Chapter 2
Database Concepts and Extensions

- The notion of "semistructured data (SSD)" has mainly been coined by the "TSIMMIS" project (The Stanford-IBM Manager of Multiple Information Sources, 1995-2000; persons: J. Ullman, H. Garcia-Molina, J. Widom, Y. Papakonstantinou).
- The problem has already been investigated before in several areas and projects.

Why Semistructured Data?

... mainly two requirements:

1. *data integration* from different sources
   (late 1980s/early 1990s):
   - increasing networks
   - combination of contents of several databases
     * multi-database-systems
     * federated database systems
     * different schemata
     * mostly only different relational schemata,
       · partially also under the aspect of integration of metadata into the DML – this aspect is originally independent from semistructured data.
     * sometimes different data models ("legacy"-databases according to earlier data models)
     * since mid-90s increasingly data from the Web

WHY SEMISTRUCTURED DATA?

2. storage of "unregular" data:
   no fixed/homogeneous/known schema, many null values (e.g. biochemistry)
   - data exchange (B2B); standard formats e.g. for suppliers in automobile industry
   - partly also full-text portions,
   - management of document content
     * coarsely structured
       SGML (special form: HTML)
   - annotated binaries (pictures, films, etc.)
   - mixed forms between databases and documents
     * collections: (tax) laws, partially in SGML
     * health care and clinical information systems

THE EVOLUTION TOWARDS XML

The evolution in the area of semistructured data and XML combined concepts, experiences and developments from many previous approaches:

- network data model, hierarchical model ("legacy"-databases),
- relational databases,
- object-oriented databases,
- distributed and federated databases,
- data integration (purely relational environments, or mixed ones),
- document management.

Different lines of evolution have been brought together with XML & friends:

⇒ (nearly) nothing new, but a perfect combination!

Textbook on "Databases" in general (but without document management and XML):

- R.Elmasri, S.Navathe: "Foundations of databases"/"Grundlagen von Datenbanksystemen". Pearson Studium, 3rd edition, 2002.

2.1 Early Databases: the Network Data Model

Situation 1960: first primitive "high-level" programming languages for "calculations"

- FORTRAN 1957: "formula translator"
- COBOL 1959: "common business-oriented language"

Goal: somehow store and organize lots of data:

- first development in the database system *IDS (Integrated Data Store)* at *General Electric* (Bachman & Williams, 1964; Turing-Award 1973)
- specification of the "Conference on Data Systems Languages Data Base Task Group (CODASYL)", 1971.
- products: e.g. VAX-DBMS (Digital Equipment)

Network Data Model

- data is stored in *data records*,
- classified by *data record types*, with attributes (name and datatype to be specified).

Country				
Name	Code	Population	Area	...

City		
Name	Population	...

- Sample data records:

"Germany"	"D"	83536115	356910	...

"Berlin"	"3472009"	...

- So far, the same as the mapping of entity types in the relational model.
- difference: the *organization* of the records (and their relationships) in the database ...

Relationships: Set Types

Relationships are represented as sets: “all B that are in a given relationship with a certain A”
(E.g. all cities in a given country)

Definition of set types:

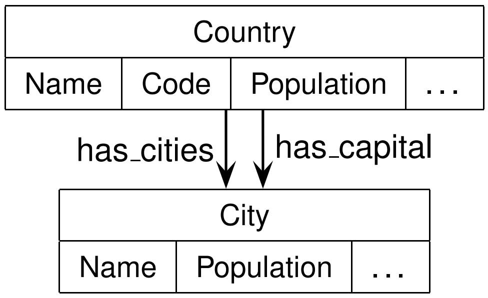
- name of the set type
- owner record type (“owner”; “where the relationship starts from”)
- member record type (“member”)

A set instance then represents the relationship for exact one owner of type “A”. Each instance of a set consists of

- a data record of the owner record type
- an *ordered* set of data records of the member record type
- comparison with XML: parent-children relationship, ordered children, but all of the same type
- intuitition: not as a set, but as a wire that is fixed at the owner and then pulled through all members.

Relationships: Set Types

Graphical representation:
“Bachman Diagram”



has_ cities:

Germany	D	83536115	...
Berlin	3472009	...	
Hamburg	1705872	...	
Frankfurt	652412	...	
⋮			

has_ capital:

Germany	D	83536115	...
Berlin	3472009	...	

- similar sets for France//Paris/Lyon/Marseille/... and France//Paris
- a member record can belong to only one instance of a set of *each* set type (thus, only 1:N-relationships can be modeled directly)
- n:m relationships: later

Entry Points

- system-owned instances of a set serve as entry points
  (e.g. an instance of a set "countries" whose members are the country-data records)

Access Operations

Access to (and navigation through) the database only via sets.

Actually, this is again an *abstract datatype*:

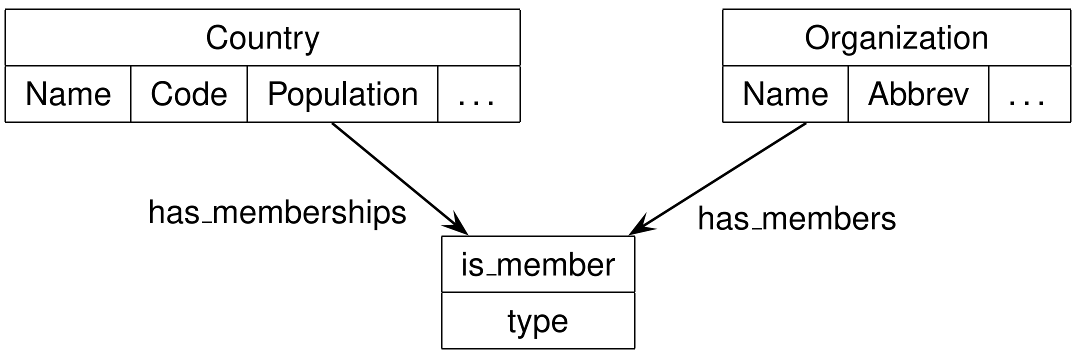
- access to the attribute values of a record,
- an *iterator* (first, next) for traversing the relationships ,
- a selector "find_owner" for inverse relationships.

⇒ the user does not explicitly work with pointers or identifiers, but already uses the semantic notions of the data model.

N:M-Relationships

Cannot be represented by a single set type (analogously for attributed relationships).

- split into a 1:M and an inverse N:1-relationship
  Problem: consistency maintenance (symmetry!)
- introduce an auxiliary data record type that represents the relationship, and two set types:



- later, there is a mapping from the ER model (1976) to the network model.

ORGANISATION OF THE SET TYPES

Each data record contains reference entries for each set type where it belongs to (either as owner or as member):

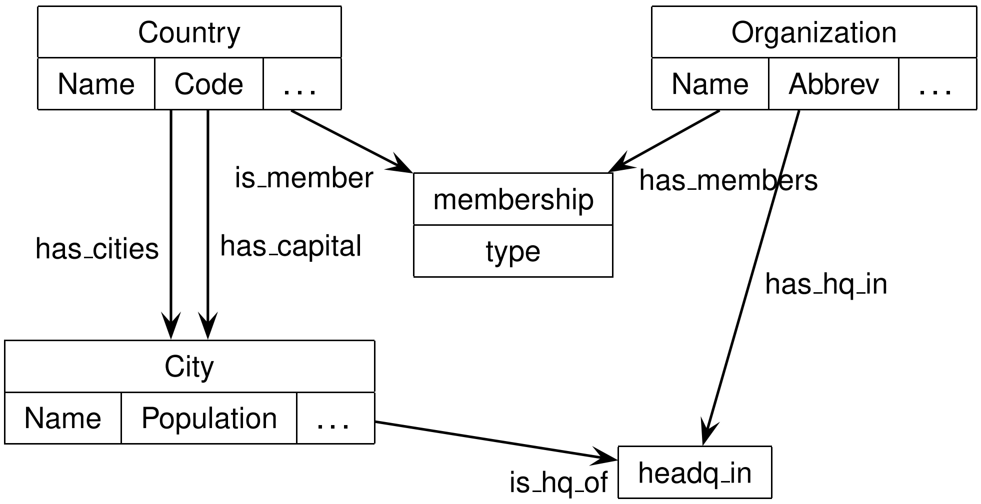
- as owner: a "first"-reference, labeled with the name of the set type, pointing to the first member record
- as member:
  - a "next"-reference, labeled with the name of the set type, pointing to the next member record
  - additionally a labeled backwards pointer to the owner of the set instance
  - a labeled null pointer if there exists no first/next element.

