

course:

Database Systems (NDBlo25)

SS2017/18

lecture 1:

Conceptual database modeling

doc. RNDr. Tomáš Skopal, Ph.D.

Mgr. Martin Nečaský, Ph.D.

RNDr. Michal Kopecký, Ph.D.

Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague

Organizational stuff

- student duties
 - credit test ($\geq 60\%$ points)
 - exam test
 - attendance strongly recommended (but not mandatory)
 - the slides alone are not comprehensive
 - other sources
 - textbook:
Ramakrishnan, Gehrke: Database Systems Management,
McGraw-Hill, 2003 (available in faculty library)
 - software:
see references in particular lectures and also the web site below
- web: <http://www.ms.mff.cuni.cz/~kopecky/vyuka/ndbio25/>

What is the course (not) about?

- it is about
 - conceptual data modeling
 - relational model
 - physical implementation of database management
 - transactional processing
 - introduction to database applications, multimedia and XML databases
- it is NOT about
 - data mining
 - text databases
 - data warehousing, OLAP
 - cloud computing

Follow-up courses

- other topics are subject to follow-up courses
 - Database languages I, II
 - Datalog
 - Database applications
 - Oracle and MS SQL Server administration
 - Transactions
 - Stochastic methods in databases
 - Searching the web and multimedia databases
 - Retrieval of multimedia content on the web
 - XML technology
 - NoSQL databases

Today's lecture outline

- what is a database?
- conceptual modeling
 - ER
 - UML
 - and beyond (OCL)
 - See e.g.
 - Cabot, Jordi; Gogolla, Martin: Object Constraint Language (OCL): a Definitive Guide ([https://modeling-languages.com/
wp-content/uploads/2012/03/OCLEChapter.pdf](https://modeling-languages.com/wp-content/uploads/2012/03/OCLEChapter.pdf))
 - <https://www.slideshare.net/jcabot/ocl-tutorial>

Database management system

- database
 - logically ordered collection of related data instances
 - self-describing, meta-data stored together with data
- database management system
 - general software system for shared access to database
 - provides mechanisms to ensure security, reliability and integrity of the stored data
- database administrator
 - the necessary human factor

Why database systems?

- means of data sharing and reusability
- unified interface and languages for data definition and data manipulation
- data consistency and correctness
- redundancy elimination (compact storage)

Basic Terminology

Model vs. Schema vs. Diagram

- **model** = modeling language
 - set of constructs you can use to express something
 - e.g., UML model = {class, attribute, association}
 - e.g., relational model = {relation schema, attribute}
- **schema** = modeling language expression
 - instance of a model
 - e.g., relational schema = {Person(name, email)}
- **diagram** = schema visualization

Basic Terminology

Stakeholder

- stakeholder is any person which is relevant for your application(s).
 - e.g., application user, investor, owner, domain expert, etc.
- you have to communicate with all stakeholders and balance their requirements when developing a (database) application.

Basic Terminology

Three layers of database modeling

- conceptual layer
 - models a part of the 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 database structures
 - different logical models (e.g. relational, object-relational, XML, graph, multimedia, etc.)
- physical model
 - specifies how logical database structures are implemented in a specific technical environment
 - data files, index structures (e.g. B+ trees), etc.

Conceptual Modeling Process

- process of creating a conceptual schema of an application (or applications) in a selected conceptual model on the base of given requirements of various stakeholders
- in fact you do not create only one conceptual schema but multiple
 - each schema describes the application(s) from a different point of view
 - there is a different conceptual model suitable for each viewpoint
- two basic viewpoints
 - **conceptual data viewpoint** (this lecture)
 - conceptual functional viewpoint (different courses, e.g., NSWIo41)

Conceptual Modeling Process

Requirements analysis

- identify types of entities
- identify types of relationships
- identify characteristics

Model identified types

- choose modeling language
- create conceptual schema

Iteratively adapt your schemas to requirements changing over time

Conceptual Data Modeling Process

STEP 1: Requirements Analysis

- start with requirements of different stakeholders
 - usually expressed in a natural language
 - informal discussions, inquiries
- identify important types of real-world entities, their characteristics, and types of relationships between them
 - ambiguous process
(because of informal requirements)

Conceptual Data Modeling Process

STEP 1.1: Identify types of entities

"Our social network consists of persons which may have other persons as their colleagues. A person can also be a member of several research teams. And, she can work on various research projects. A team consists of persons which cooperate together. 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 types of entities
 - **Person**
 - **Research Team**
 - **Research Project**
 - **Academic Professor**
 - **Assistant Professor**
 - **Associate Professor**
 - **Full Professor**
 - **Official Institution**
 - **University Department**

Conceptual Data Modeling Process

STEP 1.2: Identify types of relationships

“Our social network consists of *persons* which may have other persons as their *colleagues*. A *person can* also be a member of several *research teams*. And, she (*person*) can work on various research *projects*. A *team* consists of persons which cooperate together. 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.”

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

Conceptual Data Modeling Process

STEP 1.3: Identify characteristics of types

“Each person has a **name** and is identified by its **personal number**. A person can be called to its registered **phone numbers**. We need to know at least one phone number. We also need to send her **emails**.”

- identified characteristics of type **Person**:
 - **personal number**
 - **name**
 - *one or more phone numbers*
 - **email**

Conceptual Data Modeling Process

STEP 1.3: Identify characteristics of types

“We need to know
when a person
became a member of
a project and **when**
she **finished** her
membership.”

- identified characteristics of type **is member of**:
 - from
 - to

Conceptual Data Modeling Process

STEP 2: Model Identified Types

- model your types using a suitable conceptual data model (i.e., create conceptual data schema) and visualize it as a diagram
- you can use various tools for modeling, so-called **Case Tools**, e.g.,
 - commercial Enterprise Architect, IBM Rational Rose
 - academic eXolutio

Conceptual Data Modeling Process

STEP 2.1: Choose suitable modeling language

- there are various languages for modeling conceptual data schemas
 - each is associated with a well-established visualization in diagrams
 - in this lecture, you will see the model UML class diagrams (shortly **UML**) and Entity-Relationship model (shortly **ER**)
 - there are also others (out of scope of this lecture)
 - Object Constraints Language (OCL)
 - Object-Role Model (ORM)
 - Web Ontology Language (OWL)
 - Predicate Logic
 - Description Logic
- 
- used in practice
- 
- rarely used in practice,
used to define formal background
and properties of modeling languages

Conceptual Data Modeling Process

STEP 2.2: Create conceptual schema

- express your identified types of entities, relationships and their characteristics with constructs offered by the selected conceptual modeling language
- **UML**: classes, associations, attributes
- **ER**: entity types, relationship types, attributes
 - ER is more oriented to **data design**
 - UML is more oriented to **code design**
 - but is suitable for data design as well (wide scope language)
 - both used in practice,
UML has became more popular in recent years

Conceptual Data Modeling Process

ER vs. UML

- ER model
 - not standardized, various notations, various extensions (e.g., IS-A hierarchy)
- UML
 - family of models, e.g., **class diagrams**, use case diagrams, state diagrams,
 - standardized by OMG (object management group), current version UML 2.5 (Jul 2015)

UML and ER Basic Constructs

How to model types of entities?

UML	ER	Real-world
<p>The UML section shows three classes: 'Team' (top), 'Person' (middle), and 'Project' (bottom). Each class is represented by a rectangle with a double-lined border. The 'Team' class contains the word 'Team' inside. The 'Person' class contains the word 'Person' inside. The 'Project' class contains the word 'Project' inside.</p>	<p>The ER section shows three entity types: 'Team' (top), 'Person' (middle), and 'Project' (bottom). Each entity type is represented by a rectangle with a single-lined border. The 'Team' entity type contains the word 'Team' inside. The 'Person' entity type contains the word 'Person' inside. The 'Project' entity type contains the word 'Project' inside.</p>	<p>Real-world persons, research teams and research projects.</p>
<p>UML construct: Class</p> <ul style="list-style-type: none">• name	<p>ER construct: Entity type</p> <ul style="list-style-type: none">• name	<p>Type of real-world entities</p>

UML and ER Basic Constructs

How to model characteristics of entities?

UML	ER	Real-world
<div style="border: 1px solid black; padding: 10px; width: fit-content; margin: auto;"><p>Person</p><ul style="list-style-type: none">- personNumber- name- email [0..1]- phone [1..n]</div>	<pre>classDiagram class Person { personNumber name email phone } Person "0..1" --> "1..n" Person : name Person "0..1" --> "1..n" Person : email Person "1..n" --> "1..n" Person : phone</pre>	A person is characterized by its personal number, name, optional email and one or more phone numbers.
<p>UML construct: Attribute of class</p> <ul style="list-style-type: none">• name• cardinality	<p>ER construct: Attribute of entity type</p> <ul style="list-style-type: none">• name• cardinality	Characteristics of a type of real-world entities

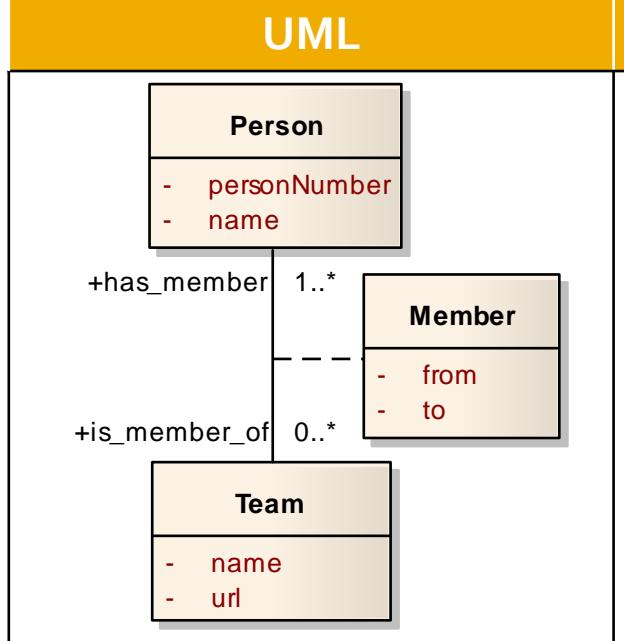
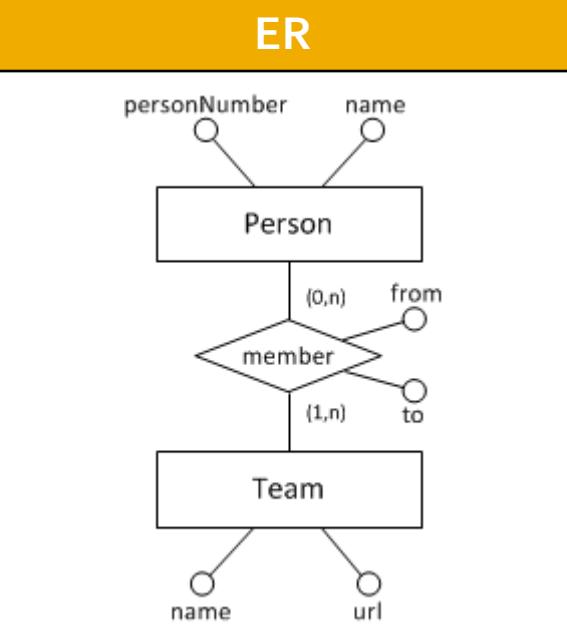
UML and ER Basic Constructs

How to model types of relationships?

UML	ER	Real-world
<p>The UML diagram shows two classes: Person and Team. The Person class has attributes: - personNumber and - name. It has three associations: +has_leader with Team (cardinality 1), +has_member with Team (cardinality 1..*), and +is_member_of with Team (cardinality 0..*). The Team class has attributes: - name and - url. It has one association: +leads with Person (cardinality 0..*).</p> <pre>classDiagram class Person { -personNumber -name } class Team { -name -url } Person "1" *-- "1..*" Team : +has_member Person "1" *-- "0..*" Team : +is_member_of Team "0..*" *-- "1" Person : +leads</pre>	<p>The ER diagram shows two entities: Person and Team. Person has attributes: personNumber and name. Team has attributes: name and url. There are two binary relationship types: member and leader. The member relationship connects Person and Team with cardinalities (0,n) at Person and (1,n) at Team. The leader relationship connects Team and Person with cardinalities (0,n) at Team and (1,1) at Person.</p> <pre>erDiagram Person --o Team : member Person --o Team : leader Person { string personNumber string name } Team { string name string url }</pre>	<p>A team has one or more members. A person can be a member of zero or more teams.</p> <p>A team has exactly one leader. A person can be a leader of zero or more teams.</p>
<p>UML construct: Binary association</p> <ul style="list-style-type: none">optional nametwo participants with optional name and cardinality	<p>ER construct: Binary relationship type</p> <ul style="list-style-type: none">nametwo participants with cardinality	<p>A type of relationship between two real-world entities.</p>

UML and ER Basic Constructs

How to model characteristics of relationships?

UML	ER	Real-world
 <pre>classDiagram class Person { -personNumber -name } class Member { -from -to } class Team { -name -url } Person "1..*" --> "0..*" Member : +has_member Person "0..*" --> "1..*" Team : +is_member_of Member < --> Team : member</pre>	 <pre>erDiagram Person --o Team : member Person { string personNumber string name } Team { string name string url } relationship { string from string to }</pre>	A person is a team member in a given time period.
<p>UML construct: Attribute of binary association class</p> <ul style="list-style-type: none">combination of class and binary association	<p>ER construct: Attribute of relationship type</p> <ul style="list-style-type: none">namecardinality	Characteristics of a type or relationships between two real-world entities.

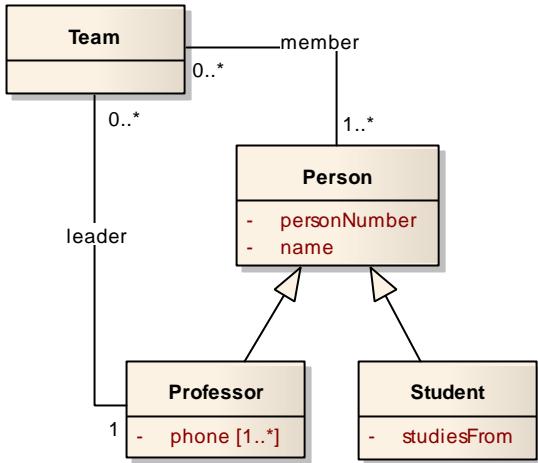
UML and ER Basic Constructs

How to model generalizations/specializations?

UML	ER	Real-world
<p>The UML diagram illustrates a generalization relationship. A 'Team' class has two associations: one to 'Person' with multiplicity 0..* at Team and 1..* at Person, labeled 'member'; and another to 'Professor' with multiplicity 0..* at Team and 1 at Professor, labeled 'leader'. The 'Person' class contains attributes 'personNumber' and 'name'. The 'Professor' class contains attribute 'phone [1..*]' and the 'Student' class contains attribute 'studiesFrom'.</p>	<p>The ER diagram shows the same entities and relationships as the UML diagram. It includes a 'Team' entity connected to 'Person' via a 'member' relationship (multiplicity 1..n at Team, 0..n at Person) and to 'Professor' via a 'leader' relationship (multiplicity 1..1 at Team, 0..n at Professor). The 'Person' entity has attributes 'personNumber' and 'name'. The 'Professor' entity has attribute 'phone' (multiplicity 1..n) and the 'Student' entity has attribute 'studiesFrom'.</p>	<p>A professor is a person which, in addition to a number and name, has one or more phones and can lead teams.</p> <p>A student is a person which, in addition to a number and name, has a data from which (s)he studies.</p>
<p>Construct: Specific kind of binary association called generalization</p> <ul style="list-style-type: none">no name, roles and cardinalities	<p>Construct: IS-A relationship</p> <ul style="list-style-type: none">no name and cardinalities	<p>A type of real-world entities which is a specialization of another type.</p>

UML and ER Basic Constructs

How to model generalizations/specializations?



Additional constraints

(independently of the modeling language used):

Covering Constraint:

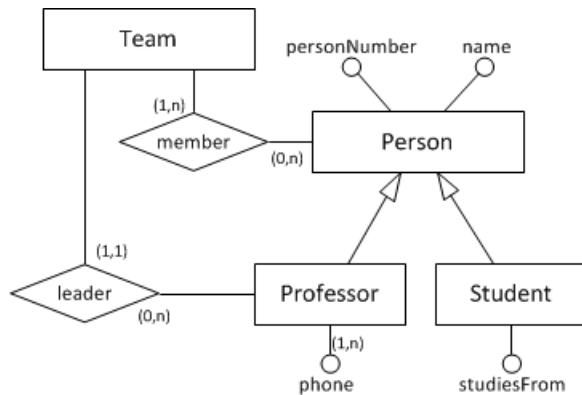
$$\text{Professors} \cup \text{Students} = \text{Persons}$$

- each person is either professor or student

Overlap Constraint:

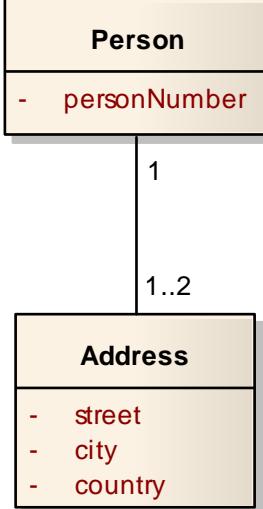
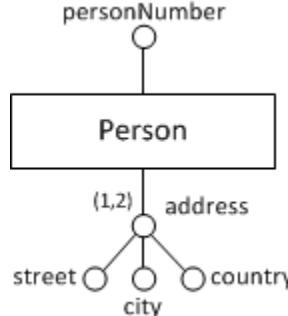
$$\text{Professors} \cap \text{Students} = \emptyset$$

- there is no person which is both professor and student



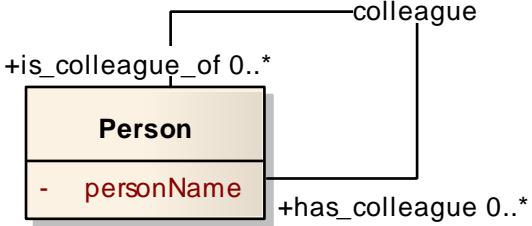
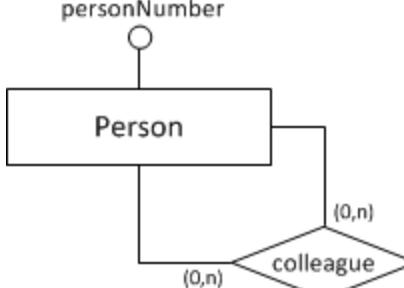
UML and ER Advanced Constructs

Composite attributes

UML	ER	Real-world
 <pre>classDiagram class Person { -personNumber } class Address { -street -city -country } Person "1" -- "1..2" Address</pre>	 <pre>erDiagram Person --o{ Address : o Person { string personNumber } Address { string street string city string country } Person } --o{ Address : o Address } --o{ street : o Address } --o{ city : o Address } --o{ country : o</pre>	A person has one or two addresses comprising street, city and country.
UML construct: No specific construct; composite attributes can be expressed with an auxiliary class.	ER construct: Composite attribute <ul style="list-style-type: none">namecardinalitysub-attributes	

UML and ER Advanced Constructs

Recursive associations

UML	ER	Real-world
		A person has zero or more colleagues.
UML construct: Normal association with the same participants.	ER construct: Normal relationship type with the same participants.	

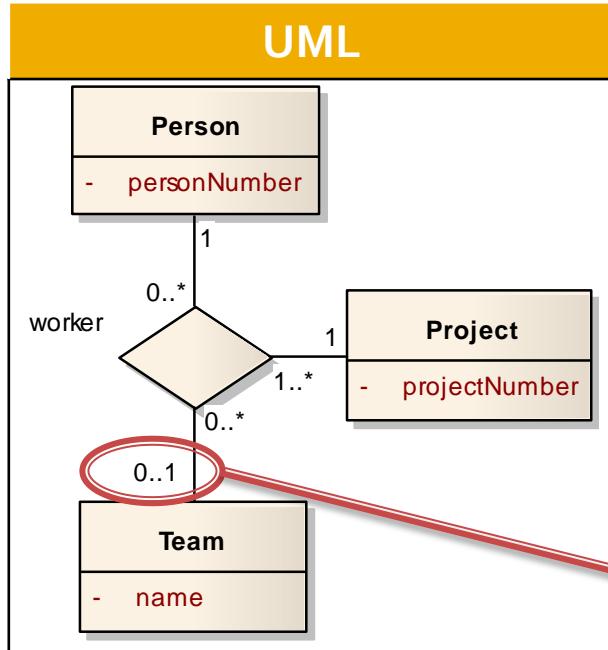
UML and ER Advanced Constructs

N-ary associations

UML	ER	Real-world
<pre> classDiagram class Person { -personNumber } class Project { -projectNumber } class Team { -name } Person "1" *--o "0..*" "1..*" "1" Project Person "1" *--o "0..*" "1..*" "1" Team </pre>	<pre> erDiagram Person --o Project : "worker" Person --o Team : "worker" Project --o Team : "worker" </pre>	<p>A person works on a project but only as a team member.</p>
<p>Construct: N-ary association Similar to binary association but with three or more participants.</p>	<p>Construct: N-ary relationship type Similar to binary relationship type but with three or more participants.</p>	<p>NOTE: Attributes can be expressed in the same way as for binary variants.</p>

UML and ER Advanced Constructs

N-ary associations



- UML n-ary associations have stronger expressive power in their cardinalities than ER n-ary relationship types
- A person can also work on a project as an individual person without a relationship to a team.

UML and ER Advanced Constructs

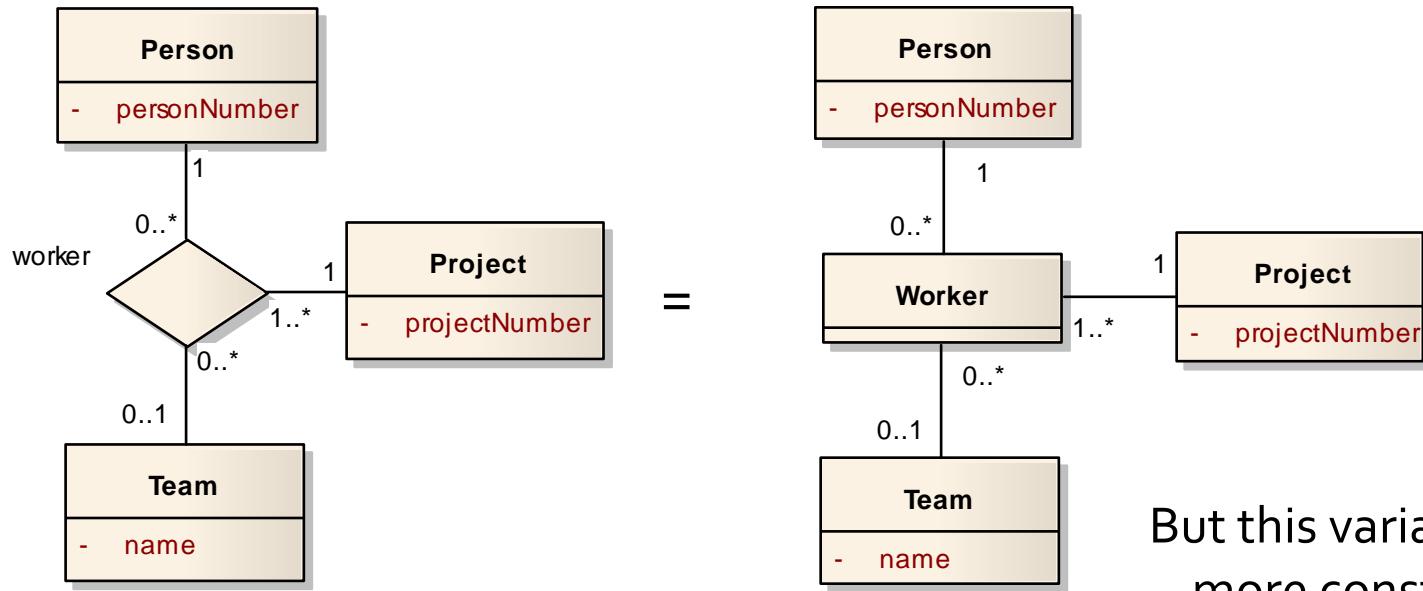
N-ary associations vs. binary associations

	<pre> classDiagram class Person { -personNumber } class Project { -projectNumber } class Team { -name } Person "0..*" o--> "1..*" Project : worker Person "0..*" o--> "1" Team : worker </pre>		<p>Which projects does JP work on as a member of SIRET?</p> <p>P3S.</p>
	<pre> classDiagram class Person { -personNumber } class Project { -projectNumber } class Team { -name } Person "1..*" o--> "0..*" Project : member Person "0..*" o--> "0..*" Team : member </pre>		<p>Which projects does JP work on as a member of SIRET?</p> <p>GraphDB or P3S?</p>

UML and ER Advanced Constructs

N-ary associations vs. binary associations

- n-ary association = class + separate binary association for each original participant



UML and ER Advanced Constructs

Identifiers

UML	ER	Real-world
	<p>The diagram shows a UML class named "Person". It has three attributes: "personNumber" (represented by a solid black circle), "firstName" (represented by an open circle), and "surname" (also represented by an open circle). "personNumber" is marked as an identifier with a multiplicity of 1 and a solid line connecting it to the class. "firstName" and "surname" are marked as identifiers with a multiplicity of 1 and dashed lines connecting them to the class.</p>	<p>A person is identified by its personal number or by a combination of its first name and surname.</p> $(\forall p_1, p_2 \in \text{Persons})$ $(p_1.\text{personNumber} = p_2.\text{personNumber} \rightarrow p_1 = p_2)$ $(\forall p_1, p_2 \in \text{Persons})$ $\left((p_1.\text{firstName} = p_2.\text{firstName} \wedge p_1.\text{surname} = p_2.\text{surname}) \rightarrow p_1 = p_2 \right)$
n/a	<p>Construct: Attribute or a group of attributes marked as identifier.</p>	

UML and ER Advanced Constructs

Weak entity types

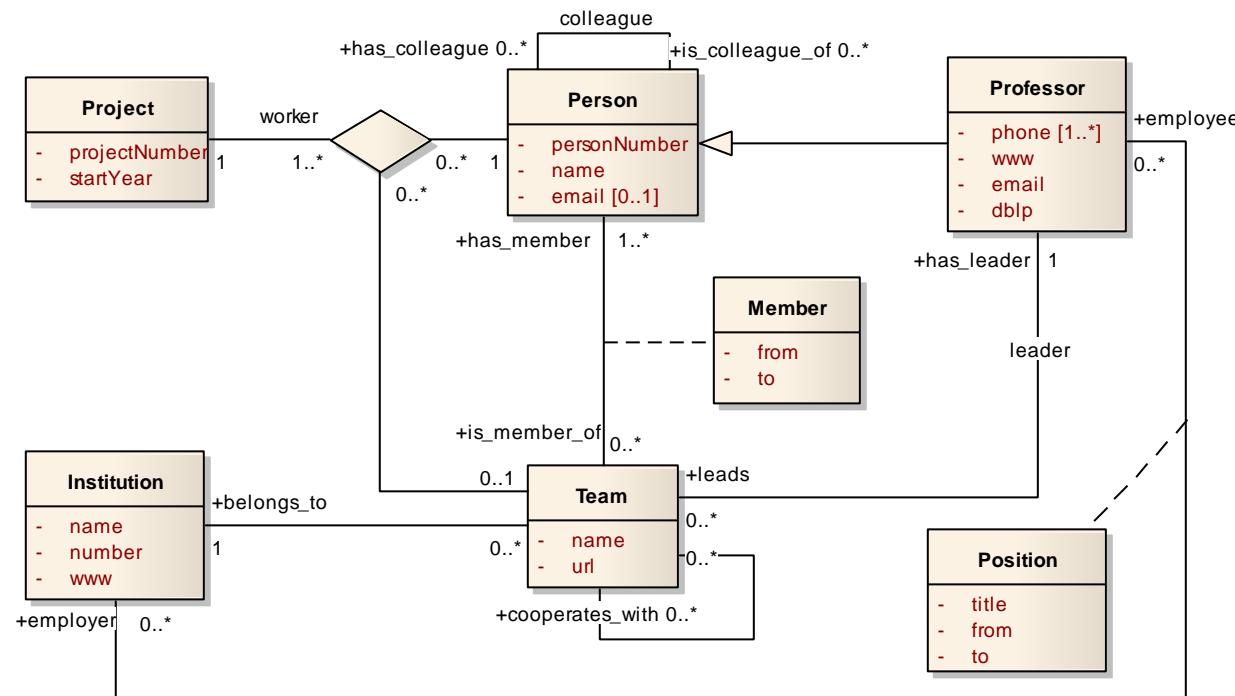
UML	ER	Real-world
	<pre>classDiagram class Team { name } class Institution { name } Team "1..1" *-- "1..n" Institution : belong to</pre>	<p>A team is identified by a combination of its name and a name of its institution.</p> $(\forall t_1, t_2 \in Teams) ((t_1.Institution.name = t_2.Institution.name \wedge t_1.name = t_2.name) \rightarrow t_1 = t_2)$
n/a	<p>Construct: Weak entity type = entity type which participates in a relationship type with card. (1,1) and the relationship is a part of its identifier.</p>	

UML and ER Advanced Constructs

Data types

UML	ER	Real-world
<div style="border: 1px solid black; padding: 10px;"><p>Person</p><ul style="list-style-type: none">- personNumber: int- name: string- email: string- phone: string</div>	<pre>classDiagram class Person { personNumber: int name: string email: string phone: string }</pre> The ER diagram shows a rectangular box labeled "Person". Above it, the attribute "personNumber: int" is shown with a line connecting to the box. Below the box, three attributes are listed: "name: string", "email: string", and "phone: string", each connected by lines to the box. The multiplicity "(0,1)" is placed between "name" and "email", and the multiplicity "(1,n)" is placed between "email" and "phone".	A person has a person number which is an integer and name, email and phone which are strings.
<p>Construct: Attribute of class may have a data type.</p>	<p>Construct: Attribute of entity type may have a data type.</p>	<p>NOTE:</p> <ol style="list-style-type: none">1. Set of data types not specified strictly.2. Data types are not very important at the conceptual layer.

Complete Example in UML



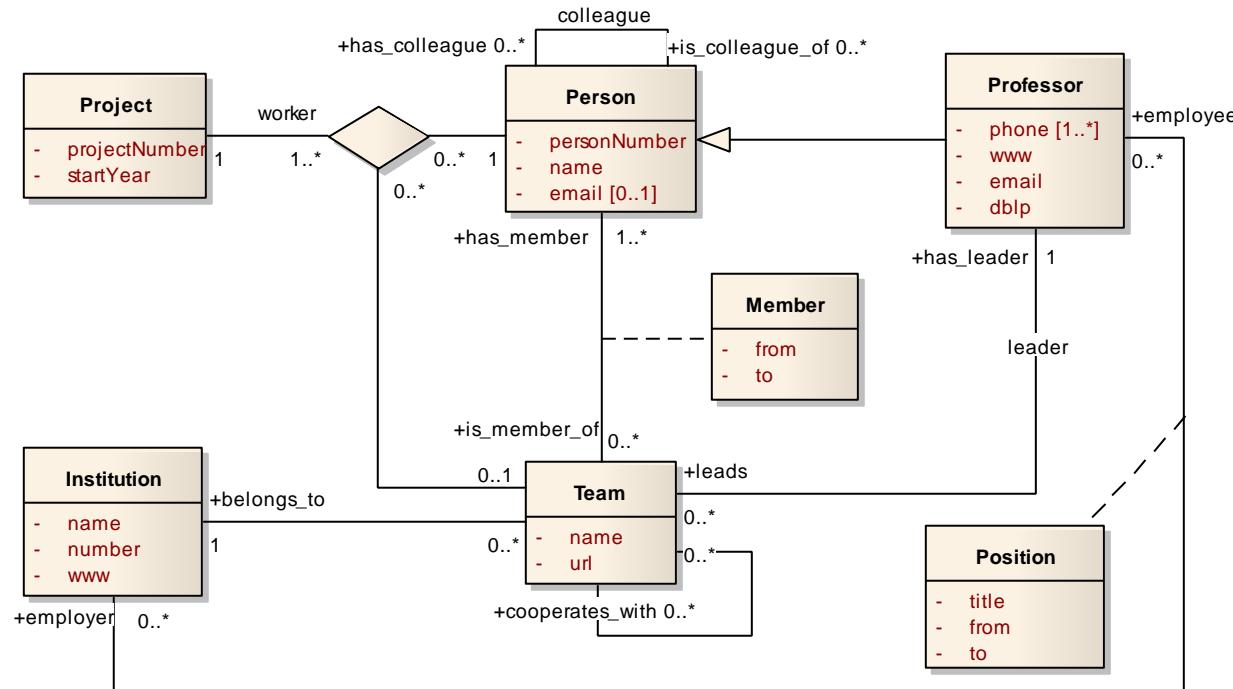
Object Constraint Language (OCL)

- language for formal specification of advanced integrity constraints
- supports invariants, derived values, method pre- and post-conditions, etc.
- we focus on invariants

```
context variable : Class inv  
... constraint ...
```

- constraint
 - variables (**t** : **Team**)
 - navigation paths in your conceptual schema (**t.has_leader.employer**)
 - Assigns a team to **t**
 - From **t** it goes to the **associated collection** of leaders of **t** (the set contains only one leader because of the cardinality 1)
 - From the set of leaders it goes to the **associated collection** of all employers (the set contains zero or more employers of the leader because of the cardinality 0..*)
 - NOTE: The result of navigation (".") is always a **collection**
 - logical operators (**and**, **or**, **implies**)
 - operators on (collection of) instances
 - t.has_leader.employer->size()** > 0 //size of a set
 - t.Project.startYear->min()** //the first project of **t**
 - t.has_leader.employer->forAll(e | e.www->size()>0)**
//size of a string (overloaded)

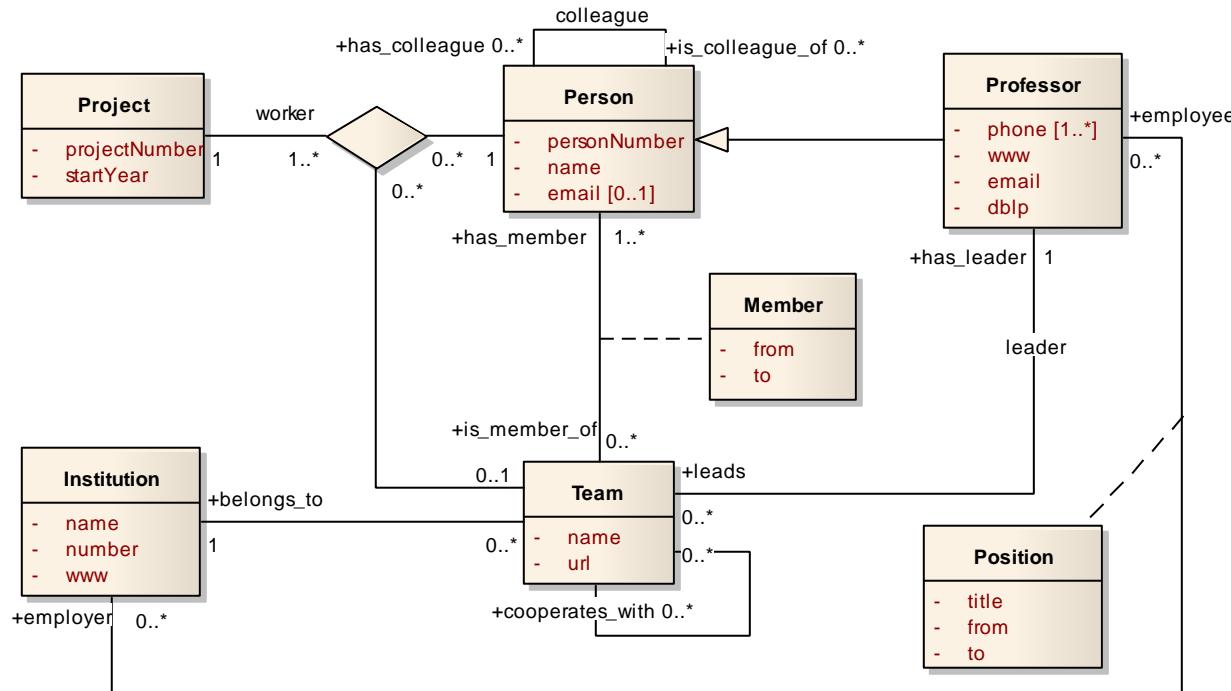
Object Constraint Language (OCL)



```
context p : Project inv  
p.startYear > 1990
```

"Each project must start after 1990."

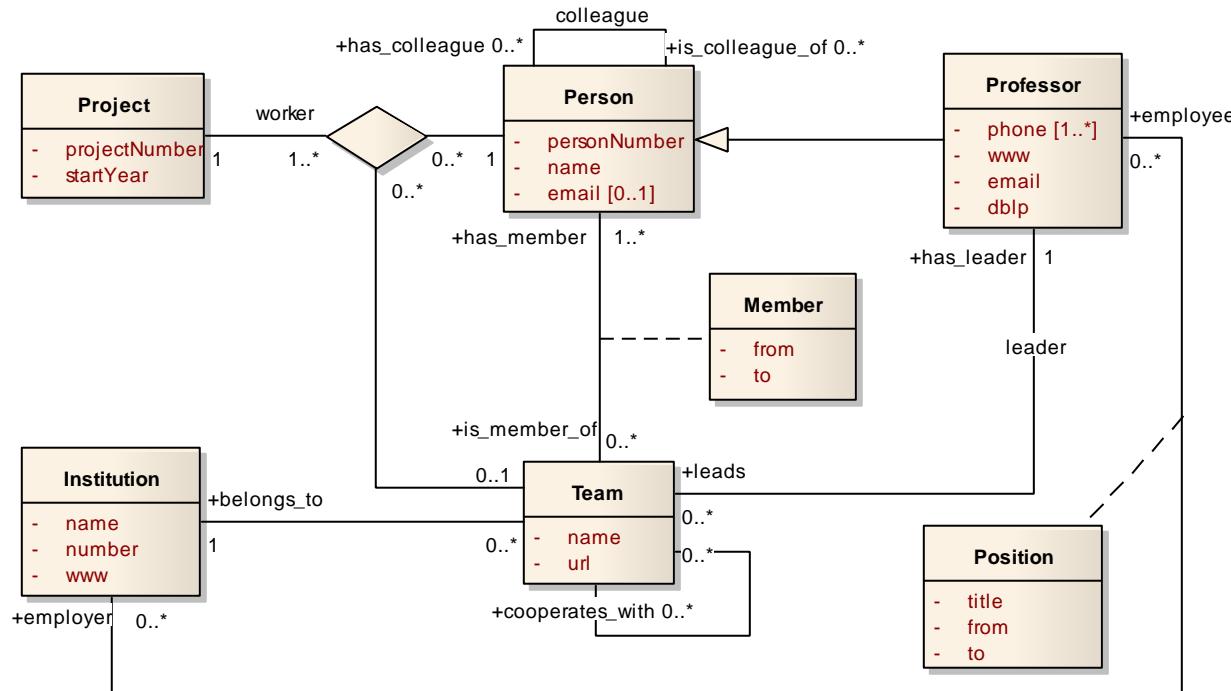
Object Constraint Language (OCL)



```
context Team inv
    self.has_member->size() > 10 implies
        self.worker->size() > 0
```

"Each team with more than 10 members must have a project."

