Kapitel 2
Verschiedene Konzepte und Erweiterungen

- Der Begriff “semistrukturierte Daten (SSD)” wurde im wesentlichen durch das “TSIMMIS”-Projekt (The Stanford-IBM Manager of Multiple Information Sources, 1995-2000); Personen u.a. J. Ullman, H. Garcia-Molina, J. Widom, Y. Papakonstantinou geprägt.
- Das Problem wurde aber vorher schon in verschiedenen Bereichen bearbeitet.

Warum Semistrukturierte Daten?

... im wesentlichen zwei Anforderungen:

1. *Datenintegration* aus verschiedenen Quellen (Ende 1980er/Anfang 1990er):
   - zunehmende Vernetzung
   - Verknüpfung von Inhalten verschiedener Datenbanken
     * Multidatenbanksysteme
     * föderierte Datenbanksysteme
     * unterschiedliche Schemata
     * meistens relational, aber nicht immer (“Legacy”-Netzwerkdatenbanken)
     * [ab Mitte 90er zunehmend auch aus dem Web]
     * teilweise unter dem Aspekt der Einbeziehung von Metadaten in die DML, der *eigentlich* unabhängig von semistrukturierten Daten ist.

WARUM SEMISTRUKTURIERTE DATEN?

2. Speicherung "unregelmäßiger" Daten:
   kein festes/homogenes/bekanntes Schema, viele Nullwerte, zum Teil auch Volltext-Anteile
   - z.B. Biochemie
   - Datenaustausch (B2B); Standardformate z.B. für Automobilzulieferer
   - Verwaltung von Dokument-Inhalten
     * sehr grobe Struktur
       SGML (als Sonderform: HTML)
   - Mischformen zwischen Datenbank und Dokumenten
     * Sammlungen: Steuergesetzgebung, Jura ... zum Teil in SGML
     * Krankenhaus-Informationssysteme

DIE ENTWICKLUNG

Die Entwicklung im Bereich semistrukturierter Daten und XML kombinierte Konzepte, Erfahrungen und Entwicklungen aus vielen vorhergehenden Ansätzen:

- Netzwerkmodell, hierarchisches Modell ("Legacy"-Datenbanken)
- Relationale Datenbanken
- Objektorientierte Datenbanken
- verteilte und föderierte Datenbanken
- Datenintegration (rein relationales Umfeld, oder gemischt)
- Dokumentmanagement

Verschiedene Entwicklungslinien, die mit XML & friends zusammengeführt wurden.

⇒ (fast) nichts neues, aber eine perfekte Kombination

Kompendium/Lehrbuch zu "Datenbanken" allgemein (aber ohne Dokumente und XML):

- R.Elmasri, S.Navathe: "Foundations of Databases"/"Grundlagen von Datenbanksystemen". Pearson Studium, 3.Auflage, 2002.

2.1 Netzwerkdatenbanken

- erste Entwicklungen bei dem Datenbanksystem *IDS (Integrated Data Store)* bei *General Electric* (Bachman & Williams, 1964; Turing-Award 1973)
- Spezifikation der “Computer Data Systems Language Data Base Task Group (CODASYL)”, 1971.
- Produkte: VAX-DBMS (Digital Equipment) etc.

NETZWERKMODELL

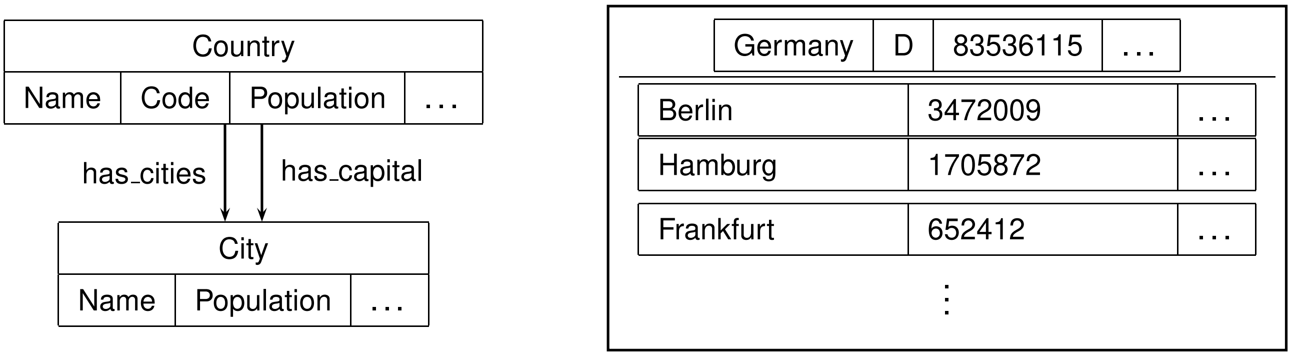
- Daten in *Datensätzen* gespeichert
- klassifiziert nach *Datensatztypen*, mit Attributen (Name und Datentyp anzugeben)
- soweit also wie Abbildung von Entitätstypen im relationalen Modell.

Country				
Name	Code	Population	Area	...

City		
Name	Population	...

Beziehungen: Mengentypen

- Name des Mengentyps
- Eigentümerdatensatztyp (“owner”; “von wo die Beziehung ausgeht”)
- Mitgliedsdatensatztyp (“member”)
- Jede solche Mengeninstanz besteht aus
  - einem Datensatz des Eigentümerdatensatztyps
  - einer *geordneten* Menge von Datensätzen des Mitgliedsdatensatztyps
- ein Mitgliedsdatensatztyp kann maximal in einer Mengeninstanz jedes Mengentyps vorkommen (also nur 1:N-Beziehungen direkt modellierbar)



Organisation der Mengentypen

Jeder Datensatz enthält zwei Zeigereinträge für jeden Mengentyp, an dem er als Eigentümer oder als Mitglied teilnimmt:

- als Eigentümer: einen “first”-Zeiger, beschriftet mit dem Mengentypnamen auf den ersten Mitgliedsdatensatz
- als Mitglied: einen “next”-Zeiger, beschriftet mit dem Mengentypnamen auf den nächsten Mitgliedsdatensatz
- Außerdem einen beschrifteten Rückwärtszeiger oder direkten Zeiger auf den Eigentümer innerhalb jedes Mengentyps.
- einen beschrifteten Nullzeiger, falls kein erstes oder nächstes solches Element existiert.

