Datenbanksysteme II

Prof. Dr. E. Rahm

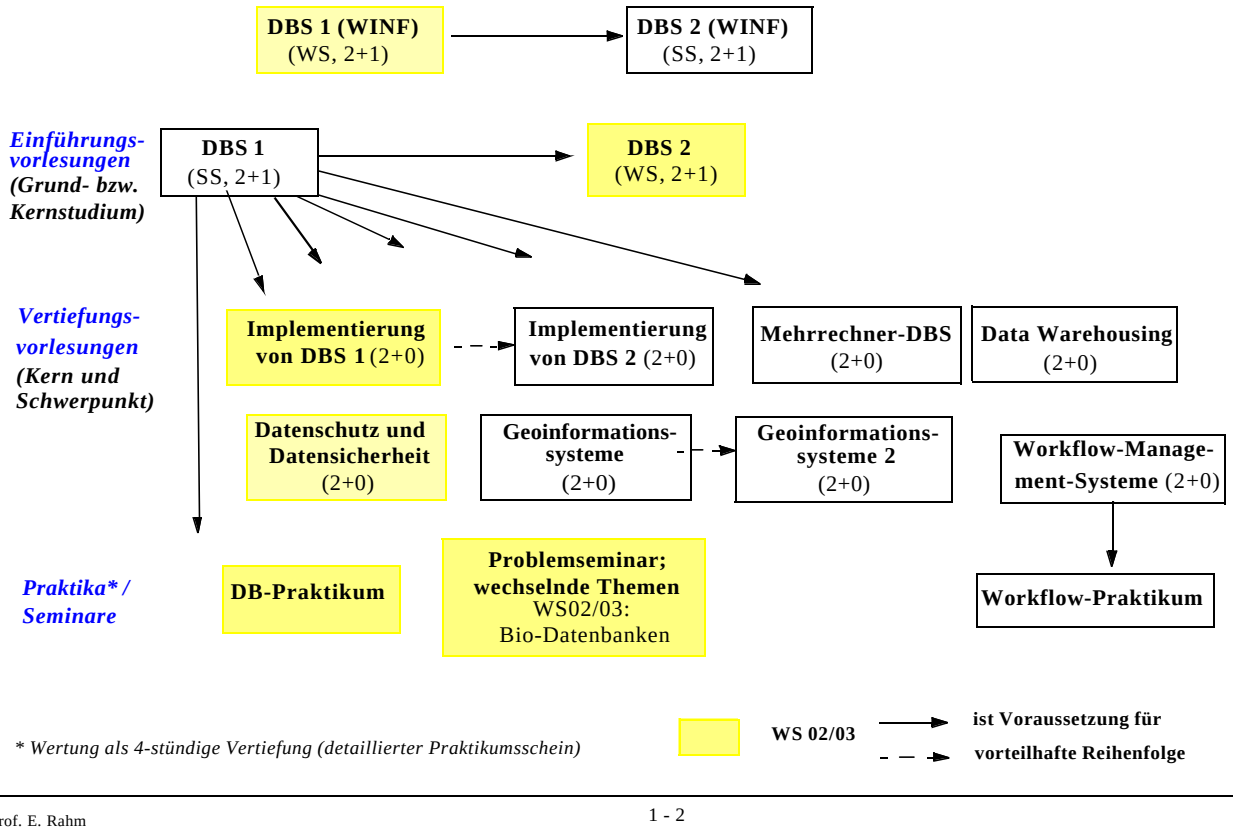
Wintersemester 2002/2003

Universität Leipzig

Institut für Informatik

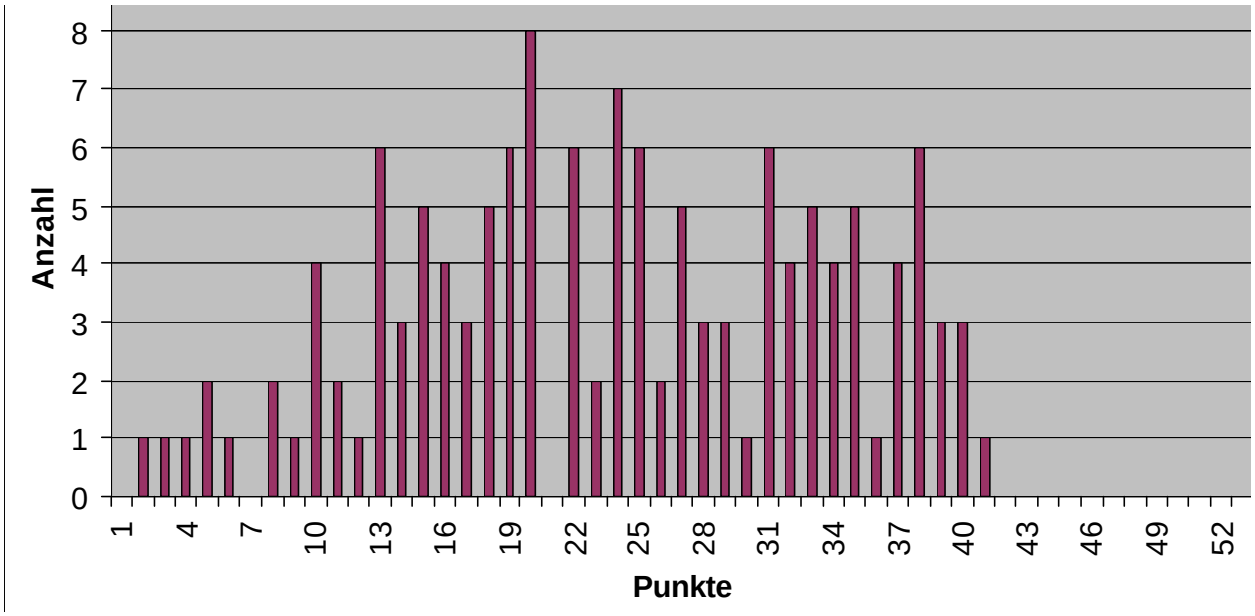
http://dbs.uni-leipzig.de

Lehrveranstaltungen zu "Datenbanken"(WS02/03)



* *Wertung als 4-stündige Vertiefung (detaillierter Praktikumsschein)*

Ergebnisse DBS1-Scheinklausur



- 134 Teilnehmer, erreichbar 53 Punkte, erreicht: 2 - 40,5 Punkte, durchschnittlich 23,3 Punkte
- 77 bestandene, 57 nicht bestandene Klausuren
- ADS-Vordiplomsklausur: 99 von 119 Klausuren bestanden (5x Note 1, 32x Note 2)

Leistungsbewertung

- Informatik, Hauptstudium (z.B. 5. Semester)
  - Kernfach „Praktische Informatik“
  - Modulklausur DBS2 im Februar als Teil der Diplomprüfung des Kerngebiets „Praktische Informatik“
  - Kernstudium umfasst insgesamt Prüfungen zu 32 SWS Vorlesungen, davon 6 - 14 SWS Praktische Informatik
- Übungsbetrieb (DBS2 hat 2+1 SWS)
  - 2-wöchentliche Übungsblätter (Beginn 28.10.)
  - Besprechung jeweils 1 Woche später in den Übungen
  - keine Abgabe von Lösungen, kein Übungsschein
  - zwei Übungsgruppen:

Nr.	Termin	Hörsaal	**Beginn**	Weitere Termine	Übungsleiter
1	Mo, 17:15	HS 16	**4.11.**	18.11. 9.12. (Ausweichtermin wg. Dies Acad.) 6.1.; 20.1.; 3.2.	Müller
2	Di, 11:15	SG 3-01	**5.11.**	19.11; 3.12; 7.1.; 21.1.; 4.2.	Müller

- Zeiten Sun-Pool HG 1-46 (für praktische Übungen)
  - Di 12-20 Uhr, Mi 8-14 Uhr, Do 12-20 Uhr