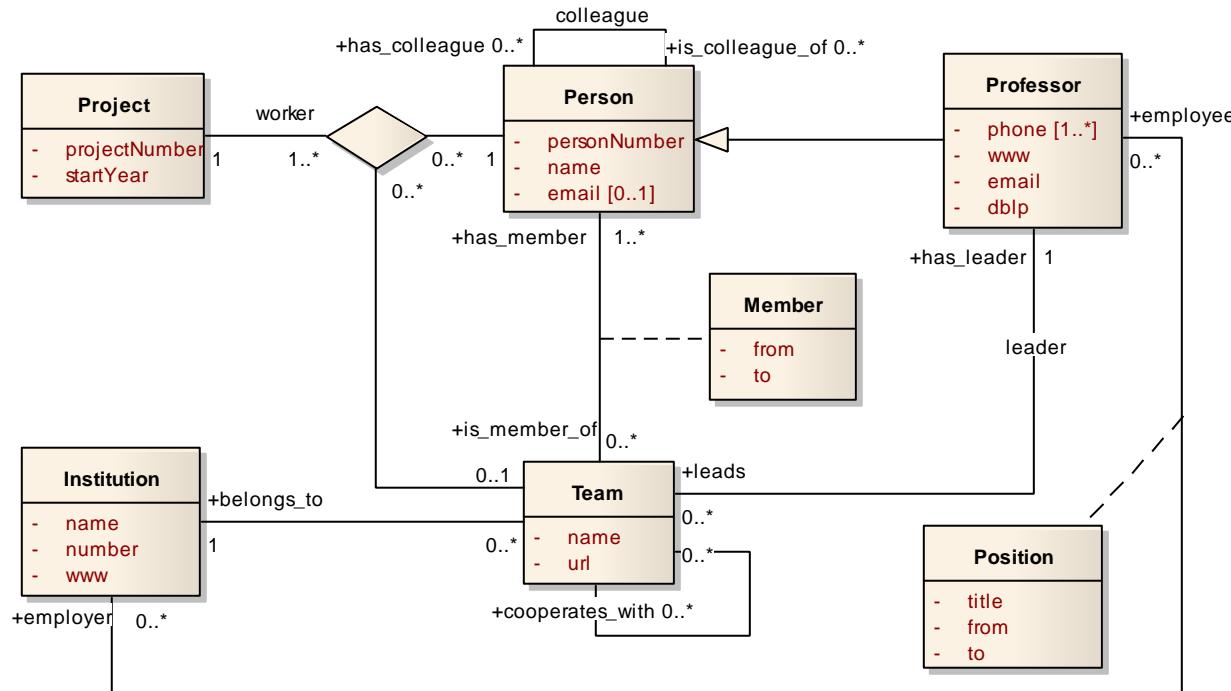
Object Constraint Language (OCL)



```
context p1, p2 : Person inv
    p1.personNumber = p2.personNumber implies p1 = p2
```

"A person is identified by its personal number."

Object Constraint Language (OCL)



```
context t : Team inv
t.belongs_to.employee->exists(p | p = t.has_leader)
```

"A team leader must be an employee of the institution of the team."

OCL Invariants

- Invariants are defined in the context of a specific type, named the context type of the constraint.
- Its body, the boolean condition to be checked, must be satisfied by all instances of the context type.
- Invariants can:
 - restrict the value of single objects, like
 - **context** p : Project **inv**
p.startYear > 1990
 - **context** p : Project **inv** newerProjectsOnly:
p.startYear > 1990
- The *self* variable represents an arbitrary instance of the *Project* class and the dot notation is used to access the properties of the *self* object
- All instances of *Project* class must evaluate this condition to true.

OCL Invariants

- Many invariants express more complex conditions limiting the possible relationships between different objects in the system, usually related through association links.
 - For instance
 - `context t : Team inv`
`t.belongs_to.employee->exists(`
`p | p = t.has_leader`
`)`
- Conjunctions
 - not, and, or, xor, implies
 - Boolean comparisons =, <>

OCL Invariants - Boolean

- Three values: null, false, true
- Unary operator not(b)
 - b not(b)
 - null null
 - false true
 - true false

OCL Invariants - Boolean

- Three values: null, false, true
- Binary operators

<u>a</u>	<u>b</u>	<u>a and b</u>	<u>a or b</u>	<u>a xor b</u>	<u>a implies b</u>	<u>a=b</u>	<u>a<>b</u>
null	null	null	null	null	null	true	false
null	false	false	null	null	null	false	true
null	true	null	true	null	true	false	true
false	null	false	null	null	true	false	true
false	false	false	false	false	true	true	false
false	true	false	true	true	true	false	true
true	null	null	true	null	null	false	true
true	false	false	true	true	false	false	true
true	true	true	true	false	true	true	false

OCL Invariants – Navigation

- Navigation
 - Dot followed by data member
 - Provides – in general – set of values typed by the type of the data member
 - If the set has cardinality 1, it can be maintained as a single value
 - Dot followed by the opposite role name of some association
 - Provides – in general – set of instances of the class on the denoted side of association
 - If the set has cardinality 1, it can be maintained as a single instance
 - For instance – assume t of type Team
 - t.name ... String (set of Strings with cardinality one)
 - t.has_leader ... Professor (set of *Professor* instances with cardinality one)
 - t.has_member ... set of *Person* instances
 - t.belongs_to ... Institution (set of *Institution* instances with cardinality one)
 - t.belongs_to.employee ... set of *Professor* instances

OCL Invariants – Set operators

- Uses arrow notation set->operator(...)
- Single values and single instances can be understood as sets
 - ->size() ... Integer, size of the set
 - ->min() ... minimal value from the set of values
 - ->max() ... maximal value from the set of values
 - ->forAll(x | condition)
... big quantifier – all elements in the set must meet the condition
 - ->exists(x | condition)
... small quantifier – at least one element in the set must meet the condition

OCL Invariants

■ Navigation

- Dot followed by data member
 - Provides – in general – set of values typed by the type of the data member
 - If the set has cardinality 1, it can be maintained as a single value
- Dot followed by the opposite role name of some association
 - Provides – in general – set of instances of the class on the denoted side of association
 - If the set has cardinality 1, it can be maintained as a single instance
- For instance – assume t of type Team
 - t.name ... String (set of Strings with cardinality one)
 - t.has_leader ... Professor (set of *Professor* instances with cardinality one)
 - t.has_member ... set of *Person* instances
 - t.belongs_to ... Institution (set of *Institution* instances with cardinality one)
 - t.belongs_to.employee ... set of *Professor* instances

Software for practices

- ER
 - [ERtoS](#)
- UML
 - trial version of [Enterprise architect](#)
 - full version available in course NSWIo41
 - trial version of [Rational Rose](#)
 - [eXolutio](#)

course:

Database Systems (NDBlo25)

SS2017/18

lecture 2:

Logical database models, relational model

doc. RNDr. Tomáš Skopal, Ph.D.

Mgr. Martin Nečaský, Ph.D.

RNDr. Michal Kopecký, Ph.D.

Today's lecture outline

- introduction to logical database models
 - object, (object-)relational model
 - other (XML, RDF)
- (object-)relational model in detail
 - introduction
 - UML schema conversion

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 model
 - specifies how logical database structures are implemented in a specific technical environment
 - data files, index structures (e.g. B+ trees), etc.



implementation

More on Logical Layer

- specifies how conceptual entities are represented in data structures used by your system for various purposes, e.g.,
 - for data storage and access
 - graph of objects
 - tables in (object-)relational database
 - XML schemas in native XML database
 - for data exchange
 - XML schemas for system-to-system data exchange
 - for data publication on the Web
 - XML schemas for data publication
 - RDF schemas/OWL ontologies for publication on the Web of Linked Data in a machine-readable form
 - other

More on Logical Layer

- → different **logical models** for data representation
 - for data storage and access
 - object model
 - relational model (or object-relational model)
 - XML model
 - for data exchange
 - XML model
 - for data publication on the Web
 - XML model
 - RDF model
 - other
- modern software systems have to cope with many of these different logical models

More on Logical Layer

- problem 1: Designers need to **choose the right logical model** with respect to the nature of the prospective access to data.

or

- problem 2: Designers need to design **several logical schemas** for different models applied in the system.
 - logical schemas can be derived automatically or semi-automatically from a single conceptual schema

→ ***Model-Driven Development***

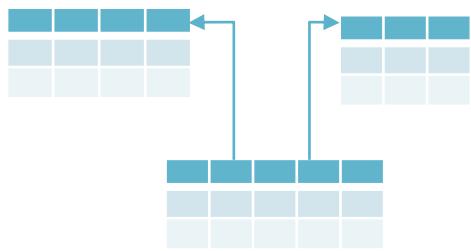
More on Logical Layer

problem 1:

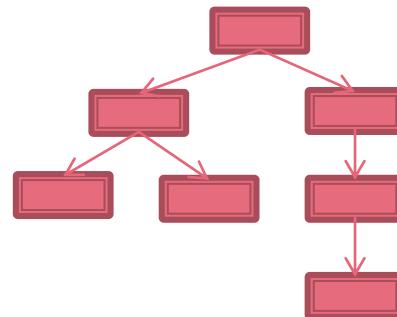
Conceptual Schema



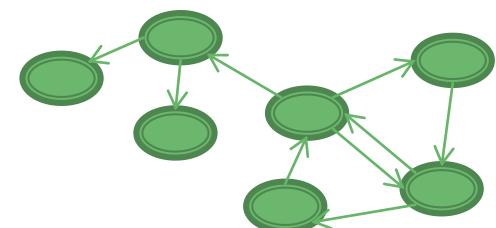
Relational model



XML model

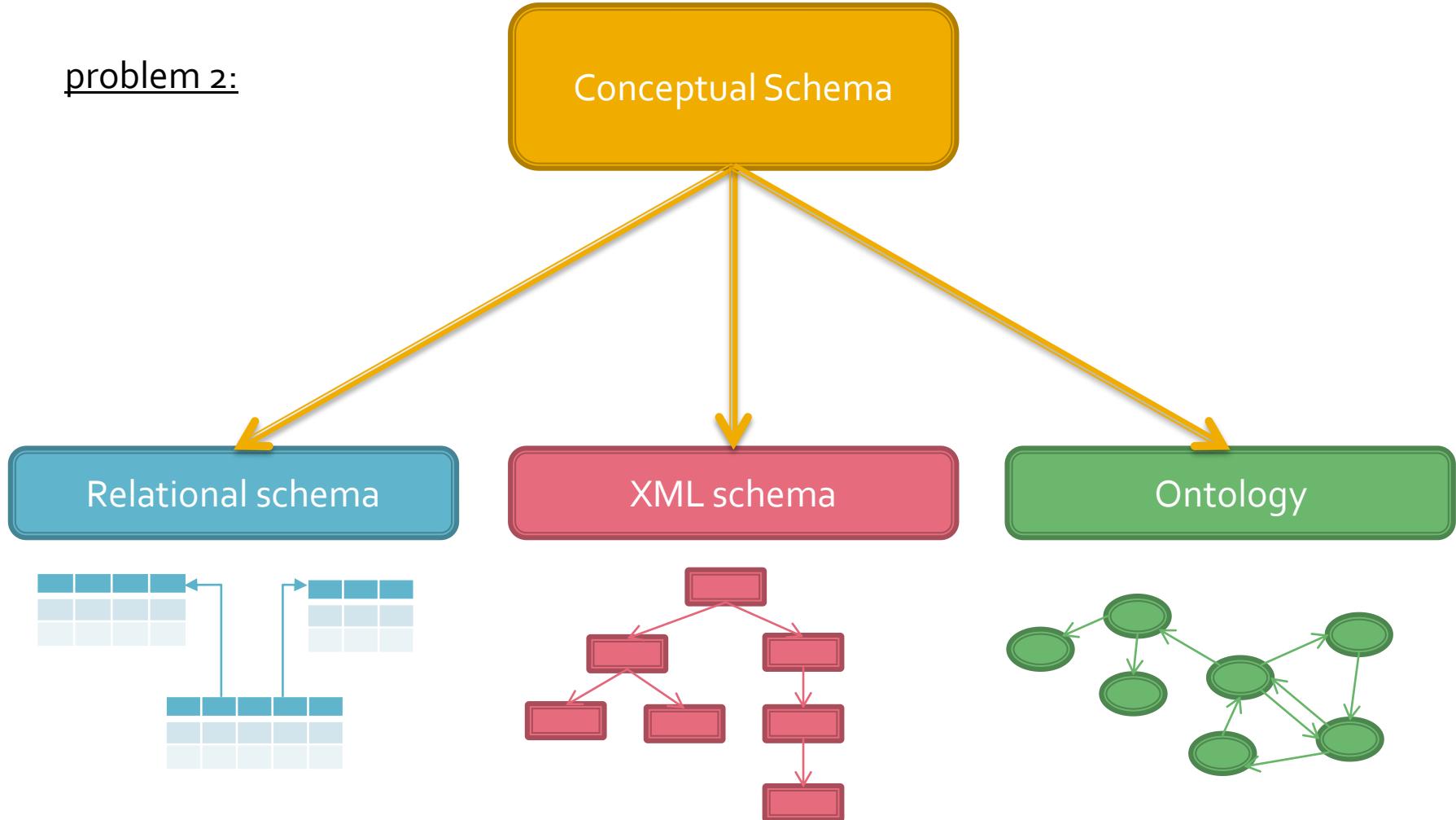


Object model



More on Logical Layer

problem 2:

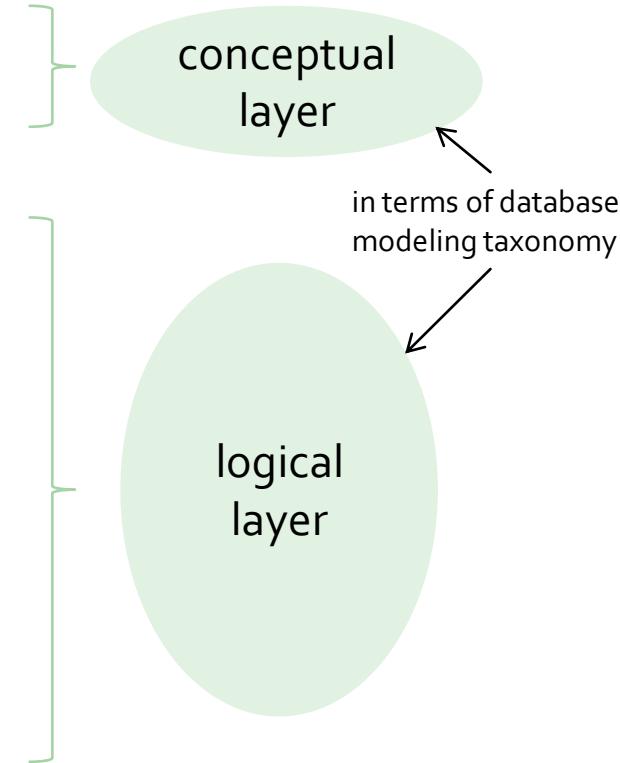


Model-Driven Development (MDD)

- an approach to software development
- theoretically, it enables to create executable schemas instead of executable code (= classical approach)
 - more precisely, schemas are automatically converted to executable code
 - recent idea, not applicable for software development in practice today
 - lack of tool support
- BUT, we will show you how it can be profitably applied for designing logical data schemas
 - allows to deal with different logical models which we need to apply in our system

MDD for Logical Database Schemas

- considers UML as a general modeling language
- distinguishes several levels of abstraction of logical data schemas
 - Platform-Independent Level
 - hides particular platform-specific details
 - Platform-Specific Level
 - mapping of the conceptual schema (or its part) to a given logical model
 - adds platform-specific details to the conceptual schema
 - Code Level
 - schema expression in a selected machine-interpretable logical language
 - for us, e.g., SQL, XML Schema, OWL

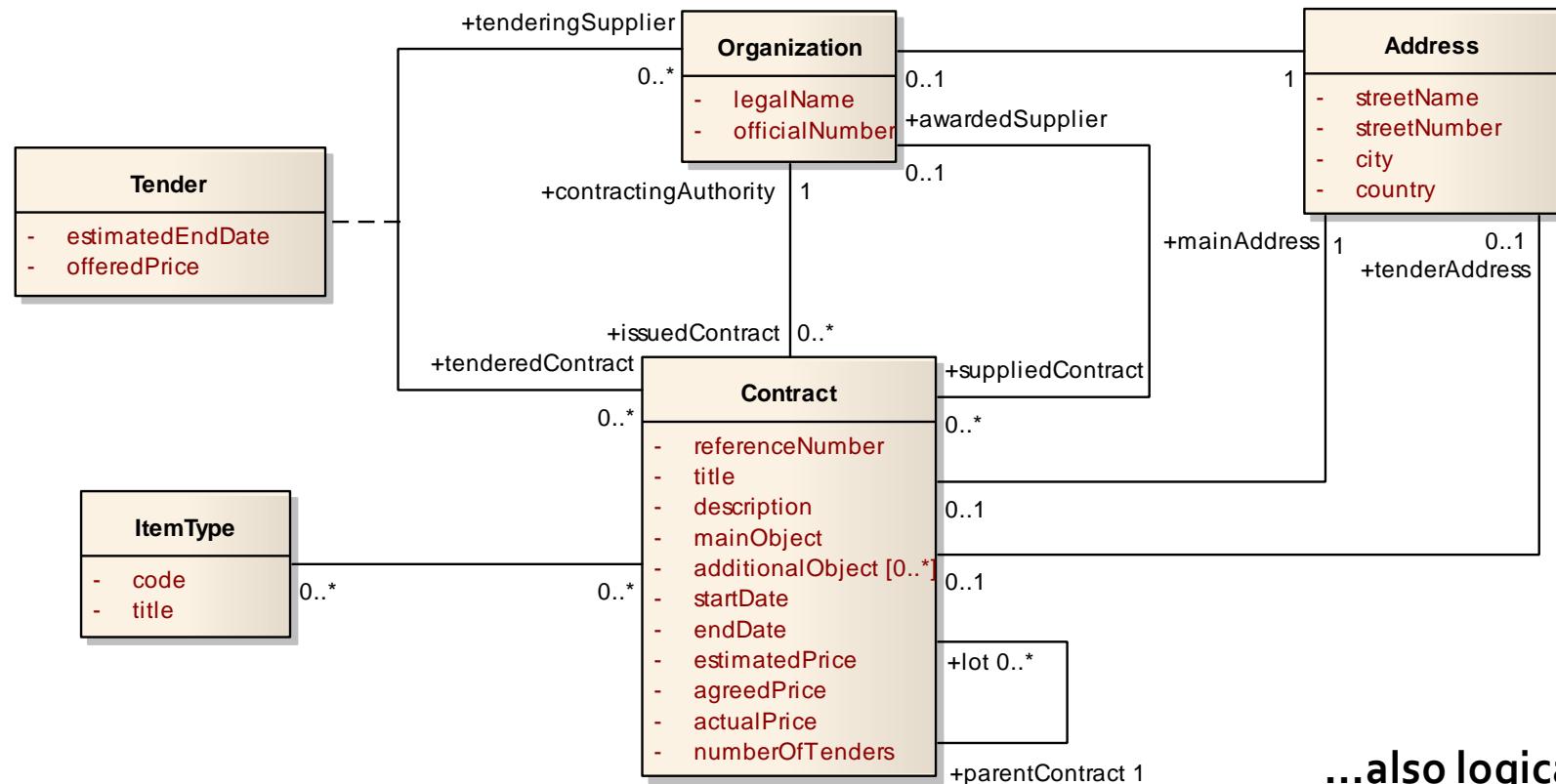


Practical Example 1

- Information System for Public Procurement
 - <http://www.isvzus.cz> (in Czech only ☹)
 - current project of Czech eGov
 - aims at increasing transparency in Public Procurement in Czech Rep.
- system has to deal with many logical models:
 - relational data model for data storage
 - XML data model for exchanging data with information systems of public authorities who issue public contracts
 - *RDF data model for publishing data on the Web of Linked Data in a machine-readable form*
 - *this is not happening in these days (March 2012) but we, Charles University, OpenData.cz, are working hard to show how to do it ☺*

Practical Example

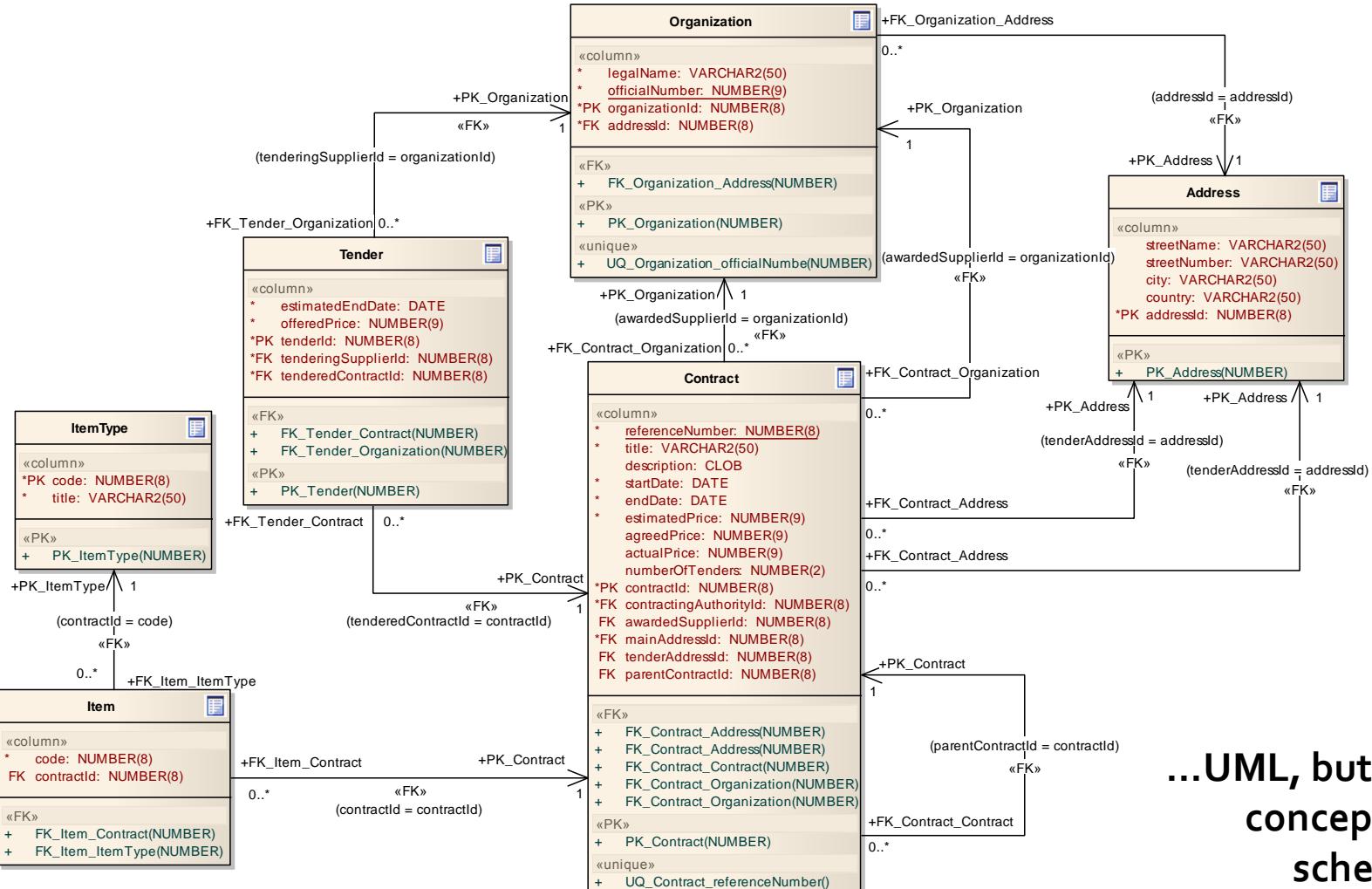
Platform-Independent Level (conceptual model)



...also logical
object schema?

Practical Example 1

Platform-Specific Level (relational model)



...UML, but not
conceptual
schema!

Practical Example 1

Platform-Specific Level (relational model)

- notes to previous UML diagram
 - it is UML class diagram
 - but enhanced with features for modeling schema in **(object-)relational model**
 - enhancement = stereotypes
- stereotype = UML construct assigned to a basic construct (class, attribute, association) with a specific semantics, e.g.,
 - <<table>> specifies that a class models a table
 - <<PK>> specifies that an attribute models a primary key
 - <<FK>> specifies that an attribute/association models a foreign key
 - etc.

Practical Example 1

Code Level (SQL)

```
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,
    agreedPrice          NUMBER(9),
    actualPrice          NUMBER(9),
    numberOfWorkers      NUMBER(2),
    contractId           NUMBER(8) NOT NULL,
    contractingAuthorityId NUMBER(8) NOT NULL,
    awardedSupplierId    NUMBER(8),
    mainAddressId        NUMBER(8) NOT NULL,
    tenderAddress         NUMBER(8),
    parentContractId     NUMBER(8));

ALTER TABLE Contract ADD CONSTRAINT PK_Contract PRIMARY KEY (contractId);
ALTER TABLE Contract ADD CONSTRAINT FK_Contract_Address FOREIGN KEY (mainAddressId) REFERENCES Address (addressId);
ALTER TABLE Contract ADD CONSTRAINT FK_Contract_Address FOREIGN KEY (tenderAddress) REFERENCES Address (addressId);
ALTER TABLE Contract ADD CONSTRAINT FK_Contract_Contract FOREIGN KEY (parentContractId) REFERENCES Contract (contractId);
ALTER TABLE Contract ADD CONSTRAINT FK_Contract_Organization FOREIGN KEY (contractingAuthorityId) REFERENCES Organization (organizationId);
ALTER TABLE Contract ADD CONSTRAINT FK_Contract_Organization FOREIGN KEY (awardedSupplierId) REFERENCES Organization (organizationId);

CREATE TABLE Organization(
    legalName      VARCHAR2(50) NOT NULL,
    officialNumber NUMBER(9) NOT NULL,
    organizationId NUMBER(8) NOT NULL,
    addressId      NUMBER(8) NOT NULL);

ALTER TABLE Organization ADD CONSTRAINT PK_Organization PRIMARY KEY (organizationId);
ALTER TABLE Organization ADD CONSTRAINT FK_Organization_Address FOREIGN KEY (addressId) REFERENCES Address (addressId);

...
```

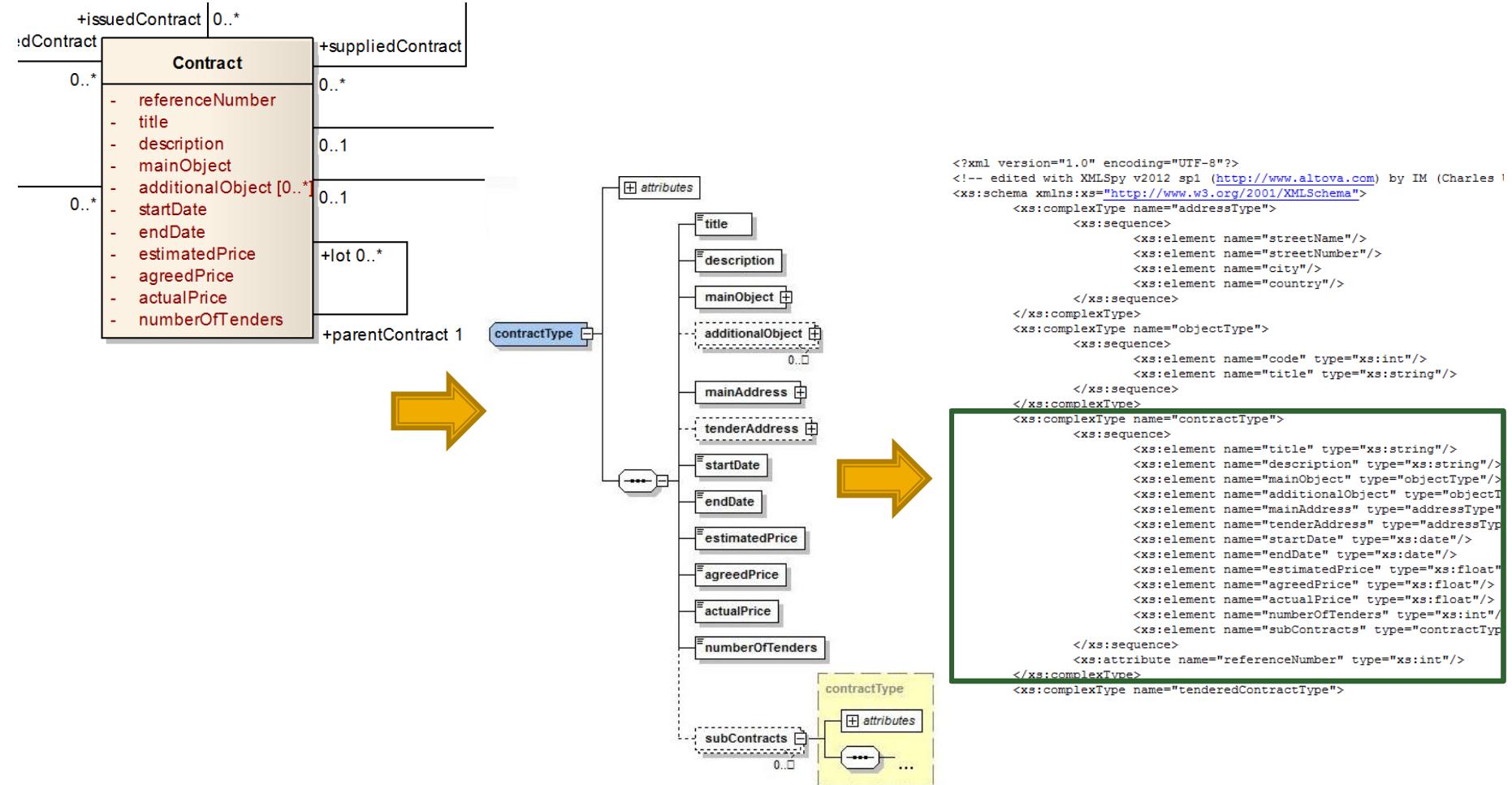
Practical Example 1

Code Level (SQL)

- notes to previous SQL code
 - generated fully automatically from the platform-specific diagram by a CASE tool
 - possibility to automatically generate the code requires all necessary information present in the platform-specific diagram
 - however, it is easier and less error prone to specify the information at the platform-specific level (CASE tool can detect errors and helps with the specification)

Practical Example 2

XML



Object vs. relational model

- object model
 - data stored as graph of objects (in terms of OOP)
 - suitable for individual navigational access to entities, follows the conceptual model
 - not suitable for “batch operations” (data-intensive applications)
 - comfort support in software development platforms, e.g., Hibernate in Java or ADO.NET Entity Framework
 - the goal:
the programmer need not to care of her/his object hierarchy persistency
 - application data is loaded/stored from/to the database as needed,
the data exist regardless of the application runtime

Object vs. relational model

- physical implementation of object model
 - native object DBMS (e.g., Caché)
 - object-relational mapping
 - object schema must be mapped to relational schemas
 - used also by Hibernate, etc.
 - brings overhead
 - generally, storage/serialization of general object hierarchy (graph data) is problematic
 - suitable only for enterprise applications with most of the logic not based on data processing
 - simply, structurally complex but low volume data
 - for data-intensive applications with „flat“ data the relational model is crucial

Object vs. relational model

- relational model
 - data stored in flat tables
 - attribute values are simple data types
 - suitable for data-intensive “batch operations”
 - not suitable for individual navigational access to entities (many joins)
- object-relational model
 - relational model enriched by object elements
 - attributes general classes (not only simple types)
 - methods/functions defined on attribute types

Relational model

- founded by E.F Codd in article „*A relational model of data for large shared data banks*“, Communications of ACM, 1970
 - much sooner than UML and even ER modeling!
- model for storage of objects and their associations in **tables (relations)**
- object or association is represented by one **row** in the table (member of relation)
- an **attribute** of an object/association is represented by a **column** in the table
- **table schema (relation schema)** – description of the table structure (everything except the data, i.e., metadata)
 - $S(A_1:T_1, A_2:T_2, \dots)$ – S the schema name, A_i are attributes and T_i their types
- **schema of relational database** – set of relation schemas (+ other stuff, like integrity constraints, etc.)

Relational model

- basic integrity constraints
 - two identical rows cannot exist (unique identification)
 - the attribute domain is given by its type (simple typed attributes)
 - the table is complete, there are no “holes”, i.e., every cell in the table has its value (value NULL could be included as a „metavalue“)
- every row is identified by one or more attributes of the table
 - called a **superkey** of a table (special case is the entire row)
 - superkey with minimal number of attributes is called a **key** (multiple keys allowed)
- foreign key (“inter-table” integrity constraint)
 - is a group of attributes that exists in another (referred) table, where the it is a (super)key
 - consequence: usually it is not superkey in the referring table since there may exist duplicate values in that attributes

Notation

- notation



Relation1(keyAttr, otherKeyAttr, normalAttr)



Relation2(keyAttrPart1, keyAttrPart2)

Relation3(keyAttr, normalAttr, **foreignKeyAttr**),
foreignKeyAttr \subseteq Relation1.keyAttr

foreign key

Relational model

Product(Name: string, Producer:string, Price: float, Availability: int), Product.Producer \subseteq Producer.Name

Name	Producer	Price	Availability
Mouse	Dell	5,80	100000
Windows	Microsoft	12,50	NULL
Printer	Toshiba	325,-	15000
Phone	Nokia	135,-	32000
Laptop	Dell	500	9000
Bing	Microsoft	0	NULL

key

foreign key
(key in table Producer)

Producer(Name: string, Address:string, Debt: bool)

Name	Address	Debt
Dell	USA	YES
Microsoft	USA	NO
Samsung	Korea	NO
Nokia	Norway	YES
E.ON	Germany	NO
Toshiba	Japan	NO

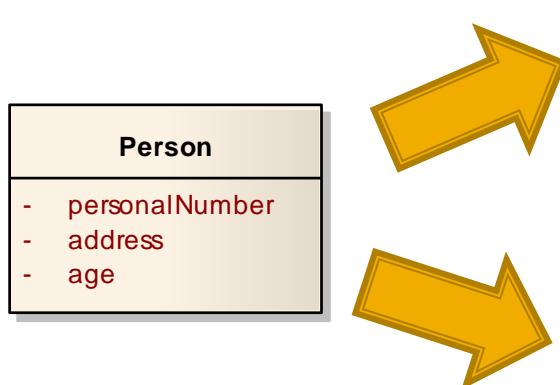
key

Translation of conceptual schema

- designing relational database based on a conceptual schema
- UML or ER diagram consists of
 - classes – objects directly stored in table rows
 - associations – must be also stored in tables
 - either separate, or together with the classes (depends on cardinalities)

Translation of Classes

- class translated to separate table
 - in some cases more classes can be translated to a single table (see next slides)



Person(personalNumber, address, age)

- key is an attribute derived from an attribute in the conceptual schema
 - from identifier (ER) or stereotype/OCL (UML)

Person(personID, personalNumber, address, age)

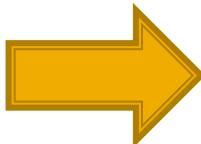
- key is also an artificial attribute with no correspondence in real world
 - automatically generated values

- artificial identifiers are preferred candidates for primary keys!
- **NOTE: we will use only artificial identifiers in the following text**

Translation of Multiplied Attributes

- multiplied attributes must have separate table

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

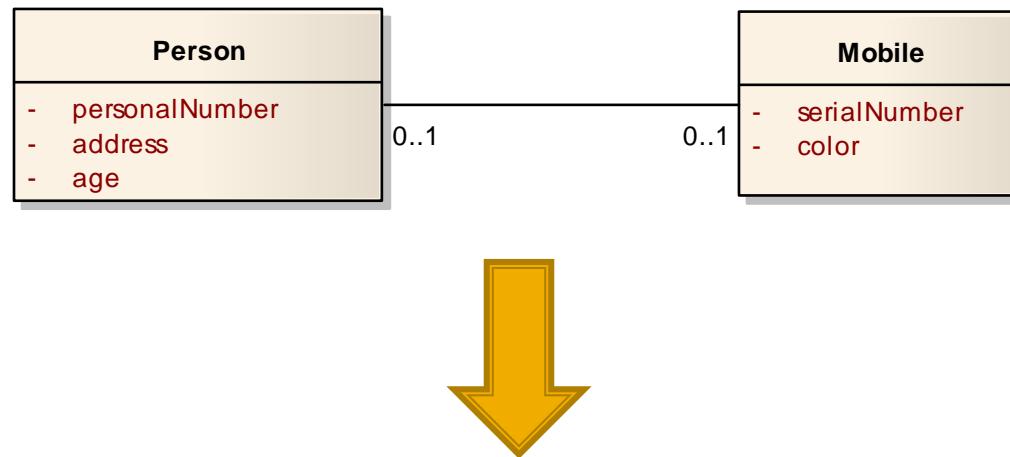


Person(personID, personalNumber)
Phone(phoneID, phone, **personID**)
personID \subseteq Person.personID

Translation of Associations

Multiplicity (0,1):(0,1)

- three tables T_1 , T_2 , and T_3
 - T_3 represents the association



Person(personID, personalNumber, address, age)

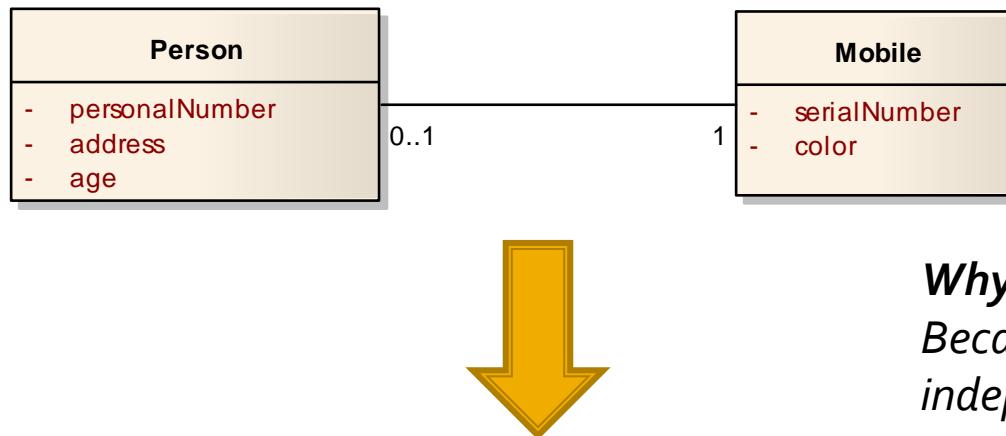
Mobile(mobileID, serialNumber, color)

Owns(personID, mobileID), personID \subseteq Person.personID, mobileID \subseteq Mobile.mobileID

Translation of Associations

Multiplicity (1,1):(0,1)

- two tables T_1 (0,1) and T_2 (1,1)
 - T_1 independent of T_2
 - T_2 contains a foreign key to T_1



Person(personID, personalNumber, address, age, mobileID)
mobileID \subseteq Mobile.mobileID

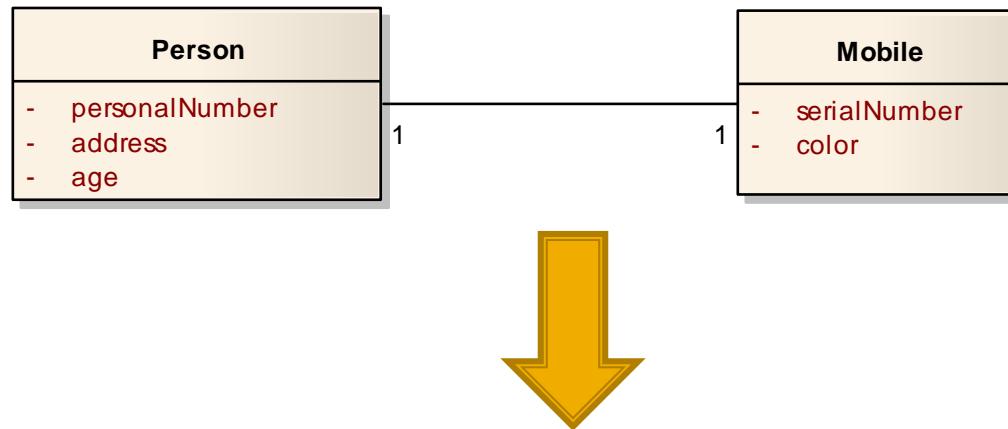
Mobile(mobileID, serialNumber, color)

Why not only 1 table?
Because a mobile can exist independently of a person.
Having it in a single table with persons would lead to empty "person" fields for mobiles without a person.

