Advanced Database Models

Dr.-Ing. Eike Schallehn

OvG Universität Magdeburg
Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

2019

Organization

- Weekly lecture
  - http://www.dbse.ovgu.de/dbse/en/Lectures/Lehrveranstaltungen/Advanced+Database+Models.html
- Weekly exercises (3 alternative slots, currently)
  - Starting April 15th
- Exams
  - Written exams of 120 minutes at the end of lecture period

History of this Lecture

- Previous titles
  - Object-oriented Database Systems
  - Object-relational Database Systems
- Latest addition: semi-structured data models
- Previously held and refined by
  - Gunter Saake
  - Can Türker
  - Ingo Schmitt
  - Kai-Uwe Sattler

Overview of the Lecture

- Data and design models
  - Recapitulation: the relational model
  - NF^2 and eNF^2
  - Object-oriented models
  - Object-relational models
  - Models for semi-structured data (XML)
- For these models we discuss
  - Theoretical foundations
  - Query Languages
  - Application
  - Standards and systems

English Literature /1

- Stonebraker, M.; Moore, D.: *Object-Relational DBMSs: The Next Great Wave*, Morgan Kaufmann Publishers, 1999
- Date, C. J.; Darwen, H.: *Foundation for Object / Relational Databases: The Third Manifesto*, Addison-Wesley, 1998
- Melton, J.:*Advanced SQL: 1999 - Understanding Object-Relational and Other Advanced Features*,Morgan Kaufmann, 2002

English Literature /2

- Russel, C. et al.: *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann, 2000
- Chaudhri, A. B., Rashid, A.; Zicari, R.: *XML Data Management: Native XML and XML-Enabled Database Systems*,Addison-Wesley, 2003
- Elmasri, R.; Navathe, S.: *Fundamentals of Database Systems*, Addison Wesley, 2003

German Literature

- Türker, C.; Saake, G.: *Objektrelationale Datenbanken*, dpunkt.verlag, Heidelberg 2006
- Türker, C.: *SQL:1999 & SQL:2003*, dpunkt.verlag, Heidelberg, 2003
- Klettke, M.; Meyer, H.: *XML und Datenbanken*, dpunkt.verlag, Heidelberg, 2003
- Saake, G.; Türker, C.; Schmitt, I.: *Objektdatenbanken*, Int. Thomson Publishing, 1997
- Elmasri, R.; Navathe, S.: *Grundlagen von Datenbanksystemen*, Pearson Studium, 2002

Teil I

Introduction

Overview

- Some Basic Terms
- History of Database Models
- Problems with RDBMS
- Aspects of Advanced Data Models

Some Basic Terms

- Data Model
- Database Model
- Database
- Database System
- Database Management System
- Conceptual Model
- Database Schema(s)

Data Model

A **data model** is a model that describes in an abstract way how data is represented in an information system or a database management system.

- Generic term includes models of
  - Programming languages
  - Database systems
  - Conceptual design methods
- Often (but not here) ambiguously used for schemata

Data Model /2

- A data model
  - is a system of concepts and their interrelations
  - is the “‘language” used to describe data
  - defines syntax and semantic of data
  - is the fundamental to other aspects of systems such as integrity and operations for access and manipulation

Data Models: Examples

The data model of the Java programming language allows structuring data as objects of classes consisting of attributes of basic data-types or references to other objects, specifying methods to access the data, etc.

The relational database model allows structuring data as tables of tuples with attributes, foreign keys, integrity constraints, etc.

The ER model for conceptual design describes data as instances of entity types, relationship types with cardinalities, attributes, etc.

Database Model

A **database model** is a data model for a database system. It provides a theory or specification describing how a database is structured and used.

- Tightly coupled with database management system (DBMS), i.e. a database management system usually implements one (or more) data models

Database Models: Examples

- Hierarchical model
- Network model
- Relational model
- NF^2 and eNF^2
- Object-Relational model
- Object-oriented model (or short, object model)
- Multi-dimensional model
- Semi-structured model
- Multimedia data model
- Spatio-temporal data model
- ...

Database Management Systems

A **database management system (DBMS)** is a suite of computer programs designed to manage a database and run operations on the data requested by numerous clients.

A **database (DB)** is an organized collection of data.

A **database system (DBS)** is the concrete instance of a database managed by a database management system.

Database Management Systems: Examples

- Current database management systems such as
  - Oracle DBMS
  - IBM DB2
  - Microsoft SQL Server

  are mainly based on the **object-relational** model with extensions / extensibility for
  - Multi-dimensional data (Data Warehouse)
  - Multimedia data (Image, Video, Audio, ...)
  - Semi-structured data (HTML, XML)
  - ...
- Specific other systems:
  - Object-oriented database systems
  - XML-database systems
  - ...

Database Systems: Examples

The EBay Web-database running on several servers managing the offered items, the users, and their activities.

The Wallmart Data Warehouse managing information about customer behavior from all their supermarkets and supporting complex analytical tasks.

The student database of our university running on a server in the students department managing the identity, credits, status, etc. of our students.

Database Systems: Examples /2

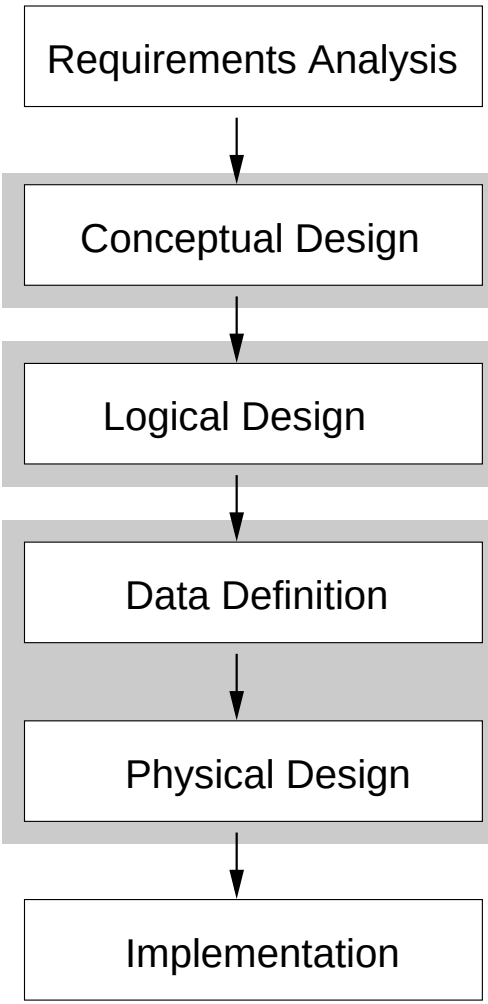
- Database systems in almost all larger companies and organizations as the core components of systems supporting
  - Human resource management
  - Customer relationship management
  - Product development / construction
  - Collaborative work
  - Sales and marketing
  - Decision and management support
  - ...

Conceptual Models

A **conceptual model** is a data model or diagram for high-level descriptions of schemata independently of their implementation. It often provides a graphical notation and is used in the first stages of information-system design (conceptual design).

- Entity-relationship model for (relational) DB design
- Unified Modeling Language (UML) for application design, but gaining support in database design

Database Design Process



Conceptual Models
e.g. ER or UML

Database Models
e.g. RM, ORM, OO

Concrete implementation of a database model, e.g. the object−relational model of the Oracle 10g DBMS

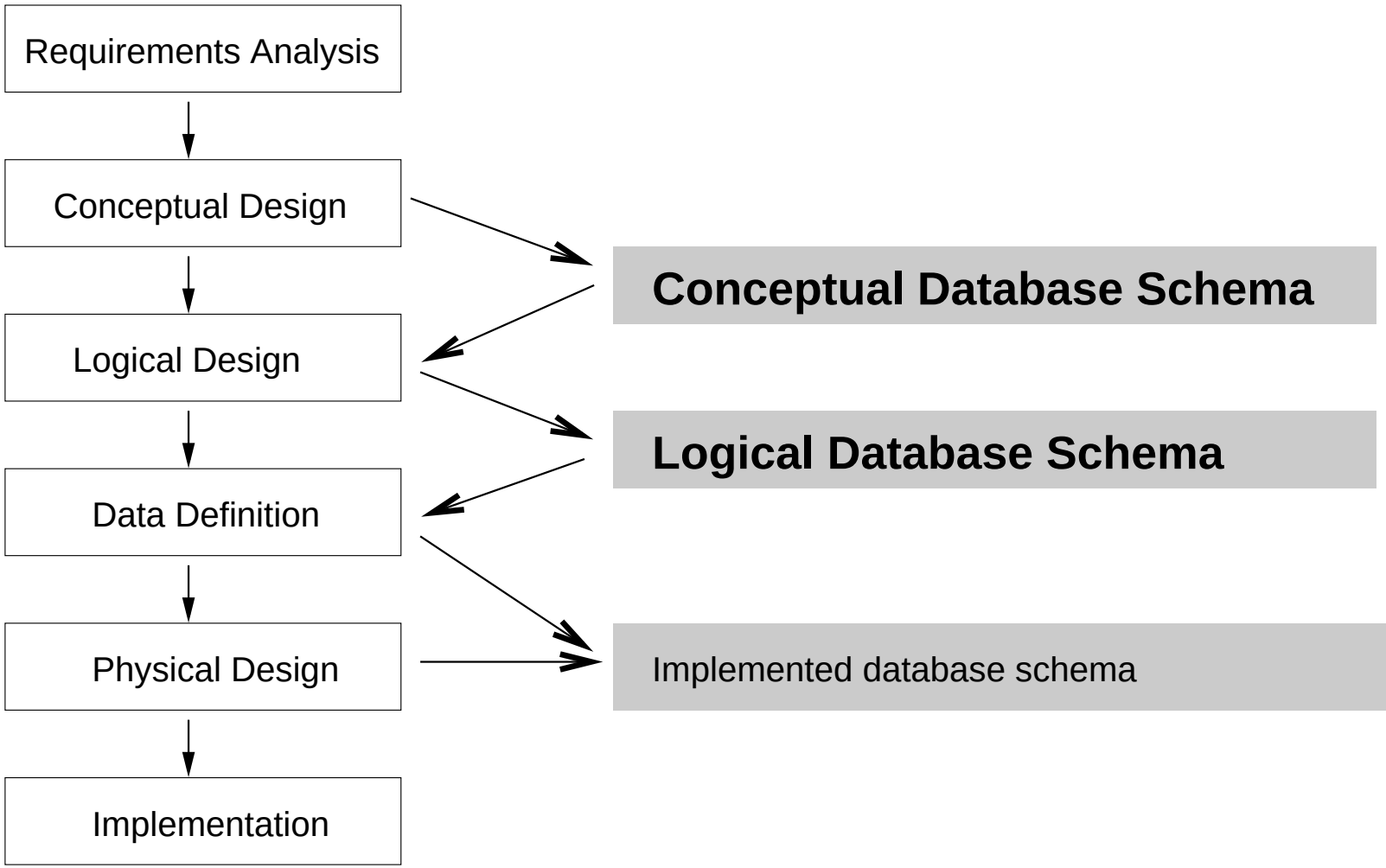
Database Schema

A **database schema** is a map of concepts and their relationships for a specific universe of discourse. It describes the structure of a database.

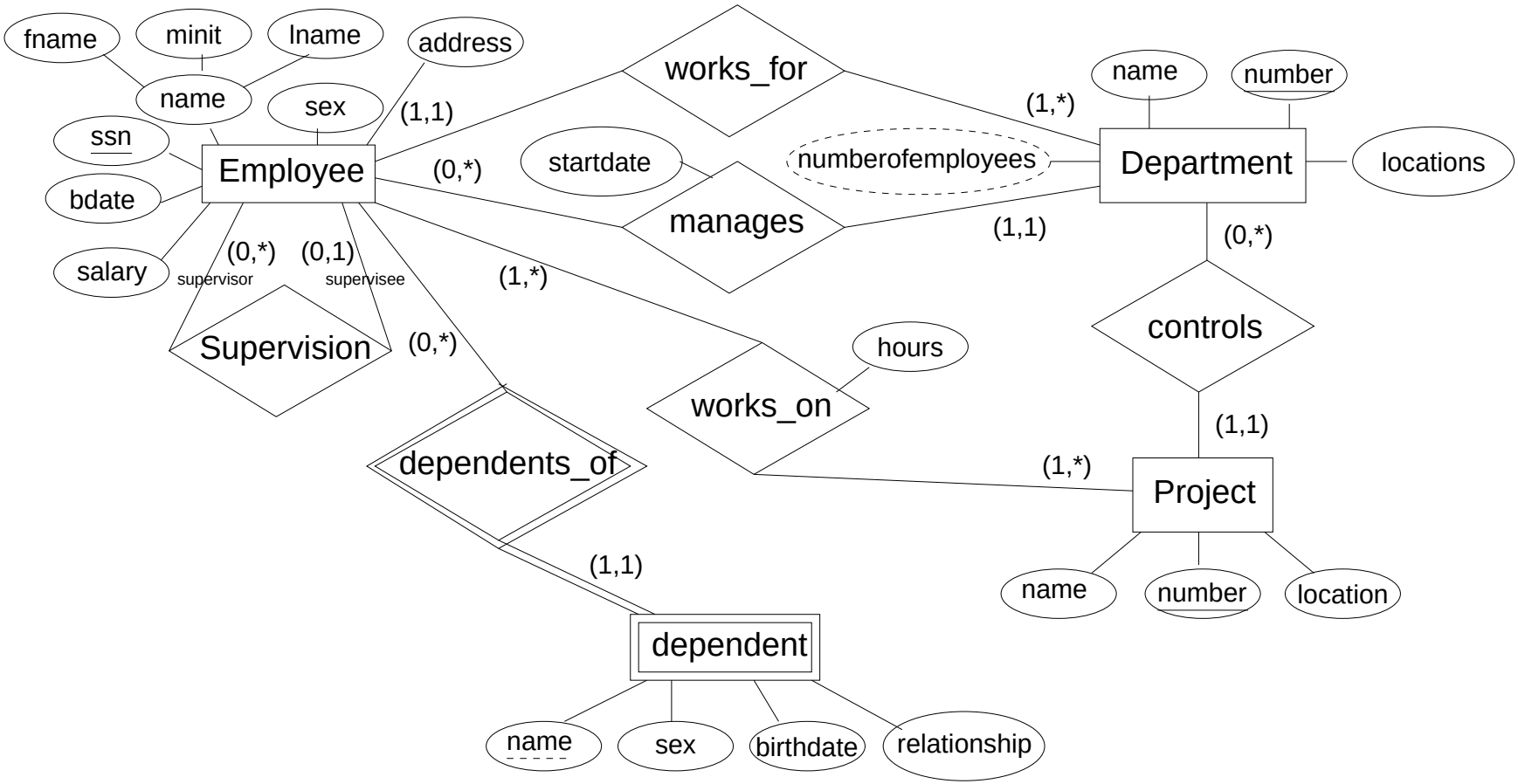
A **conceptual database schema** is a high-level and implementation-independent database schema expressed using the constructs of a conceptual data model.

A **logical database schema** is a detailed database schema expressed using the constructs of a database model and is ready for implementation.

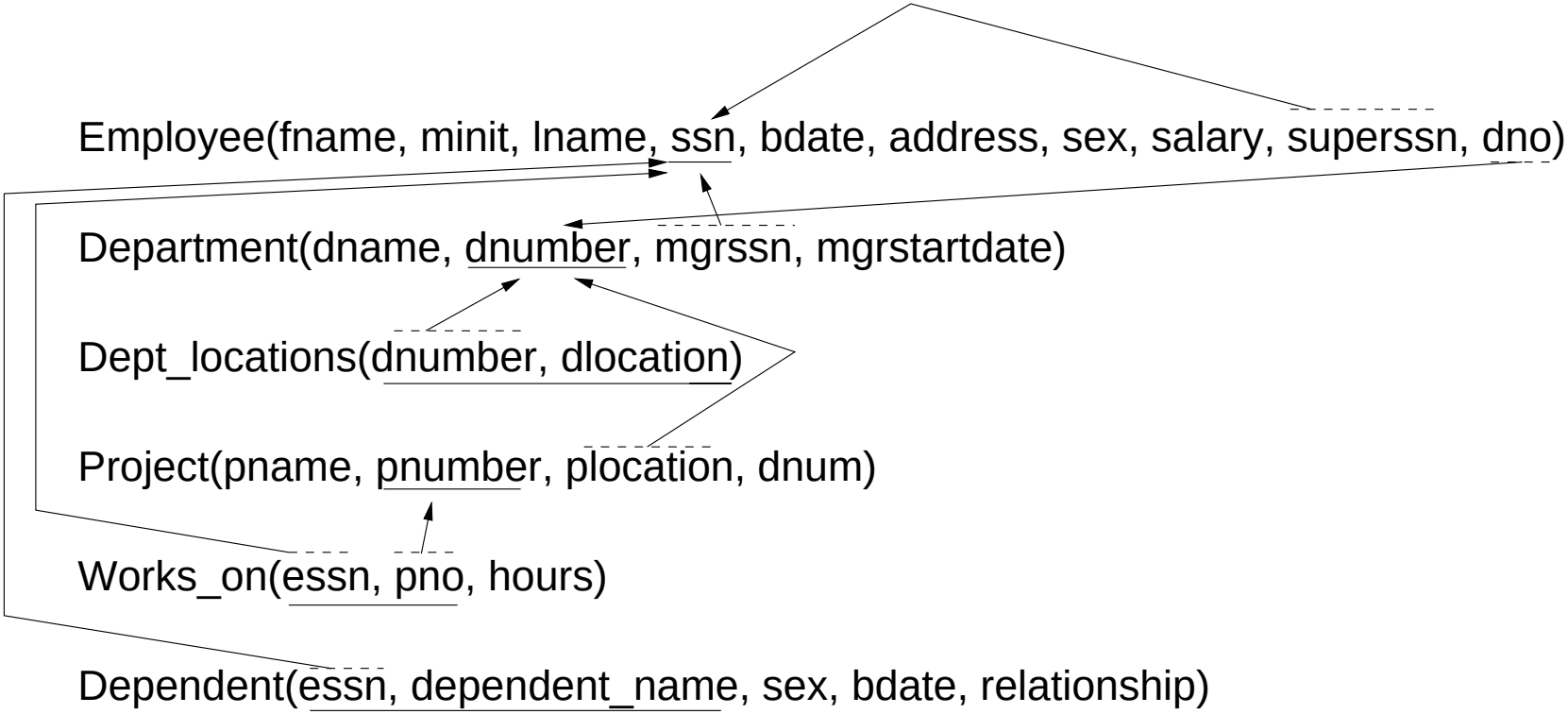
Database Design Process /2



Conceptual Database Schema: Example



Logical Database Schema: Example



Database: Example

EMPLOYEE									
FNAME	**MINIT**	**LNAME**	**SSN**	**BDATE**	**ADDRESS**	**SEX**	**SALARY**	**SUPERSSN**	**DNO**
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-07-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	null	1

DEPARTMENT			
DNAME	**DNUMBER**	**MGRSSN**	**MGRSTARTDATE**
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS	
DNUMBER	**DLOCATION**
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

Database: Example /2

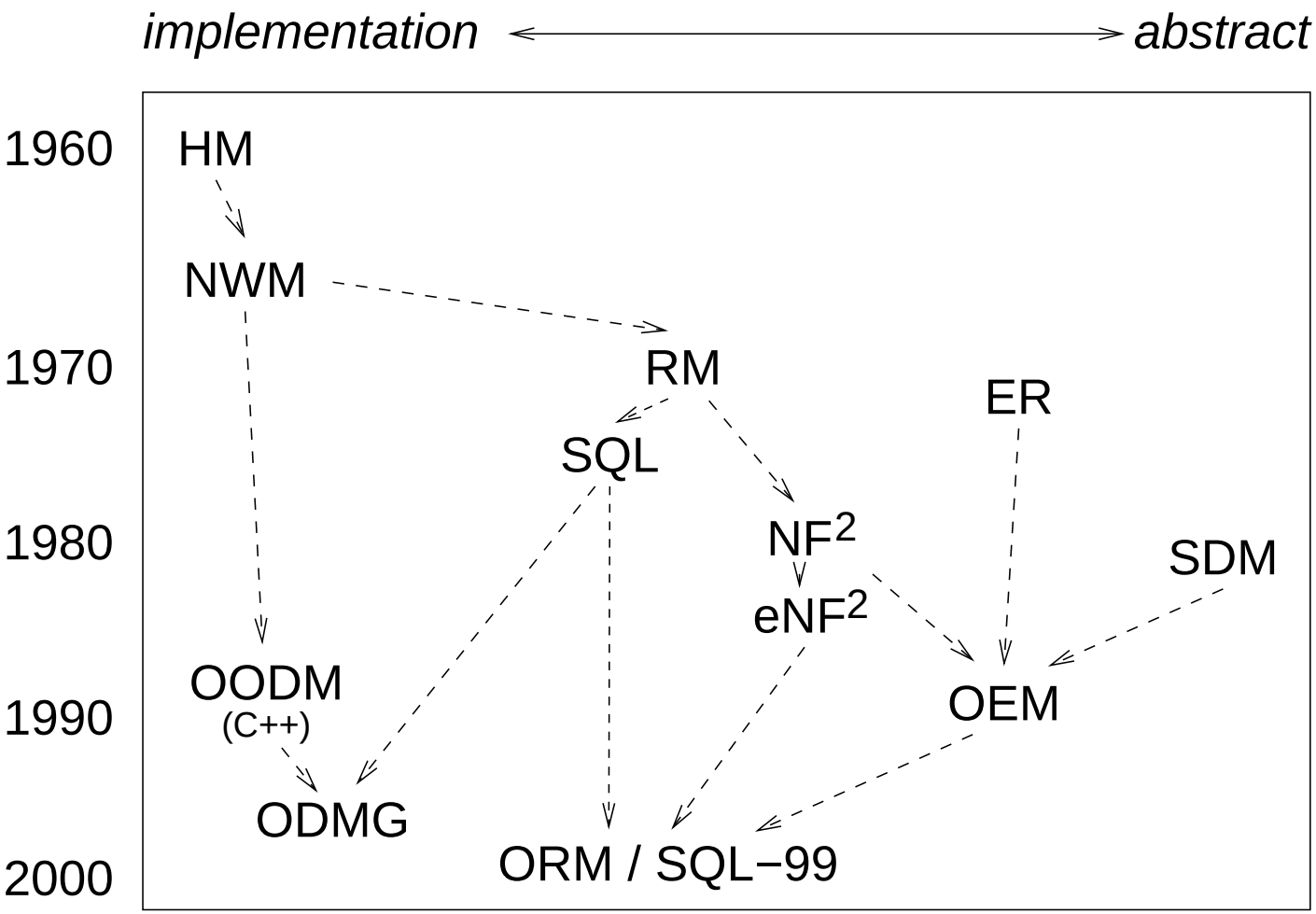
WORKS_ON		
ESSN	**PNO**	**HOURS**
123456789	1	32,5
123456789	2	7,5
666884444	3	40,0
453453453	1	20,0
453453453	2	20,0
333445555	2	10,0
333445555	3	10,0
333445555	10	10,0
333445555	20	10,0
999887777	30	30,0
999887777	10	10,0
987987987	10	35,0
987987987	30	5,0
987654321	30	20,0
987654321	20	15,0
888665555	20	null

PROJECT			
PNAME	**PNUMBER**	**PLOCATION**	**DNUM**
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

Database: Example /3

DEPENDENT				
ESSN	**DEPENDENT_NAME**	**SEX**	**BDATE**	**RELATIONSHIP**
333445555	Alice	F	1986-04-05	DAUGHTER
333445555	Theodore	M	1983-10-25	SON
333445555	Joy	F	1958-05-03	SPOUSE
987654321	Abner	M	1942-02-28	SPOUSE
123456789	Michael	M	1988-01-04	SON
123456789	Alice	F	1988-12-30	DAUGHTER
123456789	Elizabeth	F	1967-05-05	SPOUSE

History of Database Models

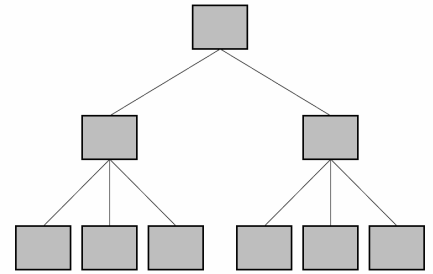


Database Models, Conceptual Models, Standards

- Database models
  - Hierarchical model (HM)
  - Network model (NM)
  - Relational model (RM)
  - (extended) Non-first normal form (NF^2 and eNF^2)
  - Object-relational model (ORM)
  - Object-oriented database model (OODM)
- Conceptual Models
  - Entity-relationship model (ER)
  - Semantic data models (SDM)
- Standards
  - Object Data Management Group (ODMG)
  - Structured Query Language (SQL:1999 and SQL:2003)

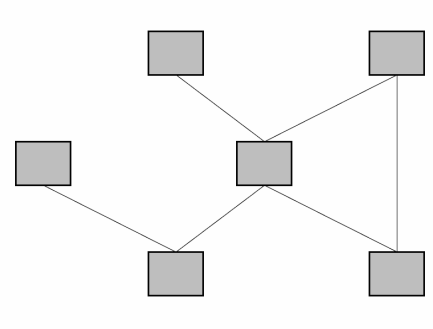
The Hierarchical Model

- First important database model
- Developed in the late ’50s
- Data organized as tree-structured networks of flat records
- IBM Information Management System (IMS) one of the most successful DBMS
  - Developed since 1966
  - Used until today
- Important concepts of hierarchical model still relevant for XML and semi-structured data



The Network Model

- Developed in the late ’60s by Charles Bachman
- Database extension to the data model of the successful COBOL programming language
- Also know as CODASYL (COnference on DAta SYstems Languages) database model
- Data organized as arbitrary networks of flat records
- Became an official standard in 1969
- Influenced object-oriented data(base) models developed later on



The Relational Model

- Developed by Edgar F. Codd (1923-2003) in 1970
- Derived from mathematical model of n-ary relations
- Colloquial: data is organized as tables (relations) of records (n-tuples) with columns (attributes)
- Currently most commonly used database model
- Relational Database Management Systems (RDBMS)
- First prototype: IBM System R in 1974
- Implemented as core of all major DBMS since late ’70s: IBM DB2, Oracle, MS SQL Server, Informix, Sybase, MySQL, PostgreSQL, etc.
- Database model of the DBMS language standard SQL

New Applications

- Concepts of RDBMS developed for classic database applications, e.g.
  - Administration
  - Banking
  - Sales / Customer Relations
- Mass data in new applications
  - Engineering / product development
  - Telecommunications
  - Multimedia systems
  - Document management / WWW

  introduce new requirements

More Complex and Flexible Structures

- Real-world objects can not be described by flat records
  - Consist of other objects
  - Contain sets or lists of other objects
  - Complex relationships to other objects
- Real-world objects have no fixed structure
  - Properties may be optional
  - Properties / parts may have varying cardinality
  - Properties (or their consideration) can be added or removed

More Semantic Concepts

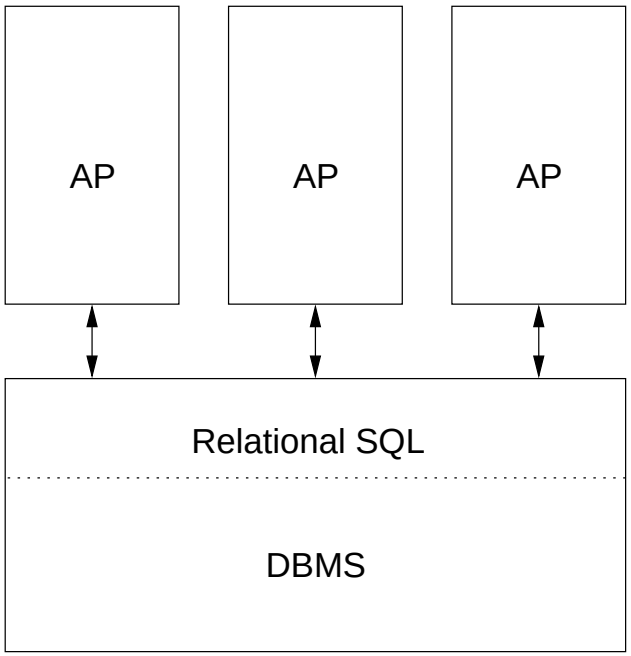
- Typical abstraction concepts
  - Specialization / Generalization
  - Aggregation
  - Association
- Differing representations of objects
  - Temporal: versions
  - Conditional alternatives: variants
- Object identity: the property of an object to be itself independently of changing values

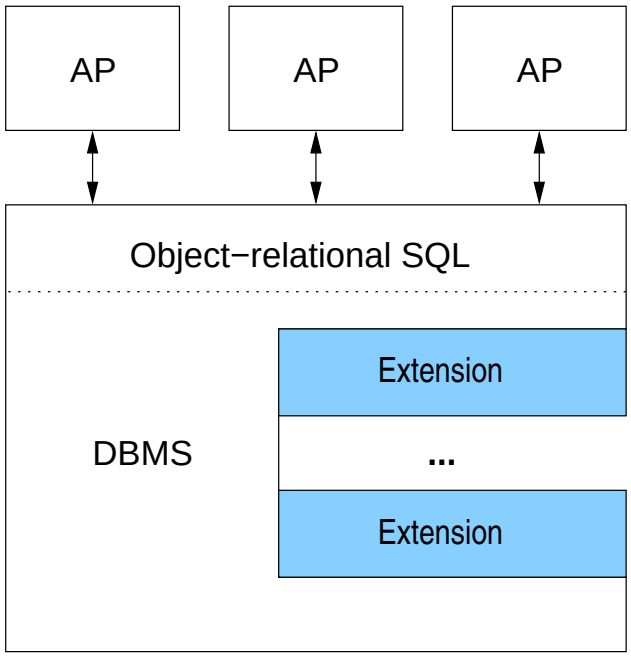
Description of Behavior

- Relational model: focus on structure of data
- Integrity constraints only way to restrict dynamics
- Operations and transactions are independent of application
- In programming combined view of data and behavior common practice
  - Methods for access and modification
  - Constructors / destructors
  - Encapsulation
  - Processes composition

Extensions and Extensibility

Support for new (e.g. multimedia) or application-specific data types





Impedance Mismatch

The (object-relational) **impedance mismatch** is the difference of data models used in programming and in database technology.

