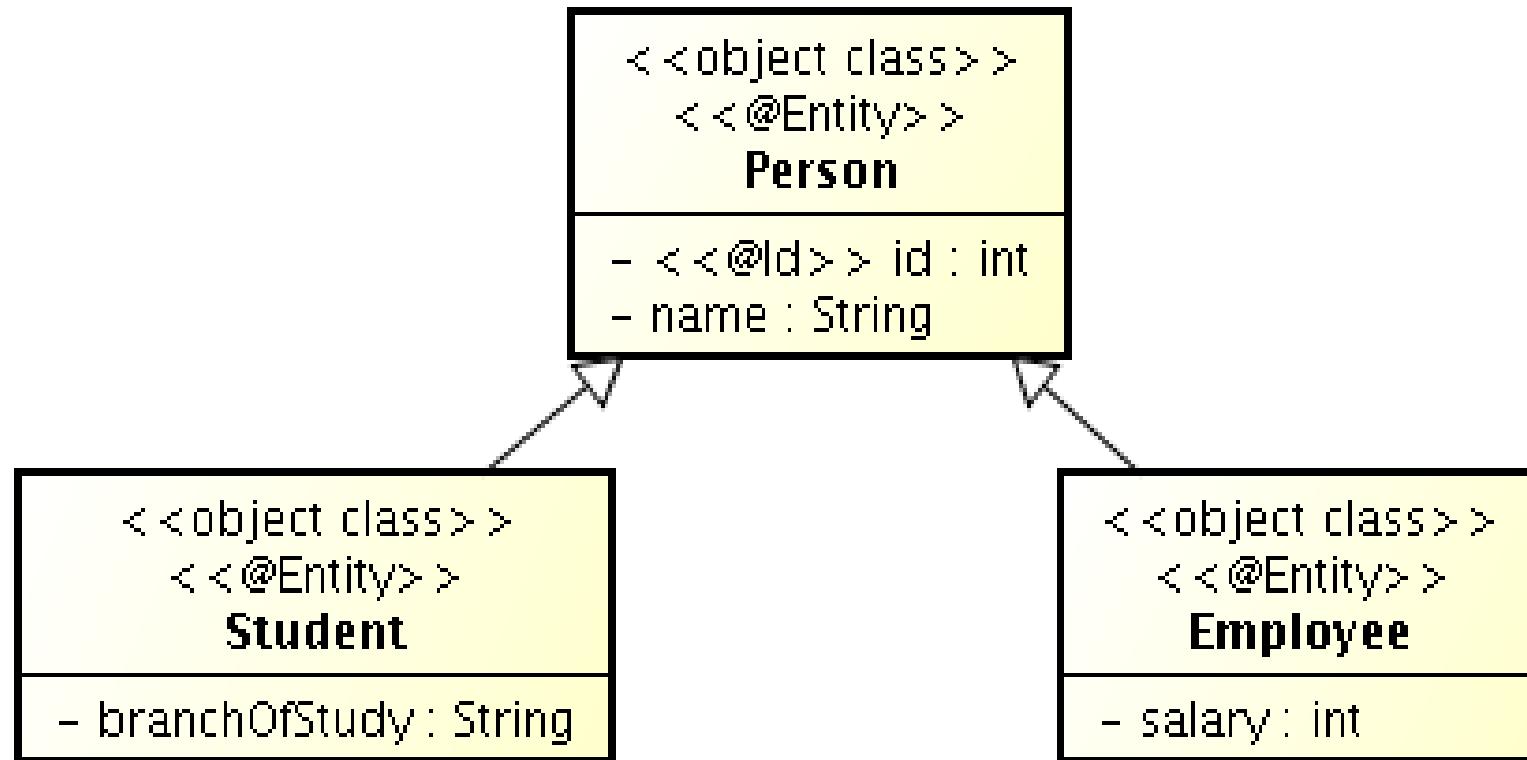
# Compound Join Columns



```
@Entity
@IdClass(EmployeeId.class)
public class Employee
{
    @Id private String country;
    @Id
    @Column(name="EMP_ID")
    private int id;
    @ManyToMany
    @JoinTable(
        name="EMP_PROJECT",
        joinColumns={
            @JoinColumn(name="EMP_COUNTRY", referencedColumnName="COUNTRY"),
            @JoinColumn(name="EMP_ID", referencedColumnName="EMP_ID")},
        inverseJoinColumns=@JoinColumn(name="PROJECT_ID"))
    private Collection<Project> projects;
}
```

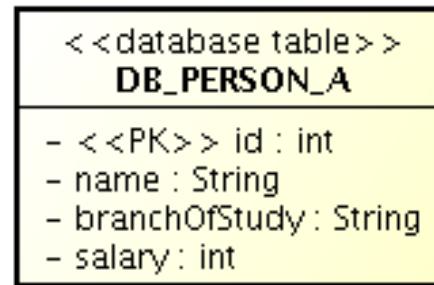
# Inheritance

- How to map inheritance into RDBMS ?

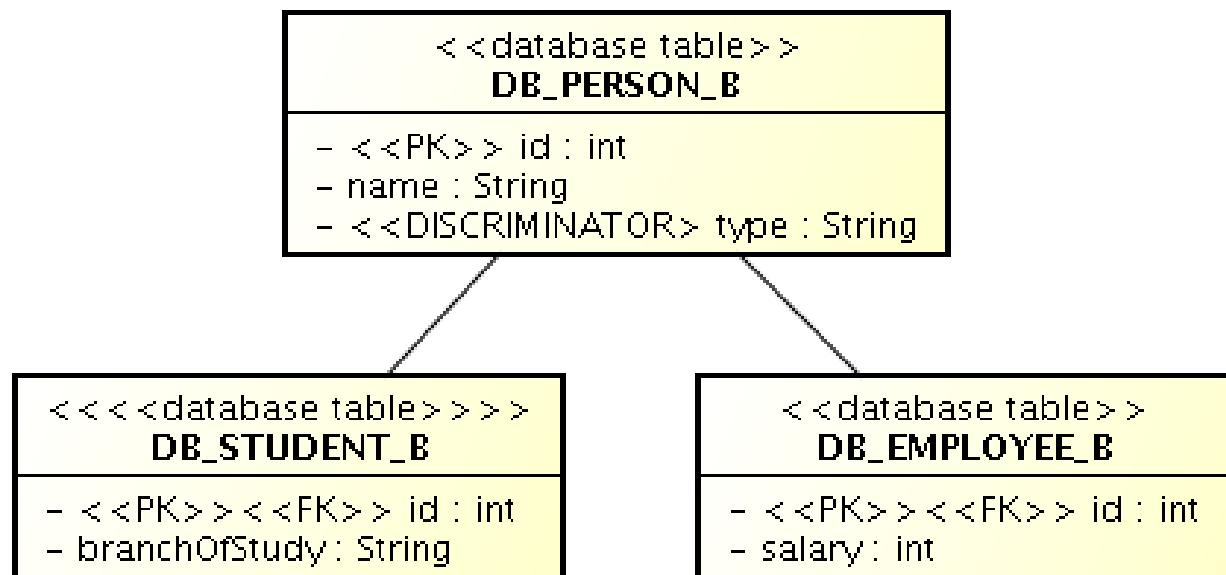


# Strategies for inheritance mapping

- Single table

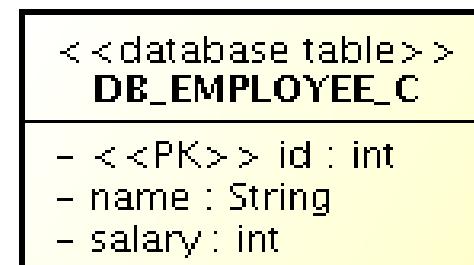
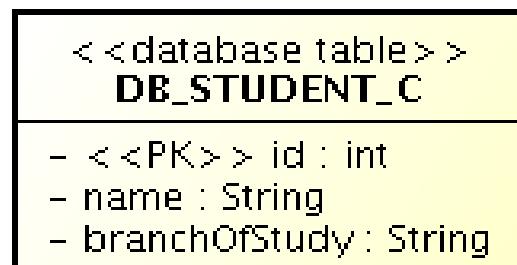
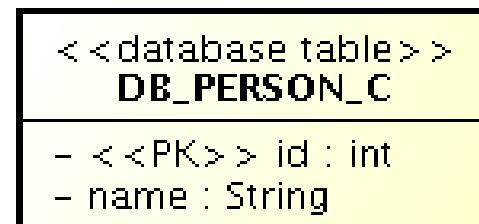


- Joined



# Strategies for inheritance mapping

- Table-per-concrete-class



# Inheritance mapping single-table strategy

```
@Entity
@Table(name="DB_PERSON_A")
@Inheritance //same as @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminationColumn(name="EMP_TYPE")
public abstract class Person { ...}

@Entity
@DiscriminatorValue("Emp")
Public class Employee extends Person {...}

@Entity
@DiscriminatorValue("Stud")
Public class Student extends Person {...}
```

# Inheritance mapping joined strategy

```
@Entity
@Table(name="DB_PERSON_B")
@Inheritance(strategy=InheritanceType.JOINED)
@DiscriminationColumn(name="EMP_TYPE",
                      discriminatorType=discriminatorType.INTEGER)
public abstract class Person { ...}

@Entity
@Table(name="DB_EMPLOYEE_B")
@DiscriminatorValue("1")
public class Employee extends Person {...}

@Entity
@Table(name="DB_STUDENT_B")
@DiscriminatorValue("2")
public class Student extends Person {...}
```

# Inheritance mapping table-per-concrete-class strategy

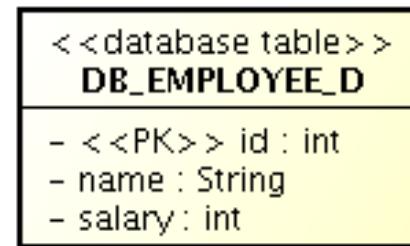
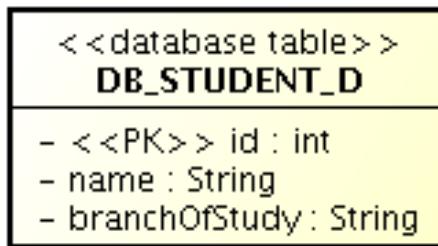
```
@Entity
@Table(name="DB_PERSON_C")
public abstract class Person { ...}

@Entity
@Table(name="DB_EMPLOYEE_C")
@AttributeOverride(name="name", column=@Column(name="FULLNAME"))
@DiscriminatorValue("1")
public class Employee extends Person {...}

@Entity
@Table(name="DB_STUDENT_C")
@DiscriminatorValue("2")
public class Student extends Person {...}
```

# Strategies for inheritance mapping

- If Person is not an @Entity, but a @MappedSuperClass



- If Person is not an @Entity, neither @MappedSuperClass, the deploy fails as the @Id is in the Person (non-entity) class.

# Queries

- JPQL (Java Persistence Query Language)
- Native queries (SQL)

# JPQL

JPQL very similar to SQL (especially in JPA 2.0)

```
SELECT p.number
FROM Employee e JOIN e.phones p
WHERE e.department.name = 'NA42' AND p.type = 'CELL'
```

Conditions are not defined on values of database columns,  
but on entities and their properties.

```
SELECT d, COUNT(e), MAX(e.salary), AVG(e.salary)
FROM Department d JOIN d.employees e
GROUP BY d
HAVING COUNT(e) >= 5
```

# JPQL – query parameters

- positional

```
SELECT e
FROM Employee e
WHERE e.department = ?1 AND e.salary > ?2
```

- named

```
SELECT e
FROM Employee e
WHERE e.department = :dept AND salary > :base
```

# JPQL – defining a query dynamically

```
public class Query {  
    EntityManager em;  
  
    //...  
  
    public long queryEmpSalary(String deptName, String empName)  
    {  
        String query = "SELECT e.salary FROM Employee e " +  
                      "WHERE e.department.name = '" + deptName +  
                      "' AND e.name = '" + empName + "'";  
        return em.createQuery(query, Long.class)  
                  .getSingleResult();  
    }  
}
```

# JPQL – using parameters

```
static final String QUERY = "SELECT e.salary FROM Employee e " +
    "WHERE e.department.name = :deptName " +
    "AND e.name = :empName";

public long queryEmpSalary(String deptName, String empName) {
    return em.createQuery(QUERY, Long.class)
        .setParameter("deptName", deptName)
        .setParameter("empName", empName)
        .getSingleResult();
}
```

# JPQL – named queries

```
@NamedQuery(name="Employee.findByName",
    query="SELECT e FROM Employee e " +
        "WHERE e.name = :name")
```

```
public Employee findEmployeeByName(String name)  {
    return em.createNamedQuery("Employee.findByName",
        Employee.class)
        .setParameter("name", name)
        .getSingleResult();
}
```

# JPQL – named queries

```
@NamedQuery(name="Employee.findByDept",
    query="SELECT e FROM Employee e " +
        "WHERE e.department = ?1")
```

```
public void printEmployeesForDepartment(String dept) {
    List<Employee> result =
        em.createNamedQuery("Employee.findByDept",
            Employee.class)
        .setParameter(1, dept)
        .getResultList();
    int count = 0;
    for (Employee e: result) {
        System.out.println(++count + ":" + e.getName());
    }
}
```

# JPQL – pagination

```
private long pageSize      = 800;
private long currentPage = 0;

public List getCurrentResults()  {
    return em.createNamedQuery("Employee.findByDept",
                               Employee.class)
        .setFirstResult(currentPage * pageSize)
        .setMaxResults(pageSize)
        .getResultList();
}

public void next() {
    currentPage++;
}
```

# JPQL – bulk updates

Modifications of entities not only by em.persist() or em.remove();

```
em.createQuery("UPDATE Employee e SET e.manager = ?1 " +
              "WHERE e.department = ?2")
    .setParameter(1, manager)
    .setParameter(2, dept)
    .executeUpdate();
```

```
em.createQuery("DELETE FROM Project p „ +
              "WHERE p.employees IS EMPTY")
    .executeUpdate();
```

If REMOVE cascade option is set for a relationship, cascading remove occurs.

Native SQL update and delete operations should not be applied to tables mapped by an entity (transaction, cascading).

# Native (SQL) queries

```
@NamedNativeQuery(  
    name="getStructureReportingTo",  
    query = "SELECT emp_id, name, salary, manager_id,"+  
            "dept_id, address_id " +  
            "FROM emp",  
    resultClass = Employee.class  
)
```

Mapping is straightforward

# Native (SQL) queries

```
@NamedNativeQuery(  
    name="getEmployeeAddress",  
    query = "SELECT emp_id, name, salary, manager_id," +  
            "dept_id, address_id, id, street, city," +  
            "state, zip " +  
            "FROM emp JOIN address " +  
            "ON emp.address_id = address.id"  
)
```

Mapping less straightforward

```
@SqlResultSetMapping(  
    name="EmployeeWithAddress",  
    entities={@EntityResult(entityClass=Employee.class),  
              @EntityResult(entityClass=Address.class)})
```

# Native (SQL) queries

```
@SqlResultSetMapping(name="OrderResults",
    entities={
        @EntityResult(entityClass=Order.class,
            fields={
                @FieldResult(name="id", column="order_id"),
                @FieldResult(name="quantity",
                    column="order_quantity"),
                @FieldResult(name="item",
                    column="order_item")}),
        columns={
            @ColumnResult(name="item_name")
    }
)
```

```
Query q = em.createNativeQuery(
    "SELECT o.id AS order_id, " +
        "o.quantity AS order_quantity, " +
        "o.item AS order_item, " +
        "i.name AS item_name, " +
    "FROM order o, item i " +
    "WHERE (order_quantity > 25) AND (order_item = i.id)",
    "OrderResults");
```

```
List<Object[]> results = q.getResultList();

results.stream().forEach((record) -> {
    Order order = (Order)record[0];
    String itemName = (String)record[1];
    ...
});
```

B0B36DBS, BD6B36DBS: **Database Systems**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/>

Lecture 7

# **Relational Algebra**

**Martin Svoboda**

[martin.svoboda@fel.cvut.cz](mailto:martin.svoboda@fel.cvut.cz)

31. 3. 2020

**Czech Technical University in Prague, Faculty of Electrical Engineering**

# Lecture Outline

## Relational algebra

- **Operations:** syntax, semantics and examples
  - Selection, projection, attribute renaming
  - Cartesian product, natural join, theta join, ...
  - Division
  - Outer join
- **Relational completeness**

# Relational Model

## Relational model

- Logical model where all data is represented in terms of **tuples** (rows) that are grouped into **relations** (tables)

## Schema of a relation

- $S(a_1 : T_1, \dots, a_n : T_n)$ 
  - $S$  is a relation name
  - $a_i$  are attribute names,  $T_i$  are optional domains (data types)

## Relation = data

- Set of tuples
- Unordered, no duplicities, without missing values (null), atomic values only (first normal form)

# Relational Model

## Relation structure revisited

- *Formal definition for the purpose of this lecture...*

$$\boxed{\langle R, A_R \rangle}$$

- $R = \text{set of tuples} = \text{actual data}$ 
  - Tuple  $t = \{(a_1, v_1), \dots, (a_n, v_n)\}$ , where:
    - $a_i \in A_R$  is an **attribute** name
    - $v_i \in T_i$  is a **value** this attribute is associated with
    - $(a_i, v_i)$  is an attribute **binding**
  - I.e. each tuple acts as a function
    - $t : A_R \rightarrow \bigcup_{i=1, \dots, n} T_i$
    - $t(a_i) = v_i \in T_i$
- $A_R = \text{set of attributes} = \text{schema of a relation}$ 
  - We continue to omit the domains  $T_i$

# Relation Structure: Example

Sample relation of actors

**Actor(name, surname, year)**

name	surname	year
Ivan	Trojan	1964
Jiří	Macháček	1966
Jitka	Schneiderová	1973

```
<
  { { (name, Ivan), (surname, Trojan), (year, 1964) },
    { (name, Jiří), (surname, Macháček), (year, 1966) },
    { (name, Jitka), (surname, Schneiderová), (year, 1973) } },
  { name, surname, year }
>
```

# Query Languages

Formal **query languages** based on the **relational model**

- **Relational algebra**

- Algebraic **expressions with relations and operations on them**
- E.g. names and surnames of all actors born in 1970 or earlier  
 $\pi_{\text{name}, \text{surname}}(\sigma_{\text{year} \leq 1970}(\text{Actor}))$   
 $\text{Actor}(\text{year} \leq 1970)[\text{name}, \text{surname}]$

- **Relational calculi**

- Expressions **based on the first-order predicate logic**
- Domain relational calculus
  - E.g.  $\{ (n, s) \mid \exists y : \text{Actor}(n, s, y) \wedge y \leq 1970 \}$
- Tuple relational calculus
  - E.g.  $\{ t[\text{name}, \text{surname}] \mid \text{Actor}(t) \wedge t.\text{year} \leq 1970 \}$

# Query Languages: Terminology

## Query expression

- **Expression in a given language describing the intended query**
- Multiple equivalent expressions often exist

## Query

- **Actual data** we are attempting to retrieve
  - I.e. result of the evaluation of a given query expression
- E.g. relation, table, ...

## Query language

- **Set of all syntactically well-formed query expressions** with respect to a given grammar
- E.g. relational algebra, SQL, ...

# Sample Query

First names of all actors born in 1960 or later

$$\pi_{\text{name}}(\sigma_{\text{year} \geq 1960}(\text{Actor}))$$
$$\text{Actor}(\text{year} \geq 1960)[\text{name}]$$

name	surname	year
Ivan	Trojan	1964
Jiří	Macháček	1966
Jitka	Schneiderová	1973
Zdeněk	Svěrák	1936
Jitka	Čvančarová	1978

name
Ivan
Jiří
Jitka

# Relational Algebra

## Inductive construction of RA expressions

- Basic expressions
  - **Relation name:**  $R$
  - **Constant relation**
- General expressions are formed using smaller subexpressions
  - **Projection:**  $\pi_{a_1, \dots, a_n}(E)$
  - **Selection:**  $\sigma_\varphi(E)$
  - **Attribute renaming:**  $\rho_{b_1/a_1, \dots, b_n/a_n}(E)$
  - **Union:**  $E_R \cup E_S$
  - **Difference:**  $E_R \setminus E_S$
  - **Cartesian product:**  $E_R \times E_S$
  - ...