Exercise 2.1

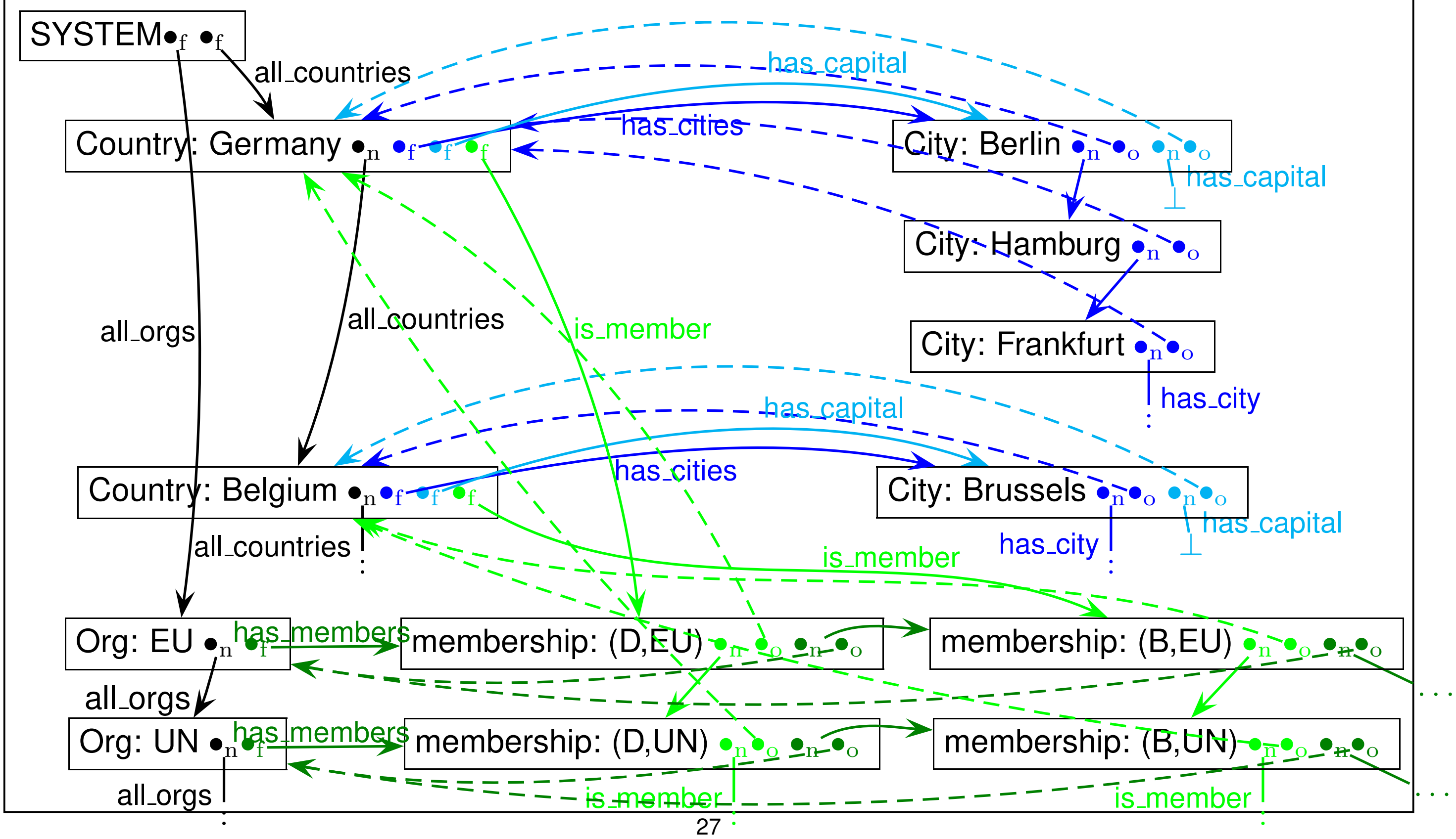
a) Visualize the model by drawing some country, city and organization data records.

b) Consider the "has_headq"-relationship that describes that organizations have their headquarters in a city. □

SOLUTION: NETWORK SCHEMA DIAGRAM



SOLUTION: INSTANCE LEVEL



DATA DEFINITION LANGUAGE: EXAMPLE

RECORD NAME IS country
  DUPLICATES ARE NOT ALLOWED FOR Code
    Name TYPE IS CHARACTER 20
    Code TYPE IS CHARACTER 4
    Population TYPE IS NUMERIC INTEGER
    Area TYPE IS NUMERIC INTEGER

RECORD NAME IS city
    Name TYPE IS CHARACTER 25
    Population TYPE IS NUMERIC INTEGER

SET NAME IS all_countries
  OWNER IS SYSTEM
  MEMBER IS country

SET NAME IS has_cities
  OWNER IS country
  MEMBER IS city

Query and Data Manipulation Language

- record-at-a-time DML
- based on *iterators* (common design pattern/interface, e.g. in Java!) over sets
  - commands for navigation, access and data manipulation
  - embedded into a host language (COBOL, PL/I, later ... Pascal, C)
- "Current of" (cf. PL/SQL: "cursor") that points to an instance of a record/set type in the DB
  - current of each record type
  - current of each set type (pointing on either the owner or one of the member records)
  - current of run unit (CRU): the record most recently accessed – any record type
- UWA (User Work Area) in the programming language runtime environment
  - one variable for each record type (auto-defined from the schema)
  - current of ... can be "fetched" into the corresponding UWA record

Retrieval and Navigation Commands

Query answering consists of stepwise navigation, carefully tracing currency indicators, and fetching tuples to the UWA:

- Retrieval: move the CRU into the corresponding UWA record,
- Navigation: navigate by using iterators and currency indicators to specific records and set owners/members.

Search for a Record of a Record Type

- FIND ANY <data record type> [USING <UWA.field.list>]
- FIND DUPLICATE <data record type> [USING <UWA.field.list>]
- tests/loops can be programmed by `IF/WHILE DBSTATUS=0` // 0: successfully found
- FIND sets all current of record/set type in which the record participates to that record. Can be avoided with RETAINING clause.

```
UWA.city.name = "Santiago";
FIND ANY city USING name;
// sets also current of city indicator
while DBSTATUS=0 do begin
  GET city     // fetches data record into UWA.city
  if UWA.city.population > 1.000.000 then writeln (UWA.city.name|UWA.city.population);
  FIND DUPLICATE city USING name;
end;
```

- How to print out the city name and the country where it is located? Needs the "owner" of the city wrt. "has_cities".