- Derived from electrical engineering
- Involved data models
  - Programming: nowadays mostly object-oriented
  - Databases: mostly relational
- The same application objects are
  - Persistently stored in the database
  - Transiently processed by the application
- Transformations in both directions required

Impedance Mismatch /2

```
public class Person {
    String firstname;
    String lastname;
}

public class Employee
    extends Person {

    Employee(...) {
        ...;
    }
}
```

```
SELECT FNAME, LNAME
FROM EMPLOYEE
```

RESULT:EMPLOYEE	
FNAME	**LNAME**
John	Smith
Franklin	Wong
Alicia	Zelaya
Jennifer	Wallace
Ramesh	Narayan
Joyce	English
Ahmad	Jabbar
James	Borg

Aspects of Advanced Data Models

- More
  - complex,
  - flexible, and
  - meaningful

  data structures
- Integration of behavior
- Extensions and extensibility
- Solving the “impedance mismatch” problem

Teil II

Recapitulation: Relational Database Systems

Overview

- Basic Principles of Database Systems
- The Relational Database Model
- The Relational Algebra
- Relational Database Management Systems
- SQL

Database Management Systems (DBMS)

- Nowadays commonly used
  - to store huge amounts of data persistently,
  - in collaborative scenarios,
  - to fulfill high performance requirements,
  - to fulfill high consistency requirements,
  - as a basic component of information systems,
  - to serve as a common IT infrastructure for departments of an organization or company.

Codd's 9 Rules for DBMS /1

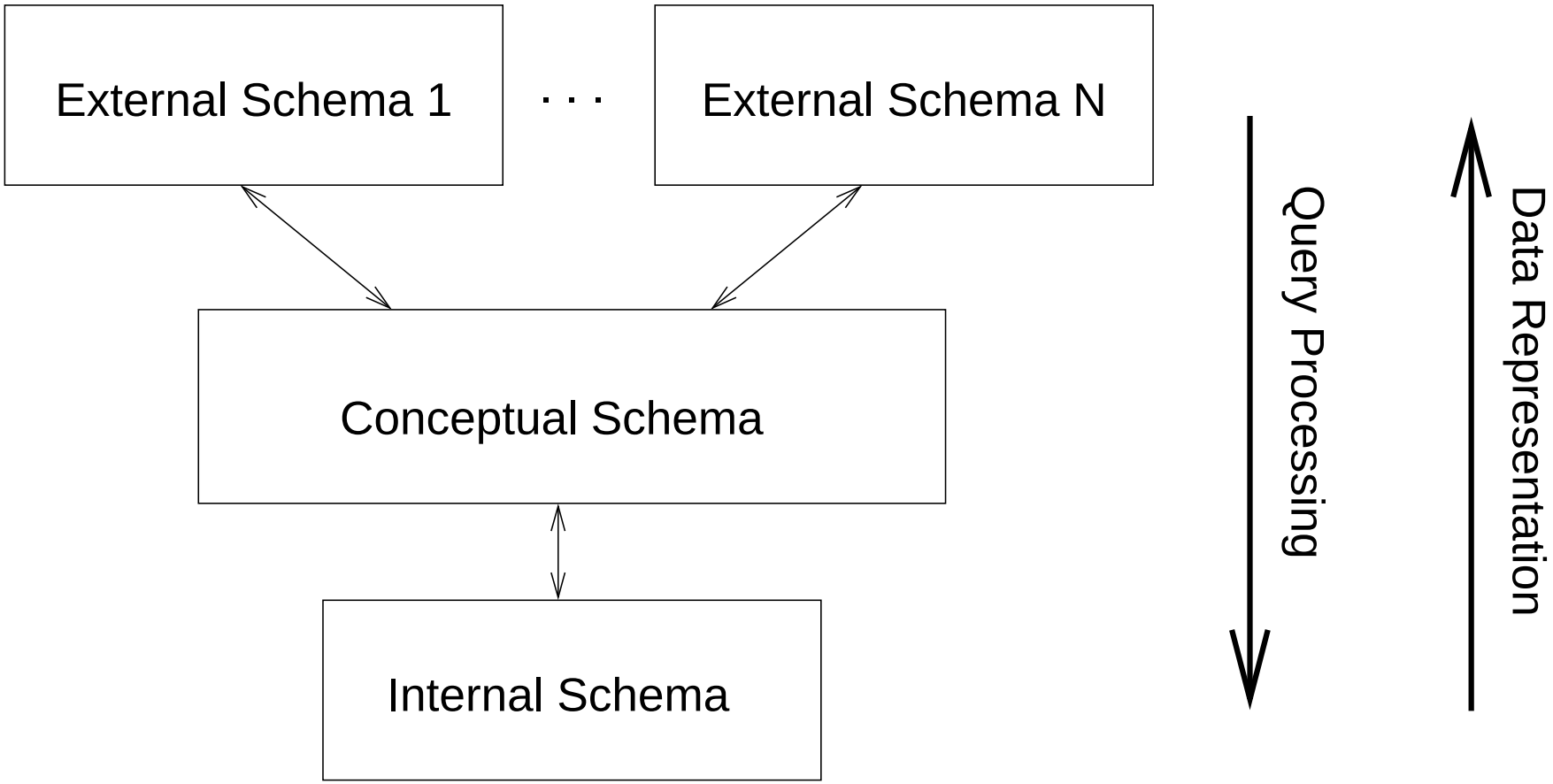
- Differentiate DBMS from other systems managing data persistently, e.g. file systems

1. **Integration:** homogeneous, non-redundant management of data
2. **Operations:** means for accessing, creating, modifying, and deleting data
3. **Catalog:** the data description must be accessible as part of the database itself
4. **User views:** different users/applications must be able to have a different perception of the data

Codd's 9 Rules for DBMS /2

5. **Integrity control:** the systems must provide means to grant the consistency of data
6. **Data security:** the system must grant only authorized accesses
7. **Transactions:** multiple operations on data can be grouped into a logical unit
8. **Synchronization:** parallel accesses to the database are managed by the system
9. **Data backups:** the system provides functionality to grant long-term accessibility even in case of failures

3 Level Schema Architecture



3 Level Schema Architecture /2

- Important concept of DBMS
- Provides
  - transparency, i.e. non-visibility, of storage implementation details
  - ease of use
  - decreased application maintenance efforts
  - conceptual foundation for standards
  - portability
- Describes abstraction steps:
  - Logical data independence
  - Physical data independence

Data Independence

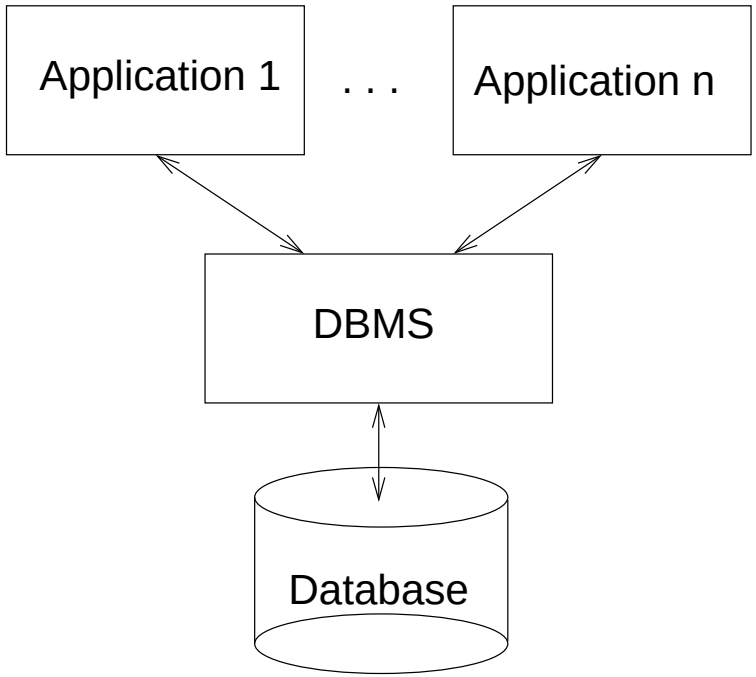
Logical data independence: Changes to the logical schema level must not require a change to an application (external schema) based on the structure.

Physical data independence: Changes to the physical schema level (how data is stored) must not require a change to the logical schema.

Architecture of a DBS

Schema architecture roughly conforms to general architecture of a database systems

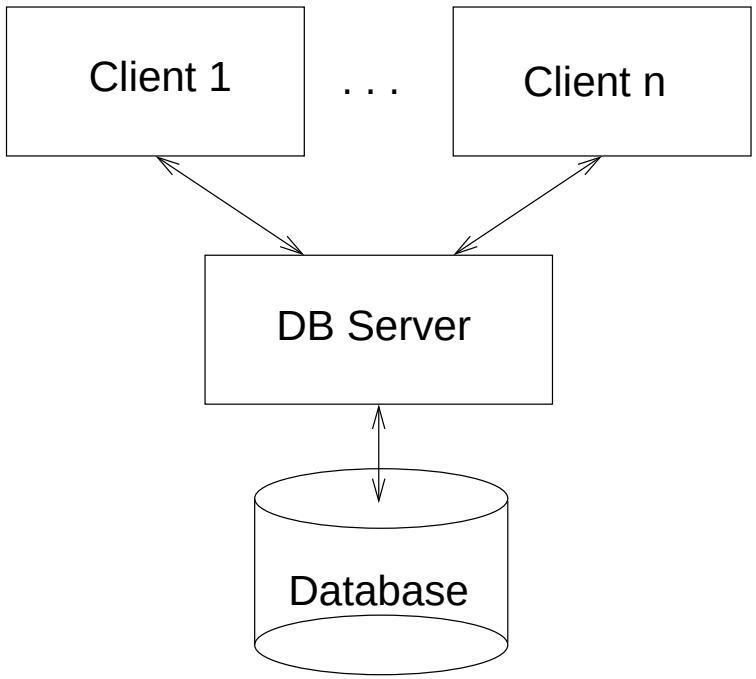
- **Applications** access database using specific **views (external schema)**
- The **DBMS** provides access for all applications using the **logical schema**
- The **database** is stored on secondary storage according to an **internal schema**



Client Server Architecture

Schema architecture does not directly relate to client server architecture (communication/network architecture)

- Clients may run several applications
- Applications may run on several clients
- DB servers may be distributed
- ...

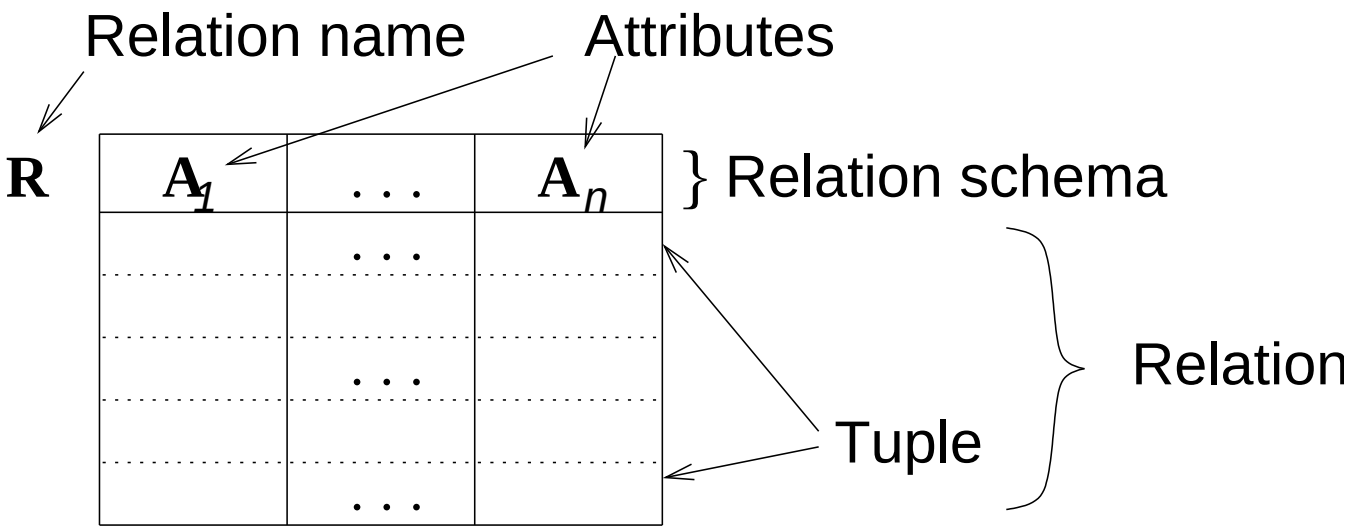


The Relational Model

- Developed by Edgar F. Codd (1923-2003) in 1970
- Derived from mathematical model of n-ary relations
- Colloquial: data is organized as tables (relations) of records (n-tuples) with columns (attributes)
- Currently most commonly used database model
- Relational Database Management Systems (RDBMS)
- First prototype: IBM System R in 1974
- Implemented as core of all major DBMS since late ’70s: IBM DB2, Oracle, MS SQL Server, Informix, Sybase, MySQL, PostgreSQL, etc.
- Database model of the DBMS language standard SQL

Basic Constructs

A **relational database** is a database that is structured according to the relational database model. It consists of a set of relations.



Basic Constructs /2

A **relation** is a set of tuples conforming to one relation schema and is identified by a relation name (*relvar*).

A **relation schema** is a set of attributes, each identified within the relation by an attribute name and conforming to a domain/type of valid values.

A **tuple** is a set of attribute values, conforming to a relation schema.

Integrity Constraints

- Static integrity constraints describe valid tuples of a relation
  - Primary key constraint
  - Foreign key constraint (referential integrity)
  - Value range constraints
  - ...
- In SQL additionally: uniqueness and not-NULL
- Transitional integrity constraints describe valid changes to a database

The Relational Algebra

A **relational algebra** is a set of operations that are closed over relations.

- Each operation has one or more relations as input
- The output of each operation is a relation

Relational Operations

Primitive operations:

- Selection σ
- Projection π
- Cartesian product (cross product) $\times$
- Set union $\cup$
- Set difference $-$
- Rename β

Non-primitive operations

- Natural Join $\bowtie$
- θ-Join and Equi-Join $\bowtie_{\varphi}$
- Semi-Join $\ltimes$
- Outer-Joins $= \times$
- Set intersection $\cap$
- $\ldots$

Notation for Relations and Tuples

- If R denotes a relation schema (set of attributes), than the function $r(R)$ denotes a relation conforming to that schema (set of tuples)
- R and $r(R)$ are often erroneously used synonymously to denote a relation, assuming that for a given relation schema only one relation exists
- $t(R)$ denotes a tuple conforming to a relation schema
- $t(R.a)$ denotes an attribute value of a tuple for an attribute $a \in R$

The Selection Operation σ

Select tuples based on predicate or complex condition

PROJECT			
PNAME	**PNUMBER**	**PLOCATION**	**DNUM**
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

$$\sigma_{PLOCATION='Stafford'}(r(PROJECT))$$

PNAME	**PNUMBER**	**PLOCATION**	**DNUM**
Computerization	10	Stafford	4
Newbenefits	30	Stafford	4

The Projection Operation π

Project to set of attributes - remove duplicates if necessary

PROJECT			
PNAME	**PNUMBER**	**PLOCATION**	**DNUM**
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

$$\pi_{PLOCATION,DNUM}(r(PROJECT))$$

PLOCATION	**DNUM**
Bellaire	5
Sugarland	5
Houston	5
Stafford	4
Houston	1

Cartesian or cross product $\times$

Create all possible combinations of tuples from the two input relations

R	
A	**B**
1	2
3	4

S		
C	**D**	**E**
5	6	7
8	9	10
11	12	13

$$r(R) \times r(S)$$

A	**B**	**C**	**D**	**E**
1	2	5	6	7
1	2	8	9	10
1	2	11	12	13
3	4	5	6	7
3	4	8	9	10
3	4	11	12	13

Set: Union, Intersection, Difference

- All require compatible schemas: attribute names and domains
- Union: duplicate entries are removed
- Intersection and Difference: $\emptyset$ as possible result

The Natural Join Operation $\bowtie$

- Combine tuples from two relations $r(R)$ and $r(S)$ where for
  - all attributes $a = R \cap S$ (defined in both relations)
  - is $t(R.a) = t(S.a)$.
- Basic operation for following key relationships
- If there are no common attributes result is Cartesian product
  $R \cap S = \emptyset \Longrightarrow r(R) \bowtie r(S) = r(R) \times r(S)$
- Can be expressed as combination of π, σ and $\times$
  $r(R) \bowtie r(S) = \pi_{R \cup S}(\sigma_{\bigwedge_{a \in R \cap S} t(R.a)=t(S.a)}(r(R) \times r(S)))$

The Natural Join Operation $\bowtie$ /2

R	
A	**B**
1	2
3	4
5	6

S		
B	**C**	**D**
4	5	6
6	7	8
8	9	10

$$r(R) \bowtie r(S)$$

A	**B**	**C**	**D**
3	4	5	6
5	6	7	8

The Rename Operation β

Can be used to create

- compatible schemas for set operations, or
- overlapping schemas for join operations.

PERSON	
ID	**NAME**
1273	Dylan
3456	Reed

CAR	
OWNERID	**BRAND**
1273	Cadillac
1273	VW Beetle
3456	Stutz Bearcat

$$r(PERSON) \bowtie (\beta_{OWNERID \rightarrow ID}(r(CAR)))$$

ID	NAME	BRAND
1273	Dylan	Cadillac
1273	Dylan	VW Beetle
3456	Reed	Stutz Bearcat

The Semi-Join Operation $\ltimes$

- Results all tuples from one relation having a (natural) join partner in the other relation
  $r(R) \ltimes r(S) = \pi_R(r(R) \bowtie r(S))$

PERSON	
PID	**NAME**
1273	Dylan
2244	Cohen
3456	Reed

CAR	
PID	**BRAND**
1273	Cadillac
1273	VW Beetle
3456	Stutz Bearcat

$r(PERSON) \ltimes r(CAR)$

PID	NAME
1273	Dylan
3456	Reed

Other Join Operations

- **Conditional join:** join condition φ is explicitly specified
  $r(R) \bowtie_{\varphi} r(S) = \sigma_{\varphi}(r(R) \times r(S))$
- **θ-Join:** special conditional join, where φ is a single predicate of the form $a\theta b$ with $a \in R$, $b \in S$, and $\theta \in \{=, \neq, >, <, \leq, \geq, \dots\}$
- **Equi-Join:** special θ-Join where θ is $=$.
- **(Left or Right) Outer Join:** union of natural join result and tuples from the left or right input relation which could not be joined (requires NULL-values to grant compatible schemas).

Relational Completeness

- **Important question: Is it possible to compute all derivable relations using these operations? $\rightarrow$ No!**
- Computational completeness, such as provided by e.g. imperative programming languages (C++, Java, . . .) not given
- Problems
  Negation: return all tuples that are not in a relation.
  Recursion: return the transitive closure of a tuple following foreign key relationships.

An algebra or query language providing the same computational power as a relational algebra based on the primitive operations is **relationally complete**.

Further concepts

- Data manipulations (insert, delete, update) are described as status changes of a database relation
- Functional dependencies are used to define quality criteria $\rightarrow$ normal forms avoid redundancy

Relational Database Management Systems

A **Relational Database Management System (RDBMS)** is a database management system implementing the relational database model.

- Today, most relational DBMS implement the SQL database model
- There are some significant differences between the relational model and SQL discussed later on

Codd's 12 Rules for RDBMS

- Similar intention as 9 rules for DBMS
- 12 rules list requirements on DBMS implementing the relational database model
- Intended to differentiate RDBMS from other DBMS
- Very rigid rules, that even some well established commercial RDBMS fail to fully comply with
- Already include aspects of SQL (e.g. NULL-values)
- To add some more confusion: the 12 rules are actually 13

Codd's 12 Rules for RDBMS /1

Rule 0 - System is relational, a database, and a management system: i.e. to qualify as an RDBMS, it must use its relational facilities (exclusively) to manage the database.

Rule 1 - The information rule: All information in the database to be represented only by values in column positions within rows of tables.

Rule 2 - The guaranteed access rule: All data must be accessible with no ambiguity, i.e. every individual scalar value in the database must be logically addressable the name of the containing table, the name of the column and the primary key value of the containing row.

Codd's 12 Rules for RDBMS /2

Rule 3 - Systematic treatment of null values: The DBMS must support a representation of "missing information and inapplicable information" that is systematic, distinct from all regular values, and independent of data type.

Rule 4 - Active online catalog based on the relational model: The system supports a relational catalog that is accessible by means of the query language.

Rule 5 - The comprehensive data sub-language rule: The system must support a relational language that has a linear syntax, can be used interactively and within application programs, supports data definition and manipulation, security, integrity, and transactions.

Codd's 12 Rules for RDBMS /3

Rule 6 - The view updating rule: All views that are theoretically updatable must be updatable by the system.

Rule 7 - High-level insert, update, and delete: Insert, update, and delete operators can be applied to sets of tuples.

Rule 8 - Physical data independence: Changes to the physical schema level (how data is stored) must not require a change to the logical schema.

Rule 9 - Logical data independence: Changes to the logical schema level must not require a change to an application (external schema level) based on the structure.

Codd's 12 Rules for RDBMS /4

Rule 10 - Integrity independence: Integrity constraints must be specified separately from application programs and stored in the catalog.

Rule 11 - Distribution independence: The distribution of portions of the database to various locations should be invisible to users of the database.

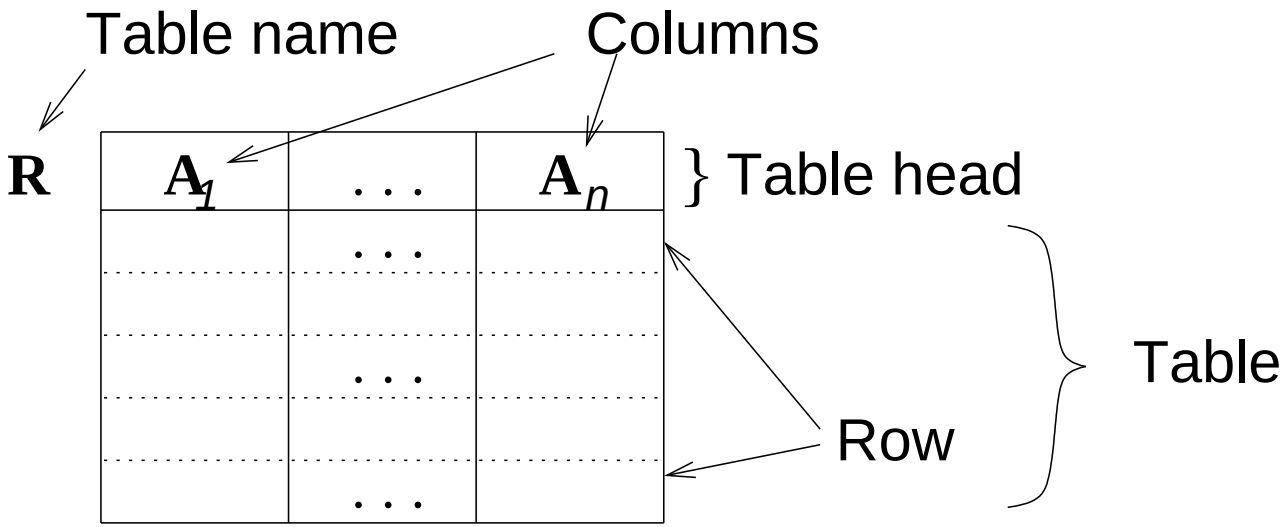
Rule 12 - The non-subversion rule: If the system provides a low-level (record-at-a-time) interface, then that interface cannot be used to subvert the system, for example, bypassing a relational security or integrity constraint.

SQL History

- SEQUEL (1974, Chamberlin and Boyce, IBM Research Labs San Jose)
- SEQUEL2 (1976, IBM Research Labs San Jose)
- SQL (1982, IBM)
- ANSI-SQL (SQL-86; 1986)
- ISO-SQL (SQL-89; 1989; Level 1, Level 2, + IEF)
- (ANSI / ISO) SQL-92 (full relational standard)
- (ANSI / ISO) SQL:1999 (most object-relational extensions, recursive queries + more)
- (ANSI / ISO) SQL:2003 (nested tables, XML, OLAP + more)

SQL Data Model

- Said to implement relational database model
- Defines own terms



- Some significant differences exist

Differences between SQL and RM

Duplicate rows: The same row can appear more than once in an SQL table(bag or list), while relations are sets.

Anonymous columns: A column in an SQL table can be unnamed (e.g. aggregate functions).

Duplicate column names: Two or more columns of the same SQL table can have the same name.

Column order significance: The order of columns in an SQL table is defined and significant.

Row order significant: SQL provides operations to provide ordered results (lists).

NULL-values: can appear instead of a value. Implies the use of three-valued logic.

Structured Query Language

- **Structured Query Language (SQL):** declarative language to describe requested query results
- Realizes relational operations (with the mentioned discrepancies)
- Basic form: `SELECT-FROM-WHERE`-block (SFW)

```
SELECT FNAME, LNAME, MGRSTARTDATE
FROM EMPLOYEE, DEPARTMENT
WHERE SSN=MGRSSN
```

SQL: Selection σ

$\sigma_{DNO=5 \wedge SALARY>30000}(r(EMPLOYEE))$

```
SELECT *
FROM EMPLOYEE
WHERE DNO=5 AND SALARY>30000
```

SQL: Projection π

$\pi_{LNAME,FNAME}(r(EMPLOYEE))$

```
SELECT LNAME,FNAME
FROM EMPLOYEE
```

- Difference to RM: does not remove duplicates
- Requires additional `DISTINCT`

```
SELECT DISTINCT LNAME,FNAME
FROM EMPLOYEE
```

SQL: Cartesian Product $\times$

$r(EMPLOYEE) \times r(PROJECT)$

```
SELECT *
FROM EMPLOYEE, PROJECT
```

SQL: Natural Join $\bowtie$

$r(DEPARTMENT) \bowtie r(DEPARTMENT\_LOCATIONS)$

```
SELECT *
FROM DEPARTMENT
     NATURAL JOIN DEPARTMEN_LOCATIONS
```

SQL: Equi-Join

$r(EMPLOYEE) \bowtie_{SSN=MGRSSN} r(DEPARTMENT)$

```
SELECT *
FROM EMPLOYEE, DEPARTMENT
WHERE SSN=MGRSSN
```

SQL: Union

$r(R) \cup r(S)$

```
SELECT * FROM R
UNION
SELECT * FROM S
```

- Other set operations: `INTERSECT`, `MINUS`
- Does remove duplicates (in compliance with RM)
- If duplicates required:

```
SELECT * FROM R
UNION ALL
SELECT * FROM S
```

SQL: Other Features

- SQL provides several features not in the relational algebra
  - Grouping And Aggregation Functions, e.g. `SUM`, `AVG`, `COUNT`, ...
  - Sorting

```
SELECT PLOCATION, AVG(HOURS)
FROM EMPLOYEE, WORKS_ON, PROJECT
WHERE SSN=ESSN AND PNO=PNUMBER
GROUP BY PLOCATION
HAVING COUNT(*) > 1
ORDER BY PLOCATION
```

SQL DDL

- **Data Definition Language** to create, modify, and delete schema objects

```
CREATE DROP ALTER TABLE mytable ( id INT, ...)
DROP TABLE ...
ALTER TABLE ...
CREATE VIEW myview AS SELECT ...
DROP VIEW ...
CREATE INDEX ...
DROP INDEX ...
...
```

Simple Integrity Constraints

```
CREATE TABLE employee(
    ssn INTEGER,
    lname VARCHAR2(20) NOT NULL,
    dno INTEGER,
    ...
    FOREIGN KEY (dno)
        REFERENCES department(dnumber),
    PRIMARY KEY (ssn)
)
```

- Additionally: triggers, explicit value domains, ...