# Projection

**Projection:** preserves only attributes we are interested in

$$\pi_{a_1, \dots, a_n}(E) \quad \text{or} \quad E[a_1, \dots, a_n]$$

- $\langle R, A_R \rangle = \llbracket E \rrbracket$  is a relation  $R$  with attributes  $A_R$
- $a_1, \dots, a_n$  is a set of **attributes to be preserved**, each  $a_i \in A_R$ , all the other attributes are to be removed

$$\llbracket \pi_{a_1, \dots, a_n}(E) \rrbracket = \langle \{t[a_1, \dots, a_n] \mid t \in R\}, \{a_1, \dots, a_n\} \rangle$$

- $t[a_1, \dots, a_n] = \{(a, v) \mid (a, v) \in t, a \in \{a_1, \dots, a_n\}\}$   
is a restriction of a tuple  $t$  to attributes  $a_1, \dots, a_n$
- **Duplicate tuples** in the result are (of course) **suppressed!**

# Projection: Example

First names of all actors

$\pi_{\text{name}}(\text{Actor})$

$\text{Actor}[\text{name}]$

name	surname	year
Ivan	Trojan	1964
Jiří	Macháček	1966
Jitka	Schneiderová	1973
Zdeněk	Svěrák	1936
Jitka	Čvančarová	1978

name
Ivan
Jiří
Zdeněk
Jitka

# Selection

**Selection:** preserves only tuples we are interested in

$$\sigma_{\varphi}(E) \quad \text{or} \quad E(\varphi)$$

- $\langle R, A_R \rangle = \llbracket E \rrbracket$
- $\varphi$  is a condition (**Boolean expression**) to be satisfied
  - **Connectives:**  $\wedge$  (and),  $\vee$  (or),  $\neg$  (negation)
  - Two forms of **atomic formulae**:  $a \Theta b$  or  $a \Theta v$
  - $a, b \in A_R$  are **attributes**,  $v$  is a value **constant**
  - $\Theta \in \{<, \leq, =, \neq, \geq, >\}$  is a **comparison operator**

$$\llbracket \sigma_{\varphi}(E) \rrbracket = \langle \{t \mid t \in R, t \models \varphi\}, A_R \rangle$$

# Selection: Example

Actors born in **1960** or later having a first name other than *Jitka*

$$\sigma_{\text{year} \geq 1960 \wedge \text{name} \neq \text{Jitka}}(\text{Actor})$$

$$\text{Actor}(\text{year} \geq 1960 \wedge \text{name} \neq \text{Jitka})$$

name	surname	year
Ivan	Trojan	1964
Jiří	Macháček	1966
Jitka	Schneiderová	1973
Zdeněk	Svěrák	1936
Jitka	Čvančarová	1978

name	surname	year
Ivan	Trojan	1964
Jiří	Macháček	1966

# Attribute Renaming

**Rename:** changes names of certain attributes

$$\rho_{b_1/a_1, \dots, b_n/a_n}(E) \quad \text{or} \quad E\langle a_1 \rightarrow b_1, \dots, a_n \rightarrow b_n \rangle$$

- $\langle R, A_R \rangle = \llbracket E \rrbracket$
- $a_1, \dots, a_n$  are **current attributes**, each  $a_i \in A_R$ ,  
 $b_1, \dots, b_n$  are **new attributes** (distinct)

$$\llbracket \rho_{b_1/a_1, \dots, b_n/a_n}(E) \rrbracket = \langle \begin{array}{l} \{t[b_1/a_1, \dots, b_n/a_n] \mid t \in R\}, \\ (A_R \setminus \{a_1, \dots, a_n\}) \cup \{b_1, \dots, b_n\} \end{array} \rangle$$

- $t[b_1/a_1, \dots, b_n/a_n] =$   
 $\{(a, v) \mid (a, v) \in t, a \notin \{a_1, \dots, a_n\}\} \cup$   
 $\{(b_i, v) \mid (a_i, v) \in t, i \in \{1, \dots, n\}\}$

# Attribute Renaming: Example

Actors with renamed attributes of first and last names

$\rho_{\text{fname}/\text{name}, \text{lname}/\text{surname}}(\text{Actor})$

$\text{Actor}\langle \text{name} \rightarrow \text{fname}, \text{surname} \rightarrow \text{lname} \rangle$

name	surname	year
fname	lname	year
Ivan	Trojan	1964
Jiří	Macháček	1966
Jitka	Schneiderová	1973
Zdeněk	Svěrák	1936
Jitka	Čvančarová	1978

# Set Operations

**Union, intersection, difference:** standard set operations

$$E_R \cup E_S \quad \llbracket E_R \cup E_S \rrbracket = \langle R \cup S, A \rangle$$

$$E_R \cap E_S \quad \llbracket E_R \cap E_S \rrbracket = \langle R \cap S, A \rangle$$

$$E_R \setminus E_S \quad \llbracket E_R \setminus E_S \rrbracket = \langle R \setminus S, A \rangle$$

- $\langle R, A \rangle = \llbracket E_R \rrbracket$  and  $\langle S, A \rangle = \llbracket E_S \rrbracket$
- Both the relations must be **compatible**
  - I.e. they must have the same attributes

# Set Operations: Difference: Example

Movies that do not have a good rating

AllMovies \ GoodMovies

title	rating
Vratné lahve	76
Samotáři	84
Medvídek	53
Štěstí	72

\

title	rating
Samotáři	84
Kolja	86

=

title	rating
Vratné lahve	76
Medvídek	53
Štěstí	72

# Cartesian Product

**Cartesian product (cross join)**: yields all combinations of tuples from two relations, i.e. unconditionally joins two relations

$$E_R \times E_S$$

- $\langle R, A_R \rangle = \llbracket E_R \rrbracket$  and  $\langle S, A_S \rangle = \llbracket E_S \rrbracket$
- Both the relations must have **disjoint attributes**
  - *Dot convention based on names of relations is often used in practice, but this approach is not always applicable*
    - $R.a$  and  $S.a$  for ambiguous attributes  $a$

$$\llbracket E_R \times E_S \rrbracket = \langle \{t_1 \cup t_2 \mid t_1 \in R, t_2 \in S\}, A_R \cup A_S \rangle$$

- Resulting relations are **flat**
  - *I.e. our Cartesian product differs to the one in the set theory*
- Cardinality of the result:  $|R|.|S|$

# Cartesian Product: Example

All possible combinations of movies and actors

Movie  $\times$  Actor

title	rating	actor	=
Vratné lahve	76	Ivan Trojan	
Samotáři	84	Jiří Macháček	
Medvídek	53		

title	rating	actor
Vratné lahve	76	Ivan Trojan
Vratné lahve	76	Jiří Macháček
Samotáři	84	Ivan Trojan
Samotáři	84	Jiří Macháček
Medvídek	53	Ivan Trojan
Medvídek	53	Jiří Macháček

# Natural Join

**Natural join:** joins two relations based on the pairwise equality of values of all the attributes they mutually share

$$E_R \bowtie E_S \text{ or } E_R * E_S$$

- $\langle R, A_R \rangle = \llbracket E_R \rrbracket$  and  $\langle S, A_S \rangle = \llbracket E_S \rrbracket$

$$\llbracket E_R \bowtie E_S \rrbracket = \langle$$

$$\{t_1 \cup t_2 \mid t_1 \in R, t_2 \in S, \forall a \in A_R \cap A_S : t_1(a) = t_2(a)\} ,$$

$$A_R \cup A_S$$

}

- When there are no shared attributes (i.e.  $A_R \cap A_S = \emptyset$ ),  
 $\bowtie$  corresponds to  $\times$

# Natural Join: Example

Movie characters with full actor names

Cast  $\bowtie$  Actor

Cast  $*$  Actor

title	actor	$\bowtie$	actor	name
Vratné lahve	2		1	Ivan Trojan
Vratné lahve	4		2	Jiří Macháček
Samotáři	1		3	Jitka Schneiderová
Medvídek	1			
Medvídek	2			

=

title	actor	name
Vratné lahve	2	Jiří Macháček
Samotáři	1	Ivan Trojan
Medvídek	1	Ivan Trojan
Medvídek	2	Jiří Macháček

# Natural Join: Inference

$$E_R \bowtie E_S \equiv$$

$$\pi_{r_1, \dots, r_m, a_1, \dots, a_n, s_1, \dots, s_o} \left( \sigma_{x_1 = a_1 \wedge \dots \wedge x_n = a_n} \left( \rho_{x_1 / a_1, \dots, x_n / a_n} (E_R) \times E_S \right) \right)$$

- $\langle R, A_R \rangle = \llbracket E_R \rrbracket$  and  $\langle S, A_S \rangle = \llbracket E_S \rrbracket$ 
  - $a_1, \dots, a_n$  are all the attributes shared by  $R$  and  $S$
  - $x_1, \dots, x_n$  are unused attributes, i.e. each  $x_i \notin A_R, x_i \notin A_S$
  - $r_1, \dots, r_m$  are all the attributes from  $A_R \setminus \{a_1, \dots, a_n\}$
  - $s_1, \dots, s_o$  are all the attributes from  $A_S \setminus \{a_1, \dots, a_n\}$

# Theta Join

**Theta join** ( $\Theta$ -join): joins two relations based on a certain condition

$$E_R \bowtie_{\varphi} E_S \quad \text{or} \quad E_R[\varphi]E_S$$

- $\langle R, A_R \rangle = \llbracket E_R \rrbracket$  and  $\langle S, A_S \rangle = \llbracket E_S \rrbracket$
- **Disjoint attributes**, i.e.  $A_R \cap A_S = \emptyset$
- $\varphi$  is a **condition to be satisfied**
  - Works the same way as conditions in selections

$$\begin{aligned} \llbracket E_R \bowtie_{\varphi} E_S \rrbracket &= \langle \\ &\{t_1 \cup t_2 \mid t_1 \in R, t_2 \in S, (t_1 \cup t_2) \models \varphi\} , \\ &A_R \cup A_S \\ &\rangle \end{aligned}$$

# Theta Join

## Inference

- $E_R \bowtie_{\varphi} E_S \equiv \sigma_{\varphi}(E_R \times E_S)$

# Theta Join: Example

Suitable combinations of movies and actors based on years

Movie  $\bowtie_{\text{filmed} \geq \text{born}}$  Actor

Movie[filmed  $\geq$  born]Actor

$\bowtie_\varphi$

title	filmed
Vratné lahve	2006
Ecce homo Homolka	1970

actor	born
Trojan	1964
Macháček	1966
Schneiderová	1973

=

title	filmed	actor	born
Vratné lahve	2006	Trojan	1964
Vratné lahve	2006	Macháček	1966
Vratné lahve	2006	Schneiderová	1973
Ecce homo Homolka	1970	Trojan	1964
Ecce homo Homolka	1970	Macháček	1966

# Semijoin

**Left / right (natural) semijoin:** yields tuples from the left / right relation that can be naturally joined with the other relation

$$E_R \ltimes E_S \quad \text{or} \quad E_R <* E_S \quad / \quad E_R \rtimes E_S \quad \text{or} \quad E_R *> E_S$$

- $\langle R, A_R \rangle = \llbracket E_R \rrbracket$  and  $\langle S, A_S \rangle = \llbracket E_S \rrbracket$

**Left semijoin:**

$$\llbracket E_R \ltimes E_S \rrbracket = \langle$$

$$\{t_1 \mid t_1 \in R, \exists t_2 \in S : \forall a \in A_R \cap A_S : t_1(a) = t_2(a)\} ,$$

$$A_R$$

}

**Right semijoin:** analogously

# Semijoin

## Inference

- $E_R \ltimes E_S \equiv \pi_{r_1, \dots, r_n}(E_R \bowtie E_S)$ 
  - where  $r_1, \dots, r_n$  are all attributes from the left relation
- Analogously for the right semijoin

# Semijoin: Example

Movie characters who have actor details available

Cast  $\times$  Actor

Cast  $<*$  Actor

title	actor
Vratné lahve	2
Vratné lahve	4
Samotáři	1
Medvídek	1
Medvídek	2

$\times$

actor	name
1	Ivan Trojan
2	Jiří Macháček
3	Jitka Schneiderová

=

title	actor
Vratné lahve	2
Samotáři	1
Medvídek	1
Medvídek	2

# Antijoin

**Left / right antijoin:** yields tuples from the left / right relation that cannot be naturally joined with the other relation

$$E_R \triangleright E_S \quad / \quad E_R \triangleleft E_S$$

- $\langle R, A_R \rangle = \llbracket E_R \rrbracket$  and  $\langle S, A_S \rangle = \llbracket E_S \rrbracket$

**Left antijoin:**

$$\llbracket E_R \triangleright E_S \rrbracket = \langle \{t_1 \mid t_1 \in R, \neg \exists t_2 \in S : \forall a \in A_R \cap A_S : t_1(a) = t_2(a)\} ,$$

$$A_R$$

}

**Right antijoin:** analogously

# Antijoin

## Inference

- $E_R \triangleright E_S \equiv E_R \setminus (E_R \times E_S)$
- Analogously for the right antijoin

# Antijoin: Example

Movie characters that do not have actor details available

Cast  $\triangleright$  Actor

title	actor
Vratné lahve	2
Vratné lahve	4
Samotáři	1
Medvídek	1
Medvídek	2

$\triangleright$

actor	name
1	Ivan Trojan
2	Jiří Macháček
3	Jitka Schneiderová

=

title	actor
Vratné lahve	4

# Theta Semijoin

**Left / right theta semijoin ( $\Theta$ -semijoin):** yields tuples from a given relation that can be joined using a certain condition

$$E_R \ltimes_{\varphi} E_S \quad \text{or} \quad E_R \langle \varphi \rangle E_S \quad / \quad E_R \times_{\varphi} E_S \quad \text{or} \quad E_R [\varphi] E_S$$

- $\langle R, A_R \rangle = \llbracket E_R \rrbracket$  and  $\langle S, A_S \rangle = \llbracket E_S \rrbracket$
- Disjoint attributes, condition  $\varphi$  to be satisfied

**Left  $\Theta$ -semijoin:**

$$\llbracket E_R \ltimes_{\varphi} E_S \rrbracket = \langle \{t_1 \mid t_1 \in R, \exists t_2 \in S : (t_1 \cup t_2) \models \varphi\}, A_R \rangle$$

**Right  $\Theta$ -semijoin:** analogously

# Theta Semijoin

## Inference

- $E_R \ltimes_{\varphi} E_S \equiv \pi_{r_1, \dots, r_n}(E_R \bowtie_{\varphi} E_S)$ 
  - where  $r_1, \dots, r_n$  are all attributes from the left relation
- Analogously for the right  $\Theta$ -semijoin

# Division

**Division:** returns restrictions of tuples from the first relation such that all combinations of these restricted tuples with tuples from the second relation are present in the first relation

$$E_R \div E_S$$

- $\langle R, A_R \rangle = \llbracket E_R \rrbracket$  and  $\langle S, A_S \rangle = \llbracket E_S \rrbracket$
- **Assumption on attributes:**  $A_S \subset A_R$  (proper subset)

$$\llbracket E_R \div E_S \rrbracket = \langle \{t \mid \forall t_2 \in S : (t \cup t_2) \in R\} , A_R \setminus A_S \rangle$$

- Division allows for the simulation of the **universal quantifier**

# Division: Example: 1

Movies in which all the actors played

Cast ÷ Actor

title	actor	÷	actor	=
Vratné lahve	Jiří Macháček		Ivan Trojan	
Vratné lahve	Zdeněk Svěrák		Jiří Macháček	
Samotáři	Ivan Trojan			
Medvídek	Ivan Trojan			
Medvídek	Jiří Macháček			

title
Medvídek

# Division: Example: 2

Movies in which all the actors played

Cast ÷ Actor

title	name	surname
Vratné lahve	Jiří	Macháček
Vratné lahve	Zdeněk	Svěrák
Samotáři	Ivan	Trojan
Medvídek	Ivan	Trojan
Medvídek	Jiří	Macháček

÷

name	surname
Ivan	Trojan
Jiří	Macháček

=

title
Medvídek

# Division: Inference

$$E_R \div E_S \equiv$$

$$\pi_{r_1, \dots, r_m}(E_R) \setminus \\ \pi_{r_1, \dots, r_m} \left( ( \pi_{r_1, \dots, r_m}(E_R) \times E_S ) \setminus E_R \right)$$

- $\langle R, A_R \rangle = \llbracket E_R \rrbracket$  and  $\langle S, A_S \rangle = \llbracket E_S \rrbracket$
- $A_R = \{r_1, \dots, r_m\} \cup \{s_1, \dots, s_n\}$  and  $A_S = \{s_1, \dots, s_n\}$ 
  - I.e.  $s_1, \dots, s_n$  are all the attributes shared by  $R$  and  $S$ ,  
 $r_1, \dots, r_m$  are all the remaining attributes in  $R$

# Join Operations

## Inner joins (and antijoins)

- **Cartesian product:**  $E_R \times E_S$
- **Natural join:**  $E_R \bowtie E_S$
- **Theta join:**  $E_R \bowtie_{\varphi} E_S$
- Left / right **semijoin:**  $E_R \ltimes E_S, E_R \rtimes E_S$
- Left / right **antijoin:**  $E_R \triangleright E_S, E_R \triangleleft E_S$
- Left / right **theta semijoin:**  $E_R \ltimes_{\varphi} E_S, E_R \rtimes_{\varphi} E_S$
- Left / right **theta antijoin:**  $E_R \triangleright_{\varphi} E_S, E_R \triangleleft_{\varphi} E_S$

# Join Operations

## Outer joins

- Left / right / full **outer join**:

$$E_R \bowtie E_S, E_R \ltimes E_S, E_R \bowtie\! E_S$$

- Left / right / full **outer theta join**:

$$E_R \bowtie_\varphi E_S, E_R \ltimes_\varphi E_S, E_R \bowtie_\varphi\! E_S$$

**Extended relational model with null values** is required

# Outer Join

**Left / right / full outer join:** natural join of two relations extended by tuples of the first / second / both relations that cannot be joined

$$E_R \bowtie E_S \quad / \quad E_R \ltimes E_S \quad / \quad E_R \bowtie\! E_S$$

$$E_R *_L E_S \quad / \quad E_R *_R E_S \quad / \quad E_R *_F E_S$$

- $\langle R, A_R \rangle = \llbracket E_R \rrbracket$  and  $\langle S, A_S \rangle = \llbracket E_S \rrbracket$
- $A_R = \{r_1, \dots, r_m\}$ ,  $A_S = \{s_1, \dots, s_n\}$

$$E_R \bowtie E_S \equiv (E_R \bowtie E_S) \cup ((E_R \triangleright E_S) \times \{(\text{null}, \dots, \text{null})\}_{s_1, \dots, s_n})$$

$$E_R \ltimes E_S \equiv (E_R \bowtie E_S) \cup (\{(\text{null}, \dots, \text{null})\}_{r_1, \dots, r_m} \times (E_R \triangleleft E_S))$$

$$E_R \bowtie\! E_S \equiv (E_R \bowtie E_S) \cup (E_R \ltimes E_S)$$

# Outer Join: Example

Movie characters with full actor names if possible

Cast  $\bowtie$  Actor

Cast  $*_L$  Actor

title	actor		actor	name
Vratné lahve	2		1	Ivan Trojan
Vratné lahve	4		2	Jiří Macháček
Samotáři	1		3	Jitka Schneiderová
Medvídek	1			

title	actor	name
Vratné lahve	2	Jiří Macháček
Vratné lahve	4	null
Samotáři	1	Ivan Trojan
Medvídek	1	Ivan Trojan

=

# Observations

## Relational algebra

- **Declarative** query language
  - Query expressions describe **what data to retrieve**, not (necessarily) how such data should be retrieved
- Both *inputs* and *outputs* of queries are **relations**
- Only values actually present in the database can be returned
  - I.e. **derived data cannot be returned**  
(such as various calculations, statistics, aggregations, ...)

# Observations

## Query evaluation

- Construction of a **syntactic tree** (query expression parsing)
  - Based on an inductive structure of a given query expression
  - I.e. based on parentheses (often omitted), operation priorities, associativity conventions, ...
- Nodes
  - **Leaf nodes** correspond to individual input relations
  - **Inner nodes** correspond to individual operations
- Evaluation
  - Node can be evaluated when all its child nodes are evaluated, i.e. when all operands of a given operation are available
  - **Root node** represents the result of the entire query

# Observations

## Equivalent expressions

- Query expressions that **define the same query** (regardless the input relations)
- Various causes
  - **Inference of extended operations** using the basic ones
  - **Commutativity, distributivity or associativity** of (some) operations
  - ...
- Examples
  - Commutativity of selection:  $(E(\varphi_1))(\varphi_2) \equiv (E(\varphi_2))(\varphi_1)$
  - Selection cascade:  $(E(\varphi_1))(\varphi_2) \equiv E(\varphi_1 \wedge \varphi_2)$
  - ...

# Observations

## Basic operations

- Not all the introduced operations are actually necessary in order to form expressions of all the possible queries
- The minimal set of required operations:
  - **Projection:**  $\pi_{a_1, \dots, a_n}(E)$
  - **Selection:**  $\sigma_\varphi(E)$
  - **Attribute renaming:**  $\rho_{b_1/a_1, \dots, b_n/a_n}(E)$
  - **Union:**  $E_R \cup E_S$
  - **Difference:**  $E_R \setminus E_S$
  - **Cartesian product:**  $E_R \times E_S$

## Extended operations

- Intersection, division, all types of joins except the Cartesian product, ...

# Observations

## Relational completeness

- Query language that is able to express all queries of RA is relational complete
  - SQL is relational complete

# Conclusion

## Relational algebra

- Declarative query language for the relational model
- Operations
  - Basic: **projection, selection, attribute renaming, union, difference, Cartesian product**
  - Extended: intersection, natural join, theta join, semijoin, antijoin, division, outer join
  - ...
- **Relational completeness**
- Motivation
  - Evaluation of SQL queries

B0B36DBS, BD6B36DBS: **Database Systems**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/>

Lecture 8

# **Physical Layer**

Authors: **Tomáš Skopal, Irena Holubová**

Lecturer: **Martin Svoboda**, martin.svoboda@fel.cvut.cz

7. 4. 2020

**Czech Technical University in Prague, Faculty of Electrical Engineering**

# Today's lecture outline

- disk management, paging, buffer manager
- database files organization
- indexing
  - B<sup>+</sup>-tree
  - bitmaps
  - hashing

# Three layers of database modeling

abstraction

- conceptual layer
  - models a part of the “**structured**” real world relevant for applications built on top of your database
    - real world part
    - = **real-world entities and relationships between them**
  - different conceptual models (e.g. ER, UML)
- logical layer
  - specifies how conceptual components are represented in logical machine interpretable data structures
  - different logical models (e.g. object, relational, object-relational, XML, graph, etc.)
- **physical layer**
  - specifies how logical database structures are implemented in a specific technical environment
  - data files, index structures (e.g. B+ trees), etc.

implementation



# Introduction

- relations/tables are stored in files on the disk
- we need to organize table records within a file
  - efficient storage, update and access

## Example:

Employees (name char(20), age integer, salary integer)

# Paging

- records are stored in **disk pages** of fixed size (a few kB)
  - reason: hardware
    - assuming a magnetic disk based on rotational plates and reading heads
  - the data organization must be adjusted w.r.t. this mechanism
- the HW firmware can only access entire pages
  - I/O operations – reads, writes
- **real time for I/O operations =**  
**= seek time + rotational delay + data transfer time**
- **sequential access to pages is much faster than random access**
  - the seek time and rotational delay not needed

Example: reading 4 KB could take  $8 + 4 + 0,5 \text{ ms} = 12,5 \text{ ms}$ ;  
i.e., the reading itself takes only  $0,5 \text{ ms} = 4\%$  of the real time!!!