Vorlesungsziele

- Vermittlung vertiefter Kenntnisse, Fähigkeiten und Fertigkeiten
  - in der Nutzung von Informations- und Datenmodellen, insbesondere
    - Erweiterungen des Relationenmodells und SQL
    - objektorientierte und objekt-relationale DBS
    - Web-Datenbanken
  - in der Modellierung von anwendungsbezogenen Realitätsausschnitten
  - Überblick zu Anbindung von Datenbanken ans WWW

- Voraussetzung für Übernahme von Tätigkeiten:
  - Entwicklung von datenbankgestützten Anwendungen
  - Nutzung von Datenbanken unter Verwendung von (interaktiven) Datenbanksprachen
  - Systemverantwortlicher für Datenbanksysteme, insbesondere Datenbank-, Datensicherungs-, Anwendungs- und Unternehmensadministrator

Vorläufiges Inhaltsverzeichnis

Vorläufiges Inhaltsverzeichnis (2)

Lehrbücher (Auswahl)

Autoren	Titel	Verlag	Auflage	Jahr
Kemper, A.; Eickler, A.	Datenbanksysteme	Oldenbourg	4	2001
Ramakrishnan, R.; Gehrke, J.	Database Management Systems	McGraw Hill	3	2003
Garcia-Molina, H. et al.	A First Course in Database Systems	Prentice Hall	2	2001
Geppert, A.	Objektrelationale und objekt-orientierte Datenbankkonzepte und -systeme	dpunkt	1	2002
Saake, G.; Sattler, K.	Datenbanken & Java	dpunkt	1	2000
Rahm, E.; Vossen, G. (Hrsg.)	Web & Datenbanken	dpunkt	1	2003
Kazakos, W., et al.	Datenbanken und XML	Springer	1	2002

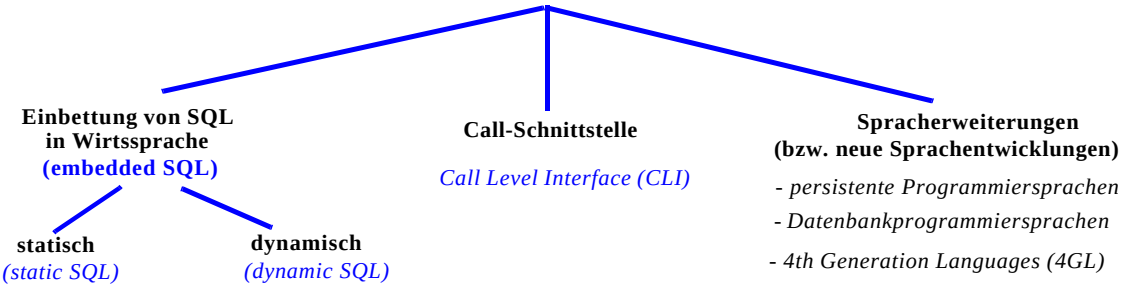
SQL-Eigenschaften

- Einfachheit
  - relativ leichte Erlernbarkeit eines wichtigen Subsets; Gleichförmigkeit der Syntax (SELECT FROM WHERE)
  - einheitliche Datendefinition und -manipulation
- Mächtigkeit
  - hohes Auswahlvermögen der Operatoren, Aggregatfunktionen, Sortierung, Gruppenbildung, ...
  - Benutzung als stand-alone und eingebettete Sprache
  - Sichtkonzept
  - flexibles Autorisierungskonzept
  - integrierter Katalog zur Datenbeschreibung
- hohe Datenunabhängigkeit
  - vollständiges Verbergen physischer Aspekte (Indexstrukturen etc.): physische Datenunabhängigkeit
  - Leistungsoptimierung durch DBS
  - Sichtkonzept unterstützt zum Teil logische Datenunabhängigkeit
- Standardisierung, weite Verbreitung
- Nachteile
  - fehlende formale Definition
  - Zusammenarbeit mit Wirtssprachen
  - hohe Redundanz; stark zunehmende Komplexität durch SQL92 und SQL99