SQL DML

- **Data Manipulation Language** to create, modify, and delete tuples

```
INSERT INTO (<COLUMNS>) mytable VALUES (...)

INSERT INTO (<COLUMNS>) mytable SELECT ...

UPDATE mytable
SET ...
WHERE ...

DELETE FROM mytable
WHERE ...
```

Other Parts of SQL

- Data Control Language (DCL):
  `GRANT, REVOKE`
- Transaction management:
  `START TRANSACTION, COMMIT, ROLLBACK`
- Stored procedures and imperative programming concepts
- Cursor definition and management

Teil III

Models for Conceptual Design

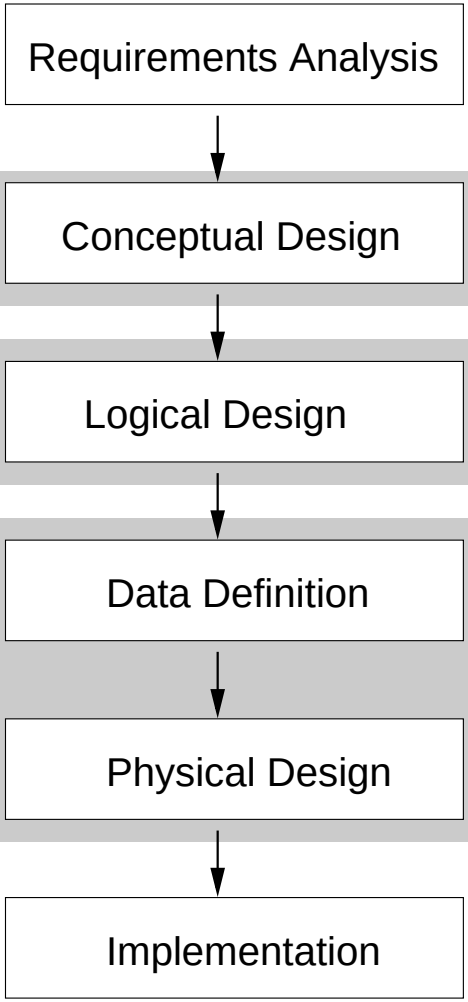
Overview

- Conceptual Database Design
- The Entity-Relationship Model
- Basic Extensions to the ER Model
- Object-oriented Extensions to the ER Model
- Mapping ER to the Relational Model
- UML as a Conceptual Design Model

Models for Conceptual Design

- First design stages of a database system
- (Mostly) independent of logical design or implementation
- Most common and popular design models
  - The Entity-Relationship Model (ER Model) as the still state-of-the-art of conceptual database design
  - Numerous Extensions to the ER Model covering semantic and object-oriented aspects
  - The (object-oriented) Unified Modeling Language (UML) developed for software design but less recently often applied for database design

Database Design Process



Conceptual Models
e.g. ER or UML

Database Models
e.g. RM, ORM, OO

Concrete implementation of a database model, e.g. the object−relational model of the Oracle 10g DBMS

The Entity-Relationship Model

The **Entity-Relationship Model** is a data model or diagram for high-level descriptions of conceptual schemata. It provides a graphical notation for representing such schemata in the form of entity-relationship diagrams.

- Developed by P. P. Chen in 1976
- Based on the Data Structure Model for conceptual design of Network Model-based database schemata

Basic ER Modeling Constructs

Entities – or more precise Entity Types – represent sets of objects in a given universe of discourse sharing the same properties and relationships as other instances of this particular Entity Type. Their graphical notation are rectangular boxes.

Relationships – or Relationship Types – represent how instances of Entity Types can be related in the universe of discourse. Their graphical notation are diamond-shaped boxes.

Attributes represent properties of Entity or Relationship Types. Their graphical notation are rounded boxes or ellipses.

Further ER Modeling Constructs

Key attributes: an attribute or a set of attributes may be defined as identifying for an entity type, i.e. the values are unique within the set of instances

Relationship cardinalities: used to describe how many times an entity may participate in a relationship

Relationship roles: assign special names to entities for participation in a relation

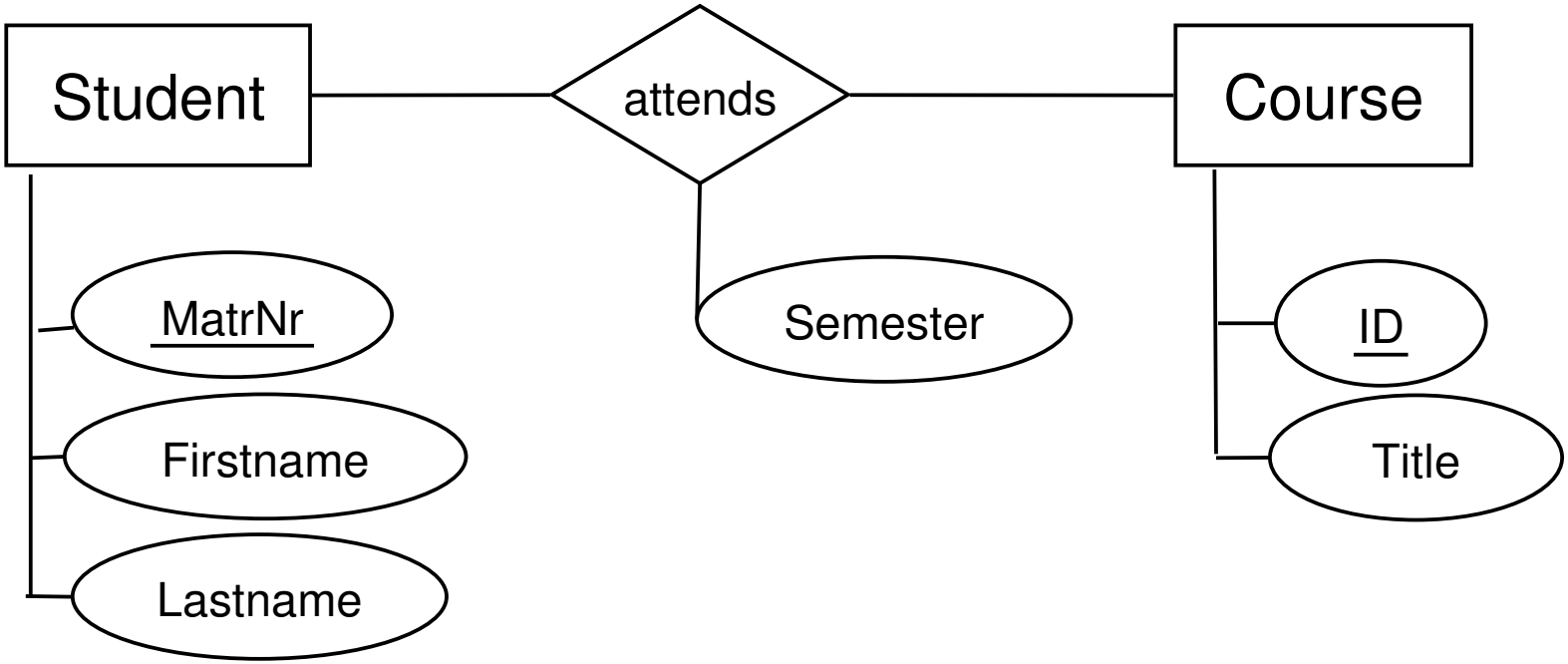
Functional relationships: each entity of one type is assigned an entity of another type (according to mathematical concept of a function)

Further ER Modeling Constructs /2

N-ary relationships: a relation may be defined for only one participating entity (unary, self-referential), two (binary, standard), or more (ternary . . . n-ary)

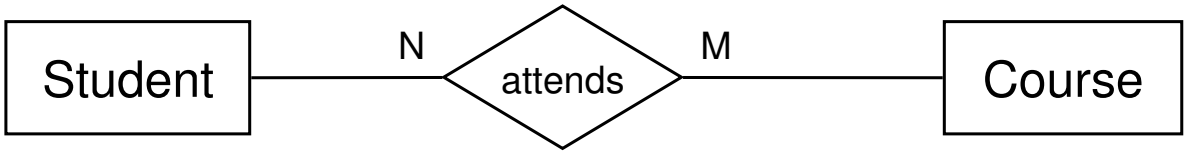
Weak entities: entities with existential dependency to entity of another type

ER Model: Basic Constructs

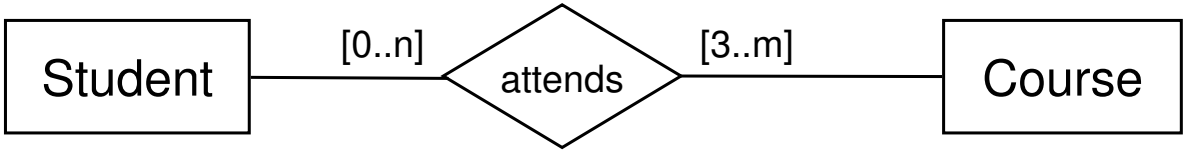


ER Model: Cardinalities

- N:M and 1:N relationships, if not specified N:M
- Alternative notation using minimal and maximal number of participations

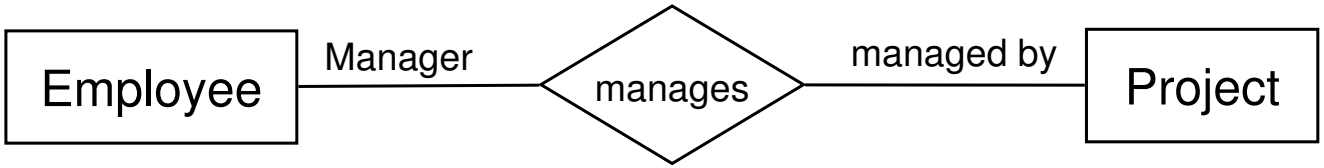


Alternative notation:



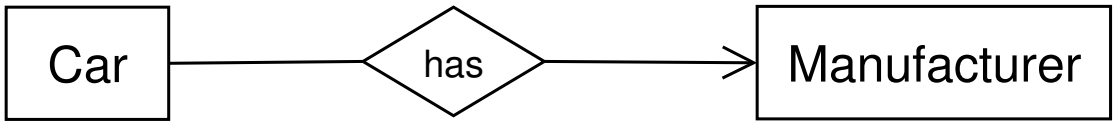
ER Model: Roles

- Can be used to describe participation semantics if not obvious from entity or relationship type

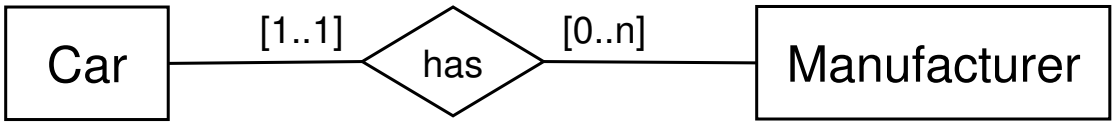


ER Model: Functional Relationships

- Assigns instances of one entity type exactly one instance of the related entity type
- Mathematical concept of a function
- Not necessarily *injective* (target entities referenced only once) or *surjective* (each target entity is referenced)

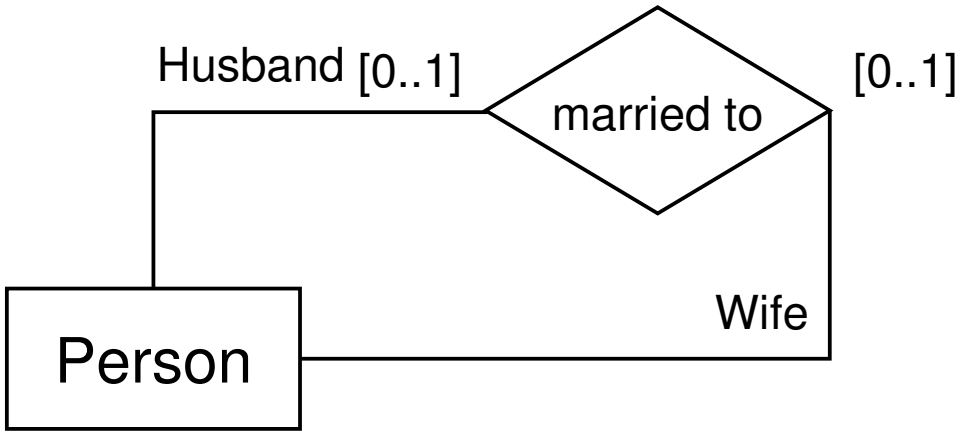


Alternative for:



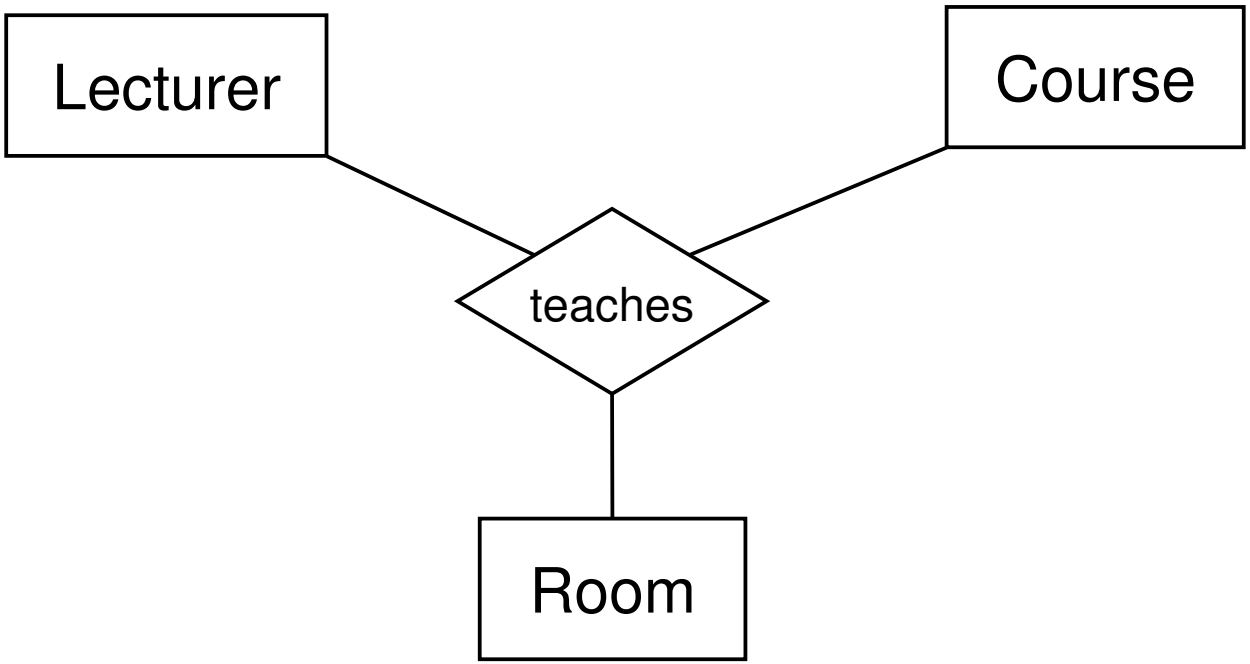
ER Model: Unary Relationship

- Self-referential on type-level
- Different instances of the same type are related



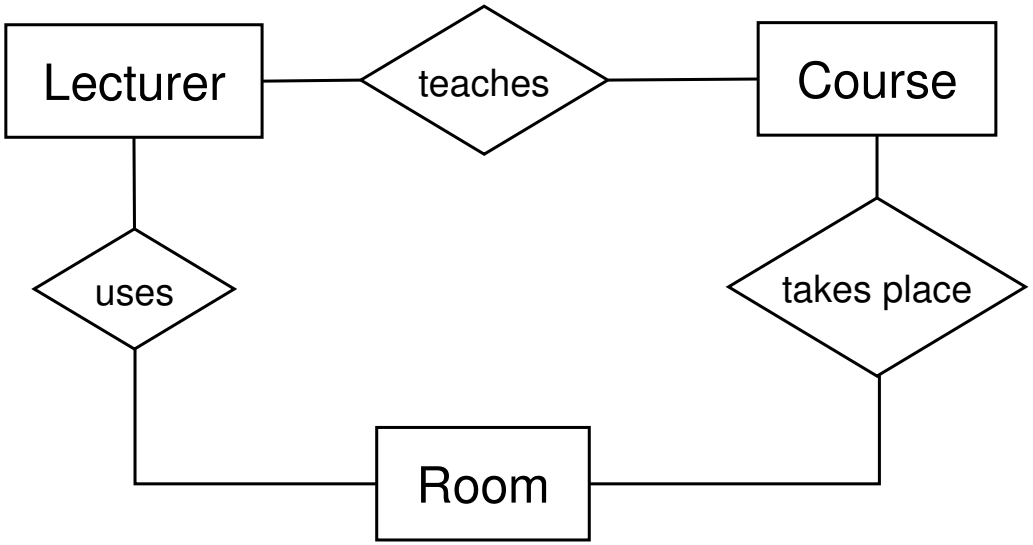
ER Model: N-ary Relationship

- Relationship type might involve more than 1 or 2 entity types
- Example for ternary relationship type



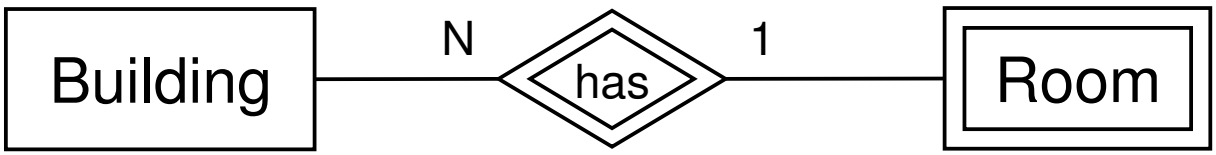
ER Model: N-ary Relationship /2

- N-ary relationship type can not be replaced by n binary relationship types
- Semantics of following example different from previous: allows lecturers to use rooms without a course taking place



ER Model: Weak Entity Types

- Existential dependency to other entity Types
- Identity (key attribute) includes key of master entity type
- Relationship between weak entity type and its master type is called **identifying relationship**

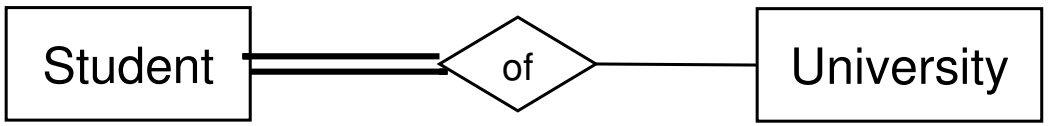


Basic Extensions to the ER Model

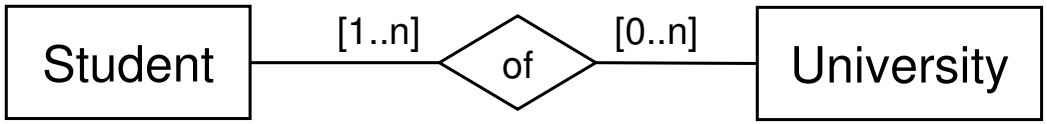
- Total or partial participation
- Complex attributes
- Derived attributes
- Multi-valued attributes

ER Extensions: Total Participation

- Each entity of one type must relate to at least one of the other entity type at least once
- Graphical notation: thick or double line
- Difference to functional relationship: entity may participate in relationship several time

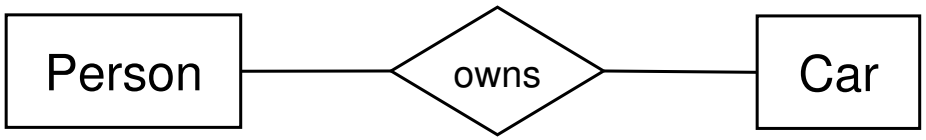


Alternative for:

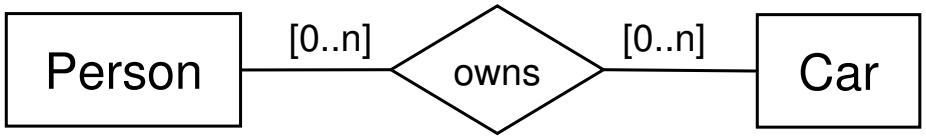


ER Extensions: Partial Participation

- Standard case: each entity of one type may or may not relate to one or more entities of the other entity type
- Graphical notation: thin line
- Optional N:M-relationship

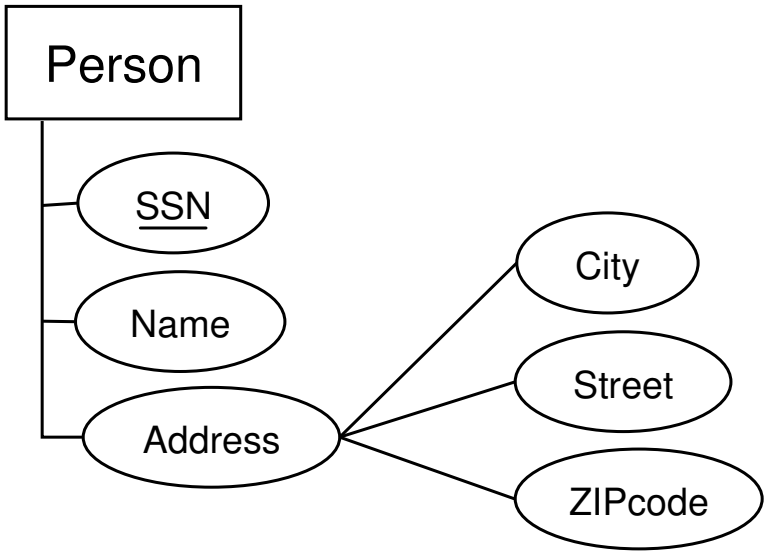


Alternative for:



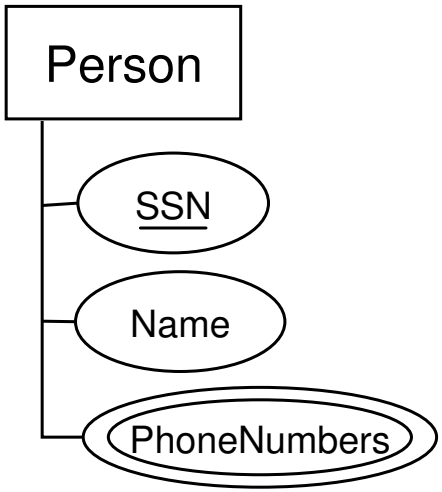
ER Extensions: Complex Attributes

- Composite attributes consisting of atomic attributes of different types



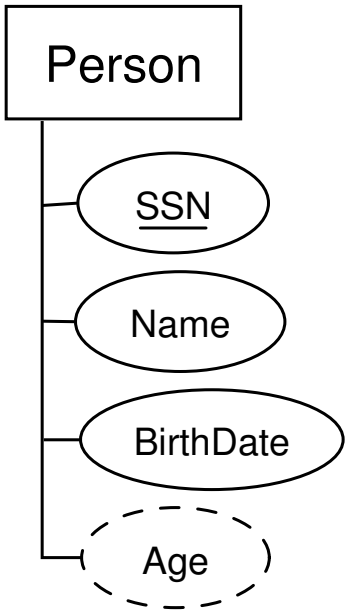
ER Extensions: Multi-valued Attributes

- Composite attributes consisting of variable set of values of the same type



ER Extensions: Complex Attributes

- Virtual attribute with value that can be computed from other attributes or via relationships



Basic ER Extensions Summary

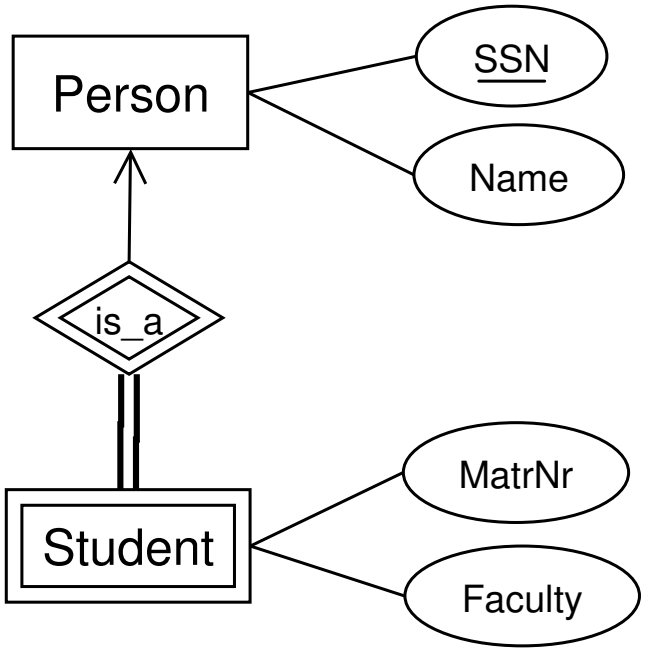
- Only most common extensions were introduced here
- Many others exist in books and DB design tools
- Many different notations exist for the same concepts

Object-oriented Extensions to the ER Model

- Specialization as the derivation of sub-types
  - Semantic concept of the **is_a**-relationship
  - Various notations exist
  - Total or partial
  - Disjunctive or overlapping
  - Multiple inheritance
- Generalization: inverse specialization, creation of a super-type
- Usually no special constructs for typical OO aspects
  - Description of behavior and dynamics (methods, states, etc.)
  - Aggregation, grouping/association

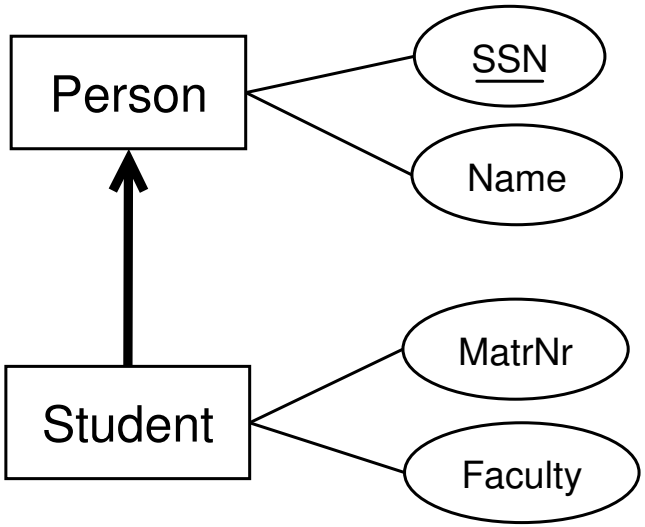
OO ER Extensions: Notations

- Can be partially expressed using previously introduced constructs
- Semantic of inheritance is lost, if **is_a**-relationship is not defined as “special” relation



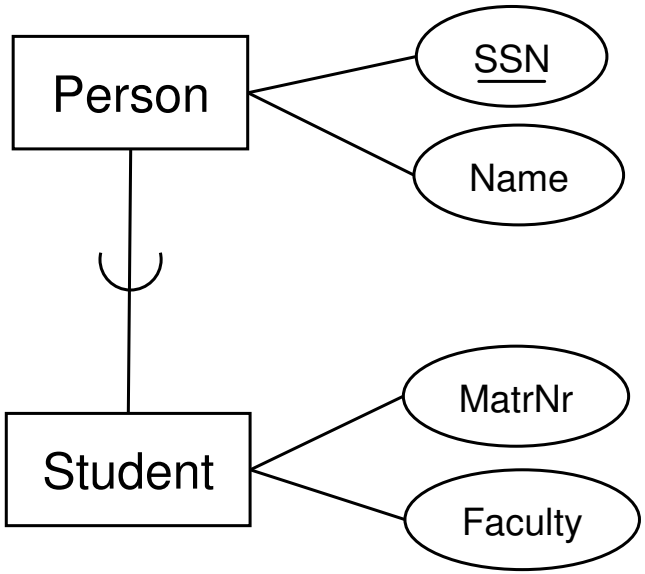
OO ER Extensions: Notations /2

- Notation used in Heuer/Saake



OO ER Extensions: Notations /3

- Notation used in Elmasri/Navathe
- Half circle hints at subset relation *students* $\subset$ *persons*



OO ER Extensions: Total/Partial Specialization

Total specialization: every instance of a type must be an instance of at least one sub-type (super-type is abstract)

Partial specialization: instance of a type can be instance of sub-types or instance of only type itself (super-type can be instantiated and is not abstract)

OO ER Extensions: overlapping/disjunctive Specialization

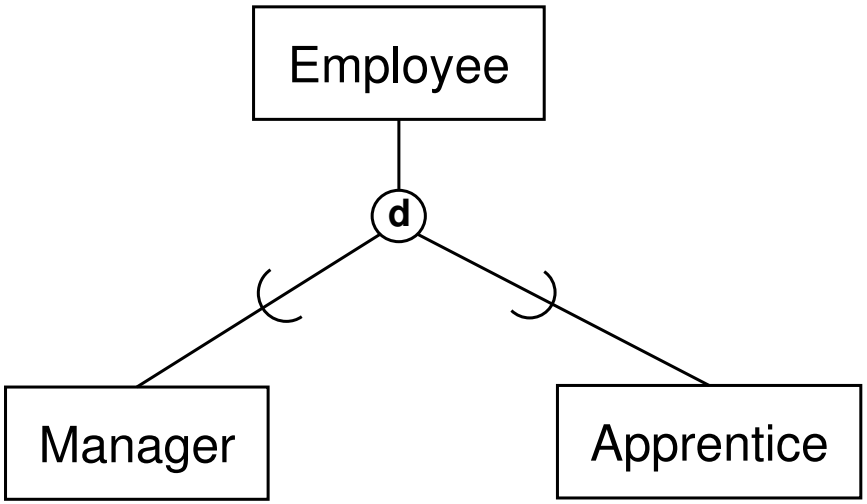
Disjunctive specialization: instance of super-type can be instance of at most one sub-type (typical in programming languages)