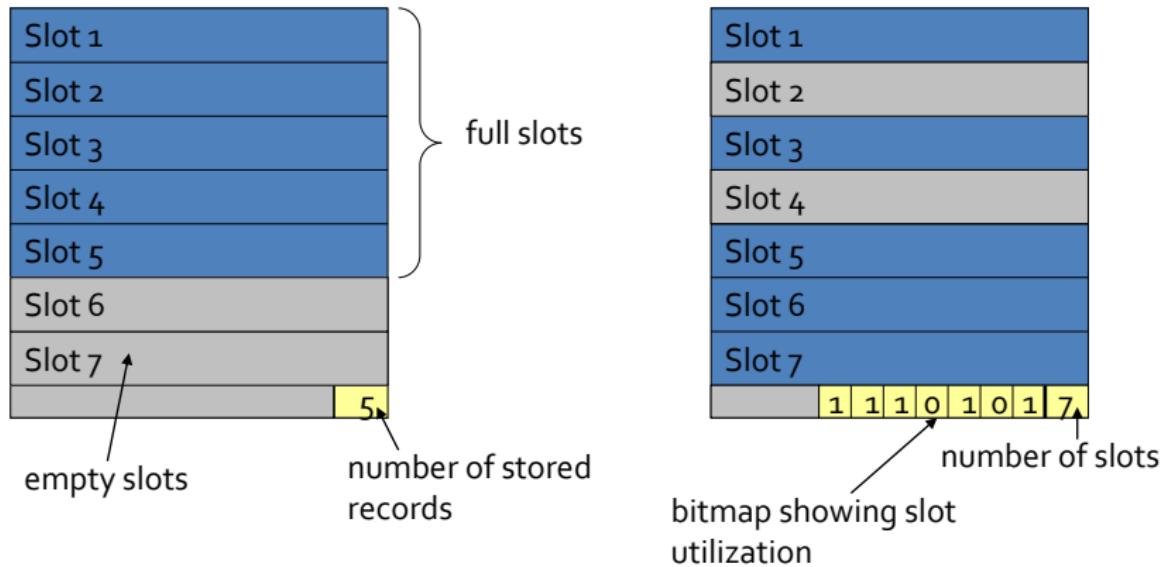
# Paging

- I/O is a unit of time cost
- the page is divided into **slots**, that are used to store **records**
  - a record is identified by **rid** (record id) = **page id + slot id**
- a record can be stored
  - in multiple pages  $\Rightarrow$ 
    - better space utilization
    - need for more I/Os for record manipulation
  - in a single page (assuming it fits)  $\Rightarrow$ 
    - a part of page may not be used
    - less I/Os
  - ideally: records fit the entire page

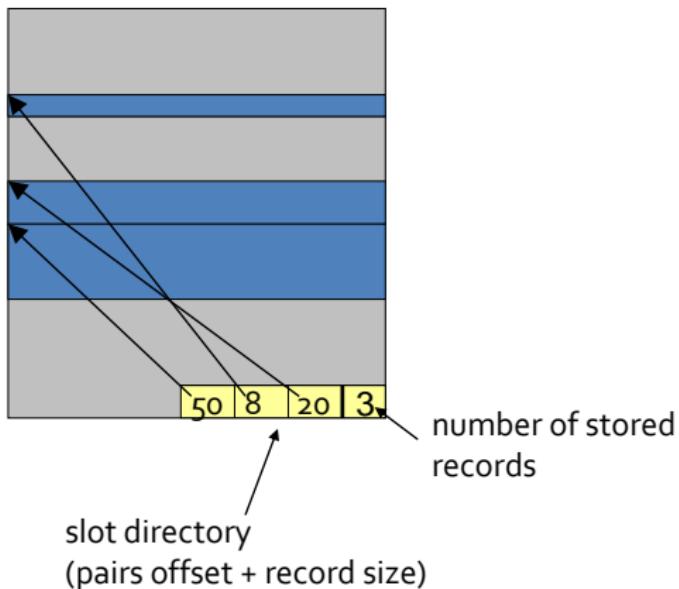
# Paging

- only **fixed-size data types** are used in the record  
⇒ fixed record size
- also **variable-size data types** are used in the record  
⇒ variable size of the records,
  - e.g., types varchar (X) , BLOB, ...
- fixed-size records = fixed-size slots
- variable-size records = need for slot directory in the page header

# Fixed-size page organization, example



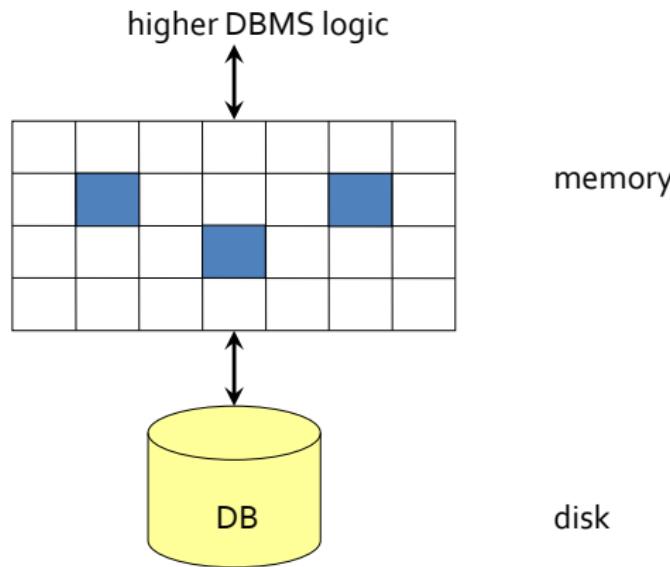
# Variable-size page organization, example



# Buffer management

- **buffer** = a piece of main memory for temporary storage of disk pages
    - disk pages are mapped into **memory frames** 1:1
  - every frame has 2 flags:
    - **pin\_count** = number of references to the page in frame
    - **dirty** = indication of containing a modified record
  - buffer manager
    - implements read and write operations for higher DBMS logic
  - **read:** retrieves the page from buffer + **increasing pin\_count**
    - if it is not there, it is first fetched from the disk
  - **write:** puts the page into the buffer + setting **dirty**
  - if the buffer is full (during read or write), some page must be replaced  
⇒ various policies, e.g., LRU (least-recently-used), ↴ ieli 2 OSY ↴
    - if the replaced page is **dirty**, it must be stored

# Buffer management



# Database storage

- **data files** – contain table data
- **index files** – speed up processing of queries
- **system catalogue** – contains **metadata**
  - table **schemas**
  - index names
  - **integrity constraints**, keys, etc.

# Data files

1. heap → basically linear (linked list)
  2. sorted file → logarithmic (trees, bisection).
  3. hashed file → amortized constant (hash table).
- } lookup

Observing average I/O cost of simple operations:

- 1) sequential access to records
- 2) searching records based on equality (w.r.t search key)
- 3) searching records based on range (w.r.t search key)
- 4) record insertion
- 5) record deletion

Cost model:

$N$  = number of pages,  $R$  = records per page

# Simple operations, SQL examples

- sequential reading of pages

```
SELECT * FROM Employees
```

- searching on equality

```
SELECT * FROM Employees WHERE age = 40
```

- searching on range

```
SELECT * FROM Employees
```

```
WHERE salary > 10000 AND salary < 20000
```

- record insertion

```
INSERT INTO Employees VALUES (...)
```

- record deletion based on rid

```
DELETE FROM Employees WHERE rid = 1234
```

- record deletion

```
DELETE FROM Employees WHERE salary < 5000
```

# Heap file

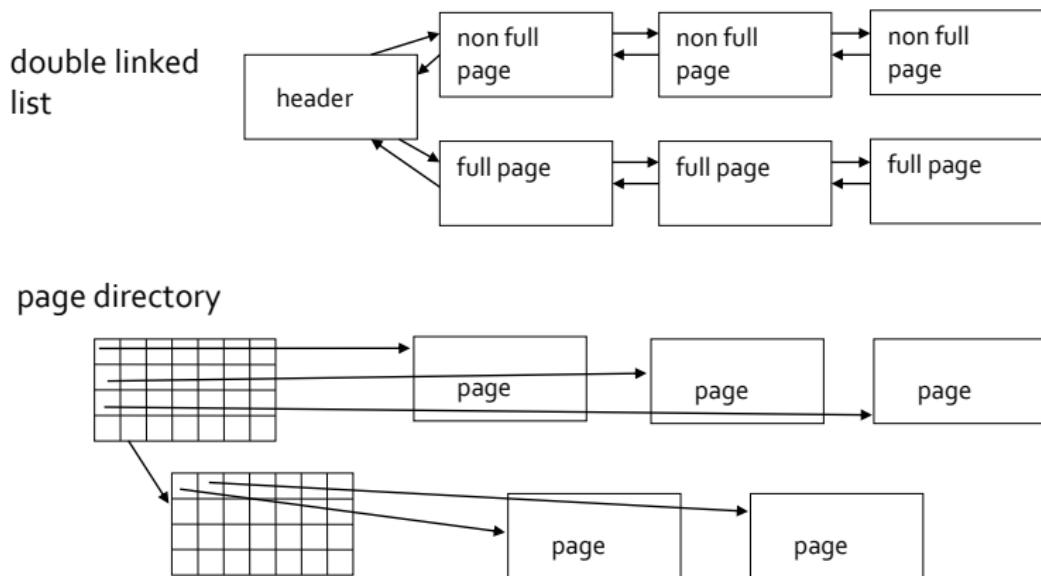
- records stored in pages are not ordered (e.g., according to key)
  - they are stored in the order of insertion
- page search can only be achieved by sequential scan (**GetNext** operation)
- quick record insertion (at the end of file)
- deletion problems: „holes“ (pieces of not utilized space)

# Maintenance of empty heap pages

- double linked list
  - header + lists of full and non full pages
- page directory
  - linked list of directory pages
  - every item in the directory refers to a data page
    - flag = item utilization

→ 20se Os  
↳ tojsmc  
Primo implem.  
(malloc).

# Maintenance of empty heap pages



# Heap, cost of simple operations

- sequential access = N
- search on equality = N
- search on range = N
- record insertion = 1
- record deletion

- 2,  
assuming **rid** based search costs 1 I/O
- N or  $2^*N$ ,  
if deleted based on equality or range

→ good for insertion

# Sorted file

- records stored in pages based on an ordering according to a search key
  - single or multiple attributes
- file pages maintained contiguous, i.e., no „holes“
- fast: search on equality and/or range
- slow: insertion and deletion
  - „moving“ the rest of pages
- in practice:
  - sorted file at the beginning
  - each page has an overhead space where to insert
  - if the overhead space is full, update pages are used (linked list)
  - a reorganization needed from time to time
    - i.e., sorting

*— so basically combined with the previous one.*

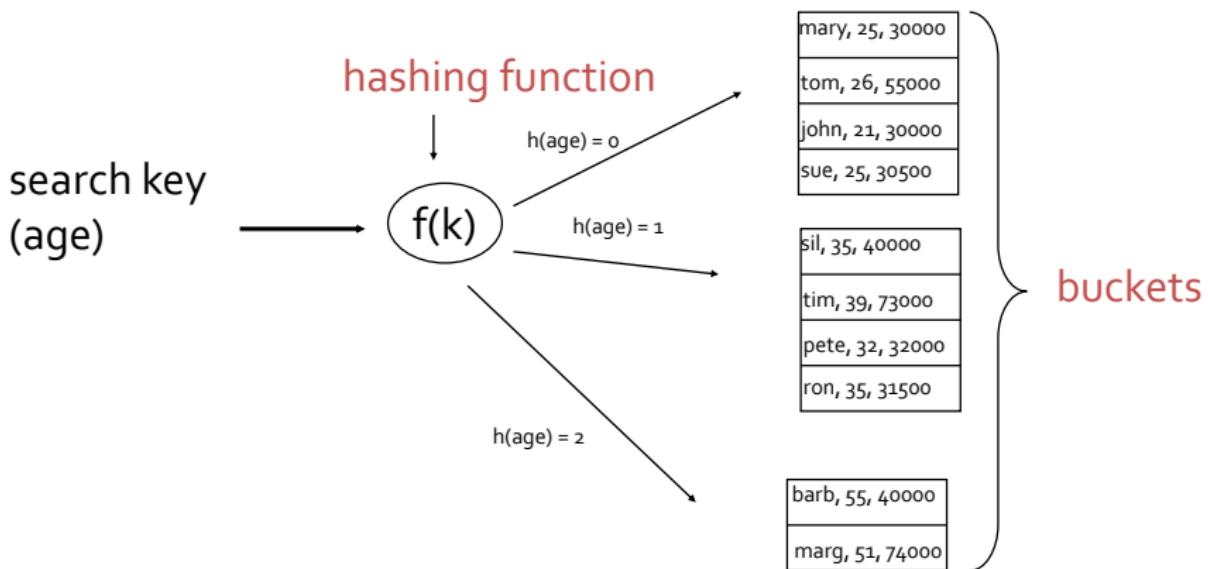
# Sorted file, cost of simple operations

- sequential access =  $N$
  - search on equality =  $\log_2 N$
  - search on range =  $\log_2 N + M$ 
    - where  $M$  is the number of relevant pages
  - record insertion =  $N$
  - record deletion =  $\log_2 N + N$  (based on key)
- like in the  
(binary search).*

# Hashed file

- organized in K buckets
  - a bucket is extensible to multiple disk pages
- a record is inserted into/read from a bucket determined by hashing function  $f$  applied on search key
  - **bucket id =  $f(key)$**
- if the bucket is full, new pages are allocated and linked to the bucket (linked list)
- fast search / deletion on equality
- higher space overhead, problems with chained pages (solved by dynamic hashed techniques)

# Hashed file



# Hashed file, cost of simple operations

- sequential access = N
- search on equality =  $N/K$  (best case)
  - $K$  = number of buckets *↪ OK, so number of buckets is important.*
- search on range = N
- record insertion =  $N/K$  (best case)
- deletion on equality =  $N/K + 1$  (best case)

# Indexing

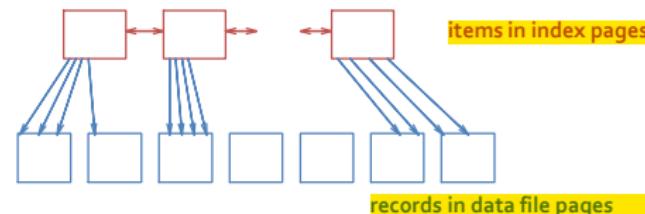
- **index** is a helper structure that provides fast search based on search key(s)
- organized into disk pages (like data files)
  - usually different file than data files
- usually contains only search keys and links to the respective records
  - i.e., rid
- need much less space than data files
  - e.g., 100x less

# Indexing, principles

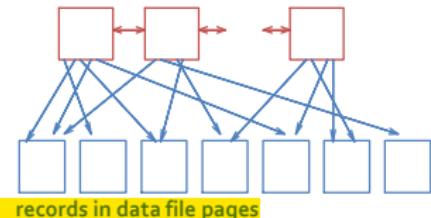
- index item can contain
  - the whole record (then index and data file are the same)
  - pair `<key, rid>`
  - pair `<key, rid-list>`, where `rid-list` is a list of links to records with the same search key value
- 1. **clustered:** ordering of index items is (almost) the same as ordering in the data file
  - tree-based index, index containing the entire records, hashed index, ...
  - primary key = search key used in clustered index
- 2. **unclustered:** the order of search keys is not preserved

# Indexing, principles

CLUSTERED INDEX



UNCLUSTERED INDEX



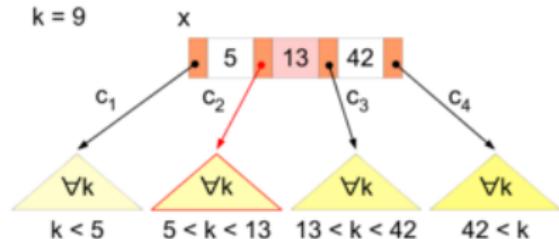
## Clustered index:

Pros: huge speedup when searching on range – result record pages are read sequentially from data file

Cons: large overhead for keeping the data file sorted

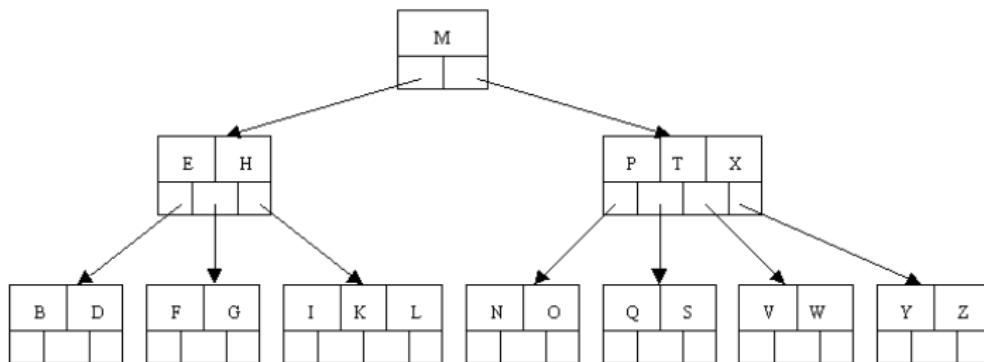
# B<sup>+</sup>-tree

↳ here comes  
ALG



- extends B-tree
  - balanced tree-based index
- provides logarithmic complexity for insertion, search on equality (no duplicates), deletion on equality (no duplicates)
- guarantees 50% node (page) utilization
- B<sup>+</sup>-tree extends B-tree by
  - all keys are in the leaves – inner nodes contain indexed intervals
  - linking leaf pages for efficient range queries

# Princip B-stromu

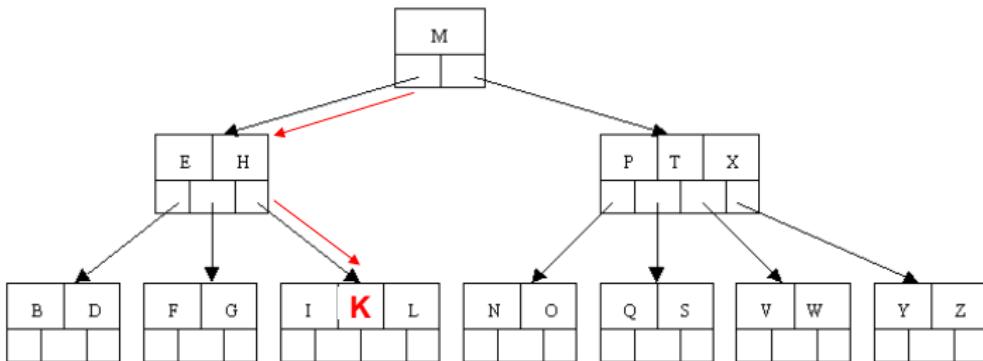


B-strom má definovánu:

- maximální kapacitu uzlu (max. počet záznamů v uzlu)
- minimální kapacitu uzlu (min. počet záznamů v uzlu)

Záznamy uvnitř uzlu jsou setříděné podle hodnoty klíče.

# Princip B-stromu



- Hloubka B-stromu

v nejlepším případě (všechny uzly naplněny na 100%) ...

$$\log_{\max} n$$

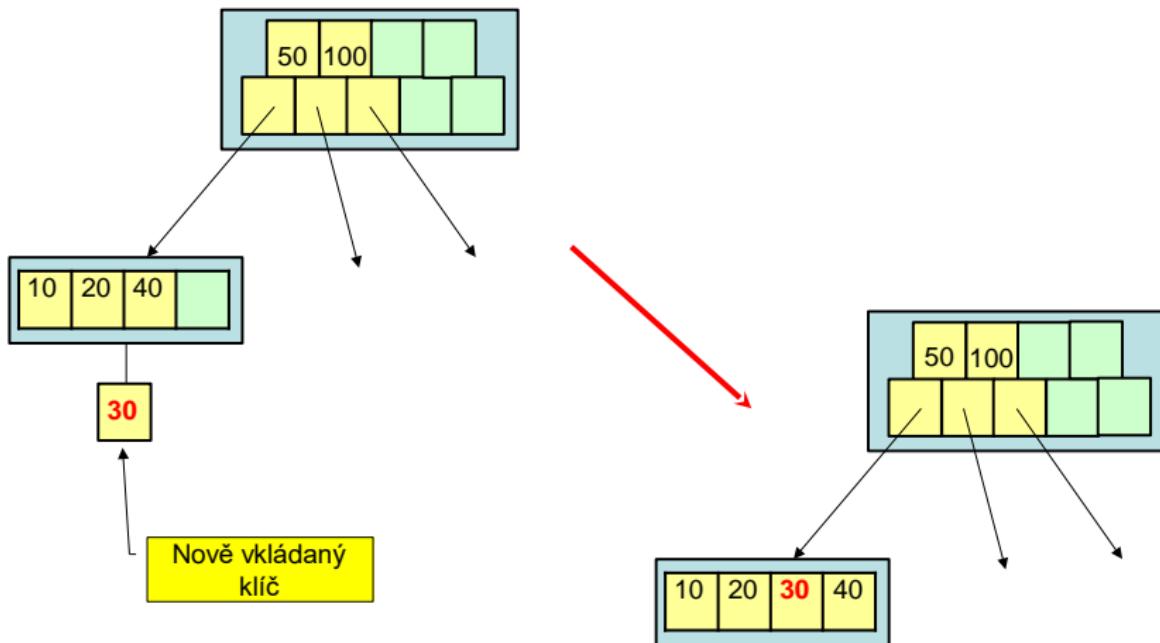
v nejhorším případě (všechny uzly naplněny na *min*) ...

$$\log_{\min} n$$

*Max / min* ... maximální / minimální počet záznamů v uzlu  
*n* ... počet záznamů v databázi