Translation of Associations

Multiplicity (1,1):(1,1)

- single table, key from one class or both (two keys)

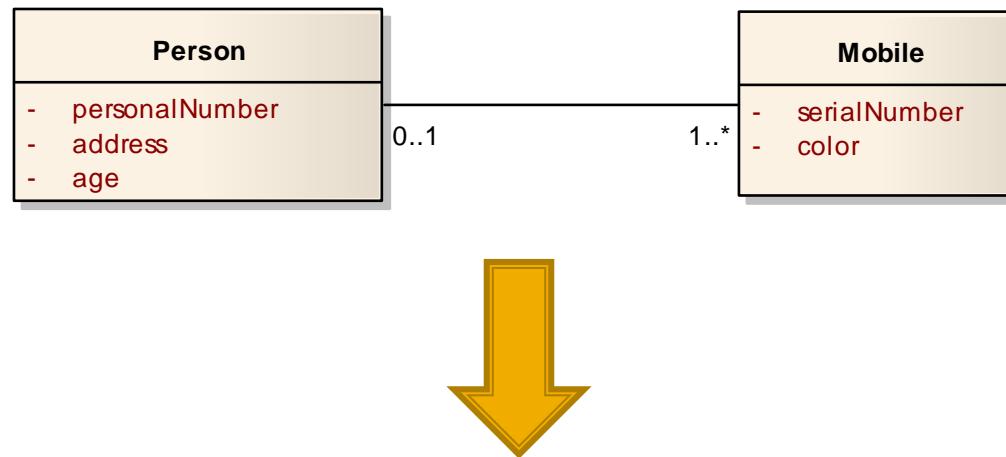


Person(personID, personalNumber, address, age, mobileID, serialNumber, color)

Translation of Associations

Multiplicity $(1,n)/(0,n):(0,1)$

- three tables T_1 , T_2 , and T_3 similarly to $(0,1):(0,1)$



Person(personID, personalNumber, address, age)

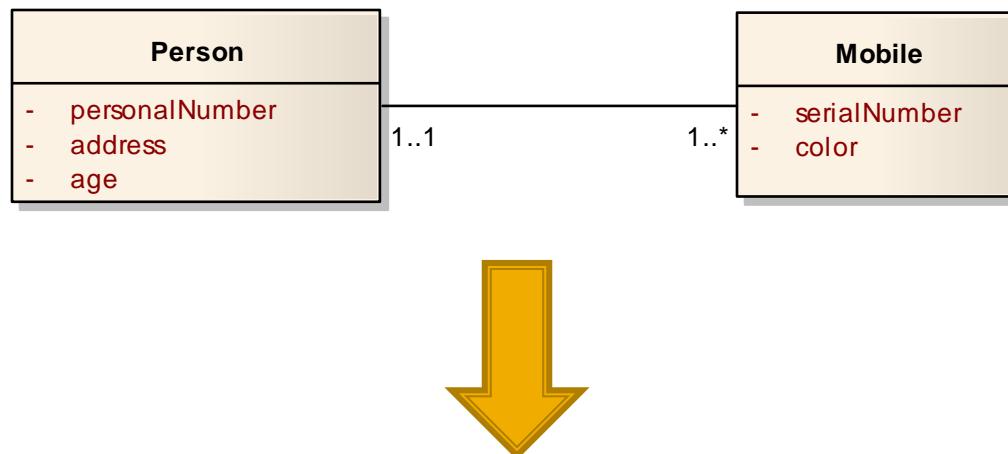
Mobile(mobileID, serialNumber, color)

Owns(personID, mobileID), personID \subseteq Person.personID, mobileID \subseteq Mobile.mobileID

Translation of Associations

Multiplicity $(1,n)/(0,n):(1,1)$

- two tables T_1 and T_2 , similarly to $(0,1):(1,1)$
 - T_1 independent of T_2



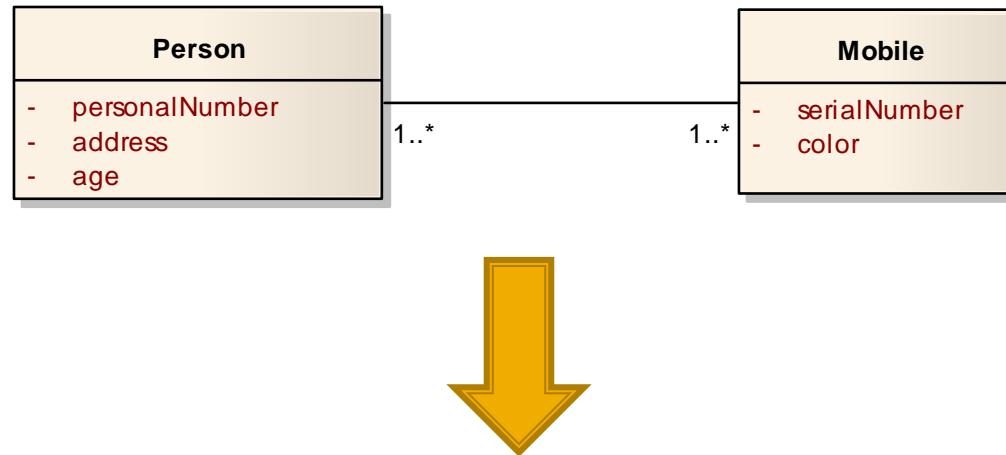
Person(personID, personalNumber, address, age)

Mobile(mobileID, serialNumber, color, **personID**) , **personID** \subseteq Person.personID

Translation of Associations

Multiplicity $(1,n)/(0,n):(1,n)/(0,n)$

- three tables T_1 , T_2 , and T_3
 - T_3 is association table, its key is **combination** of keys of T_1 and T_2 (!)



Person(personID, personalNumber, address, age)

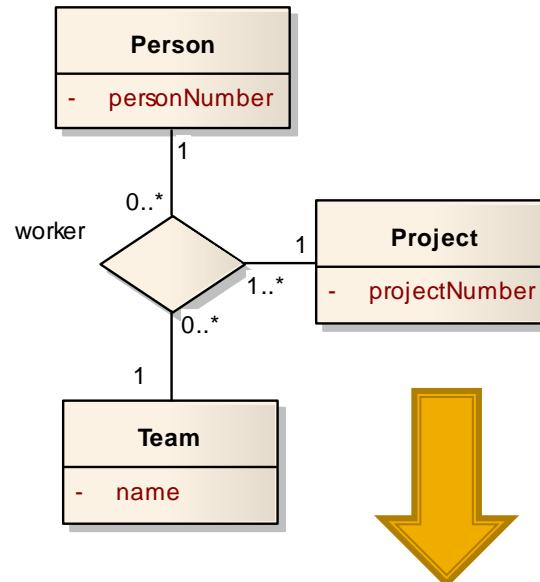
Mobile(mobileID, serialNumber, color)

Owns(personID, mobileID) , personID \subseteq Person.personID, mobileID \subseteq Mobile.mobileID

Translation of Associations

N-ary Associations

- N tables + 1 association table



Less tables?

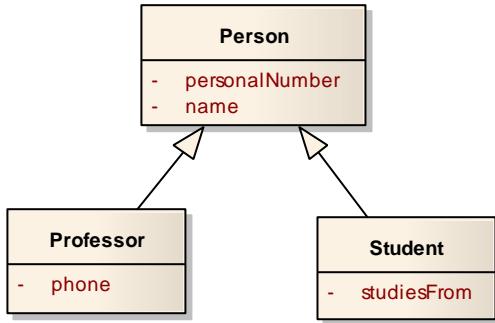
Yes, similarly to binary associations when ..1 appears instead of ...*

Person(personID, personNumber), Team(teamID, name), Project(projectID, projectNumber)

Worker(personID, teamID, projectID),

$\text{personID} \subseteq \text{Person}.\underline{\text{personID}}$, $\text{teamID} \subseteq \text{Team}.\underline{\text{teamID}}$, $\text{projectID} \subseteq \text{Project}.\underline{\text{projectID}}$

Translation of ISA hierarchy



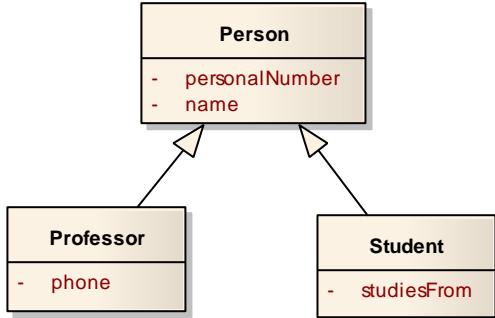
Person(personID, personalNumber, name)

Professor(personID, phone), personID \subseteq Person.personID)

Student (personID, studiesFrom), personID \subseteq Person.personID)

- general solution applicable in any case
- table for each type, each has
- pros:
 - flexibility (e.g., when adding new attributes to Person, only table Person needs to be altered)
- cons:
 - joins

Translation of ISA hierarchy



Suitable when overlap constraint is false, i.e.:

$$Professors \cap Students \neq \emptyset$$

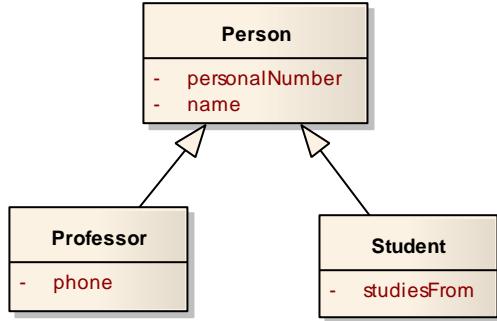
and, ideally, most of persons are both professor and student

$$|Professors \cap Students| \cong |Persons|$$

Person(personID, personalNumber, name, phone, studiesFrom, type)

- all instances stored in single table
- instance type (Person, Professor or Student) is distinguished by artificial type attribute
- pros:
 - one table, no joins
- cons:
 - NULL values when overlap constraint is true, or false but $|Professors \cap Students| \ll |Persons|$ (i.e. most persons are professor or student but not both)

Translation of ISA hierarchy



Suitable when covering constraint is true, i.e.:

$$\text{Professors} \cup \text{Students} = \text{Persons}$$

Unsuitable when overlap constraint is false, i.e.:

$$\text{Professors} \cap \text{Students} \neq \emptyset$$

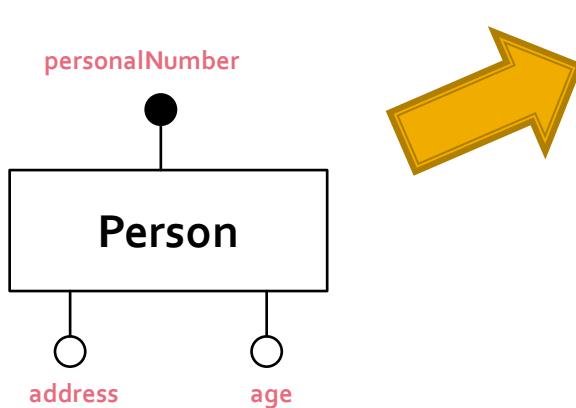
Professor(professorID, personalNumber, name, phone)

Student(studentID, personalNumber, name, studiesFrom)

- tables only for “leaf” (only non-abstract) types
- pros: no joins
- cons:
 - problem with representing persons which are neither professors nor students (i.e. covering constraint is false)
 - redundancies when a professor can be a student or vice versa (i.e. overlap constraint is false)

Translation of Entity from E-R model

- entity translated to separate table
 - In fact the same rules as in UML model

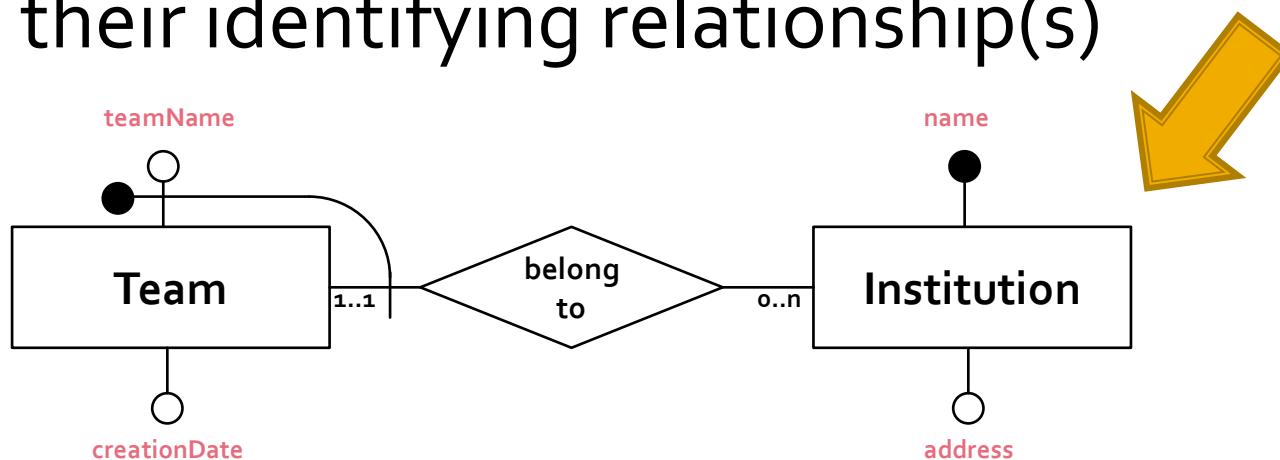


Person(personalNumber, address, age)

- key is an attribute derived from an attribute in the E-R conceptual schema
- No artificial keys are supposed

Translation of E-R Relation

- The same rules as in UML
- Moreover, E-R model uses *weak entity types* and their identifying relationship(s)



- Non-weak entity has to be translated first
Institution(**name**, address)
- Weak entity then can inherit its key to form its composite key
Team(**name**, **teamName**, creationDate),
name ⊂ Institution.name

course:

Database Systems (NDBlo25)

SS2017/18

lecture 3:

SQL – queries

doc. RNDr. Tomáš Skopal, Ph.D.

RNDr. Michal Kopecký, Ph.D.

Today's lecture outline

- introduction to SQL
- queries
 - SELECT command
 - sorting
 - set operations

SQL

- structured query language
 - standard language for access to (relational) databases
 - originally ambitions to provide “natural language”
(that’s why, e.g., SELECT is so complex – a single phrase)
- also language for
 - data definition (DDL)
 - creation and altering of relational (table) schemas
 - data manipulation (DML)
 - querying
 - data insertion, deletion, updating
 - transaction management
 - modules (programming language)
 - definition of integrity constraints
 - administration

SQL

- standards ANSI/ISO SQL 86, 89, 92, 1999, 2003
(backwards compatible)
- commercial systems implement SQL at different standard level (most often SQL 99, 2003)
 - unfortunately, not strict implementation – some extra nonstandard features supported while some standard ones not supported
 - specific extensions procedural, transactional and other functionality, e.g., TRANSACT-SQL (Microsoft SQL Server), PL/SQL (Oracle)

SQL evolution

- **SQL 86** – first „shot“, intersection of IBM SQL implementations
 - **SQL 89** – small revision triggered by industry, many details left for 3rd parties
- **SQL 92** – stronger language, specification 6x longer than for SQL 86/89
 - schema modification, tables with metadata, inner joins, cascade deletes/updates based on foreign keys, set operations, transactions, cursors, exceptions
 - four subversions – Entry, Transitional, Intermediate, Full
- **SQL 1999** – many new features, e.g.,
 - object-relational extensions
 - types STRING, BOOLEAN, REF, ARRAY, types for full-text, images, spatial data
 - triggers, roles, programming language, regular expressions, recursive queries, etc.
- **SQL 2003** – further extensions, e.g., XML management, autonumbers, std. sequences, but also type BIT removed

Queries in SQL

- query in SQL vs. relational calculus and algebra
 - command SELECT shares elements of both formalisms
 - extended domain relational calculus (usage of columns, quantifiers, aggregation functions)
 - algebra (some operations – projection, selection, join, cartesian product, set operations)
 - allowed duplicate rows and NULL attribute values
- syntax validators for SQL 92, 1999, 2003
 - allows to check a query (or other SQL command) based on the norm
 - <http://developer.mimer.com/validator/index.htm>

Queries in SQL

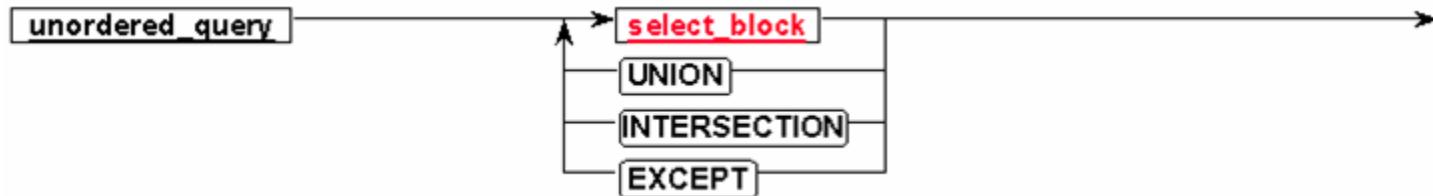
- for simplicity we consider SQL 86 syntax, and use syntax diagrams

source: prof. H.J. Schek (ETH Zurich)

 - oriented graph (DFA automaton accepting SQL)
 - terms in diagrams
 - small letters, underlined – subexpression in give construction
 - capital letters – SQL keyword
 - small letters, italic – name (table/column/...)
 - we use usual terminology table/column instead of relation/attribute
 - diagrams do not include ANSI SQL 92 syntax for joins
(clauses CROSS JOIN, NATURAL JOIN, INNER JOIN, LEFT | RIGHT |
FULL OUTER JOIN, UNION JOIN) – shown later

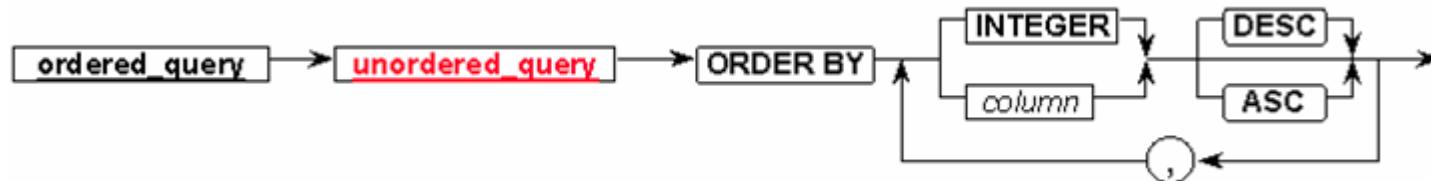
Basic query construction

- unordered query consists of
 - always command **SELECT** (main query logic)
 - optionally commands **UNION, INTERSECTION, EXCEPT**
(union/intersection/subtraction of two or more results obtained by query defined by **SELECT** command)
 - elements in results have no defined order (their order is determined by the implementation of query evaluation, resp.)



Ordered query

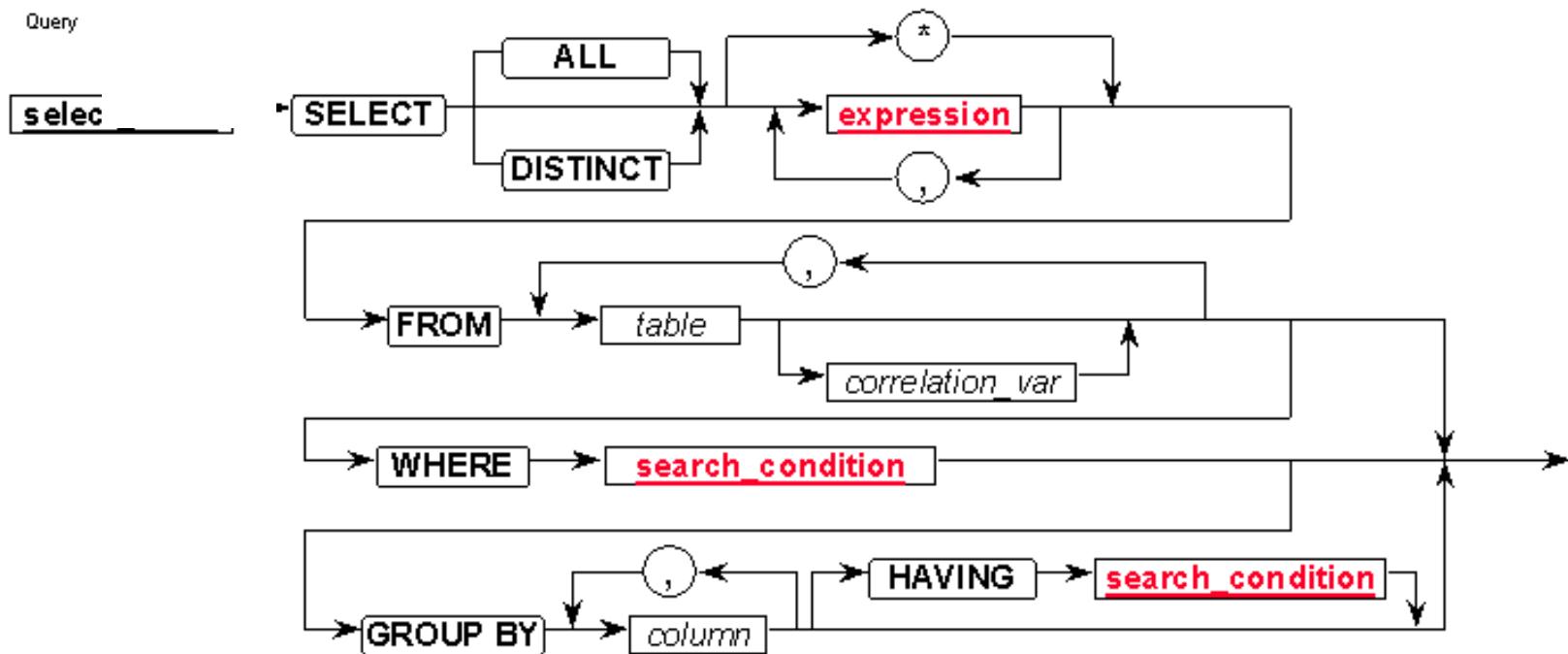
- the result of unorderd query can be ordered/sorted
 - clause **ORDER BY**, ordering according to column(s)
 - ascending (**ASC**) or descending (**DESC**) order
 - multiple secondary columns can be specified that apply in case of duplicate values in the primary (higher secondary) columns



SELECT

- **SELECT** command consists of 2-5 clauses (+ optionally **ORDER BY**)
 - clause **SELECT** – projection into output schema, or definition of new, derived, aggregated columns
 - clause **FROM** – which tables (in case of SQL ≥ 99 also nested queries, views) we query
 - clause **WHERE** – condition that must a row (record) satisfy in order to get into the result
 - clause **GROUP BY** – which attributes should serve for aggregation
 - clause **HAVING** – condition that must **an aggregated** row (record) satisfy in order to get into the result

SELECT – diagram



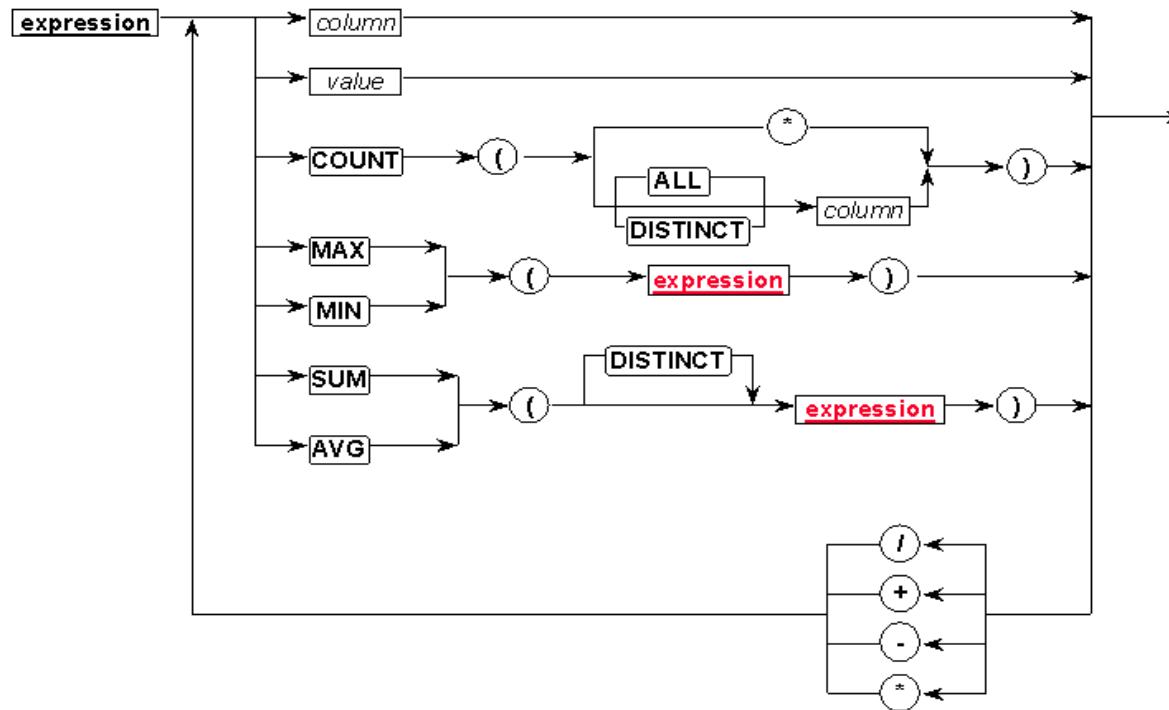
Logical order of evaluation (associativity of SELECT clauses, resp.):

FROM → WHERE → GROUP BY → HAVING → projection SELECT (→ ORDER BY)

SELECT – expression

Context: SELECT ALL | DISTINCT **expression** FROM ...

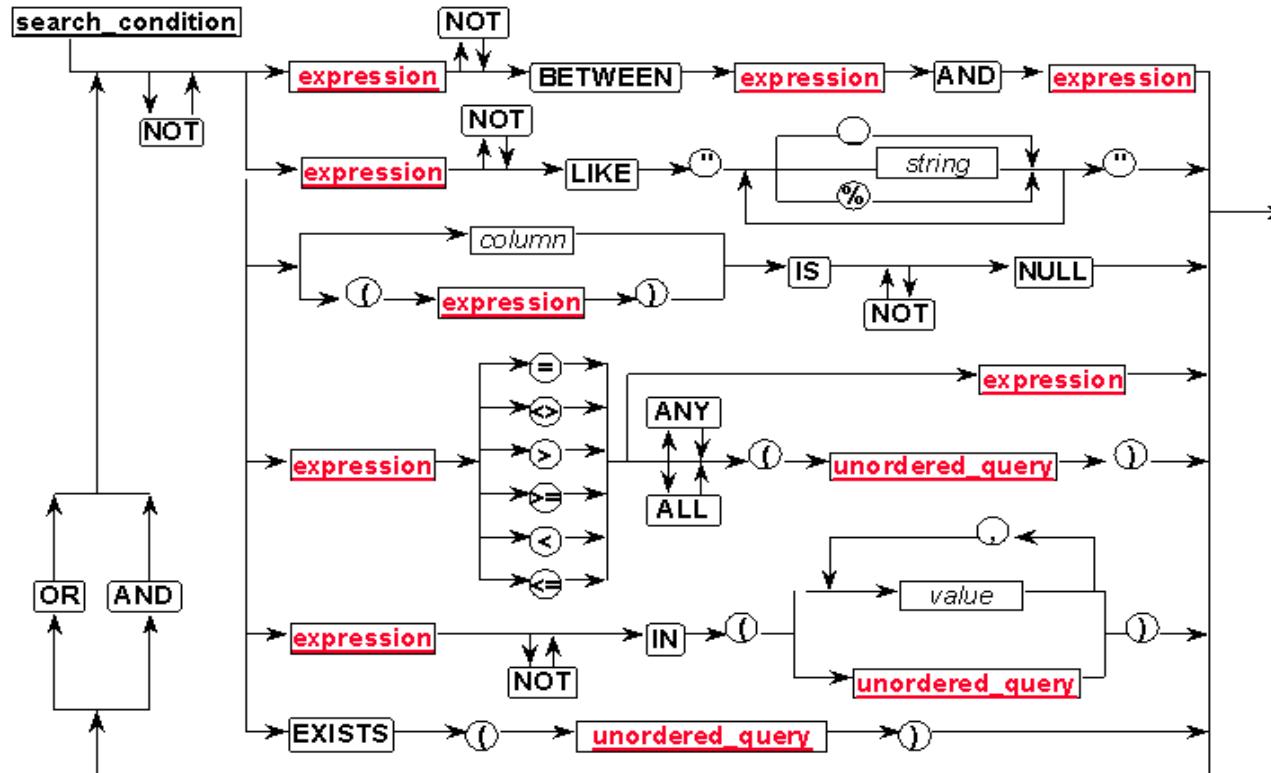
multiple times inside **search_condition**



SELECT – search condition

Context: SELECT ... FROM ... WHERE **search_condition**

SELECT ... FROM ... WHERE ... GROUP BY ... HAVING **search_condition**



Tables used in the examples

tabule

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	Torronto	116
KL7621	KLM	Rotterdam	75
KL1245	KLM	Amsterdam	130

table

Planes

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

SELECT ... FROM ...

- simplest query:
SELECT [ALL] | DISTINCT expression FROM table1, table2, ...
- the expression might contain
 - columns (* is surrogate for all columns)
 - constants
 - aggregation functions on expressions
 - if there is at least one aggregation, in the expression we can use only the aggregated columns (those specified in clause **GROUP BY**), while other columns must be „wrapped“ into aggregation functions
 - if the clause **GROUP BY** is not defined, the aggregation results in a single group (all the sources defined in the clause FROM is aggregated into single row)
 - **DISTINCT** eliminates suplicate rows in the output, **ALL** (default) allows duplicate rows in the output (note that this affects aggregations – with **DISTINCT** there is less values entering the aggregation)
- **FROM** contains one or more tables that serve as the source for the query
 - if there is multiple tables specified, the cartesian product is created

Examples – SELECT ... FROM ...

Which companies are in service?

SELECT DISTINCT 'Comp.:', Company FROM Flights

'Comp.:'	Company
Comp.:	CSA
Comp.:	Lufthansa
Comp.:	Air Canada
Comp.:	KLM

What plane pairs can be created (regardless the owner) and what is their capacity:

**SELECT L1.Plane, L2.Plane,
L1.Capacity + L2.Capacity
FROM Planes AS L1, Planes AS L2**

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

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

L1.Plane	L2.Plane	L1.Capacity + L2.Capacity
Boeing 717	Boeing 717	212
Airbus A380	Boeing 717	661
Airbus A350	Boeing 717	359
Boeing 717	Airbus A380	661
Airbus A380	Airbus A380	1110
Airbus A350	Airbus A380	803
Boeing 717	Airbus A350	359
Airbus A380	Airbus A350	803
Airbus A350	Airbus A350	506

Examples – SELECT ... FROM ...

How many companies are in service?

SELECT COUNT(DISTINCT Company) **FROM** Flights

COUNT(Company)

4

How many flights were accomplished?

SELECT COUNT(Company) **FROM** Flights

SELECT COUNT(*) **FROM** Flights

COUNT(Company), resp. COUNT(*)

7

How many planes is in the fleet, what is their max, min, average and total capacity?

SELECT COUNT(*), MAX(Capacity), MIN(Capacity), AVG(Capacity), SUM(Capacity) **FROM** Planes

COUNT(*)	MAX(Capacity)	MIN(Capacity)	AVG(Capacity)	SUM(Capacity)
3	555	106	304,666	914

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

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

SELECT ... FROM ... WHERE ...

- selection condition, i.e., table row (or row of a join) for which the condition is true gets into the result
 - simple conditions can be combined by **AND, OR, NOT**
- predicates
 - (in)equation predicate ($=$, \neq , $<$, $>$, \leq , \geq) on values of two columns
 - interval predicate – $\text{expr1 } [\text{NOT}] \text{ BETWEEN } (\text{expr2 } \text{AND } \text{expr3})$
 - string matching predicate **[NOT] LIKE "mask"**,
where **%** in the mask represents an arbitrary substring, and **_** represents an arbitrary character
 - NULL value predicate – **(expr1) IS [NOT] NULL**
 - set membership predicate – **expr1 [NOT] IN (unordered_query)**
 - empty set predicate – **EXISTS (unordered_query)**
 - extended quantifiers
 - existence quantifier **expr1 = | <> | < | > | \leq | \geq | ANY (unordered_query)**
 - i.e., **at least one** row in **unordered_query** results in true when compared with **expr1**
 - universal quantifier **expr1 = | <> | < | > | \leq | \geq | ALL (unordered_query)**
 - i.e., **all** rows in **unordered_query** result in true when compared with **expr1**

Examples – SELECT ... FROM ... WHERE ...

What flights are occupied by more than 100 passengers?

SELECT Flight, Passengers **FROM** Flights
WHERE Passengers > 100

Flight	Passengers
OK251	276
OK321	156
AC906	116
KL1245	130

Which planes are ready to fly if we require utilization of a plane at least 30%?

SELECT Flight, Plane,
(100 * Flights.Passengers / Planes.Capacity) **AS** Utilization **FROM** Flights, Planes
WHERE Flights.Company = Planes.Company **AND**
Flights.Passengers <= Planes.Capacity **AND**
Utilization >= 30

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

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

Flight	Plane	Utilization
OK012	Boeing 717	35
KL1245	Airbus A350	51
KL7621	Airbus A350	30

Examples – SELECT ... FROM ... WHERE ...

Which destinations can fly Airbus of any company (regardless utilization)?

SELECT DISTINCT Destinations **FROM** Flights

WHERE Flights.Company **IN** (**SELECT** Company **FROM** Planes

WHERE Plane **LIKE** "Airbus%")

or

SELECT DISTINCT Destination **FROM** Flights, Planes

WHERE Flights.Company = Planes.Company **AND** Plane **LIKE** "Airbus%"

Destination
Rotterdam
Amsterdam

Passengers of which flights fit any plane (regardless plane owner)?

SELECT * FROM Flights **WHERE** Passengers <= **ALL** (**SELECT** Capacity **FROM** Planes)

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

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

Flight	Company	Destination	Passengers
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
KL7621	KLM	Rotterdam	75

Joins

How to join without special language constructions (SQL 86):

- (inner) join on condition and natural join can be realized as restricted cartesian product, i.e.,
SELECT ... FROM table1, table2 **WHERE** table1.A = table2.B
- left/right semi-join is specified in clause **SELECT**, i.e., by projection

New SQL 92 constructions:

- cartesian product – **SELECT ... FROM** table1 **CROSS JOIN** table2 ...
- natural join – **SELECT ... FROM** table1 **NATURAL JOIN** table2 **WHERE** ...
- inner join –
SELECT ... FROM table1 **INNER JOIN** table2 **ON search_condition WHERE** ...
- union join – returns rows of the first table having NULLs in attributes of the second one
+ the same with second table
SELECT ... FROM table1 **UNION JOIN** table2 **WHERE** ...
- left, right, full outer join –
SELECT ... FROM table1 **LEFT | RIGHT | FULL OUTER JOIN** table2 **ON search_condition WHERE** ...

Examples – joins, ORDER BY

Get pairs flight-plane sorted in ascendant order w.r.t. unoccupied space in plane when the passengers board (consider just the planes the passengers can fit in)?

```
SELECT Flight, Plane, (Capacity - Passengers) AS Unoccupied FROM Flights INNER JOIN Planes  
ON (Flights.Company = Planes.Company AND Passengers <= Capacity)  
ORDER BY Unoccupied
```

Which flights cannot be operated (because the company does not own any/suitable plane)?

```
SELECT Flight, Destination FROM Flights LEFT OUTER JOIN Planes  
ON (Flights.Company = Planes.Company AND  
Passengers <= Capacity)  
WHERE Planes.Company IS NULL
```

Flight	Plane	Unoccupied
OK012	Boeing 717	69
KL1245	Airbus A350	123
KL7621	Airbus A350	178
KL1245	Airbus A380	425
KL7621	Airbus A380	480

Flight	Destination
OK251	New York
LH438	Stuttgart
OK321	London
AC906	Torronto

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

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

SELECT ... / GROUP BY ... HAVING ...

- aggregation clauses that group the rows of the result table after **FROM** and **WHERE** based on identical values in defined columns into columns groups
- the result is table of „superrows“, where the aggregating columns have defined values (as in the original rows), while the other columns must be created by aggregation (otherwise the values would be ambiguous)
- generalization of aggregating function shown earlier (**COUNT**, **MAX**, **MIN**, **AVG**, **SUM**), where the result is not single-row table (as in case of not using **GROUP BY**), but a table of as many rows as the number of superrows is (after **GROUP BY**)
- from this „superrow“ table we can filter out row based on **HAVING** clause, similarly as using **WHERE** after **FROM**
 - note that **HAVING** expression can only use aggregated columns as in **SELECT**

Examples – SELECT /.../ GROUP BY ... HAVING ...

What (positive) transport capacity have the companies?

SELECT Company, SUM(Capacity) FROM Planes GROUP BY Company

Company	SUM(Capacity)
CSA	106
KLM	803

What companies can fit all their passengers into their planes (regardless of their destinations)?

**SELECT Company, SUM(Passengers) FROM Flights
GROUP BY Company HAVING SUM(Passengers) <=**
(SELECT SUM(Capacity) FROM Planes WHERE Flights.Company = Planes.Company)

Company	SUM(Passengers)
KLM	205

Nested queries

- standard SQL92 (full) allows to use nested queries also in the **FROM** clause
 - while in SQL 86 the nested queries were allowed only in
ANY, ALL, IN, EXISTS
- two ways of application
 - **SELECT ... FROM (unordered_query) AS q1 WHERE ...**
 - allows selection right from the subquery result (instead of table/join)
 - subquery result is called **q1** while this identifier is used in other clauses as it would be a table
 - **SELECT ... FROM ((unordered_query) AS q1 CROSS | NATURAL | INNER | OUTER | LEFT | RIGHT JOIN (unordered_query) AS q2 ON (expression))**
 - usage in joins of all kinds

Examples – nested queries

Get sums of passengers and plane capacities for all companies that own a plane.

```
SELECT Flights.Company, SUM(Flights.Passengers), MIN(Q1.TotalCapacity)
FROM Flights, (SELECT SUM(Capacity) AS TotalCapacity, Company FROM Planes GROUP BY Company) AS Q1
WHERE Q1.Company = Flights.Company
GROUP BY Flights.Company;
```

Company	SUM(Passengers)	TotalCapacity
CSA	469	106
KLM	205	808

Get triplets of different flights for which the numbers of passengers differ by max 50.

```
SELECT Flights1.Flight, Flights1.Passengers, Flights2.Flight, Flights2.Passengers,
       Flights3.Flight, Flights3.Passengers
  FROM Flights AS Flights1
 INNER JOIN (Flights AS Flights2 INNER JOIN Flights AS Flights3 ON Flights2.Flight < Flights3.Flight)
            ON Flights1.Flight < Flights2.Flight
 WHERE abs(Flights1.Passengers - Flights2.Passengers) <= 50 AND
       abs(Flights2.Passengers - Flights3.Passengers) <= 50 AND
       abs(Flights1.Passengers - Flights3.Passengers) <= 50
 ORDER BY Flights1.Flight, Flights2.Flight, Flights3.Flight;
```

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

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

Flights1.Flight	Flights1.Passengers	Flights2.Flight	Flights2.Passengers	Flights3.Flight	Flights3.Passengers
AC906	116	KL1245	130	OK321	156
AC906	116	KL7621	75	LH438	68
KL7621	75	LH438	68	OK012	37

Limited number of rows

- often we want to return only first **k** rows (based on ORDER BY)
- slow version (SQL 92)
 - `SELECT ... FROM tab1 AS a
WHERE (SELECT COUNT(*)
 FROM tab1 AS b
 WHERE a.<ordering attribute> < b.<ordering attribute>) < k;`
- fast version (SQL:1999 non-core, implements Oracle, DB2)
 - `SELECT ... FROM (
 SELECT ROW_NUMBER() OVER (ORDER BY <ordering attribute(s)>
 ASC | DESC) AS rownumber
 FROM tablename) WHERE rownumber <= k`
- proprietary implementation (nonstandard)
 - `SELECT TOP k ... FROM ... ORDER BY <ordering attribute(s)> ASC | DESC`
 - MS SQL Server
 - `SELECT ... FROM ... ORDER BY <ordering attribute(s)> ASC | DESC LIMIT k`
 - MySQL, PostgreSQL

Example (MS-SQL notation)

What is the largest plane?