Zugriffsoperationen

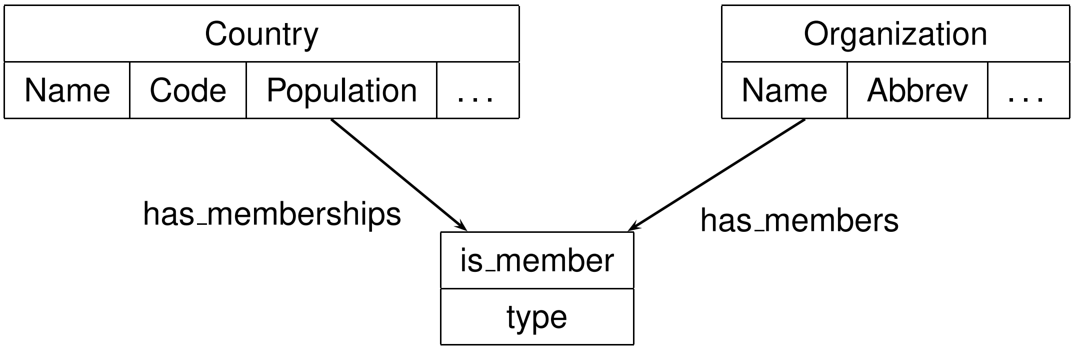
Damit hat man de facto einen Abstrakten Datentyp:

- Zugriff auf die einzelnen Attributwerte
- einen *Iterator* (first, next) zum Durchlaufen der Beziehungen
- einen Zugriff “find_owner” für inverse Beziehungen

N:M-BEZIEHUNGEN

können nicht durch einen einzigen Mengentyp dargestellt werden (analog für attributierte Beziehungen).

- Aufspalten in eine 1:M und eine inverse N:1-Beziehung
  Problem: Konsistenzerhaltung
- Einführen eines Datensatztyps, der die Beziehung repräsentiert und zweier Mengentypen.

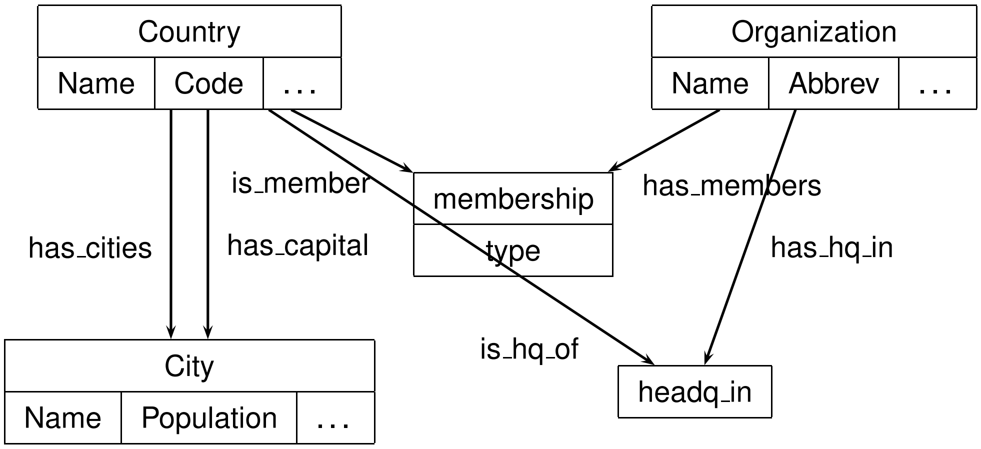


SONSTIGES

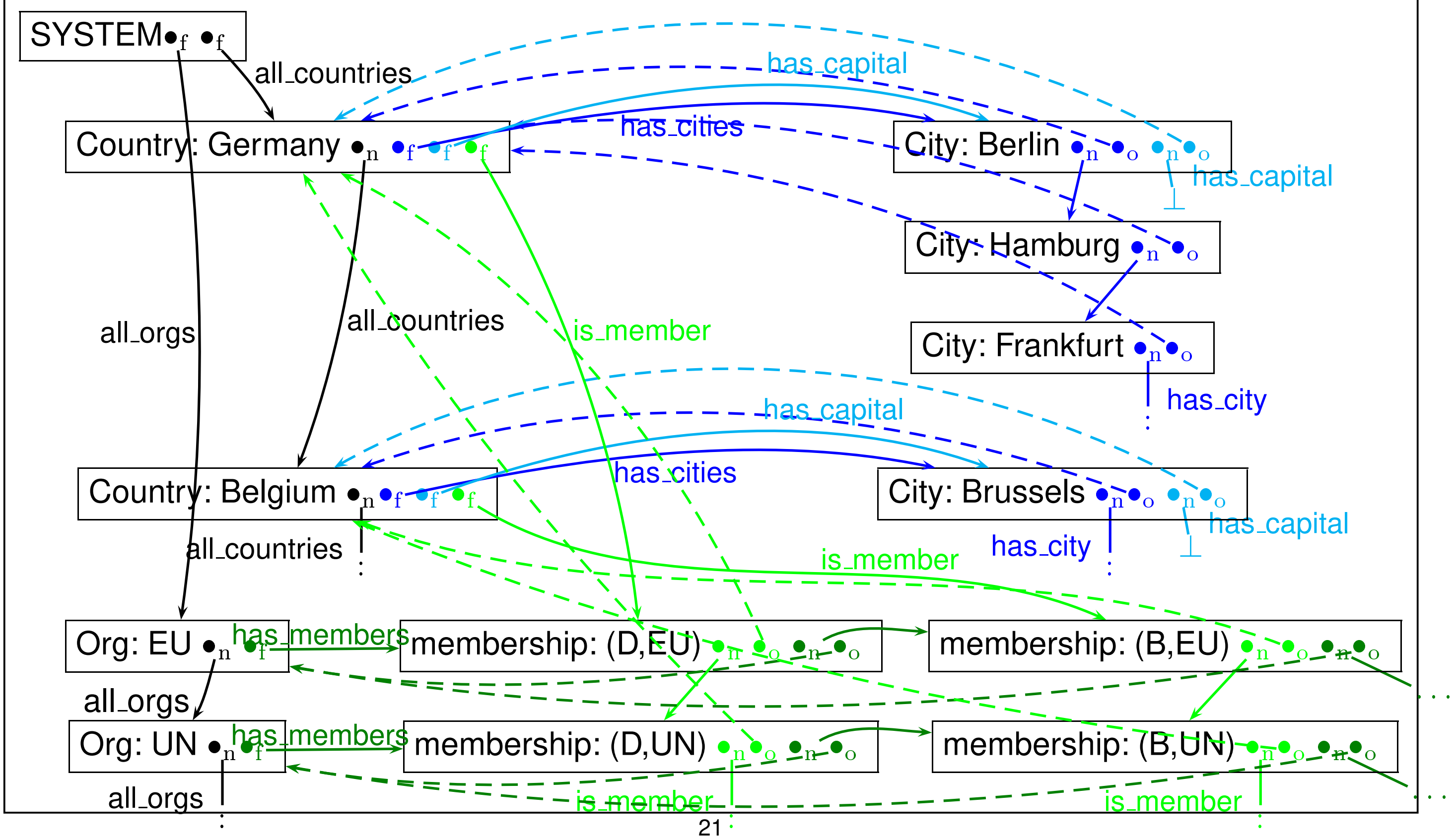
- Systemeigene Mengeninstanzen dienen als Einstiegspunkte
  (z.B. eine Mengeninstanz “all_countries”, deren Mitglieder die country-Datensätze sind)