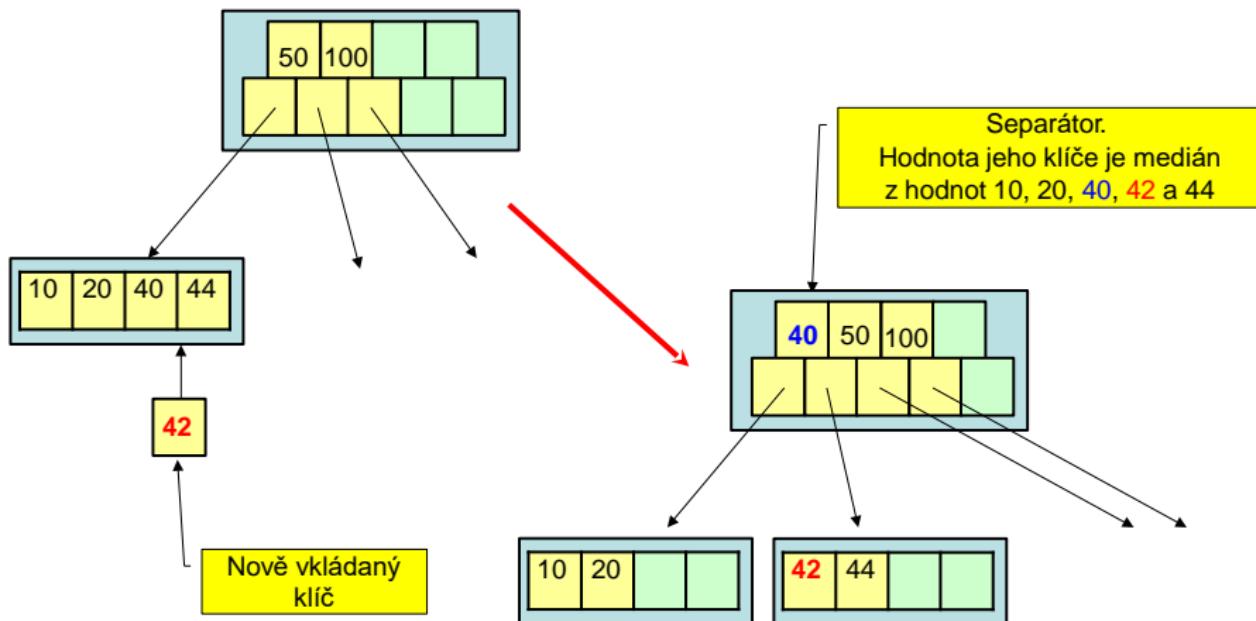
# Vkládání záznamu do B-stromu

Triviální, není-li kapacita daného uzlu naplněna



# Vkládání záznamu do B-stromu

Je-li kapacita daného uzlu naplněna, musí dojít k rozdělení uzelů:



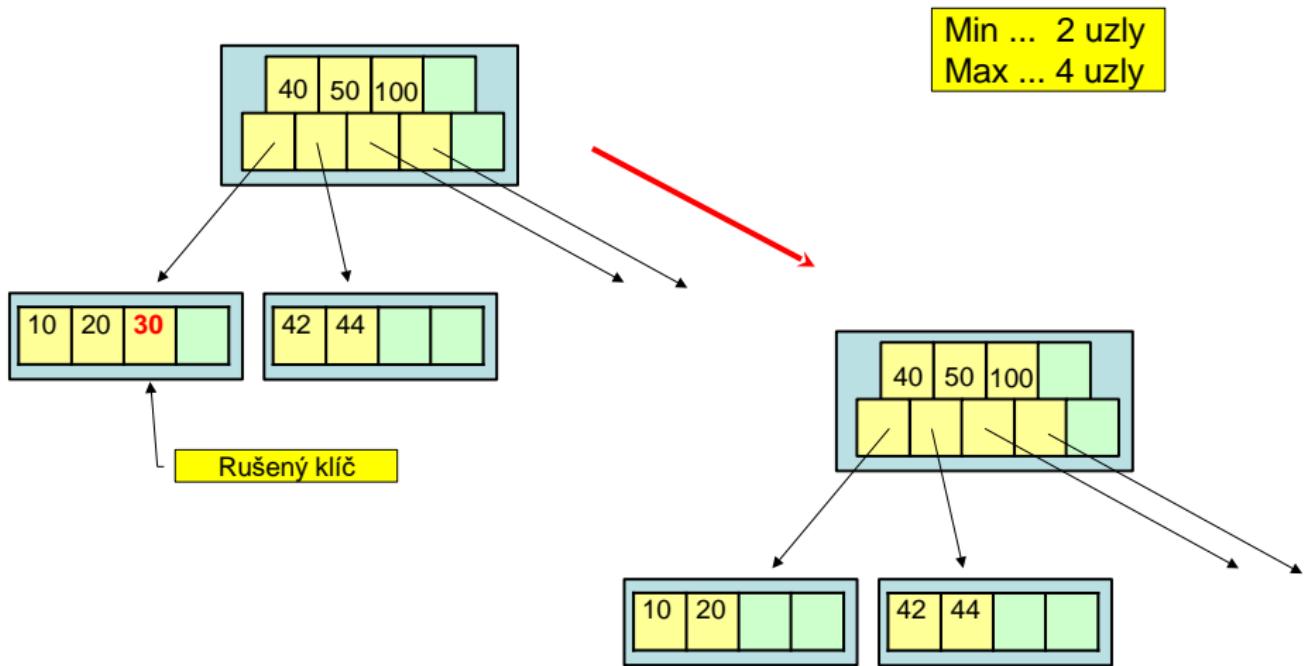
# Vkládání záznamu do B-stromu

## Algoritmus:

1. Pokud uzel, do něhož máme přidat nový záznam, obsahuje méně záznamů, než je jeho kapacita (maximální počet záznamů, který se „vejde“ do jednoho uzlu), vložíme nový záznam do tohoto uzlu  
při zachování uspořádání záznamů.
2. V opačném případě (uzel je naplněn na svou kapacitu), rozdělíme stávající uzel na dva uzly.
  - a. Nalezneme separační záznam jako medián množiny klíčů, která je tvořena klíči záznamů existujících v děleném uzlu plus klíče vkládaného uzlu.
  - b. Záznamy s klíči menšími než klíč separačního záznamu necháme v původním uzlu, záznamy s klíčem větším než klíč separačního záznamu uložíme do nového uzlu (zařazeného napravo od původního uzlu).
  - c. Separaci záznam vložíme do rodičovského uzlu, který se zase může rozlomit, pokud jeho kapacita již byla naplněna. Pokud uzel nemá rodiče (t.j. byl kořenem B-stromu), vytvoříme nový kořen nad tímto uzlem (dojde ke zvětšení hloubky stromu).

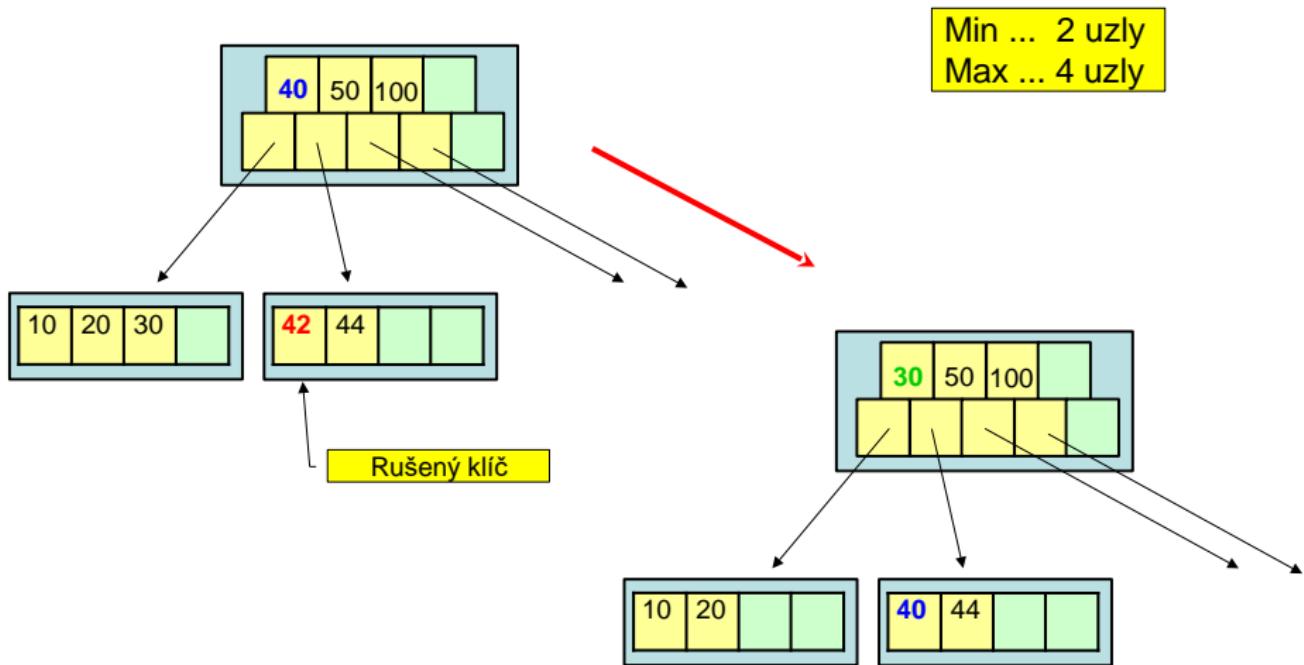
# Rušení záznamu v listu B-stromu

## Základní varianta



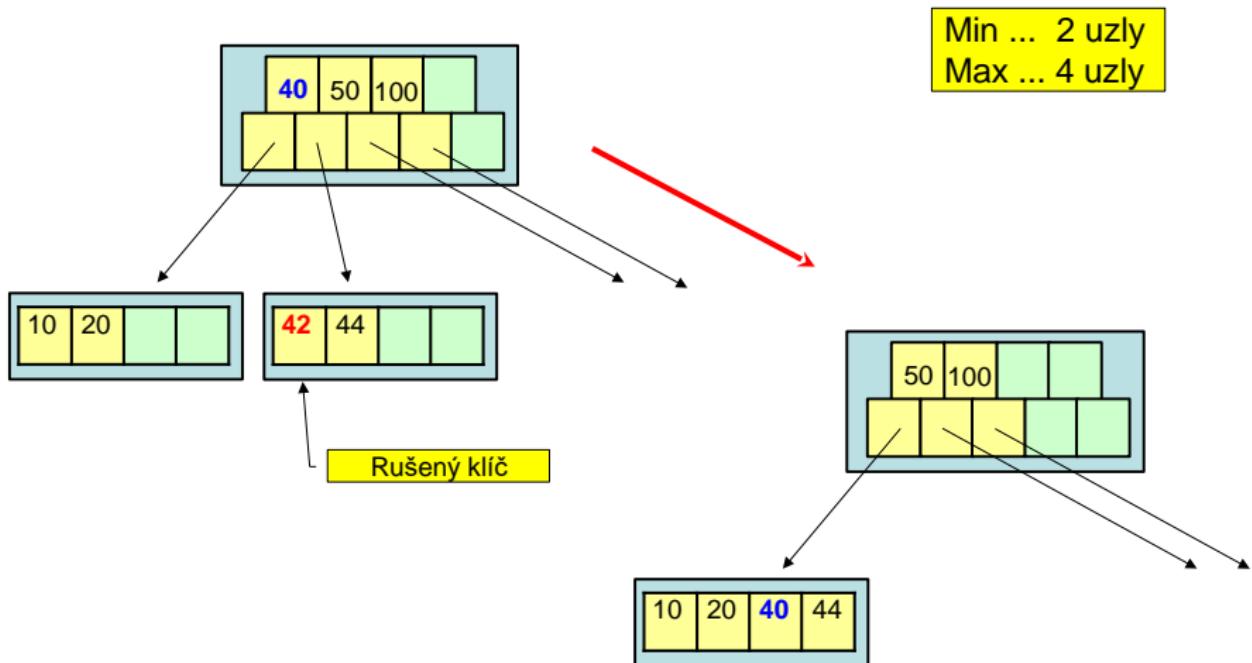
# Rušení záznamu v listu B-stromu

## Přesun hodnot



# Rušení záznamu v listu B-stromu

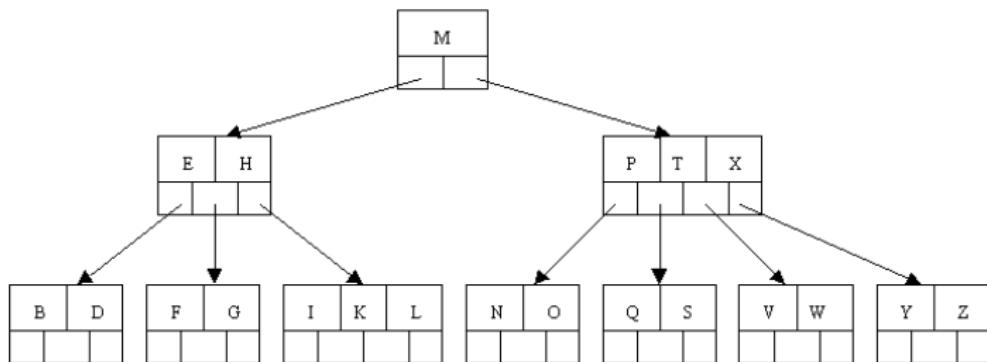
## Spojení uzlů



# Rušení záznamu v **listu** B-stromu

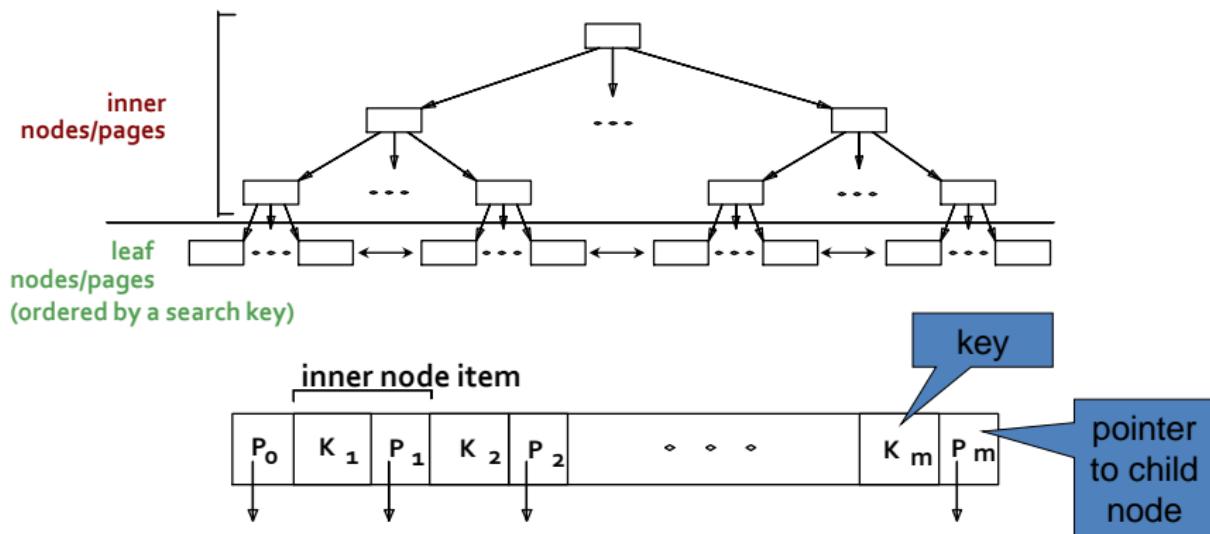
- Rušení záznamu v **listu** B-stromu je jednodušší než rušení záznamu v nelistovém uzlu.
- Pokud po zrušení záznamu klesne počet záznamů pod minimální kapacitu a sourozenecký uzel má více záznamů, než je minimální kapacita uzlu, **přesuneme** záznam ze sourozeneckého uzlu.
- Klesne-li počet záznamů v obou sourozeneckých uzlech pod minimální kapacitu uzlu, uzly **spojíme**.

# Od B-stromu k B<sup>+</sup>-stromu



Hodnoty nemusí být unikátní  
Hodnoty jsou jen v listech  
Listy obsahují říd záznamů

# B<sup>+</sup>-tree, schema



# Hashed index

- similar to hashed data file
  - i.e., buckets + hashing function
- buckets contain only key values together with the **rids**
- same pros/cons

# Bitmaps

- suitable for indexing attributes of low-cardinality data types
  - e.g., attribute **FAMILY\_STATUS** = {single, married, divorced, widow}
- for each value **h** of an indexed attribute **a** a bitmap (binary vector) is constructed, where 1 on *i*<sup>th</sup> position means the value **h** appears in the *i*<sup>th</sup> record (in the attribute **a**), while it holds
  - bitwise OR = 1 (every attribute has a value)
  - bitwise AND = 0 (attribute values are deterministic)

(like bool etc.)

Name	Address	Family status
John Smith	London	single
Rostislav Drobil	Prague	married
Franz Neumann	Munich	married
Fero Lakatoš	Malacky	single
Sergey Prokofjev	Moscow	divorced

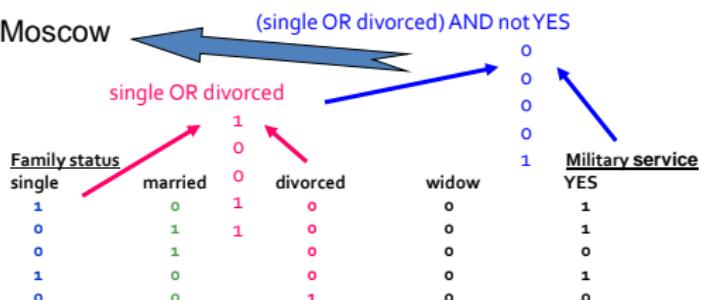
single	married	divorced	widow
1	0	0	0
0	1	0	0
0	1	0	0
1	0	0	0
0	0	1	0

# Bitmaps

- query evaluation
  - bitwise operations with attribute bitmaps
  - resulting bitmap marks the queried records
- example
  - Which single or divorced people did not complete the military service?**  
 $(\text{bitmap}(\text{single}) \text{ OR } \text{bitmap}(\text{divorced})) \text{ AND not } \text{bitmap}(\text{YES})$

Name	Address	Military service	Family status
John Smith	London	YES	single
Rostislav Drobil	Prague	YES	married
Franz Neumann	Munich	NO	married
Fero Lakatoš	Malacky	YES	single
Sergey Prokofjev	Moscow	NO	divorced

answer: Sergey Prokofjev, Moscow



# Bitmaps

- pros
  - efficient storage, could be also compressed
  - fast query processing, bitwise operations are fast
  - easy parallelization
- cons
  - suitable only for attributes with small cardinality domain
  - range queries get slow linearly with the number of values in the range (bitmaps for all the values must be processed)



B0B36DBS, BD6B36DBS: **Database Systems**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/>

Lecture 9

# **Functional Dependencies**

Authors: **Tomáš Skopal, Irena Holubová**

Lecturer: **Martin Svoboda**, martin.svoboda@fel.cvut.cz

14. 4. 2020

**Czech Technical University in Prague, Faculty of Electrical Engineering**

# Today's lecture outline

- motivation
  - data redundancy and update/insertion/deletion anomalies
- functional dependencies
  - Armstrong's axioms
  - attribute and dependency closures
- normal forms
  - 3NF
  - BCNF

→ to see functional dependencies

# **Functional Dependencies**

# Motivation

- result of **relational design** = a set of relational schemas
- problems:
  - **data redundancy**
    - unnecessary multiple storage of the same data
    - increased **space cost**
  - **insert/update/deletion anomalies**
    - insertions and updates must preserve redundant data storage
    - deletion might cause **loss of some data**
  - **null values**
    - **unnecessary empty space**
    - increased space cost
- solution
  - relational schema **normalization**

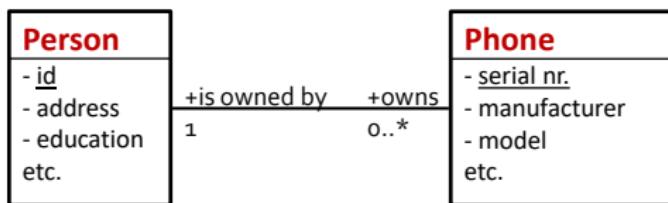
# Example of “abnormal” schema

Empld	Name	Position	Hourly salary	Hours completed
1	John Goodman	accountant	200	50
2	Paul Newman	salesman	500	30
3	David Houseman	salesman	500	45
4	Brad Pittman	accountant	200	70
5	Peter Hitman	accountant	200	66
6	Adam Batman	lecturer	300	10

- 1) From functional analysis we know that position determines hourly salary:  
**However, hourly salary data is stored multiple times – redundancy.**
- 2) If we delete employee 6, we **lose the information on lecturer salary.**
- 3) If we **change** the accountant hourly salary, we **must do that in three places.**

# How could this even happen?

- simply
  - during “manual” design of relation schemas
  - badly designed conceptual model
    - e.g., too many attributes in a class



the UML diagram results in 2 tables:

Person(id, address, education, ...)

Mobil(serial nr., manufacturer, model, ..., id)

# How could this even happen?

Serial nr.	Manufacturer	Model	Made in	Certificate
13458	Nokia	Lumia	Finland	EU, USA
34654	Nokia	Lumia	Finland	EU, USA
65454	Nokia	Lumia	Finland	EU, USA
45464	Apple	iPhone 4S	USA	EU, USA
64654	Samsung	Galaxy S2	Taiwan	Asia, USA
65787	Samsung	Galaxy S2	Taiwan	Asia, USA

Redundancy in attributes Manufacturer, Model, Made in, Certificate

What happened?

Class Phone includes also other classes – Manufacturer, Model, ...

How to fix it?