SELECT TOP 1 Plane FROM Planes ORDER BY Capacity DESC

Plane
Airbus A380

What are the two largest companies (w.r.t. sums of their passengers)?

SELECT TOP 2 Company, SUM(Passengers) AS Number FROM Flights GROUP BY Company ORDER BY Number DESC

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

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

Company	Number
CSA	469
KLM	205

course:

Database Systems (NDBlo25)

SS2017/18

lecture 4:

SQL – data definition & modification, views

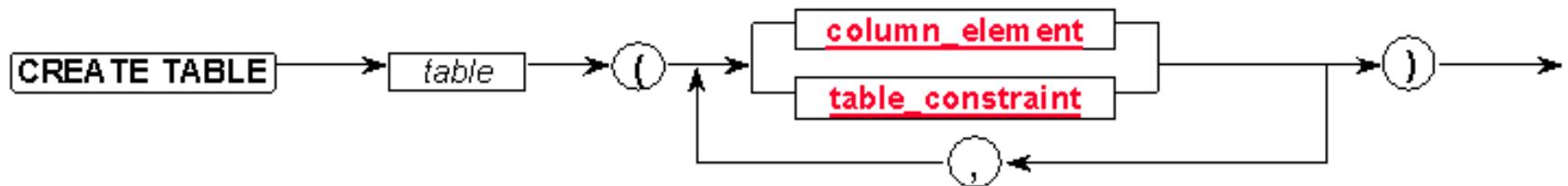
doc. RNDr. Tomáš Skopal, Ph.D.
RNDr. Michal Kopecký, Ph.D.

Today's lecture outline

- data definition
 - definition of tables (their schemes) and integrity constraints – CREATE TABLE
 - altering the definitions – ALTER TABLE
- data manipulation
 - INSERT, UPDATE, DELETE
- database views

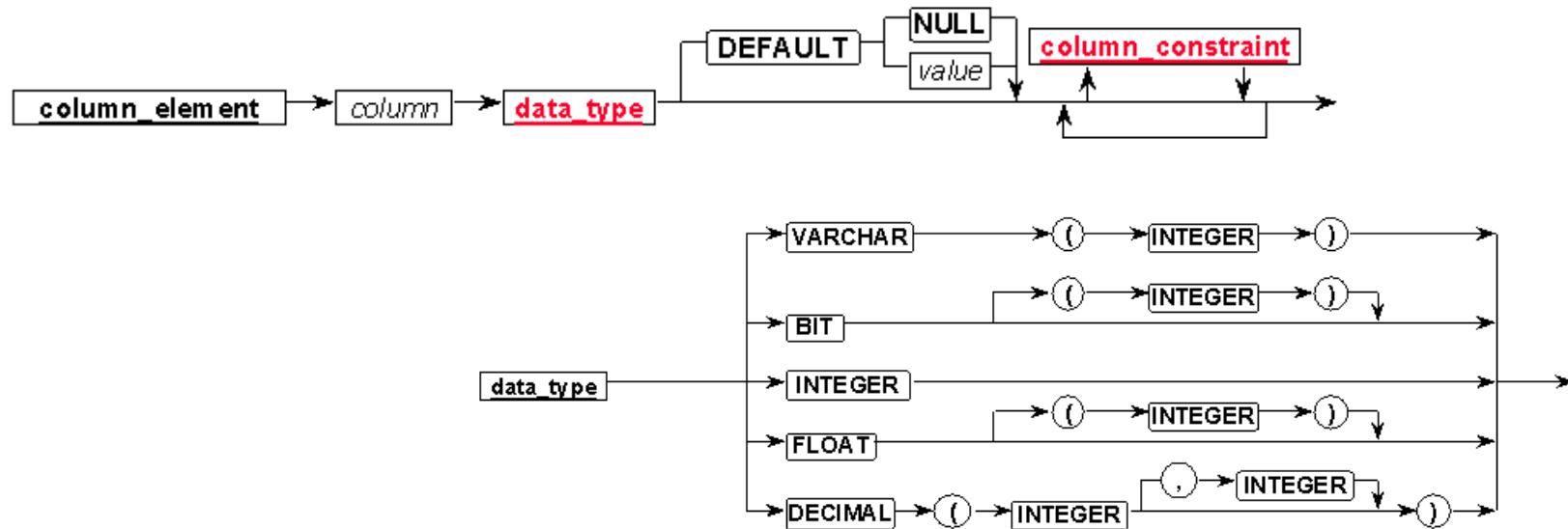
CREATE TABLE – basic construction

- construction of table schema and an empty table
- table name, columns defined, column-scope integrity constraints (IC), table-scope integrity constraints



CREATE TABLE – column definition

- each column has a data_type
- optionally
 - default value within a new record (DEFAULT NULL | value)
 - column-scope integrity constraints

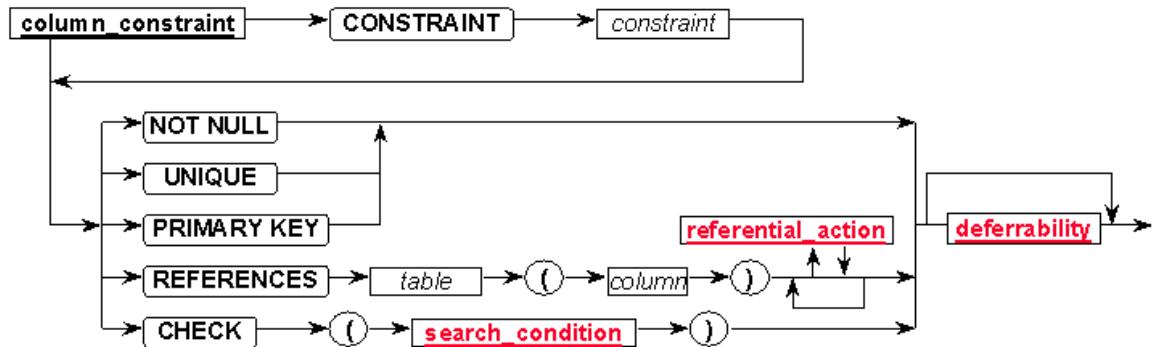


Example – simple table (w/o IC)

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

CREATE TABLE – column-scope integrity constraint

- column-based IC allows to limit the domain of values in column in record (new or modified)
 - named **CREATE TABLE ... (..., CONSTRAINT constraint ...)**
 - unnamed
- 5 types limiting a valid value
 - **NOT NULL** – value cannot be null
 - **UNIQUE** – value must be unique (w.r.t all records in the table)
 - **PRIMARY KEY** – primary key definition (key = **NOT NULL** + **UNIQUE**, primary = primary index)
 - **REFERENCES** – one-column foreign key (both columns must share definition)
 - **CHECK** – generic condition, similarly as in **SELECT ... WHERE**
 - applied only on the inserted/updated row(s)
 - data inserted/update only if TRUE
- if an IC is not validated, the record is not updated



Example – definition of tables with column-scope ICs

CREATE TABLE Product

(Id **INTEGER CONSTRAINT** pk **PRIMARY KEY**, Name **VARCHAR(128) UNIQUE**,
Price **DECIMAL(6,2) NOT NULL**, ProductionDate **DATE**, Available **BIT DEFAULT TRUE**, Weight **FLOAT**,
Producer **INTEGER REFERENCES Producer (Id)**)

CREATE TABLE Producer

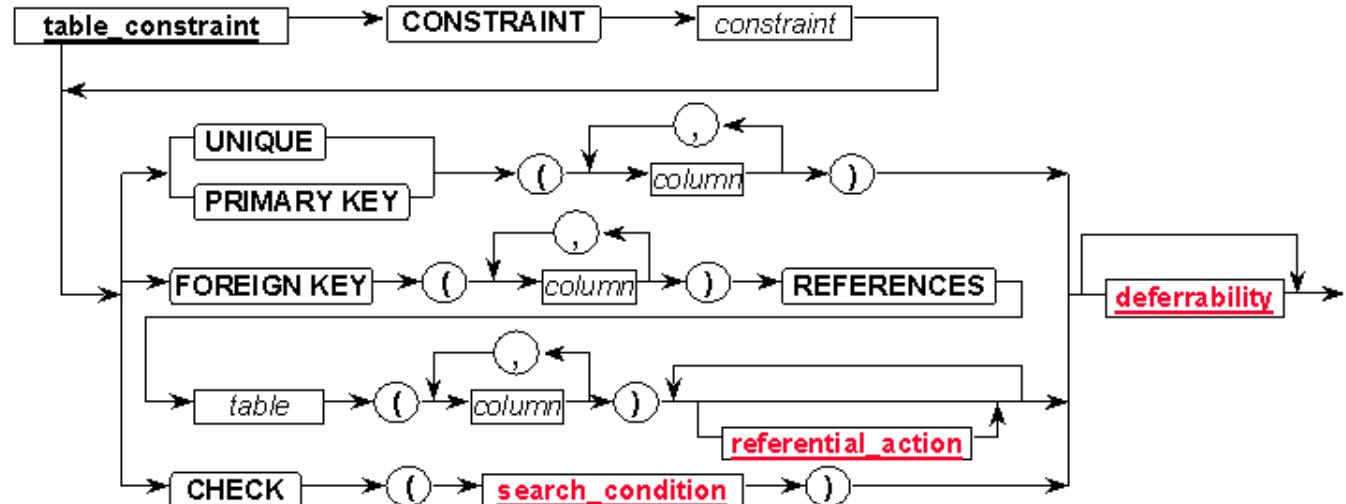
(Id **INTEGER PRIMARY KEY**, ProducerName **VARCHAR(128)**,
HQ **VARCHAR(256)**)

Example – foreign key (single table)

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

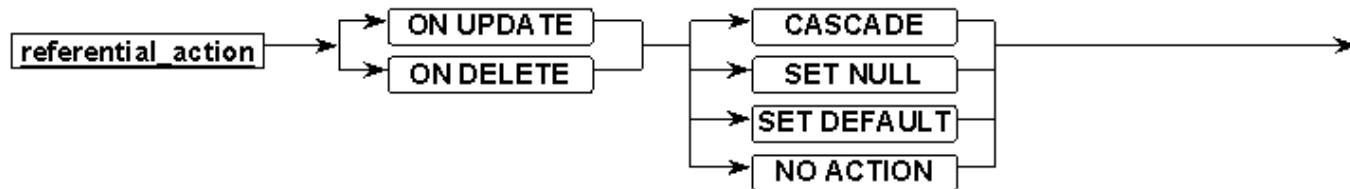
CREATE TABLE – table-scope integrity constraint

- generalization of column-scope IC to use for multiple columns
 - apart NOT NULL, this is only meaningful for single column
- **UNIQUE** – n-tuple of values is unique
- **FOREIGN KEY** – the same as **REFERENCES** in column-scope IC
- **CHECK**



Referential integrity

- when updating **referencing** or **referenced** table, violation of foreign keys may occur
 - record update with foreign key value that is not valid (not present in column of referenced table)
 - delete/update of a record in the referenced table (so that value in referencing table becomes not covered)
- when a violation of foreign keys occurs, there are two options
 - if there is no referential action defined, an error message appears, aborting the action (SQL 89)
 - referential action is triggered, **referential_action** (SQL 92)
 - **ON UPDATE**, **ON DELETE** – definition of when to trigger the action, either by update or delete of row in the referenced table
 - **CASCADE** – the row with the referencing value is treated the same (i.e., updated or deleted)
 - **SET NULL** – referencing value in the row is set to NULL
 - **SET DEFAULT** – referencing value in the row is set to a default value defined in CREATE TABLE
 - **NO ACTION** – default, no action takes place, resp., DBMS displays a message (SQL 89)



Example – table with table-scope IC

CREATE TABLE Product

(Id **INTEGER PRIMARY KEY**, Name **VARCHAR(128) UNIQUE**, Price
DECIMAL(6,2) NOT NULL, ProductionDate **DATE**, Available **BIT DEFAULT TRUE**, Weight **FLOAT**, Producer **VARCHAR(128)**, ProducerHQ **VARCHAR(256)**,
CONSTRAINT fk FOREIGN KEY (Producer, ProducerHQ) REFERENCES
Producer (ProducerName,HQ))

CREATE TABLE Producer

(ProducerName **VARCHAR(128)**, HQ **VARCHAR(256)**, Subject **VARCHAR(64)**,
CONSTRAINT pk PRIMARY KEY(ProducerName, HQ))

Example – CHECK

```
CREATE TABLE Product
(Id INTEGER PRIMARY KEY, Name VARCHAR(128) UNIQUE, Price
DECIMAL(6,2) NOT NULL, ProductionDate DATE, Available BIT DEFAULT
TRUE, Weight FLOAT,
CONSTRAINT chk CHECK
(Weight > 0))
```

Example – ON DELETE, ON UPDATE

CREATE TABLE Product

(Id **INTEGER CONSTRAINT** pk **PRIMARY KEY**, Name **VARCHAR(128) UNIQUE**,
Price **DECIMAL(6,2) NOT NULL**, ProductionDate **DATE**, Available **BIT DEFAULT**
TRUE, Weight **FLOAT**,
Producer **INTEGER REFERENCES** Producer (Id)
ON DELETE CASCADE)

CREATE TABLE Producer

(Id **INTEGER PRIMARY KEY**, ProducerName **VARCHAR(128)**,
HQ **VARCHAR(256)**)

ALTER TABLE

- table scheme definition altering
 - column – add/remove column, change of DEFAULT value
 - IC – add/remove IC
- however, there may be data that won't allow the IC change (e.g., primary key definition)

ALTER TABLE table-name

... **ADD [COLUMN]** column-name column-definition
... **ADD** constraint-definition
... **ALTER [COLUMN]** column-name SET
... **ALTER [COLUMN]** column-name DROP
... **DROP COLUMN** column-name
... **DROP CONSTRAINT** constraint-name

Example – ALTER TABLE

CREATE TABLE Product

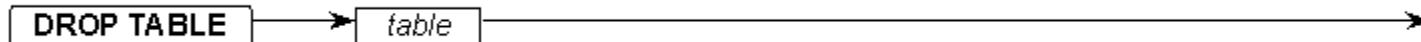
(Id **INTEGER PRIMARY KEY**, Name **VARCHAR(128) UNIQUE**, Price
DECIMAL(6,2) NOT NULL, ProductionDate **DATE**, Available **BIT DEFAULT TRUE**,
Weight **FLOAT**,
CONSTRAINT chk **CHECK**
(AND Weight > 0))

ALTER TABLE Product **DROP CONSTRAINT** chk

ALTER TABLE Product **ADD CONSTRAINT** chk **CHECK**
(Weight > 10)

DROP TABLE

- **DROP TABLE *table***
- complementary to **CREATE TABLE *table***
- the table content is deleted and also the definition (i.e., scheme)
 - if we need to delete only the content we use **DELETE FROM *table*** (see next)

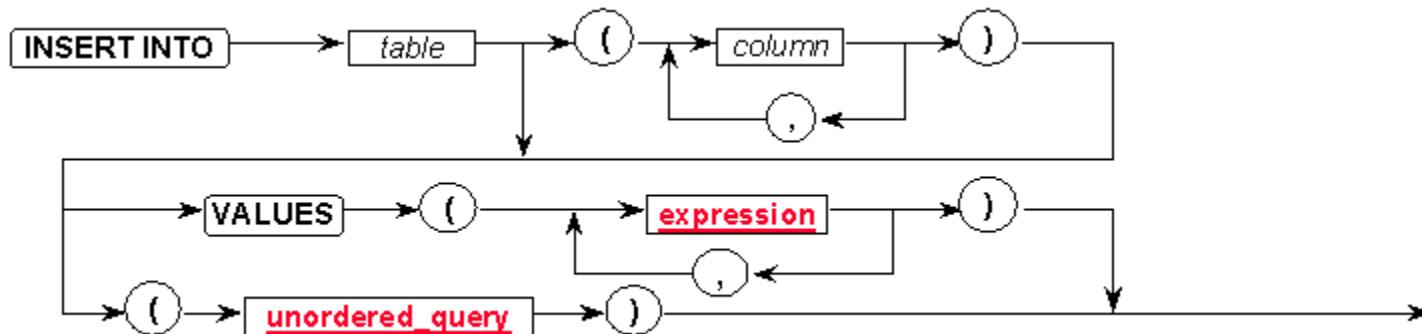


Data modification

- apart SELECT, the SQL DML offers three commands for data manipulation
 - INSERT INTO – insertion of rows
 - DELETE FROM – deletion of rows
 - UPDATE – update of rows

INSERT INTO

- insertion of row by enumeration, two options
 - **INSERT INTO** table (col₁, col₃, col₅) **VALUES** (val₁, val₃, val₅)
 - **INSERT INTO** table **VALUES** (val₁, val₂, val₃, val₄, val₅)
- insertion of multiple rows that result from a SELECT
 - **INSERT INTO** table1 | (column list) | (SELECT ... FROM ...)



Example – INSERT INTO

CREATE TABLE Product

(Id **INTEGER CONSTRAINT** pk **PRIMARY KEY**, Name **VARCHAR(128)**
UNIQUE, Price **DECIMAL(6,2) NOT NULL**, ProductionDate **DATE**, Available
BIT DEFAULT TRUE, Weight **FLOAT**,
Producer **INTEGER REFERENCES** Producer (Id))

INSERT INTO Product **VALUES** (0, 'Chair', 86, '2005-5-6', TRUE, 3, 123456)

INSERT INTO Product (Id, Name, Price, ProductionDate, Weight, Producer)
VALUES (0, 'Chair', 86, '2005-5-6', 3, 123456)

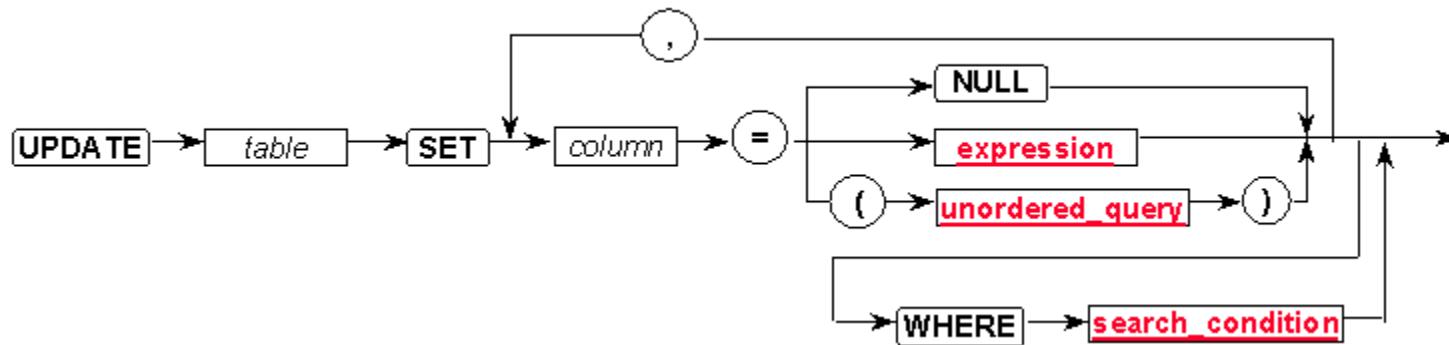
Example – INSERT INTO

```
CREATE TABLE ProductInStore  
  (Id INTEGER PRIMARY KEY, Name VARCHAR(128) UNIQUE, Price  
DECIMAL(6,2))
```

```
INSERT INTO ProductInStore  
  (SELECT Id, Name, Price FROM Product WHERE Available = TRUE)
```

UPDATE

- update of rows matching a condition
- the values of chosen columns are set to
 - NULL
 - value given by expression (also constant)
 - query result



Example – UPDATE

UPDATE Product **SET** Name = 'Notebook' **WHERE** Name = 'Laptop'

UPDATE Product **SET** Price = Price * 0.9
WHERE ProductionDate < **CAST**('2017-01-01' **AS** DATE)

UPDATE Product **SET** Price = Price * 0.9
WHERE ProductionDate < '2017-01-01' /* automatic conversion */

UPDATE Product **AS** V1 **SET**
Weight = (**SELECT** AVG(Weight)
FROM Product **AS** V2
WHERE V1.Name = V2.Name)

DELETE FROM

- deletes rows that match condition
- **DELETE FROM *table*** deletes all rows

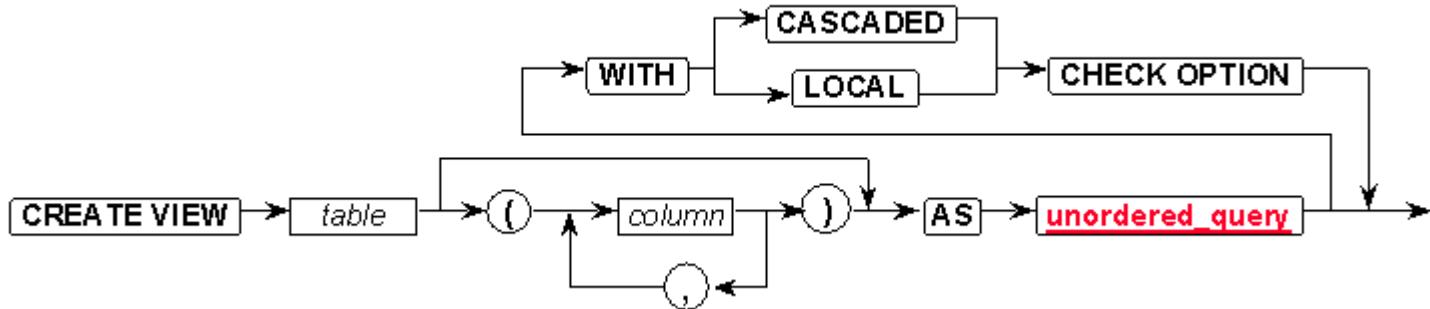
Example:

DELETE FROM Product WHERE Price > 100



Database views

- named query that could be used as a table (its result, resp.)
- evaluated dynamically
 - could be used also for data inserts/updates/deletes
- CHECK OPTION ensures that after row insert/update into the view the change will be visible
 - at least in this view
 - in all dependent views



Example – views

CREATE VIEW NewProductInStore **AS**

SELECT * FROM Product

WHERE Available = TRUE

AND ProductionDate > **CAST**('2017-01-01' **AS** DATE)

WITH LOCAL CHECK OPTION

INSERT INTO NewProductInStore **VALUES**

(0, 'SwingChair', 135, '2017-05-06', TRUE, 4, 3215)

- inserted (transitively into table Product)

INSERT INTO NewProductInStore **VALUES**

(0, 'Box', 135, '1999-11-07', TRUE, 4, 3215)

!! Error – this record cannot be inserted, as it wouldn't be visible (too old box)

course:

Database Systems (NDBI025)

SS2017/18

lecture 5:

SQL – embedded SQL, external applications, SQL/XML

doc. RNDr. Tomáš Skopal, Ph.D.

doc. RNDr. Irena Mlýnková, Ph.D.

RNDr. Michal Kopecký, Ph.D.

Today's lecture outline

- embedded SQL – „internal“ database applications
 - stored procedures
 - cursors
 - triggers
- SQL/XML
 - XML data type + functions integrated into SQL SELECT
- „external“ database applications (using interfaces/libraries)
 - ODBC, JDBC, ADO.NET
 - object-relational mapping libraries
 - Java Hibernate

Programming in embedded SQL

- procedural extensions of SQL
 - std. SQL is a subset (i.e., that's why embedded)
 - MS SQL Server – Transact SQL (T-SQL)
 - Oracle – PL/SQL
- benefits
 - controlling statements (if-then, for, while)
 - cannot be just scripted
 - cursors (iterative scan of tables)
 - smaller networking overhead (code on server), pre-compiled
 - triggers – general integrity constraints
 - better security (access rights control for server code)
- cons
 - proprietary extensions, cannot be simply transferred
 - SQL 1999 standard, but not respected by industry much

Structure (MS SQL Server)

DECLARE section
BEGIN ... END

E.g.:

```
DECLARE @avg_age FLOAT
BEGIN
    SELECT @avg_age = AVG(age) FROM Employee
END
```

Stored procedures (MS SQL Server)

CREATE PROCEDURE *procname* [*; number*]

[*declaration_parameter* [, ...]]

[WITH RECOMPILE]

AS commands [;]

- parameter declaration
 - *@name type [= expression] [OUT[PUT]]*
 - OUT[PUT] parameter is output
- *number* allows multiple versions of the same procedure
- procedure call
 - EXEC[UTE] *procname* [*expression* [, ...]]
 - parameters are passed w.r.t. order
 - EXEC[UTE] *procname* [*@name=expression* [, ...]]
 - parameters are passed w.r.t. names

Stored procedures, example

```
CREATE PROCEDURE Payment
    @accSource VARCHAR(25),
    @accTarget VARCHAR(25),
    @amount INTEGER = 0
AS
BEGIN
    UPDATE Accounts SET balance = balance - @amount
        WHERE account=@accSource;

    UPDATE Accounts SET balance = balance + @amount
        WHERE account=@accTarget;
END

EXEC Payment '21-87526287/0300', '78-9876287/0800', 25000;
```

Stored procedures (MS SQL Server)

```
CREATE FUNCTION funcname [; number]
([declaration_parameter [, ...]]) RETURNS type
[WITH RECOMPILE]
AS commands [;]
```

- parameter declaration
 - *@name type [= expression] [OUT[PUT]]*
 - OUT[PUT] parameter is output
- *number* allows multiple versions of the same procedure
- function use
 - *schemaname.funcname([expression [, ...]])*
 - parameters are passed w.r.t. order
 - *schemaname.funcname([@name=expression [, ...]])*
 - parameters are passed w.r.t. names

Stored procedures, example

```
CREATE FUNCTION AccBalance(  
    @acc VARCHAR(25)  
) RETURNS INTEGER  
AS  
DECLARE @ret INTEGER;  
BEGIN  
    SELECT @ret =balance  
        FROM Accounts  
        WHERE account=@acc;  
  
    RETURN @ret;  
END
```

```
SELECT MySchema.AccBalance(account) AS bal FROM ...
```

Cursors (MS SQL Server)

- declaration
 - **C [SCROLL] CURSOR FOR**
SELECT ...;
- data retrieval
 - **FETCH**
{NEXT | PRIOR | ABSOLUTE n | RELATIVE n | LAST | FIRST}
FROM C
[INTO @variable [, ...]]
 - If cursor not declared using SCROLL, only NEXT is allowed

Cursors, example (tax payment)

```
DECLARE
    Cur CURSOR FOR
        SELECT *
        FROM Accounts;
BEGIN
    OPEN Cur
    DECLARE @acc varchar(25), @bal int;
    FETCH NEXT FROM Cur INTO @acc, @bal;
    WHILE @@FETCH_STATUS=0
        BEGIN
            Payment(@acc, '21-87526287/0300', @bal*0.01)
            FETCH NEXT FROM Cur INTO @acc, @bal;
        END;
    CLOSE Cur;
    DEALLOCATE Cur;
END
```

Triggers – DML triggers

- event-executed stored procedure (table/view event)
- allows to extend integrity constraint logics
 - *inserted, deleted* – logical tables with the same structure as the table the trigger is bound on

```
CREATE TRIGGER trigger_name ON { table | view }
[WITH ENCRYPTION]
{FOR | AFTER | INSTEAD OF}
{[ INSERT ][,][ UPDATE ][,][ DELETE ]}
[WITH APPEND]
AS
[ {IF UPDATE ( column ) [{ AND | OR } UPDATE ( column ) ] ...
  | IF ( COLUMNS_UPDATED ( bitwise_operator updated_bitmask
  ))}
sql_statement [...]
```

DML triggers (example)

```
CREATE TRIGGER LowCredit ON PurchaseOrderHeader
AFTER INSERT
AS
DECLARE @error_count int

SELECT @error_count = count (*)
    FROM inserted i JOIN Purchasing.Vendor v on v.VendorID = i.VendorID
    WHERE v.CreditRating=5

IF @error_count > 0
BEGIN
    RAISERROR ('Vendor''s credit rating is too low to accept new purchase orders.', 16, 1)
    ROLLBACK TRANSACTION
END
```

DML triggers (example)

```
CREATE TRIGGER LowCredit ON PurchaseOrderHeader
AFTER INSERT
AS
IF EXISTS(
    SELECT *
    FROM inserted i JOIN Vendor v on (v.VendorID = i.VendorID)
    WHERE v.CreditRating=5
)
BEGIN
    RAISERROR ('Vendor''s credit rating is too low to accept new purchase orders.', 16, 1)
    ROLLBACK TRANSACTION
END
```

DDL triggers

```
CREATE TRIGGER trigger_name ON { ALL SERVER | DATABASE }
[WITH <ddl_trigger_option> [,...n]]
{ FOR | AFTER } { event_type | event_group } [,..n] AS
{sql_statement [ ; ] [,..n] | EXTERNAL NAME <method specifier> [ ; ] }
<ddl_trigger_option> ::= [ ENCRYPTION ] [ EXECUTE AS Clause ]
<method_specifier> ::= assembly_name.class_name.method_name
```

DDL triggers (example)

```
CREATE TRIGGER safety ON DATABASE FOR
    DROP_SYNONYM
AS
    RAISERROR (
        'You must disable Trigger "safety" to drop synonyms!',
        10, 1
    )
    ROLLBACK
```

What is SQL/XML

- An extension of SQL for XML data (SQL 2003)
 - New built-in data type called **XML**
 - Querying over XML data
- Note: SQL/XML ≠ SQLXML
 - SQLXML is Microsoft technology in MS SQL Server (not standard)
 - Similar strategy, but different approach
 - Not a standard
- The key aspect is an **XML value**
 - Intuitively: an XML element or a set of XML elements
 - Its semantics results from XML InfoSet data model
 - Standard of XML data model = tree consisting of nodes representing elements, attributes, text values, ...
 - graph formalism for XML

SQL/XML – XML Data Publishing

- Generating of XML data (of type XML) from relational data
 - **XMLELEMENT** – creating an XML element
 - **XMLATTRIBUTES** – creating XML attributes
 - **XMLFOREST** – creating a sequence of XML elements
 - **XMLCONCAT** – concatenation of XML values into a single value
 - **XMLAGG** – creating a sequence of elements from a group of rows

SQL/XML – XMLELEMENT

```
Employees (id, first, surname, dept, start)
```

- Creating an XML value for:
 - Name of an element
 - (Optional) list of attribute declarations
 - (Optional) list of expressions declaring element content

```
SELECT E.id,  
       XMLELEMENT ( NAME "emp",  
                     E.first || ' ' || E.surname ) AS xvalue  
FROM Employees E WHERE ...
```

id	xvalue
1001	<emp>George Clooney</emp>
...	...

XMLELEMENT – subelements

```
SELECT E.id,  
       XMLELEMENT ( NAME "emp",  
                     XMLELEMENT ( NAME "name",  
                                   E.first || ' ' || E.surname ),  
                     XMLELEMENT ( NAME "date", E.start )  
               ) AS xvalue  
FROM Employees E WHERE ...
```

id	xvalue
1001	<emp> <name>George Clooney</name> <date>2000-05-24</date> </emp>
...	...

SQL/XML – XMLATTRIBUTES

```
SELECT E.id,  
       XMLEMENT ( NAME "emp",  
                  XMLATTRIBUTES ( E.id AS "empid" ) ,  
                  E.first || ' ' || E.surname ) AS xvalue  
FROM Employees E WHERE ...
```

id	xvalue
1001	<emp empid="1001">George Clooney</emp>
...	...

SQL/XML – XMLFOREST

```
SELECT E.id,  
       XMLELEMENT ( NAME "emp",  
                     XMLFOREST (  
                         E.first || ' ' || E.last AS "name",  
                         E.start AS "date" ) ) AS xvalue  
FROM Employees E  
WHERE ...
```

id	xvalue
1001	<emp> <name>George Clooney</name> <date>2000-05-24</date> </emp>
...	...

SQL/XML – XMLAGG

- **XMLAGG** is an aggregation function combined with **GROUP BY**
 - Similarly to SUM, AVG etc.
- **XMLAGG** accepts only XML expressions (values)
- The expression is evaluated for each row in group **G** created by the GROUP BY expression
- The results are concatenated into a single resulting value
- The result can be sorted using **ORDER BY**

SQL/XML – XMLAGG

```
SELECT XMLEMENT (
    NAME "dept",
    XMLATTRIBUTES (E.dept AS "name"),
    XMLAGG (
        XMLEMENT (NAME "emp", E.surname)
        ORDER BY E.surname )
    ) AS xvalue
FROM Employees E
GROUP BY E.dept
```

xvalue

```
<dept name="hr">
    <emp>Clooney</emp>
    <emp>Pitt</emp>
</dept>

<dept name="accountant">
    ...
</dept>
```

XML Type

- Type XML can be used at same places as SQL data types (e.g., NUMBER, VARCHAR, ...)
 - Column type, SQL variable, ...
- The XML type can be
 - Queried ([XMLQUERY](#))
 - Transformed into relational data ([XMLTABLE](#))
 - Tested ([XMLEXISTS](#))

Sample Data – Table EmpXML

id	ColEmpXML
1001	<emp> <first>George</first> <surname>Clooney</surname> <date>2000-05-24</date> <dept>hr</dept> </emp>
1006	<emp> <first>Brad</first> <surname>Pitt</surname> <date>2001-04-23</date> <dept>hr</dept> </emp>
...	...

XMLQUERY

```
SELECT
    XMLQUERY (
        'for $p in $col/emp return $p/surname' ,
        PASSING EmpXML.ColempXML AS "col"
        RETURNING CONTENT NULL ON EMPTY ) AS result
FROM EmpXML WHERE ...
```

result
<surname>Clooney</surname>
<surname>Pitt</surname>
...

XMLTABLE

```
SELECT result.*  
FROM EmpXML, XMLTABLE ( XQuery query  
    'for $p in $col/emp return $p',  
    PASSING EmpXML.CollEmpXML AS "col"  
    COLUMNS firstname VARCHAR(40) PATH 'first'  
                      DEFAULT 'unknown',  
        lastname  VARCHAR(40) PATH 'surname'  
) AS TableResult
```

XPath expressions over XML data returned by the query

TableResult

firstname	lastname
George	Clooney
Brad	Pitt
unknown	Banderas

Assumption: We do not know first name of Banderas.

XMLEXISTS

```
SELECT id  
FROM EmpXML  
WHERE  
    XMLEXISTS ( '/emp/date lt "2001-04-23"'  
    PASSING BY VALUE EmpXML.ColempXML )
```

XPath query

id
1001
1006
...