Search for a Record in a Set Type

- FIND (FIRST | NEXT | PRIOR | LAST) WITHIN <set type> [USING <UWA.field.list>]
- FIND OWNER WITHIN <set type>
- starts always from the current of this set (which is implicitly set when the CRU points to a suitable record type)

```
UWA.country.name = "Belgium";
FIND ANY country USING name;
FIND FIRST city WITHIN has_capital
GET city     // fetches data record (Brussels) into UWA.city
writeln (UWA.city.name);
FIND OWNER WITHIN in_province
GET province     // fetches data record (Brabant) into UWA.province
writeln (UWA.province.name);
```

- Joins are only possible via navigation and loops in the host language.

Exercise 2.2
Write a program that outputs all organizations that have their headquarter in the capital of one of their member countries. Compare with the equivalent SQL query against Mondial. □

UPDATES

Updates on Data Records

STORE, ERASE, MODIFY (of the current data record)

Updates on Sets

CONNECT, DISCONNECT, RECONNECT (for the current data record wrt. a set)

HIERARCHICAL DATA MODEL

- In general very similar: parent-child-relationships define a tree structure; additionally, “virtual” parent-child-relationships.
- Systems: IMS (IBM & Rockwell International, 1969 for NASA Apollo), Adabas (Software AG, 1969), etc ...

SOLUTION

```
// not tested
find any organization // sets current of has_headq, current of has_members
while ok do
{ get organization // current organization into UWA
  find first headq_in within has_headq_in // auxiliary record hq(org,cty)
  find owner within is_headq_of  // is a city
  find owner within has_capital // is a country
  if ok then // city is a capital
  { get country // UWA.country now holds this country
    found = 0;
    find first membership within has_members
        // starts from the organization
        // points to an auxiliary membership record m(org,c)
    while ok & not found do
    { find owner within is_member using code // UWA.country.code
       // check if the owner country is the same as in UWA
      if ok then { println(UWA.organization.name); found = 1;}
      find next membership within has_members
    }
  }
  find duplicate organization // next organization
}
```

The same in SQL

```
SELECT name
FROM organization org
WHERE (city,country) IN (SELECT capital, code
                         FROM country
                         WHERE code IN (SELECT country
                                        FROM is_member
                                        WHERE organization = org.abbreviation))
SELECT organization.name
FROM organization, is_member, country
WHERE organization.abbreviation = is_member.organization
  AND is_member.country = country.code
  AND organization.city = country.capital
  AND organization.country = country.code

SELECT organization.name
FROM   organization, country
WHERE  organization.city = country.capital
  AND  organization.country = country.code
  AND (abbreviation, code) IN (SELECT organization, country
                               FROM is_member)
```


Conclusion

- importance decreased rapidly since SQL came up (1979), in the meantime it is only present in "legacy systems".
- no underlying theory (required as a base for normalization and optimization)
- only procedural, (data-model-level) navigation- and record-oriented query language, non-declarative, needs to be embedded into a host language (COBOL, PL/I, Pascal, C).
- not possible to state ad-hoc queries.
  Error-prone due to behavior of currency indicators.
- nevertheless, the idea of navigation and parent-child-relationships between data records is elegant (no problems with referential integrity).
  These concepts came up again in later approaches ... with high-level navigation!
- graph data model, "node + edge-labeled"
- expecially, ordered "child data records" are used again in XML. Then, there is
  - the DOM as an abstract datatype (stepwise, record-oriented),
  - XPath/XQuery as a *declarative*, set-oriented high-level language.

2.2 Object-Oriented Databases

Mid-80s: Object-orientation

- object-oriented design and modeling (UML)
- object-oriented programming (C++)

Application programs are developed and programmed in an object-oriented way.

- "impedance mismatch" between tuple-based SQL databases and the object-oriented data structures of the programming languages.

Goals:

- make objects of the application programs persistent
- bring object-orientation into the DBMS
  - class hierarchy and inheritance, polymorphism
  - implementation and encapsulation of behavior

Further Influences

- Networks: Internet and Intranets
- Data exchange and interoperability
- CORBA (1989) "Common Object Request Broker Architecture" (standardized by OMG – Object management Group; predecessor of Web Services):
  - central ORB bus where services can connect
  - service registry (predecessor of WSDL and UDDI ideas)
  - description of service interfaces in object-oriented style
    (IDL - interface description language, similar to C++ declarations)
  - exchanging objects between services

⇒ requires a format for exchanging data (DB: between databases)

In this lecture, OODBS are only discussed shortly to sketch the central ideas.
An extended lecture can be found in "Information Systems", available at
`http://user.informatik.uni-goettingen.de/~may/Lectures`.

Lifetime of Objects

- Object-oriented programming language: Objects are created during runtime of an application program, and they are destroyed when the program terminates.

Objects in OO Database Systems

- persistent: objects that are created by an activity, and then they are stored in the database system and survive also the termination of the activity that created it (until they are explicitly destroyed by another activity)
- transient: objects that are only needed temporarily for executing an activity. They exist only as long as the application is actually active, and they are only managed by the runtime environment of the programming language.

Lifetime of Objects

- Relational DBMS: all SQL types have only persistent instances that are stored in the DBMS. All non-SQL types (i.e., types of the host language) have only transient instances, these are destroyed with the termination of the application-program (= when the host language is left).

  Persistent objects can only be manipulated/used by SQL, while transient objects can only be manipulated/used by the host language.

  ⇒ "impedance mismatch".
- ODBMS: object types of the DBMS and of the application coincide. They can have parallel and transient instances at the same time.

  For persistent and transient objects the same programming language and the same operations are used.

- comparison with XML: XML nodes can also be processed uniformly in the runtime environment and stored in a database. The DOM-API can be used in both cases.

Object-Oriented Data Model

- describes only the *(database) state* (*attributes, relationships, class membership and class hierarchy*), not the behavior,
- representation of the current state of the application-domain,
- corresponding conceptual modeling language: UML (see Software Engineering)
- more expressive than the relational model/ER-model

- (behavior of objects is integrated into the data manipulation language)

OO-DBMS

Standardization activities similar to the standardization of relational databases:

Success of the relational database systems:

- not only by the simple, high-level data model,
- but also due to the standardization: SQL (at least after some time)
  - portability
  - interoperability

ODMG: Object Database Management Group

- founded 1991
- Architecture of OODBMS, DDL, query language (OQL), data formats
- ODMG-1.0 standard (1993)
- ODMG-2.0 standard (1997)
- ODMG-3.0 standard (2000); incremental changes

Literature: Cattell et al; Object Database Management (ODMG, 1993/1997/2001)

ODMG: Object Database Management Group

- Voting members: organizations/companies, who commercially work at an ODBMS, among others JavaSoft, Windward Solutions, Lucent Technologies, Unidata, GemStone, ObjectDesign, Versant, ...
  Reviewer members: Organizations who have a material interest in the work of ODMG.
- not the goal to define identical products, but to obtain source code portability (cf. Java, SQL, later also XML).
- enough freedom to define own properties and targets of products:
  - performance, optimization, (price)
  - support of certain programming languages,
  - functionality dedicated to special application areas (multimedia, CAD, ...), predefined types
  - integrated programming environments, design tools ...