Aufgabe 1
Veranschaulichen Sie sich das Modell mit ein paar Ländern, Städten und Organisationen. Betrachten Sie auch die Beziehung has_headq, die beschreibt, dass eine Organisation ihren Sitz in einer Stadt hat. □

LÖSUNG: NETZWERKSCHEMADIAGRAMM



LÖSUNG: INSTANZEBENE



DATENDEFINITIONSSPRACHE AM BEISPIEL

```
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
```


ANFRAGE- UND DATENMANIPULATIONSSPRACHE

- basiert auf o.g. Iteratorkonzept
- Navigations-, Zugriffs- und Update-Befehle
- eingebettet in eine Wirtssprache (Host-Language; COBOL, PL/I, Pascal, C)

Suchen von Datensätzen eines Datensatztyps

- FIND ANY <Datensatztyp> [USING <Record-Spec]
- FIND DUPLICATE <Datensatztyp> [USING <Record-Spec]

```
// vorher: city als geeignete Record-Variable definieren
city.name = “Neustadt”
FIND ANY city USING name
GET city     // holt Datensatz in Umgebungsvariable city
FIND DUPLICATE city USING name
GET city     // holt Datensatz in Umgebungsvariable city
```

- mit WHILE DBSTATUS=0 (= erfolgreich gefunden) kann man eine Schleife programmieren

Suchen von Datensätzen eines Mengentyps

- Voraussetzung: man sitzt auf einer aktuellen Datensatzinstanz
- FIND (FIRST | NEXT | PRIOR | LAST) WITHIN <Mengentyp> [USING <Record-Spec]
- FIND OWNER WITHIN <Mengentyp>

```
// Annahme: aktueller Datensatz ist "Germany"
FIND FIRST city WITHIN has_capital
GET city      // holt Datensatz in Umgebungsvariable city

// immer der zuletzt mit FIND geholte Datensatz ist der "aktuelle Datensatz",
// also jetzt "Berlin"
// relativ zu diesem wird weiternavigiert
FIND OWNER WITHIN has_city
GET country      // holt Datensatz in Umgebungsvariable country
```

- Joins sind also nur über Navigation und Schleifen in der Hostsprache möglich.

Aufgabe 2
Schreiben Sie ein Programm, das alle Organisationen ausgibt, die ihren Sitz in der Hauptstadt eines ihrer Mitgliedsländer haben.

Vergleichen Sie mit der entsprechenden SQL-Anfrage an Mondial. □

Updates

Datensatz-Updates

STORE, ERASE, MODIFY (des aktuell betrachteten Datensatzes)

Mengen-Updates

CONNECT, DISCONNECT, RECONNECT (jeweils für den aktuell betrachteten Datensatz)

Hierarchisches Datenbankmodell

- Im Ganzen sehr ähnlich, Eltern-Kind-Beziehungen definieren eine Baumstruktur; zusätzlich "virtuelle" Parent-Child-Relationships.
- System: IMS (IBM & Rockwell International, Ende 1960er)

LÖSUNG

```
find any organization in all_organizations
while ok do
{ get organization
  find first headq_in within has_headq_in
  get headq
  find owner within is_headq_of
  get city
  find first headq_in within is_headq_of using * // aktuellen
  find owner within has_headq_in    // DS wieder restaurieren!
  find first membership within has_members
  while ok do
  { get membership
    find owner within is_member // jetzt hat man ein Land
    find first city within has_capital using name // von city
    if ok then println(organization.name)
    find next membership within has_members
  }
  find owner within has_memberships
find duplicate organization in all_organizations //naechste Organisation
}
```


DASSELBE 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

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

Fazit

- Seit dem Aufkommen von SQL zunehmend aus dem Markt gedrängt, inzwischen nur noch in Legacy-Systemen existent.
- Wenig zugrundeliegende Theorie (als Basis für Normalisierung und Optimierung)
- nur prozedurale, datensatzorientierte Anfragesprache, nicht deklarativ, muss eingebettet werden (COBOL, PL/I, Pascal, C).
- "schnell mal eine Anfrage stellen" unmöglich.
- Idee der Navigation und Eltern/Kind-Beziehungen zwischen Datensätzen jedoch elegant (keine Probleme mit referentieller Integrität).
  Diese Konzepte sieht man in späteren Ansätzen wieder ...
- Graph-Datenmodell, "node + edge-labeled"
- insbesondere geordnete Kind-"Datensätze" treten auch in XML wieder auf. Dort hat man dann
  - DOM als abstrakten Datentyp,
  - XPath/XQuery als *deklarative* Sprache darüber.