Overlapping specialization: instance of a super-type can be instance of one or more sub-types (similar to role concept, where objects can be assigned different “roles”)

- Difference to multiple inheritance: MI allows several super-types for one type

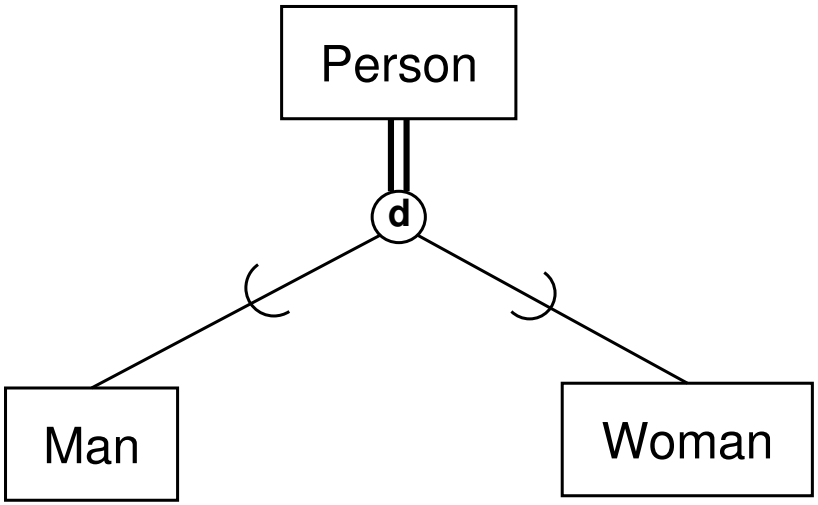
OO ER Extensions: Specialization /1

- Partial specialization: employees may be neither managers nor apprentices: $\mathit{managers} \cup \mathit{apprentices} \subset \mathit{employees}$
- Distinctiveness: $\mathit{managers} \cap \mathit{apprentices} = \emptyset$



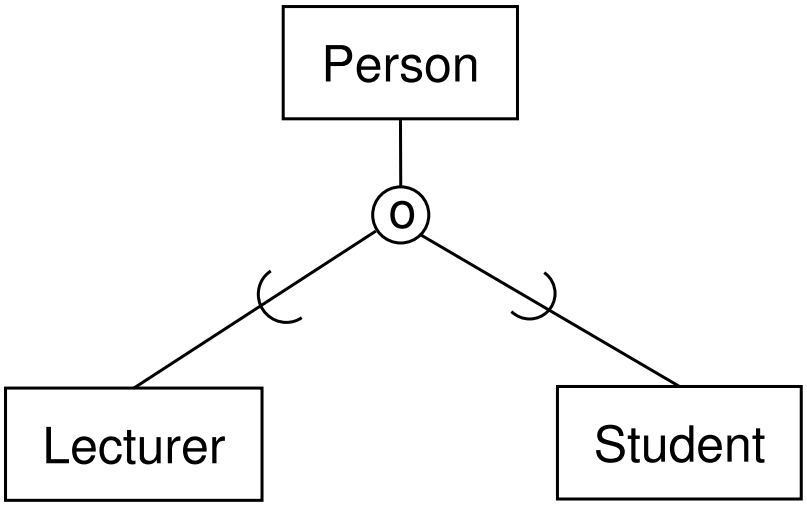
OO ER Extensions: Specialization /2

- Total specialization: a person is either a woman or a man: $men \cup women = persons$
- Distinctiveness: $men \cap women = \emptyset$

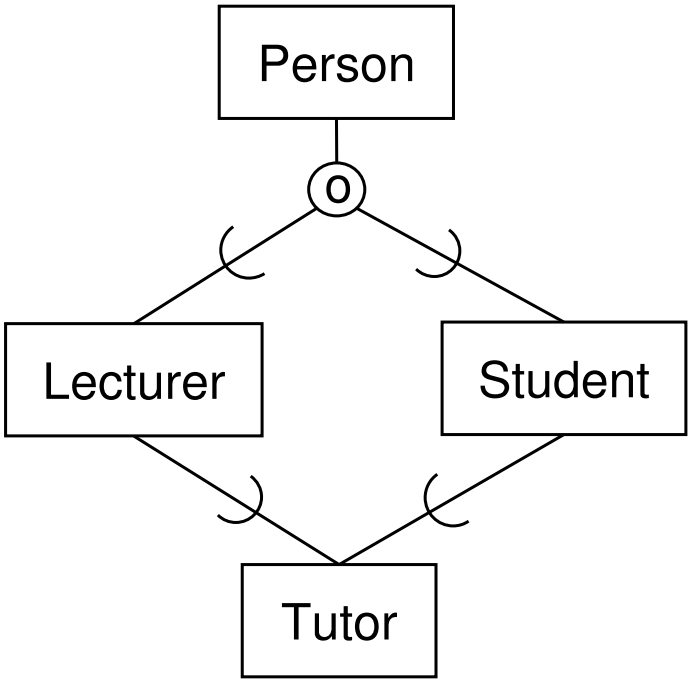


OO ER Extensions: Specialization /3

- Overlapping specialization: a person may be a lecturer, a student, or both (a tutor)
- Multiple inheritance may be used to describe overlapping type: $\mathit{tutors} = \mathit{lecturers} \cap \mathit{students}$



OO ER Extensions: Specialization /4



Mapping ER to the Relational Model

- Schemata expressed using semantically rich ER model must be mapped to flat relation schemata with atomic attributes
  - without losing information
  - without introducing ambiguities
  - without introducing redundancy
- Straightforward rules
  - Entities map to relations
  - Cardinalities indicate whether relationships become relations

Mapping Rules /1

ER Model	**Relational Model**
Entity	Relation
Attribute	Attribute
Primary key	Primary key
N:M relationship	Relation with keys of participating entity types plus own attributes

Mapping Rules /2

ER Model	Relational Model	Condition
1:N or 1:1 relationship	Foreign key plus relationship attributes included in participating N-relation	Relationship is total (not optional)
1:N or 1:1 relationship	Relation with keys of participating entity types plus own attributes	Relationship is partial (optional)
N-ary relationship	Relation with keys of participating entity types plus own attributes	

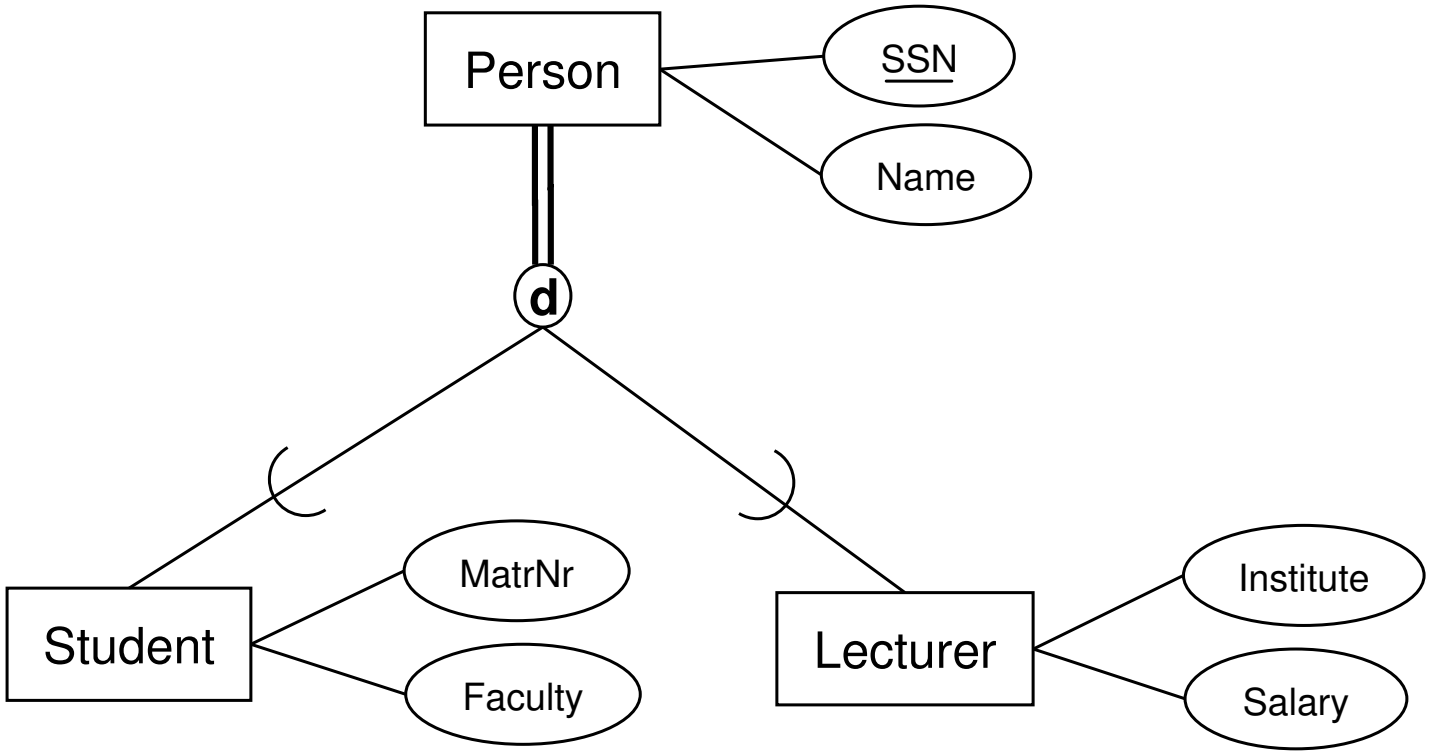
Mapping Rules /3

ER Model	**Relational Model**
Complex attributes	Set of flat attributes
Derived attributes	n.a.
Multi-valued attributes	Additional relation with primary key of master relation as foreign key

Mapping OO to RM

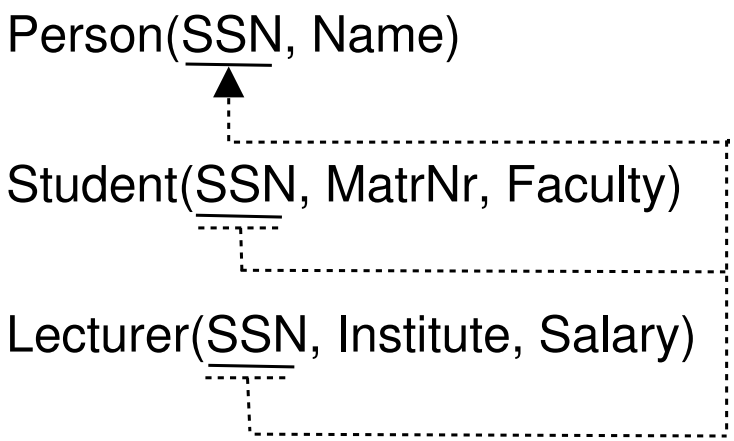
- No support for specialization in the relational database model
- Different ways to map OO schema have different disadvantages
  - Loss of information (polymorphism not expressible)
  - Inefficient access (requiring queries including union or join)
  - Redundancy
- Three most common mappings described here

OO to RM: Example



OO to RM: Solution 1

- **Map all types to relations with only direct attributes, inheritance relationship expressed by foreign key**
- Requires join operation to access sub-types



OO to RM: Solution 2

- **Map all (non-abstract) sub-types to relations with all direct and derived attributes**
- Requires projection and union operations to access super-type
- Not suitable for partial inheritance

Student(SSN, Name, MatrNr, Faculty)

Lecturer(SSN, Name, Institute, Salary)

OO to RM: Solution 3

- **Map all types to one generalized relation with all attributes of all super- and sub-types plus a discriminator attribute to indicate type membership**
- Introduces redundancy
- Requires selection to access sub-types

Person(SSN, Name, PersonType, MatrNr, Faculty, Institute, Salary)

UML as a Conceptual Design Model

- Object modeling and specification language by the Object Management Group (OMG)
- Developed in the early ’90s for OO software engineering
- Integrates different OO modeling approaches by Grady Booch, Ivar Jacobson, and James Rumbaugh
- Becomes increasingly popular for database design

Parts of UML

- Consists of several diagram types, but class diagram most relevant for database design

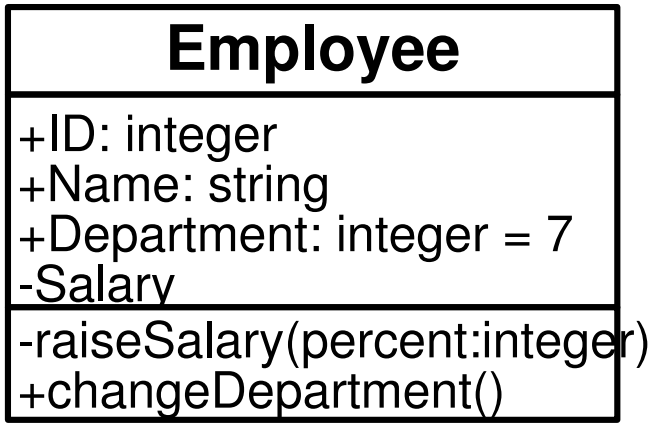
  - Class diagram
  - Component diagram
  - Object diagram
  - Composite structure diagram
  - Deployment diagram
  - Package diagram
  - Activity diagram
  - Use Case diagram
  - State Machine diagram
  - Sequence diagram
  - Collaboration
  - Interaction overview diagram
  - Timing diagram
  - Object Constraint Language

UML for Database Design

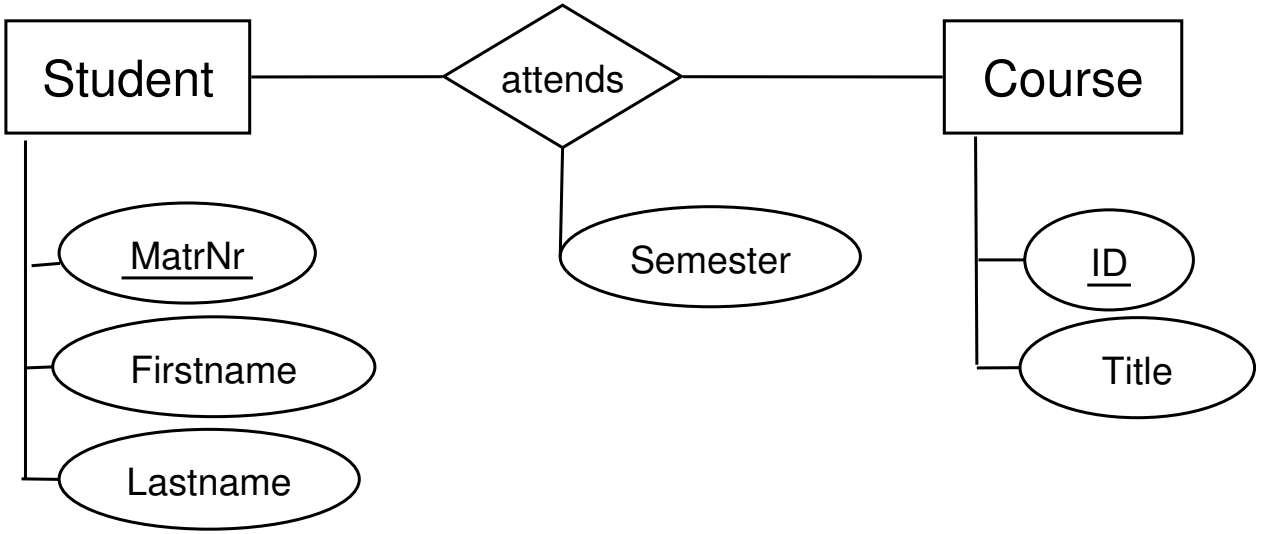
- In general, all concepts of ER and its extensions can be expressed using UML class diagram
  - Classes conform to entity types
  - Associations conform to relationship types
- Several additional concepts not required for purely relational database design
- Some concepts potentially useful for DB design, e.g.
  - **Aggregation** from class diagrams
  - **Object diagrams** to describe instance level
  - **Object Constraint Language (OCL)**
  - **Use case diagrams** to model interaction

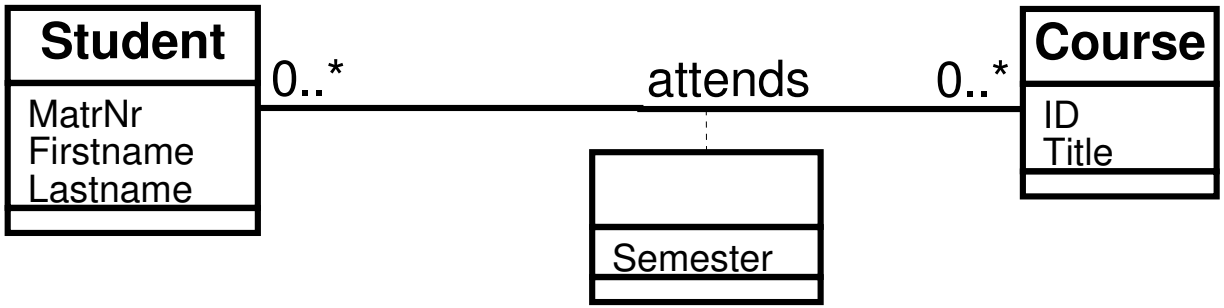
UML classes

- UML classes allow specifying many concepts typical in software engineering: interfaces, encapsulation, templates/generics, default values, methods, etc.

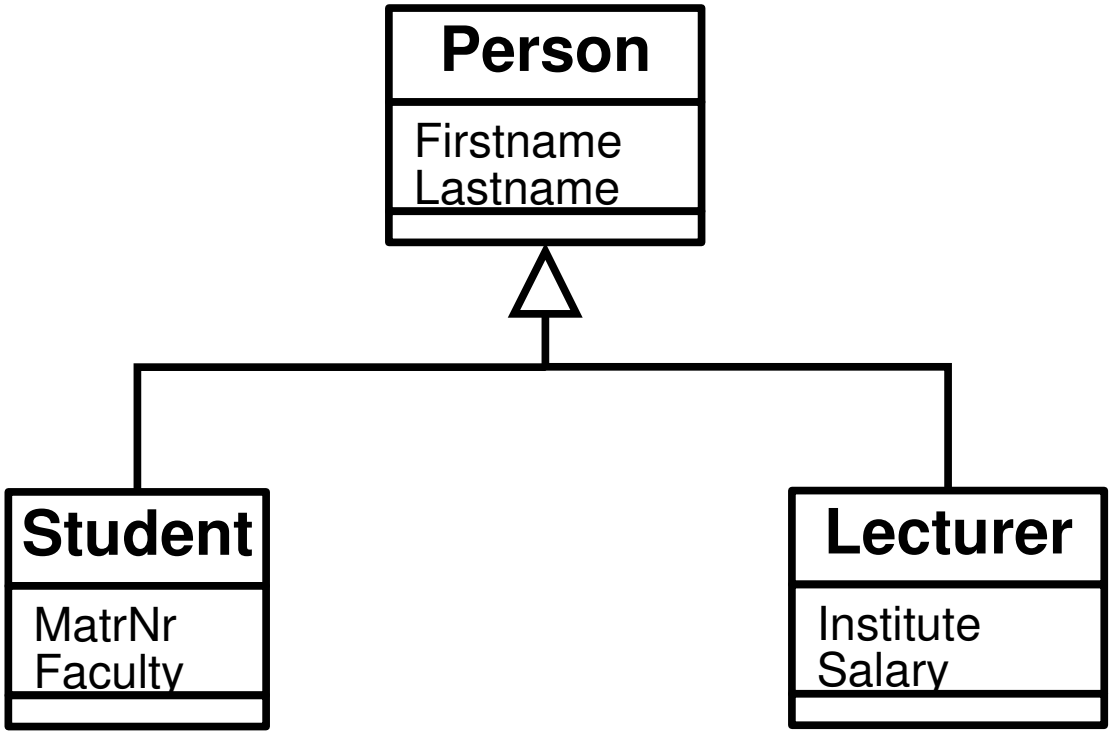


ER and UML example





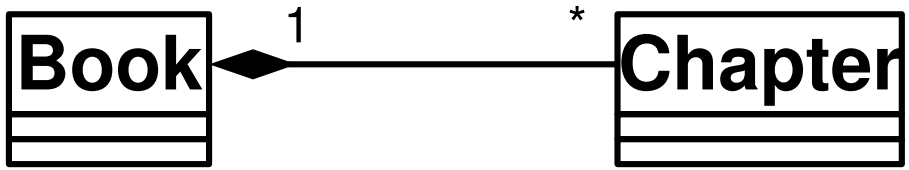
UML Specialization



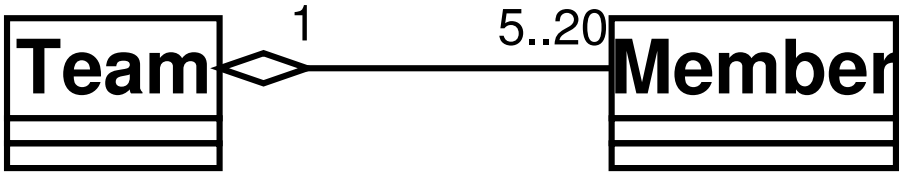
UML Aggregation

- Describes composition of complex objects from other objects

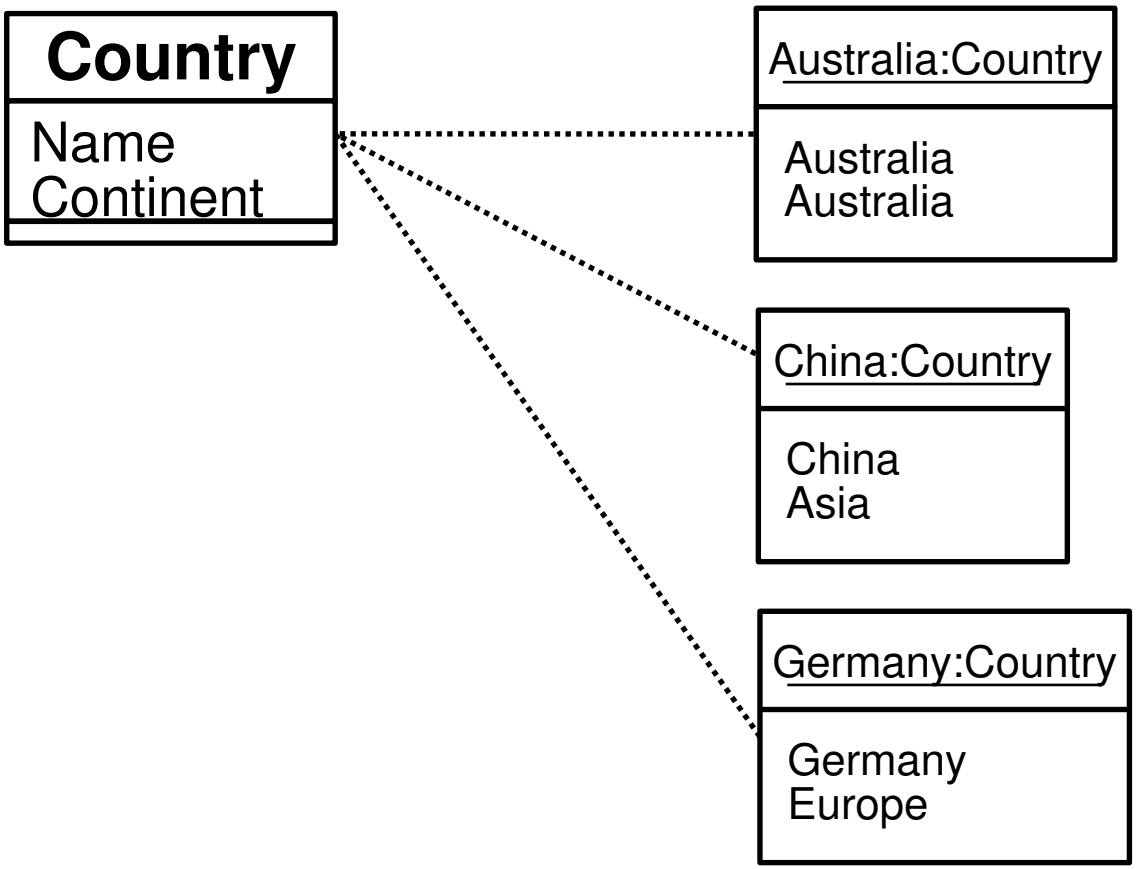
Existencial dependency:



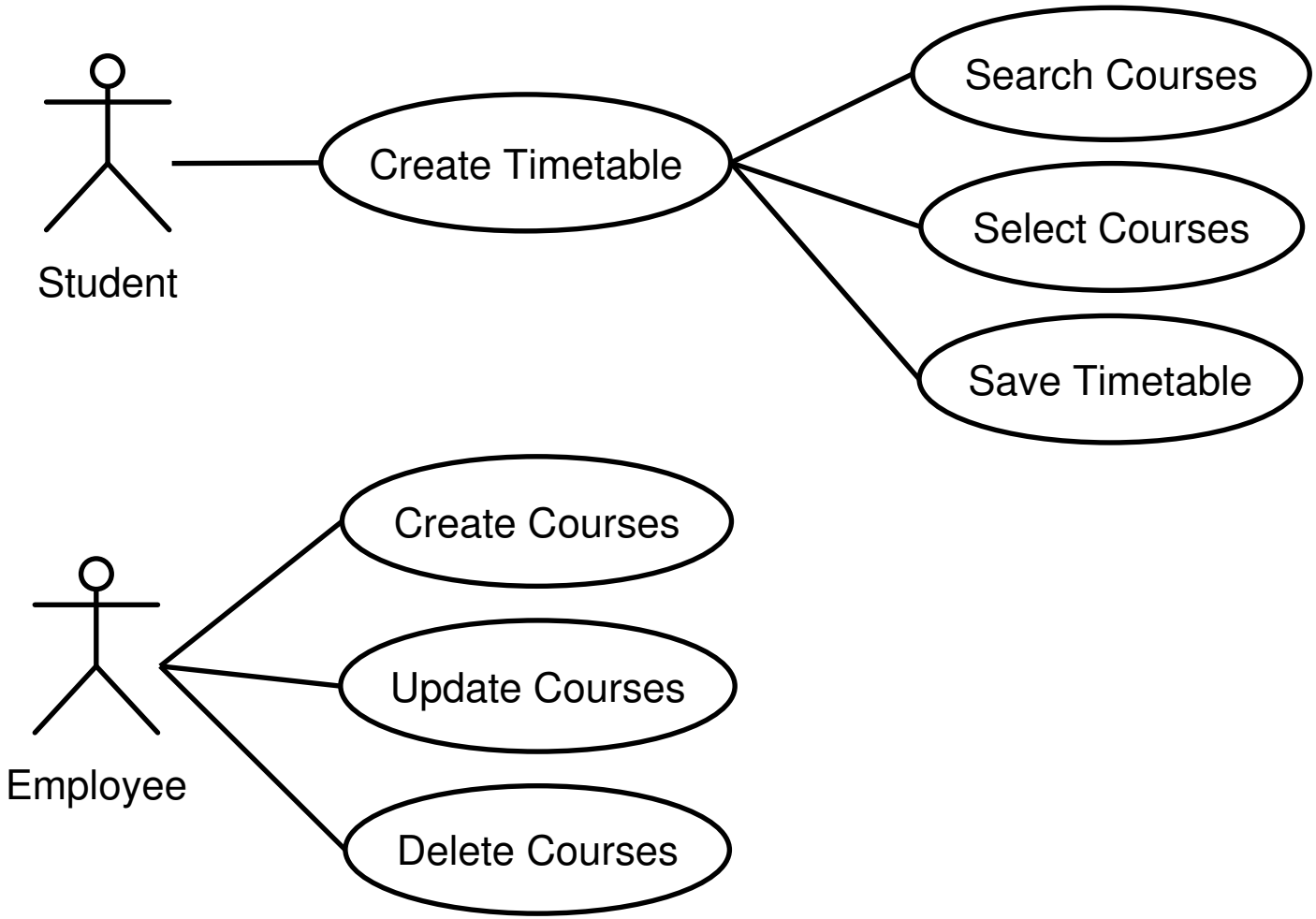
No existencial dependency:



UML Object Diagram



UML Use Case Diagram



Teil IV

NF^2 and eNF^2 Data Models

Overview

- The NF^2 Database Model
- The eNF^2 Database Model
- eNF^2 Concepts in SQL:1999 and SQL:2003
- eNF^2 in Oracle

The NF^2 Database Model

- First normal form basic requirement of the relational database model
  - Unambiguous addressability of values
  - Simple operations for access
- Requires complex objects to be stored in various relations
  - Loss of semantics not intuitive
  - Often inefficient (joins required)
- First (theoretical) extensions of the RM
  - Nested relations: NF^2
  - Type constructors and free combination: eNF^2
- Today part of SQL:1999, SQL:2003, and commercial systems

The NF^2 Database Model /2

NF^2-relations – or nested relations – provide non-atomic attributes by allowing an attribute to be of a relation type (schema) and a value of such an attribute to be a (nested) relation.