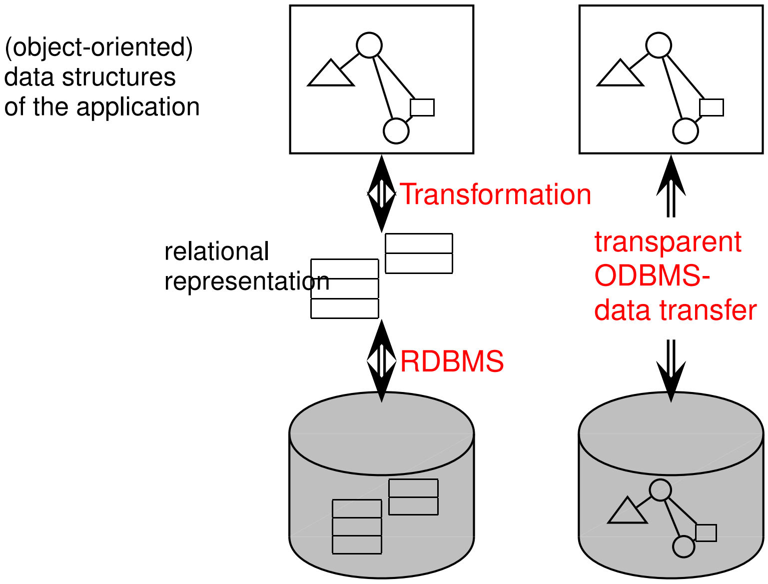
Architecture of ODBMS

- Different from "classical" relational DBMS:
  SQL: high-level language for data manipulation,
  applications are then written in other programming languages (cf. embedded approaches).
- ODBMS/ODM: transparent integration of DBMS functionality (persistence, multiuser, recovery) into application programming language (cf. Persistent Java).
  The objects of the application are simply stored in the database.
- no separate DML necessary. The application-level programming language *is* the DML.
- There is also a *set-oriented, declarative query language*
  (the impedance mismatch between variable-orientation and set-orientation remains):
  OQL
- no transformation between the (logical) database representation and the representation in the programming language (cf. datatype conversion in JDBC).

Architectures

ODMG is concerned with two types of products:

- Object Database Management Systems (ODBMSs) store the objects directly,
- Object-to-Database Mappings (ODMs) convert objects and store them in a relational (or any other) representation.



Remark:
There are similar approaches for XML databases.

ODMG-Standard

A standard that consists of several languages for implementation-level specification of object-oriented systems.

Components of the ODMG Standard

- Object specification languages/data model
  - Object Definition Language (ODL)
  - Object Interchange Format (OIF)
- Object Query Language (OQL) – based on SQL
- C++/Smalltalk/Java Language Binding
  specifies how to work with persistent objects in the target languages.

2.2.1 ODL: Object Definition Language

- Data definition language for object types:
- not a programming language, but only a language for definition of object specifications,
- characterizes object types (class hierarchy, properties and relationships)
- extends IDL (Interface Definition Language) from the OMG/CORBA (1989/1990) standard (which is in course closely related to the declaration commands in Java)

DATA TYPES: LITERALS

Literals are only values, they have no *object identity*.

Atomic literals

- long, short, unsigned long/short, float, boolean, char, string,
- enumeration {...} (“type generator”)
  Z.B. enum Weekday {Sunday, Monday, . . . , Saturday}

Structured Literals

- predefined types: date, interval, time, timestamp
  (additionally to *actual object types* Date, Interval, Time, Timestamp)
- user-defined structural types, e.g. address or

```
struct geoCoord { real   longitude;
                  real   latitude; }
```

Collection literals

- set<t>, bag<t>, list<t>, array<t>, dictionary<t>
  (additionally to the *actual collection types* Set, Bag, List, Array, Dictionary)

CLASSES

... are used to define and categorize complex object types.

Classes define the *signature* of their instances (the implementation does not belong to the object model):

```
class <name> { <attribute-defs>;
               <relationship-defs>;
               <operation-defs>;
               }
```

<attribute-def> ::= attribute <domain-type> <attribute-name>

```
class City { attribute string name;       % attributes ...
             attribute number population;
             attribute geoCoord coordinates;
             relationship Country in_country;    % ... and relationships
             }
```


RELATIONSHIPS

- relationships are defined in course of the definition of classes.
- in UML and ODMG, only *binary* relationships are allowed.
- *bidirectional* and *inverse* relationships can be specified. Inverse relationships exist in UML, and later again in the Semantic Web languages (OWL).
- one-to-one / one-to-many / many-to-many-relationships.

```
class <name> {
    <attribute-defs>;
    <relationship-defs>;
    <operation-defs>; }
```

```
<relationship-def>::= relationship <target_of_path> <relationship-name>
                      inverse <domain-type> :: <relationship-name'>
<target_of_path>::=   <domain-type> |
                      <collection type> <<domain-type>>
```

- <collection type> for -to-many-relationships

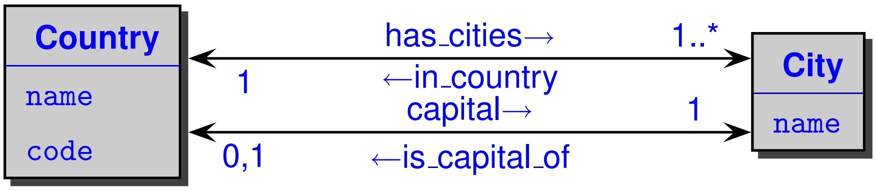
RELATIONSHIPS

class <name> { <attribute-defs>;
<relationship-defs>;
<operation-defs>; }

<relationship-def>::= relationship <target_of_path> <relationship-name>
inverse <domain-type> :: <relationship-name>
<target_of_path>::= <domain-type> |
<collection type> <<domain-type>>

class Country { attribute string name;
relationship City capital inverse City::is_capital_of;
relationship set<City> has_cities inverse City::in_country; }

class City { attribute string name;}



RELATIONSHIPS

- the instance level can be represented as a graph:
  - nodes: objects; nodes have labels (names of the object types) and an ID
  - edges: relationships; edges have labels (names of relationships)
- ODBMS is responsible for maintaining referential integrity:
  If an object is deleted, all relationships with/to it must also be deleted.
- relationships define *access paths*, e.g. Germany.capital for navigation through the graph.
- graph-data model, "node + edge-labeled"
- The set<...> is very similar to the set-oriented representation of set-valued relationships from the network data model (→ handled by iterators)
- the query language OQL solves this problem SQL-like in a declarative way (see later).

Exercise 2.3
Visualize an excerpt of the Mondial database as an object graph. □

2.2.2 Object Interchange Format (OIF)

- dump the database state to one or more files (cf. *export* in ORACLE)
- specification language for persistent objects and their states
- OIF file contains for each object its type, its attributes and values, and its relationships to other objects;
- the database schema (class definitions *and* class hierarchy) is *not* represented in OIF!

OBJECT INTERCHANGE FORMAT (OIF)

- Simplest form: only the class membership

  <object> <class> {}
  Germany Country {}
  Berlin City {}

- attribute values are enumerated in braces:

  Germany Country {name "Germany", area 356910, ...}

- structured attributes: nested brace structures

```
struct geoCoord { real longitude;
                  real latitude; }

class City { attribute string name;
             attribute geoCoord coordinates;
             relationship Country in_country; }

Berlin City {name "Berlin", in_country Germany,
             coordinates {longitude 13.3, latitude 52.45} }
```

OBJECT INTERCHANGE FORMAT (OIF)

- Collections, set-valued relationships:

  class Country {
      attribute string name;
      relationship set<City> has_cities;}

  Germany Country
    {name "Germany", capital Berlin,
     has_cities {Berlin, Frankfurt, Freiburg, ...} }

- cyclic references: no problem.
- attributed relationships (e.g. border) cannot be represented directly

⇒ OIF is already a self-describing data format!

2.2.3 OQL (overview)

- Query language of the ODMG standards (Object Query Language)
- similar to SQL:
  SELECT - FROM - WHERE - clause, extended by complex objects, object-identity, path expressions, polymorphism, operation calls and late binding.
- but: functional language (like SQL), fully orthogonal (in SQL not completely)
- no explicit UPDATE statement: instead, object methods are used
- not Turing complete (cf. SQL/transitive closure)
- OQL can be embedded into suitable object-oriented programming languages (C++, Java, Smalltalk). Results of queries (collections!) are then processed by iterators.