Database applications

- DBI026
 - Oracle and MS SQL Server (alternatives)
 - embedded SQL, administration
 - external applications
 - indexing, optimisations
 - transactions
 - security
 - see
<http://www.ms.mff.cuni.cz/~kopecky/vyuka/dbapl/>

External database programming

- external/standalone applications (i.e., outside DBMS environment) use standardized interfaces
 - ODBC (Open DataBase Connectivity)
 - 1992, Microsoft
 - JDBC (Java DataBase Connectivity)
 - using ODBC (mostly), or native driver/protocol, network driver
 - ADO.NET library (Active Data Objects .NET)
 - over OLE DB, ODBC, or directly drivers to MS SQL Server, Oracle
 - higher-level, faster and more reliable (than ODBC)
- „seminative“ database object oriented programming using object-relational mapping
 - Java Hibernate
 - the same for Microsoft .NET

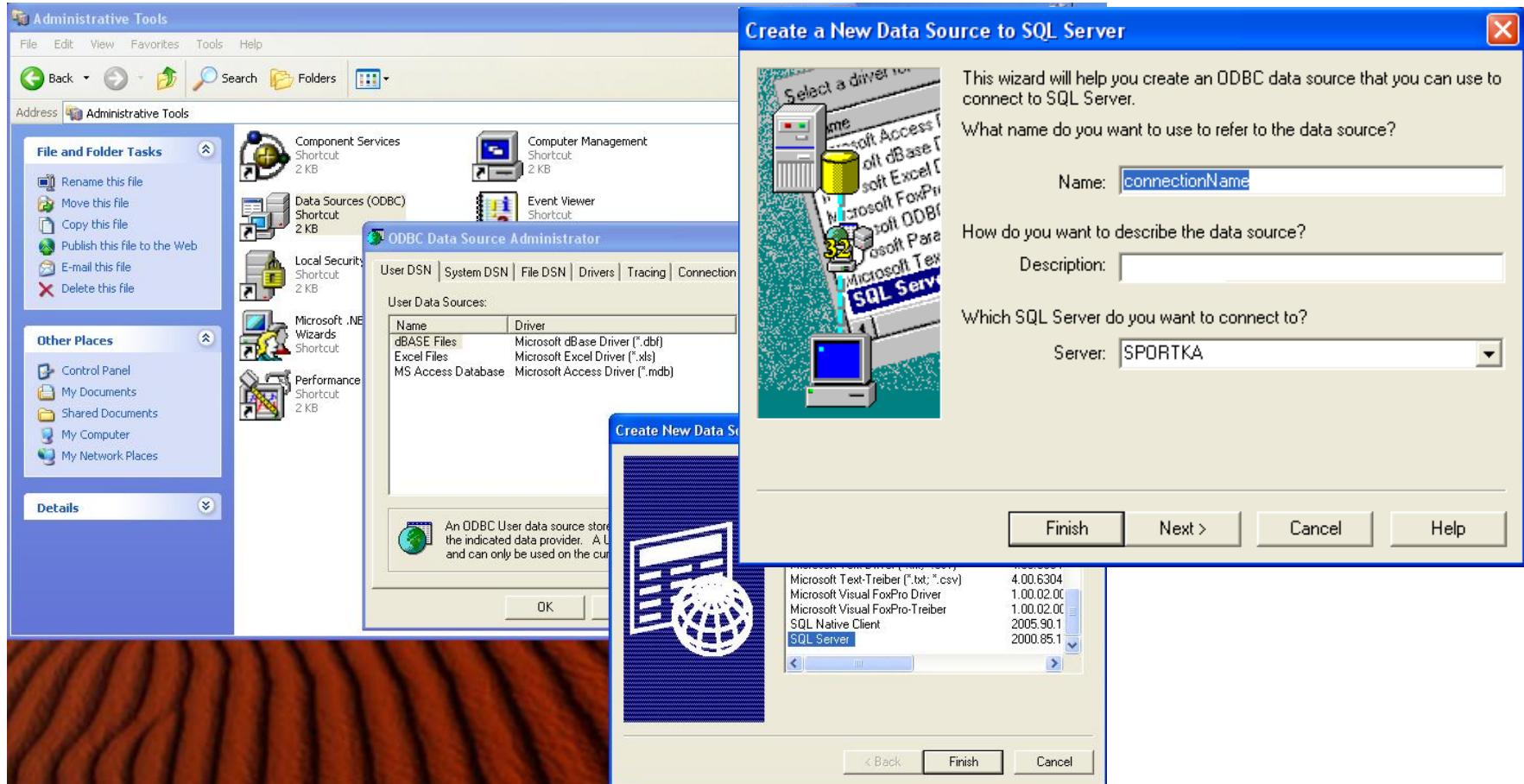
Java Hibernate Framework

- provides persistence to Java objects, i.e., provides „real“ object oriented database programming
- mapping definition between object and its persistent state in database necessary (xml file for every class)
- simplification: memory manager organizes objects directly in database (+ uses main memory as a cache when accessing object)
- HQL (Hibernate query language)
 - object query language
 - Hibernate translates HQL into SQL

Relational vs. object approach

- object-to-tables mapping brings some overhead, not visible to the user (is both good and bad)
 - implementation of object DBMS using relational DBMS
- relational DBMS suitable for data-intensive applications and batch nature (uniform actions over many instances)
 - here object DBMS would be inefficient due to creation of many small objects that are created uniformly, i.e., individual „materialization“ into general objects is not necessary
- object DBMS suitable for „Enterprise applications“, where DBMS performance is not the bottleneck
 - relational DBMS provide low-level access to data, i.e., programmer forced to manage the database

ODBC, Windows configuration



ODBC, application (C#)

```
using System.Data.Odbc;
OdbcConnection DbConnection = new OdbcConnection("DRIVER={SQL
    Server};SERVER=MyServer;Trusted_connection=yes;DATABASE=northwind; ");
DbConnection.Open();
OdbcCommand DbCommand = DbConnection.CreateCommand();
DbCommand.CommandText = "SELECT * FROM Employee";
OdbcDataReader DbReader = DbCommand.ExecuteReader();

int fCount = DbReader.FieldCount;

while( DbReader.Read() ){
    Console.Write(":");
    for (int i = 0; i < fCount; i++) {
        String col = DbReader.GetString(i); Console.Write(col + ":");
    }
    Console.WriteLine();
}

DbReader.Close(); DbCommand.Dispose(); DbConnection.Close();
```

JDBC, application (Java)

```
Class.forName( "com.somejdbcvendor.TheirJdbcDriver" );  
  
Connection conn = DriverManager.getConnection( "jdbc:somejdbcvendor:other data needed by  
some jdbc vendor", "myLogin", "myPassword" );  
  
Statement stmt = conn.createStatement();  
  
try {  
    stmt.executeUpdate( "INSERT INTO MyTable( name ) VALUES ( 'my name' )" );  
}  
finally { stmt.close(); }
```

ADO.NET, application (C#)

using ODBC:

```
SqlConnection connection = new  
    SqlConnection("server=localhost;database=myDatabase;uid=sa;pwd=");
```

using OLE DB (sqloledb = SQL Server, msdaora = Oracle):

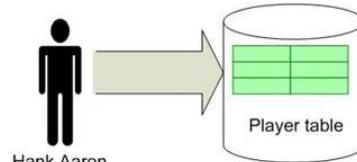
```
OleDbConnection connection = new OleDbConnection  
    ("provider=sqloledb;server=localhost;database='"+myDatabase;uid=sa;pwd=");  
  
connection.Open();  
  
SqlCommand command = new SqlCommand("SELECT * FROM table", connection);  
command.ExecuteNonQuery();
```

Java Hibernate – example

```
public class BallPlayer {  
    private Long id;  
    private String name;  
    private String nickname;  
    private Calendar dob;  
    private String birthCity;  
    private short uniformNumber;  
    //getter and setter methods removed here for brevity.  
}
```

BallPlayer	
id	
name	
nickname	
dob	
birthCity	
uniformNumber	

```
Calendar dob = Calendar.getInstance();  
dob.set(1934, Calendar.FEBRUARY, 5);  
BallPlayer player = new BallPlayer();  
player.setName("Henry Louis Aaron");  
player.setNickname("Hammerin' Hank");  
player.setDob(dob);  
player.setBirthCity("Mobile, AL");  
player.setUniformNumber((short) 44);
```



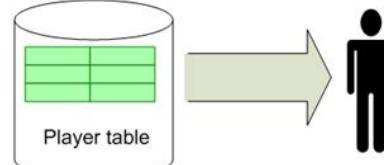
```
SessionFactory sessionFactory = new  
Configuration().configure(). buildSessionFactory();  
Session session = sessionFactory.openSession();  
Transaction transaction = session.beginTransaction();  
session.save(player);  
transaction.commit();  
session.close();
```



```
INSERT INTO PLAYER VALUES(5,'Henry Louis Aaron','Hammerin''  
Hank','1934-02-05','Mobile, AL',44)
```

```
create table player (  
    id integer primary key,  
    name varchar not null,  
    nickname varchar,  
    date_of_birth date,  
    city_of_birth varchar,  
    uniform_number integer  
)
```

Player	
PK	id
	name
	nickname
	date_of_birth
	city_of_birth
	uniform_number



```
SessionFactory sessionFactory = new  
Configuration().configure(). buildSessionFactory();  
Session session = sessionFactory.openSession();  
Transaction transaction = session.beginTransaction();  
BallPlayer aPlayer = (BallPlayer) session.get(BallPlayer.class, 1L);  
transaction.commit();  
session.close();
```



```
select ballplayer0_.id as id0_0_, ballplayer0_.name as name0_0_,  
ballplayer0_.nickname as nickname0_0_, ballplayer0_.date_of_birth as  
date4_0_0_, ballplayer0_.city_of_birth as city5_0_0_,  
ballplayer0_.uniform_number as uniform6_0_0_ from Player ballplayer0_  
where ballplayer0_.id=1
```

Java Hibernate – example

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="dialect"> org.hibernate.dialect.HSQLDialect</property>
        <property name="connection.driver_class"> org.hsqldb.jdbcDriver</property>
        <property name="connection.url"> jdbc:hsqldb:hsql:
//localhost/baseballdb</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>
        <property name="show_sql">true</property>
        <mapping resource="com/intertech/domain/BallPlayer.hbm.xml" />
    </session-factory>
</hibernate-configuration>

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="com.intertech.domain.BallPlayer" table="Player">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">common_seq</param>
            </generator>
        </id>
        <property name="name" />
        <property name="nickname" />
        <property name="dob" column="date_of_birth" />
        <property name="birthCity" column="city_of_birth" />
        <property name="uniformNumber" column="uniform_number" />
    </class>
</hibernate-mapping>
```

Equivalents of Java Hibernate for Microsoft.NET Framework

- ADO.NET Entity Framework
- NHibernate
- Persistor.NET
- etc.

course:

Database Systems (NDBlo25)

SS2017/18

lecture 6:

Query formalisms for relational model – relational algebra

doc. RNDr. Tomáš Skopal, Ph.D.

RNDr. Michal Kopecký, Ph.D.

Today's lecture outline

- relational algebra
 - relational operations
 - equivalent expressions
 - relational completeness

Querying in relational model

- the main purpose of a database is to provide access to data (by means of querying)
 - query language needed
 - query language should be strong enough to select any (meaningful) subset of the database
 - symbols defined in schemas stand for the basic constructs in the query language

Database query

- query = delimitation of particular set of data instances
 - a single query may be expressed by multiple expressions of the query language – **equivalent expressions**
- query extent (power of the query language)
 - in **classic models**, only subset of the database is expected as a query result (i.e., values actually present in the database tables)
 - in **extended models**, also derived data can be returned (i.e., computations, statistics, aggregations derived from the data)

Query language formalisms

- as “table data” model is based on the relational model, there can be used well-known formalisms
 - **relational algebra** (today lecture)
(operations on relations used as query constructs)
 - **relational calculus** (next lecture)
(database extension of the first-order logic used as a query language)

Relational algebra (RA)

- RA is a set of operations (unary or binary) on relations with schemes; their results are also relations (and schemes)
 - for completeness, to a relation (table contents) R^* we always consider also a scheme $R(A)$ consisting of name and (typed) attributes, i.e., a tuple $\langle R^*, R(A) \rangle$
- a scheme will be named by any unique user-defined identifier
 - for relation resulting from an operation we mostly do not need to define a name for the relation and the scheme – it either enters another operation or is the final result
 - if we need to “store” (or label) the result, e.g., for decomposition of complex query, we use

ResultName := *<expression consisting of relational operations>*

Relational algebra (RA)

- if clear from the context, we use just $R_1 \text{ operation } R_2$ instead of $\langle R_1, R_1(A_1) \rangle \text{ operation } \langle R_2, R_2(A_2) \rangle$
- for binary operation we use infix notation, for unary operations we use postfix notation
- the operation result can be used recursively as an operand of another operation, i.e., a tree of operations can be defined for more complex query

$\langle R_1^*, R_1(A) \rangle \text{ op1 } \langle R_1^*, R_1(A) \rangle \text{ op2}$

RA – attribute renaming

- attribute renaming – unary operation

$$\begin{aligned} R^* < a_i \rightarrow b_i, a_j \rightarrow b_j, \dots > = \\ & < R^*, R_x((A - \{a_i, a_j, \dots\}) \cup \{b_i, b_j, \dots\}) > \end{aligned}$$

- only attributes in the scheme are renamed, no data manipulation (i.e., the result is the same relation and the same scheme, just of different attribute names)

RA – set operations

- set operations (binary, infix notation)
 - union – $\langle R_1^*, R_1(A) \rangle \cup \langle R_2^*, R_2(A) \rangle = \langle R_1^* \cup R_2^*, R_x(A) \rangle$
 - intersection – $\langle R_1^*, R_1(A) \rangle \cap \langle R_2^*, R_2(A) \rangle = \langle R_1^* \cap R_2^*, R_x(A) \rangle$
 - subtraction – $\langle R_1^*, R_1(A) \rangle - \langle R_2^*, R_2(A) \rangle = \langle R_1^* - R_2^*, R_x(A) \rangle$
 - cartesian product – $\langle R_1^*, R_1(A) \rangle \times \langle R_2^*, R_2(B) \rangle = \langle R_1^* \times R_2^*, R_x(\{R_1\} \times A \cup \{R_2\} \times B) \rangle$
- union, intersection and subtraction require **compatible schemes** of the operands – it is also the scheme of the result

RA – cartesian product

- a cartesian product gives a new scheme consisting of attributes coming from both source schemes
 - if the attribute names are ambiguous, we use a prefix notation, e.g., $R_1.a$, $R_2.a$
- if both the operands are the same, we need first to rename the attributes of one operand, i.e.,

$$\langle R_1^*, R_1(\{a,b,c\}) \rangle \times R_1^* \langle a \rightarrow d, b \rightarrow e, c \rightarrow f \rangle$$

Example – set operations

- $\text{FILM}(\text{FILM_NAME}, \text{ACTOR_NAME})$
- $\text{AMERICAN_FILM} = \{(\text{'Titanic'}, \text{'DiCaprio'}), (\text{'Titanic'}, \text{'Winslet'}), (\text{'Top Gun'}, \text{'Cruise'})\}$
- $\text{NEW_FILM} = \{(\text{'Titanic'}, \text{'DiCaprio'}), (\text{'Titanic'}, \text{'Winslet'}), (\text{'Samotáři'}, \text{'Macháček'})\}$

$\text{CZECH_FILM} = \{(\text{'Vesničko má, středisková'}, \text{'Labuda'}), (\text{'Samotáři'}, \text{'Macháček'})\}$

$\text{ALL_FILM} := \text{AMERICAN_FILM} \cup \text{CZECH_FILM} =$
 $\{(\text{'Titanic'}, \text{'DiCaprio'}), (\text{'Titanic'}, \text{'Winslet'}), (\text{'Top Gun'}, \text{'Cruise'}),$
 $(\text{'Pelíšky'}, \text{'Donutil'}), (\text{'Samotáři'}, \text{'Macháček'})\}$

- $\text{OLD_AMERICAN_AND_CZECH_FILM} :=$
 $(\text{AMERICAN_FILM} \cup \text{CZECH_FILM}) - \text{NEW_FILM} =$
 $\{(\text{'Top Gun'}, \text{'Cruise'}), (\text{'Vesničko má, středisková'}, \text{'Labuda'})\}$

$\text{NEW_CZECH_FILM} := \text{NEW_FILM} \cap \text{CZECH_FILM} = \{(\text{'Samotáři'}, \text{'Macháček'})\}$

RA – projection

- projection (unary operation)

$\langle R^*[C], R(A) \rangle = \langle \{u[C] \in R^*\}, R(C) \rangle$, where $C \subseteq A$

- $u[C]$ relation element with values only in C attributes
- possible duplicities are removed

RA – selection

- selection (unary)

$$\langle R^*(\varphi), R(A) \rangle = \langle \{u \mid u \in R^* \text{ and } \varphi(u)\}, R(A) \rangle$$

- selection of those elements from R^* that match a condition $\varphi(u)$
- condition is a boolean expression (i.e., using **and**, **or**, **not**) on atomic formulas $t_1 \Theta t_2$ or $t_1 \Theta a$, where $\Theta \in \{<, >, =, \geq, \leq, \neq\}$ and t_i are names of attributes

RA – natural join

- natural join (binary)

$$\begin{aligned} & \langle R^*, R(A) \rangle * \langle S^*, S(B) \rangle = \\ & \quad \langle \{u \mid u[A] \in R^* \text{ and } u[B] \in S^*\}, R_x(A \cup B) \rangle \end{aligned}$$

- joining elements of relations A, B using **identity on all shared attributes**
- if $A \cap B = \emptyset$, natural join is cartesian product
(no shared attributes, i.e., everything in A is joined with everything in B)
- could be expressed using cartesian product, selection and projection

Example – selection, projection, natural join

FILM(FILM_NAME, ACTOR_NAME)

ACTOR(ACTOR_NAME, BIRTH_YEAR)

FILM = {('Titanic', 'DiCaprio'), ('Titanic', 'Winslet'), ('Top Gun', 'Cruise')}

ACTOR = {('DiCaprio', 1974), ('Winslet', 1975), ('Cruise', 1962), ('Jolie', 1975)}

ACTOR_YEAR := **ACTOR[BIRTH_YEAR]** =

{(1974), (1975), (1962)}

YOUNG_ACTOR := **ACTOR(BIRTH_YEAR > 1970) [ACTOR_NAME]** =

{('DiCaprio'), ('Winslet'), ('Jolie')}

FILM_ACTOR := **FILM * ACTOR** =

{('Titanic', 'DiCaprio', 1974), ('Titanic', 'Winslet', 1975), ('Top Gun', 'Cruise', 1962)}

RA – inner Θ -join

- inner Θ -join (binary)

$$\begin{aligned} & \langle R^*, R(A) \rangle [t_1 \Theta t_2] \langle S^*, S(B) \rangle = \\ & \quad \langle \{u \mid u[A] \in R^*, u[B] \in S^*, u.t_1 \Theta u.t_2\}, \\ & \quad \{R_1\} \times A \cup \{R_2\} \times B \rangle \end{aligned}$$

- generalization of natural join
- joins over predicate (condition) Θ applied on individual attributes (of schemes entering the operation)
- An inner join gives a new scheme consisting of attributes coming from both source schemes

RA – left Θ -semi-join

- left inner Θ -semi-join (binary)

$$\langle R^*, R(A) \rangle \langle t_1 \Theta t_2] \langle S^*, S(B) \rangle = (R[t_1 \Theta t_2]S)[A]$$

- join restricted to the “left side”
(only attributes of A in the result scheme)
- right semi-join similar (projection on B)

RA – relation division

- relation division (binary)

$$\begin{aligned} & \langle R^*, R(A) \rangle \div \langle S^*, S(B \subset A) \rangle = \\ & \quad \langle \{t \mid \forall s \in S^* (t \oplus s) \in R^* \}, A - B \rangle \end{aligned}$$

- ⊕ is concatenation operation
(relation elements $\langle a_1, a_2, \dots \rangle$ and $\langle b_1, b_2, \dots \rangle$ become $\langle a_1, a_2, \dots, b_1, b_2, \dots \rangle$)
- returns those elements from R^* that, when projected on $A - B$, are **duplicates** and, when projected on B , is **equal** to S^*
- alternative definition: $R^* \div S^* = R^*[A - B] - ((R^*[A - B] \times S^*) - R^*)[A - B]$
- used in situations where objects with **all properties** are needed
 - kind of universal quantifier in RA

Example – relation division

FILM(FILM_NAME, ACTOR_NAME)

ACTOR(ACTOR_NAME, BIRTH_YEAR)

What are the films where all the actors appeared?

ACTOR_ALL_FILM := $\text{FILM} \div \text{ACTOR}[\text{ACTOR_NAME}] = \{\text{'Titanic'}\}$

FILM_NAME	ACTOR_NAME
Titanic	DiCaprio
Titanic	Winslet
The Beach	DiCaprio
Enigma	Winslet
The Kiss	Zane
Titanic	Zane

ACTOR_NAME	BIRTH_YEAR
DiCaprio	1974
Zane	1966
Winslet	1975

Inner vs. outer join

- so far, we considered **inner joins**
- in practice, it is useful to introduce **null metavales** (NULL) of attributes
- **outer join** appends series of NULL values to those elements, that were not joined (i.e., they do not appear in inner join)
 - left outer join
$$R *_L S = (R * S) \cup (\underline{R} \times (\text{NULL}, \text{NULL}, \dots))$$
 - right outer join
$$R *_R S = (R * S) \cup ((\text{NULL}, \text{NULL}, \dots) \times \underline{S})$$
where \underline{R} , resp. \underline{S} consist of n-tuples not joined with S , resp. R
 - full outer join
$$R *_F S = (R *_L S) \cup (R *_R S)$$
- the above joins are defined as natural joins, outer Θ -joins are defined similarly
- the reason for outer join is a complete information on elements of a relation being joined (some are joined regularly, some only with NULLs)

Example – all types of joins

table
Flight

Flight	Company	Destination	Passengers
OK251	CSA	New York	276
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
AC906	Air Canada	Torronto	116
KL1245	KLM	Amsterdam	130

table
Plane

Plane	Capacity
Boeing 717	106
Airbus A380	555
Airbus A350	253

Query: In which planes all the passengers can travel (in the respective flights) such that the number of unoccupied seats in plane is lower than 200?

Inner Θ -join – we want the flights and planes that match the given condition

Flight [Flight.Passenger ≤ Plane.Capacity **AND** Flight.Passenger + 200 > Plane.Capacity] Plane

Left/right/full outer Θ -join – besides the above flight-plane pairs we also want those flights and planes that do not match the condition at all

The diagram illustrates the result of a full outer join between the Flight and Plane tables. The resulting table has columns: Flight, Company, Destination, Passengers, Plane, and Capacity.

Flight	Company	Destination	Passengers	Plane	Capacity
OK251	CSA	New York	276	NULL	NULL
KL1245	KLM	Amsterdam	130	Airbus A350	253
AC906	Air Canada	Torronto	116	Airbus A350	253
LH438	Lufthansa	Stuttgart	68	Airbus A350	253
LH438	Lufthansa	Stuttgart	68	Boeing 717	106
OK012	CSA	Milano	37	Boeing 717	106
NULL	NULL	NULL	NULL	Airbus A380	555

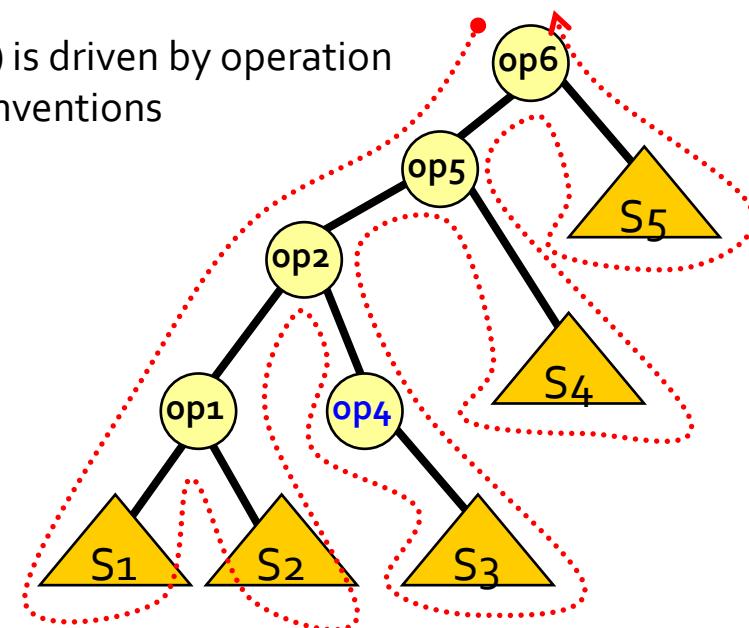
Annotations on the left side of the table indicate an "inner join" for the first four rows. Annotations on the right side indicate a "left outer join" for the first four rows and a "right outer join" for the last two rows.

At the bottom, a bracket spans the entire width of the table with the label "Query formalisms for relational model –". Below this, another bracket spans the width of the first four columns with the label "left/right semi join (w/o first and last row + after duplicates removal)".

RA query evaluation

- logical order of operation evaluation
 - nested operand evaluation needed – depth-first traversal of a syntactic tree
 - e.g., $\dots((S_1 \text{ op1 } S_2) \text{ op2 } (op4 S_3)) \text{ op5 } S_4 \text{ op6 } S_5$
 - syntactic tree construction (query parsing) is driven by operation priorities, parentheses, or associativity conventions
- operation precedence (priority)

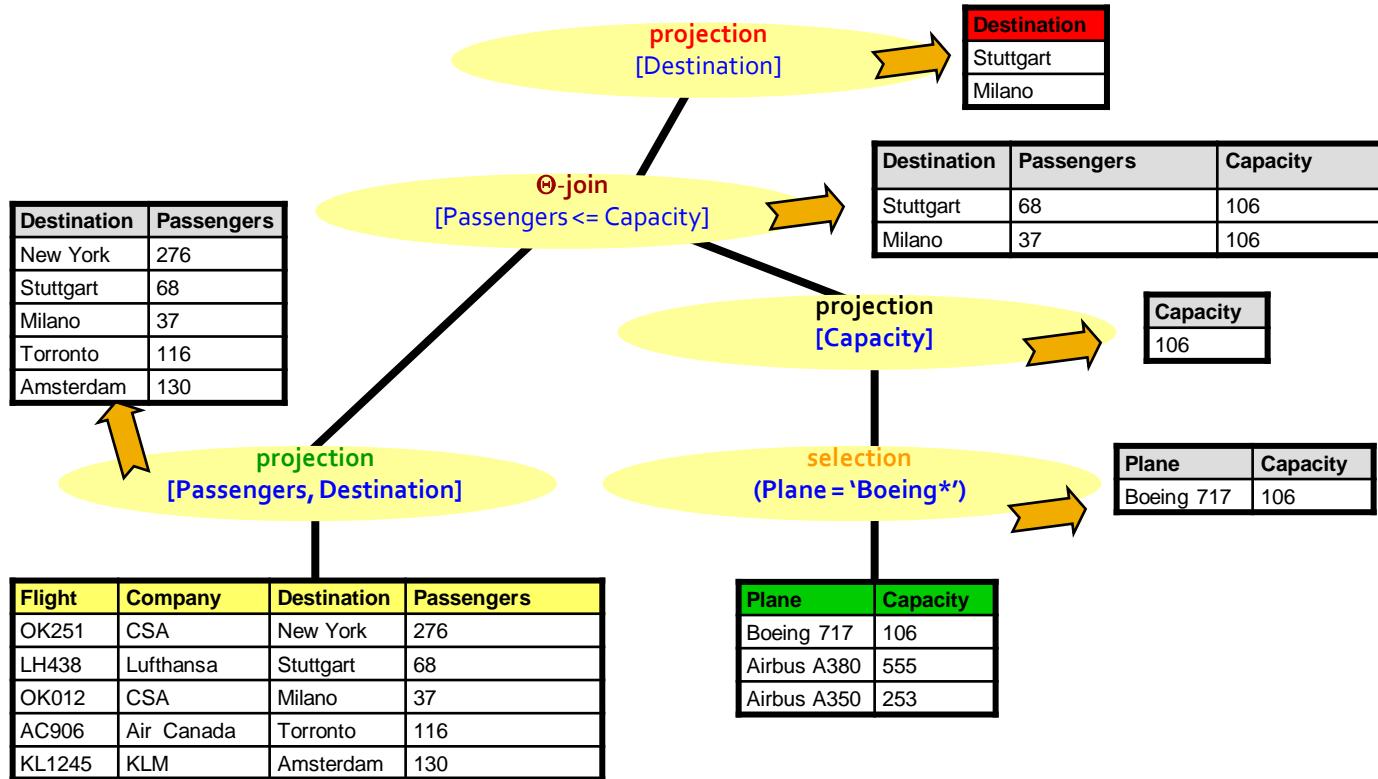
1. projection	R[] (highest)
2. selection	R()
3. cart. product	\times
4. join, division	\ast, \div
5. subtraction	$-$
6. union, intersection	\cup, \cap (lowest)



Example – query evaluation

Which destination can fly Boeings? (such that all passengers in the flight fit the plane)

(Flight[Passengers, Destination][Passengers <= Capacity] (Plane(Plane = 'Boeing*')[Capacity]))[Destination]



Equivalent expressions

- a single query may be defined by multiple expressions
 - by replacing “redundant” operations by the basic ones (e.g., division, natural join)
 - by use of commutativity, distributivity and associativity of (some) operations
- selection
 - selection cascade $\dots((R(\varphi_1))(\varphi_2))\dots)(\varphi_n) \equiv R(\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n)$
 - commutativity of selection $(R(\varphi_1))(\varphi_2) \equiv (R(\varphi_2))(\varphi_1)$
- projection
 - projection cascade $\dots(R[A_1])[A_2])\dots)[A_n] \equiv R[A_n]$, where $A_n \subseteq A_{n-1} \subseteq \dots \subseteq A_2 \subseteq A_1$
- join and cartesian product
 - commutativity $R \times S \equiv S \times R$, $R[\Theta]S \equiv S[\Theta]R$, etc.
 - associativity $R \times (S \times T) \equiv (R \times S) \times T$, $R[\Theta](S[\Theta]T) \equiv (R[\Theta]S)[\Theta]T$, etc.
 - combination, e.g., $R[\Theta](S[\Theta]T) \equiv (R[\Theta]T)[\Theta]S$

Equivalent expressions

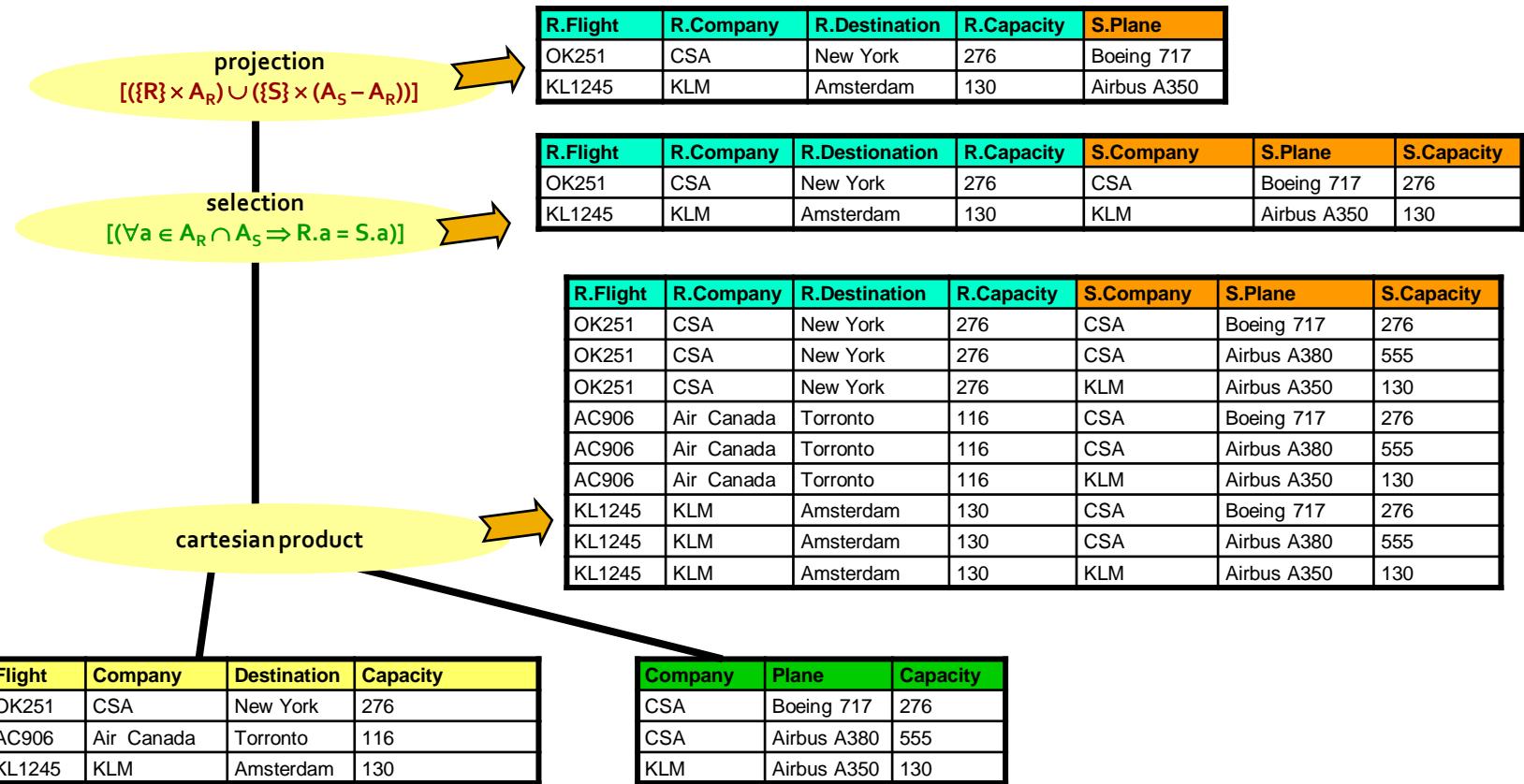
- complex equivalences for selection, projection and join
 - selection and projection swap
$$(R[A_i])(\varphi) \equiv (R(\varphi))[A_i], \text{ if } \forall a \in \varphi \Rightarrow a \in A_i$$
 - combination of selection and cartesian product (join definition):
$$R[\Theta]S \equiv (R \times S)(\Theta)$$
 - distributive swap of selection and cartesian product (or join)
$$(R \times S)(\varphi) \equiv R(\varphi) \times S, \text{ if } \forall a \in \varphi \Rightarrow a \in A_R \wedge a \notin A_S$$
 - distributive swap of projection and cartesian product (or join)
$$(R \times S)[A_1] \equiv R[A_2] \times S[A_3],$$

if $A_2 \subseteq A_1 \wedge A_2 \subseteq R_A$ and $A_3 \subseteq A_1 \wedge A_3 \subseteq S_A$

similarly for join, $(R[\Theta]S)[A_1] \equiv R[A_2][\Theta]S[A_3],$
where moreover $\forall a \in \Theta \Rightarrow a \in A_1$
- other equivalences can be obtained when including set operations

Example – natural join

$$\langle R, A_R \rangle * \langle S, A_S \rangle \equiv (R \times S) (\forall a \in A_R \cap A_S \Rightarrow R.a = S.a) [(\{R\} \times A_R) \cup (\{S\} \times (A_S - A_R))]$$



Example – relation division

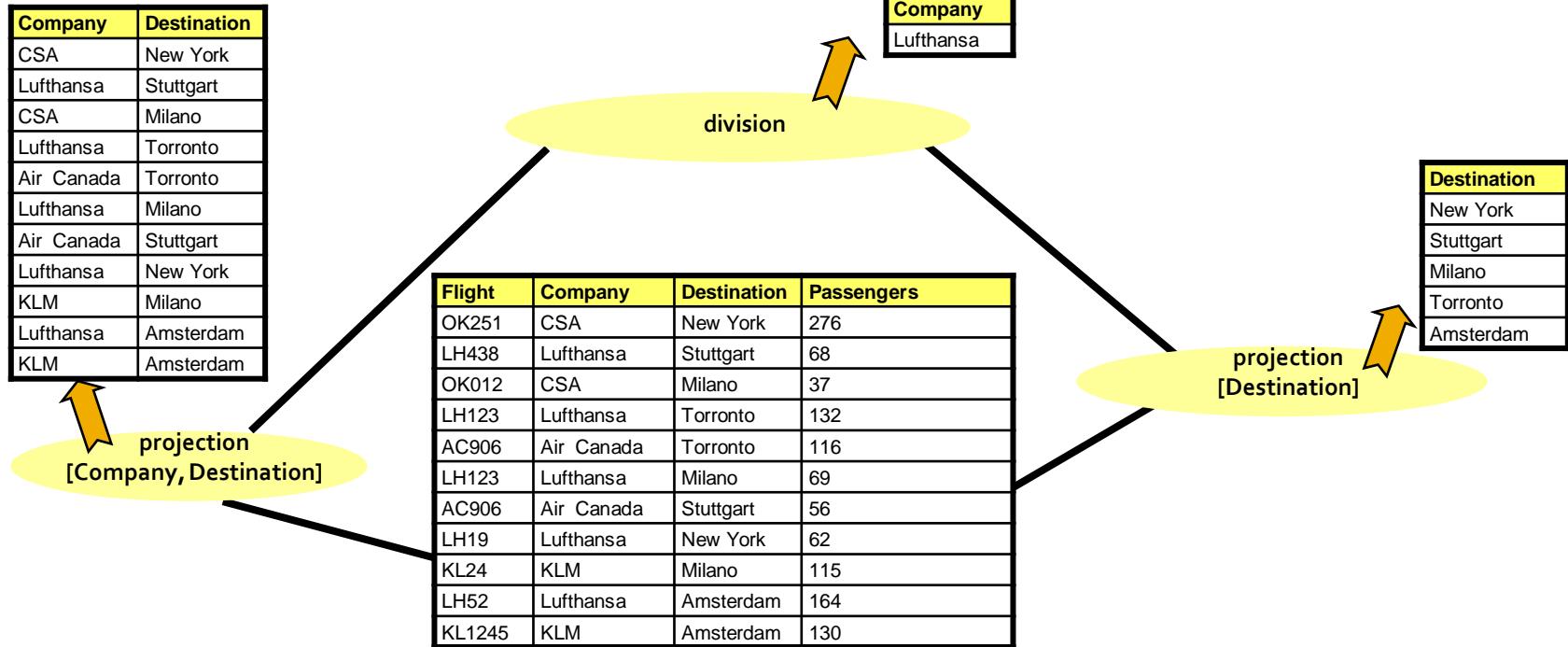
Which companies fly to every destination?

$\text{Flight}[\text{Company}, \text{Destination}] \div \text{Flight}[\text{Destination}]$

Company	Destination
CSA	New York
Lufthansa	Stuttgart
CSA	Milano
Lufthansa	Torronto
Air Canada	Torronto
Lufthansa	Milano
Air Canada	Stuttgart
Lufthansa	New York
KLM	Milano
Lufthansa	Amsterdam
KLM	Amsterdam

Flight	Company	Destination	Passengers
OK251	CSA	New York	276
LH438	Lufthansa	Stuttgart	68
OK012	CSA	Milano	37
LH123	Lufthansa	Torronto	132
AC906	Air Canada	Torronto	116
LH123	Lufthansa	Milano	69
AC906	Air Canada	Stuttgart	56
LH19	Lufthansa	New York	62
KL24	KLM	Milano	115
LH52	Lufthansa	Amsterdam	164
KL1245	KLM	Amsterdam	130

Destination
New York
Stuttgart
Milano
Torronto
Amsterdam



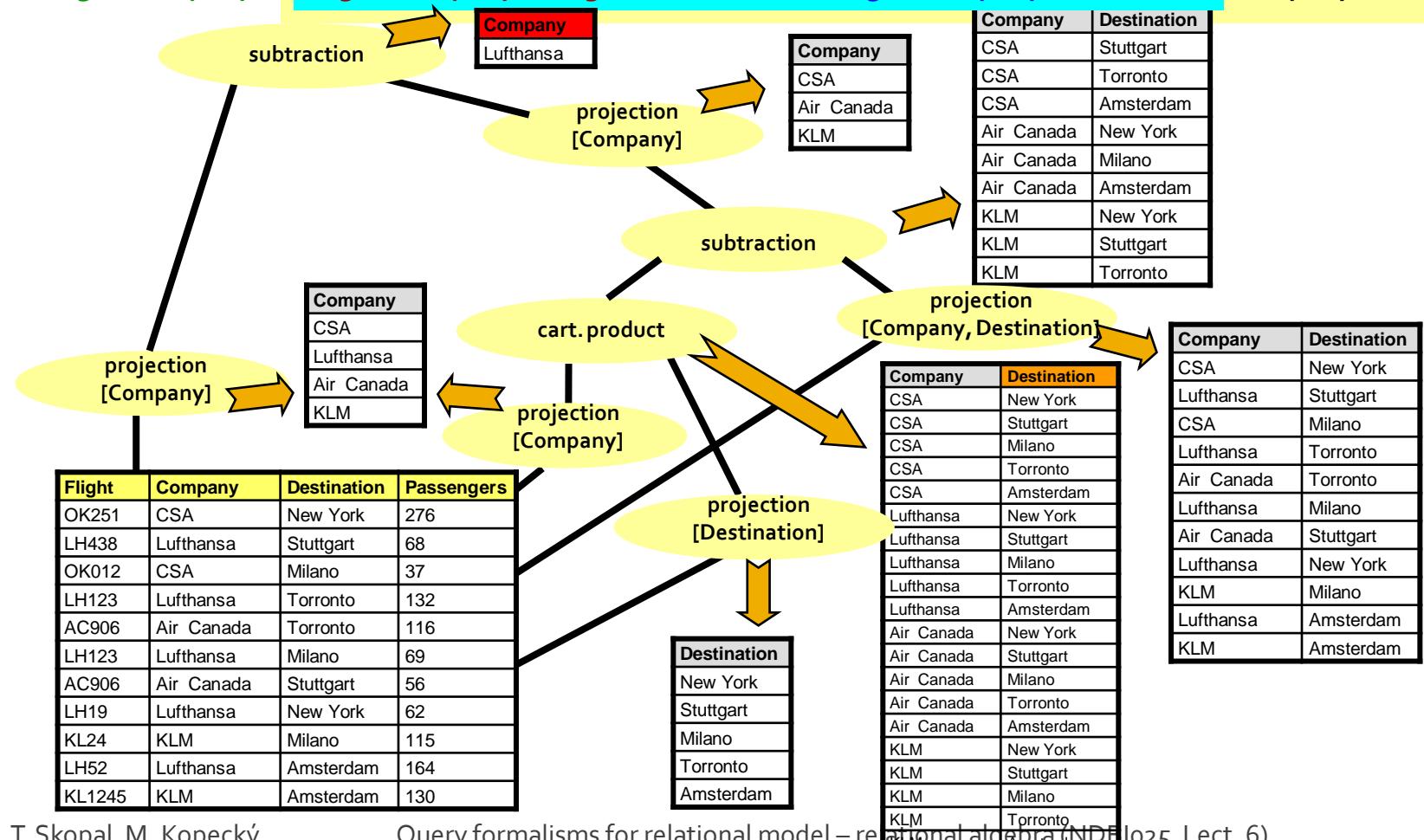
Example – “division without division”

Which companies fly to every destination?

$\text{Flight}[\text{Company}, \text{Destination}] \div \text{Flight}[\text{Destination}]$

$\text{Flight}[\text{Company}] - ((\text{Flight}[\text{Company}] \times \text{Flight}[\text{Destination}]) - \text{Flight}[\text{Company}, \text{Destination}])[\text{Company}]$

$$(R^* \div S^* = R^*[A-B] - ((R^*[A-B] \times S^*) - R^*[A-B]))$$



Relational completeness

- not all the mentioned operations are necessary for expression of every query
 - minimal set consists of the following operations
B = {union, cartesian product, subtraction, selection, projection, attribute renaming}
- relational algebra query language is the of expressions that result from composition of operations in B over scheme given by database scheme
- if two expressions denote the same query they are equivalent
- query language that is able to express all queries of RA is **relational complete**

RA – properties

- RA = declarative query language
 - i.e., non-procedural, however, the structure of the expression suggests the sequence of operations
- the result is **always finite** relation
 - „safely“ defined operations
- operation properties
 - associativity, commutativity
 - cart. product, join

course:

Database Systems (NDBlo25)

SS2017/18

lecture 7:

Query formalisms for relational model – relational calculus

doc. RNDr. Tomáš Skopal, Ph.D.

RNDr. Michal Kopecký, Ph.D.

Today's lecture outline

- relational calculus
 - domain relational calculus
 - tuple relational calculus
 - safe formulas

Relational calculus

- application of first-order calculus (predicate logic) for database querying
- extension by „database“ predicate testing a membership of an element in a relation, defined at two levels of granularity
 - domain calculus (DRC) – variables are attributes
 - tuple calculus (TRC) – variables are tuples (whole elements of relation)
- query result in DRC/TRC is relation (and the appropriate schema)

Relational calculus

- the “language”
 - terms – variables and constants
 - predicate symbols
 - standard binary predicates { $<$, $>$, $=$, \geq , \leq , \neq }
 - „database predicates“ (extending the first-order calculus)
 - formulas
 - atomic – $R(t_1, t_2, \dots)$, where R is predicate symbol and t_i is term
 - complex – expressions that combine atomic or other complex formulas using logic predicates $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$
 - quantifiers \exists (existential), \forall (universal)

Domain relational calculus (DRC)

- variables stand for attributes, resp. their values
- database predicate $R(x, y, \dots)$
 - R is stands for the name of table the predicate is applied on
 - predicate that for interpreted x, y, \dots returns *true* if there is an element in R (table row) with the same values
 - i.e., row membership test
 - predicate scheme (input parameters) is the same as the scheme of relation R , i.e., each parameter x, y, \dots is substituted by a (interpreted) variable or constant

Domain relational calculus (DRC)

- database predicate
 - variable and constants in the predicate have an attribute name assigned, determining the value testing, e.g.:
CINEMA(*NAME_CINEMA* : x, *FILM* : y)
Then relation schema can be defined as a (unordered) set
{*NAME_CINEMA*, *FILM*}.
 - if the list of variables and constants does not contain the attribute names, it is assumed they belong to the attributes given by the relation schema R, e.g.:
CINEMA(x, y), where <*NAME_CINEMA*, *FILM*> is the schema if CINEMA – in the following we consider this short notation

Domain relational calculus (DRC)

- the result of query given in DRC is a set of all interpretations of variables and constants in the form of ordered n-tuple, for which the formula of the query is true
 - $\{(t_1, t_2, \dots) \mid \text{query formula that includes } t_1, t_2, \dots\}$
 - t_i is either a constant or a **free variable**, i.e., a variable that is not quantified inside the formula
 - the schema of the result relation is defined directly by the names of the free variables
 - e.g., query $\{(x, y) \mid \text{CINEMA}(x, y)\}$ returns relation consisting of all CINEMA elements
 - query $\{(x) \mid \text{CINEMA}(x, 'Titanic')\}$ returns names of cinemas that screen the Titanic film

Domain relational calculus (DRC)

- quantifiers allow to declare (bound) variables that are interpreted within the database predicates
 - formula $\exists x R(t_1, t_2, \dots, x, \dots)$ is evaluated as true if there **exists** domain interpretation of x such that n-tuple $(t_1, t_2, \dots, x, \dots)$ is a member of R
 - formula $\forall x R(t_1, t_2, \dots, x, \dots)$ is evaluated as true if **all** domain interpretations of x leading to n-tuples $(t_1, t_2, \dots, x, \dots)$ are members of R
 - e.g., query $\{(film) \mid \exists name_cinema CINEMA(name_cinema, film)\}$ returns names of all films screened **at least** in one cinema

Domain relational calculus (DRC)

- it is important to determine which domain is used of interpretation of variables (both bound and free variables)
 1. domain can be not specified (i.e., interpretation is not limited) – the domain is the whole **universum**
 2. domain is an attribute type – **domain** interpretation
 3. domain is a set of values of a given attribute that exist in the relation on which the interpretation is applied – **actual domain** interpretation

Domain relational calculus (DRC)

- e.g., query $\{(film) \mid \forall name_cinema CINEMA(name_cinema, film)\}$ could be evaluated differently based on the way variable **name_cinema** (type/domain string) is interpreted
 - if the **universe** is used, the query result is an empty set because the relation CINEMA is surely not infinite also is type-restricted
 - e.g., values in **NAME_CINEMA** will not include '**horse**', **125**, '**quertyuiop**'
 - if the **domain** (attribute type) is used, the query answer will be also empty – still infinite relation CINEMA assumed, containing all strings, e.g., '**horse**', '**qwertyuiop**', ...
 - if the **actual domain** is used, the query result consists of names of films screened in all cinemas (that are contained in the relation **CINEMA**)

Domain relational calculus (DRC)

- if we implicitly consider interpretation based on the actual domain, we call such limited DRC as **DRC with limited interpretation**
- because schemas often consist of many attributes, we can use simplifying notation of quantification
 - an expression $R(t_1, \dots, t_i, t_{i+2}, \dots)$,
i.e., t_{i+1} is missing,
is understood as $\exists t_{i+1} R(t_1, \dots, t_i, t_{i+1}, t_{i+2}, \dots)$
 - the position of variables then must be clear from the context,
or strict attribute assignment must be declared
 - e.g., query $\{(x) \mid \text{CINEMA}(x)\}$ is the same as $\{(x) \mid \exists y \text{CINEMA}(x, y)\}$

Examples – DRC

FILM(NAME_FILM, NAME_ACTOR)

ACTOR(NAME_ACTOR, YEAR_BIRTH)

In what films all the actors appeared?

$\{(f) \mid \text{FILM}(f) \wedge \forall a (\text{ACTOR}(a) \Rightarrow \text{FILM}(f, a))\}$

Which actor is the youngest?

$\{(a, y) \mid \text{ACTOR}(a, y) \wedge \forall a_2 \forall y_2 ((\text{ACTOR}(a_2, y_2) \wedge a \neq a_2) \Rightarrow (y_2 \leq y))\}$

or

$\{(a, y) \mid \text{ACTOR}(a, y) \wedge \forall a_2 (\text{ACTOR}(a_2) \Rightarrow \neg \exists y_2 (\text{ACTOR}(a_2, y_2) \wedge a \neq a_2 \wedge y_2 > y))\}$

Which pairs of actors appeared at least in one film?

$\{(a_1, a_2) \mid \text{ACTOR}(a_1) \wedge \text{ACTOR}(a_2) \wedge a_1 \neq a_2 \wedge \exists f (\text{FILM}(f, a_1) \wedge \text{FILM}(f, a_2))\}$

Evaluation of DRC query

Which actor is the youngest?

$$\{(a,y) \mid \text{ACTOR}(a,y) \wedge \forall a_2 (\text{ACTOR}(a_2) \Rightarrow \exists y_2 (\text{ACTOR}(a_2, y_2) \wedge a \neq a_2 \wedge y_2 > y))\}$$