- NF^2 = NFNF = Non First Normal Form
- New Requirements
  - Operations as extension to relational algebra
  - Normal form to provide consistency

NF^2 Example Relation

Department	Employees			
	SSN	Name	Telephones	Salary
			Telephone	
Computer Science	4711	Todd	038203-12230 0381-498-3401 0381-498-3427	6000
	5588	Whitman	0391-345677 0391-5592-3800	6000
	7754	Miller		550
	8832	Kowalski		2800
Mathematics	6834	Wheat		750

NF^2 Algebra

- $\cup, -, \pi, \bowtie$ as in relational algebra
- σ condition extended to support
  - Relations as operands (instead of constants of basic data types)
  - Set operations like θ: $=, \subseteq, \subset, \supset, \supseteq$
- Recursively structured operation parameters, e.g.
  - π : nested projection attribute lists
  - σ : selection conditions on nested relations
- Additional operations
  - ν (*Nu*) = Nest
  - μ (*Mu*) = Unnest

NF^2: Nest and Unnest

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

$\longrightarrow$
$\nu_{B,C;D}(r)$

$\longleftarrow$
$\mu_D(r')$

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

NF^2: Nest and Unnest /2

Nesting not generally reversible:

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

$\not\longrightarrow$ $\nu_{B,C;D}(r)$

$\longleftarrow$ $\mu_D(r')$

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

NF^2: Partitioned Normal Form

The **Partitioned Normal Form (PNF)** requires a *flat key* on every nesting level of a nested relation.

PNF relation:

A	D	
	B	C
1	2 4	3 2
2	1 4	1 1
3	1	1

Non-PNF relation:

A	C
B	D
1 2	2 3
1 2	3 4

NF^2: Partitioned Normal Form /2

PNF relation and unnested equivalent relation

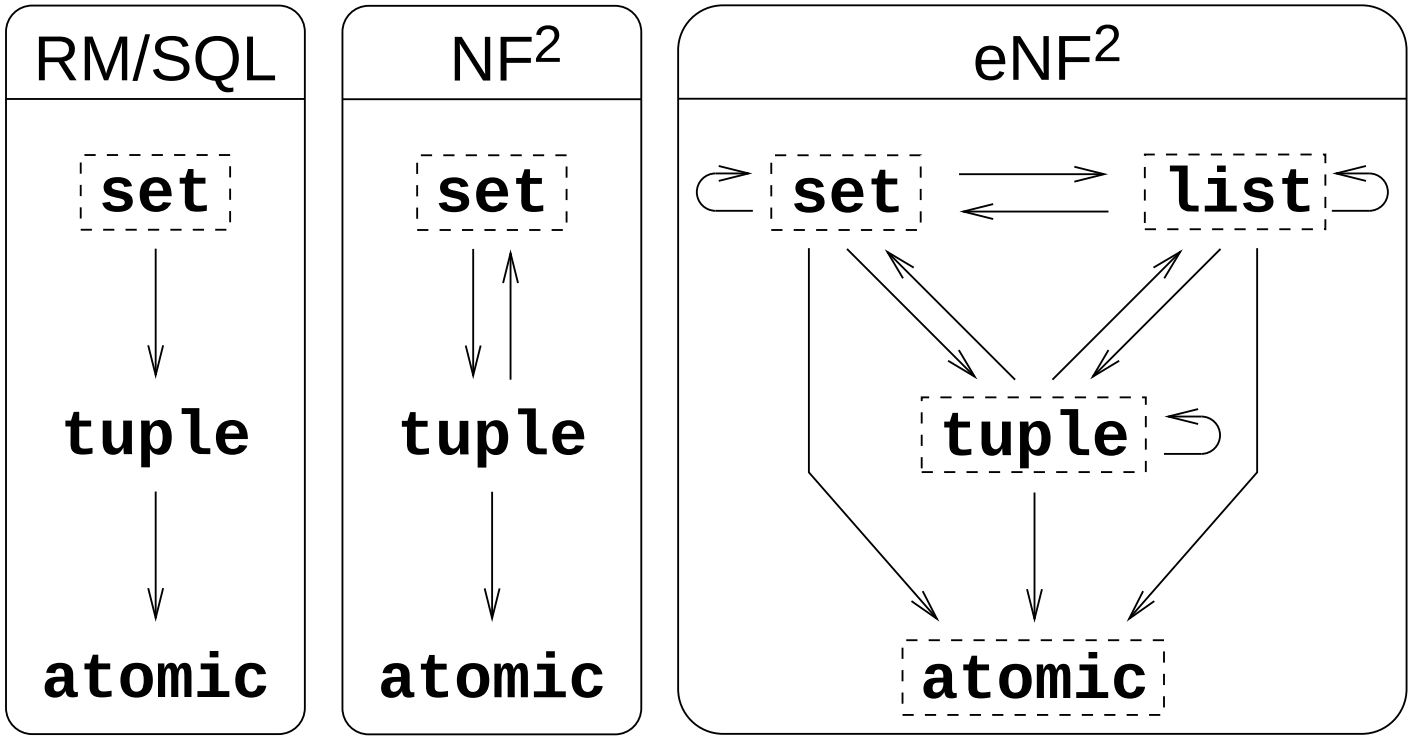
A	D	
	B	C
1	2	3
	4	2
2	1	1
	4	1
3	1	1

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

The *eNF*^{2} Database Model

- *Extended NF*^{2} *Model* (*eNF*^{2} Model)
- Relaxes *NF*^{2} model by introducing various type constructors and allowing their free combination
- Type constructors:
  - **`set`**: create a set type of the nested type,
  - **`tuple`**: tuple type of nested type,
  - **`list`**: list type of nested type,
  - **`bag`**: bag (multi-set) type of nested type
  - **`array[..]`**: array type of nested type
- First two are also available in RM and *NF*^{2}
- Additionally, can be combined freely,
  e.g. *set of bag of integer*

eNF^2: Combination of Type Constructors



eNF^2: Combination of Type Constructors /2

- Also called **Parameterizable Data Types**
- Construction based on input data types:
  - Basic/atomic data types
  - Input data type complex itself
- Define own operations for access and modification
- Similar to pre-defined parameterizable data types of programming languages
  - Generics in Java `java.util`
  - Templates in C++ STL
- Array, list, bag, and set type constructors are also called *collection data types*

eNF^2: Tuple Type Constructor

- *Tuple Type Constructor* (**`tuple`**) – fixed number of components of different (heterogeneous) types
- Access to components based on field (attribute) name
- Based on idea of Cartesian product
- Operations
  - **`tuple`** constructor
  - Component access using (`.`), e.g. `address.street`
- Example: pre-defined tuple types: *Date*, *Timestamp*

```
Date: tuple(Day: integer, Month: integer,
            Year: integer)
```

eNF^2: Set Type Constructor

- *Set Type Constructor* (**`set`**) – finite set of instances of same (homogeneous) type
- Contains no duplicates
- Value domain is power set of values in input type

```
Telephones: set(string)
```

eNF^2: Set Type Constructor /2

Set operations:

- **`set`**: Constructor: create an empty set
- **`insert`**: insertion of an element
- **`remove`**: deletion of an element
- **`in`**: test for containment of an element
- **`union`**: union of two sets
- **`intersection`**: intersection of two sets
- **`difference`**: set difference of two sets
- **`count`**: cardinality of a set

eNF^2: Combination of Set and Tuple

```
Price_Supplier: set(tuple(
                  Supplier: string,
                  Article: string
                  Price: integer))
```

- Prices of an article depending on supplier
- Equivalent to relational model

eNF^2: Bag Type Constructor

- *Bag Type Constructor* (**`bag`**) – finite multi-set of instances of same (homogeneous) type
- Allows duplicates
- Similar operations as set type constructor
  - Insertion and deletion considers duplicate entries
  - Containment test returns number of same entries
  - Union of bags keeps duplicate entries (`UNION ALL`)
  - Intersection returns minimal number of same entries of both input bags

eNF^2: Bag Type Constructor

- *List Type Constructor* (**`list`**) – finite sorted collection of instances of same (homogeneous) type
- Allows duplicates
- Operations similar to set +
  - **`insert`** at various positions: first, last, or *n*
  - **`append`** to concatenate two lists
  - Element access using explicit position **`[`***n*`]`**

eNF^2: Bag Type Constructor /2

- Authors of a document:

```
Authors: list(tuple(Firstname: string,
                    Lastname: string))
```

eNF^2: Bag Type Constructor

- *Array Type Constructor* (`array[..]`) – fixed (maximal) number of instances of a same (homogeneous) type
- Allows duplicates
- One-dimensional, multiple dimension by means of nesting arrays
- Access and modification of an entry by means of explicit position `[`*n*`]`

eNF^2: Bag Type Constructor /2

- Each employee can have up to 4 telephones

```
Telephones: array [1..4] of string
```

eNF^2: Comparison of Type constructors

Type	Duplicates	#Elements	Order	Element access	Heterogenity
`Tuple`	$\surd$	constant	$\surd$	Name	$\surd$
`Array`	$\surd$	constant	$\surd$	Index	—
`List`	$\surd$	variable	$\surd$	Iterator/Pos.	—
`Bag`	$\surd$	variable	—	Iterator	—
`Set`	—	variable	—	Iterator	—

eNF^2 Concepts in SQL:1999 and SQL:2003

- SQL:1999 introduced numerous extensions to type system
  - New data types: `BOOLEAN`, `LOB`
  - Distinct types
  - Type constructors: `ROW`,`ARRAY`
  - Structured object types with type hierarchies and methods
  - Object identities and `REF` type constructor
  - Multi-media data types
- SQL:2003 focused on other parts of the language (e.g. XML extensions), only few changes to type system
  - Bag type constructor `MULTISET`
  - XML data types
- Implementations in commercial DBMS most often do NOT comply to standard!!

ROW Type Constructor

- ROW implements tuple type constructor
- Example usage as embedded type:

```
CREATE TABLE Customers (
    Name VARCHAR(40),
    Address ROW (Street VARCHAR(30),
          Town VARCHAR(30),
          Zip VARCHAR(10)));
```

ROW Type Constructor /2

- Creation of tuples requires call of row constructor

```
INSERT INTO Customers
VALUES ('Myers', ROW('42nd','NY','111'));
```

ROW Type Constructor /3

- Components can be accessed using .

```
SELECT Name, Address.Town
FROM Customers;
```

ARRAY Type Constructor

- ARRAY of fixed length

```
CREATE TABLE Contacts (
    Name VARCHAR(40),
    PhoneNumbers VARCHAR(20) ARRAY[4],
    Addresses ROW (Street VARCHAR(30),
          Town VARCHAR(30),
          Zip VARCHAR(10)) ARRAY[3]);
```

ARRAY Type Constructor /2

- Creation of tuples requires call of array constructor

```
INSERT INTO Contacts
VALUES ('Myers',
          ARRAY['1234','5678'],
          ARRAY[ROW('42nd','NY','111')]);

UPDATE Contacts
SET PhoneNumbers[3]='91011'
WHERE Name='Myers';
```

`ARRAY` Type Constructor /3

- Array components can be accessed in two ways

  Explicit position: using the array position `[`*n*`]`

  Unnesting of collection: using the **`UNNEST`** operation in the **`FROM`**-clause allows declarative access to flattened relation

- Further operations: size **`CARDINALITY()`** and concatenation `||`

`ARRAY` Type Constructor /4

- Explicit position

```
SELECT Name, PhoneNumbers[1]
FROM Contacts;
```

- Unnesting of array

```
SELECT Name, Tel.*
FROM Contacts,
    UNNEST (Contacts.PhoneNumbers) Tel (Phone, Pos)
    WITH ORDINALITY
WHERE Name='Myers';
```

`MULTISET` Type Constructor

- `MULTISET` implements bag type constructor
- Allows creation of nested tables (NF^2)
- Can be combined freely with other type constructors (eNF^2)

`MULTISET` Type Constructor /2

- Operations:
  - **`MULTISET`** constructor
  - **`UNNEST`** as for arrays
  - **`COLLECT`**: special aggregate function to implement **`NEST`** operation
  - **`FUSION`**: special aggregate function to build union of aggregated multi-sets
  - **`UNION`** and **`INTERSECT`**
  - **`CARDINALITY`** as for arrays
  - **`SET`** to remove duplicates
- Predicates:
  - **`MEMBER`**: containment
  - **`SUBMULTISET`**: multi-set containment
  - **`IS A SET`**: test if duplicates exist

MULTISET Type Constructor /3

```
CREATE TABLE Departments (
    Name VARCHAR(40),
    Buildings INTEGER MULTISET,
    Employees ROW (Firstname VARCHAR(30),
          Lastname VARCHAR(30),
          Office INTEGER) MULTISET);
```

MULTISET Type Constructor /4

```
INSERT INTO Departments
VALUES ('Computer Science',
        MULTISET[29,30],
        MULTISET(ROW(...)));

INSERT INTO Departments
VALUES ('Computer Science',
        MULTISET[28],
        MULTISET(SELECT ... FROM ...);

UPDATE Departments
SET Buildings=Buildings
        MULTISET UNION MULTISET[17]
WHERE Name='Computer Science';
```

MULTISET Type Constructor /5

- Unnesting of a multi-set

```
SELECT Name, Emp.LastName
FROM Departments,
    UNNEST (Departments.Employees) Emp;
```

MULTISET Type Constructor /6

- Nesting using the **COLLECT** aggregation function

```
SELECT Name, COLLECT(hobby) AS hobbies
FROM Person NATURAL JOIN SpareTime,
GROUP BY Name;
```

eNF^2 in Oracle

- Oracle supports majority of standard functionality as part of its object-relational extension (since Oracle 8i)
  - Row type constructor as **`OBJECT`** type
  - Array type constructor as **`VARRAY`** type
  - Multiset type constructor as **`NESTED TABLE`** type
- Uses different syntax than standard

Oracle `OBJECT` type

- Type constructor and basis for object-oriented features of Oracle data model
- User-defined object types can be used for
  - Embedded structured attributes
  - (Object) table types
  - Variables
  - Parameters and return values of methods
- Needs to be defined explicitly by `CREATE TYPE` statement
- Constructor call required where ambiguities may exist

Oracle **CREATE TYPE** statement

```
CREATE TYPE AddressType AS OBJECT(
    Street VARCHAR(30),
    Town VARCHAR(30),
    Zip VARCHAR(10));

CREATE TYPE PersonType AS OBJECT (
    Name VARCHAR(40),
    Address AddressType);

CREATE TABLE Persons OF PersonType;

INSERT INTO Persons
VALUES ('M',AddressType('42nd','NY','1111'));
```

Oracle `VARRAY` type

- Similar to **`ARRAY`** from SQL:1999
- Can only be used indirectly through type definition with **`CREATE TYPE`**
- Actual array is stored as LOB
  - *in-line*: within record
  - *out-of-line*: separate from record
- Access through unnest operation **`TABLE`**

Oracle **VARRAY** type /2

```
CREATE TYPE AddressListType
AS VARRAY(4) OF AddressType;

CREATE TYPE PhoneListType
AS VARRAY(4) OF VARCHAR(20);

CREATE TABLE Contacts (
   Name VARCHAR(20),
   PhoneList PhoneListType,
   AddressList AddressListType);

INSERT INTO Contacts
VALUES ('Bates',
   PhoneListType(('1'),('2'),('3')),
   AddressListType(AddressType(...)));
```

Unnesting using the **TABLE** operator

- Used similar as **UNNEST** in SQL:1999: computes Cartesian product within nested record
- Access to flattened records of basic types may require **VALUE** operator

```
SELECT *
FROM Contacts, TABLE(Contacts.PhoneList) tel
WHERE VALUE(tel)='2';

SELECT *
FROM Contacts,
   TABLE(Contacts.PhoneList)
   TABLE(Contacts.AddressList);
```

Oracle `TABLE` type

- Similar to **`MULTISET`** from SQL:1999
- Like **`VARRAY`**: can only be used indirectly through type definition with **`CREATE TYPE`**
- Actual nested table is stored as separate table – needs to be named during outer table creation
- Access through unnest operation **`TABLE`**
- **`COLLECT`** aggregate function for nesting

Oracle **TABLE** type /2

```
CREATE TYPE EmployeeListType
AS TABLE OF PersonType;

CREATE TABLE Departments (
   Name VARCHAR(40),
   EmployeeList EmployeeListType)
NESTED TABLE EmployeeList STORE AS EmpTable;

INSERT INTO Departments
VALUES ('Product Development',
   EmployeeListType(
      PersonType('Lector',...),
      PersonType('Jason',...)));
```

Unnesting and Nesting Tables

```
SELECT *
FROM Departments,
    TABLE(Departments.EmployeeList) emps;

SELECT Companies.Name,COLLECT(Subsidiaries.Town)
FROM Companies NATURAL JOIN Subsidiaries
GROUP BY(Companies.Name);
```

Teil V

Object-oriented Data Models

Overview

- Object-oriented Database Management Systems
- Concepts of OO Data Models
- Concepts of ODBMS
- The ODMG Standard

Object-oriented Database Management Systems

- Based on the development of
  - Object-oriented analysis and design (OOA, OOD)
  - Object-oriented programming (OOP)
  - Object-oriented distributed computing (e.g. CORBA)
- Objectives:
  - More and advanced semantic concepts to describe real world facts
  - More efficiency for non-standard applications
  - Overcome impedance mismatch between RM and OOP

Definition

An **Object-oriented Database Management System (ODBMS)** is a database management systems that implements an object-oriented data model.

- Problem: there is no ONE object-oriented model, but several offering similar concepts
- Basic requirements were listed in *The Object-oriented Database System Manifesto* (Atkinson et al., 1989)
- ODMG supposed to provide a standard for ODBMS (1993), but limited support by commercial systems

History

- First research prototypes in the early ’80s (ORION, ITASCA, EXODUS)
- First commercial systems in the late ’80s/ early ’90s (GemStone, ONTOS, Objectivity, ObjectStore, Versant, Poet, O^2)
- First version of the ODMG standard in 1994
- Heydays during the mid-’90s
- Systems easily integrated support for Java, the WWW, XML, multi-media, spatial data, etc.
- Since mid-’90s: RDBMS integrated object-oriented concepts into their systems $\rightarrow$ ORDBMS
- Few commercial ODBMS survived until today

Different Approaches

Based on OOP data models: these ODBMS used the standardized data models of popular OO programming languages such as C++ (ONTOS, Objectivity, ObjectStore, Versant, Poet) or Smalltalk (GemStone), later on Java (Jasmine, JD4O), and added DBMS functionality (persistence, TXNs, collection types, queries, ...)

Extensions of relational models: introduce object-oriented concepts (types/classes, inheritance, object identity, methods, ...) in a relational model (Postgres, Illustra) or build on top of existing DBMS (Oracle, DB2, ...) $\rightarrow$ *object-relational DBMS*

Innate OO database models: developed independently of existing models and systems (O^2, ORION, Itasca)

Object Database Management Systems

- Most successful system were tightly integrated with OOPL
  - C++: Versant, Objectivity, Poet (now Versant Fast Objects)
  - Smalltalk: GemStone

  $\rightarrow$ solved problem of Impedance Mismatch!
- Object-relational systems provide similar concepts but are decoupled from programming language (advantages of logical data independence)

Advantages of ODBMS

- Provide semantically rich object-oriented modeling constructs
- Solve Impedance Mismatch
- Decreased implementation efforts
  - Developer needs knowledge of only one data model
  - No mapping code/layer required
- Efficiency for applications with complex data (navigational access, no joins required)

Disadvantages of ODBMS

- No logical data independence
  - Application changes may trigger schema changes
  - Schema changes require application changes
- Existing standard (ODMG) is hardly implemented
- Lack of interoperability
- Many systems with insufficient support for typical DBMS functionality
  - Logical views
  - Query languages
  - Query/access optimization
  - TXNs
  - Distribution
  - ...

The Object-oriented Database System Manifesto

- Though there is no common OO data(base) model, all models share important characteristics
- *The Object-oriented Database System Manifesto* summarized minimum requirements
  - Mandatory features: the Golden Rules
    - Object-oriented concepts
    - Database concepts
  - Optional features: the goodies
  - Open choices

OO Manifesto: Golden Rules /1

- OO concepts (part I of mandatory concepts)
  - Complex objects
  - Object identity
  - Encapsulation
  - Types and/or Classes
  - Class and/or Type Hierarchies
  - Overriding, overloading and late binding
  - Computational completeness
  - Extensibility

OO Manifesto: Golden Rules /2

- DBMS concepts (part II of mandatory concepts)
  - Persistence
  - Secondary storage management
  - Concurrency
  - Recovery
  - Ad Hoc Query Facility

OO Manifesto: The Goodies

- Optional concepts that may be implemented
  - Multiple inheritance
  - Type checking and type inferencing
  - Distribution
  - Design transactions
  - Versions

OO Manifesto: Optional Choices

- A very short discussion of further design considerations
  - Programming paradigm
  - Representation system
  - Type system
  - Uniformity

Complex Objects and Extensibility

- System must provide basic types: integer, float, boolean, character, strings
- System must provide at least the following type constructors for complex objects
  - **`tuple`**
  - **`set`**
  - **`list`**
- Complex objects may require specific operations, e.g.
  - **`deep`** or **`shallow delete`**
  - **`deep`** or **`shallow copy`**
- Extensibility: user-defined types can be used in the same way as system-defined types

Object Identity

The **object identity** of an object represents the fact, that it has an existence which is independent of all its values. Two objects are **identical** if they have the same object identity. Two objects are **equal** if they have the same values.

- Used for referencing objects (similar to OOPL pointers or references)
- Can be mapped to physical location or logical address (e.g. table, primary key)
- May be system- or user-generated

Object Identity /2

- Equality for complex objects
  - objects are **`shallow equal`** if they refer to identical component objects
  - objects are **`deep equal`** if they refer to (deep) equal component objects
- Object identity must be unique and non-volatile
  - in time: can not change and can not be re-assigned to another object, when the original object is deleted
  - in space: must be unique within and across database boundaries

Types and/or Classes

- Manifesto requires support of types or classes
- Abstraction of common features of a set of objects with the same characteristics
- Differences in definitions exist for OO programing languages

A **type** describes the **intension**, i.e. the internal (the set of encapsulated, possibly complex attributes) and external structure (the interface of the type with methods and their signatures).

A **class** consists of a type description, and in addition includes the notions of an object factory (e.g. constructors) and of a class **extension** which represents the set of all instances of each particular class.

Encapsulation

Encapsulation represents the postulation, that only the interface part of a type is visible to the users of the type, the implementation of the object is seen only by the type designer.

- Compile-time aspect of OOPL
  - Access modifiers, e.g. **`public`** (accessible for all), **`private`** (accessible within class), **`protected`** (accessible within class or derived class) checked by compiler
  - Strict encapsulation allows access to internal structure (attributes) only via interface (methods)
- Becomes run-time aspect for DBMS

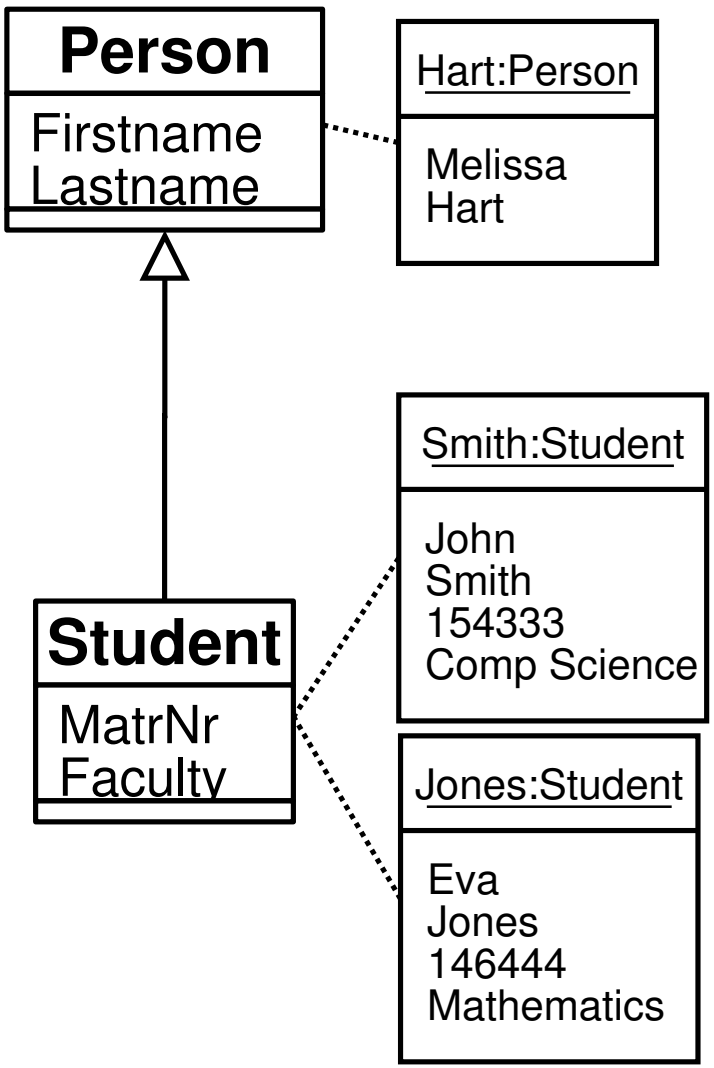
Type and Class Hierarchies

- Type and class hierarchies imply several aspects

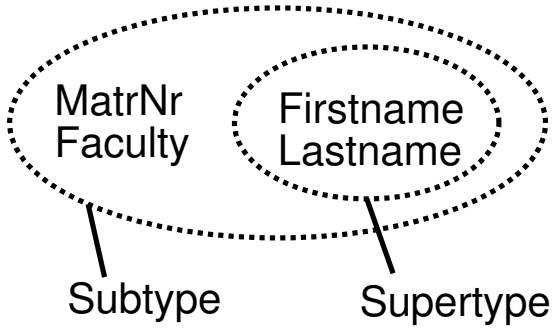
Specialization inheritance, or **intensional specialization**, between a super-type *A* and a subtype *B* means that the set of attributes and methods defined in *A* are a subset of the attributes and methods defined in *B*.

Substitution inheritance, or extensional specialization, between a superclass *A* and a subclass *B* means that the set of instances of *B* are a subset of the instance of class *B*. **Polymorphism** means that all instances of class *B* can be used wherever objects of class *A* can be used (substitutability).

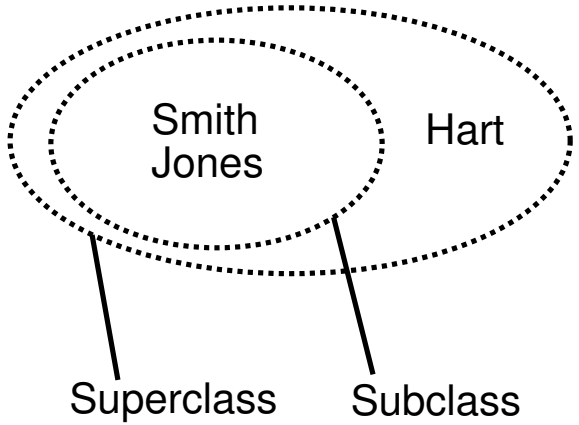
Type and Class Hierarchies /2



Intensional Specialization



Extensional Specialization



Overriding, Overloading and Late Binding

Different aspects of polymorphism:

Method overloading: allows method signatures to be re-defined within a class or in derived classes with different return or parameter types.

Method overriding: allows method implementations to be re-defined in derived classes.

Late Binding: the correct implementation of an overridden method of an object is determined and executed during run-time.