Extents

SQL: SELECT ... FROM <relation> ...

What corresponds to a *relation* in an ODBMS ?

⇒ *Extension*: set of all instances of a class (similar to system-owned sets in NWDBMS).
Extensions are defined in ODL together with the class declaration:

```
class <name> (extent <extent_name>)
{ <attribute-def>;
  <relationship-def>;
  <operations-def>; }
```

```
class Country (extent Countries)
{ attribute string name;
  relationship City capital;
  set<string> languages;
  ...}
```


Queries

Queries against the database are expressed with the SELECT statement, with the same simple basic structure as in SQL:

```
SELECT  <expression>
FROM    <extents>
WHERE   ...
```

```
SELECT c.population
FROM Countries c
WHERE c.name = 'Germany'
```

- with an iterator variable (here: c) – cf. SQL Aliasing

Similar to SQL:

- DISTINCT, aggregate functions: COUNT, SUM, . . . , set functions: UNION, INTERSECT, EXCEPT (MINUS)

QUERIES

- SQL: all results of queries of the form

  SELECT a,b,c FROM ...

  are virtual “relations” (i.e. sets of tuples),
- OQL: the result is a virtual set of objects,
- in most cases an (implicit) collection.

```
SELECT c.capital
FROM Countries c
```

Result is of the type

collection <City>

⇒ queries can be nested arbitrarily (like in SQL)

QUERIES

in case that the result has more than one attribute (e.g. with SELECT *), a

bag <struct{...}>

is automatically generated:

```
SELECT c.name, c.population
FROM Countries c
WHERE c.name = 'Germany'
```

Result is of the type

bag <struct {string name; number population}>

Complex Results

- bags (here: set-valued relationship) can be handled as a whole,
- by explicit generation of a struct, the properties of the result can be renamed:

```
SELECT struct(name: c.name,
              cities: c.has_cities)
FROM Countries c
```

result is of the type

```
collection <struct {string name;
                    collection<city> cities}>
```

How can something *in the collection* be selected?

... straightforwardly: apply a SELECT statement to the collection:

```
SELECT struct(name: c.name,
              cities: (SELECT cty
                       FROM c.has_cities as cty
                       WHERE cty.population>1000000))
FROM Countries c
```

- Traversing the relationship *has_cities* by a *path expression* in the query
- nested SELECT in the SELECT statement: the inner SELECT ranges over the (virtual) set *c.has_cities* of instances of type *set<city>*.
- the inner SELECT is evaluated separately for every result (i.e. for each instance c) of the outer SELECT.

PATH EXPRESSIONS

for navigation along *scalar* relationships:

```
SELECT name: c.name,
       cpp: c.capital.province.population
FROM Countries c
```


SELECT IN THE FROM-CLAUSE

Navigation along *set-valued* relationships:

not allowed:

```
SELECT name: c.has_cities.name,
       pop: c.has_cities.population
FROM Countries c
WHERE c.name = “Germany”
```

has_cities is a *set of cities*, thus, the method *population* cannot be applied (to the set).

This can be done e.g. by a SELECT statement in the FROM-clause:

```
SELECT name: cty.name,
       pop: cty.population
FROM (SELECT c.has_cities
      FROM Countries c
      WHERE c.name = “Germany”) as cty
```

Correlated Joins

... do the above example even better:

```
SELECT name: cty.name,
           pop: cty.population
FROM Countries c, c.has_cities cty
WHERE c.name = “Germany”
```

This would be a nice feature also in SQL ... the right side of the join is computed dependent on the left one.

⇒ asymmetric joins that express nested iteration in a declarative way

⇒ not aligned with the relational algebra

OQL: Functional Language Concept

SQL:

- declarative, relational algebra as theoretical base,
- somewhat ad-hoc language (around SELECT – FROM – WHERE),
- not completely orthogonal composition (aggregate functions, method applications)

OQL:

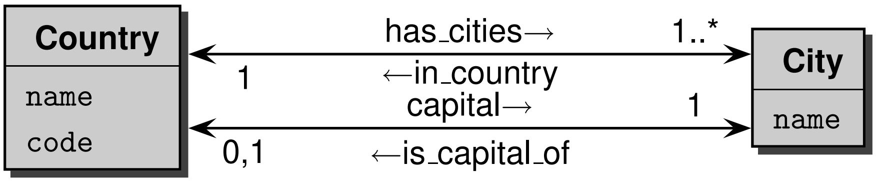
- orthogonal composition rules: operators can be nested as long as the type system is not violated
- functional concept, includes the *simple* queries in SQL syntax.
- result of a query is always a

  collection()
- can be processed in the same way as an extension (*intensional* part of the database).

Conclusion

- Object-oriented databases have not been accepted by the market.
- Products: ObjectStore, Adabas, O2, GemStone, Poet, ...
  Some of them served as the base for the first commercial XML database systems (Excelon, Tamino [Software AG]).
- Object-relational extensions to SQL and relational systems (SQL-3-Standard): evolutionary instead of revolutionary development.
- graph data model, “node + edge-labeled”
- set-oriented (extents similar to relations) and navigation-based access, integrated in a declarative language.
  Problems with navigating along set-valued properties.
- OQL as a functional language with fully orthogonal constructs and the possibility to generate structures in the SELECT-clause.
  The XML-Query language XQuery will be very similar ...
- OIF as self-describing ASCII-based data exchange format, but still with a fixed schema.

2.2.4 Analysis: 1:n-Relationships

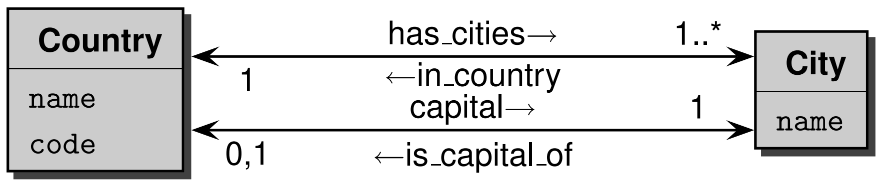


class Country { attribute string name;
relationship City capital   inverse City::is_capital_of;
relationship set<City> has_cities   inverse City::in_country; }

class City { attribute string name;}

- correct: germany.capital.name
- not correct: germany.has_cities.name
- translation to set<City> “country is in relation with a set of cities” is a tribute to programming language influence: must be something that exists in programming languages and that can be bound to a single variable.
  “set-valued” – one answer which is a set.
- applying “.name” to a set is obviously not correct.