Two options

- 1) fix the UML model (design of more classes)
- 2) alter the already created schemas (see next)

# Functional dependencies

- attribute-based integrity constraints defined by the user
  - e.g., DB application designer
- a kind of alternative to conceptual modelling
  - ER and UML invented much later
- functional dependency (FD)  $X \rightarrow Y$  over schema  $R(A)$ 
  - mapping  $f_i : X_i \rightarrow Y_i$ , where  $X_i, Y_i \subseteq A$  (where  $i = 1..$  number of FDs in  $R(A)$ )
  - $n$ -tuple from  $X_i$  determines  $m$ -tuple from  $Y_i$
  - $m$ -tuple from  $Y_i$  is determined by (is dependent on)  $n$ -tuple from  $X_i$

# Functional dependencies

- simply, for  $X \rightarrow Y$ ,  
values in X **together determine** the values in Y
- if  $X \rightarrow Y$  and  $Y \rightarrow X$ , then X and Y are **functionally equivalent**
  - could be denoted as  $X \leftrightarrow Y$
- if  $X \rightarrow a$ , where  $a \in A$ , then  $X \rightarrow a$  is an **elementary FD**
  - i.e., only a single attribute on right-hand side
- FDs represent a **generalization of the key concept (identifier)**
  - **key is a special case**, see next slides

# Example – wrong interpretation

Empld	Name	Position	Hourly salary	Hours completed
1	John Goodman	accountant	200	50
2	Paul Newman	salesman	500	30
3	David Houseman	salesman	500	45
4	Brad Pittman	accountant	200	70
5	Peter Hitman	accountant	200	66
6	Adam Batman	lecturer	300	10

One might **observe** from the **data**, that:

**Position → Hourly salary** and also **Hourly salary → Position**

**Empld → everything**

**Hours completed → everything**

**Name → everything**

**(but that is nonsense w.r.t. the natural meaning of the attributes)**

# Example – wrong interpretation

Empld	Name	Position	Hourly salary	Hours completed
1	John Goodman	accountant	200	50
2	Paul Newman	salesman	500	30
3	David Houseman	salesman	500	45
4	Brad Pittman	accountant	200	70
5	Peter Hitman	accountant	200	66
6	Adam Batman	lecturer	300	10
7	Fred Whitman	advisor	300	70
8	Peter Hitman	salesman	500	55

newly  
inserted  
records

Position → Hourly salary  
Empld → everything

~~Hourly salary → Position~~  
~~Hours completed → everything~~  
~~Name → everything~~

# Example – correct interpretation

- at first, after the data analysis the FDs are set “forever”, **limiting** the content of the tables
  - e.g., **Position → Hourly salary**  
**Empld → everything**
  - insertion of the last row is **not allowed** as it violates both the FDs

Empld	Name	Position	Hourly salary	Hours completed
1	John Goodman	accountant	200	50
2	Paul Newman	salesman	500	30
3	David Houseman	salesman	500	45
4	Brad Pittman	accountant	200	70
5	Peter Hitman	accountant	200	66
5	Adam Batman	salesman	300	23

# Armstrong's axioms

Let us have  $R(A, F)$ . Let  $X, Y, Z \subseteq A$  and  $F$  is the set of FDs

- 1) if  $Y \subseteq X$ , then  $X \rightarrow Y$  (trivial FD)
  - 2) if  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$  (transitivity)
  - 3) if  $X \rightarrow Y$  and  $X \rightarrow Z$ , then  $X \rightarrow YZ$  (composition)
  - 4) if  $X \rightarrow YZ$ , then  $X \rightarrow Y$  and  $X \rightarrow Z$  (decomposition)
- Známe riaďko 'y už.*

# Armstrong's axioms

Armstrong's axioms:

- **are correct (sound)**
  - what is derived from F is valid for any instance from R
- **are complete**
  - all FDs valid in all instances in R (w.r.t. F) can be derived using the axioms
- **1,2,3 (trivial, transitivity, composition) are independent**
  - removal of any axiom 1,2,3 violates the completeness (decomposition could be derived from trivial FD and transitivity)

# Example – deriving FDs

$R(A,F)$

$A = \{a,b,c,d,e\}$

$F = \{ab \rightarrow c, ac \rightarrow d, cd \rightarrow ed, e \rightarrow f\}$

We could derive, e.g.,:

$ab \rightarrow a$  (trivial)

$ab \rightarrow ac$  (composition with  $ab \rightarrow c$ )

$ab \rightarrow d$  (transitivity with  $ac \rightarrow d$ )

$ab \rightarrow cd$  (composition with  $ab \rightarrow c$ )

$ab \rightarrow ed$  (transitivity with  $cd \rightarrow ed$ )

$ab \rightarrow e$  (decomposition)

$ab \rightarrow f$  (transitivity)

## Example – deriving the decomposition rule

$R(A,F)$

$A = \{a,b,c\}$

$F = \{a \rightarrow bc\}$

Deriving:

$a \rightarrow bc$  (assumption)

$bc \rightarrow b$  (trivial FD)

$bc \rightarrow c$  (trivial FD)

$a \rightarrow b$  (transitivity)

$a \rightarrow c$  (transitivity)

i.e.,  $a \rightarrow bc \Rightarrow a \rightarrow b \wedge a \rightarrow c$

# Closure of set of FDs

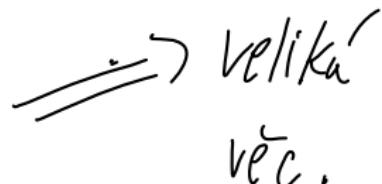
- **closure  $F^+$**  of FDs set  $F$  (FD closure) is the set of all FDs derivable from  $F$  using the Armstrong's axioms
  - generally exponential size w.r.t.  $|F|$

# Example – closure of set of FDs

$R(A, F)$ ,  $A = \{a, b, c, d\}$ ,  $F = \{ab \rightarrow c, cd \rightarrow b, ad \rightarrow c\}$

$F^+ =$

{ $a \rightarrow a, b \rightarrow b, c \rightarrow c, d \rightarrow d,$   
 $ab \rightarrow a, ab \rightarrow b, ab \rightarrow c,$   
 $cd \rightarrow b, cd \rightarrow c, cd \rightarrow d,$   
 $ad \rightarrow a, ad \rightarrow c, ad \rightarrow d,$   
 $abd \rightarrow a, abd \rightarrow b, abd \rightarrow c, abd \rightarrow d,$   
 $abd \rightarrow abcd, \dots\}$

 velika  
več.

# Cover

- **cover** of a set  $F$  is any set of FDs  $G$  such that  $F^+ = G^+$ 
  - i.e., a set of FDs which have the same closure (= generate the same set of FDs)
- **canonical cover** = cover consisting of **elementary FDs**
  - decompositions are performed to obtain singleton sets on the right-hand side

# Example – cover

R1(A,F), R2(A,G),

A = {a,b,c,d},

F = {a → c, b → ac, d → abc},

G = {a → c, b → a, d → b}

For checking that  $G^+ = F^+$  we do not have to establish the whole covers, it is sufficient to derive F from G, and vice versa, i.e.,

$F' = \{a \rightarrow c, b \rightarrow a, d \rightarrow b\}$  – decomposition

$G' = \{a \rightarrow c, b \rightarrow ac, d \rightarrow abc\}$  – transitivity and composition

$\Rightarrow G^+ = F^+$

Schemas R1 and R2 are equivalent because G is cover of F, while they share the attribute set A.

Moreover, G is minimal cover, while F is not (for minimal cover see next slides).

# Redundant FDs

- FD  $f$  is **redundant** in  $F$  if  $(F - \{f\})^+ = F^+$ 
  - i.e.,  $f$  can be derived from the rest of  $F$
- **non-redundant cover** of  $F$  = cover of  $F$  after **removing all redundant FDs**
  - note the order of removing FDs matters – a redundant FD could become non-redundant FD after removing another redundant FD
  - i.e., there may exist multiple non-redundant covers of  $F$

# Example – redundant FDs

$R(A,F)$

$A = \{a,b,c,d\}$ ,

$F = \{a \rightarrow c, b \rightarrow a, b \rightarrow c, d \rightarrow a, d \rightarrow b, d \rightarrow c\}$

FDs  $b \rightarrow c, d \rightarrow a, d \rightarrow c$  are redundant

after their removal  $F^+$  is not changed, i.e., they could be derived from the remaining FDs

$b \rightarrow c$  derived using transitivity  $a \rightarrow c, b \rightarrow a$

$d \rightarrow a$  derived using transitivity  $d \rightarrow b, b \rightarrow a$

$d \rightarrow c$  derived using transitivity  $d \rightarrow b, b \rightarrow a, a \rightarrow c$

# Attribute closure, key

- **attribute closure  $X^+$**  (w.r.t.  $F$ ) is a subset of attributes from  $A$  determined by  $X$  (using  $F$ )
  - consequence: if  $X^+ = A$ , then  $X$  is a **super-key**
- if  $F$  contains a FD  $X \rightarrow Y$  and there exist an attribute  $a$  in  $X$  such that  $Y \subseteq (X - a)^+$ , then  $a$  is an **attribute redundant in  $X \rightarrow Y$** 
  - i.e., we do not need  $a$  in  $X$  to determine right-hand side  $Y$
- **reduced FD** does not contain any redundant attributes
- For  $R(A)$  **key** is a set  $K \subseteq A$  s.t. it is a super-key (i.e.,  $K \rightarrow A$ ) and  $K \rightarrow A$  is reduced
  - there could exist multiple keys (at least one)
  - if there is no FD in  $F$ , it trivially holds  $A \rightarrow A$ , i.e., the key is the entire set  $A$
  - **key attribute** = attribute that is in **any** key

# Example – attribute closure

$R(A,F)$ ,  $A = \{a,b,c,d\}$ ,  $F = \{a \rightarrow c, cd \rightarrow b, ad \rightarrow c\}$

$\{a\}^+ = \{a, c\}$  it holds  $a \rightarrow c$  (+ trivial  $a \rightarrow a$ )

$\{b\}^+ = \{b\}$  (trivial  $b \rightarrow b$ )

$\{c\}^+ = \{c\}$  (trivial  $c \rightarrow c$ )

$\{d\}^+ = \{d\}$  (trivial  $d \rightarrow d$ )

$\{a,b\}^+ = \{a,b,c\}$   $a \rightarrow c$  (+ trivial)

$\{a,d\}^+ = \{a,b,c,d\}$   $ad \rightarrow c, cd \rightarrow b$  (+ trivial)

$\{c,d\}^+ = \{b,c,d\}$   $cd \rightarrow b$  (+ trivial)

# Example – redundant attribute

$R(A,F)$ ,  $A = \{i,j,k,l,m\}$ ,

$F = \{m \rightarrow k, lm \rightarrow j, ijk \rightarrow l, j \rightarrow m, l \rightarrow i, l \rightarrow k\}$

## Hypothesis:

$k$  is redundant in  $ijk \rightarrow l$ , i.e., it holds  $ij \rightarrow l$

## Proof:

1. based on the hypothesis let's construct FD  $ij \rightarrow ?$
2. note that  $ijk \rightarrow l$  remains in  $F$  because we **ADD** new FD  $ij \rightarrow ?$   
⇒ so we can use  $ijk \rightarrow l$  for construction of the attribute closure  $\{i,j\}^+$
3. we obtain  $\{i,j\}^+ = \{i, j, m, k, l\}$ ,  
i.e., there exists  $ij \rightarrow l$  which we add into  $F$  (it is the member of  $F^+$ )
4. now forget how  $ij \rightarrow l$  got into  $F$
5. because  $ijk \rightarrow l$  could be trivially derived from  $ij \rightarrow l$ ,  
it is redundant FD and we can remove it from  $F$
6. so, we removed the redundant attribute  $k$  in  $ijk \rightarrow l$

In other words, we transformed the problem of removing redundant attribute  
on the problem of removing redundant FD.

# FDs vs. attributes

## FDs:

- can be redundant
  - “we don’t need it”
- can have a closure
  - “all derivable FDs”
- can be elementary
  - “single attribute on the right-hand side”
- can be reduced
  - “no redundancies on the left-hand side”

## Attributes:

- can be redundant
  - “we don’t need it”
- can have a closure
  - “all derivable attributes”
- can form (super-)keys

# Minimal cover

- **non-redundant canonical cover that consists of only reduced FDs**
  - i.e. **no redundant FDs, no redundant attributes, decomposed FDs**
  - is constructed by removing **redundant attributes in FDs followed by removing of redundant FDs**
    - i.e., the order matters!!!

Example:  $abcd \rightarrow e$ ,  $e \rightarrow d$ ,  $a \rightarrow b$ ,  $ac \rightarrow d$

## Correct order of reduction:

1. b,d are redundant  
in  $abcd \rightarrow e$ , i.e., removing them
2.  $ac \rightarrow d$  is redundant

*remove same physical attributes, so joinanic, pak*

## Wrong order of reduction:

1. no redundant FD
2. redundant b,d in  $abcd \rightarrow e$   
(now not a minimal cover, because  $ac \rightarrow d$  is redundant)

# Keys

# Determining (first) key

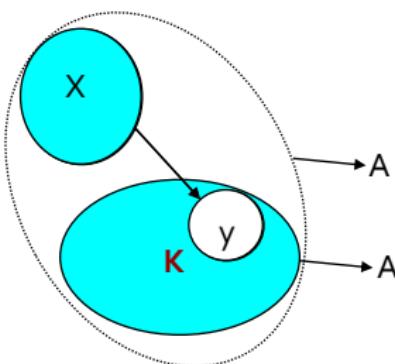
- redundant attributes are iteratively removed from left-hand side of trivial FD  $A \rightarrow A$

```
algorithm GetFirstKey(set of deps. F, set of attributes A)
  : returns a key K;
  return ReduceAttributes(F, A → A);
```

Note: Because multiple keys can exists, the algorithm finds only one of them.  
Which one? It depends on the traversing of the attribute set within the algorithm ReduceAttributes.

# Determining all keys, the principle

Let us have a schema  $S(A, F)$ .  
Simplify  $F$  to minimal cover.



1. Find any key  $K$  (see the previous slide).
2. Take a FD  $X \rightarrow y$  in  $F$  such that  $y \in K$  or terminate if not exists (there is no other key).
3. Because  $X \rightarrow y$  and  $K \rightarrow A$ , it transitively holds also  $X\{K - y\} \rightarrow A$ , i.e.,  $X\{K - y\}$  is super-key.
4. Reduce FD  $X\{K - y\} \rightarrow A$  so we obtain key  $K'$  on the left-hand side.  
This key is surely different from  $K$  (we removed  $y$ ).
5. If  $K'$  is not among the determined keys so far, we add it, declare  $K = K'$  and continue from step 2.  
Otherwise we finish.

# Determining all keys, the algorithm

- Formally: **Lucchesi-Osborn algorithm**
  - having an already determined key, we search for equivalent sets of attributes, i.e., other keys
- NP-complete problem (theoretically exponential number of keys/FDs)

```
algorithm GetAllKeys(set of FDs F, set of attr. A)
    : returns set of all keys Keys;
    let all dependencies in F be non-trivial
    K := GetFirstKey(F, A);
    Keys := {K};
    for each K in Keys do
        for each X → Y in F do
            if (Y ∩ K ≠ ∅ and ¬∃K' ∈ Keys : K' ⊆ (K ∪ X) - Y) then
                N := ReduceAttributes(F, ((K ∪ X) - Y) → A);
                Keys := Keys ∪ {N};
            endif
        endfor
    endfor
return Keys;
```

# Example – determining all keys

Contracts(A, F)

A = {c = **ContractId**, s = **SupplierId**, j = **ProjectId**, d = **DeptId**,  
p = **PartId**, q = **Quantity**, v = **Value**}  
F = {**c** → *all*, **sd** → **p**, **p** → **d**, **jp** → **c**, **j** → **s**}

1. Determine the first key – Keys = {**c**}
2. Iteration 1: take **jp** → **c** that has a part of the last key on the right-hand side (in this case the whole key – **c**) and **jp** is not a super-set of already determined key
3. **jp** → *all* is reduced (no redundant attribute), i.e.,  
Keys = {**c**, **jp**}
4. Iteration 2: take **sd** → **p** that has a part of the last key on the right-hand side (**jp**),  
{**jsd**} is not a super-set of **c** nor **jp**, i.e., it is a key candidate
5. in **jsd** → *all* we get redundant attribute **s**, i.e.,  
Keys = {**c**, **jp**, **jd**}
6. Iteration 3: take **p** → **d**, however, **jp** was already found so we do not add it
7. Finish as the iteration 3 resulted in no key addition.

# **Normal Forms**

# First normal form (1NF)

Every attribute in a relational schema is of  
**simple non-structured type.**

- 1NF is the basic condition on „flat database“
- a table is really two-dimensional array
  - not involving arrays, subtables, trees, structures, ...

# Example – 1NF

Person(Id: **Integer**, Name: **String**, Birth: **Date**)

**is in 1NF**

Employee(Id: **Integer**, Subordinate : Person[ ], Boss : Person)

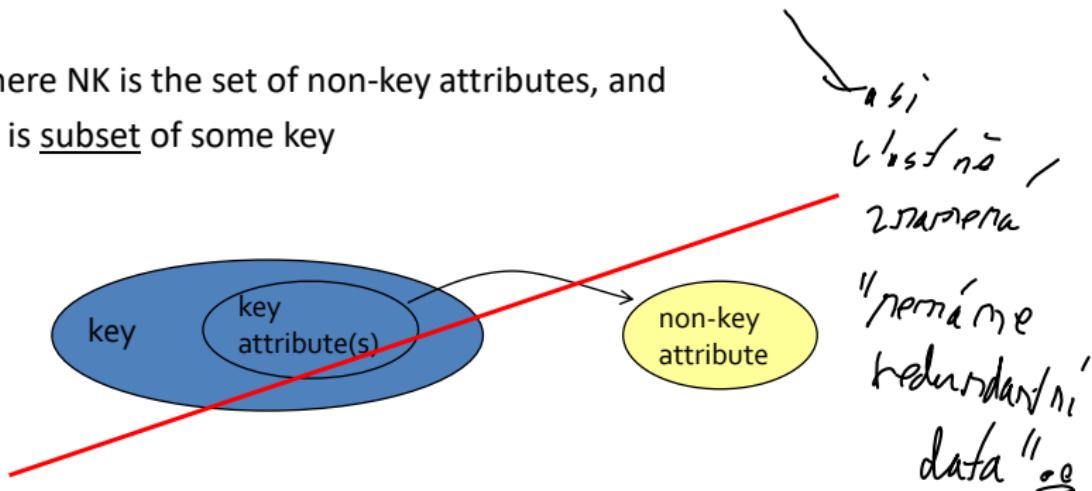
**not in 1NF**

(nested table of type Person in attribute Subordinate,  
and the Boss attribute is structured)

# 2<sup>nd</sup> normal form (2NF)

- there **do not exist** partial dependencies of non-key attributes on (any) key, i.e.,  
it holds  $\forall x \in NK \nexists KK : KK \rightarrow x$

- where NK is the set of non-key attributes, and
- KK is subset of some key



# Example – 2NF

Company	DB server	HQ	Purchase date
John's firm	Oracle	Paris	1995
John's firm	MS SQL	Paris	2001
Paul's firm	IBM DB2	London	2004
Paul's firm	MS SQL	London	2002
Paul's firm	Oracle	London	2005

← not in 2NF, because HQ is determined by a part of key (Company)  
consequence:  
**redundancy of HQ values**

Company, DB Server → everything

Company → HQ

both schemas are in 2NF →

Company	DB server	Purchase date
John's firm	Oracle	1995
John's firm	MS SQL	2001
Paul's firm	IBM DB2	2004
Paul's firm	MS SQL	2002
Paul's firm	Oracle	2005

Company	HQ
John's firm	Paris
Paul's firm	London

Company → HQ

Company, DB Server → everything

# Transitive dependency on key

- FD  $A \rightarrow B$  such that  $A \not\rightarrow some\ key$   
( $A$  is not a super-key), i.e., we get transitivity  $key \rightarrow A \rightarrow B$
- i.e., unique values of **key** are mapped to the same or **less** unique values of **A**, and those are mapped to the same or **less** unique values of **B**

Example in 2NF:

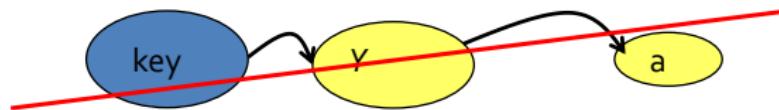
ZIPcode  $\rightarrow$  City  $\rightarrow$  Country

ZIPcode	City	Country
CZ 118 00	Prague	Czech rep.
CZ 190 00	Prague	Czech rep.
CZ 772 00	Olomouc	Czech rep.
CZ 783 71	Olomouc	Czech rep.
SK 911 01	Trenčín	Slovak rep.