Computational Completeness

- ODBMS require tight integration with computationally complete programming language for
  - Method implementations
  - Navigational access
  - Control of DBMS functionality (persistence, TXNs, etc.)
- Not typical for RDBMS: only offer relational completeness
- Computational completeness often realized by means of integration with language of implemented data model (C++, Smalltalk, Java, ...)
- O^2 and object-relational systems also offer own languages

Concepts of ODBMS

- ODBMS require non-standard functionality beyond definition of data model
  - How are schemas defined (DDL) an implemented?
  - How is data made persistent and modified (DML)?
  - How is data accessed and queried?
  - How is consistency and concurrency controlled (advanced TXN models)

Schema Definition and Implementation

Schema definition:

Application-independent: ODMG ODL files

Application-dependent: source or object files of C++, Java, Smalltalk, C#, etc. application

Schema implementation:

Pre-processor: tool that takes source files (`.h`, `.java`, `.odl`, etc.) as input and creates modified or new (`.odl`) source code, possibly with added functionality / interfaces / super-classes, and, furthermore, installs the schema in the database

Post-processor: tool that derives the according schema information from an object file (`.o`, `.obj`, `.class`, etc.), possibly modifies the object file, and installs the schema

Sample Input Files

Java :

```
public abstract class Person {
    private String firstname, lastname;
}
```

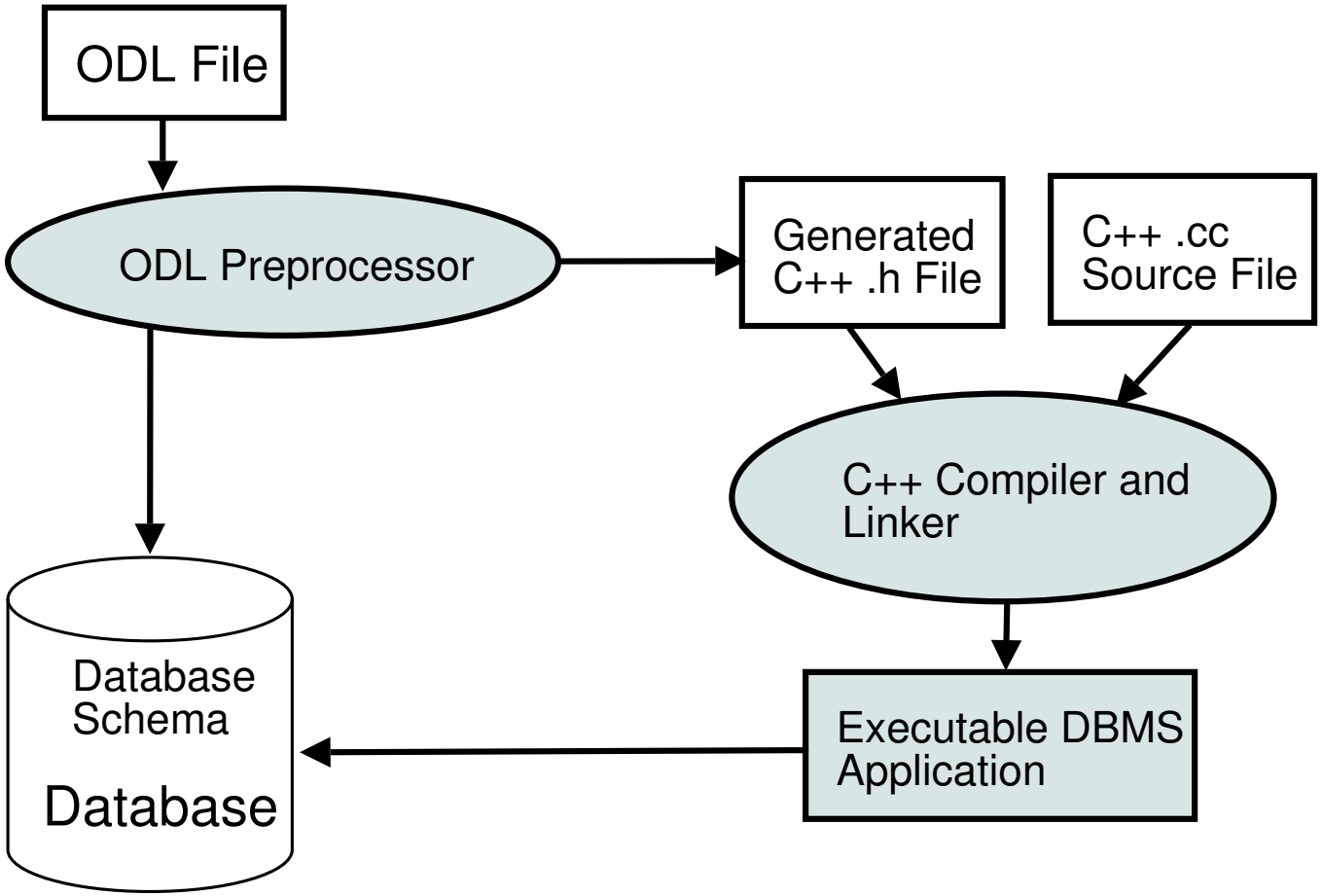
C++ :

```
class Person {
    private:
        char* _firstname, _lastname;
};
```

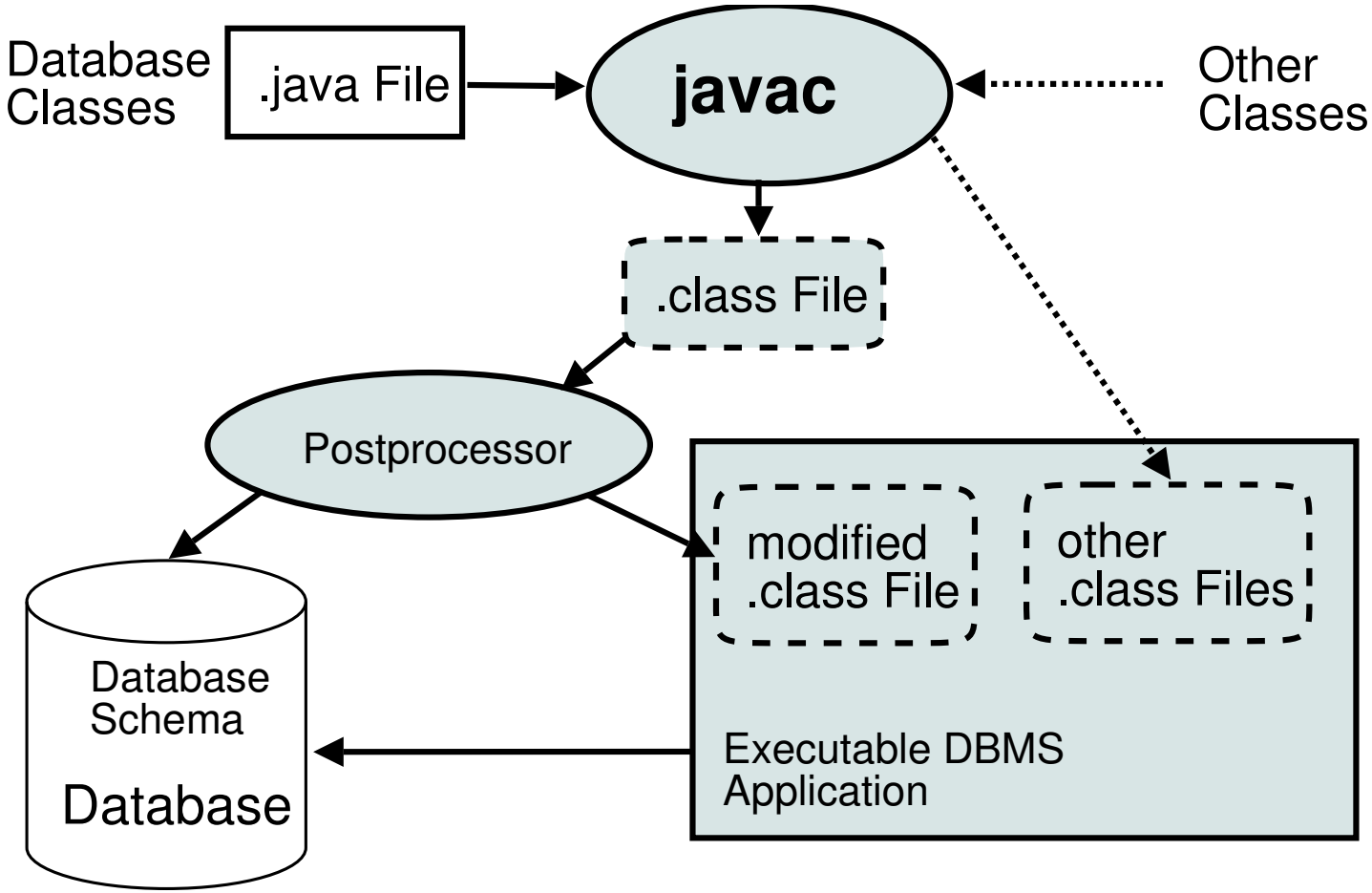
ODL :

```
class Person (
        extent persons) {
    attribute string firstname;
    attribute string lastname;
}
```

ODMG C++ Preprocessor



Generic Java Post-processor



Persistence Concepts

Persistent-capable classes are created by schema definition and implementation. They can have persistent as well as transient instances.

Persistence independence requires that persistent and transient objects of a persistent-capable class can be handled uniformly.

Persistence orthogonality requires that persistence is independent of the application schema types/classes, e.g. that no common base class or interface must be inherited or implemented.

Persistence Concepts /2

- Persistent objects can be modified by accessing their attributes and methods according by means of the OO programming language (restricted by encapsulation)
- Persistent objects can be created in several ways
  - Explicit persistence
  - Named root objects
  - Persistence by reachability (transitive persistence)
- Persistent objects can be deleted by either
  - Explicit removal from database $\rightarrow$ may lead to dangling references and memory leaks
  - Database garbage collection for unreferenced objects
- Alternatively, data manipulations can be carried out using the query language of some systems

Explicit Persistence

- Database run-time system offers explicit functionality to create a persistent object or make existing object persistent
- Create a persistent object, e.g. ODMG C++ Binding with overloaded **new**-operator

```
d_Database* db;
d_Ref<Person> p1 = new(db,"Person") Person(SSmith","
```

- make a transient object persistent, e.g. ODMG Java Binding

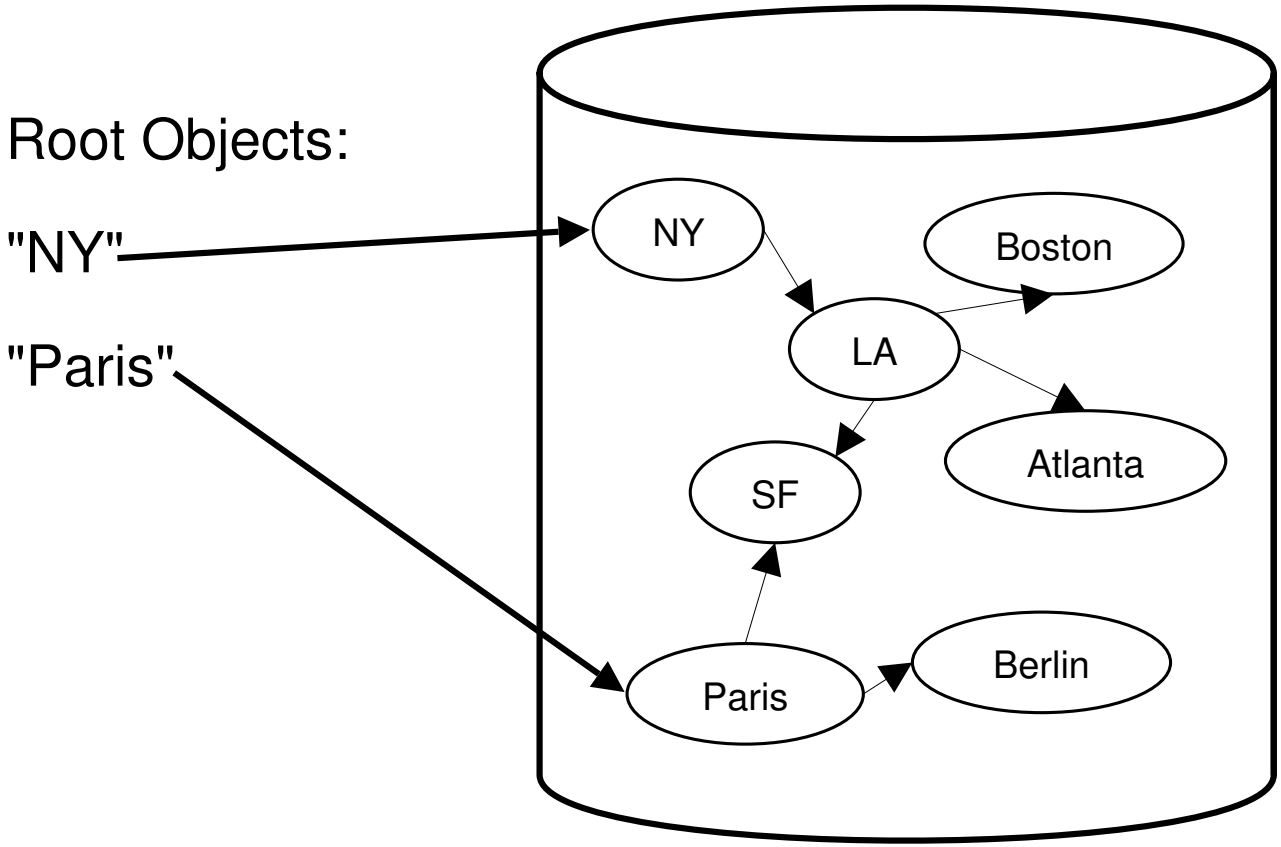
```
Person p1 = new Person(SSmith","Carl");
db.makePersistent(p1);
...
db.deletePersistent(p1);
```

Named Objects

- Retrieving the first object from an ODB by unique names
- Other objects can be accessed starting from this root object by following references
- Typical root objects: tree roots and collections
- Example: ODMG Java Binding

```
City c1 = new City("NY", 45.67, 12.34);
db.bind(c1, "NY");
...
City c = (City) db.lookup("NY");
```

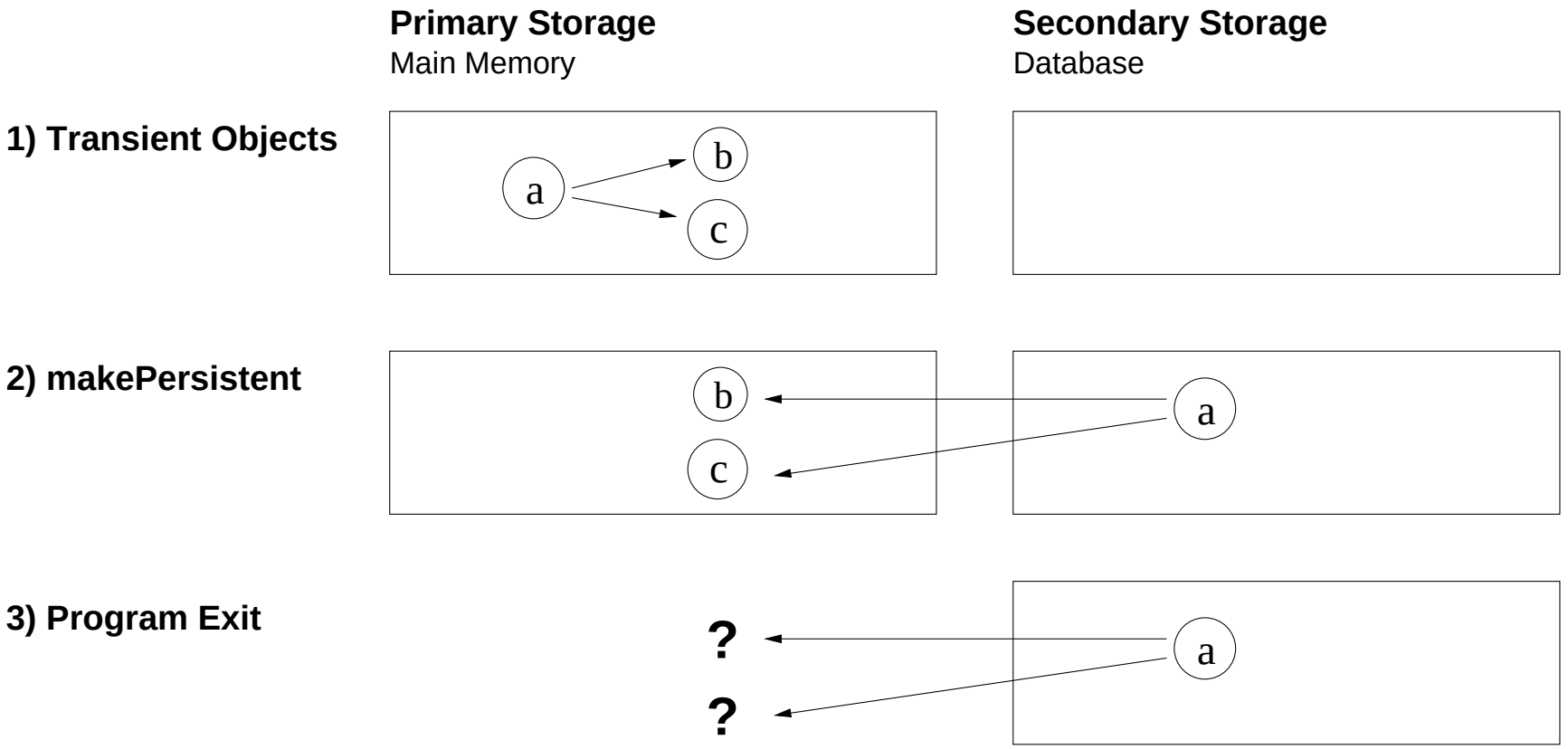
Named Objects /2



Persistence By Reachability

- Possible consistency problem: object *a* is made persistent and referenced objects *b* and *c* remain transient
  - References from persistent to transient storage
  - References become invalid when application is stopped or *b* and *c* are removed from memory
- Solution: if one object is made persistent, all transitively reachable objects are made persistent, too
- Example: ODMG Java Binding, JDO

Persistence By Reachability /1

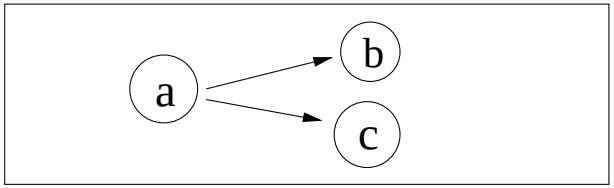


Persistence By Reachability /2

Primary Storage
Main Memory

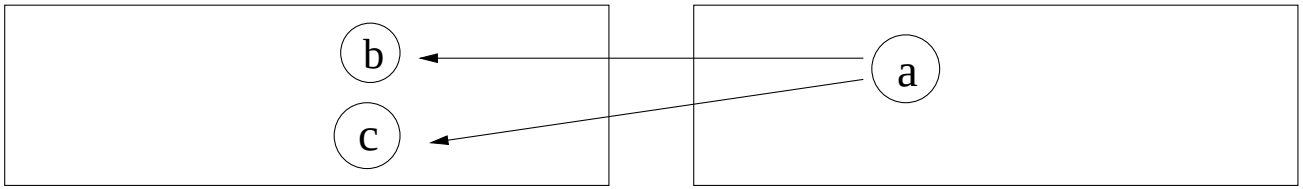
Secondary Storage
Database

1) Transient Objects

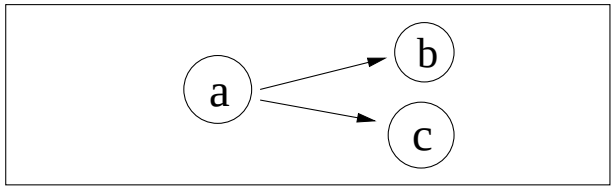


2) makePersistent

a) object itself



b) all reachable Objects



OO Query Languages

- What are the results of OO queries?
  - Relations of tuples (without defined object type)
  - Existing objects
  - Objects created by queries
- How are queries used?
  - Ad-hoc queries: arbitrary queries, e.g. from command line tools
  - Query API: usage from within application program
- How are OO features accessed by query language?
  - Complex objects and navigation
  - Type hierarchies
  - Methods

OO Query Languages: Results

Relational query semantics: returns set, bag, or list of struct (tuple) of literal data types as in relational query languages

Object selection semantics: selects existing persistent objects from the database

Object creation semantics: creates new (transient) objects based on values derived from persistent objects by calling a constructor in the `SELECT` clause (π)

- ODMG OQL supports all result types
- Most commercial systems support only object selection

OO Query Languages: Usage /1

- Ad-hoc queries, e.g. using ODMG interface OQLQuery, require input
  - Named root objects
  - Named extensions
  - Collections bound to variable

```
OQLQuery query = new OQLQuery();
query.create(ßtudents");
DBag allStudents = (DBag) query.execute();
```

OO Query Languages: Usage /2

- Alternative: query interface bound to collection classes for object selection
  - E.g. `query()`-method defined in ODMG collection classes
  - Specification of predicate

```
DBag badStudents = (DBag)
   students.query("grade > C");
DBag myStudents = (DBag)
   students.query("
      EXISTS s IN this.lectures:
      lecture.title ='ADBM'");
```

OO Query Languages: OO Features

- Complex objects and navigation
  - Following path expressions using ”.”-notation
  - Collection-type references require unnesting or nested queries
- Type inheritance / specialization
  - Access to flat or deep extension
- Methods
  - Overriding, Overloading, Late Binding must be supported
  - Problem: may have side effects, e.g. read-only query calls method that changes attribute values

Extended Transactional Concepts

- Support for ACID transactions in most systems
- Also extended transactional concepts

  Nested transactions: hierarchical structure of transactions to support complex operations on complex objects

  Design transactions: typical requirement in non-standard applications long transactions avoiding concurrency problems, e.g. including check-out/check-in, possibly in conjunction with workspaces and versioning

The ODMG Standard

- Not well supported, but summarizes common features of ODBMS
- Influenced development of, for instance, JDO mapping standard
- Consists of several pieces
  - ODMG Object Model
  - Object Definition Language (ODL)
  - Object Query Language (ODL)
  - Language Bindings
    - C++
    - Smalltalk
    - Java

Object Definition language ODL

- Based on OMG IDL (CORBA Interface Definition Language)
- Implementation-independent schema definition
- Provides means to describe
  - Modules (application schemata)
  - Interfaces and Classes
  - Inheritance
  - Attributes and Relationships
  - Operations (method declarations)
  - Exceptions
- Translated to implementation language using pre-processor

ODMG ODL: Classes

```
class Employee (
      extent employees) {
  attribute long employeeNr;
  attribute struct Name {
      string firstname;
      string lastname } name;
  attribute Date dob;
  attribute List<string> tel;
  ...
  void raise_salary (in short amount);
};
```

ODMG ODL: Classes /2

```
class Student : Person (
      extent students, key matrnr)
   attribute char matrnr[6];
   attribute string faculty;
   attribute set<struct<float grade, string lecture>>
      grades;

   relationship Person mother inverse Person::child;
   relationship Person father inverse Person::child;

   float avgGrade ()
      raises (no_Grade);
   void enlist (in string faculty)
      raises (already_enlisted);
};
```

ODMG ODL: Inheritance

- Multiple inheritance only for interfaces

```
interface Student { ...};
interface Employee { ...};
interface PhDStudent : Student, Employee { ...};
```

- Class inheritance with **extends** supports only singular inheritances

```
class Book extends Publication { ...};
```

ODMG ODL: Relationships

- m:n-Relationship
  attends(Student[0,], Course[0,*])*

```
class Course {
   relationship set<Student> attended_by
      inverse Student::attends;
   ...
};

class Student {
   relationship set<Course> attends
      inverse Course::attended_by;
   ...
};
```

Object Query Language OQL

- Declarative object-oriented query language based on O^2 system
- Very comprehensive and flexible, but only small subset supported by most systems
- Syntax
  - SFW-style queries
  - Additionally: every named object (e.g. root) or collection (e.g. class extent) is a query
  - Specification of predicates for collection-bound queries

ODMG OQL: Simple Example Queries

```
SELECT e FROM employees e;

employees;

newyork.neighbor_cities;
```

ODMG OQL: SFW Queries /1

- **SELECT**-clause
  - Allows method calls and sub-queries
  - Allows constructor calls
  - **DISTINCT** for result type **set<...>**

```
SELECT DISTINCT STRUCT (e.name, projects:(
    SELECT p.projectID
    FROM e.particpates_in p))
FROM employee e;

Result type:
set<struct<name: string,
    projects: bag<long>>>
```

ODMG OQL: SFW Queries /2

- Single object as result type using **ELEMENT**

```
ELEMENT (SELECT p
    FROM projects p
    WHERE projectID = 4711)
```

- Type casting for type hierarchies

```
SELECT (Employee) a
FROM apprentices a;
```

ODMG OQL: SFW Queries /3

- Object creation by means of constructor call

```
SELECT Component(
    productNr: p.productNr,
    title: p.title,
    status: p.status)
FROM products p
WHERE status = äctive";
```

ODMG OQL: SFW Queries /4

- **FROM**-clause
  - Class extension, collection-valued attribute or reference, result of a method call, or sub-query
  - Automatic conversion to **bag**

```
SELECT e.name
FROM (SELECT p.managed_by
    FROM projects p
    WHERE p.status = 'finished') e
WHERE e.salary > 9000
```

ODMG OQL: Path Expressions

- Inner path steps can not be multi-valued, e.g. the following is not possible:

```
proj.participants.name;
```

  Instead:

```
SELECT e.name
FROM proj.participants e
```

- Multiple multi-valued references

```
SELECT prod.name, prod.cost
FROM employees.participates_in proj,
   proj.develops prod
```

ODMG OQL: Further Concepts

- Arbitrary method calls $\rightarrow$ problem of side effects
- SQL-like functionality
  - Joins (can often be replaced by following path references)
  - Grouping and aggregate functions
  - Sorting $\rightarrow$ result is of **`list`** type
  - exists and all quantifiers in predicates, e.g.

```
EXISTS p IN projects:
   p.status = 'finished'
```

  - Named queries as simple view concept

ODMG Language Bindings

- Exist for C++, Smalltalk, and Java
- Include language-dependent ODL (requires property file for DB specific aspects, such as extents, keys, referential integrity)
- Offer API and type mappings
  - Type mapping for basic data types
  - Type mappings for ODL type constructors (collections, struct, references)
  - Type mapping for pre-defined object types (Date, String)
  - Mapping of exception classes
  - Basic API for accessing database functionality (database connections, queries, transactions)
  - Schema access (catalog, reflection API)

ODMG Java Binding

- Mapping of data types

ODL	**Java**
Long	int
Float	float
Boolean	boolean
String	String
Long	int
Long	int
Date	java.sql.Date
Time	java.sql.Time
...	...

ODMG Java Binding /2

- Mapping of collection classes in `org.odmg`

ODL	Java
Set	DSet
Bag	DBag
Array	DArray
Map	DMap
...	...