2.2 Objektorientierte Datenbanken

Mitte 80er: Objektorientierung (Programmiersprachen und Modellierung (UML)): Anwendungsprogramme werden objektorientiert entwickelt und programmiert.

- "Impedance Mismatch" zwischen DB und Programm-Datenstrukturen

Ziele:

- Objekte des Anwendungsprogramms persistent machen
- Objektorientierung bis in das DBMS durchziehen
  - Klassenhierarchie und Vererbung, Polymorphie
  - Implementierung und Kapselung von Verhalten

In dieser Vorlesung werden OODBS nur kurz und ausschnittweise behandelt.
Ein längeres Vorlesungsmodul in der Vorlesung "Information Systems"
(verfügbar auf `user.informatik.uni-goettingen.de/~may/Lectures`) zu finden).

Lebenszeit von Objekten

- Objektorientierte Programmiersprache: Objekte werden zur Laufzeit eines Anwendungsprogramms erzeugt und bei Programmende wieder zerstört.
- Datenbanksysteme:
- persistent: Objekte werden bei einer Aktivität erzeugt und im Datenbanksystem abgelegt und bleiben erhalten, auch wenn die erzeugende Aktivität beendet ist.
- transient: Objekte werden nur zur Ausführung einer Aktivität benötigt. Sie existieren nur, solange eine Anwendung konkret abläuft und werden entsprechend nur vom Laufzeitsystem der Programmiersprache verwaltet.

Lebenszeit von Objekten

- Relationale DBMS: Alle SQL-Typen haben nur persistente Instanzen, die im DBMS liegen, alle Nicht-SQL-Typen (also Typen der umgebenden Hostsprache) haben nur transiente Instanzen, die mit Beendigung des Anwendungsprogramms (= Verlassen der Hostsprache) zerstört werden.

  Andererseits können persistente Objekte nur durch SQL manipuliert werdem während transiente Objekte nur mit den Mitteln der Host-Programmiersprache verwendet werden können.

- ODBMS: Objekttypen des DBMS und der Anwendung sind dieselben. Sie können gleichzeitig persistente und transiente Instanzen besitzen.

  Für persistente und transiente Objekte wird dieselbe Programmiersprache und dieselben Operationen verwendet.

OBJEKTORIENTIERTES DATENMODELL

- beschreibt nur den *(Datenbank-)Zustand* (*Attribute, Beziehungen, Klassenzugehörigkeit und Klassenhierarchie*), nicht das *Verhalten*
- Abbild der Anwendungsdomäne, “Miniwelt”
- ausdruckskräftiger als das relationale Modell/ER-Modell
- (Objektverhalten wird in die Datenmanipulationssprache integriert)

OO-DBMS

Standardisierungsansätze ähnlich zur Standardisierung relationaler Datenbanken:

Erfolg der relationalen Datenbanksysteme:

- nicht nur durch das einfache, high-level Datenmodell,
- sondern auch durch die Standardisierung: SQL
  - Portabilität
  - Interoperabilität

ODMG: Object Database Management Group

- Gegründet 1991
- Architektur von OODBMS, DDL, Anfragesprache (OQL), Datenformat
- ODMG-93 (1.0) Standard
- ODMG-2.0-Standard (1997)
- ODMG-3.0-Standard (2000); inkrementelle Änderungen

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

ODMG: Object Database Management Group

- Voting Members: Organisationen/Firmen, die ein ODBMS kommerziell anbieten, u.a. von JavaSoft, Windward Solutions, Lucent Technologies, Unidata, GemStone, Versant, ...
  Reviewer Members: Organisationen, die ein materielles Interesse an der Arbeit von ODMG haben.
- Es ist nicht das Ziel, identische Produkte zu definieren, sondern Source-Code Portabilität zu erreichen (vgl. Java, SQL, später auch XML).
- Dabei bleibt genügend Freiheit, eigene Produktziele zu definieren:
  - Leistungsfähigkeit, Optimierung, (Preis)
  - Unterstützung von Programmiersprachen,
  - auf bestimmte Anwendungsbereiche abgestimmte Funktionalität (Multimedia, CAD, ...), vordefinierte Typen
  - integrierte Programmierumgebungen, Design Tools ...