```
$result =  $\emptyset$ 
for each (a,y) do
    if (ACTOR(a,y) and
        (for each a2 do
            if (not ACTOR(a2) or not (for each y2 do
                if (ACTOR(a2, y2)  $\wedge$  a  $\neq$  a2  $\wedge$  y2 > y) = true then return true
                end for
                return false)) = false then return false
            end for
            return true) ) = true then Add (a,y) into $result
        end for
```

universal quantifier = chain of conjunctions
existential quantifier = chain of disjunctions

Tuple relational calculus (TRC)

- almost the same as DRC, the difference is variables/constants are whole elements of relations (i.e., row of tables), i.e., predicate $R(t)$ is interpreted as true if (a row) t belongs to R
 - the result schema is defined by concatenation of schemas of the free variables (n-tuples)
- to access the attributes within a tuple t , a “dot notation” is used
 - e.g., query $\{t \mid \text{CINEMA}(t) \wedge t.\text{FILM} = \text{'Titanic'}\}$ returns cinemas which screen the film Titanic
- the result schema could be projected only on a subset of attributes
 - e.g., query $\{t[\text{NAME_CINEMA}] \mid \text{CINEMA}(t)\}$

Examples – TRC

FILM(NAME_FILM, NAME_ACTOR)

ACTOR(NAME_ACTOR, YEAR_BIRTH)

Get the pairs of actors of the same age acting in the same film.

$$\{a_1, a_2 \mid \text{ACTOR}(a_1) \wedge \text{ACTOR}(a_2) \wedge a_1.\text{YEAR_BIRTH} = a_2.\text{YEAR_BIRTH} \\ \wedge \exists f_1, f_2 \text{ FILM}(f_1) \wedge \text{FILM}(f_2) \wedge f_1.\text{NAME_FILM} = f_2.\text{NAME_FILM} \\ \wedge f_1.\text{NAME_ACTOR} = a_1.\text{NAME_ACTOR} \\ \wedge f_2.\text{NAME_ACTOR} = a_2.\text{NAME_ACTOR}\}$$

Which films were casted by all the actors?

$$\{\text{film}[\text{NAME_FILM}] \mid \forall \text{actor}(\text{ACTOR}(\text{actor}) \Rightarrow \\ \exists f(\text{FILM}(f) \wedge f.\text{NAME_ACTOR} = \text{actor}.\text{NAME_ACTOR} \wedge \\ f.\text{NAME_FILM} = \text{film}.\text{NAME_FILM}))\}$$

Safe formulas in DRC

- unbound interpretation of variables (domain-dependent formulas, resp.) could lead to infinite query results
 - negation: $\{x \mid \neg R(x)\}$
 - e.g. $\{j \mid \neg \text{Employee}(\text{Name}: j)\}$
 - disjunction: $\{x, y \mid R(\dots, x, \dots) \vee S(\dots, y, \dots)\}$
 - e.g. $\{i, j \mid \text{Employee}(\text{Name}: i) \vee \text{Student}(\text{Name}: j)\}$
 - universal quantifiers lead to an empty set
 $\{x \mid \forall y R(x, \dots, y)\}$, and generally $\{x \mid \forall y \varphi(x, \dots, y)\}$, where φ does not include disjunctions (implications, resp.)
- even if the query result is finite,
how to manage **infinite** quantifications in **finite** time?
- the solution is to limit the set of DRC formulas – set of **safe formulas**

Safe formulas in DRC

- to simply avoid infinite quantification ad-hoc, it is good to constrain the quantifiers so that the **interpretation of bound variables is limited to a finite set**
 - using $\exists x (R(x) \wedge \varphi(x))$ instead of $\exists x (\varphi(x))$
 - using $\forall x (R(x) \Rightarrow \varphi(x))$ instead of $\forall x (\varphi(x))$
 - by this convention the evaluation is implemented as
`for each x in R // finite enumeration`
instead of
`for each x // infinite enumeration`
- free variables in $\varphi(x)$ can be limited as well – by conjunction
 - $R(x) \wedge \varphi(x)$

Safe formulas in DRC – 1

- more generally, a formula is safe if
 - for each disjunction $\varphi_1 \vee \varphi_2$ it holds that φ_1, φ_2 share the same free variables
(we consider all implications $\varphi_1 \Rightarrow \varphi_2$ transformed to disjunctions $\neg\varphi_1 \vee \varphi_2$ and the same for equivalences)

$\{(x,y) \mid P(x) \vee Q(y)\}$ not safe

(the disjunction elements $P(x)$, $Q(y)$ do not share the same variables)

$\{(x,y) \mid P(x,y) \vee Q(x,y)\}$ safe

Safe formulas in DRC – 2

- more generally, a formula is safe if
 1. all free variables in each maximal conjunction $\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n$ are limited, i.e., for each free variable x at least one of the following conditions holds:
 1. there exists a φ_i with the variable which is not a negation or binary (in)equation
(i.e., φ_i is non-negated complex formula or non-negated „database predicate”)
 2. there exists $\varphi_i \equiv x = a$, where a is constant
 3. there exists $\varphi_i \equiv x = v$, where v is limited
- | | |
|------------------------------------|--|
| $\{(x,y) \mid x = y\}$ | not safe (x, y not limited) |
| $\{(x,y) \mid x = y \vee R(x,y)\}$ | not safe
(the disjunction elements share both free variables, but the first maximal conjunction ($x=y$) contains equation of not limited variables) |
| $\{x,y \mid x = y \wedge R(x,y)\}$ | is safe |

Safe formulas in DRC – 3

- more generally, a formula is safe if
 - the negation is only applicable on conjunctions of step 2
- $\{(x,y,z) \mid R(x,y) \wedge \neg(P(x,y) \vee \neg Q(y,z))\}$ not safe
(z is not limited in the conjunction
+ the disjunction elements do not share the same variables)
- $\{(x,y,z) \mid R(x,y) \wedge \neg P(x,y) \wedge Q(y,z)\}$ is safe
equivalent formula to the previous one

Safe formulas in DRC – 4

- more generally, a formula is safe if
 - it does not contain \forall (not a problem, $\forall x \varphi(x)$ can be replaced by $\neg \exists x (\neg \varphi(x))$)

Relational calculus – properties

- “even more declarative” than relational algebra
(where the structure of nested operations hints the evaluation)
 - just specification of what the result should satisfy
- both DRC and TRC are relational complete
 - moreover, could be extended to be stronger
- besides the different language constructs, the three formalisms can be used for differently “coarse” access to data
 - operations of relational algebra work with entire relations (tables)
 - database predicates of TRC work with relation elements (rows)
 - database predicates of DRC work with attributes (attributes)

Examples – comparison of RA, DRC, TRC

FILM(NAME_FILM, NAME_ACTOR)

ACTOR(NAME_ACTOR, YEAR_BIRTH)

Which films were casted by all the actors?

RA:

FILM % ACTOR[NAME_ACTOR]

DRC:

$\{(f) \mid \text{FILM}(f) \wedge \forall a (\text{ACTOR}(a) \Rightarrow \text{FILM}(f, a))\}$

TRC:

$\{\text{film}[\text{NAME_FILM}] \mid \forall \text{actor} (\text{ACTOR}(\text{actor}) \Rightarrow$
 $\exists f (\text{FILM}(f) \wedge f.\text{NAME_ACTOR} = \text{actor}.\text{NAME_ACTOR} \wedge$
 $f.\text{NAME_FILM} = \text{film}.\text{NAME_FILM}))\}$

Examples – comparison of RA, DRC, TRC

EMPLOYEE(firstname, surname, status, children, qualification, practice, health, crimerecord, salary)
– the key is everything except for **salary**

Pairs of employees having similar salary (max. \$100 difference)?

DRC:

$$\{(e_1, e_2) \mid \exists p_1, s_1, pd_1, k_1, dp_1, zs_1, tr_1, sa_1, p_2, s_2, pd_2, k_2, dp_2, zs_2, tr_2, sa_2 \\ \text{EMPLOYEE}(e_1, p_1, s_1, pd_1, k_1, dp_1, zs_1, tr_1, sa_1) \wedge \\ \text{EMPLOYEE}(e_2, p_2, s_2, pd_2, k_2, dp_2, zs_2, tr_2, sa_2) \wedge |sa_1 - sa_2| \leq 100 \wedge \\ (e_1 \neq e_2 \vee s_1 \neq s_2 \vee pd_1 \neq pd_2 \vee k_1 \neq k_2 \vee dp_1 \neq dp_2 \vee zs_1 \neq zs_2 \vee tr_1 \neq tr_2)\}$$

TRC:

$$\{e_1[\text{firstname}], e_2[\text{firstname}] \mid \text{EMPLOYEE}(e_1) \wedge \text{EMPLOYEE}(e_2) \wedge \\ e_1 \neq e_2 \wedge |e_1.\text{salary} - e_2.\text{salary}| \leq 100\}$$

Extension of relational calculus

- relational completeness is not enough
 - just data in tables can be retrieved
- we would like to retrieve also derived data,
 - use derived data in queries, respectively
- e.g., queries like

„Which employees have their salary by 10% higher than an average salary?“
- solution – introducing aggregation functions
 - as shown in SQL SELECT ... GROUP BY ... HAVING

course:

Database Systems (NDBlo25)

SS2017/18

lecture 8:

Relational design – normal forms

doc. RNDr. Tomáš Skopal, Ph.D.
RNDr. Michal Kopecký, Ph.D.

Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague

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

Motaivation

- result of relational design
 - a set of relation 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
 - solution
 - relation schema normalization

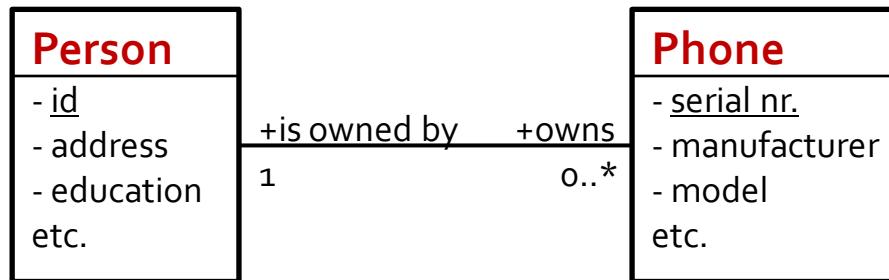
Example of “abnormal” schema

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)
- kind of alternative to conceptual modeling (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 $X_i, Y_i \subseteq A$ (kde 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 **elementary FD**
- FDs represent a generalization of the key concept (identifier)
 - the 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**
EmplId → everything
 - insertion of the last row is **not allowed** as it violates both the FDs

EmplId	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, really axiom)
- 2) if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$ (transitivity, rule)
- 3) if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$ (composition, rule)
- 4) if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$ (decomposition, rule)

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)

Proof follows from the FD definition
(from the properties of mapping, respectively).

Example – deriving FDs

$R(A, F)$

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

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

We could derive, e.g.,:

$$ab \rightarrow a \quad (\text{trivial})$$

$$ab \rightarrow ac \quad (\text{composition with } ab \rightarrow c)$$

$$ab \rightarrow d \quad (\text{transitivity with } ac \rightarrow d)$$

$$ab \rightarrow cd \quad (\text{composition with } ab \rightarrow c)$$

$$ab \rightarrow ed \quad (\text{transitivity with } cd \rightarrow ed)$$

$$ab \rightarrow e \quad (\text{decomposition})$$

$$ab \rightarrow f \quad (\text{transitivity})$$

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. $|F|$
- a FD f is **redundant** in F if
$$(F - \{f\})^+ = F^+, \text{ i.e., } f \text{ can be derived from the rest of } F$$

Example – closure of set of FDs

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

$F^+ =$

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

Cover

- **cover** of a set F is any set of FDs G such that $F^+ = G^+$
- **canonic cover** = cover consisting of **elementary FDs**
(decompositions are performed to obtain singleton sets on the right side)
- **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
 - implies that there may exist multiple non-redundant covers of F

Example – cover

$R_1(A, F)$, $R_2(A, G)$,

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

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

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

For the check of $G^+ = F^+$ we do not have to establish the whole closures, 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 R_1 and R_2 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).

Example – redundant FDs

$R_1(A, F)$, $R_2(A, G)$,

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

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

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 **attribute redundant in $X \rightarrow Y$**
- **reduced FD** is such FD that does not contain any redundant attributes (otherwise it is a partial FD)
- **key** is a set $K \subseteq A$ such that it is a super-key (i.e., $K \rightarrow A$) and $K \rightarrow A$ is moreover 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 $d \rightarrow d$)
$\{a, b\}^+$	$= \{a, b, c\}$		$ab \rightarrow c$	(+ trivial)
$\{a, d\}^+$	$= \{a, b, c, d\}$		$ad \rightarrow bc$	(+ 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. $ijk \rightarrow l$ remains in F because we **ADD** new FD $ij \rightarrow ?$
so that 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.

Minimal cover

- non-redundant canonic cover that consists of only reduced 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:

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

Wrong order of reduction:

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

First normal form (1NF)

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

(1NF is the basic condition on „flat database“ – a table is really
two-dimensional array, not a hidden graph or tree)

Example – 1NF

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

is in 1NF

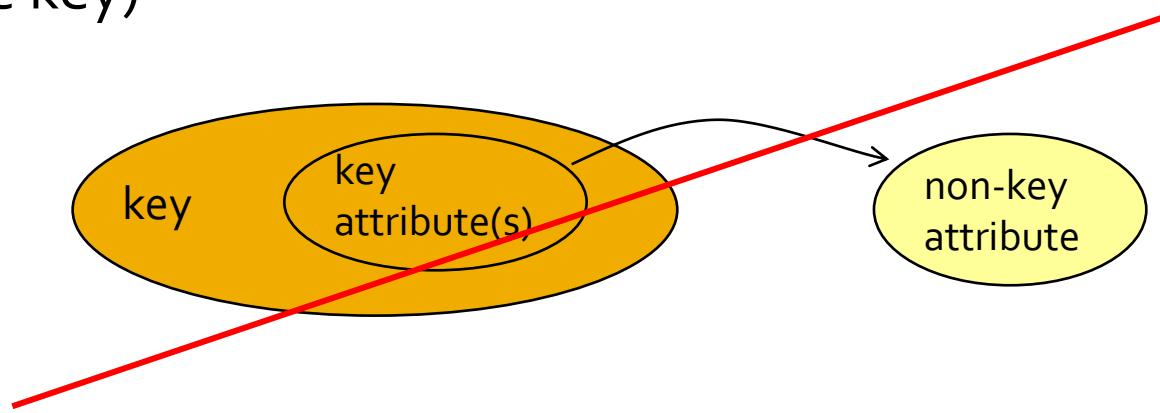
Employee(Id: **Integer**, Subordinate : **Person**[], Boss : **Person**)

not in 1NF

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

2nd normal form (2NF)

- 1NF + 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 $A \not\rightarrow \text{some key}$
(A is not a super-key), i.e., we get transitivity $\text{key} \rightarrow A \rightarrow B$
- Transitivities point to probable redundancies, because from the definition of FD as a mapping:
 - 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:

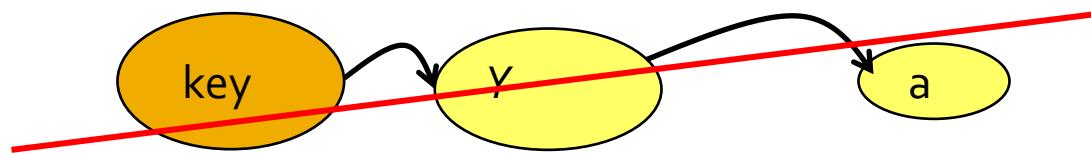
$\text{ZIPcode} \rightarrow \text{City} \rightarrow \text{Country}$

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.

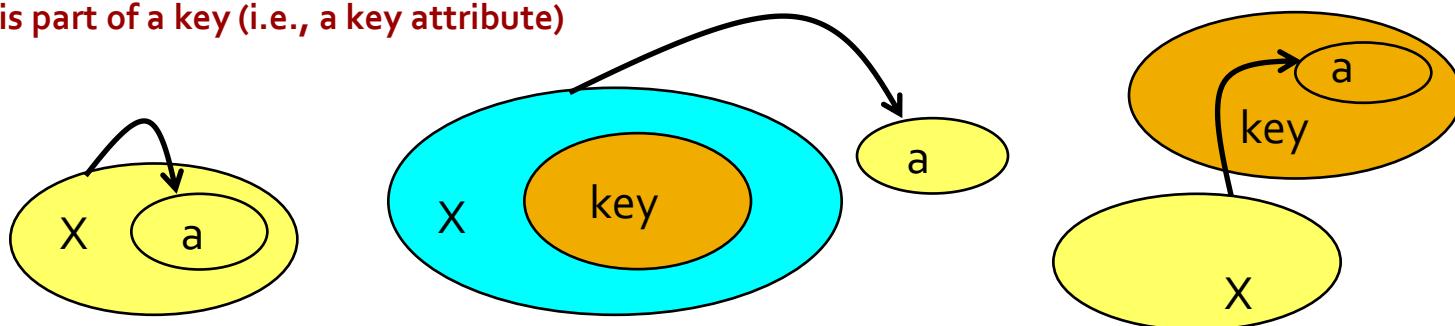
no redundancy medium redundancy high redundancy

3rd normal form (3NF)

- 1NF + non-key attributes are not transitively dependent on key



- 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*

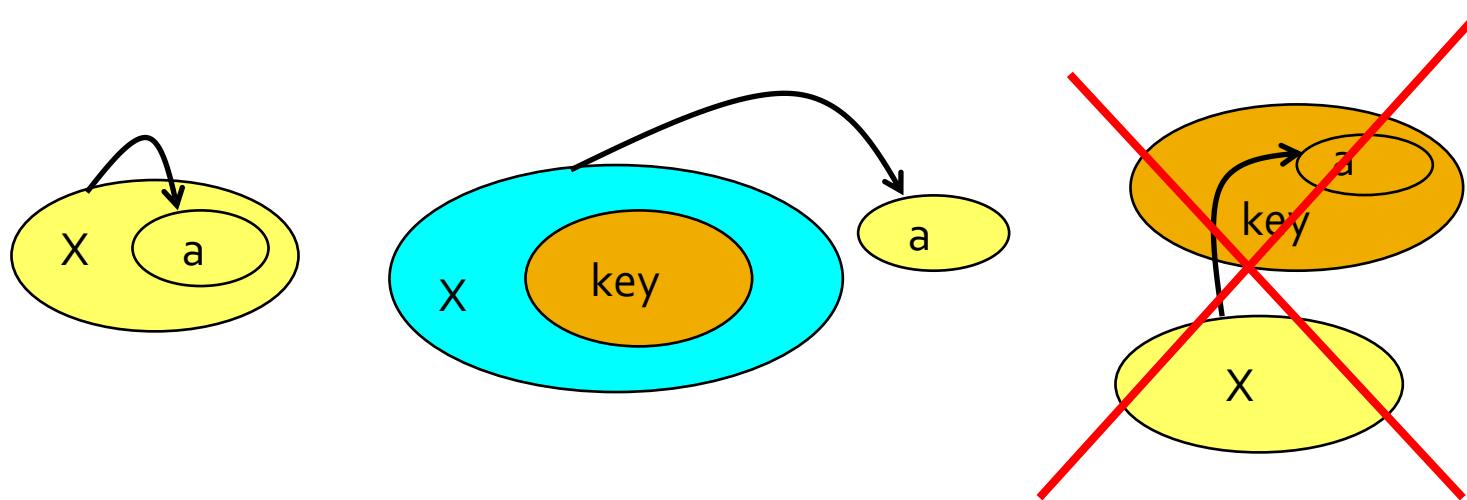
both schemas **are in 3NF**

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

Boyce-Codd normal form (BCNF)

- 1NF + every attribute is (non-transitively) dependent on key
- more exactly, in a give 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
- 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**

course:

Database Systems (NDBlo25)

SS2017/18

lecture 9:

Relational design – algorithms

doc. RNDr. Tomáš Skopal, Ph.D.

RNDr. Michal Kopecký, Ph.D.

Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague

Today's lecture outline

- schema analysis
 - basic algorithms (attribute closure, FD membership and redundancy)
 - determining the keys
 - testing normal forms
- normalization of universal schema
 - decomposition (to BCNF)
 - synthesis (to 3NF)

Attribute closure

- closure X^+ of attribute set X according to FD set F
 - principle: we iteratively derive all attributes „ F -determined“ by attributes in X
 - complexity $O(m*n)$, where n is the number of attributes and m is number of FDs

algorithm ***AttributeClosure***(set of dependencies F , set of attributes X) : **returns set** X^+

```
ClosureX := X; DONE := false; m = |F|;  
while not DONE do  
    DONE := true;  
    for i := 1 to m do  
        if (LS[i] ⊆ ClosureX and RS[i] ⊈ ClosureX) then  
            ClosureX := ClosureX ∪ RS[i];  
            DONE := false;  
        endif  
    endfor  
endwhile  
return ClosureX;
```

Note: expression $LS[i]$ ($RS[i]$, respectively) represents left (right, resp.) side of i -th FD in F

The trivial FD is used (algorithm initialization) and then transitivity (test of left side in the closure).

The composition and decomposition usage is hidden in the inclusion test.

Example – attribute closure

$$F = \{a \rightarrow b, bc \rightarrow d, bd \rightarrow a\}$$

$$\{b, c\}^+ = ?$$

1. $\text{ClosureX} := \{b, c\}$ (initialization)
2. $\text{ClosureX} := \text{ClosureX} \cup \{d\} = \{b, c, d\}$ ($bc \rightarrow d$)
3. $\text{ClosureX} := \text{ClosureX} \cup \{a\} = \{a, b, c, d\}$ ($bd \rightarrow a$)

$$\{b, c\}^+ = \{a, b, c, d\}$$

Membership test

- we often need to check if a FD $X \rightarrow Y$ belongs to F^+ , i.e., to solve the problem $\{X \rightarrow Y\} \in F^+$
- materializing F^+ is not practical,
we can employ the attribute closure

algorithm ***IsDependencyInClosure***(set of dependencies F , FD $X \rightarrow Y$)
return $Y \subseteq AttributeClosure(F, X)$;

Redundancy testing

The membership test can be easily used when testing redundancy of

- FD $X \rightarrow Y$ in F .
- attribute in X (according to F and $X \rightarrow Y$).

```
algorithm IsDependencyRedundant(set of dependencies F, dependency  $X \rightarrow Y \in F$ )
    return IsDependencyInClosure( $F - \{X \rightarrow Y\}$ ,  $X \rightarrow Y$ );
```

```
algorithm IsAttributeRedundant(set of deps.  $F$ , dep.  $X \rightarrow Y \in F$ , attribute  $a \in X$ )
    return IsDependencyInClosure( $F$ ,  $X - \{a\} \rightarrow Y$ );
```

In the ongoing slides we find useful the algorithm for reduction of the left side of a FD:

```
algorithm GetReducedAttributes(set of deps.  $F$ , dep.  $X \rightarrow Y \in F$ )
     $X' := X;$ 
    for each  $a \in X$  do
        if IsAttributeRedundant( $F$ ,  $X' \rightarrow Y$ ,  $a$ ) then  $X' := X' - \{a\}$ ;
    endfor
return  $X'$ ;
```

Minimal cover

- for all FDs we test redundancies and remove them

algorithm *GetMinimumCover*(set of dependencies F): returns minimal cover G

decompose each dependency in F into elementary ones