ALTERNATIVE TRANSLATION

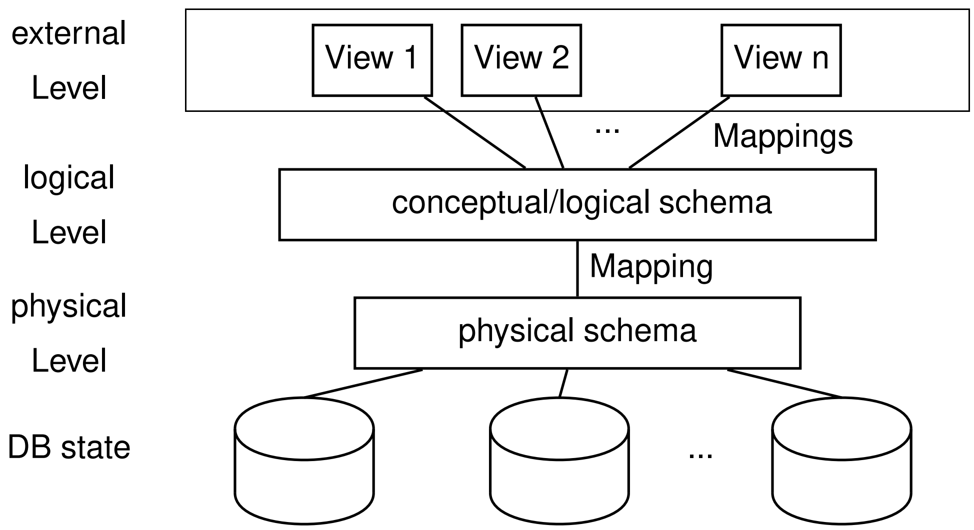


- database style: "country is in relation with multiple cities"
  "multi-valued" – a set of *answers*, each of them is a city,
- "set of answers" is a meta-concept of the query language, not of the underlying programming language,
- applying ".name" to a set of answers can be defined by the semantics of the query language!
- "Modern" query languages change to multivalued semantics:
  - F-Logic (1989, see later): germany.has_cities.name,
  - XPath (for XML, 1998): //country[name="Germany"]/city/name,
  - semantics of path expressions stays within the semantics of the query language.

2.3 Data Integration and Metadata Queries: SchemaSQL

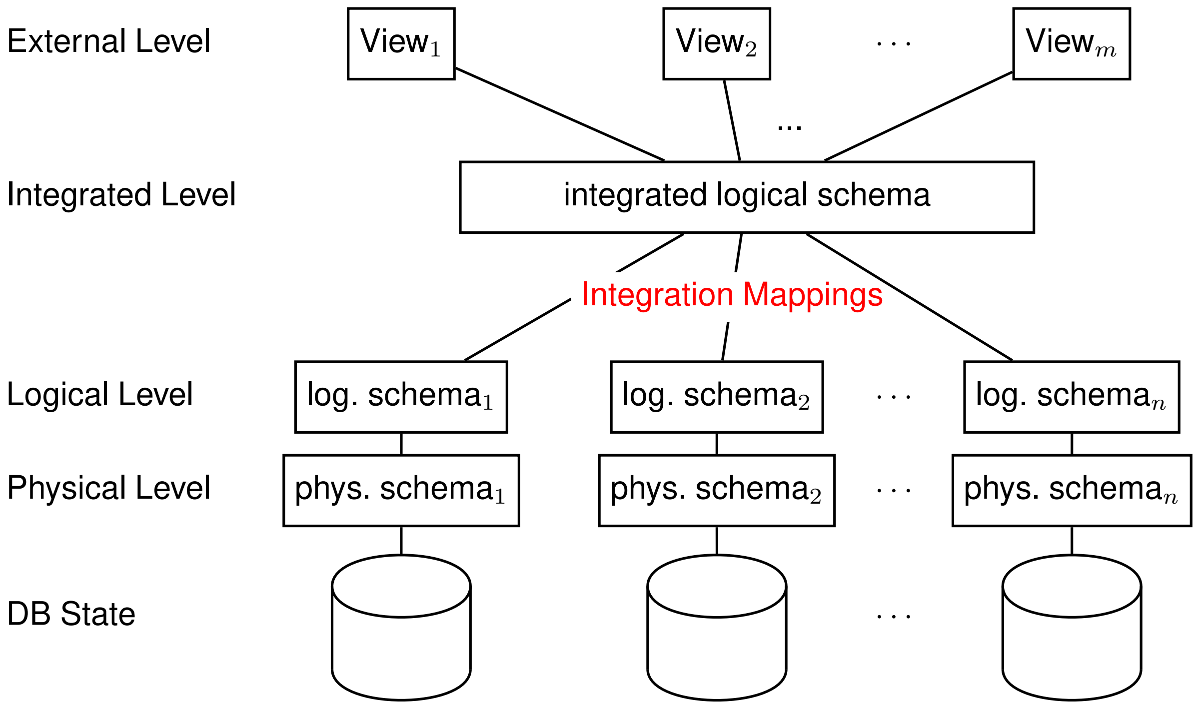
2.3.1 Introduction

- So far: single databases
- according to the classical 3-level architecture



Multidatabase Systems and Federated Database Systems

- providing a common, integrated view over several databases



Data Integration and Metadata Queries in SQL: SchemaSQL

SchemaSQL (Lakshmanan et al. 1996; non-commercial academic system) extends SQL:

- combination of relations and attributes of different (federated) databases.
- uniform handling of data and metadata (by SchemaSQL variables).
- possible domains of variables are the names of the components of a federation, names of the relations of a database, names of the attributes of a relation, tuples of a relation, and values of a column of a relation.
- additionally to the “vertical” aggregations over columns, also “horizontal” aggregations over relations or even tables are possible.

Example

univ-A

salInfo

category	dept	salary
Prof	CS	65,000
Assoc Prof	CS	50,000
Prof	Math	60,000
Assoc Prof	Math	55,000

univ-C

CS

category	salary
Prof	60,000
Assoc Prof	55,000

Math

category	salary
Prof	70,000
Assoc Prof	60,000

univ-B

salInfo

category	CS	Math
Prof	55,000	65,000
Assoc Prof	50,000	55,000

univ-D

salInfo

dept	Prof	Assoc Prof
CS	75,000	60,000
Math	60,000	45,000

2.3.2 Declaration of Variables

... as known from SQL in the FROM-clause: FROM <range> <var>

SQL: FROM <table> <var>

```
SELECT cty.name, cty.population
FROM City cty
```

$$< \text{range} > \in \{\rightarrow, \ \text{db} \rightarrow, \ \text{db} :: \text{rel}, \ \text{db} :: \text{rel} \rightarrow, \ \text{db} :: \text{rel.attr}\} \ .$$

- $\rightarrow$: names of the databases of the federation
- $db \rightarrow$: names of the relations of the database db.
- $db :: rel$: tuples of the relation rel of the database db [as in SQL].
- $db :: rel \rightarrow$: names of the attributes of the schema of the relation rel of the database db.
- $db :: rel.attr$: values of the column of the attribute $attr$ of the relation rel of the database db.
- SchemaSQL: iterated declarations of variables are allowed!
  $\Rightarrow$ joins not longer symmetrical (cf. OQL).

Declaration of Variables: Tuple- and Domain Variables

- tuple variables as known from SQL:
  $db :: rel$ ranges over the set of tuples of the relation rel of the database db.

  SELECT tuple.category, tuple.salary
  FROM univ-C::CS tuple

category	salary
"Prof"	60000
"AssocProf"	55000

- Domain-Variables:
  $db :: rel.attr$ ranges over the set of values of the attribute $attr$ of the relation rel of the database db

  SELECT cat
  FROM univ-A::salInfo.category cat

cat
"Prof"
"AssocProf"

  Note: SQL-style SELECT category FROM univ-A::salInfo
  yields the same result – but does not allow to bind the values
  to a variable (that can be used somewhere else)

Declaration of Variables: Metadata Variables

- $db \rightarrow$ ranges over the relation names of the database db.

  SELECT relname FROM univ-C $\rightarrow$ relname

relname
"CS"
"math"

- Nested declarations: Second variable depends on the first one:

  SELECT dept, tuple.category, tuple.salary
  FROM univ-C $\rightarrow$ dept, univ-C::dept tuple

dept	category	salary
"CS"	"Prof"	60000
"CS"	"AssocProf"	55000
"Math"	"Prof"	70000
"Math"	"AssocProf"	60000

  ... *integrates* both tables from univ-C in one.

Declaration of Variables

- Variables over names of attributes:
  $db :: rel \rightarrow$ ranges over the set of attribute names of the schema of the relation rel of the database db.