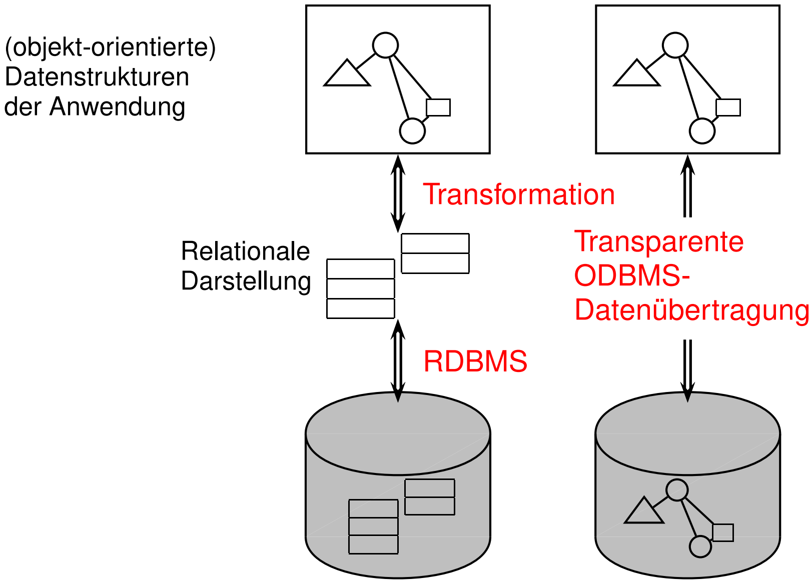
Architektur von ODBMS

- Unterschiedlich zu “klassischen” relationalen DBMS:
  SQL: High-Level Sprache zur Datenmanipulation,
  Anwendung muß in einer anderen Programmiersprache geschrieben werden (vgl. Embedded-Konstrukte).
- ODBMS/ODM: transparente Integration von DBMS-Funktionalität (Persistenz, Mehrbenutzerbetrieb, Recovery) in Anwendungsprogrammiersprache (vgl. Persistent Java).
  Es werden einfach die Objekte der Anwendung in der Datenbank gespeichert.
- keine separate DML mehr notwendig. Anwendungsprogrammiersprache *ist* die DML.
- Fast, zumindest. Eine mengenorientierte, deklarative Anfragesprache ist immer noch etwas schönes.
  (Ein Rest des Impedance Mismatch zwischen Variablen- und Mengenorientierung bleibt bestehen)
- keine Transformation zwischen der (logischen) Datenbankrepräsentation und Repräsentation in der Programmiersprache (vgl. Datentypkonvertierung in JDBC).

ARCHITEKTUREN

ODMG befaßt sich mit zwei Produkttypen:

- Object Database Management Systems (ODBMSs) speichern die Objekte direkt,
- Object-to-Database Mappings (ODMs) konvertieren Objekte und speichern sie in einer relationalen (oder sonstwie) Repräsentation.



Anmerkung: Für XML-Datenbanken gibt es entsprechende unterschiedliche Realisierungsansätze.

ODMG-STANDARD

Aus verschiedenen Sprachen bestehender Standard zur implementierungsnahen Spezifikation von objektorientierten Systemen.

KOMPONENTEN DES ODMG-STANDARDS

- Objektspezifikationssprachen/Datenmodell
  - Object Definition Language (ODL)
  - Object Interchange Format (OIF)
- Object Query Language (OQL) – basiert auf SQL
- C++/Smalltalk/Java Language Binding
  spezifiziert die Arbeit mit persistenten Objekten in den Zielsprachen.

2.2.1 ODL: Object Definition Language

- Data Definition Language für Objekttypen:
- keine Programmiersprache, sondern eine Sprache zur Definition von Objektspezifikationen,
- erweitert IDL (Interface Definition Language) aus dem ODMG/CORBA-Standard (die wiederum eng mit der Java-Deklarationssprache verwandt ist)
- charakterisiert Objekttypen (Klassenhierarchie, Eigenschaften und Beziehungen)

DATENTYPEN: LITERALE

Literale sind nur Werte, sie haben keine *Objektidentität*.

Atomare Literale

- long, short, unsigned long/short, float, boolean, char, string,
- enumeration {...} ("Typgenerator")
  Z.B. enum Weekday {Sunday, Monday, . . . , Saturday}

Strukturierte Literale

- vordefinierte Typen: date, interval, time, timestamp
  (neben den *echten Objekttypen* Date, Interval, Time, Timestamp)
- benutzerdefinierte Strukturtypen, z.B. address oder

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

Kollektionsliterale

- set<t>, bag<t>, list<t>, array<t>, dictionary<t>
  (neben den *echten Kollektionstypen* Set, Bag, List, Array, Dictionary)

Klassen

... dienen dazu, komplexe Objekttypen zu definieren und Objekte zu kategorisieren.

Klassen definieren im Objektmodell die *Signatur* ihrer Instanzen (die Implementierung gehört nicht zum Objektmodell):

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

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

```
class City { attribute string name;      % Attribute ...
             attribute number population;
             attribute geoCoord coordinates;
             relationship Country in_country;    % ... und Beziehungen
             }
```


Beziehungen

- Beziehungen werden bei der Definition von Klassen spezifiziert.
- in UML/ODMG nur *binäre* Beziehungen erlaubt.
- *Bidirektionale* Beziehungen spezifizierbar.
- one-to-one / one-to-many / many-to-many-Beziehungen.

```
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> für -to-many-Beziehungen

BEZIEHUNGEN

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;
             relationship Country in_country   inverse Country::has_cities; }
```


BEZIEHUNGEN

- Die Instanzebene bildet einen Graphen:
  - Knoten: Objekte; Knoten haben Labels (Namen der Objekttypen) sowie eine ID
  - Kanten: Beziehungen; Kanten haben Labels (Namen der Beziehungen)
- ODBMS ist für Aufrechterhaltung der referentiellen Integrität verantwortlich:
  Wenn ein Objekt gelöscht wird, müssen auch alle Beziehungen zu ihm gelöscht werden.
- Beziehungen definieren *Zugriffspfade*, z.B. Germany.capital zur Navigation innerhalb des Graphen.
- Graph-Datenmodell, “node + edge-labeled”
- Mit set<...> lebt die mengenorientierte Repräsentation mehrwertiger Beziehungen wie im Netzwerkmodell wieder auf.
- Die Anfragesprache OQL wird das aber SQL-mäßig deklarativ lösen.

Aufgabe 3
Veranschaulichen Sie einen Ausschnitt aus der Mondial-Datenbank als Graphen. □

2.2.2 Object Interchange Format (OIF)

- Datenbankzustand in ein oder mehrere Files speichern (vgl. *export* in ORACLE)
- Spezifikationssprache für persistente Objekte und ihre Zustände
- OIF-File enthält für jedes Objekt den Typ, die Attributwerte und Beziehungen zu anderen Objekten;
- das Datenbankschema (Klassendefinitionen *und* Klassenhierarchie) wird *nicht* in OIF abgespeichert!