```
for each  $X \rightarrow Y$  in F do
     $F := (F - \{X \rightarrow Y\}) \cup \{GetReducedAttributes(F, X \rightarrow Y) \rightarrow Y\};$ 
endfor
for each  $X \rightarrow Y$  in F do
    if IsDependencyRedundant(F,  $X \rightarrow Y$ ) then  $F := F - \{X \rightarrow Y\};$ 
endfor
return F;
```

Determining (first) key

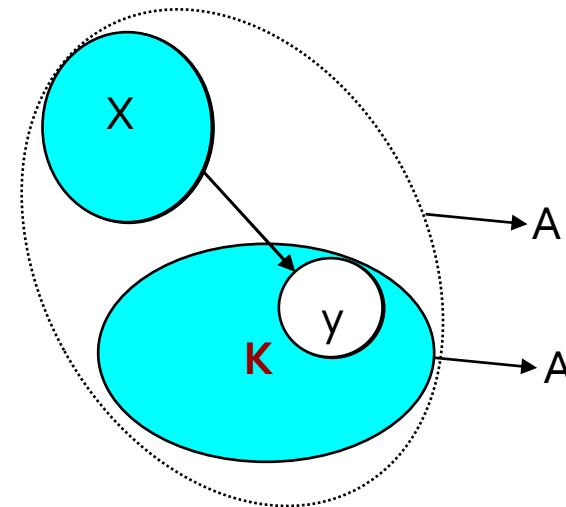
- the algorithm for attribute redundancy testing could be used directly for determining a key
- redundant attributes are iteratively removed from left side of $A \rightarrow A$

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

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

Determining all keys, the principle

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



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

Determining all keys, the algorithm

- Lucchesi-Osborn algorithm
 - to 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 deps. F, set of attributes A) : **returns set of all keys Keys;**

 let all dependencies in F be non-trivial, i.e. replace every $X \rightarrow Y$ by $X \rightarrow (Y - X)$

 K := *GetFirstKey*(F, A);

 Keys := {K};

 Done := **false**;

while Done = **false** **do**

 Done := **true**;

for each $X \rightarrow Y \in F$ **do**

if $(Y \cap K \neq \emptyset \text{ and } \neg \exists K' \in \text{Keys} : K' \subseteq (K \cup X) - Y)$ **then**

 K := *GetReducedAttributes*(F, $((K \cup X) - Y) \rightarrow A$);

 Keys := Keys $\cup \{K\}$;

 Done := **false**;

endfor

endwhile

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 = { $c \rightarrow all$, $sd \rightarrow p$, $p \rightarrow d$, $jp \rightarrow c$, $j \rightarrow s$ }

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

Testing normal forms

- NP-complete problem
 - we must know all keys – then it is sufficient to test a FD in F , so we do not need to materialize F^+
 - or, just one key needed, but also needing extension of F to F^+
- fortunately, in practice the keys determination is fast
 - thanks to limited size of F and „separability“ of FDs

Design of database schemas

Two means of modeling relational database:

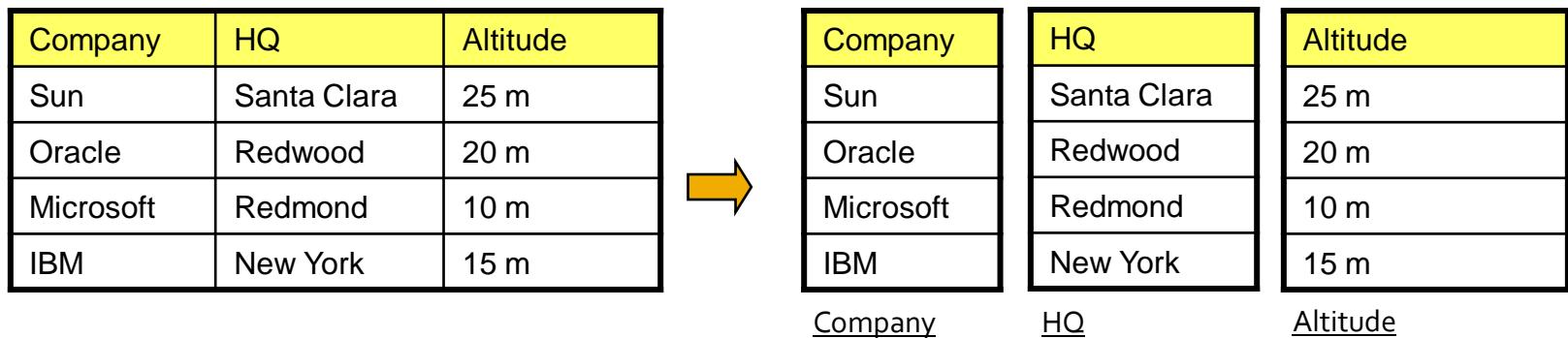
- we get a set of relational schemas
(as either direct relational design or conversion from conceptual model)
 - normalization performed separately on each table
 - the database could get unnecessarily highly “granularized” (too many tables)
- considering the whole database as a bag of (global) attributes results in a single *universal database schema* – i.e., one big table – including single set of FDs
 - normalization performed on the universal schema
 - less tables (better „granulating“)
 - „classes/entities“ are generated (recognized) as the consequence of FD set
 - modeling at the attribute level is less intuitive than the conceptual modeling (historical reasons)
- both approaches could be combined – i.e., at first, create a conceptual database model, then convert it to relational schemas and finally merge some | (all in the extreme case)

Relational schema normalization

- just one way – decomposition to multiple schemas
 - or merging some „abnormal“ schemas and then decomposition
- different criteria
 - **data integrity preservation**
 - **lossless join**
 - **dependency preserving**
 - requirement on normal form (3NF or BCNF)
- manually or algorithmically

Why to preserve integrity?

If the decomposition is not limited, we can decompose the table to several single-column ones that surely are all in BCNF.



Company,
HQ → Altitude

Clearly, there is something wrong with
such a decomposition...

...it is **lossy** and it does not
preserve dependencies

Lossless join

- a property of decomposition that ensures correct joining (reconstruction) of the universal relation from the decomposed ones
- Definition 1:
Let $R(\{X \cup Y \cup Z\}, F)$ be universal schema, where $Y \rightarrow Z \in F$.
Then decomposition $R_1(\{Y \cup Z\}, F_1), R_2(\{Y \cup X\}, F_2)$ is lossless.
- Alternative Definition 2:
Decomposition of $R(A, F)$ into $R_1(A_1, F_1), R_2(A_2, F_2)$ is lossless, if $A_1 \cap A_2 \rightarrow A_1$ or $A_2 \cap A_1 \rightarrow A_2$
- Alternative Definition 3:
Decomposition of $R(A, F)$ into $R_1(A_1, F_1), \dots, R_n(A_n, F_n)$ is lossless, if $R' = *_{i=1..n} R'[A_i]$.

Note:

R' is an instance of schema R (i.e., actual relation/table – the data).

Operation $*$ is natural join and $R'[A_i]$ is projection of R' on an attribute subset $A_i \subseteq A$.

Example – lossy decomposition

Company	Uses DBMS	Data managed
Sun	Oracle	50 TB
Sun	DB2	10 GB
Microsoft	MSSQL	30 TB
Microsoft	Oracle	30 TB

Company, Uses DBMS



Company	Uses DBMS	Company	Data managed
Sun	Oracle	Sun	50 TB
Sun	DB2	Sun	10 GB
Microsoft	MSSQL	Microsoft	30 TB
Microsoft	Oracle		

Company, Data managed

Company, Uses DBMS

Company	Uses DBMS	Data managed
Sun	Oracle	50 TB
Sun	Oracle	10 GB
Sun	DB2	10 GB
Sun	DB2	50 TB
Microsoft	MSSQL	30 TB
Microsoft	Oracle	30 TB

„reconstruction“
(natural join)

Company, Uses DBMS, Data managed



Example – lossless decomposition

Company	HQ	Altitude
Sun	Santa Clara	25 m
Oracle	Redwood	20 m
Microsoft	Redmond	10 m
IBM	New York	15 m



Company	HQ
Sun	Santa Clara
Oracle	Redwood
Microsoft	Redmond
IBM	New York

HQ	Altitude
Santa Clara	25 m
Redwood	20 m
Redmond	10 m
New York	15 m

Company,
HQ → Altitude



Company

HQ

„reconstruction“
(natural join)

Dependency preserving

- a decomposition property that ensures no FD will be lost
- Definition:
Let $R_1(A_1, F_1)$, $R_2(A_2, F_2)$ is decomposition of $R(A, F)$, then such decomposition preserves dependencies if $F^+ = (\cup_{i=1..n} F_i)^+$.
- Dependency preserving could be violated in two ways
 - during decomposition of F we do not derive all valid FDs – we lose FD that should be preserved in a particular schema
 - even if we derive all valid FDs (i.e., we perform projection of F^+), we may lose a FD that is valid **across the schemas**

Example – dependency preserving

dependencies not preserved, we lost
 $\text{HQ} \rightarrow \text{Altitude}$



Company	HQ	Altitude
Sun	Santa Clara	25 m
Oracle	Redwood	20 m
Microsoft	Redmond	10 m
IBM	New York	15 m

Company,
 $\text{HQ} \rightarrow \text{Altitude}$



dependencies
preserved

Company	Altitude
Sun	25 m
Oracle	20 m
Microsoft	10 m
IBM	15 m

Company	HQ
Sun	Santa Clara
Oracle	Redwood
Microsoft	Redmond
IBM	New York

Company

HQ

Company	HQ
Sun	Santa Clara
Oracle	Redwood
Microsoft	Redmond
IBM	New York

HQ	Altitude
Santa Clara	25 m
Redwood	20 m
Redmond	10 m
New York	15 m

Firma

Sídlo

The “Decomposition” algorithm

- algorithm for decomposition into BCNF, preserving lossless join
- does not preserve dependencies
 - not an algorithm property – sometimes we simply cannot decompose into BCNF with all FDs preserved

algorithm **Decomposition**(set of elem. deps. F , set of attributes A) : **returns set** $\{R_i(A_i, F_i)\}$

Result := $\{R(A, F)\};$

Done := **false**;

Create F^+ ;

while not Done **do**

if $\exists R_i(F_i, A_i) \in \text{Result}$ not being in BCNF **then**

 Let $X \rightarrow Y \in F_i$ such that $X \rightarrow A_i \notin F^+$.

// if there is a schema in the result violating BCNF

// X is not (super)key and so X → Y violates BCNF

 Result := $(\text{Result} - \{R_i(A_i, F_i)\}) \cup$

// we remove the schema being decomposed

$\{R_i(A_i - Y, \text{cover}(F, A_i - Y))\} \cup$

// we add the schema being decomposed without attributes Y

$\{R_j(X \cup Y, \text{cover}(F, X \cup Y))\}$

// we add the schema with attributes XY

else

 Done := **true**;

endwhile

return Result;

This partial decomposition on two tables is lossless, we get two schemas that both contain X , while the second one contains also Y and it holds $X \rightarrow Y$. X is now in the second table a super-key and $X \rightarrow Y$ is no more violating BCNF (in the first table there is not Y anymore).

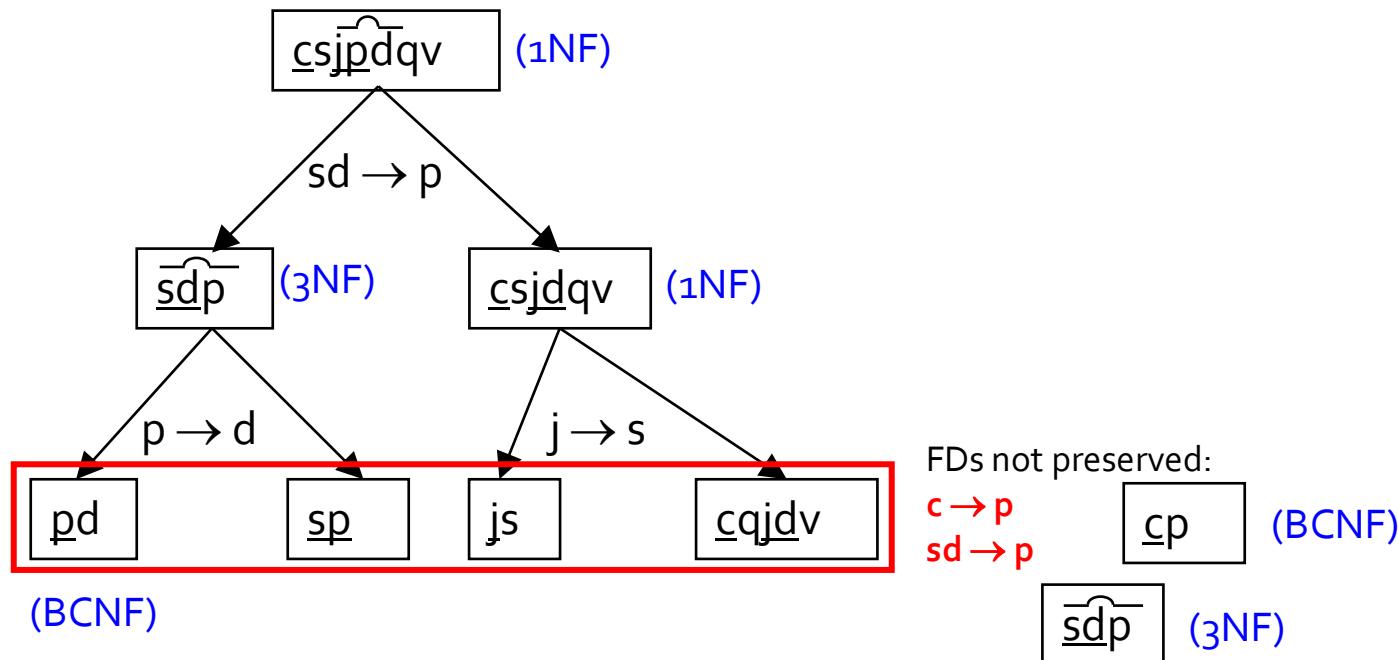
Note: Function $\text{cover}(X, F)$ returns all FDs valid on attributes from X , i.e., a subset of F^+ that contains only attributes from X . Therefore it is necessary to compute F^+ .

Example – decomposition

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}



The “Synthesis” algorithm

- algorithm for decomposition into 3NF, preserving dependencies
 - basic version not preserving lossless joins

algorithm ***Synthesis***(set of elem. deps. F , set of attributes A) : **returns set** $\{R_i(F_i, A_i)\}$

create minimal cover from F into G

compose FDs having equal left side into a single FD

every composed FD forms a schema $R_i(A_i, F_i)$ of decomposition

return $\cup_{i=1..n} \{R_i(A_i, F_i)\}$

- lossless joins can be preserved by adding another schema into the decomposition that contains *universal key* (i.e., a key from the original universal schema)
- a schema in decomposition that is a subset of another one can be deleted
- we can try to merge schemas that have functionally equivalent keys, but such an operation can violate 3NF! (or BCNF if achieved)

Example – synthesis

Contracts(A, F)

$$\begin{aligned} A &= \{c = \text{ContractId}, s = \text{SupplierId}, j = \text{ProjectId}, d = \text{DeptId}, p = \text{PartId}, \\ &\quad q = \text{Quantity}, v = \text{Value}\} \\ F &= \{c \rightarrow sjdpqv, sd \rightarrow p, p \rightarrow d, jp \rightarrow c, j \rightarrow s\} \end{aligned}$$

Minimal cover:

- There are no redundant attributes in FDs. There were removed redundant FDs $c \rightarrow s$ and $c \rightarrow p$.
- $G = \{c \rightarrow j, c \rightarrow d, c \rightarrow q, c \rightarrow v, sd \rightarrow p, p \rightarrow d, jp \rightarrow c, j \rightarrow s\}$

Composition:

- $G' = \{c \rightarrow jdqv, sd \rightarrow p, p \rightarrow d, jp \rightarrow c, j \rightarrow s\}$

Result:

- $R_1(\{\underline{cqjd}\}, \{c \rightarrow jdqv\}), R_2(\{sd\}, \{sd \rightarrow p\}), \cancel{R_3(\{pd\}, \{p \rightarrow d\})}, R_4(\{jp\}, \{jp \rightarrow c, c \rightarrow jp\}), R_5(\{js\}, \{j \rightarrow s\})$
(subset in R_2)

Equivalent keys: {c, jp, jd}

$$R_{1,4}(\{\underline{cqjd}\}, \{c \rightarrow jdqv, jp \rightarrow c, p \rightarrow d\}), R_{2,3}(\{\overline{sd}\}, \{sd \rightarrow p, p \rightarrow d\}), R_5(\{js\}, \{j \rightarrow s\})$$

merging R_1 and R_4
(however, now $p \rightarrow d$ violates BCNF)

$p \rightarrow d$ violates BCNF

Bernstein's extension

- if merging the schemas using equivalent keys K_1, K_2 violated 3NF, we perform the decomposition again
 1. $F_{\text{new}} = F \cup \{K_1 \rightarrow K_2, K_2 \rightarrow K_1\}$
 2. we determine redundant FDs in F_{new} , but remove them from F
 3. the final tables are made from reduced F and $\{K_1 \cup K_2\}$

Demo

- program Database algorithms
 - [download](#) from my web page
- [example 1](#)
- [example 2](#)

course:

Database Systems (NDBlo25)

SS2017/18

lecture 10:

Database transactions

doc. RNDr. Tomáš Skopal, Ph.D.

RNDr. Michal Kopecký, Ph.D.

Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague

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

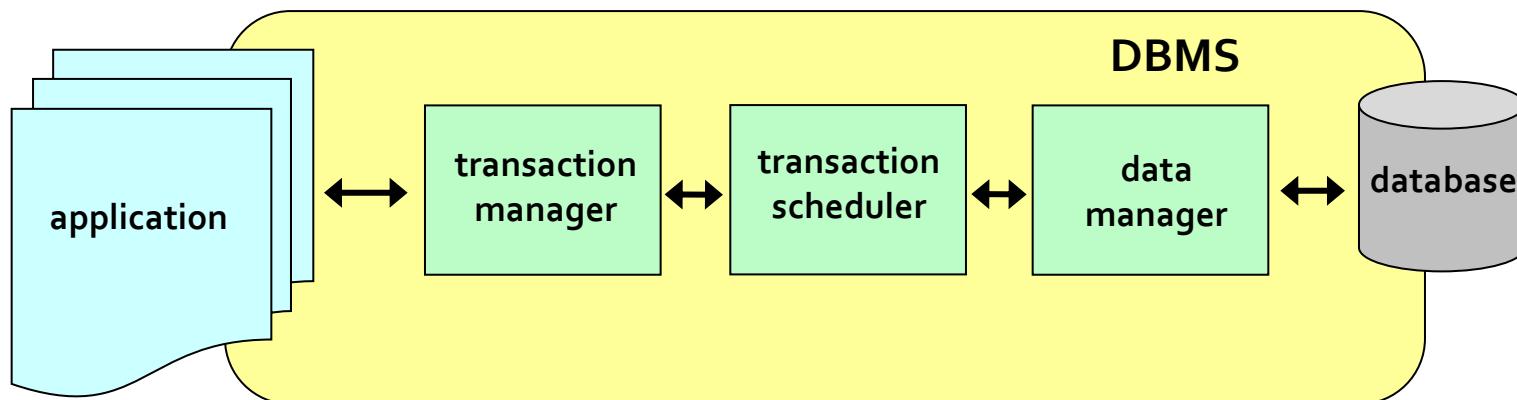
Motivation

- need to execute complex database operations
 - like stored procedures, triggers, etc.
 - multi-user and parallel environment
- database transaction
 - sequence of actions on database objects (+ others like arithmetic, etc.)
- example
 - Let's have a bank database with table **Accounts** and the following transaction to transfer 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 čÚčtu = fromAcc;
    4. UPDATE Accounts SET Balance = Balance + amount WHERE čÚčtu = 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
 - **ABORT** (or **ROLLBACK**) command – termination by the transaction code – user could be notified
 - system abort – DBMS aborts the transaction
 - for some integrity constraint violated – user is notified
 - by transaction scheduler (e.g., a deadlock occurs) – user is not notified
 - system failure – HW failure, power loss – transaction must be restarted
- main objectives of transaction management
 - enforcement of **ACID properties**
 - maximal performance (throughput trans per sec)
 - 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 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 and/or fair multitasking
- 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
$$T = \langle A_T^1, A_T^2, \dots, \text{COMMIT or ABORT} \rangle$$
- basic database actions (operations)
 - for now consider a **static database** (no inserts/deletes, just updates), let **A** is some 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 actions executed so far as valid, terminates transaction
 - **ABORT** – cancels action executed so far, terminates transaction (with error)
- SQL commands **SELECT, INSERT, UPDATE**, could be viewed as transactions implemented using the basic actions (in SQL **ROLLBACK** is used instead of abort)

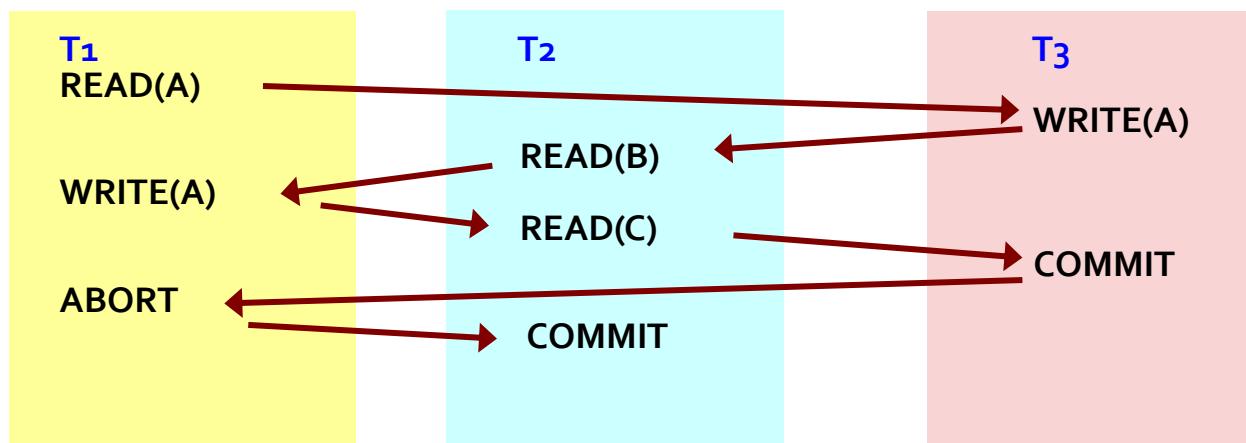
Example:

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



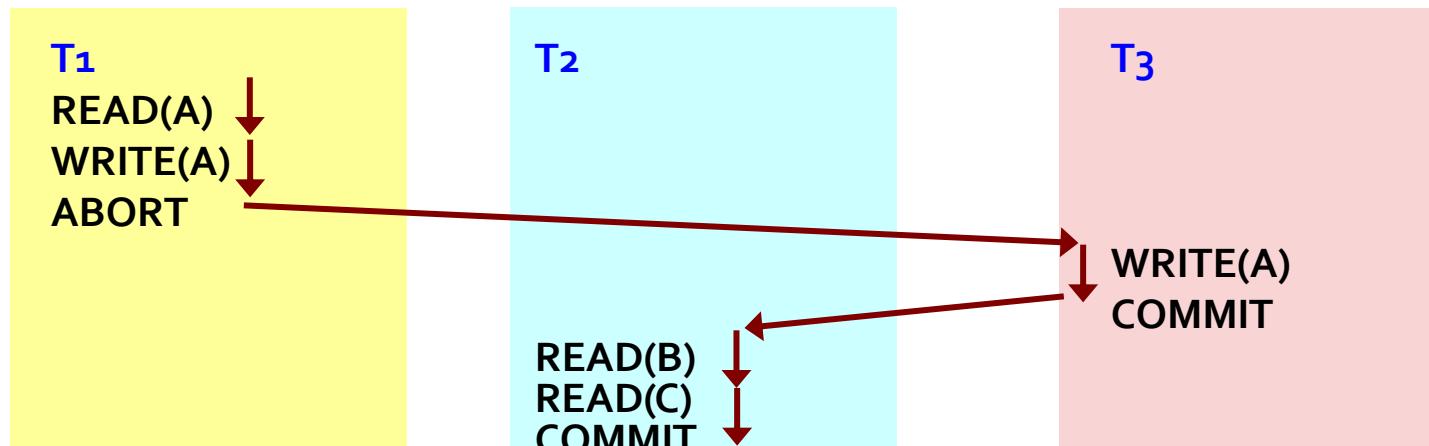
Transaction scheduler

- part of the transaction manager that is responsible for scheduling concurrent execution of actions of particular transactions
 - i.e., schedule is just history, once a schedule is created it was executed (so there nothing like offline schedule optimization or correction, it could be just analyzed)
- the database state is changing during concurrent execution
 - threat – temporarily inconsistent database state is exposed to other transactions
 - scheduler must provide the „isolation illusion“ and so avoid conflicts
 - example:

T₁	T₂	
READ(A)		// A = 5
A := A + 1	READ(B)	// B = 3
WRITE(A)	READ(A)	// A = 6 !!!
COMMIT	COMMIT	

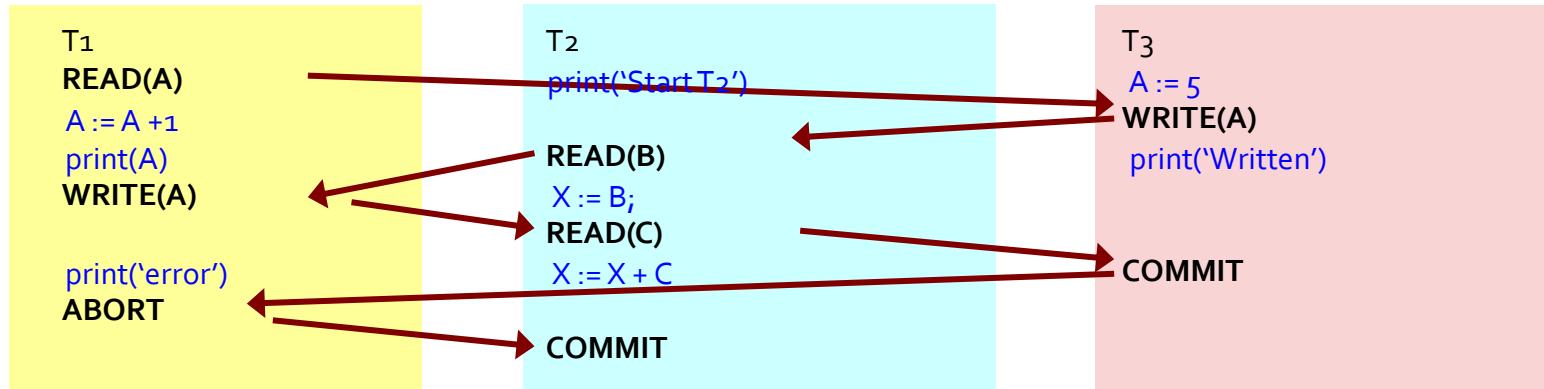
Serial schedules

- 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
- so why to interleave transactions when the number of steps is the same as serial schedule?
- two reasons
 - parallel execution of non-database operations with database operations
 - response proportional to transaction complexity (e.g., OldestEmployee vs. ComputeTaxes)
- example

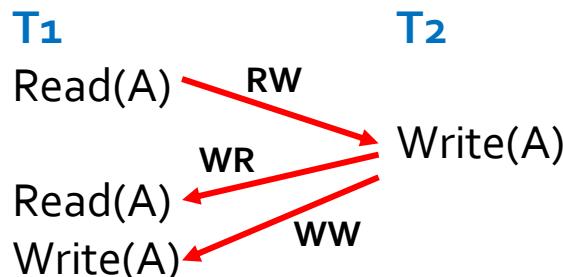


Serializability

- 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 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)
 - 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 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 just 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 didn't 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 computes annual interests on accounts (adds 1% per account)

T ₁	T ₂
R(A) // A = 12000	
A := A - 1000	
<u>W(A)</u> // database is now inconsistent – account B still contains the old balance	 R(A) // uncommitted data is read
	R(B)
	A := 1.01*A
	B := 1.01*B
	W(A)
	W(B)
	COMMIT
R(B) // B = 10100	
B := B + 1000	
W(B)	
COMMIT	// 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)

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

The diagram illustrates a transaction flow between two threads, T1 and T2. Thread T1 starts with a read operation R(A). A red arrow points from T1 to T2, labeled with the comment // A = 12000, indicating that T1 has updated the shared variable A to 12000. Thread T2 begins with its own read operation R(A). The transaction then proceeds through update operations W(A) and W(B), followed by a COMMIT. A pink arrow points from the end of T1's operations to the start of T2's updates, labeled with the comment // update of A, indicating that T2 is using the updated value of A from T1.

// database now contains A = 12120 – unrepeatable read