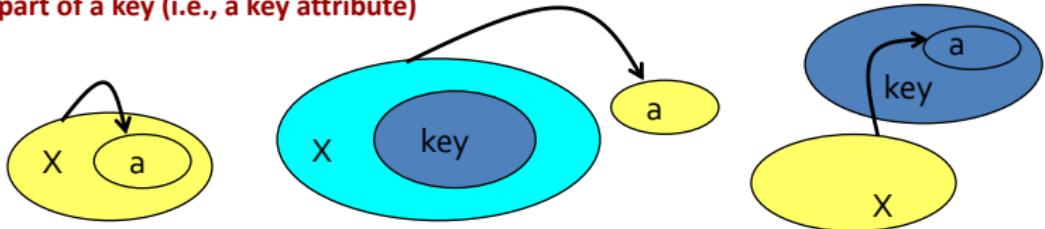
no redundancy   medium redundancy   high redundancy

# 3<sup>rd</sup> normal form (3NF)

- non-key attributes **are not transitively dependent** on key



- note: as the 3NF using the above definition cannot be tested without construction of  $F^+$ , we use a definition that assumes only  $R(A, F)$ :
- at least one condition holds for each FD  $X \rightarrow a$  (where  $X \subseteq A, a \in A$ ):
  - **FD is trivial**
  - **X is super-key**
  - **a is part of a key (i.e., a key attribute)**



# Example – 3NF

Company	HQ	ZIPcode
John's firm	Prague	CZ 11800
Paul's firm	Ostrava	CZ 70833
Martin's firm	Brno	CZ 22012
David's firm	Prague	CZ 11000
Peter's firm	Brno	CZ 22012

Company → everything  
ZIPcode → HQ

is in 2NF, **not in 3NF** (transitive dependency of HQ on key through ZIPcode)

consequence:  
redundancy of HQ values

Company → everything  
ZIPcode → everything

both schemas **are in 3NF**

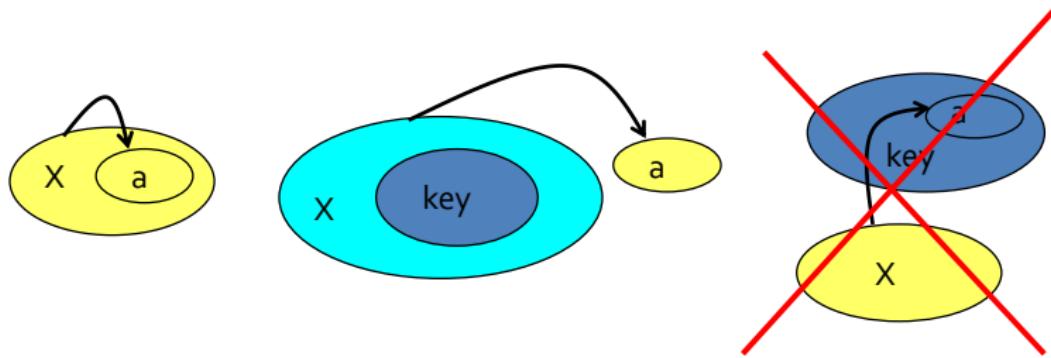
Company	ZIPcode
John's firm	CZ 11800
Paul's firm	CZ 70833
Martin's firm	CZ 22012
David's firm	CZ 11000
Peter's firm	CZ 22012

ZIPcode	HQ
CZ 11800	Prague
CZ 70833	Ostrava
CZ 22012	Brno
CZ 11000	Prague

posse  
in tam vhe  
nikle nejso u  
redundancy  
data

# Boyce-Codd normal form (BCNF)

- **every attribute** is (non-transitively) dependent on key
- more exactly, in a given schema  $R(A, F)$  there holds at least one condition for each FD  $X \rightarrow a$  (where  $X \subseteq A, a \in A$ ):
  - FD is trivial
  - X is super-key
- note: the same as 3NF without the last option ( $a$  is key attribute)



# Example – BCNF

Destination	Pilot	Plane	Day
Paris	cpt. Oiseau	Boeing #1	Monday
Paris	cpt. Oiseau	Boeing #2	Tuesday
Berlin	cpt. Vogel	Airbus #1	Monday

Pilot, Day → *everything*

Plane, Day → *everything*

Destination → Pilot

is in 3NF, **not in BCNF**

(Pilot is determined by Destination,  
which is not a super-key)

consequence:

**redundancy of Pilot values**

Destination → Pilot  
Plane, Day → *everything*

Destination	Pilot
Paris	cpt. Oiseau
Berlin	cpt. Vogel

Destination	Plane	Day
Paris	Boeing #1	Monday
Paris	Boeing #2	Tuesday
Berlin	Airbus #1	Monday

both schemas **are in BCNF**



B0B36DBS, BD6B36DBS: **Database Systems**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/>

Lecture 10

# **Database Transactions**

Authors: **Tomáš Skopal, Irena Holubová**

Lecturer: **Martin Svoboda**, martin.svoboda@fel.cvut.cz

21. 4. 2020

**Czech Technical University in Prague, Faculty of Electrical Engineering**

# Today's lecture outline

- motivation and the ACID properties
- schedules („interleaved“ transaction execution)
  - serializability
  - conflicts
  - (non)recoverable schedule
- locking protocols
  - 2PL, strict 2PL, conservative 2PL
  - deadlock and prevention
  - phantom
- alternative protocols

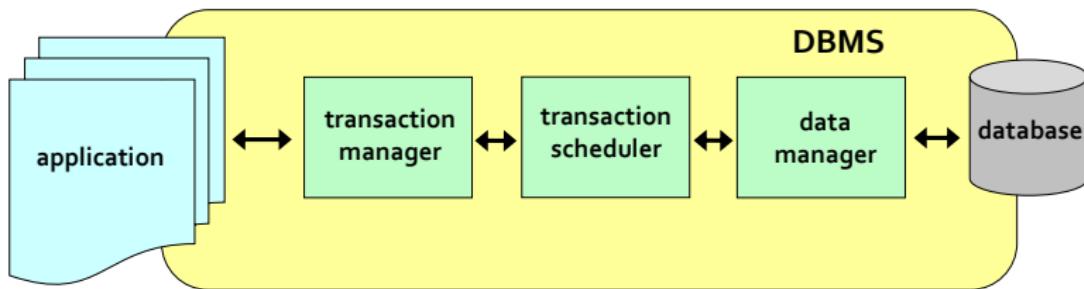
# Motivation

- problem: we need to execute complex database operations
  - e.g., stored procedures, triggers, etc.
  - in a multi-user and parallel environment
- database transaction
  - sequence of actions on database objects (+ others like arithmetic, etc.)
- example:
  - Let us have a bank database with table **Accounts** and the following transaction to transfer the money (pseudocode):

```
transaction PaymentOrder(amount, fromAcc, toAcc)
{
    1. SELECT Balance INTO X FROM Accounts WHERE accNr = fromAcc;
    2. if (X < amount) AbortTransaction("Not enough money!");
    3. UPDATE Accounts SET Balance = Balance - amount WHERE accNr = fromAcc;
    4. UPDATE Accounts SET Balance = Balance + amount WHERE accNr = toAcc;
    5. CommitTransaction;
}
```

# Transaction management in DBMS

- application launches transactions
- **transaction manager** executes transactions
- **scheduler** dynamically schedules the parallel transaction execution, producing a **schedule** (history)
- **data manager** executes partial operation of transactions



# Transaction management in DBMS

- transaction termination
  - **successful** – terminated by **COMMIT** command in the transaction code
    - the performed actions are confirmed
  - **unsuccessful** – transaction is cancelled
    1. termination by the transaction code – **ABORT** (or **ROLLBACK**) command
      - user can be notified
    2. system abort – DBMS aborts the transaction
      - some integrity constraint is violated – user is notified
      - by transaction scheduler (e.g., a deadlock occurs) – user is not notified
    3. system failure – HW failure, power loss – transaction must be restarted
- main objectives of transaction management
  - enforcement of **ACID properties**
  - maximal performance (throughput)
    - parallel/concurrent execution of transactions

# ACID – desired properties of transaction management

- **Atomicity** – partial execution is not allowed (all or nothing)
  - prevents from incorrect transaction termination (or failure)  
= consistency at the DBMS level
- **Consistency**
  - any transaction will bring the database from one consistent (valid) state to another  
= consistency at application level
- **Isolation**
  - transactions executed in parallel do not “see” effects of each other unless committed
  - parallel/concurrent execution is necessary to achieve high throughput
- **Durability**
  - once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors
  - logging necessary (log/journal maintained)



# Transaction

- an executed transaction is a sequence of actions
$$T = \langle A_T^1, A_T^2, \dots, COMMIT \text{ or } ABORT \rangle$$
- basic database actions (operations)
- for now consider a **static database** (no inserts/deletes, just updates), let **A** be a database object (table, row, attribute in row)
  - we omit other actions such as control construct (if, for), etc.
- **READ(A)** – reads A from database
- **WRITE(A)** – writes A to database
- **COMMIT** – confirms executed actions as valid, terminates transaction
- **ABORT** – cancels executed actions, terminates transaction (with error)
- SQL commands **SELECT, INSERT, UPDATE**, could be viewed as transactions implemented using the basic actions (in SQL command **ROLLBACK** is used instead of abort)

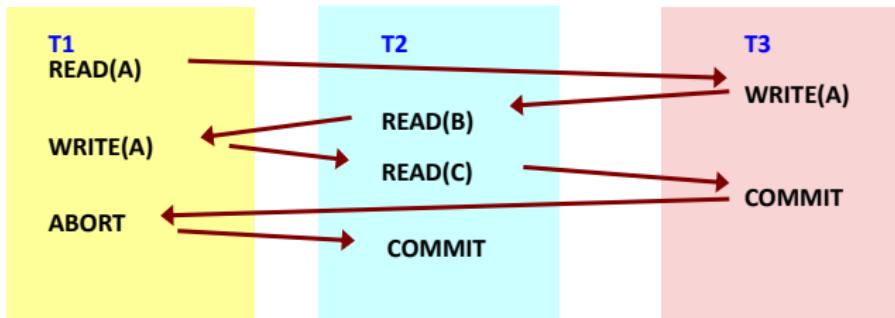
## Example:

Subtract 5 from A (some attribute), such that  $A > 0$ .

```
T = <READ(A), // action 1  
    if(A ≤ 5) then ABORT  
    else WRITE(A - 5), // action 2  
         COMMIT> // action 3  
or  
T = <READ(A), // action 1  
    if(A ≤ 5) then ABORT // action 2  
    else ... >
```

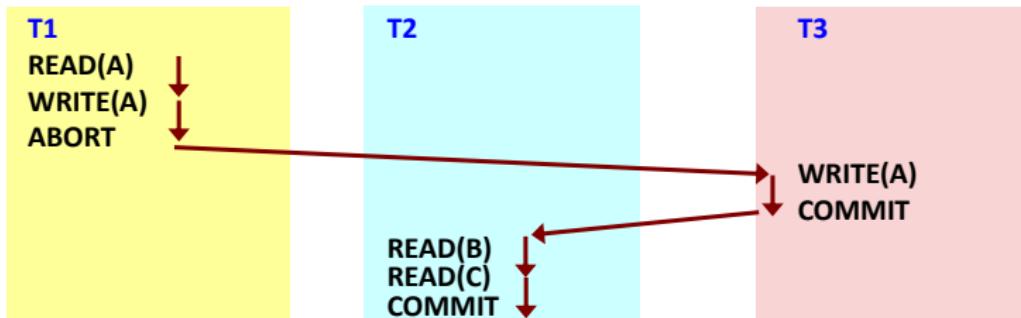
# Transaction programs vs. schedules

- **database program**
  - “design-time” (not running) piece of code (that will be executed as a transaction)
  - i.e., nonlinear – branching, loops, jumps
- **schedule (history)** is a sorted list of actions coming from several transactions (i.e., **transactions as interleaved**)
  - „runtime“ history **of already concurrently executed** actions of **several** transactions
  - i.e., linear – sequence of primitive operations, w/o control constructs



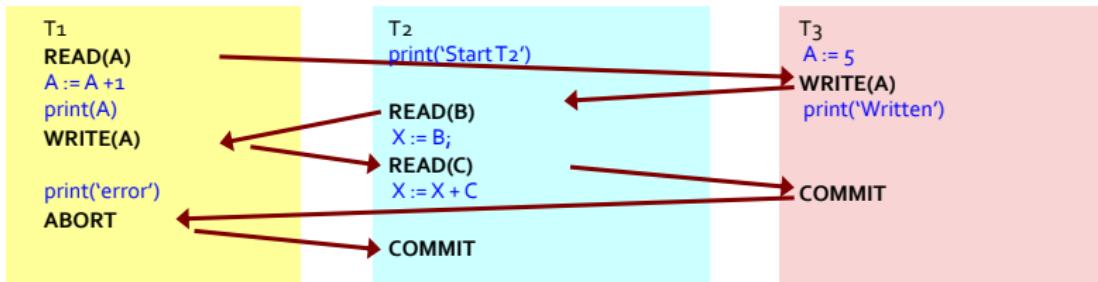
# Serial schedules

- specific schedule, where **all actions of a transaction are coupled together**
  - **no action interleaving**
- given a set  $S$  of transactions, we can obtain  $|S|!$  serial schedules
  - from the definition of ACID properties, all the schedules are equivalent – it does not matter if one transaction is executed before or after another one
    - if it matters, they are not independent and so they should be merged into single transactions
- example:



# Why to interleave transactions?

- every schedule leads to interleaved **sequential** execution of transactions (there is no parallel execution of database operations)
  - simplified model justified by single storage device
- Question: So why to interleave transactions when the number of steps is the same as in a serial schedule?
- two reasons
  - parallel execution of non-database operations with database operations
  - response proportional to transaction complexity (e.g., OldestEmployee vs. ComputeTaxes)
- example



# Serializability

- a schedule is **serializable** if its execution leads to consistent database state, i.e., if the schedule is **equivalent to any serial schedule**
  - for now we consider only committed transactions and a static database
  - note that non-database operations are not considered so that consistency cannot be provided for non-database state (e.g., print on console)
  - it does not matter which serial schedule is equivalent (independent transactions)
- strong property
  - secures the Isolation and Consistency in ACID
- **view serializability** extends serializability by including aborted transactions and dynamic database
  - however, testing is NP-complete, so it is not used in practice
  - instead, **conflict serializability + other techniques** are used

# “Dangers” caused by interleaving

- to achieve serializability (i.e., consistency and isolation), the action of interleaving cannot be arbitrary
- there exist 3 types of local dependencies in the schedule, so-called conflict pairs
- four possibilities of reading/writing the same resource in schedule
  - **read-read** – ok, by reading the transactions do not affect each other
  - **write-read (WR)** – T1 writes, then T2 reads – reading uncommitted data
  - **read-write (RW)** – T1 reads, then T2 writes – unrepeatable reading
  - **write-write (WW)** – T1 writes, then T2 writes – overwrite of uncommitted data



# Conflicts (WR)

- reading uncommitted data (**write-read conflict**)
  - transaction T2 reads A that was earlier updated by transaction T1, but T1 did not commit so far, i.e., T2 reads potentially inconsistent data
    - so-called **dirty read**

Example: T1 transfers 1000 USD from account A to account B (A = 12000, B = 10000)  
T2 adds 1% per account

T1	T2
<b>R(A)</b> // A = 12000	
A := A - 1000	
<b>W(A)</b> // database is now inconsistent – account B still contains the old balance	 <b>R(A)</b> // uncommitted data is read
	<b>R(B)</b>
	A := 1.01*A
	B := 1.01*B
	<b>W(A)</b>
	<b>W(B)</b>
	<b>COMMIT</b>
<b>R(B)</b> // B = 10100	
B := B + 1000	
<b>W(B)</b>	
<b>COMMIT</b>	// inconsistent database, A = 11110, B = 11100

# Conflicts (RW)

- unrepeatable read (**read-write conflict**)
  - transaction T2 writes A that was read earlier by T1 that didn't finish yet
  - T1 cannot repeat the reading of A (A now contains another value)
    - so-called unrepeatable read

Example:      T1 transfers 1000 USD from account A to account B (A = 12000, B = 10000)  
                        T2 adds 1% per account

T1	T2
$R(A)$	$R(A)$
$// A = 12000$	$R(B)$
	$A := 1.01 * A$
	$B := 1.01 * B$
	$W(A)$ // update of A
	$W(B)$
	COMMIT
$//$ database now contains $A = 12120$	
$R(B)$	
$A := A - 1000$	
$W(A)$	
$B := B + 1000$	
$W(B)$	
COMMIT	$//$ inconsistent database, $A = 11000, B = 11100$

A red arrow points from the  $R(A)$  step of T1 to the  $W(A)$  step of T2, indicating that T1's read operation is being overwritten by T2's write operation.

# Conflicts (WW)

- overwrite of uncommitted data (**write-write conflict**)
  - transaction T2 overwrites A that was earlier written by T1 that still runs
  - loss of update (original value of A is lost)
    - so-called **blind write** (update of unread data)

Example: Set the same price to all DVDs.

(let's have two instances of this transaction, one setting price to 10 USD, second 15 USD)

T1

DVD2 := 10  
W(DVD2)

T2

DVD1 := 15  
W(DVD1)

DVD2 := 15  
W(DVD2) // overwrite of uncommitted data  
COMMIT

DVD1 := 10  
W(DVD1)  
COMMIT

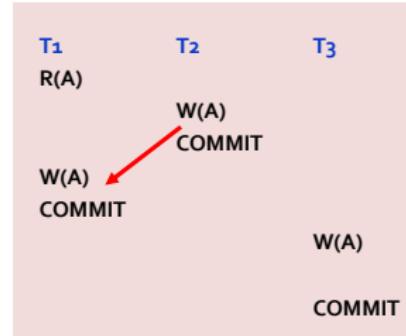
// inconsistent database, DVD1 = 10, DVD2 = 15

# Conflict serializability

- two schedules are **conflict equivalent** if they share the set of conflict pairs
- a schedule is **conflict serializable** if it is conflict-equivalent to some serial schedule, i.e., there are no “real” conflicts
  - more restrictive than serializability (defined only by consistency preservation)
- conflict serializability alone **does not consider:**
  - **cancelled transactions**
    - ABORT/ROLLBACK, so the schedule could be **unrecoverable**
  - **dynamic database** (inserting / deleting database objects)
    - so-called **phantom** may occur
  - hence, conflict serializability is **not sufficient condition** to provide ACID (**view serializability** is ultimate condition)



Example: schedule, that is **serializable** (serial schedule  $\langle T_1, T_2, T_3 \rangle$ ), but is **not conflict serializable** (writes in  $T_1$  and  $T_2$  are in wrong order)

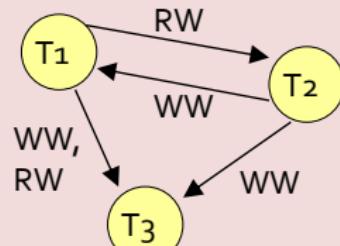


# Detection of conflict serializability

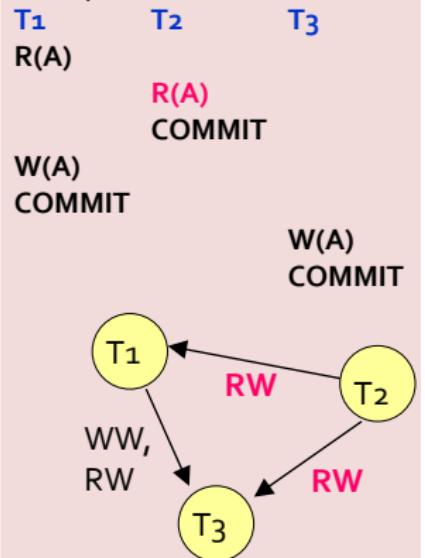
- **precedence graph** (also serializability graph) on a schedule
  - nodes  $T_i$  are **committed** transactions
  - edges represent RW, WR, WW conflicts in the schedule
- schedule is conflict serializable if its **precedence graph is acyclic**

Example: not conflict serializable

T1	T2	T3
R(A)		
	W(A) COMMIT	
W(A) COMMIT		
		W(A) COMMIT



Example: conflict serializable



# Unrecoverable schedule

- at this moment we extend the transaction model by **ABORT** which brings another “danger” – **unrecoverable schedule**
  - one transaction aborts so that undos of every write must be done, however, this cannot be done for already committed transactions that read changes caused by the aborted transaction
    - durability property of ACID
- in **recoverable schedule** a transaction T is **committed** after all other transactions that affected T commit (i.e., they changed data later read by T)
- if reading changed data is allowed only for committed transactions, we also avoid **cascade aborts** of transactions

Example: T<sub>1</sub> transfers 1000 USD from A to B,  
T<sub>2</sub> adds annual interests

T<sub>1</sub>    T<sub>2</sub>

R(A)

A := A - 1000

W(A)

committed,  
cannot be  
undone! →

cascade aborts

ABORT

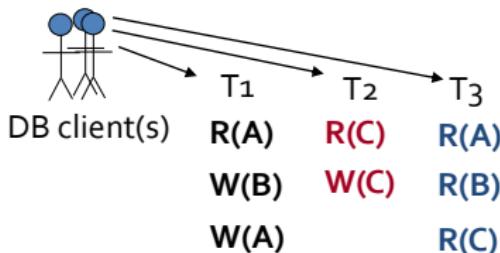
R(A)  
A := A \* 1.01

W(A)  
R(B)  
B := B \* 1.01

W(B)  
COMMIT

# Protocols for concurrent transaction scheduling

- transaction scheduler works under some **protocol** that allows to guarantee the ACID properties and maximal throughput
- pessimistic control** (highly concurrent workloads)
  - locking protocols
  - time stamps
- optimistic control** (not very concurrent workloads)
- why protocol?
  - the scheduler cannot create the entire schedule beforehand
  - scheduling is performed in local time context – dynamic transaction execution, branching parts in code



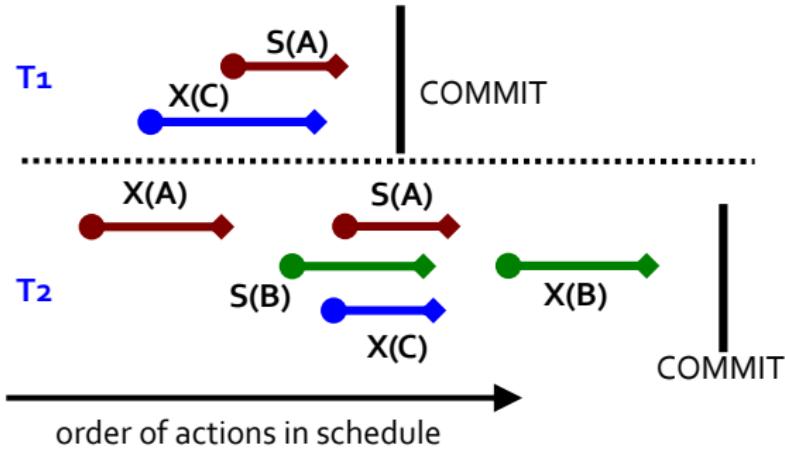
Schedule		
T1	T2	T3
R(A)	R(C)	
	R(A)	
	W(C)	
		R(B)
		W(B)
		W(A)
		R(C)

# Locking protocols

- locking of database entities can be used to control the order of reads and writes and so to secure the conflict serializability
- **exclusive locks**
  - $X(A)$  locks A so that reads and writes of A are allowed only to the lock owner/creator
  - can be granted to just one transaction
- **shared locks**
  - $S(A)$  – only reads of A are allowed
  - can be granted to (shared by) multiple transactions
- **unlocking by  $U(A)$**
- if a lock that is not available is required for a transaction, the transaction execution is suspended and waits for releasing the lock
  - in the schedule, the lock request is denoted, followed by empty rows of waiting
- the un/locking code is added by the transaction scheduler
  - i.e., operation on locks appear just in the schedules, not in the original transaction code

# Example: schedule with locking

T1	T2
$X(C)$	$X(A)$
$R(C)$	$W(A)$
$W(C)$	
$S(A)$	$U(A)$
$S(B)$	
$R(A)$	
$U(B)$	
$U(C)$	
$X(C)$	
$U(A)$	
$S(A)$	
COMMIT	
$W(C)$	
$R(A)$	
$U(B)$	
$U(C)$	
$U(A)$	
$X(B)$	
$W(B)$	
$U(B)$	
COMMIT	



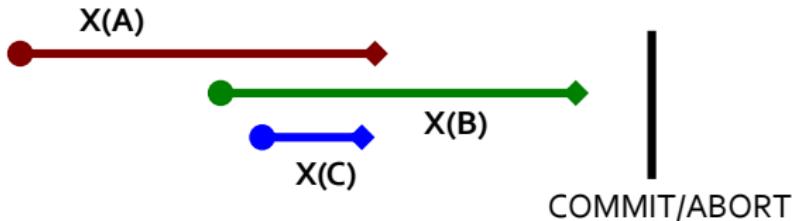
# Two-phase locking protocol (2PL)

**2PL protocol** applies two rules for building the schedule:

- 1) if a transaction wants to read (write) an entity A, it must first acquire a shared (exclusive) lock on A
- 2) transaction **cannot request a lock**, if it already released one (regardless of the locked entity)

Two obvious phases – locking and unlocking

Example: 2PL adjustment of the second transaction in the previous schedule



# Properties of 2PL

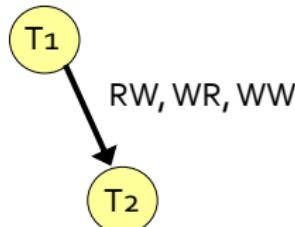
- the 2PL restriction of schedule ensures that the precedence graph is acyclic, i.e., the schedule is **conflict serializable**
- 2PL does **not guarantee recoverable schedules**

Example: 2PL-compliant schedule, but not recoverable, if T1 aborts

T1  
X(A)  
R(A)  
W(A)  
U(A)

X(A)  
R(A)  
A := A \*1.01  
W(A)  
X(B)  
U(A)  
R(B)  
B := B \*1.01  
W(B)  
U(B)  
COMMIT

ABORT / COMMIT

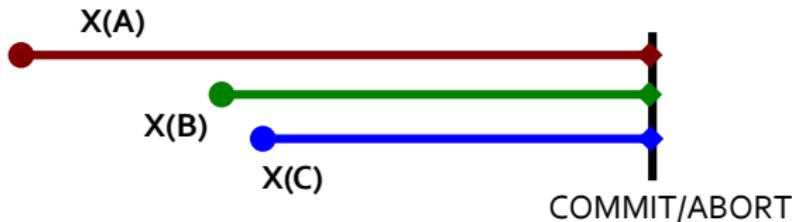


# Strict 2PL

**Strict 2PL protocol** makes the second rule of 2PL stronger, so that both rules become:

- 1) if a transaction wants to read (write) an entity A, it must first acquire a shared (exclusive) lock on A
- 2) **all locks are released at the transaction termination**

Example: strict 2PL adjustment of second transaction in the previous example



Insertions of U(A) are not needed (implicit at the time of COMMIT/ABORT).