OBJECT INTERCHANGE FORMAT (OIF)

- Einfachste Form: nur die Klassenzugehörigkeit

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

- Attributwerte werden in der Klammer aufgezählt:

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

- Strukturierte Attribute: Geschachtelte Klammerstrukturen

```
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, Mehrwertige Beziehungen:

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

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

- Zyklische Referenzen: problemlos.
- Attributierte Beziehungen (z.B. border) problematisch

⇒ OIF ist schon ein selbstbeschreibendes Datenformat!

2.2.3 OQL (Kurzübersicht)

- Anfragesprache des ODMG-Standards (Object Query Language)
- ähnlich SQL:
  SELECT - FROM - WHERE - Klausel, erweitert um komplexe Objekte, Objektidentität, Pfadausdrücke, Polymorphismus, Operationsausführung und Late Binding.
- aber: funktionale Sprache (ist SQL auch), voll orthogonal (ist SQL nicht)
- kein explizites UPDATE-Statement: Hier werden stattdessen Objektmethoden benutzt.
- nicht berechnungsvollständig (vgl. SQL/transitive Hülle)
- OQL kann in geeignete objektorientierte Programmiersprachen (C++, Java, Smalltalk) eingebettet werden.

EXTENTS

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

Was entspricht einer *Relation* im ODBMS ?

⇒ *Extension*: Menge aller Instanzen einer Klasse.

Extensionen werden in ODL bei der Klassendeklaration mitdefiniert:

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

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


ANFRAGEN

Anfragen an die Datenbank werden mit dem SELECT-Befehl formuliert. Dieser hat prinzipiell eine sehr einfache Grundstruktur:

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

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

- mit Iteratorvariable (hier: c) – vgl. SQL Aliasing

Wie in SQL:

- DISTINCT, Aggregatfunktionen: COUNT, SUM, . . . , Mengenfunktionen: UNION, INTERSECT, EXCEPT (MINUS)

ANFRAGEN

- In SQL liefern alle Anfragen

  SELECT a,b,c FROM ...

  virtuelle "Relationen" (d.h. Mengen von Tupeln) als Ergebnis,
- in OQL ist das Ergebnis eine virtuelle Menge von Objekten,
- meistens eine (implizite) Kollektion

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

Ergebnis ist vom Typ

collection <City>

⇒ Anfragen beliebig schachtelbar

ANFRAGEN

Ist das Ergebnis "mehrspaltig" (u.a. mit SELECT *), wird automatisch ein

bag <struct{...}>

erzeugt:

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

Ergebnis ist vom Typ

bag <struct {string name; number population}>

KOMPLEXE ERGEBNISSE

- bags (hier: mengenwertige Beziehung) können im ganzen behandelt werden,
- durch explizite Erzeugung eines structs können die Eigenschaften des Ergebnisses benannt werden:

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

Ergebnis ist vom Typ

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

Was ist, wenn man jedoch *in der Collection* etwas auswählen will?

... ganz einfach: ein SELECT-Statement darauf anwenden ;)

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

- Traversierung der Beziehung *has_cities* mittels *Pfadausdruck* in der Anfrage
- geschachteltes SELECT im SELECT-Statement: das innere SELECT läuft über die (virtuelle) Instanzenmenge *c.has_cities* vom Typ *set<city>*.
- das innere SELECT wird für jede Iteration (d.h. für jede Instanz c) des äußeren SELECT ausgewertet.

PFADAUSDRÜCKE

zum Navigieren entlang *skalarer* Beziehungen:

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


SELECT IN DER FROM-ZEILE

Navigation entlang *mengenwertiger* Beziehungen:

`nicht erlaubt:`

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

has_cities ist eine *Menge von Städten*, daher kann *darauf* die Methode *population* nicht angewendet werden.

kann z.B. durch SELECT-Statements in der FROM-Zeile formuliert werden:

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

KORRELIERTE JOINS

... damit geht das obige Beispiel noch schneller:

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

Das hätte man gerne auch in SQL ... die rechte Seite des Joins wird abhängig von der linken berechnet.

⇒ Unsymmetrische Joins, die geschachtelte Iteration deklarativ ausdrücken

OQL: FUNKTIONALES SPRACHKONZEPT

SQL:

- deklarativ
- ad-hoc Sprache (rund um SELECT – FROM – WHERE)
- keine orthogonalen Kompositionsregeln (man stellt sich häufig die Frage, warum irgend etwas nicht geht)

OQL:

- orthogonale Kompositionsregeln: Operatoren können geschachtelt werden, solange das Typsystem nicht verletzt wird.
- funktionales Konzept, umfaßt für die *einfachen* Anfragen die SQL-Syntax.
- Ergebnis einer Anfrage ist immer eine

  collection()
- kann also wie eine Extension weiterverwendet werden (*intensionaler* Anteil der Datenbasis).

FAZIT

- Objektorientierte Datenbanken haben sich am Markt nicht durchgesetzt.
- Produkte: u.a. ObjectStore, Adabas, O2, GemStone, Poet, ...
  einige von ihnen bildeten die Grundlage der ersten kommerziellen XML-Datenbanksysteme (Excelon, Tamino [Software AG]).
- Objektrelationale Erweiterungen zu SQL und relationalen Systemen (SQL-3-Standard): evolutionäre anstatt revolutionäre Entwicklung.
- Graph-Datenmodell, “node + edge-labeled”
- Mengenorientierter (Extents ähnlich zu Relationen) und Navigationsbasierter Zugriff, in einer deklarativen Sprache integriert.
  Probleme bei der Navigation mengenwertiger Eigenschaften.
- OQL als funktionale Sprache mit voll orthogonalen Sprachkonstrukten und der Möglichkeit, Strukturen in der SELECT-Zeile zu generieren.
  So ähnlich sieht auch die XML-Anfragesprache XQuery aus.
- OIF als selbstbeschreibendes ASCII-Datenaustauschformat, aber bei immer noch festem Schema.

2.3 Metadaten-Anfragen in SQL: SchemaSQL

SchemaSQL (Lakshmanan et al. 1996; nichtkommerzielles akademisches System) erweitert SQL:

- Kombination von Relationen und Attributen unterschiedlicher (föderierter) Datenbanken.
- uniforme Behandlung von Daten und Metadaten (durch SchemaSQL-Variablen).
- Mögliche Definitionsbereiche von Variablen sind die Namen der Komponenten der Föderation, Namen der Relationen einer Datenbank, Namen der Attribute einer Relation, Tupel einer Relation, und Werte einer Spalte einer Relation.
- Zusätzlich zu “vertikalen” Aggregierungen über Spalten sind “horizontale” Aggregierungen über Relationen oder Tabellen möglich.

Beispiel

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

2.3.1 Deklaration von Variablen

... wie von SQL bekannt in der FROM-Zeile: FROM <range> <var>

SQL: FROM <table> <var>

SELECT cty.name, cty.population
FROM City cty

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

- $\rightarrow$: Datenbanknamen der Föderation.
- $db \rightarrow$: Relationsnamen der Datenbank db.
- $db :: rel$: Tupel der Relation rel der Datenbank db [wie in SQL].
- $db :: rel \rightarrow$: Attributnamen des Schemas der Relation rel der Datenbank db.
- $db :: rel.attr$: Werte in der Spalte des Attributs $attr$ der Relation rel der Datenbank db.
- SchemaSQL: aufeinander aufbauende Variablendeklarationen sind erlaubt!
  $\Rightarrow$ Joins nicht mehr symmetrisch (vgl. OQL).

Deklaration von Variablen: Tupel- und Domainvariablen

- Tupel-Variablen wie von SQL bekannt:
  $db :: rel$ benennt die Menge der Tupel der Relation rel der Datenbank db.

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

category	salary
"Prof"	60000
"AssocProf"	55000

- Domain-Variablen:
  $db :: rel.attr$ benennt die Menge der Werte in der Spalte des Attributs $attr$ der Relation rel der Datenbank db

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

cat
"Prof"
"AssocProf"

Deklaration von Variablen: Metavariablen

- $db \rightarrow$ benennt die Menge der Relationsnamen der Datenbank db.

  SELECT relname FROM univ-C $\rightarrow$ relname

relname
"CS"
"math"

- Geschachtelte Deklaration: Zweite Variable ist von der ersten abhängig:

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

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

... und damit hat man beide Tabellen aus univ-C in einer.

Deklaration von Variablen

- Variablen über Attributnamen:
  $db :: rel \rightarrow$ benennt die Menge der Attributnamen des Schemas der Relation rel der Datenbank db.

  SELECT attrname
  FROM univ-C::CS $\rightarrow$ attrname

attrname
"category"
"Salary"

- `SELECT C`: *Bezeichner* des Attributes,
  `SELECT T.C`: jeweiliger Attribut*wert* des betrachteten Tupels.

  SELECT attrname, univ-C::CS.attrname
  FROM univ-C::CS $\rightarrow$ attrname

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

Deklaration von Variablen

- $\rightarrow$ benennt die Menge der Datenbanknamen der Föderation.

  SELECT dbname FROM $\rightarrow$ dbname

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

- SELECT dbname, relname
  FROM $\rightarrow$ dbname, dbname $\rightarrow$ relname

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

2.3.2 Anfragen

Alle Fachbereiche von Univ-A, die ihren Professoren ein höheres Gehalt zahlen, als die entsprechenden Fachbereiche in Univ-B:

```
select A.dept
   -- alle Variablen unabhängig voneinander
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.
```


Anfragen (Cont’d)

Dasselbe für C/D:

```
select RelC
  -- C abhängig von 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

Wie in SQL kann man über eine Variable aggregieren.

⇒ hier auch *horizontale* und *blockweise* Aggregation möglich.

Durchschnittliches Gehalt für jede Mitarbeitergruppe über alle Fachbereiche von univ-C:

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

- Werte für D bestimmen,
- kartesisches Produkt mit univ-B::salInfo T bilden
- Spalte T.D mit einbeziehen
- auswerten, gruppieren, aggregieren

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

Durchschnittliches Gehalt für jede Mitarbeitergruppe über alle Fachbereiche von univ-C:

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

- Werte für D bestimmen,
- Join mit Tupelvariable D T

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

- gruppieren
- aggregieren

RESTRUKTURIERUNG

... wie üblich mit 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’
```

erzeugt eine virtuelle Datenbank BtoA mit einer virtuellen Relation salInfo in demselben Format wie A::salInfo.

Restrukturierung

A to B: Breite der auszugebenden Tabelle hängt von der Anzahl der Departements ab.

⇒ Dynamisches Ausgabeschema

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

Ergebnis der FROM-Zeile:

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

AtoB::salInfo

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

Many-To-One-Mapping in ein Schema der Form salInfo(category,dept$_1$, ..., dept$_n$).

2.3.3 Aufgabe

Erstellen Sie das folgende View, das die Informationen der vier Datenbanken uniform darstellt:

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


LÖSUNG

```
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''
```

2.3.4 Anfrageberechnung

Federation System Table (FST): Meta-Information über die Komponentendatenbanken, wie Namen der Datenbanken, Relationen, Attribute, oder für Anfrageoptimierung nützliche statistische Information (analog Data Dictionary).

Variable Instantiation Tables (VIT): enthalten die während der Ausführung einer Anfrage möglichen Variablenbindungen (auf Metaebene).

Input: eine *SchemaSQL*-Anfrage

Output: Bindungen der Variablen der SELECT-Klausel der Anfrage

Auswertung: Zwei Phasen:

1. Erzeugen der VITs entsprechend den Variablen in der FROM-Klausel. Dazu werden SQL-Anfragen an lokale Datenbasen sowie die FST gestellt.
2. Umschreiben der *SchemaSQL*-Anfrage in eine äquivalente Anfrage unter Verwendung der VITs (Dynamic SQL). Diese Anfrage wird dann durch den *resident SQL server* ausgewertet.

AUSWERTUNG: BEISPIEL

```
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
```

Bindungen für Metavariablen (Anfrage an FST):

VIT_{RelC}
RelC
CS Math

Bindungen für Tupelvariablen (Anfragen an Komponenten-DBS):

VIT_C (abh. von 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

Auswertung: Beispiel

... nochmal die Anfrage:

```
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
```

Anfragebearbeitung mittels Standard-SQL auf den $VIT's$.