  SELECT attrname
  FROM univ-C::CS → attrname

attrname
"category"
"Salary"

- SELECT C: *name* of the attribute,
  SELECT T.C: *value* of the respective attribute of the current tuple.

  SELECT attrname, univ-C::CS.attrname
  FROM univ-C::CS → attrname

"category"	"Prof"
"category"	"AssocProf"
"Salary"	60000
"Salary"	55000

Declaration of Variables

- → ranges over the names of the databases of the federation.

  SELECT dbname FROM → dbname

dbname
"univ-a"
"univ-b"
"univ-c"
"univ-d"

- SELECT dbname, relname
  FROM → dbname, dbname → relname

dbname	relname
"univ-A"	"SalInfo"
"univ-B"	"SalInfo"
"univ-C"	"CS"
"univ-C"	"math"
"univ-D"	"SalInfo"

2.3.3 Queries

All departments of Univ-A that pay a higher salary to their professors than the corresponding departments of Univ-B:

```
select A.dept
   -- all variables are independent
from   univ-A::salInfo A, univ-B::salInfo B,
       univ-B::SalInfo-> AttB
where     AttB <> ``category'' and
          A.dept = AttB and
          A.category = ``Prof'' and
          B.category = ``Prof'' and
          A.salary > B.AttB.
```


Queries (Cont'd)

Same for C/D:

```
select RelC
   -- C depends on RelC
from     univ-C-> RelC, univ-C::RelC C,
         univ-D::salInfo D
where    RelC = D.dept and
         C.category = ``Prof'' and
         C.salary > D.Prof
```

AGGREGATION

Similar to SQL, there can be aggregation over a variable.

⇒ here also *horizontal* and *blockwise* aggregation possible.

Average salary for each kind of professors over all departments of Univ-B:

```
select T.category, avg(T.D)
from univ-B::salInfo→D,  univ-B::salInfo T
where D <> ‘‘category’’
group by T.category
```

- select the values for D,
- compute the cartesian product with univ-B::salInfo T
- include column T.D
- evaluate, do the grouping, compute the aggregate

D	category	CS	Math	T.D
category	Prof	55,000	65,000	55,000
CS	Prof	55,000	65,000	55,000
math	Prof	55,000	65,000	65,000
category	Assoc Prof	50,000	55,000	50,000
CS	Assoc Prof	50,000	55,000	50,000
math	Assoc Prof	50,000	55,000	55,000

Aggregation

Average salary for each kind of professors over all departments of Univ-C:

```
select T.category, avg(T.salary)
from univ-C→D, univ-C::D T
group by T.category
```

- compute values for D,
- join with tuple variable D T

D	category	salary
CS	Prof	60,000
CS	Assoc Prof	55,000
math	Prof	70,000
math	Assoc Prof	60,000

- grouping
- compute the aggregate

RESTRUCTURING

... as usual via views:

```
create view
BtoA::salInfo(category, dept, salary) as
     select T.category, D, T.D
     from univ-B::salInfo→D, univ-B::salInfo T
     where D <> ‘category’
```

creates a virtual database BtoA with a virtual relation salInfo in the same format as A::salInfo.

Restructuring

A to B: number of attributes of the result table depends on the number of departments.

⇒ Dynamic result schema

```
create view AtoB::salInfo(category,D) as
select A.category, A.salary
from univ-A::salInfo A, A.dept D
```

Result of the FROM-clause:

A.category	A.salary	A.dept D
Prof	65,000	CS
Assoc Prof	50,000	CS
Prof	60,000	Math
Assoc Prof	55,000	Math

Many-to-one-mapping into a schema of the form salInfo(category, dept$_1$, . . . , dept$_n$).

AtoB::salInfo

category	CS	Math
Prof	65,000	60,000
Assoc Prof	50,000	55,000

2.3.4 Exercise

Create the following view that represents the information of all four databases in a uniform way:

```
create view
globalSchema::salInfo(univ, dept, category, salary) as
[TO BE COMPLETED]
```


Solution

```
create view
globalSchema::salInfo(univ, dept, category, salary) as
   select ‘‘univ-A’’, T.dept, T.category, T.salary
   from univ-A::salInfo T
union
   select ‘‘univ-B’’, D, T.category, T.D
   from univ-B::salInfo T, univ-B::salInfo→D
   where D<>‘‘category’’
union
   select ‘‘univ-C’’, T, T.category, T.salary
   from univ-C→D, univ-C::D T
union
   select ‘‘univ-D’’, T.dept, C, T.D
   from univ-D::salInfo T, univ-D::salInfo→C
   where C<>‘‘dept’’
```

2.3.5 Query Evaluation

Federation System Table (FST): meta-information about the component databases, i.e. names of the databases, relations, attributes, or other statistical information that is useful for query evaluation (similar to the Data Dictionary in SQL).

Variable Instantiation Tables (VIT): contain the possible variable bindings during the evaluation (meta level).

Input: a *SchemaSQL* query

Output: bindings of the variables of the SELECT-clause of the query

Evaluation: two phases:

1. generation of the VITs according to the variables in the FROM-clause. For this, SQL queries are stated against the local databases and against the FST.
2. rewriting of the *SchemaSQL* query into an equivalent query using the VITs (Dynamic SQL). This query is then evaluated by the *resident SQL server*.

EVALUATION: EXAMPLE

```
select RelC
from univ-C→ RelC, univ-C::RelC C, univ-D::salInfo D
where RelC = D.dept and C.category = ‘‘Prof’’ and C.salary > D.Prof
```

Bindings for meta-variables (query against an FST):

VIT_{RelC}
RelC
CS
Math

Bindings for tuple variables (queries against component-DBS):

VIT_C (abh. of Rel_C)		
RelC	category	salary
CS	Prof	60,000
CS	Assoc Prof	55,000
Math	Prof	70,000
Math	Assoc Prof	60,000

VIT_D		
Dept	Prof	AssocProf
CS	75,000	60,000
Math	60,000	45,000

Evaluation: Example

... again the query:

```
select RelC
from univ-C→ RelC, univ-C::RelC C,
   univ-D::salInfo D
where RelC = D.dept and
      C.category = ''Prof'' and
      C.salary > D.Prof
```

Query evaluation via standard SQL over the $VIT's$.

```
select VIT_RelC.RelC
from VIT_RelC, VIT_C, VIT_D
where VIT_C.RelC = VIT_RelC.RelC    % Correlation RelC, C
   and VIT_RelC.RelC = VIT_D.dept
   and VIT_C.category = ''Prof''
   and VIT_C.salary > VIT_D.Prof
```


Exercise: Schema-SQL

Describe the evaluation of the query given on Slide 76 with its FST and VITs.