# Properties of strict 2PL

- the 2PL restriction of schedule ensures that the precedence graph is acyclic, i.e., the schedule is **conflict serializable**
- moreover, strict 2PL ensures
  - schedule **recoverability**
  - avoids **cascade aborts**

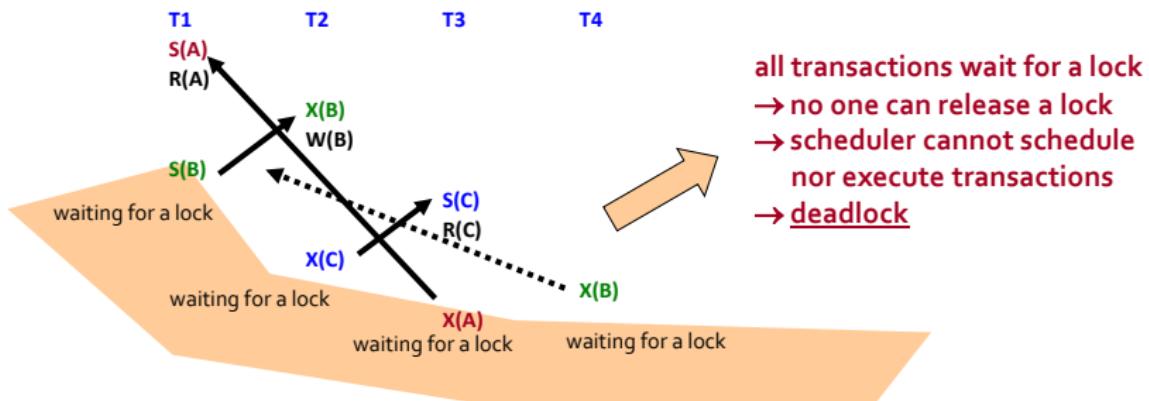
Example: schedule built using strict 2PL



# Deadlock

- during transaction execution it may happen that transaction  $T_1$  requests a lock that was already granted to  $T_2$ , but  $T_2$  cannot release it because it waits for another lock kept by  $T_1$ 
  - could be generalized to multiple transactions,  
 $T_1$  waits for  $T_2$ ,  $T_2$  waits for  $T_3$ , ...,  $T_n$  waits for  $T_1$
- strict 2PL cannot prevent from deadlock (not speaking about the weaker protocols)

Example:

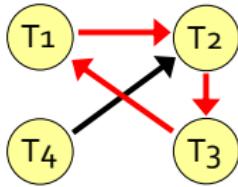


# Deadlock detection

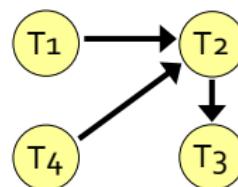
- deadlock can be detected by repeated checking the waits-for graph
- waits-for graph** is a dynamic graph that captures the waiting of transactions for locks
  - nodes are active transactions
  - an edge denotes waiting of transaction for lock kept by another transaction
  - a cycle in the graph = deadlock

Example: waits-for graph for the previous example

(a) T3 requests X(A)



(b) T3 does not request X(A)



# Deadlock resolution and prevention

- deadlocks are usually not very frequent, so the **resolution** could be simple
  - abort of the waiting transaction and its restart (user will not notice)
  - testing waits-for graph – if a deadlock occurs, abort and restart a transaction in the cycle
    - such transaction is aborted, that
      - holds the smallest number of locks
      - performed the least amount of work
      - is far from completion
    - an aborted transaction is not aborted again (if another deadlock occurs)
- deadlocks could be **prevented**
  - prioritizing
    - each transaction has a priority (e.g., time stamp); if T1 requests a lock kept by T2, the lock manager chooses between two strategies
      - **wait-die** – if T1 has higher priority, it can wait, if not, it is aborted and restarted
      - **wound-wait** – if T1 has higher priority, T2 is aborted, otherwise T1 waits

# Coffman Conditions

- Deadlocks can arise if all of the following conditions hold simultaneously in a system
  - **Mutual exclusion** – resources can be held in a non-shareable mode
  - **Resource holding (hold and wait)** – additional resources may be requested even when already some resources are held
  - **No preemption** – resources can be released only voluntarily
  - **Circular wait** – transactions can request and wait for resources in cycles
- Unfulfillment of any of these conditions is enough to prevent deadlocks from occurring

# Phantom

- now consider dynamic database
  - allowing inserts and deletes
- if one transaction works with some set of data entities, while another transaction changes this set (inserts or deletes), it could lead to inconsistent database (inserializable schedule)
  - Why? T1 locks all entities that at the given moment are relevant
    - e.g., fulfill some WHERE condition of a SELECT command
  - during execution of T1 a new transaction T2 could logically extend the set of entities
    - i.e., at that moment the number of locks defined by WHERE would be larger
    - so that some entities are locked and some are not
- applied also to strict 2PL

# Example – phantom

**T1:** find the oldest male and female employees

(**SELECT \* FROM Employees ...**) + **INSERT INTO Statistics ...**

**T2:** insert new employee Phill and delete employee Eve (employee replacement)

(**INSERT INTO Employees ...**, **DELETE FROM Employees ...**)

Initial state of the database: {[Peter, 52, m], [John, 46, m], [Eve, 55, f], [Dana, 30, f]}

**T1**

*lock men, i.e.,*

**S(Peter)**

**S(John)**

**M = max{R(Peter), R(John)}**

**T2**

**X(Eve)**

**Delete(Eve)**

**COMMIT**

**Insert(Phill, 72, m)**

**phantom**

a new male employee can be inserted, although **all men** should be locked

*lock women, i.e.,*

**S(Dana)**

**F = max{R(Dana)}**

**Insert(M, F) // result is inserted into table Statistics**

**COMMIT**

Although the schedule is **strict 2PL** compliant, the result **[Peter, Dana]** is not correct as it does not follow the serial schedule T<sub>1</sub>, T<sub>2</sub>, resulting in **[Peter, Eve]**, nor T<sub>2</sub>, T<sub>1</sub>, resulting **[Phill, Dana]**.

# Phantom – prevention

- if there do not exist indexes, everything relevant must be locked
  - e.g., entire table or even multiple tables must be locked
- if there exist indexes (e.g., B<sup>+</sup>-trees) on the entities defined by the „lock condition“, it is possible to “watch for phantom” at the index level – **index locking**
  - external attempt for the set modification is identified by the index locks updated
  - as an index usually maintains just one attribute, its applicability is limited
- generalization of index locking is **predicate locking**, when the locks are requested for the logical sets, not particular data instances
  - however, this is hard to implement and so not used much in practice

# Optimistic (not locking) protocols

- if concurrently executed transactions are not often in conflict (not competing for resources), the locking overhead is unnecessarily large
- 3-phase optimistic protocol
  1. **Read:** transaction reads data from database but writes into its private local data space
  2. **Validation:** if the transaction wants to commit, it forwards the private data space to the transaction manager (i.e., request on database update)
    - the transaction manager decides if the update is in conflict with another transaction
      - if there is a conflict, the transaction is aborted and restarted
      - if not, the last phase takes place:
  3. **Write:** the private data space is copied into the database



B0B36DBS, BD6B36DBS: **Database Systems**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/>

Lecture 11

# **Database Architectures**

Authors: **Tomáš Skopal, Irena Holubová**

Lecturer: **Martin Svoboda**, martin.svoboda@fel.cvut.cz

28. 4. 2020

**Czech Technical University in Prague, Faculty of Electrical Engineering**

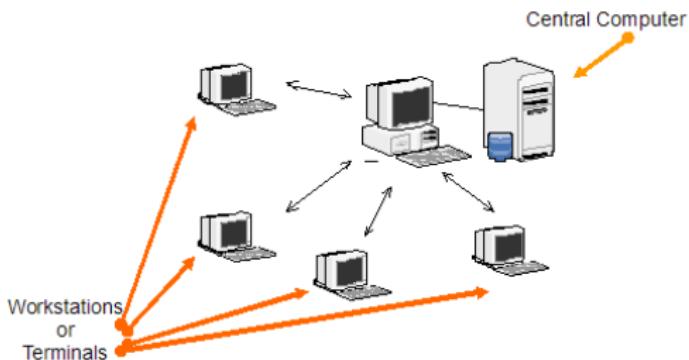
# Today's Lecture Outline

- architectures of database systems
  - centralized systems
  - client – server systems
  - parallel systems
  - distributed systems
- types of queries

# Centralized Systems

- run on a **single computer system**
- do not interact with other computer systems
- **general-purpose computer system**
  - one to a few CPUs and a number of device controllers
  - connected through a common bus
    - provides access to a shared memory
- **single-user system** (e.g., personal computer or workstation)
  - **desk-top unit**, single user, usually has one or two CPUs and one or two hard disks
  - the OS may support only one user
- **multi-user system:**
  - more disks, more memory, multiple CPUs, and a multi-user OS
  - serve a large number of users who are connected to the system via **terminals**

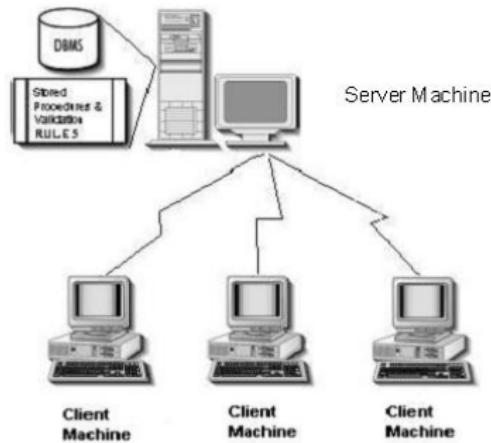
# Multi-User Systems



# Client-Server Systems

- server systems satisfy requests generated at  $m$  client systems
- advantages of replacing mainframes with networks of workstations or personal computers connected to back-end server machines:
  - better functionality for the cost
  - flexibility in locating resources and expanding facilities
  - better user interfaces
  - easier maintenance

# Client-Server Systems



# Front-End vs. Back-End

- database functionality can be divided into:
  - **back-end**: manages access structures, query evaluation and optimization, concurrency control and recovery
  - **front-end**: consists of tools such as forms, report-writers, and graphical user interface facilities
- interface between the front-end and the back-end:
  - SQL
  - application program interface

# Parallel Systems

- consist of multiple processors and multiple disks connected by a fast interconnection network
  - a **coarse-grain parallel** machine consists of a small number of powerful processors
  - a **massively parallel** or **fine-grain parallel** machine utilizes thousands of smaller processors
- two main performance measures:
  - **throughput** – the number of tasks that can be completed in a given time interval
  - **latency** (response time) – the amount of time it takes to complete a single task from the time it is submitted

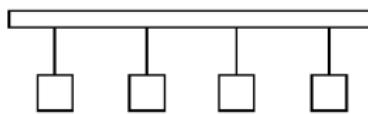
# Parallel Systems

- **speed-up**: a fixed-sized problem executing on a small system is given to a system which is **N**-times larger (more efficient)
- **scale-up**: increase the size of both the problem and the system
  - **N**-times larger system used to perform **N**-times larger job
- both often **sub-linear due to:**
  - Start-up costs: Cost of starting up multiple processes > computation time
    - If the degree of parallelism is high
  - Interference: Processes accessing **shared resources** (e.g., system bus, disks, or locks) compete with each other  $\Rightarrow$  spend time waiting on other processes rather than performing useful work
  - Skew: Increasing the degree of parallelism increases the variance in service times of tasks executed in parallel
    - Overall execution time is determined by the slowest of executing tasks

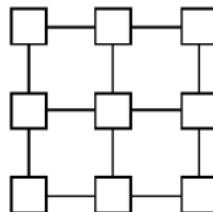
# Interconnection Architectures

- **Bus:** components send data on and receive data from a single communication bus
  - cons: does not scale well with increasing parallelism
- **Mesh:** components are arranged as nodes in a grid, and each component is connected to adjacent components
  - pros: communication links grow with growing number of components
    - scales better
  - cons: may require  $2\sqrt{n}$  hops to send message to a node
- **Hypercube:** components are numbered in binary representation  $\Rightarrow$  components are connected to one another if their binary representations differ in exactly one bit.
  - $n$  components are connected to  $\log(n)$  other components and can reach each other via at most  $\log(n)$  links
  - reduces communication delays

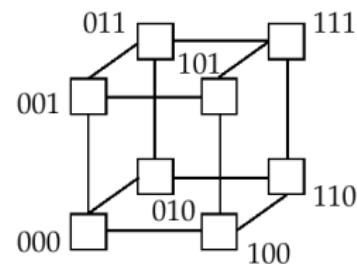
# Interconnection Architectures



(a) bus



(b) mesh

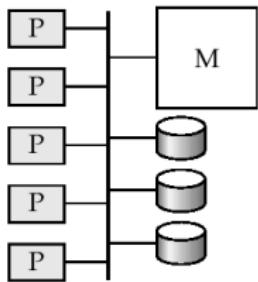


(c) hypercube

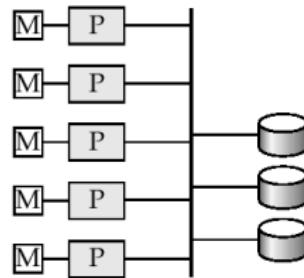
# Parallel (Database) Architectures

- **Shared memory** – processors share a common memory
  - efficient communication between processors
  - not scalable much
    - the bus or the interconnection network becomes a bottleneck
- **Shared disk** – processors share a common disk
  - a degree of fault tolerance – if a processor fails, other processors can take over its tasks
    - data are accessible from all processors
  - bottleneck = interconnection to the disk
- **Shared nothing** – processors share neither a common memory nor common disk
  - processors communicate using an interconnection network
  - drawback: cost of communication and non-local disk access
- **Hierarchical** – combination of the above architectures
  - top level is a shared-nothing
  - each node of the system could be a shared-memory sub-system

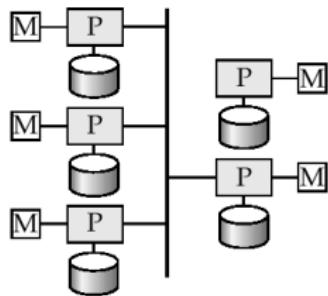
# Parallel Database Architectures



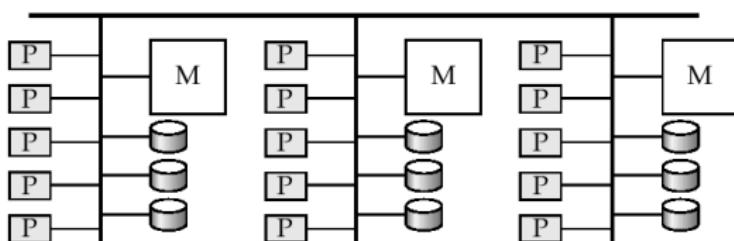
(a) shared memory



(b) shared disk



(c) shared nothing



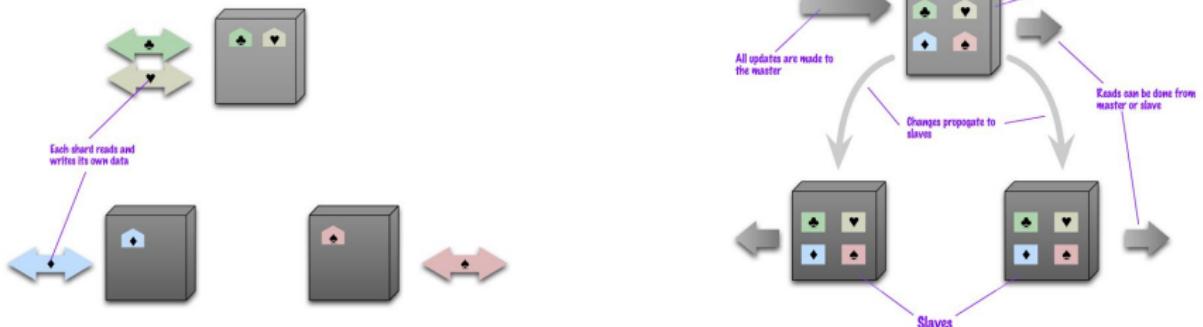
(d) hierarchical

# Distributed Systems

- **scale-out:** data are **distributed** (spread) over multiple machines = nodes
- data are **replicated**
  - system can work even if a node fails
- **homogeneous** distributed databases
  - same software/schema on all nodes, data may be partitioned among nodes
  - goal: provide a view of a single database, hiding details of distribution
- **heterogeneous** distributed databases
  - different software/schema on different nodes
  - goal: integrate existing databases to provide useful functionality

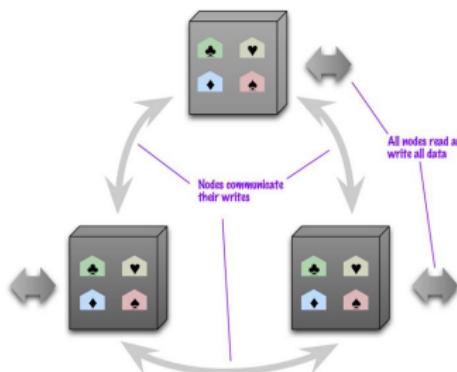
# Distribution Models

- **single server** – no distribution
- **sharding** – putting different parts of the data onto different servers
  - too many data to be stored on a single node
- **master/slave replication** – master provides reads/writes, slaves provide reads
  - no scalability of writes
- **peer-to-peer replication** – all replicas have equivalent weight
  - each node is a master
- often: **combination of sharding and replication**



sharding = **distribution**

master/slave **replication**



peer-to-peer **replication**

# Types of Queries

- **declarative**
  - we describe the **data we want**, but not how to get it
  - e.g., DRC, TRC
- **procedural**
  - we describe **how to get the data** we want
    - i.e., **what operations should be done**
  - e.g., relational algebra (partially)
- SQL has both the features
- **QBE** (Query by Example)
  - graphical query language from mid 70-ies (IBM)
    - developed as an alternative to SQL
  - many graphical front-ends for databases re-use the idea today

# QBE

Sailors (sid: integer, sname: string, rating: integer, age: real)  
 Boats (bid: integer, bname: string, color: string)  
 Reserves (sid: integer, bid: integer, day: dates)

<i>Sailors</i>	<i>sid</i>	<i>sname</i>	<i>rating</i>	<i>age</i>
P.			10	

Sailors with rating 10

<i>Sailors</i>	<i>sid</i>	<i>sname</i>	<i>rating</i>	<i>age</i>
		P._N		P._A

Names and ages of all sailors

<i>Sailors</i>	<i>sid</i>	<i>sname</i>	<i>rating</i>	<i>age</i>	<i>Reserves</i>	<i>sid</i>	<i>bid</i>	<i>day</i>
	_Id	P._S		> 25			_Id	'8/24/96'

Sailors who have reserved a boat for 8/24/96 and who are older than 25

<i>Sailors</i>	<i>sid</i>	<i>sname</i>	<i>rating</i>	<i>age</i>
	_Id			> 25

<i>Reserves</i>	<i>sid</i>	<i>bid</i>	<i>day</i>	<i>Boats</i>	<i>bid</i>	<i>bname</i>	<i>color</i>
	_Id	_B	'8/24/96'		_B	Interlake	P.