- In C++ template classes with prefix `d_`, e.g.

```
template<class T> class d_Set :
   public d_Collection<T> { ... };
```

ODMG Java Binding /3

- API also in `org.odmg`
- Interfaces
  - `Implementation` (object factory)
  - `Database`
  - `Transaction`
  - `OQLQuery`
- Offer method declarations for minimum functionality
- Furthermore: exception classes

Teil VI

Object-relational Data Models

Overview

- Object-relational Database Models
- Concepts of Object-relational Database Models
- Object-relational Features of SQL:2003
- Object-relational Features in Oracle10g

Object-relational Database Models

- Based on relational/SQL database model
- Extensions
  - Type constructor for objects $\rightarrow$ object types
  - Object identity and references
  - Collection type constructors for nested tables and complex objects
  - Reference type constructor
  - Methods for object types
  - Object tables
  - Inheritance: type and table hierarchies
  - Object views and view hierarchies
  - Recursive queries

Relational vs. Object-relational

- Advantages inherited from relational SQL database model
  - Variety of integrity constraints
  - Concept of data dictionary
  - Descriptive Query language
  - Views
- Introduction of non-orthogonal features
  - Triggers versus Methods
  - Generic DML operations versus Methods
  - Primary/foreign keys versus object references
- ORDBMS still have impedance mismatch

Object Types

- Constructor for structured types
- Components can be
- Can be used for object tables, as embedded data type, or for variables and parameters
- Additions compared to **`tuple`** type constructor
  - Object types allow inclusion of object identity as ”hidden” attribute
  - Methods can be defined for object types
  - Specialization inheritance between object types (intension)

Object Identity

- Non-volatile and unique property of an object that describes it existence independently of its values
- Maybe generated by system or defined by user
- Used for references
- ”Hidden column” of a table

Reference Type Constructor

- Reference types are special data types that reference objects of a specified type
- Value is OID of an object of specified type
- Allows reference to an object similar to a programming language reference or pointer
- Scope of a reference can be specified $\rightarrow$ because referenced type may be used in different places, e.g. different object tables or views of the same type
- Using references in a query can replace join computation

Methods

- Methods can be defined for object types
- Includes two parts
  - Declaration of signature: defines interface of method
  - Method implementation: encapsulated implementation in either external programming language (e.g. C, C++, or Java) or DBMS programming language (e.g. Oracle's PL/SQL)
- Constructors, overriding, overloading, and late binding must be supported
- Often additional aspects
  - Declaration of read-only methods to avoid side effects
  - Special methods for comparison to use operators $=$, $>$, $<$, $\leq$, and $\geq$

Object Tables

- Object tables are tables defined based on object type, i.e. they contain objects instead of tuples
  - Have a ”hidden” OID column
  - Always have a set semantic because of unique OID
  - Can be scope of a typed reference
  - Methods can be used on objects
  - Can be part of an table hierarchy based on according type hierarchy
- 1-to-many relationship between types and tables, i.e. the same type can be used to define several tables

Type and Table hierarchies

Type hierarchies: intensional specialization between types, i.e. a sub-type can inherit attributes and methods from a super-type

Table hierarchies: object tables defined based on types which are in a super-type / sub-type relationship can (optionally) be in an extensional specialization relationship, i.e. there can be a subset relation among object tables

- Querying object tables can involve the
  - flat extension, i.e. only the super-table
  - deep extension, i.e. objects in super-table and all derived sub-tables

Object views and view hierarchies

- Views can just like object tables be defined based on a type
- To comply with type definition two possible ways
  - view definition query must return objects of view type
  - view definition calls type constructor in `SELECT`-clause, OIDs are temporary
- According to object table hierarchies, views can be in a view hierarchy with extensional specialization

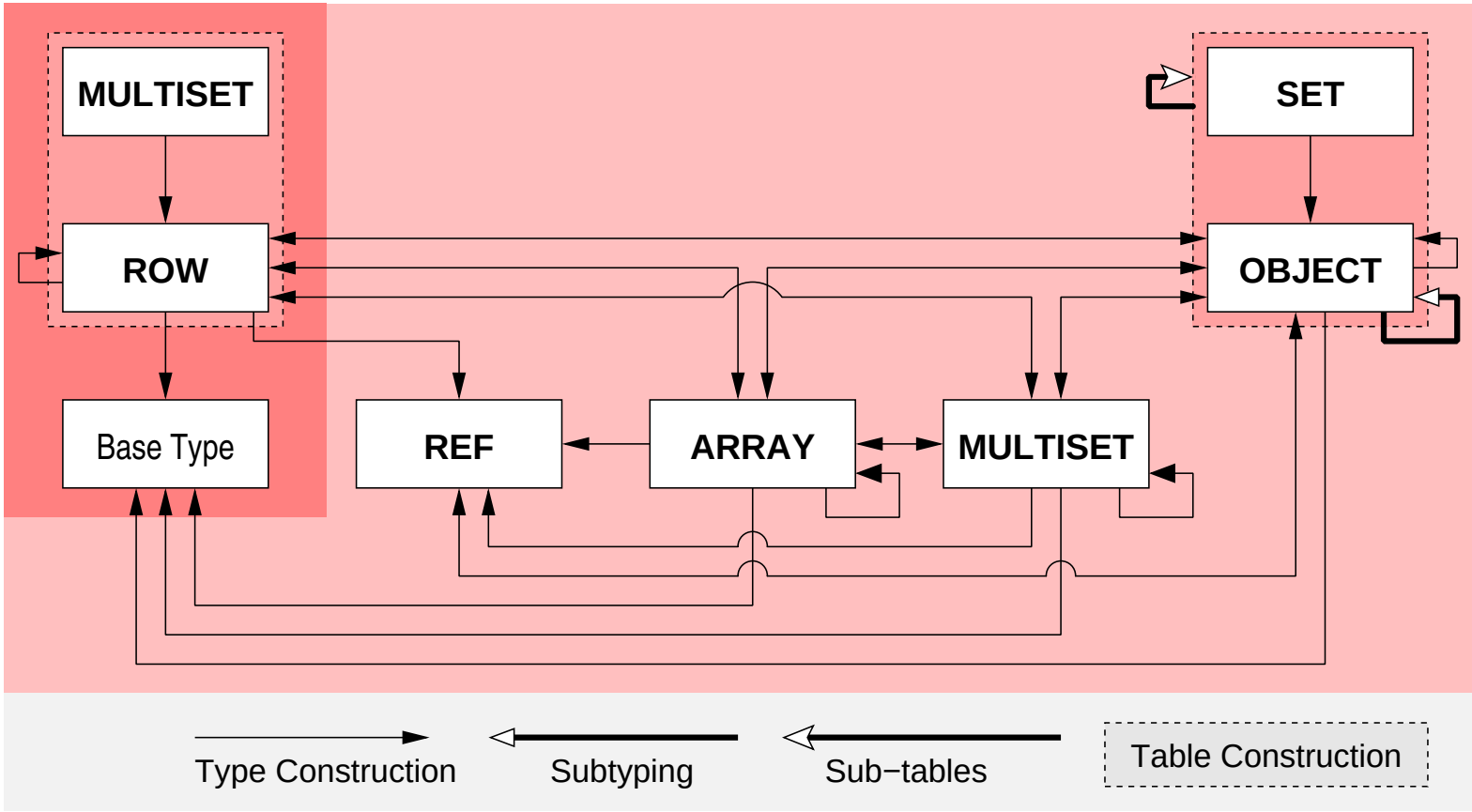
Object-relational Features of SQL:2003

- Majority of object-relational concepts introduced already with SQL:1999
- Reaction to growing popularity of ODBMS and inclusion of object-relational aspects in commercial RDBMS
- Special importance: extensibility opened standard and systems for new applications
  - Multi-media
  - Spatial data and Geographic Information Systems
  - Engineering data
  - Document management and work-flow systems
  - Semi-structured data and WWW contents
  - ...

SQL:2003 Type System

- Support for the following type constructors
  - **`ROW`** - for tables and embedded row types
  - **`OBJECT`** - for object tables and embedded object types
  - **`SET`** - only for object tables
  - **`ARRAY`** - for embedded object types
  - **`MULTISET`** - for tables and nested tables
  - **`REF`** - for references to object tables and views

SQL:2003 Type System



Object Types

- Object type constructor for structured types

```
CREATE TYPE Address_Type AS (
    street VARCHAR(30),
    zip VARCHAR(10),
    town VARCHAR(30)
) NOT FINAL

CREATE TYPE Customer_Type AS (
    cnr INT,
    name VARCHAR(30),
    address Address_Type
) NOT FINAL
```

Object Tables

```
CREATE TABLE customers OF Customer_Type (
    REF IS oid SYSTEM GENERATED)
```

- `Customer_Type` is table type
- `Address_Type` is embedded column type

Object Identifiers

- **`REF IS ...USER GENERATED`**
- **`REF IS ...SYSTEM GENERATED`**
- **`REF FROM`** (*`attributelist`*)
  OID is derived from other attributes, e.g. primary key

Reference Type Constructor

```
<attributename> REF(<typename>)
    [ SCOPE scopedescription ]
```

Scope is

- One specified object table
- One specified object view
- Any of the tables and views of this type, if not specified

Reference Type Constructor /2

```
CREATE TYPE Employee_Type AS (
    enr             INTEGER,
    name            VARCHAR(30));

CREATE TABLE employees OF Employee_Type;

CREATE TABLE departments (
    name VARCHAR(10),
    manager REF(Employee_Type) SCOPE(employees),
    members REF(Employee_Type)
                    SCOPE(employees) ARRAY[10]
)
```

Reference Type Constructor /2

- Following a reference using
  - **DEREF**-operator

```
SELECT DEREF(manager)
FROM departments;
```

  - Arrow operator

```
SELECT name, manager->name
FROM departments;
```

$\rightsquigarrow$ implicit join !

Type Hierarchies

- No multiple inheritance
- Control of further sub-typing
  - **`NOT FINAL`**: definition of sub-types allowed
  - **`FINAL`**: no sub-types allowed

```
CREATE TYPE Person_Type AS ( ...) NOT FINAL;
CREATE TYPE Customer_Type UNDER Person_Type AS
    ( ...) FINAL;
```

Table Hierarchies

- Object tables can be defined to be in extensional specialization hierarchy if according types are in a type hierarchy

```
CREATE TYPE Person_Type AS ( ...) NOT FINAL;
CREATE TYPE Customer_Type UNDER Person_Type AS
    ( ...) FINAL;

CREATE TABLE Persons OF Person_Type;
CREATE TABLE Customers OF Customer_Type
   UNDER Persons;
```

- If defined, there is a subset relation between sub-table and super-table
- If not defined, only intensional specialization holds

Type and Table Hierarchies in SQL:2003

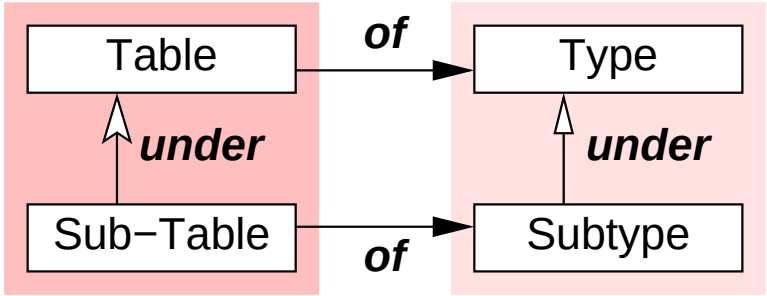
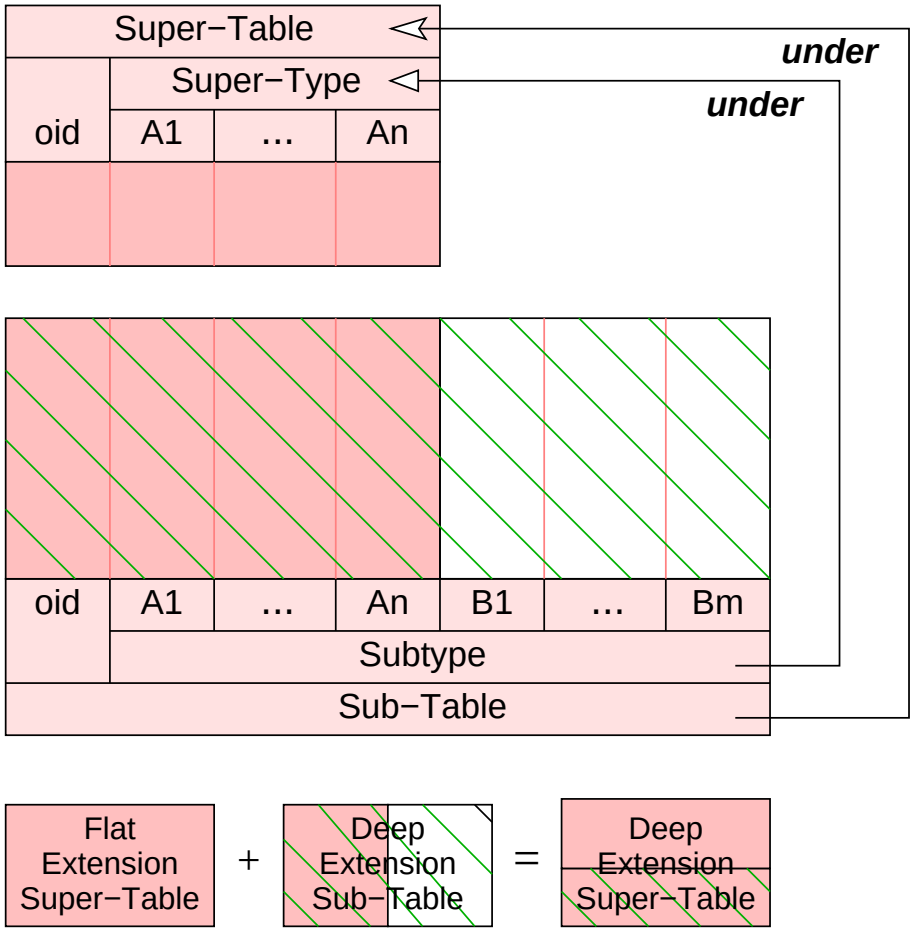


Table Hierarchies in SQL:2003



Accessing Table Hierarchies

- Default behavior: deep extension is queried when accessing super-table
  - All persons and customers, only person attributes

```
SELECT * FROM Persons;
```

  - All customers with person and customer attributes

```
SELECT * FROM Customers;
```

- Flat extension of persons using keyword **ONLY**
  - Persons, which are no customers

```
SELECT * FROM ONLY(Persons);
```

Methods

- SQL-Functions/Procedures defined for object types
- Implicit **`SELF`**-parameter similar to **`this`** in C++ or Java to access object itself
- Separation of declaration (part of the type) and implementation of a method
- For all attributes `set` and `get`-methods are generated automatically, but can be overridden

Methods /2

```
CREATE TYPE Customer_type AS (
    ...
) NOT FINAL
METHOD overall_orders()
       RETURNS DECIMAL(9,2));

CREATE METHOD overall_orders() FOR Customer_Type
BEGIN
    ...
END
```

Object Views

```
CREATE TYPE Person_Type AS (
   personalnr      INTEGER,
   name            VARCHAR(30),
   salary          DECIMAL(8,2),
   manager         REF(Person_Type))
NOT FINAL REF USING INTEGER;
```

```
CREATE VIEW RichEmployees OF Person_Type (
   REF IS oid USER GENERATED,
   manager WITH OPTIONS
      SCOPE RichEmployees
) AS SELECT RichEmployees(personalnr),
      personalnr, name, salary,
      RichEmployees(manager->personalnr)
   FROM ONLY (Employees)
   WHERE salary > 5000;
```

View Hierarchies

- View of sub-type with exactly one super-view of the according super-type
- OID and attribute restrictions are inherited
- Table used in sub-view definition must be sub-table of table used in super-view
- Subset relation among view extensions
- Definition

```
CREATE VIEW <viewname> OF <typename>
UNDER <super-view-name> [(
    <column-options>
)] AS <view-definition-query>
```

Recursive Queries /1

- Independent of object-relational concepts, but useful for dealing with complex objects
- Typical applications: hierarchical tree-like data (e.g. bill of material-queries, transitive closure in networks (flight plans, street maps)
- Based on re-usable query in **`WITH`**-clause

```
WITH <query-name> AS ( SFW-block )
    SFW-block-using-named-query
```

Recursive Queries /2

- Recursion:

```
WITH RECURSIVE <recursive-table> AS (
      SELECT ... FROM <table> ...
   UNION
      SELECT ...
      FROM <table>, <recursive-table> ...)
SELECT ... FROM <recursive-table> WHERE ...
```

Recursive Queries /3

machine

part	component	amount	partcost
car	motor	1	5356
motor	plunger	4	124
motor	crankshaft	1	89
motor	gear	1	560
gear	clutch	1	290
...	...	...	...

Recursive Queries /4

```
WITH RECURSIVE partlist (part, amount, cost)
    AS (SELECT component, 1, 0.00
        FROM machine
        WHERE part = 'car'
    UNION ALL
        SELECT m.component, m.amount,
            m.amount * m.part_cost
        FROM partlist p, machine m
        WHERE p.component = m.part)
SELECT part, sum(amount), SUM(cost)
FROM partlist
GROUP BY part;
```

Recursive Queries /5

- Computation: breadth first vs. depth first
  - Influences result tuple order
  - Can be specified

```
WITH RECURSIVE ...
SEARCH BREADTH | DEPTH FIRST BY <attrib-list>
...
```

- Query safety: endless recursion (cycles) might occur → user's responsibility, but syntax support

```
CYCLE <attrib> SET cyclemark TO 'Y' default 'N'
USING cyclepath
```

Object-relational Features in Oracle10g

- Majority of object-relational features supported
  - Object types +
  - Object identity and references +
  - Collection type constructors for nested tables and complex objects +
  - Reference type constructor +
  - Methods for object types +
  - Object tables +
  - Type hierarchies +
  - Table hierarchies -
  - Object views and view hierarchies +
  - Recursive queries +
- Different syntax from standard

Object Types and Object Tables

```
CREATE TYPE Person_Type AS OBJECT (...);
CREATE TABLE Persons OF Person_Type;
```

- Interpretation
  - Table with columns for type attributes
  - Each row gets assigned OID

Type Hierarchies

- Intensional specialization

```
CREATE TYPE Student_Type UNDER Person_Type ( ...);
```

- **No table hierarchies (extensional specialization)**
- Nevertheless, substitutability granted:
  - Table of super-type may contain objects of sub-types
  - Column of super-type may contain embedded object of sub-type

Object Identifiers

- Default: hidden, system-generated OID
  - 16 Byte per OID plus extra index structure for mapping to tuples
- Alternative: primary key is user-generated OID
  - Save disk space, but uniqueness is not granted

```
CREATE TABLE Stocks OF Stock_Type (
    StockID PRIMARY KEY)
OBJECT IDENTIFIER IS PRIMARY KEY ;
```

Reference Type Constructor

- Logical references using OIDs

```
CREATE TYPE Employee_Type AS OBJECT (
    ...,
    manager REF Employee_Type);
```

- Definition of scope as integrity constraint for object tables

```
CREATE TABLE Employees OF Employee_Type (
    manager SCOPE IS Employees);
```

- Insert requires query to retrieve value, e.g.

```
INSERT INTO Employees
SELECT 'John Miller', 20000, REF(e)
FROM Employees e WHERE e.name = 'Jake Jones';
```

Methods

- Implementation using PL/SQL, C, or Java)
- **`SELF`** as implicit Parameter
- Constructor generated implicitly
- Overloading, overriding, and late binding supported
- Method declaration in type definition, method implementation as part of type body specified separately
- Pre-defined interface for special functions used by DBMS for sorting and grouping based on complex object types
  - **`MAP MEMBER FUNCTION`**: test for equality
  - **`ORDER MEMBER FUNCTION`**: comparison

Methods /2

```
CREATE TYPE Employee_type AS OBJECT (
    ...
    salary NUMBER,
    MEMBER FUNCTION yearly RETURN NUMBER
);

...

CREATE TYPE BODY Employee_type IS
    MEMBER FUNCTION RETURN NUMBER IS
    BEGIN
        RETURN 12*salary+bonus;
    END;
END;
```

Recursive Queries

- Supported using **`CONNECT BY PRIOR`** syntax

```
SELECT component, SUM(part_cost), SUM(amount)
FROM machine
START WITH part = 'Auto'
CONNECT BY PRIOR component = part
GROUP BY component;
```

Teil VII

Semi-structured Database Models

Overview

- Introduction
- XML
- XML: DTDs and XML Schema
- XPath and XQuery

Semi-structured Data

- Semi-structured data/documents
  - Data with an internally encoded, often changing, and not strongly typed structured
  - Used for data exchange or the WWW
- No explicit structure
  - Unknown, ambiguous, not required/needed
  - Explicit description of structure too complex and expensive
- Data not easily representable in tables
  - Optional or alternative parts
  - Repetitions
  - Order is relevant

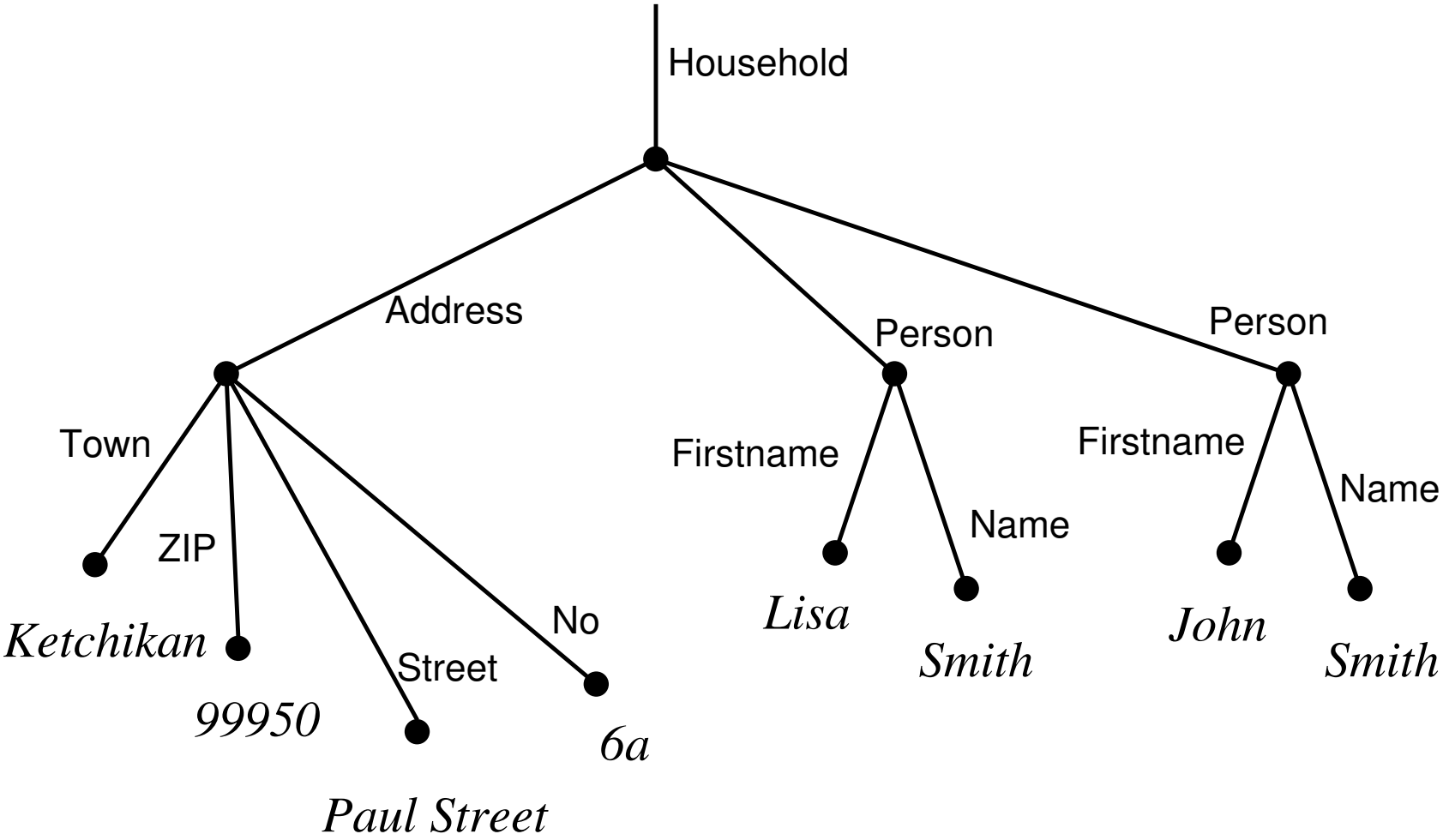
Features of semi-structured data

- No centrally stored schema, schema is part of data/document („self-describing“)
- Changing structures
- May contain data without further structure
- No or few data types (no integrity control)
- Great number of possible attributes
- Mix of data and schema

Data Models for Semi-Structured Data

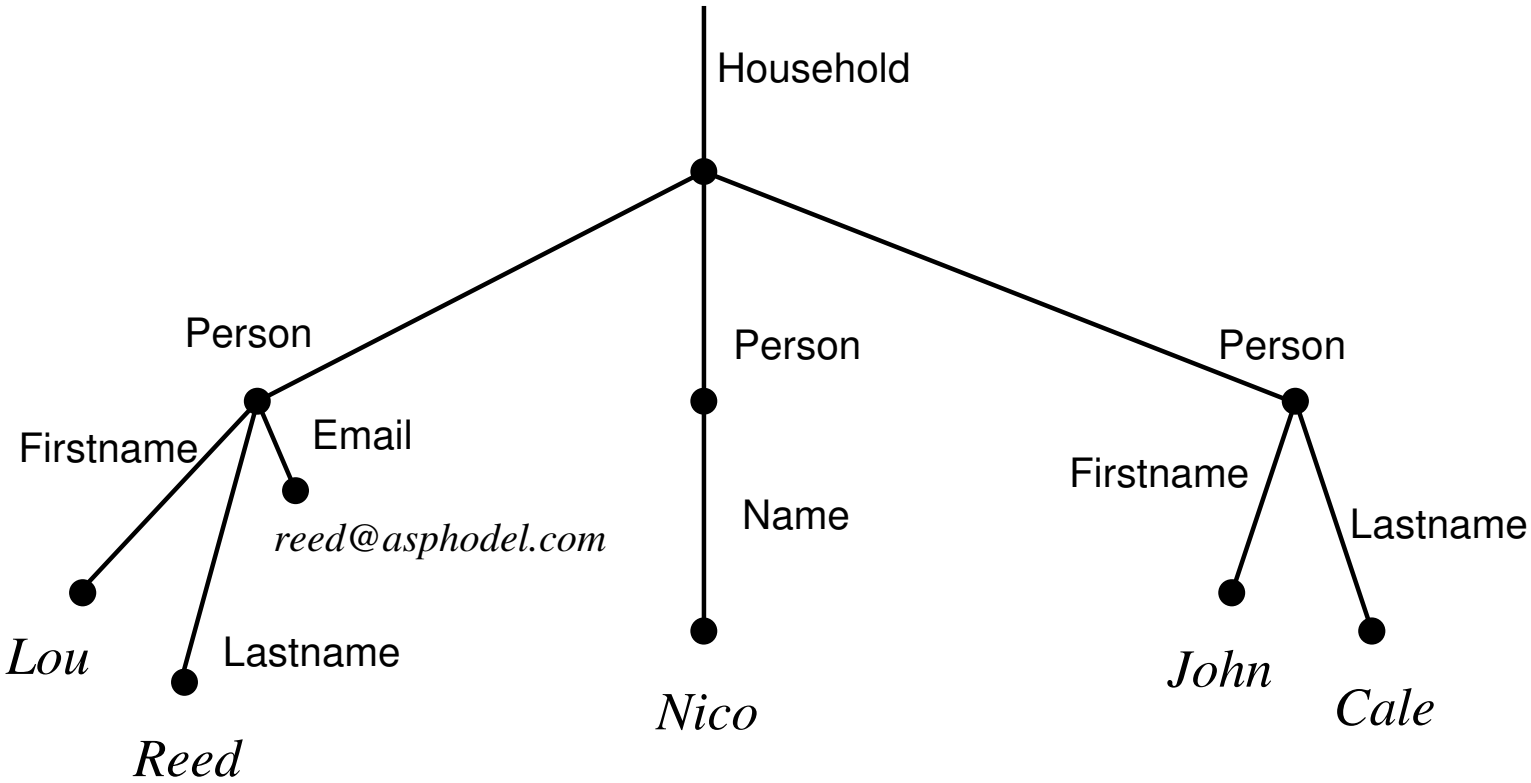
- Graph-based Models
- Examples: Object Exchange Model (OEM), Extensible Markup Language (XML)
- Document modeled as graph with
  - Edges with element tag names
  - Nodes with attribute/value pairs
  - Leaves with values (Strings)
  - Root node
- Node = object

Example Graph



Flexibility

- Different structures for persons



Representation of other Data Models

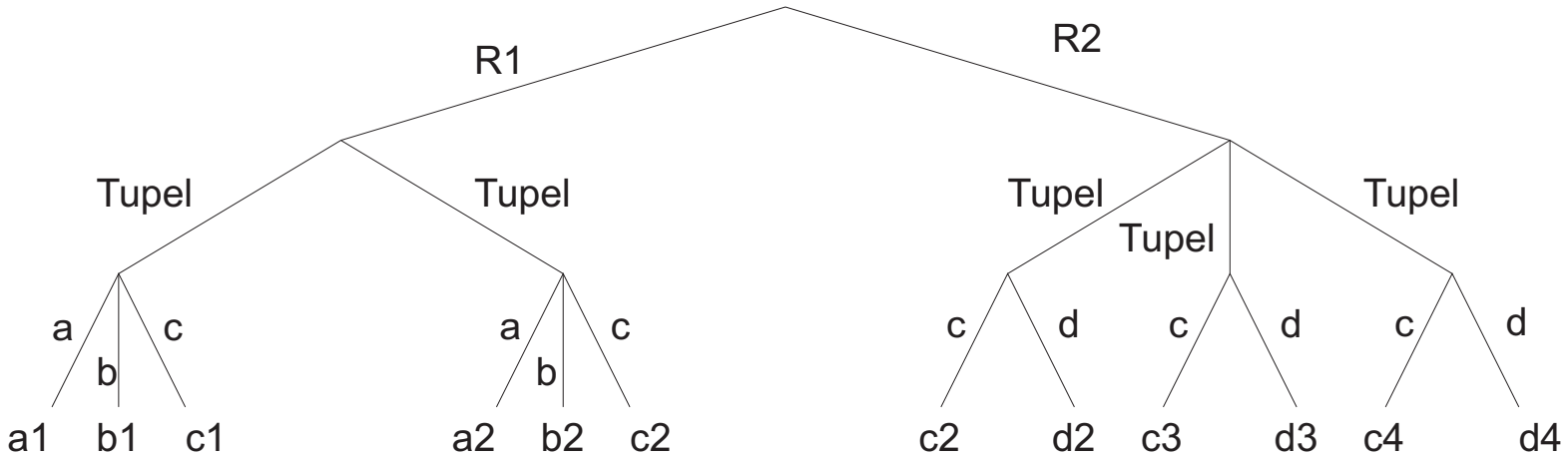
- Simple mapping of relational structures

R1

a	b	c
a1	b1	c1
a2	b2	c2

R2

c	d
c2	d2
c3	d3
c4	d4



XML - An Overview

- XML = Extensible Markup Language
  - Meta language based on SGML
- Meta language: allows definition of valid document types (DTD)
- Numerous complementary technologies and standards
  - Schema description: XML Schema (more powerful than DTD)
  - Query languages: XPath, XQuery
  - Transformation language: XSL(T)
  - ...