```
R(B)  
A := A - 1000  
W(A)  
B := B + 1000  
W(B)  
COMMIT
```

// inconsistent database, A = 11000, B = 11100

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)
 - inconsistency will show at 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)
COMMIT

DVD₁ := 10
W(DVD₁)
COMMIT

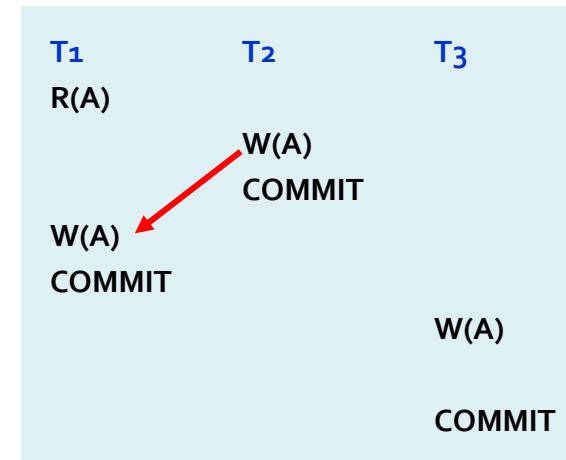
// inconsistent database, DVD₁ = 10, DVD₂ = 15

// overwrite of uncommitted data

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 (on the same transactions), i.e., there are no “real” conflicts
- more restrictive than serializability (defined by consistency preservation)
- **conflict serializability** alone does not consider
 - cancelled transactions
 - ABORT/ROLLBACK, so the schedule could be **unrecoverable**
 - dynamic database (inserting and deleting database objects)
 - so-called **phantom** may occur
 - hence, conflict serializability is not sufficient condition to provide ACID (**view-serializability** is ultimate condition)

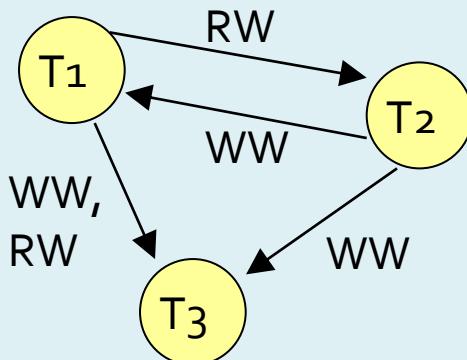
Example: schedule, that is **Serializable** (serial schedule $\langle T_1, T_2, T_3 \rangle$), but is **not conflict-serializable** (writes in T_1 and T_2 are in wrong order)



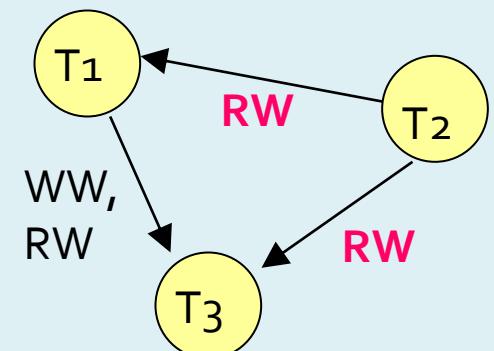
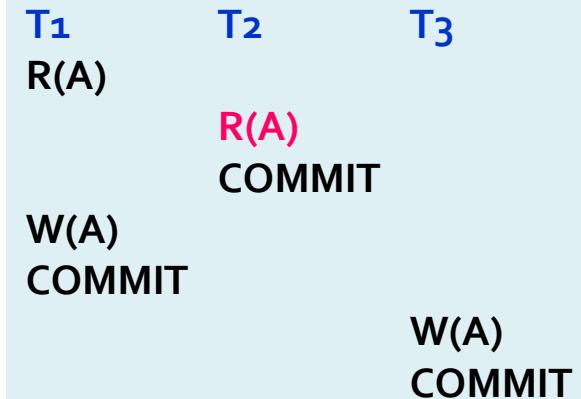
Detection of conflict serializability

- precedence graph (also serializability graph) on a schedule
 - nodes T_i are **committed** transactions
 - arcs represent RW, WR, WW conflicts in the schedule
- schedule is conflict-serializable if its precedence graph is **acyclic**

Example: not conflict-serializable



Example: conflict-serializable



View serializability

- Two schedules are **view-equivalent** if
 - Transaction T_i reads the initial value of X in one schedule if and only if it reads the initial value in the second schedule
 - Operation O_i in transaction T_i reads the value of X produced by operation O_j in transaction T_j in one schedule if and only if it reads the same value in the second schedule
 - Transaction T_i writes the final value of X in one schedule if and only if it writes the initial value in the second schedule
- A schedule is **view-serializable** if it is conflict-equivalent to some serial schedule
- Every conflict-serializable schedule is also view-serializable, but not vice-versa.

T_2

View serializability

Example:

T₁

R(X)

W(X)

COMMIT

T₂

W(X)

W(X)

COMMIT

T₃

W(X)

COMMIT

// lost write – can be ignored

Timestamps

Every transaction receives a unique timestamp
 $TS(T)$

- The system's clock
- A unique counter, incremented by the scheduler
- The timestamp order defines the serialization order of the transaction

Timestamps

Associate to each element X:

- $R_T(X)$ = the highest timestamp of any transaction that read X
- $W_T(X)$ = the highest timestamp of any transaction that wrote X

These are associated to each page X in the buffer pool

Main Idea

For any two conflicting actions check that
their order is correct:

In each of these cases

- $w_U(X)$ before $r_T(X)$
- $r_U(X)$ before $w_T(X)$
- $w_U(X)$ before $w_T(X)$

Check that $TS(U) < TS(T)$

Timestamp-based Scheduling

When a transaction T requests R(X) or W(X),
the scheduler examines RT(X), WT(X)

- Grant request, if timestamps are OK
- Rollback T (and restart it with later timestamp)

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 cause by the aborted transaction (Durability property of ACID)
- in **recoverable schedule**
 - a transaction T is committed after all other transactions commit that affected T (i.e., they changed data later read by T)
 - moreover, if reading changed data is allowed only for committed transactions, we avoid **cascade aborts of transactions**
 - in the example T₂ would begin after T₁'s abort

Example: T₁ transfers 1000 USD from A to B,
T₂ adds annual interests

T₁
R(A)
 $A := A - 1000$
W(A)

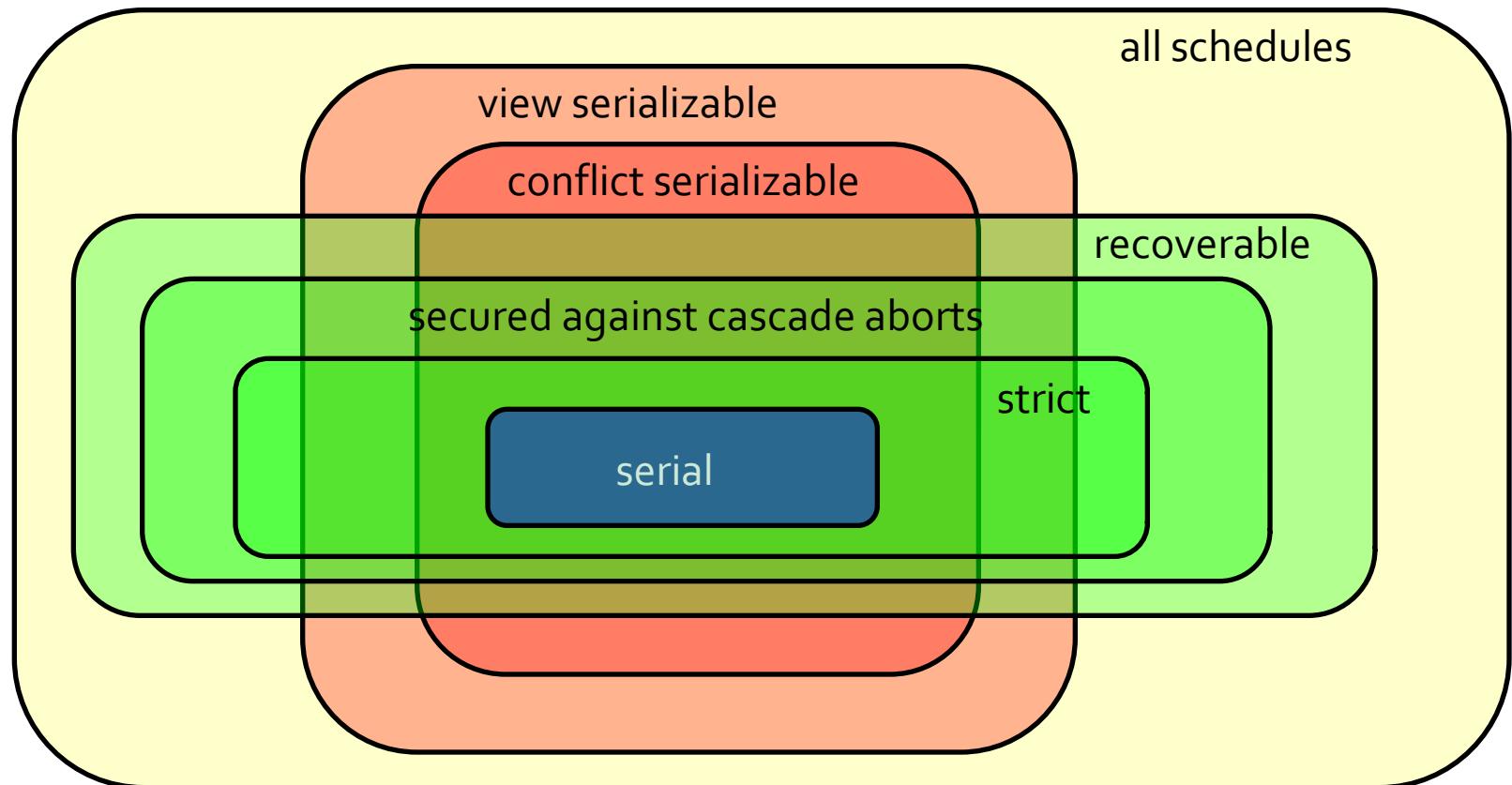
committed,
cannot be
undone! →

cascade aborts

ABORT

T₂
R(A)
 $A := A * 1.01$
W(A)
R(B)
 $B := B * 1.01$
W(B)
COMMIT

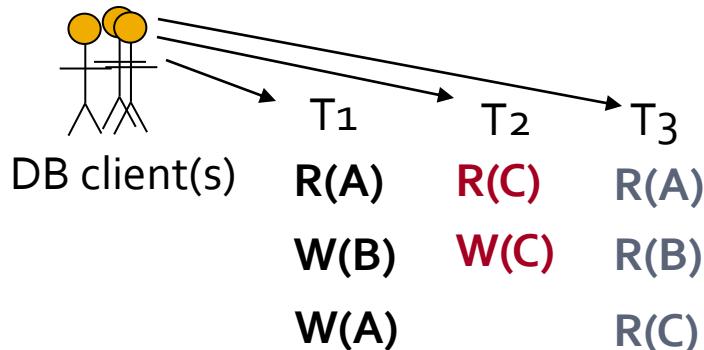
Classes of schedules



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 using **protocol**
 - dynamic transaction execution, branching parts in code

Schedule

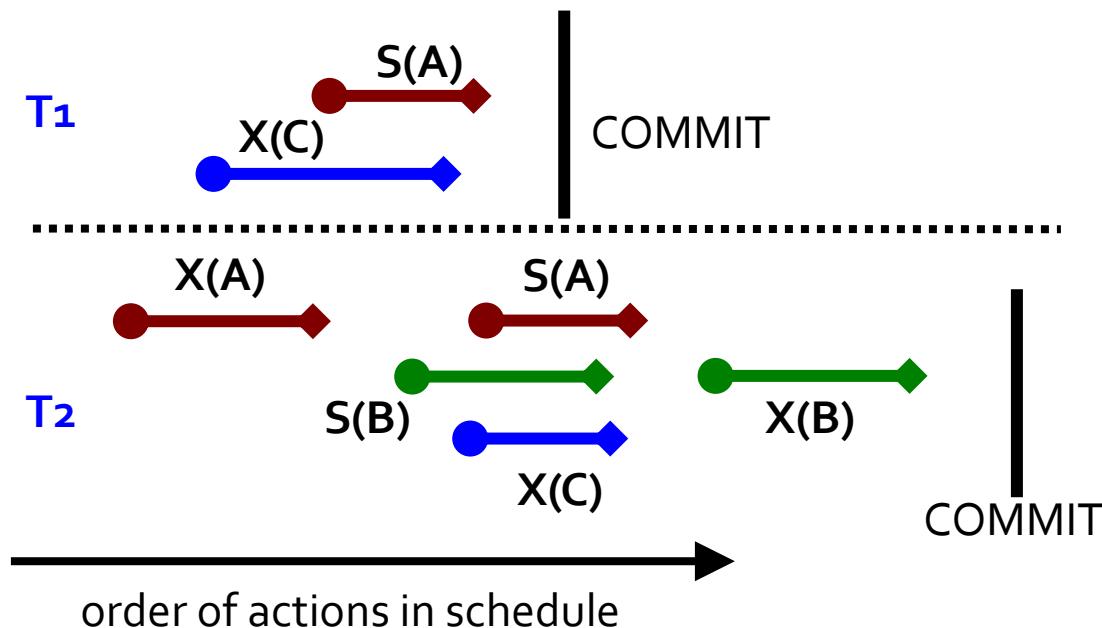


Locking protocols

- 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 is allowed only to the lock owner/creator
 - can be granted to just one transaction
- **shared locks**
 - **S(A)**, only reads of A allowed – the lock owner can read and is sure that A cannot change
 - can be granted to (shared by) multiple transactions (if X(A) not granted already)
- **unlocking by U(A)**
- if a lock is required for transaction that is not available (granted to another one), the transaction execution is suspended and wait for releasing the lock
 - in 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

T ₁	T ₂
X(C)	X(A)
R(C) W(C)	W(A)
S(A)	U(A)
R(A)	S(B)
U(C)	R(B)
U(A)	X(C)
COMMIT	S(A)
	X(A)
	S(B)
	X(C)
	S(A)
	X(B)
	COMMIT



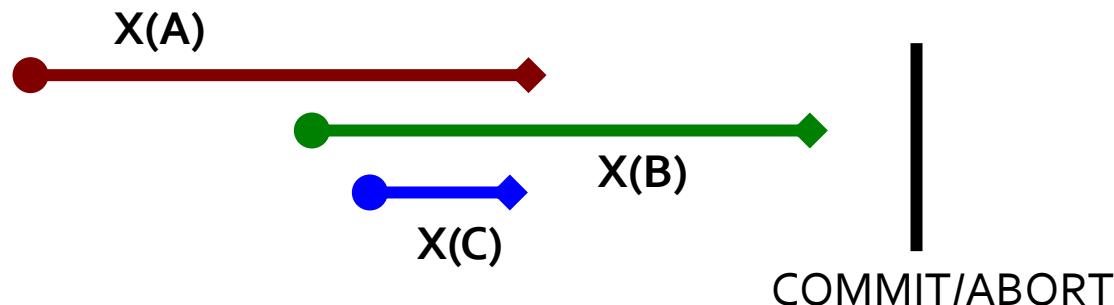
Two-phase locking protocol (2PL)

2PL 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 requests 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

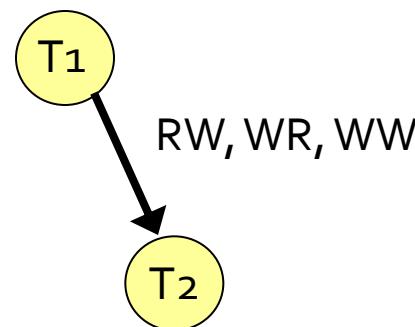
- the 2PL restriction of schedule ensures that the precedence graph is acyclic, i.e., the schedule is **conflict-serializable**
- 2PL does **not guarantee recoverable schedules**

Example: 2PL-compliant schedule, but not recoverable, if T1 aborts

T₁
X(A)
R(A)
W(A)
U(A)

T₂
X(A)
R(A)
A := A *1.01
W(A)
S(B)
U(A)
R(B)
B := B *1.01
W(B)
U(B)
COMMIT

ABORT / COMMIT

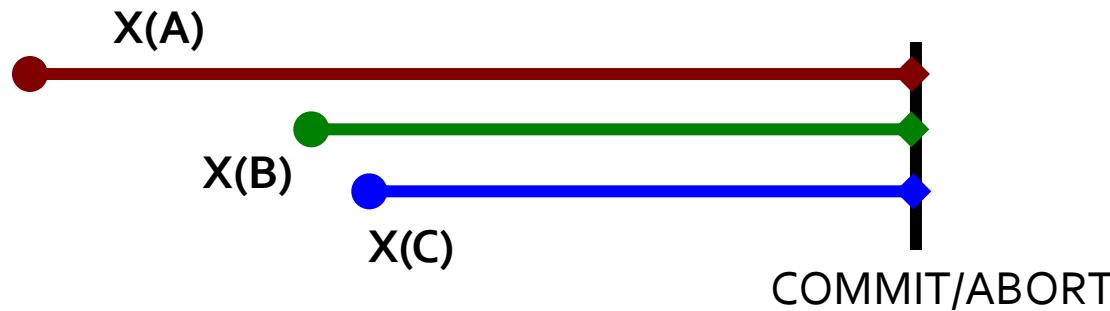


Strict 2PL

Strict 2PL 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

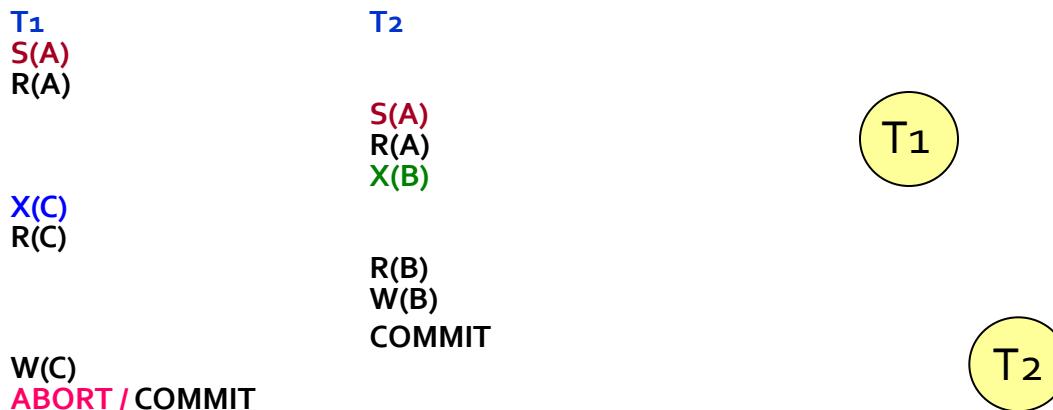


Insertions of U(A) is not needed (implicit at the time of COMMIT/ABORT).

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**

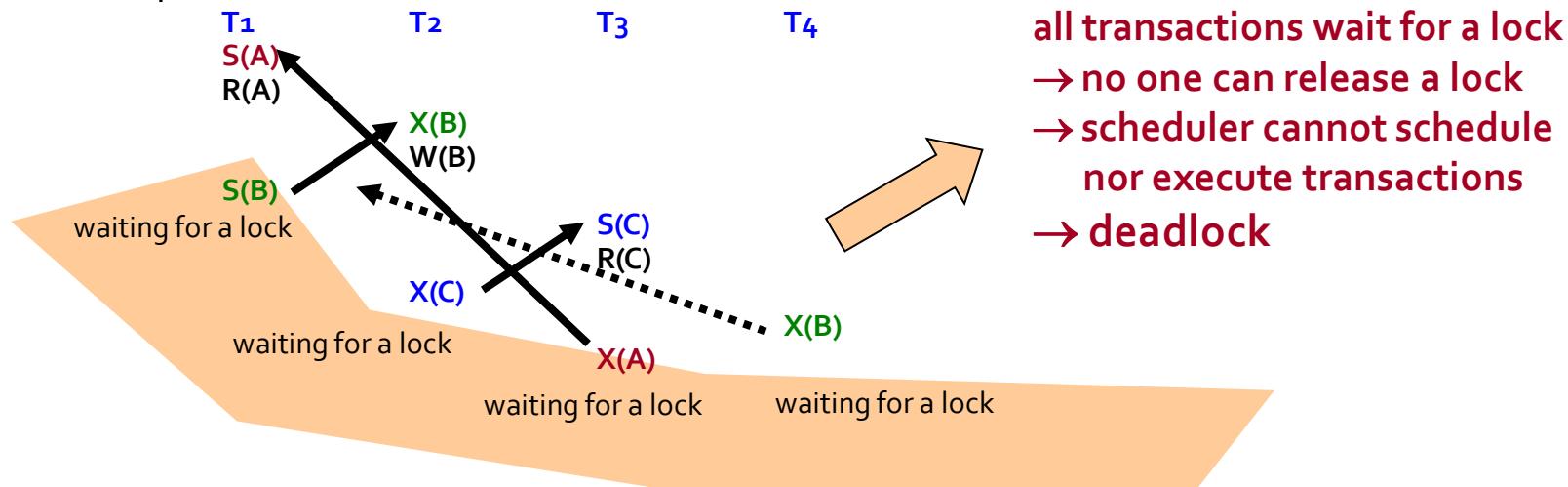
Example: schedule built using strict 2PL



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:

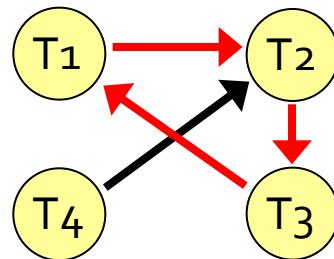


Deadlock detection

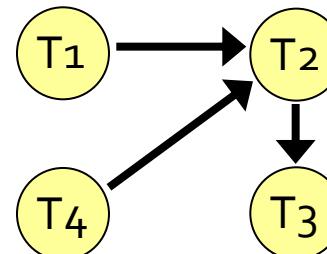
- deadlock can be detected by repeated checking the waits-for graph
- **waits-for graph** is dynamic graph that captures the waiting of transactions for locks
 - nodes are active transactions
 - an arc denotes waiting of transaction for lock kept by another transaction
 - a cycle in the graph = **deadlock**

Example: graph to 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 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
 - aborted transaction is not aborted again (if another deadlock occurs)
- deadlocks could be prevented
 - prioritizing
 - each transaction has a priority (e.g., time stamp), while if T_1 requests a lock kept by T_2 , the lock manager chooses between two strategies
 - **wait-die** – if T_1 has higher priority, it can wait, if not, it is aborted and restarted
 - **wound-wait** – if T_1 has higher priority, T_2 is aborted, otherwise T_1 waits
 - conservative 2PL protocol
 - all locks possibly used must be requested at the beginning
 - hard to guess all relevant lock, not used in practice for high locking overhead

Phantom

- now consider dynamic database, that is, allowing inserts and deletes
- if one transaction works with some *semantic set* of data entities, while another transaction (*logically*) changes this set (inserts or deletes), it could lead to inconsistent database (inserializable schedule)
 - why: T_1 locks all entities that at the given moment are relevant (e.g., fulfill some WHERE condition of a SELECT command)
 - during execution of T_1 a new transaction T_2 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(Paul), R(John)}

lock women, i.e.,

S(Dana)

F = max{R(Dana)}

Insert(M, F) // result is inserted into table Statistics

COMMIT

T₂

Insert(Phill, 72, m)

lock women, i.e.,

X(Eve)

X(Dana)

Delete(Eve)

COMMIT

phantom

new male employee can be inserted, although **all men** should be locked

Although the schedule is **strict 2PL** compliant, the result **[Paul, Dana]** is not correct as it does not follow the serial schedule T₁, T₂, resulting in **[Paul, Eve]**, nor T₂, T₁, resulting **[Phill, Dana]**.

Phantom – prevention

- if there exist indexes (e.g., B⁺-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
- if there do not exist indexes, everything relevant must be locked
 - e.g., entire table or even multiple tables must be locked
- 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

Isolation levels

- the more strict locking protocol, the worse performance of concurrent transaction management
 - for different reasons different protocols can be chosen, in order to achieve maximal performance for sufficient isolation of transactions
- SQL-92 defines isolation levels

Level	Protocol	WR	RW	Phantom
READ UNCOMMITTED (read only transactions)	No	maybe	maybe	maybe
READ COMMITTED	S2PL for X() + 2PL for S()	No	maybe	maybe
REPEATABLE READ	S2PL	No	No	maybe
SERIALIZABLE	S2PL + phantom prevention	No	No	No

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
 - **Read**: transaction reads data from database but writes into its private local data space
 - **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:
 - **Write**: the private data space is copied into the database

course:

Database Systems (NDBlo25)

SS2017/18

lecture 11:

Implementation of database structures

doc. RNDr. Tomáš Skopal, Ph.D.

RNDr. Michal Kopecký, Ph.D.

Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague

Today's lecture outline

- disk management, paging, buffer manager
- database files organization
- indexing
 - single-attribute indexes
 - B⁺-strom, bitmaps, hashing
 - multi-attribute indexes

Introduction

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

Example:

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

Paging

- records stored in disk pages of fixed size (a few kB)
- the reason is the hardware, still assuming 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)
- realtime 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 realtime!!!

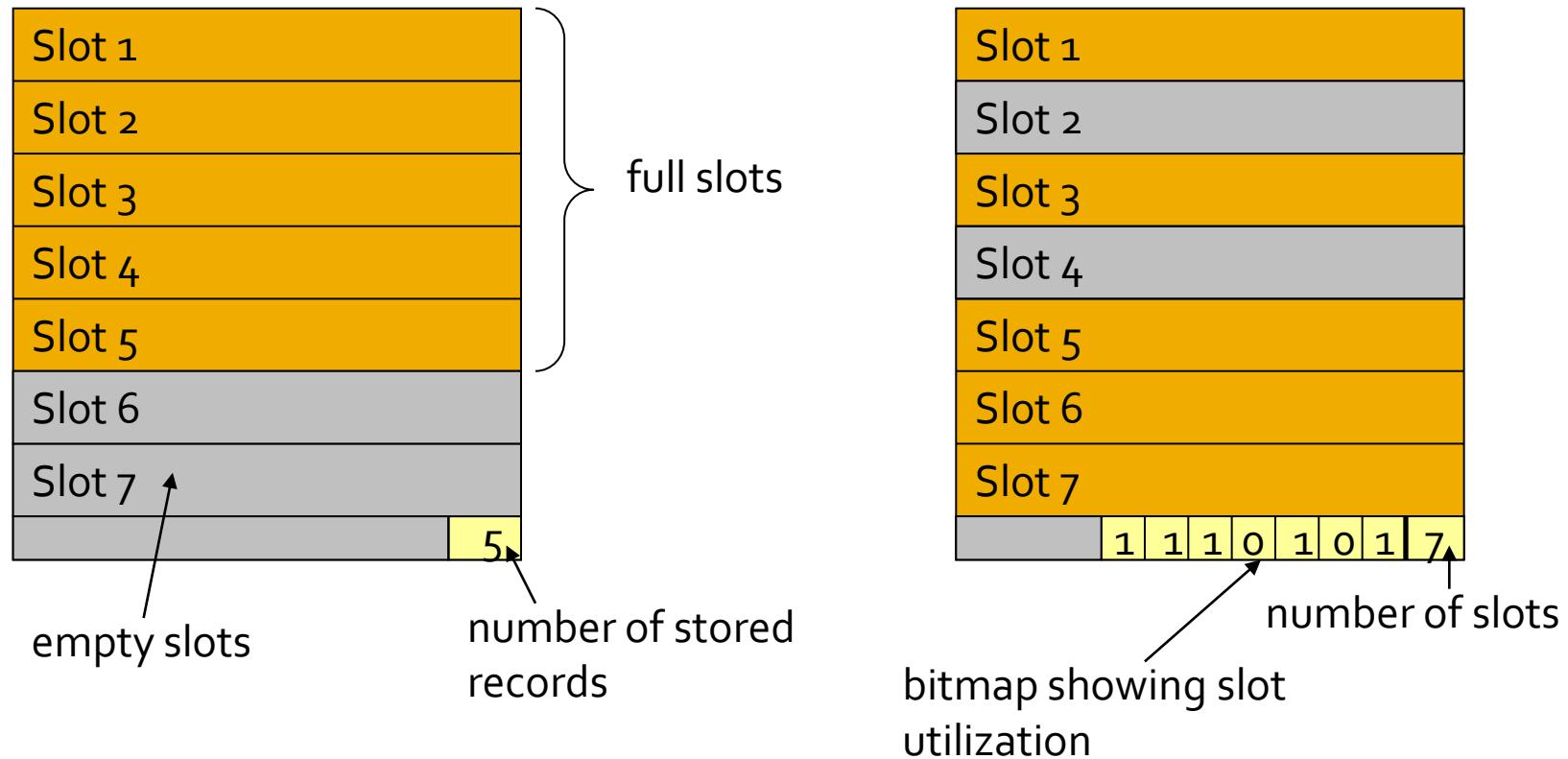
Paging, cont.

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

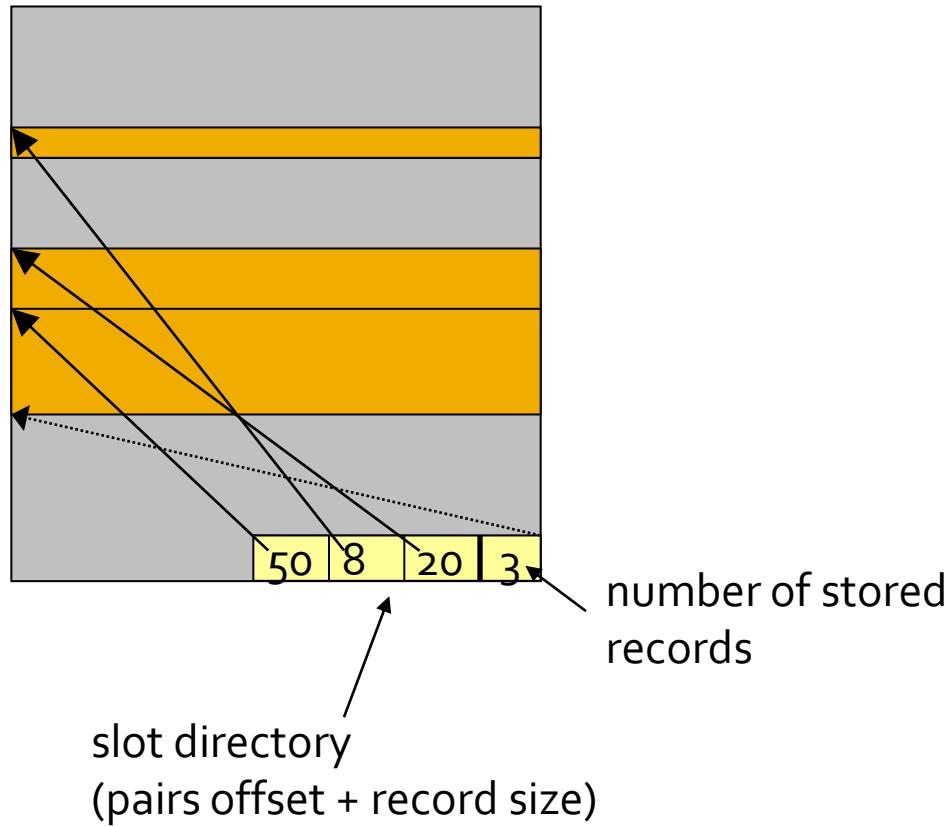
Paging, cont.

- if only fixed-size data types are used in the record
→ fixed record size
- if 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



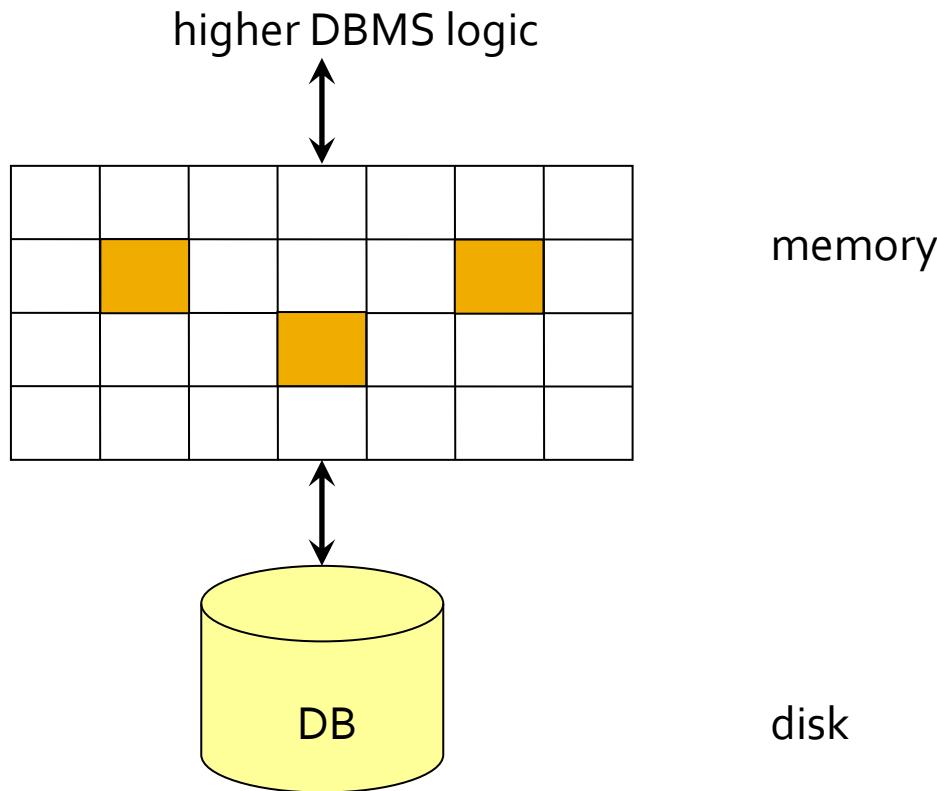
Variable-size page organization, example



Buffer management

- buffer = 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 a page in frame) and **dirty** (modified record)
- serves for speeding repeated access to pages – buffer manager implements the **read** and **write** operations
- abstraction of the higher DBMS logic from disk management
- implementation of **read** retrieves the page from buffer, if it is not there, it is first fetched from the disk, increasing **pin_count**
- implementation of **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),
 - if the replaced page is **dirty**, it must be stored

Buffer management



Database storage

- data files
(consisting of table data)
- index files
- system catalogue – contains metadata
 - table schemas
 - index names
 - integrity constraints, keys, etc.

Data files

- heap
- sorted file
- hashed file

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`

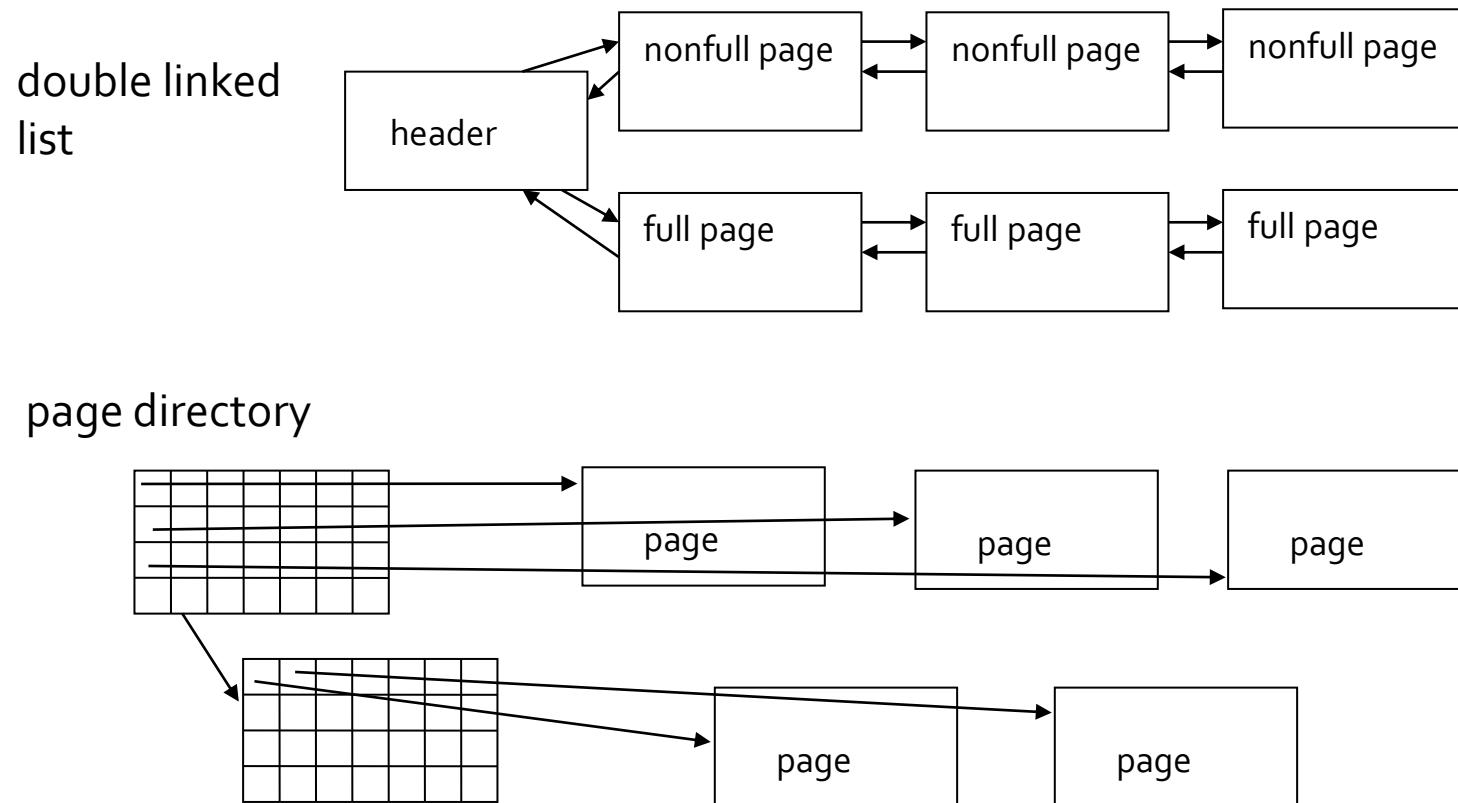
Heap file

- stored records in pages are not ordered, resp., 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 nonfull pages
- page directory
 - linked list of directory pages
 - every item in the directory refers to a data page
 - flag bit of each item utilization

Maintenance of empty heap pages



Heap, cost of simple operations

- sequential access = N
- search on equality = $0,5*N$ or 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

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“ with empty space
- allows fast search on equality and/or range
- slow insertion and deletion, „moving“ with the rest of pages
- a tradeoff used 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

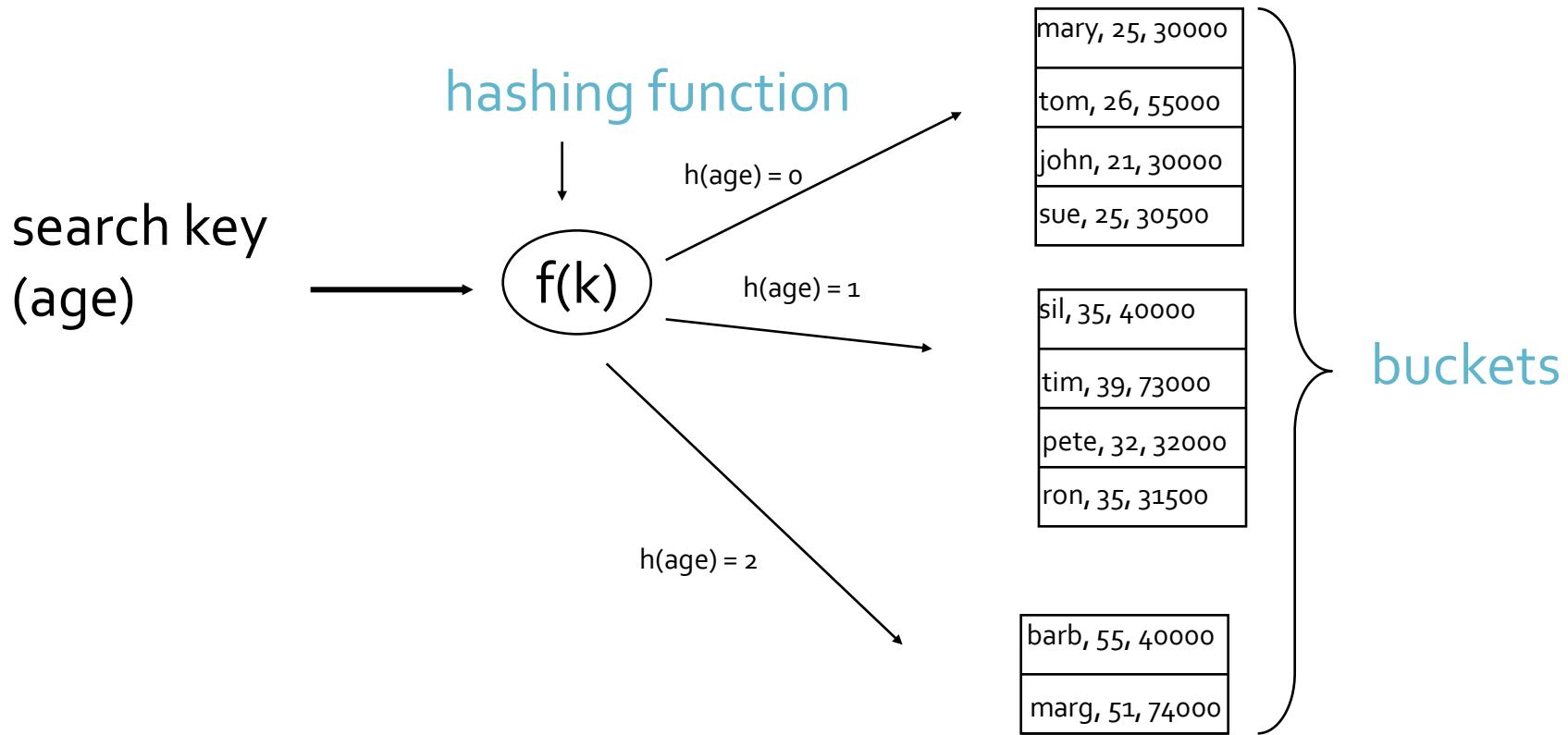
Sorted file, cost of simple operations

- sequential access = N
- search on equality = $\log_2 N$ or N
- search on range = $\log_2 N + M$
(where M is number of relevant pages)
- record insertion = N
- record deletion = $\log_2 N + N$ based on key,
otherwise $1,5 * N$ (**rid**)

Hashed file

- organized in K buckets, a bucket is extensible to multiple disk pages
- record is inserted into/read from bucket determined by a hashing function and the search key
 - bucket id = $f(key)$
- if the bucket is full, new pages are allocated and linked to the bucket (linked list)
- fast queries and 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)
- search on range = N
- record insertion = N/K (best case)
- deletion on equality = $N/K + 1$ (best case),
otherwise N

Indexing

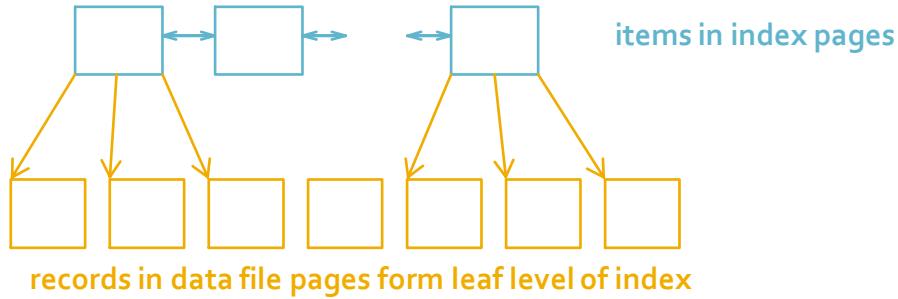
- index is a helper structure that provides fast search based on search key(s)
- organized into disk pages (like data files)
- usually different file than data files
- contains only search keys and links to the respective records (i.e., rid)
- need much less space than data files (e.g., 100x less)

Indexing, principles

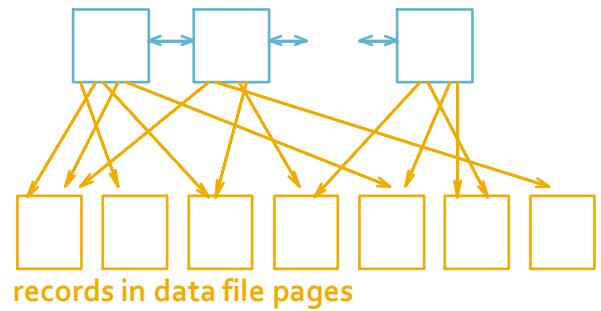
- index item could contain
 - the whole record (then index and data file are the same)
 - pair <key, rid>
 - pair <key, rid-list>, where rid-list is a list of links to records with the same search key value
- clustered vs. unclustered indexes
 - **clustered:** ordering of index items is (almost) the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

Indexing, principles

CLUSTERED INDEX



UNCLUSTERED INDEX



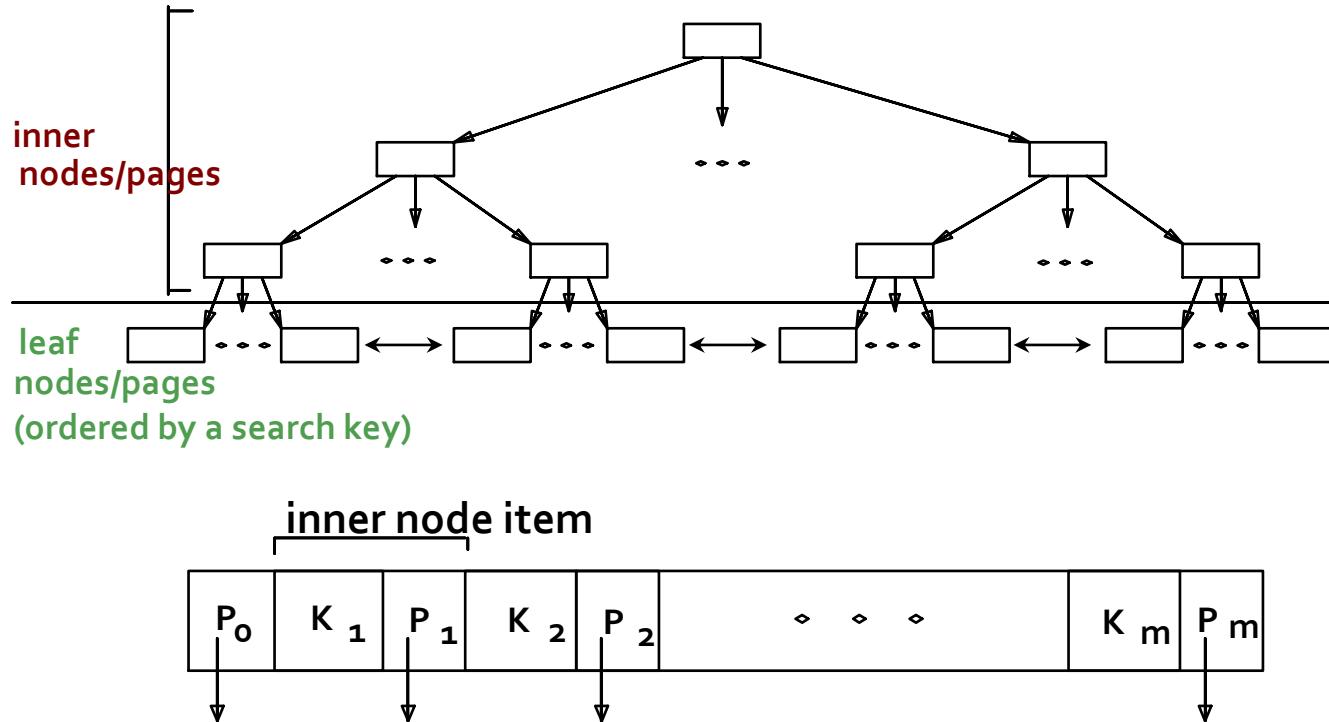
The pros of clustered index are huge speedup when searching on range, because the result record pages are read sequentially from data file. Unclustered index only allows read sequentially from index file, not from data file.

Cons: large overhead for keeping the data file sorted (even more expensive when applied to other indexes)

B⁺-tree

- follows B-tree - paged, balanced tree-based index (Rudolf Bayer, 1972).
- provides logarithmic complexity for insertion, search on equality (no duplicates), deletion on equality (no duplicates)
- guarantees 50% node (page) utilization
- B⁺-tree extends B-tree by
 - linking leaf pages for efficient range queries
 - inner nodes contain indexed intervals, i.e., all keys are in the leaves

B⁺-tree, schema



Demo: <http://slady.net/java/bt/>

Hashed index

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

Bitmaps

- suitable for indexing attributes of low-cardinality data types
 - suitable for, e.g., attribute **FAMILY_STATUS** = {single, married, divorced, widow}
 - not suitable for, e.g., attribute **PRODUCT_PRICE** (many values), better B-tree
- for each value **h** of an indexed attribute **a** a bitmap (binary vector) is constructed, where 1 on i^{th} position means the value **h** appears in the i^{th} record (in the attribute attribute **a**), while it holds
 - bitwise OR of all bitmap for an attribute forms only 1s (every attribute has a value)
 - bitwise AND of any two bitmaps for an attribute forms only zeros (attribute values are deterministic)

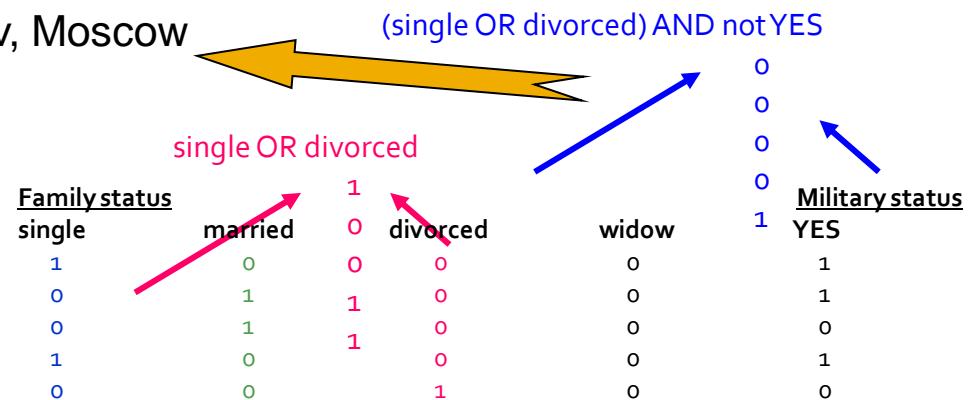
Name	Address	Family status	single	married	divorced	widow
John Smith	London	single	1	0	0	0
Rostislav Drobil	Prague	married	0	1	0	0
Franz Neumann	Munich	married	0	1	0	0
Fero Lakatoš	Malacky	single	1	0	0	0
Sergey Prokofjev	Moscow	divorced	0	0	1	0

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?**
(bitmap(single) OR bitmap(divorced)) AND not bitmap(YES)

answer: Sergey Prokofjev, Moscow

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



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 the linearly with the number of values in the range (bitmaps for all the values must be processed)

Multi-attribute indexing, examples

- consider conjunctive range query

```
SELECT * FROM Employees WHERE  
salary BETWEEN 10000 AND 30000 AND  
age < 40 AND
```

```
name BETWEEN 'Dvořák' AND 'Procházka'
```

- simple solutions using B^+ -tree

(number of indexed attributes $M = 3$)

- 1) M standalone indexes

- 2) one index of M concatenated attributes

- both solutions bad (the second is slightly better)

Multi-attribute indexing, examples

Three standalone indexes:

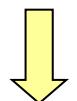
...WHERE salary BETWEEN 10000 AND 30000 AND age < 40 AND name BETWEEN 'Dvořák' AND 'Procházka'

r1	Čech Jaroslav	17000	27	r7	Oplustil Arnošt	9000	36	r6	Novák Karel	13000	19
r2	Dostál Jan	21000	33	r6	Novák Karel	13000	19	r5	Novák Josef	32000	25
r3	Malý Zdeněk	15000	45	r10	Zlámal Alois	13000	52	r1	Čech Jaroslav	17000	27
r4	Mrázek František	22000	37	r3	Malý Zdeněk	15000	45	r2	Dostál Jan	21000	33
r5	Novák Josef	32000	25	r1	Čech Jaroslav	17000	27	r7	Oplustil Arnošt	9000	36
r6	Novák Karel	13000	19	r8	Papoušek Jindřich	19000	50	r4	Mrázek František	22000	37
r7	Oplustil Arnošt	9000	36	r2	Dostál Jan	21000	33	r9	Richter Tomáš	26000	41
r8	Papoušek Jindřich	19000	50	r4	Mrázek František	22000	37	r3	Malý Zdeněk	15000	45
r9	Richter Tomáš	26000	41	r9	Richter Tomáš	26000	41	r8	Papoušek Jindřich	19000	50
r10	Zlámal Alois	13000	52	r5	Novák Josef	32000	25	r10	Zlámal Alois	13000	52

{r3, r4, r5, r6, r7, r8}

{r6, r10, r3, r1, r8, r2, r4, r9}

{r6, r5, r1, r2, r7, r4}



intersection = {r4, r6}

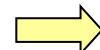
Implementation of database structures
(NDBlo25, Lect. 11)

Multi-attribute indexing, examples

Index of concatenated attributes:

...WHERE salary BETWEEN 10000 AND 30000 AND age < 40 AND name BETWEEN 'Dvořák' AND 'Procházka'

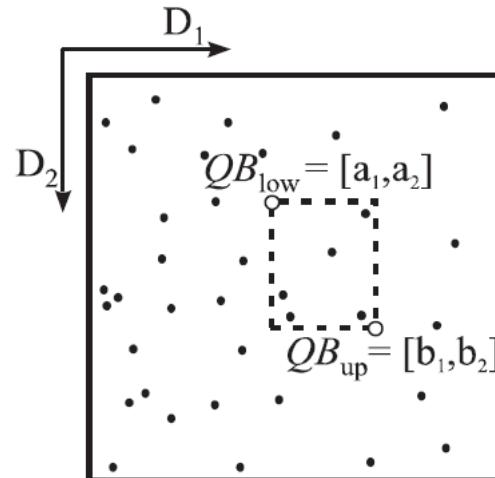
r1	Čech Jaroslav	17000	27
r2	Dostál Jan	21000	33
r3	Malý Zdeněk	15000	45
r4	Mrázek František	22000	37
r5	Novák Josef	32000	25
r6	Novák Karel	13000	19
r7	Oplustil Arnošt	9000	36
r8	Papoušek Jindřich	19000	50
r9	Richter Tomáš	26000	41
r10	Zlámal Alois	13000	52



{r4, r6}

Spatial indexing

- abstraction of M-tuples of keys as M-dimensional vectors
 $\langle \text{Novák Josef}, 32000, 25 \rangle \Rightarrow [\text{sig}(\text{'Novák Josef'}), 32000, 25]$
 - key ordering must be preserved, e.g., for $\text{sig}(*)$
- “geometerization” of the problem as searching in M-dimensional space R^M
- conjunctive range query = (hyper)-rectangle QB in space R^M delimited by two points, intervals in all dimensions



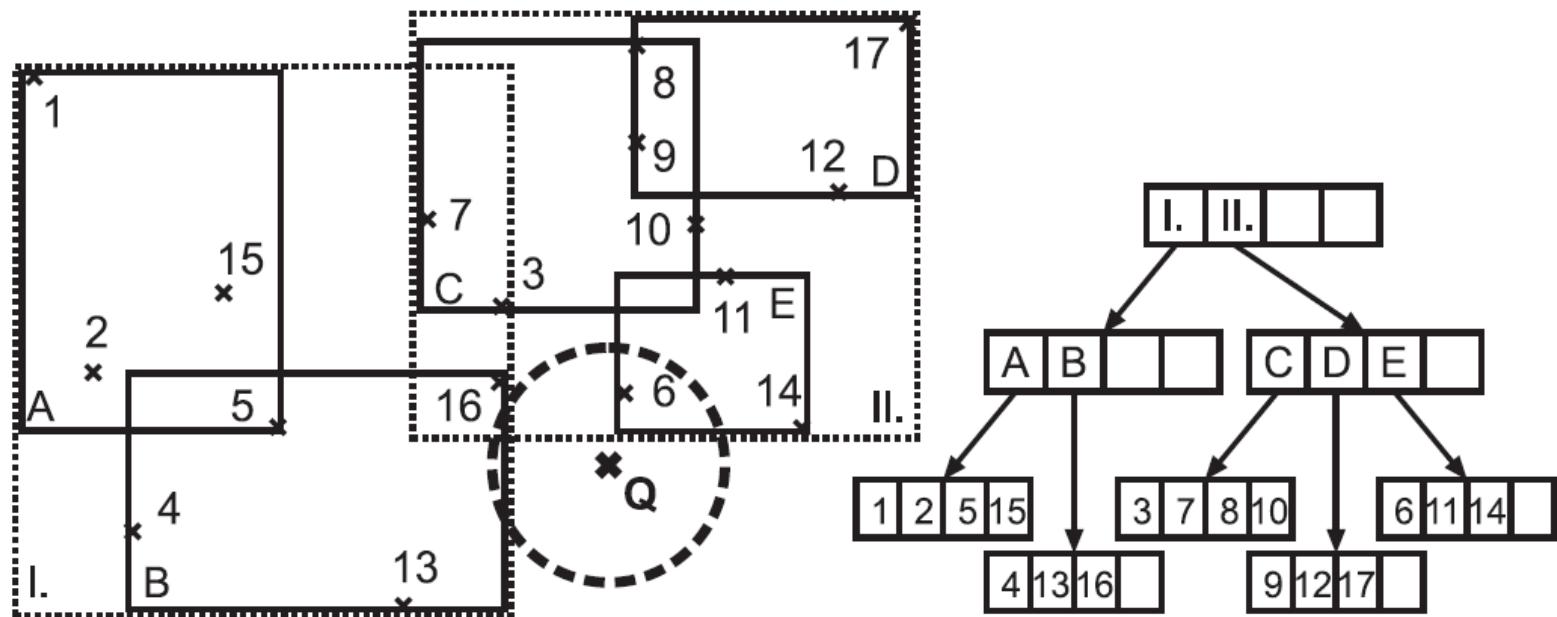
Spatial indexing

- spatial access methods (SAM), points access methods (PAM)
 - various methods and indexing structures for speeding spatial queries
 - tree-based, hashing-based and sequential indexes
- (non-sequential) indexes use clustering of the close data within index pages
 - data close in space are also close in the index
- in ideal case, during range query only those index pages are accessed that contain the relevant index keys
- work well for a few dimensions (up to 10), for higher dimensions the sequential approaches are better

Spatial indexing

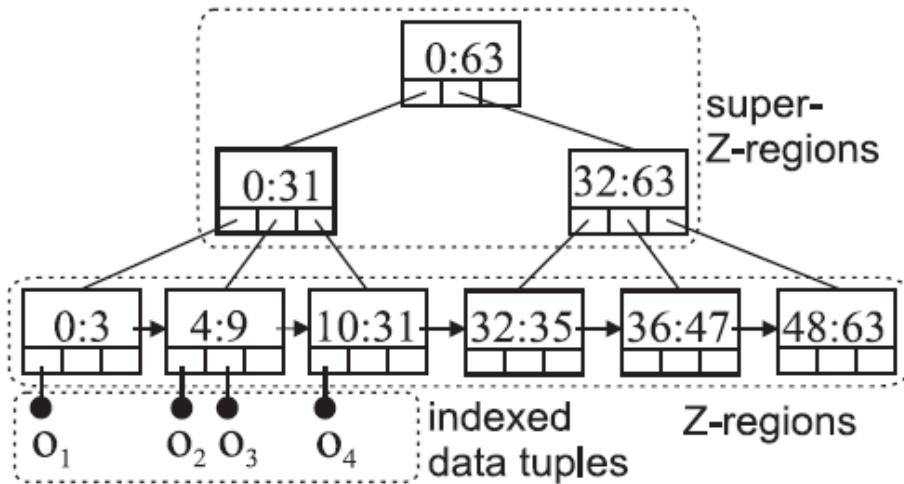
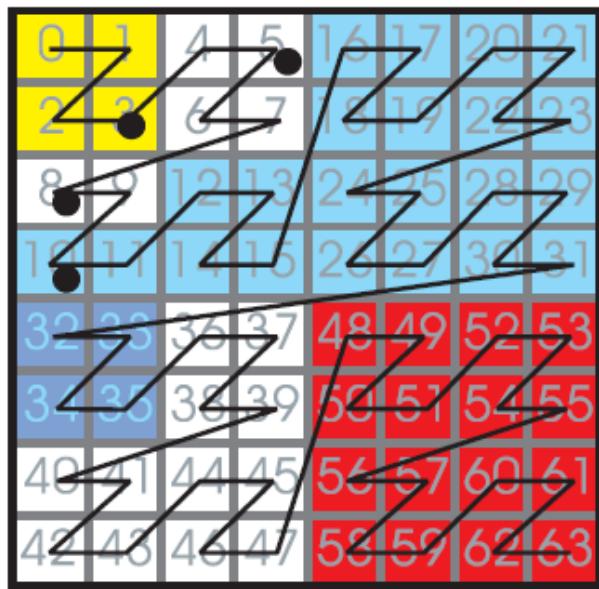
- tree indexes
 - R-tree, UB-tree, X-tree, etc.
- hashed indexes
 - Grid file
- sequential indexes
 - VA-file

R-tree



Demo: <http://www.dbnet.ece.ntua.gr/~mario/rtree/>

UB-tree



Grid file