Colors of boats Interlake reserved by sailors who have reserved a boat for 8/24/96 and who are older than 25



**B0B36DBS, BD6B36DBS: Database Systems**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/192-B0B36DBS/>

Lecture 12

# **Modern Trends**

**Martin Svoboda**

[martin.svoboda@fel.cvut.cz](mailto:martin.svoboda@fel.cvut.cz)

12. 5. 2020

**Czech Technical University in Prague, Faculty of Electrical Engineering**

# Lecture Outline

## Big Data

- Characteristics
- Current trends

## NoSQL databases

- Motivation
- Features

Overview of NoSQL database types

- **Key-value, wide column, document, graph, ...**

# Relational Databases

Data model

Instance → **database** → **table** → **row**

Query languages

- Real-world: **SQL** (*Structured Query Language*)
- Formal: **Relational algebra**, relational calculi (domain, tuple)

Query patterns

- **Selection** based on complex conditions, **projection**, **joins**, **aggregation**, derivation of new values, recursive queries, ...

Representatives

- Oracle Database, Microsoft SQL Server, IBM DB2
- MySQL, PostgreSQL

# Relational Databases

Representatives



# Relational Databases

## Features: Normal Forms

### Model

- Functional dependencies
- **1NF, 2NF, 3NF, BCNF** (Boyce-Codd normal form)

### Objective

- **Normalization of database schema** to BCNF or 3NF
- Algorithms: decomposition or synthesis

### Motivation

- Diminish **data redundancy**, prevent update anomalies
- However:
  - Data is scattered into small pieces (high granularity), and so
  - these pieces have to be joined back together when querying!

# Relational Databases

## Features: Transactions

### Model

- **Transaction** = flat sequence of database operations  
(READ, WRITE, COMMIT, ABORT)

### Objectives

- Enforcement of ACID properties
- **Efficient parallel / concurrent execution** (slow hard drives, ...)

### ACID properties

- Atomicity – partial execution is not allowed (all or nothing)
- Consistency – transactions turn one valid database state into another
- Isolation – uncommitted effects are concealed among transactions
- Durability – effects of committed transactions are permanent

# What is Big Data?

Buzzword? Bubble? Gold rush? Revolution?



Dan Ariely:

**Big Data** is like teenage sex: **everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it.**

# What is Big Data?

No standard definition

- Gartner (*research and advisory company*):  
**High Performance Computing**

**Big Data** is **high volume**, **high velocity**, and/or **high variety** information assets that require **new forms of processing** to enable enhanced decision making, insight discovery and process optimization.

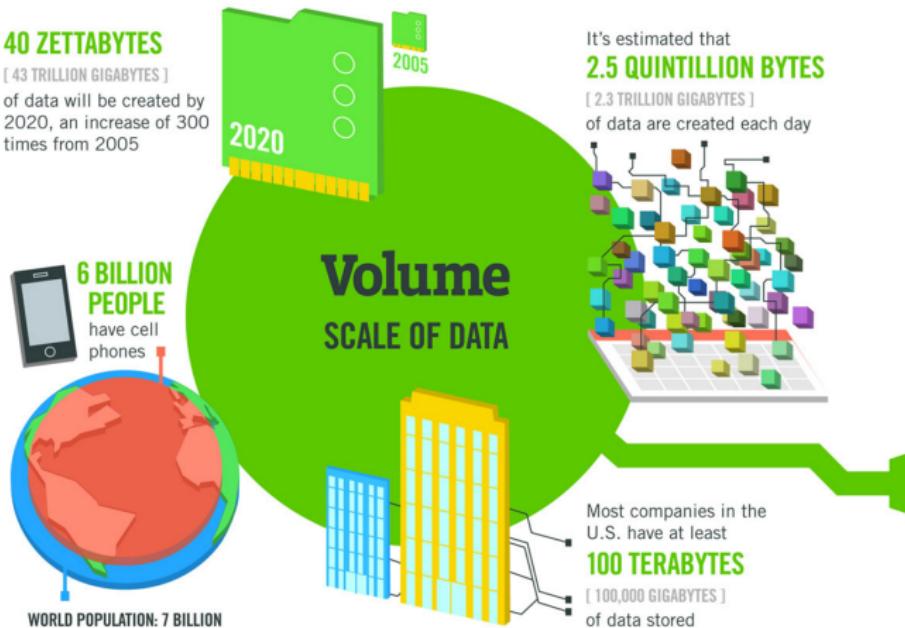
# Where is Big Data?

## Sources of Big Data

- **Social media and networks**
  - ...all of us are generating data
- **Scientific instruments**
  - ...collecting all sorts of data
- **Mobile devices**
  - ...tracking all objects all the time
- **Sensor technology and networks**
  - ...measuring all kinds of data

# Big Data Characteristics

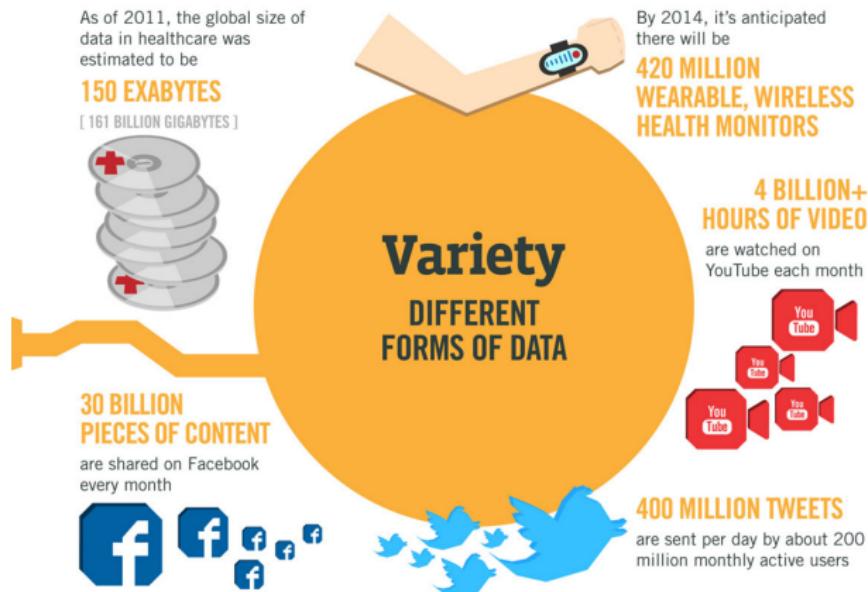
## Volume (Scale)



Source: <http://www.ibmbigdatahub.com/>

# Big Data Characteristics

## Variety (Complexity)



Source: <http://www.ibmbigdatahub.com/>

# Big Data Characteristics

## Velocity (Speed)

The New York Stock Exchange captures  
**1 TB OF TRADE INFORMATION** during each trading session



By 2016, it is projected there will be

**18.9 BILLION NETWORK CONNECTIONS**

– almost 2.5 connections per person on earth



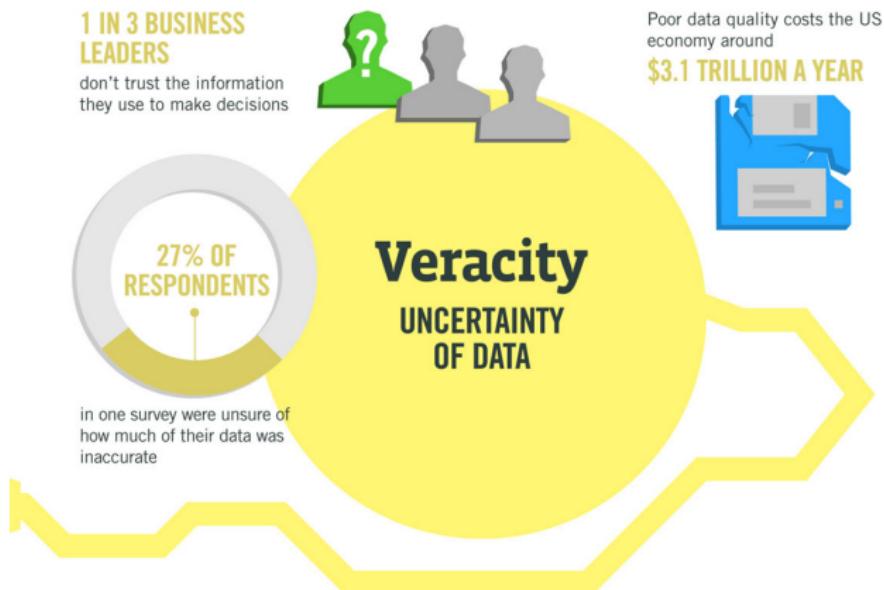
Modern cars have close to  
**100 SENSORS** that monitor items such as fuel level and tire pressure

**Velocity**  
ANALYSIS OF STREAMING DATA



# Big Data Characteristics

## Veracity (Uncertainty)



Source: <http://www.ibmbigdatahub.com/>

# Big Data Characteristics

## Basic 4V

- Volume (Scale)

- Data volume is increasing exponentially, not linearly
- Even large amounts of small data can result into Big Data

- Variety (Complexity)

- Various formats, types, and structures  
(from semi-structured XML to unstructured multimedia)

- Velocity (Speed)

- Data is being generated fast and needs to be processed fast

- Veracity (Uncertainty)

- Uncertainty due to inconsistency, incompleteness, latency, ambiguities, or approximations

# Current Trends

## Big Data

- **Volume:** terabytes → zettabytes
- **Variety:** structured → structured and unstructured data
- **Velocity:** batch processing → streaming data
- ...

## Big users

- Population online, hours spent online, devices online, ...
- Rapidly growing companies / web applications
  - Even millions of users within a few months

# Current Trends

Everything is in **cloud**

- **SaaS**: Software as a Service
- **PaaS**: Platform as a Service
- **IaaS**: Infrastructure as a Service

Processing paradigms

- **OLTP**: Online Transaction Processing
- **OLAP**: Online Analytical Processing
- *...but also...*
- **RTAP**: Real-Time Analytical Processing

# Current Trends

## Data assumptions

- **Data format** is becoming unknown or inconsistent
- Linear growth → **unpredictable exponential growth**
- **Read requests** often prevail **write requests**
- Data updates are no longer frequent
- Data is expected to be replaced
- Strong **consistency** is no longer mission-critical

# Current Trends

⇒ New approach is required

- Relational databases simply do not follow the current trends

Key technologies

- Distributed **file systems**
- **MapReduce** and other programming models
- Grid computing, cloud computing
- **NoSQL databases**
- Data warehouses
- Large scale machine learning

# NoSQL Databases

What does **NoSQL** actually mean?

A bit of history ...

- 1998
  - First used for a relational database that omitted usage of SQL
- 2009
  - First used during a conference to advocate non-relational databases

So?

- Not: *no to SQL*
- Not: *not only SQL*
- NoSQL is an **accidental term with no precise definition**

# NoSQL Databases

What does **NoSQL** actually mean?

**NoSQL movement** = The whole point of **seeking alternatives** is that you need to solve a problem that **relational databases are a bad fit for**

**NoSQL databases** = Next generation databases mostly addressing some of the points: being **non-relational, distributed, open-source and horizontally scalable**. The original intention has been modern web-scale databases. Often more characteristics apply as: **schema-free, easy replication support, simple API, eventually consistent, a huge data amount**, and more.

# Types of NoSQL Databases

## Core types

- **Key-value stores**
- **Wide column** (column family, column oriented, ...) stores
- **Document** stores
- **Graph** databases

## Non-core types

- **Object** databases
- Native **XML** databases
- **RDF** stores
- ...

# Key-Value Stores

## Data model

- The most simple NoSQL database type
  - Works as a simple hash table (mapping)
- Key-value pairs
  - Key (id, identifier, primary key)
  - Value: binary object, black box for the database system

## Query patterns

- Create, update or remove value for a given key
- Get value for a given key

## Characteristics

- Simple model ⇒ great performance, easily scaled, ...
- Simple model ⇒ not for complex queries nor complex data

# Key-Value Stores

## Suitable use cases

- Session data, user profiles, user preferences, shopping carts, ...
  - I.e. when values are only accessed via keys

## When not to use

- Relationships among entities
- Queries requiring access to the content of the value part
- Set operations involving multiple key-value pairs

## Representatives

- **Redis, MemcachedDB, Riak KV, Hazelcast, Ehcache, Amazon SimpleDB, Berkeley DB, Oracle NoSQL, Infinispan, LevelDB, Ignite, Project Voldemort**
- *Multi-model:* OrientDB, ArangoDB

# Key-Value Stores

Representatives



redis



# Wide Column Stores

## Data model

- **Column family (table)**
  - Table is a collection of **similar rows** (not necessarily identical)
- **Row**
  - Row is a collection of **columns**
    - Should encompass a group of data that is accessed together
  - Associated with a unique **row key**
- **Column**
  - Column consists of a **column name** and **column value** (and possibly other metadata records)
  - Scalar values, but also **flat sets, lists or maps** may be allowed

# Wide Column Stores

Sample data: table of movies in **Apache Cassandra**

id				
samotari	title	year	actors	genres
	Samotáři	2000	null	[ comedy, drama ]
medvidek	title	director	year	
	Medvídek	( Jan, Hřebejk )	2007	
	actors			
	{ trojan: Ivan, machacek: Jirka }			
vratnelahve	title	year	actors	
	Vratné lahve	2006	{ machacek: Robert Landa }	
zelary	title	year	actors	genres
	Želary	2003	{}	[ romance, drama ]

# Wide Column Stores

## Query patterns

- Create, update or remove a row within a given column family
- **Select rows according to a row key or simple conditions**

## Warning

- Wide column stores are not just a special kind of RDBMSs with a variable set of columns!

# Wide Column Stores

## Suitable use cases

- Event logging, content management systems, blogs, ...
  - I.e. for structured flat data with similar schema

## When not to use

- ACID transactions are required
- Complex queries: aggregation (SUM, AVG, ...), joining, ...
- Early prototypes: i.e. when database design may change

## Representatives

- Apache **Cassandra**, Apache **HBase**, Apache Accumulo, Hypertable, **Google Bigtable**

# Wide Column Stores

Representatives



HYPERTABLE<sup>INC</sup>



# Document Stores

## Data model

- **Documents**
  - Self-describing
  - **Hierarchical tree structures** (JSON, XML, ...)
    - Scalar values, maps, lists, sets, nested documents, ...
  - Identified by a **unique identifier** (key, ...)
- Documents are **organized into collections**

## Query patterns

- Create, update or remove a document
- **Retrieve documents according to complex query conditions**

## Observation

- Extended key-value stores where the value part is examinable!

# Document Stores

Sample data: collection of movies in **MongoDB**

```
{  
  _id: ObjectId("1"),  
  title: "Vratné lahve", year: 2006,  
  actors: [ "Zdeněk Svěrák", "Jiří Macháček" ]  
}
```

```
{  
  _id: ObjectId("2"),  
  title: "Samotáři", year: 2000,  
  actors: [ "Jitka Schneiderová", "Ivan Trojan", "Jiří Macháček" ]  
}
```

```
{  
  _id: ObjectId("3"),  
  title: "Medvídek", year: 2007,  
  actors: [ "Jiří Macháček", "Ivan Trojan" ]  
}
```

# Document Stores

## Suitable use cases

- Event logging, content management systems, blogs, web analytics, e-commerce applications, ...
  - I.e. for structured documents with similar schema

## When not to use

- Set operations involving multiple documents
- Design of document structure is constantly changing
  - I.e. when the required level of granularity would outbalance the advantages of aggregates

# Document Stores

## Representatives

- MongoDB, **Couchbase**, Amazon **DynamoDB**, **CouchDB**, RethinkDB, RavenDB, Terrastore
- *Multi-model*: **MarkLogic**, **OrientDB**, OpenLink Virtuoso, ArangoDB

# Document Stores

Representatives



# Graph Databases

## Data model

- Property graphs
  - Directed / undirected graphs, i.e. collections of ...
    - nodes (vertices) for real-world entities, and
    - relationships (edges) between these nodes
  - Both the nodes and relationships can be associated with additional properties

## Types of databases

- Non-transactional = small number of very large graphs
- Transactional = large number of small graphs

# Graph Databases

## Query patterns

- Create, update or remove a node / relationship in a graph
- **Graph algorithms** (shortest paths, spanning trees, ...)
- **General graph traversals**
- **Sub-graph queries or super-graph queries**
- Similarity based queries (approximate matching)

## Representatives

- **Neo4j**, **Titan**, Apache Giraph, InfiniteGraph, FlockDB
- *Multi-model*: **OrientDB**, OpenLink **Virtuoso**, **ArangoDB**

# Graph Databases

## Suitable use cases

- Social networks, routing, dispatch, and location-based services, recommendation engines, chemical compounds, biological pathways, linguistic trees, ...
  - I.e. simply **for graph structures**

## When not to use

- **Extensive batch operations** are required
  - Multiple nodes / relationships are to be affected
- **Only too large graphs** to be stored
  - Graph distribution is difficult or impossible at all

# Graph Databases

Representatives



# Native XML Databases

## Data model

- **XML documents**
  - Tree structure with nested **elements**, **attributes**, and text values (beside other less important constructs)
  - Documents are organized into collections

## Query languages

- **XPath**: *XML Path Language* (navigation)
- **XQuery**: *XML Query Language* (querying)
- **XSLT**: *XSL Transformations* (transformation)

## Representatives

- **Sedna**, **Tamino**, BaseX, eXist-db
- *Multi-model*: **MarkLogic**, OpenLink **Virtuoso**

# Native XML Databases

Sample data: XML document with movies

```
<?xml version="1.1" encoding="UTF-8"?>
<movies>
    <movie year="2006" rating="76" director="Jan Svěrák">
        <title>Vratné lahve</title>
        <actor>Zdeněk Svěrák</actor>
        <actor>Jiří Macháček</actor>
    </movie>
    <movie year="2000" rating="84">
        <title>Samotáři</title>
        <actor>Jitka Schneiderová</actor>
        <actor>Ivan Trojan</actor>
        <actor>Jiří Macháček</actor>
    </movie>
    <movie year="2007" rating="53" director="Jan Hřebejk">
        <title>Medvídek</title>
        <actor>Jiří Macháček</actor>
        <actor>Ivan Trojan</actor>
    </movie>
</movies>
```

# Native XML Databases

Representatives



# RDF Stores

## Data model

- **RDF triples**
  - Components: **subject**, **predicate**, and **object**
  - Each triple represents a **statement** about a real-world entity
- Triples can be viewed as **graphs**
  - **Vertices** for subjects and objects
  - **Edges** directly correspond to individual statements

## Query language

- **SPARQL**: *SPARQL Protocol and RDF Query Language*

## Representatives

- Apache **Jena**, **rdf4j** (Sesame), Algebraix
- *Multi-model*: **MarkLogic**, OpenLink **Virtuoso**

# RDF Stores

## Sample data: RDF graph of movies

```
@prefix i: <http://db.cz/terms#> .  
@prefix m: <http://db.cz/movies/> .  
@prefix a: <http://db.cz/actors/> .  
  
m:vratnelahve  
    rdf:type i:Movie ; i:title "Vratné lahve" ;  
    i:year 2006 ;  
    i:actor a:sverak , a:machacek .  
  
m:samotari  
    rdf:type i:Movie ; i:title "Samotáři" ;  
    i:year 2000 ;  
    i:actor a:schneiderova , a:trojan , a:machacek .  
  
m:medvidek  
    rdf:type i:Movie ; i:title "Medvídek" ;  
    i:year 2007 ;  
    i:actor a:machacek , a:trojan ;  
    i:director "Jan Hřebejk" .
```

# RDF Stores

Representatives



# Features of NoSQL Databases

## Data model

- Traditional approach: relational model
- (New) possibilities:
  - **Key-value, document, wide column, graph**
  - Object, XML, RDF, ...
- Goal
  - Respect the real-world nature of data  
(i.e. data structure and mutual relationships)

# Features of NoSQL Databases

## Aggregate structure

- Aggregate definition
  - Data unit with a complex structure
  - Collection of related data pieces we wish to treat as a unit  
(with respect to data manipulation and data consistency)
- Examples
  - Value part of key-value pairs in key-value stores
  - Document in document stores
  - Row of a column family in wide column stores

# Features of NoSQL Databases

## Aggregate structure

- Types of systems
  - **Aggregate-ignorant**: relational, graph
    - It is not a bad thing, it is a feature
  - **Aggregate-oriented**: key-value, document, wide column
- Design notes
  - No universal strategy how to draw **aggregate boundaries**
  - **Atomicity** of database operations:  
just a single aggregate at a time

# Features of NoSQL Databases

## Elastic scaling

- Traditional approach: **scaling-up**
  - Buying bigger servers as database load increases
- New approach: **scaling-out**
  - Distributing database data across multiple hosts
    - Graph databases (unfortunately): difficult or impossible at all

## Data distribution

- Sharding
  - Particular ways how database data is split into separate groups
- Replication
  - Maintaining several data copies (performance, recovery)

# Features of NoSQL Databases

## Automated processes

- Traditional approach
  - Expensive and highly trained database administrators
- New approach: **automatic recovery, distribution, tuning, ...**

## Relaxed consistency

- Traditional approach
  - **Strong consistency** (**ACID** properties and transactions)
- New approach
  - **Eventual consistency** only (**BASE** properties)
  - I.e. we have to make trade-offs because of the data distribution

# Features of NoSQL Databases

## Schemalessness

- Relational databases
  - Database schema present and **strictly enforced**
- NoSQL databases
  - **Relaxed schema or completely missing**
  - Consequences: **higher flexibility**
    - Dealing with **non-uniform data**
    - **Structural changes** cause no overhead
  - However: there is (usually) an **implicit schema**
    - We must know the data structure at the application level anyway

# Features of NoSQL Databases

## Open source

- Often community and enterprise versions (with extended features or extent of support)

## Simple APIs

- Often state-less application interfaces (HTTP)

# Conclusion

## The end of relational databases?

- Certainly no
  - They are still suitable for most projects
  - Familiarity, stability, feature set, available support, ...
- However, we should also consider different database models and systems
  - **Polyglot persistence** = **usage of different data stores in different circumstances**

# Lecture Conclusion

## Big Data

- 4V characteristics: volume, variety, velocity, veracity

## NoSQL databases

- (New) logical models
  - Core: key-value, wide column, document, graph
  - Non-core: XML, RDF, ...
- (New) principles and features
  - Horizontal scaling, data sharding and replication, eventual consistency, ...