Building Blocks of XML Documents

- Elements: denoted by Tags

```
<tag>contents </tag>
```

- Alternatively also as empty elements:

```
<tag/>
```

- Tag provides information about the meaning of the contents

Example Elements

```
<address>
   <town>Ketchikan</town>
   <zip>99950</zip>
   <street>Paul Street</street>
</address>
```

```
<telephone>56-789012</telephone>
<fax/>
```

XML Document structure

- XML declaration, DTD (optional), root element (with nested contents)
- XML declaration:
  - `version`: `1.0`
  - `encoding`: font encoding used in file, e.g. `utf-8`, `utf-16`
  - `standalone`: `yes` no external schema; `no`, external schema required

```
<?xml version="1.0" encoding="utf-16"
   standalone="yes" ?>
```

XML Documents

- Two criteria for XML documents
  - *Well formed* (necessary): document contains elements that conform to basic XML rules (e.g. nesting, closing tags after opening tags, etc.)
  - *Valid* (optional): conform to a specific DTD
    - DTD contained in the document, or
    - DTD externally linked by a reference
- XML Instance = document conforming to a given DTD

XML Example document

```
<?xml version="1.0" ?>
<!DOCTYPE household SYSTEM "household.dtd" >
<household name=”ALASKA1966”>
    <address>
        <town>Ketchikan</town>
        <zip>99950</zip>
        <street>Paul Street</street>
        <no>6a</no>
    </address>
    <person>
        <lastname>Smith</lastname>
        <firstname>Lisa</firstname>
    </person>
</household>
```

Document Type Definition

- DTD (Document Type Definition): document structure (schema)
  - Formal grammar for a specific XML language
  - Names of allowed tags and their nesting
- Structure: sequence of element declarations

```
<! ELEMENT name ( content ) >
```

- *content*:
  - `EMPTY`: no further content
  - `ANY`: arbitrary text or elements from this DTD
  - `#PCDATA`: „parsed character data“, i.e. element text without further structure
  - regular expression for nested element type

DTD: Regular Expressions

- Sequence: `(E1, E2)`
- Alternative: `(E1 | E2)`
- Repetition:
  - `E+`: 1 . . . *n* repetitions
  - `E*`: 0 . . . *n* repetitions
  - `E?`: 0 . . . 1 optional element

DTD Example

```
<!ELEMENT   household (address, person+) >
<!ELEMENT   address (zip, town, street, no>
<!ELEMENT   zip (#PCDATA)>
<!ELEMENT   town (#PCDATA)>
<!ELEMENT   street (#PCDATA)>
<!ELEMENT   no (#PCDATA)>
<!ELEMENT   person (firstname, lastname, email?)>
<!ATTLIST   person ssn CDATA #REQUIRED>
<!ELEMENT   firstname (#PCDATA)>
<!ELEMENT   lastname (#PCDATA)>
<!ELEMENT   email (#PCDATA)>
```

Attributes

- Attributes: local properties of elements, e.g.

```
<town state="IL">Chicago</town>
<tel no="123459876"/>
```

- Declaration

```
<! ATTLIST name type restrict default>
```

  - *type*: `CDATA` – „character data“ (arbitrary text), `ID` – unique value for identification, `IDREF` – reference to `ID`, `( value1 | value2 | ...)` – enumeration, ...
  - *restrict*: `#REQUIRED` – necessary attribute, `#IMPLIED` – optional attribute, `#FIXED` – set to default value

Element or Attribute?

- Alternative 1:

```
<lot>
   <lotnr>42-33-66</lotnr>
   <address>...</address>
</lot>
```

- Alternative 2:

```
<lot lotnr="42-33-66">
   <address>...</address>
</lot>
```

Element or Attribute? /2

	Element	Attribute
Identification	–	`ID` / `IDREF`
Quantifier	1 / ? / $*$ / $+$	`REQUIRED` / `IMPLIED`
Alternatives	$\surd$	–
Default values	–	$\surd$
Enumerations	–	$\surd$
Contents	complex	atomic
Fixed order	yes	no

Entities

- Declaration of re-usable text
- Notation

```
<! ENTITY name 'replacement'>
```

- Example

```
<! ENTITY unimd "University of Magdeburg">
```

- Usage

```
<description>
He studied at the &unimd; ...
</description>
```

- Similar: parameter entities used only in DTDs

XML Schema

- Schema definition for XML
- More powerful alternative to DTDs
- DTD shortcomings
  - Weak typing
  - Limited power of expression
  - Document-oriented
- Advantages of XML Schema
  - Not limited to documents (also XML DBS, data exchange)
  - More powerful means of expression
  - Data-oriented

XML Schema: Concepts

- Namespace (prefix for elements): `xs`
- Schema element: consists of sub-elements
- Element: `element`
- Complex type definitions: `complexType`
  $\rightsquigarrow$ Composition from other elements and attributes
- Simple type definition: `simpleType`
  $\rightsquigarrow$ Basic types and constraints on basic types
- Comments

Schema Definition

```
<xs:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xs:element name=”address” type="AddressType"/>

<xs:complexType name="AddressType">
  <xs:sequence>
    <xs:element name="street" type="xs:string" />
    <xs:element name="town" type="xs:string" />
    <xs:element name="zip" type="ZIPType" />
    ...
  </xs:sequence>
</xs:complexType>
...
</xs:schema>
```

Element Declaration

- Like DTD: define allowed elements and attributes
- *Typed* elements and attributes
- Notation:

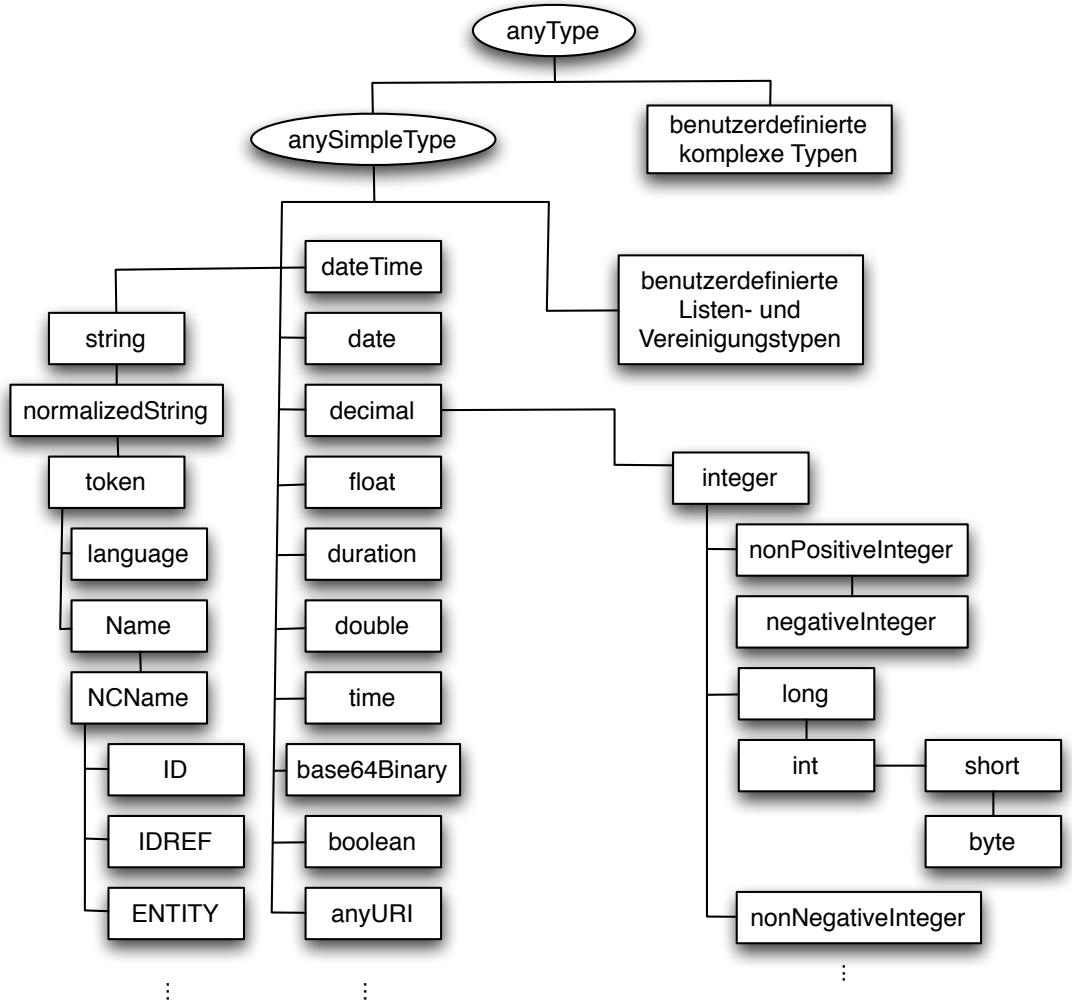
```
<xs:element name="element-name"
            type="type-name" />
```

- Further constraints for `xs:element`
  - `minOccurs`/`maxOccurs`: exact range for possible occurrences
  - `use`: `required`, `optional`
  - `default`: default value

Simple Types

- Builtin-Types: `string`, `integer`, `double`, `date`, `anyURI`, `...`
- Derived Types by restrictions on builtin-types
  - Range restrictions
  - Patterns
  - Enumeration of possible values
  - Lists and unions of other domains

Simple Types: Overview



Simple Types: Range Constraints

- Constraining to value range (1 . . . 100)

```
<xs:simpleType name="AmountType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="1" />
    <xs:maxInclusive value="100" />
  </xs:restriction>
</xs:simpleType>
```

Simple Types: Patterns

- Constraining a value domain by a pattern (regular expression)

```
<xs:simpleType name="ZIPType">
  <xs:restriction base="xs:string">
    <xs:pattern
      value="[0-9]{5}(-[0-9]{4})?" />
  </xs:restriction>
</xs:simpleType>
```

Complex Types

- „Type Constructor“ for elements with attributes and sub-elements
  - Based on simple types
  - Mixed Content (Sequence of sub-elements)
  - Selection, Grouping
  - . . .

Complex Types: Example

- Type definition

```
<xs:complexType name="AreaType">
 <xs:simpleContent>
  <xs:extension base="xs:decimal">
   <xs:attribute name="Unit"
    type="xs:string"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>
```

- Application in document instance

```
<area unit="square meters">400</area>
```

Complex Types: Example /2

```
<xs:complexType name="StreetType">
<xs:sequence>
 <xs:element name="Lot"
   minOccurs="1" maxOccurs="unbounded">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="no" type="xs:integer"/>
   <xs:element name="area" type="AreaType"/>
   <xs:element name="owner" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string"
   use="required"/>
 </xs:complexType>
 </xs:element>
</xs:sequence>
</xs:complexType>
```

Integrity Constraints

- Uniqueness: `unique`
- Key: `key`
- Reference to a key: `keyref`
- Assigned to elements or attributes using XPath (discussed later on)

```
<xs:element name=”Lot">
   <xs:complexType>
      <xs:sequence> ... </xs:sequence>
      <xs:attribute name=”lotno” type=”xs:integer”/>
   </xs:complexType>
   <xs:unique name=”LotNumber”>
      <xs:selector xpath=”lot/>
      <xs:field xpath=”@lotno/>
   </xs:unique>
</xs:element>
```

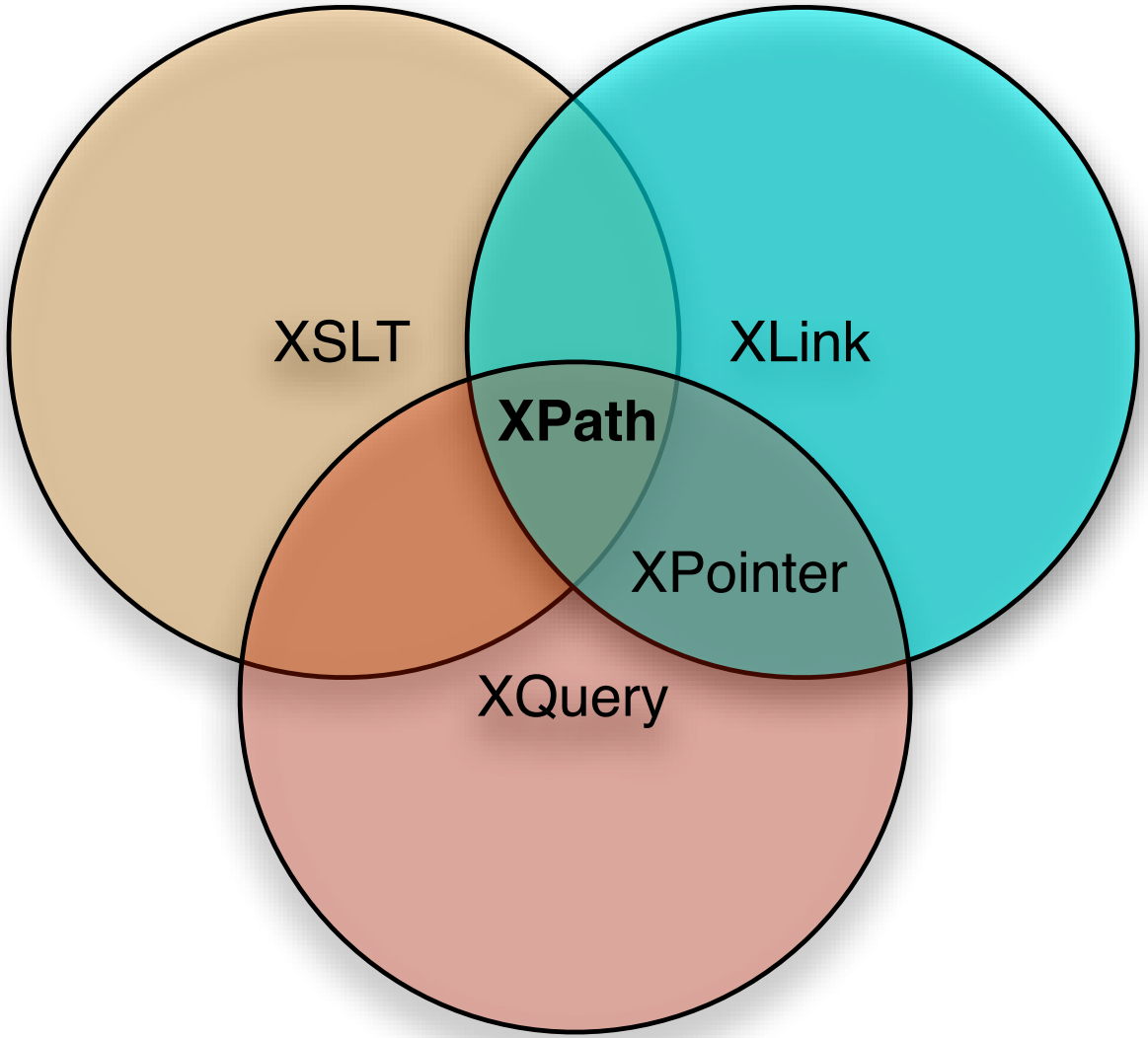
XML Query Languages

- Several query languages developed in late ’90s
- Now: set of co-operation standards
  - XQuery: fully-fledged XML query language
  - XSLT: XML transformation
  - XLink/XPointer: references between XML documents
  - XPath: basic functionality used in all above

XPath

- Language to address document fragments
- Part of XSLT, XQuery, ...
- Components:
  - Expressions to select document parts
  - Operations and functions
- No XML syntax
- No fully-fledged query language
  - Only selection of document fragments
  - No result construction, joins, grouping, etc.

Role of XPath



XPath: Data Model

- XML document as DOM tree
  - Nodes for XML elements
  - Edges with element labels
- Node types: root node, element nodes (references to sub-elements, attribute nodes, text nodes), comment nodes
- Data types: atomic values (strings, boolean, float), sequences of nodes

XPath: Path Expressions

- Consist of steps
- Steps separated by „/“
- Processed from left to right
- Each step yields sequence of nodes or value
- Absolute path starting from root node starts with „/“
- Relative path without preceding „/“
- Examples:

```
/book/title
/book/author/lastname
//author
```

XPath: Steps

- (Expanded) notation for each step:

```
axis::node-test[predicate]
```

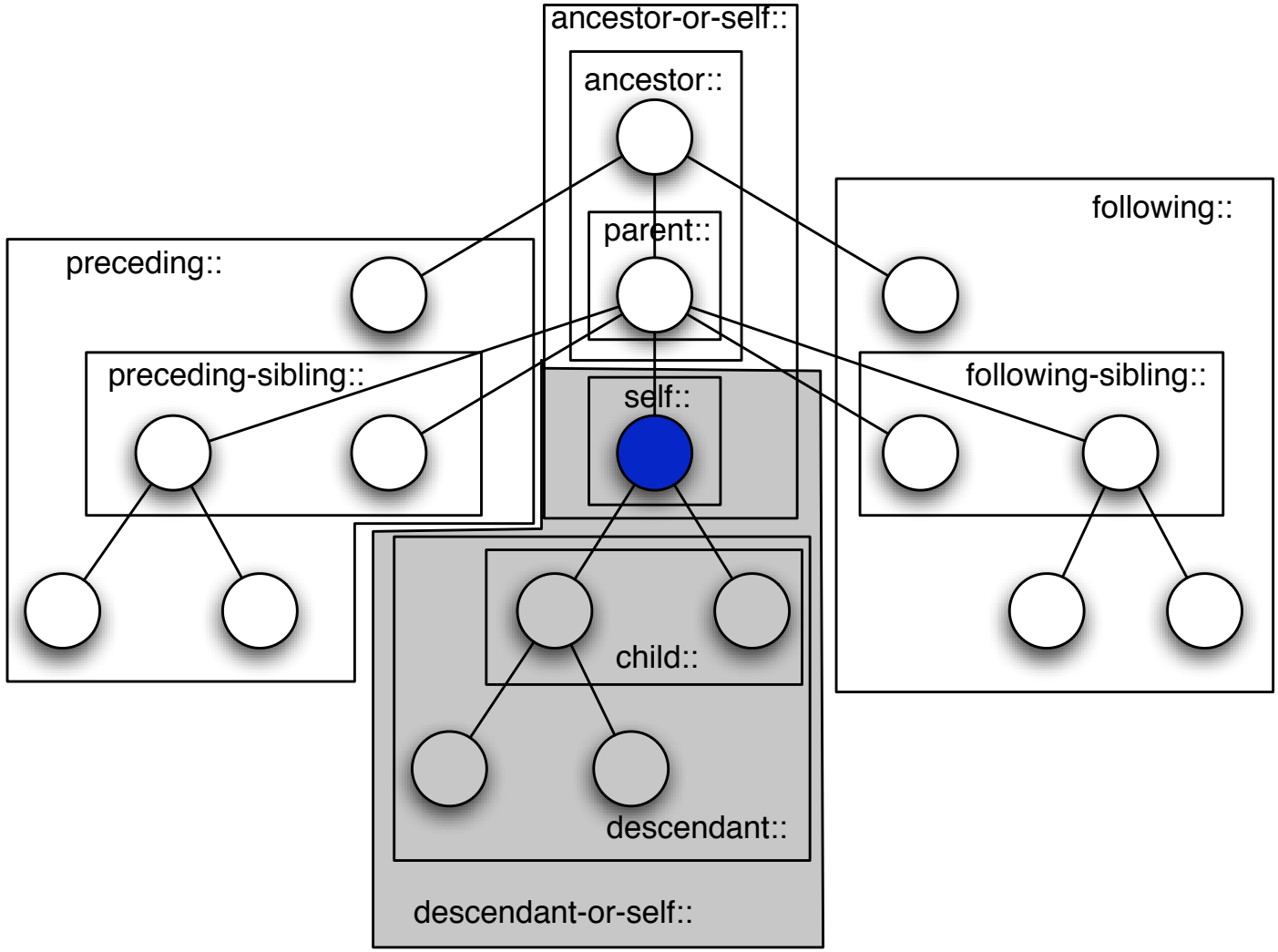
  - *axis*: relation between steps current (context) node to nodes to be selected
  - *node-test*: node type and name of nodes to be selected
  - *predicate*: logical predicate/condition to filter node set

- Abbreviations and default steps discussed later on

XPath: Axis Specifiers

- `self`: context node
- `child`: all direct sub-elements
- `descendant`: all direct or indirect sub-elements
- `descendant-or-self`: `descendant` + `self` nodes
- `parent`: parent nodes
- `ancestor`: all nodes on path from context node to root node
- `preceding-sibling`: previous (in document order) children of parent of context node
- `following-sibling`: subsequent children of parent node

XPath: Axis Specifiers /2



XPath: Node Tests

- Restriction to certain node types
  - `node()`: true for all nodes
  - `text()`: true for text nodes
  - `*`: true for element nodes
  - *qname*: true if name of element node is *qname*
- Examples:
  - `descendant::*` all sub-element nodes of context nodes
  - `child::author` all sub-elements named `author`
  - `attribute::id` attribute `id` of context node
  - `parent::*` parent node

XML - Example document

```
<?xml version="1.0" ?>
<books>
  <book isbn="1-55860-622-X" >
    <title>Data on the Web</title>
    <price>42.90</price>
    <publisher><name>Morgan Kaufmann</name>
      <address>San Francisco</address>
    </publisher>
    <author>
      <firstname>Serge</firstname>
      <lastname>Abiteboul</lastname>
    </author>
    <author>
      <firstname>Peter</firstname>
      <lastname>Buneman</lastname>
    </author>
  </book>
  <book isbn="3-8266-0618-1" > ...</book>
</books>
```

XPath: Examples

```
fn:doc("books.xml")
   /child::books/child::book
```

```
fn:doc("books.xml")/child::books
   /child::book[attribute::isbn='1-2345']
```

```
fn:doc("books.xml")
   /descendant-or-self::book
      [child::author/child::lastname='Sattler']
   /child::title
```

XPath: Abbreviations

- Default axis and short syntax for node tests
  - `child::` default axis specifier
  - `@` for `attribute::`
  - `.` for `self::node()`
  - `..` for `parent::node()`
  - `//` for `/descendant-or-self::node()/`
  - `[`n`]` denotes n-th element from sequence of nodes

Selection Predicates and Functions

- Predicates and complex conditions
  - Comparison operators: `<`, `<=`, etc.
  - Logical operators: `and`, `or`
  - Arithmetic operators: `+`, `*`, `-`, `div`
- Functions
  - `fn:contains (`*`str, substr`*`)`: test for string
  - `fn:last ()`: position of last element in current context
  - `fn:position ()`: position of current element

Conditions in Path Steps

- Context position:
  `child::author[1]`
- boolean predicate:
  `child::book[@isbn = '1234']`
- Expressions over sequences:
  `child::author[fn:position() = { 1, fn:last() }]`
- Combination of predicates:
  `child:author[1][child::name='Heuer']`

Test for Existence

- Test for existence of attributes

```
//buch[@isbn]
```

- Test for existence of elements

```
//book/publisher[address]/name
```

XPath: Examples /2

```
fn:doc("books.xml")
   /books/book[@isbn='1-55860-622-X']
```

```
fn:doc("books.xml")
   //book/author[1]/lastname
```

```
fn:doc("books.xml")
   //book[author/lastname='Sattler']/title
```

XQuery

- Functional Query Language for XML
- Combination of different possible expressions
  - XPath expressions
  - Element construction
  - Function calls (standard, user-defined)
  - Constrained and quantified expressions
  - Data type-specific operations
  - *FLWOR-expressions*

XQuery: Data Model

- Result: ordered sequence of nodes or atomic values
  - No nesting
  - Duplicates allowed
- Special cases:
  - Sequence with one element $\equiv$ element
  - Empty sequence $\equiv$ nothing

```
(1, (), (<e/>, 2), 3) ≡ (1, <e/>, 2, 3)
```

XQuery: Expressions

- Variable reference: `$name := "XQuery"`
- Value construction: `xs:double(NaN), xs:float(3.1415)`
- Function calls: `fn:compare("Goethe", "Göthe")`
- Comments: `(: comment :)`

XQuery: Input Function

- `fn:doc()`: URI as parameter
  - yields document node for referenced document
- `fn:collection()`: URI as parameter
  - Yields sequence of nodes, e.g. from database

XQuery: FLWOR expressions

- Similar to SFW-block in SQL
- Notation

```
(for-expression | let-expression )+
[ where expression ]
[ order by expression ]
return expression
```

mit

```
for-expression ::= for $var in expression
let-expression ::= let $var := expression
```

XQuery: Variables

- Binding results of expression evaluation to variables
- Notation: `$name`
- Characteristics:
  - Type derived from result of evaluation
  - No side-effects (modifications) allowed
  - Only valid for current context (query, expression)

XQuery: `let`-clause

- Set of values/nodes is bound to variable

```
let $var := expression
```

- Example:

```
let $b := fn:doc("books.xml")//book
return $b
```

- Evaluation:
  1. All `book` nodes from document `books.xml`
  2. set of values assigned to `$b`
  3. Output of set `$b`

XQuery: `for`-clause

- All elements of result set are step-wise bound to variable

```
for $var in expression
```

- Example:

```
for $b in fn:doc("books.xml")//book
return $b
```

- Evaluation:
  1. Binding for each element to `$b`
  2. Further clauses, e.g. (**`where`**, **`return`**, ...) are processed for each element, i.e. **`return`** executed for each book

XQuery: `for` vs. `let`

- **`for`**-clause

```
for $i in (1, 2, 3)
return <tuple><i>{ $i } </i></tuple>
```

```
<tuple><i>1</i></tuple>
<tuple><i>2</i></tuple>
<tuple><i>3</i></tuple>
```

- **`let`**-clause

```
let $i := (1, 2, 3)
return <tuple><i>{ $i } </i></tuple>
```

```
<tuple><i>1 2 3</i></tuple>
```

XQuery: Element Construction

- Creation of XML fragments
  - Literal XML
  - Creation of elements and attributes
- Evaluation of XQuery-expressions in XML in `{ ... }`
- Example:

```
<authors> {
for $b in fn:doc("books.xml")//book
    for $a in $b/author
    return
     <name> {$a/lastname/text()} </name>
} </authors>
```

XQuery: Element Construction /2

- Explicit element construction using **`Element`**
  - Element name can be derived from expression

```
element price {
   attribute currency { "Euro"}, "59.00"}
```

- Result:

```
<price currency"Euro">59.00</price>
```

XQuery: **where**-clause

- Filter on result set
- Condition with expressions
  - Arithmetic, logical and comparison expressions
  - Function calls
  - Path expressions
- Example:

```
for $b in fn:doc("books.xml")//book
where $b/price < 50
return
   <book> {
      <isbn> $b/@isbn </isbn>,
      $b/title
   } </book>
```

XQuery: Join

- Using variables from different **FOR**-clauses

```
for $b in fn:doc("books.xml")//book,
    $r in fn:doc("reviews.xml")//reviews
where $b/title = $r/booktitle
return <bookreview> { $b/title, $r/score }
    </bookreview>
```

XQuery: Aggregate Functions

- Aggregate functions with sequence of parameters

```
let $b := fn:doc("books.xml")//book
return <costs>
   { fn:sum($b/price) }
   </costs>
```

```
let $b := fn:doc("books.xml")//book
let $avg := fn:avg($b/price)
return $b[price > $avg]
```

Conditional Expressions

- Syntax:

```
if (expr1) then expr2 else expr3
```

- Example:

```
for $b in fn:doc("books.xml")//book
return <book> { $b/title }
   { for $a at $i in $b/author
   where $i <= 2
   return <author>{ string($a/lastname), ",",
      string($a/firstname)}</author> }
   { if (count($b/author) > 2)
      then <author>et al.</author> else () }
   </buch>
```

Quantifiers

- Quantifiers: test, if one or all elements of a sequence fulfill a condition

```
some | every $var in sequence
    satisfies condition
```

- Existential quantification: **`true`**, if at least one element yields **`true`** for condition (**`false`** for empty sequence)
- Universal quantification: **`true`**, all elements yield **`true`** for condition (**`true`** for empty sequence)

Quantifiers /2

```
for $b in fn:doc("books.xml")//book
where some $a in $b/author/lastname
   satisfies $a = "Ullman"
return $b
```

Functions

- Aggregate functions
- Pre-defined numeric functions: `fn:abs(`v`)`, `fn:ceiling(`v`)`, `fn:floor(`v`)`, `fn:round(`v`)`, ...
- String functions : `fn:compare(`s_1`, `s_2`)`, `fn:concat(`s_1`, `s_2`)`, `fn:string-length(`s`)`, `fn:upper-case(`s`)`, `fn:substring(`s`, `b`, `e`)`, `fn:string-join(`s`, `t`)`, ...
- Regular expressions: `fn:matches(`s`, `p`)`
- Functions for date and time: `fn:current-date()`, `fn:get-day-from-date(`d`)`, ...

Functions: Examples

```
fn:ceiling(42.9) (: 43 :)

fn:substring("XQuery", 2, 2) (: "Qu":)

fn:string-join(("abc", "def"), "-")
   (: "abc-def":)

fn:matches("XQuery", "^X.*y$”) (: true :)

fn:get-day-from-date("2005-06-30") (: 30 :)
```

User-defined Functions

- Declared with signature

```
declare function name (param-list)
    as sequence-type
```

  - Default type `item*`
- Function body
  - XQuery-expression
  - **`external`**: externally implemented (using programming language)