```
select VIT_RelC.RelC
from VIT_RelC, VIT_C, VIT_D
where VIT_C.RelC = VIT_RelC.RelC    % Korrelation RelC, C
   and VIT_RelC.RelC = VIT_D.dept
   and VIT_C.category = ‘‘Prof’’
   and VIT_C.salary > VIT_D.Prof
```


AUFGABE: SCHEMA-SQL

Beschreiben Sie den Ablauf der Anfrage von Folie 63 mit FST und VITs.

Lösung

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

2.3.5 Beispiel: Integration von Börseninformationen

Frankfurt::Kurs		
Datum	Name	Wert
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::Kurs			
Datum	sun	dc	fuji
3.3.93	150	151	140
4.3.93	153	154	140

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

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

New York::sun	
Datum	Wert
3.3.93	150
4.3.93	153

New York::dc	
Datum	Wert
3.3.93	151
4.3.93	154

New York::msoft	
Datum	Wert
3.3.93	148
4.3.93	74

Verschärfung:
Euro vs. Dollar vs. Yen

AUFGABE: SCHEMA-SQL

- Formulieren Sie die Anfrage “An welchen Tagen hatten welche Aktien einen Kurs von 150 $?” für die auf Folie 78 angegebenen Schemata.
- In kommerziellen Datenbanksystemen sind die Schemainformationen im *Data Dictionary* abgelegt (vgl. die beiden folgenden Ausschnitte der relevanten Tabellendefinitionen):

```
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)
```

Beschreiben Sie, wie die obigen Anfragen in einer Umgebung, wo SQL in eine prozedurale Programmiersprache eingebettet ist (z.B. embedded-SQL oder PL/SQL) ablaufen könnten (Pseudocode).

LÖSUNG: SCHEMA-SQL

- SQL-Anfragen:

```
SELECT Datum, Name
FROM Frankfurt::Kurs
WHERE Wert=150;

SELECT Datum, AttrName
FROM Tokyo::Kurs.Datum, Tokyo::Kurs → AttrName
WHERE AttrName ≠ 'Datum' AND Wert=150;

SELECT NewYork::TabName.Datum, TabName
FROM NewYork → TabName
WHERE Wert=150;

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

- Auf die Informationen des Data Dictionary muß nur in Tokyo, New York und Sydney zugegriffen werden.

LÖSUNG: SQL

Algorithmus für SQL in einer prozeduralen Umgebung (Datenbank Tokyo):

- Ausgabe von

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

speichern (Ergebnis: die Namen der Firmen) und für jedes Ergebnis <cn> die Anfrage

```
SELECT Datum, <cn>
FROM Tokyo.Kurs
WHERE <cn>= 150;
```

auswerten und alle Ergebnisse sammeln.

Lösung: SQL

- Datenbank C: Ausgabe von

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

  speichern (die verwendete Mengengleichheit muß natürlich in SQL umschrieben werden) und für jedes Ergebnis `<tn>` die Anfrage

  ```
  SELECT Datum, <tn>
  FROM <tn>
  WHERE Wert = 150;
  ```

  auswerten und alle Ergebnisse sammeln.

Problem hierbei: SQL-Statements müssen *dynamisch* generiert werden: Die Ergebnisse der ersten Anfrage müssen in das zweite Statement eingebaut werden.

LÖSUNG: DYNAMISCHES SQL

In Oracle ist dies z.B. mit dem DBMS_SQL-Package (in PL/SQL einzubinden) möglich, das es erlaubt, zur Laufzeit SQL-Statements zu erzeugen:

```
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;
   -- Suche Attribute mit Datentyp "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;

   -- Initialisiere Abfrage-Statement
   stmnt := 'select rowid from '
       || '&&table_name '
```



```
       || 'where ';

   -- Erstelle iterativ den Abfrage-Befehl
   loop_count := 1;
   WHILE loop_count <= rows_processed
   loop
      stmnt := stmnt
          || colname_table(loop_count)
          || ' = &&Wert';
      if loop_count < rows_processed
      then
        stmnt := stmnt || ' or ';
      end if;
      loop_count := loop_count + 1;
   end loop;
   DBMS_OUTPUT.PUT_LINE
          ('Berechnete Abfrage: ' || stmnt);

   -- Fuehre den neuen Befehl aus
   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);
```

```
   -- Erstelle Liste der gefundenen DS und
   -- RowIDs aus.
   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;

   -- Aufraeumarbeiten und Ende
   DBMS_SQL.CLOSE_CURSOR (doublecur);
   colname_table := empty_colname;
end;
/
```


Lösung: Dynamisches SQL

```
SQL> execute find-number;
Geben Sie einen Wert fuer table_name ein: Tokyo
Geben Sie einen Wert fuer wert ein: 150
Berechnete Abfrage:
  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
```

was man natürlich noch weiter nacharbeiten muss um die Antwort `'sun', 3.3.93` zu bekommen.

- Fazit: Schema-SQL macht solche Anfragen erheblich kürzer und übersichtlicher und ist einfacher zu lernen als PL/SQL und DBMS_SQL.

2.3.6 Aufgabe: Horizontale und blockweise Gruppierungen

- Geben Sie für die Schemata `univ-B`, `univ-C` und `univ-D` SchemaSQL-Anfragen an, die für jede Gehaltsklasse das durchschnittliche Grundgehalt über alle Departements berechnen.

LÖSUNG: HORIZONTALE UND BLOCKWEISE GRUPPIERUNGEN

- univ-A: Wie in Standard SQL: vertikale Aggregierung:

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

- univ-B: horizontale Aggregierung
  siehe Folie 68.

- univ-C: Aggregierung über verschiedene Tabellen
  siehe Folie 69.

- univ-D: Aggregierung in verschiedenen Spalten:

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

FAZIT

- Anfragen an Metadaten
- Kombination von Metadaten und Daten
- datenabhängige Schemaerzeugung möglich