Solution

VIT_{dbname}
dbname
univ-A
univ-B
univ-C
univ-D

$VIT_{relname}$	
dname	relname
univ-A	salInfo
univ-B	salInfo
univ-C	CS
univ-C	math
univ-D	salInfo

SELECT VIT_{dbname}.dbname, $VIT_{relname}$.relname
FROM $VIT_{dbname}, VIT_{relname}$
WHERE VIT_{dbname}.dbname = $VIT_{relname}$.relname

2.3.6 Example: Integration of Stock Exchange Data

Frankfurt::Quota		
Date	Name	Price
3.3.93	sun	150
3.3.93	dc	151
3.3.93	b.u.	160
4.3.93	sun	153
4.3.93	dc	154
4.3.93	b.u.	163

Tokyo::Quota			
Date	sun	dc	fuji
3.3.93	150	151	140
4.3.93	153	154	140

Sydney::3.3.	
Name	Price
sun	150
dc	151
kiwi	130

Sydney::4.3.	
Name	Price
sun	153
dc	154
kiwi	135

New York::sun	
Date	Price
3.3.93	150
4.3.93	153

New York::dc	
Date	Price
3.3.93	151
4.3.93	154

New York::msoft	
Date	Price
3.3.93	148
4.3.93	74

Possible extension:
Euro vs. Dollar vs. Yen

EXERCISE: SCHEMA-SQL

- Formulate the "On which days had which stocks the price of 150 $?" for the schemata given on Slide 91.
- In commercial database systems, the schema information is stored in the *Data Dictionary* (cf. the following excerpts of table definitions of the data dictionary):

```
SQL> desc sys.user_tables;
 Name                    Null?    Type
 ----------------------- -------- ----
 TABLE_NAME              NOT NULL VARCHAR2(30)
 SQL> desc sys.user_tab_columns;
 Name                    Null?    Type
 ----------------------- -------- ----
 TABLE_NAME              NOT NULL VARCHAR2(30)
 COLUMN_NAME             NOT NULL VARCHAR2(30)
 DATA_TYPE               VARCHAR2(30)
```

  Describe how the above queries can be formulated in an environment where SQL is embedded into a procedural programming language (e.g. embedded-SQL or PL/SQL) (Pseudocode).

SOLUTION: SCHEMA-SQL

- SELECT Date, Name

```
SELECT Date, Name
FROM Frankfurt::Quota
WHERE Price=150;
```

```
SELECT Date, AttrName
FROM Tokyo::Quota.Date, Tokyo::Quota → AttrName
WHERE AttrName ≠ 'Date' AND Price=150;
```

```
SELECT NewYork::TabName.Date, TabName
FROM NewYork → TabName
WHERE Price=150;
```

```
SELECT TabName, Sydney::TabName.Name
FROM Sydney → TabName
WHERE Price=150;
```

- Information from the Data Dictionary is only needed for Tokyo, New York and Sydney.

SOLUTION: SQL

Algorithm for SQL in a procedural environment (database Tokyo):

- Store the result of

```
SELECT ColumnName
FROM Tokyo.user_tab_columns
WHERE ColumnName ≠ 'Date';
```

(result: the names of the companies) and for each result <cn> execute the query

```
SELECT Date, <cn>
FROM Tokyo.Quota
WHERE <cn>= 150;
```

and collect all results.

Solution: SQL

- database “New York”: store the result of

```
SELECT TableName
FROM user_tables
WHERE
   (  SELECT ColumnName
      FROM user_tab_columns UTC
      WHERE UTC.TableName=TableName = {Date,Price});
```

(the comparison of sets must be formulated in SQL) and for each result <tn> evaluate the query

```
SELECT Date, <tn>
FROM <tn>
WHERE Price = 150;
```

and collect all results.

Problem: SQL statements must be generated *dynamically*: the results of the first query are used in the second statement.

Solution: Dynamic SQL

This is e.g. possible in Oracle by using the DBMS_SQL-Package (to be used with PL/SQL), which allows to generate SQL statements at runtime:

```
create procedure findnumber as
declare
 cursor col_cursor is
   select column_name, data_type
   from sys.user_tab_columns
   where table_name = upper('&&table_name')
   order by column_id;
 lv_column_name    sys.user_tab_columns.column_name%TYPE;
 lv_column_typ     sys.user_tab_columns.data_type%TYPE;
 lv_rowid          varchar2(20);
 rows_processed    number;
 loop_count        number;
 stmnt             varchar2(2000);
 doublecur         BINARY_INTEGER;
 execute_feedback INTEGER;
 type colname_typ is table of lv_column_name%TYPE
      index by binary_integer;
 type rowid_typ is table of lv_rowid%TYPE
```

```
      index by binary_integer;
 colname_table      colname_typ;
 empty_colname      colname_typ;
 rowid_table        rowid_typ;
 empty_rowid_table  rowid_typ;

begin
   DBMS_OUTPUT.ENABLE(10000);
   rows_processed := 0;
   -- Search for attributes with datatype "Number"
   open col_cursor;
   loop
      fetch col_cursor into
         lv_column_name, lv_column_typ;
      exit when col_cursor%notfound;
      IF lv_column_typ='NUMBER' THEN
         rows_processed := rows_processed+1;
         colname_table (rows_processed)
              := lv_column_name;
      END IF;
   end loop;
   close col_cursor;

   -- Initialize query statement
   stmnt := 'select rowid from '
       || '&&table_name '
```



```
       || 'where ';

   -- generate the query iteratively
   loop_count := 1;
   WHILE loop_count <= rows_processed
   loop
      stmnt := stmnt
          || colname_table(loop_count)
          || ' = &&Price';
      if loop_count < rows_processed
      then
        stmnt := stmnt || ' or ';
      end if;
      loop_count := loop_count + 1;
   end loop;
   DBMS_OUTPUT.PUT_LINE
           ('Computed Query: ' || stmnt);

   -- execute the generated statement
   doublecur := DBMS_SQL.OPEN_CURSOR;
   DBMS_SQL.PARSE (doublecur
             ,stmnt
             ,DBMS_SQL.V7);
   DBMS_SQL.DEFINE_COLUMN
        (doublecur, 1, lv_rowid, 20);
   execute_feedback := DBMS_SQL.EXECUTE (doublecur);
```

```
   -- generate list of all resulting data records and
   -- RowIDs
   loop
     if DBMS_SQL.FETCH_ROWS (doublecur) = 0
     then
       exit;
     else
       DBMS_SQL.COLUMN_VALUE (doublecur,1, lv_rowid);
       DBMS_OUTPUT.PUT_LINE('RowID: ' ||lv_rowid);
     end if;
   end loop;

   -- cleaning ...
   DBMS_SQL.CLOSE_CURSOR (doublecur);
   colname_table := empty_colname;
end;
/
```


Solution: Dynamic SQL

```
SQL> execute find-number;
Give value for table_name: Tokyo
Give a value for price: 150
Generated Query:
  select rowid from Tokyo
  where SUN = 150 or DC = 150 or FUJI = 150

RowID: AAAA2MAADAAAD7nAAA

SQL> select * from Tokyo
     where rowid='AAAA2MAADAAAD7nAAA';
03.03.93        150         151        140
```

which must still be postprocessed for obtaining the answer `'sun', 3.3.93`.

- Conclusion: SchemaSQL helps to express such queries much shorter and more concise, and it is easier to learn than PL/SQL and DBMS_SQL.

2.3.7 Exercise: Horizontal and blockwise Grouping

- Consider the schemata `univ-B`, `univ-C` and `univ-D`. Give SchemaSQL queries that return for each kind of professors the average salary over all departments.

SOLUTION: HORIZONTAL AND BLOCKWISE GROUPING

- univ-A: same as in standard SQL: vertical aggregation:

  select T.category, avg(T.salary)
  from univ-A::salInfo T   – tuple variable
  group by T.category

- univ-B: horizontal aggregation
  see Slide 81.

- univ-C: aggregation over different tables
  see Slide 82.

- univ-D: aggregation over different columns:

```
select T.category, avg(T.C)
from univ-B::salInfo T,   univ-B::salInfo → C
where C <> ‘‘dept’’
group by C
```

Conclusion

- integration of *relational* databases with different schemas
- queries against metadata
- combination of metadata and data
- data-dependent *generation* of schema

New Features

Generalization of the use of variables:

- SQL: variables only ranging over tuples of a fixed relation,
- SchemaSQL: variables ranging over "everything": data: tuples, column values metadata: names of columns, names of relations, even names of databases,
- intuitively simple extension of SQL,
- powerful feature for data integration.

Such variables are more (F-Logic) or less extremely (XML: XPath/XQuery) used in Semistructured Data and XML.