1. DB-Anwendungsprogrammierung

- Einleitung
- Eingebettetes SQL
  - Cursor-Konzept
  - positionierte Änderungsoperationen (UPDATE, DELETE)
  - Behandlung von Nullwerten (Indikatorkonzept)
  - Verwaltung von Verbindungen
  - Dynamisches SQL
- Call-Level-Interface
  - Unterschiede zu eingebettetem SQL
  - Standardisierung
- Gespeicherte Prozeduren (Stored Procedures)
  - Konzept
  - Standardisierung: Persistente SQL-Module (PSM)
  - prozedurale Spracherweiterungen von SQL

Kopplung mit einer Wirtssprache



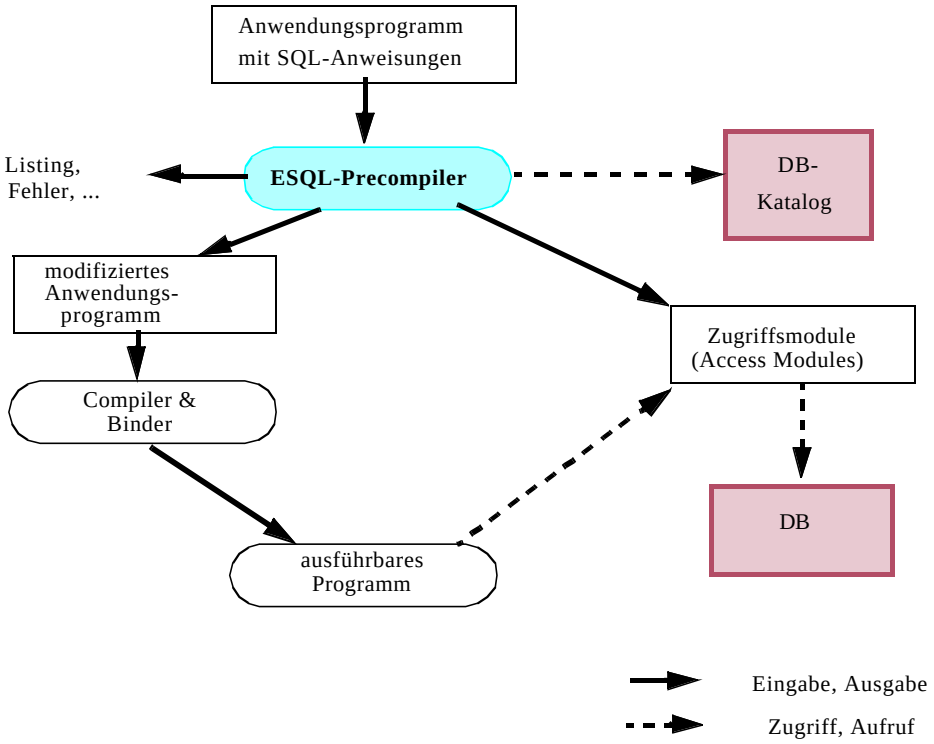
- Call-Schnittstelle (prozedurale Schnittstelle, Call-Level-Interface bzw. CLI)
  - DB-Funktionen werden durch Bibliothek von Prozeduren realisiert
  - Anwendung enthält lediglich Prozeduraufrufe
- Einbettung von SQL (Embedded SQL, ESQL)
  - Spracherweiterung um spezielle DB-Befehle (EXEC SQL ...)
  - komfortablere Programmierung als mit CLI
- statische Einbettung
  - Vorübersetzer (Prä-Compiler) wandelt DB-Aufrufe in Prozeduraufrufe um
  - Nutzung der normalen PS-Übersetzer für umgebendes Programm
  - SQL-Anweisungen müssen zur Übersetzungszeit feststehen
  - im SQL92-Standard unterstützte Sprachen: C, COBOL, FORTRAN, Ada, PL1, Pascal, MUMPS
- dynamische Einbettung: Konstruktion von SQL-Anweisungen zur Laufzeit

Embedded (static) SQL: Beispiel für C

```
exec sql include sqlca; /* SQL Communication Area */
main ()
{
exec sql begin declare section;
  char  X[8];
  int   GSum;
exec sql end declare section;
exec sql connect to dbname;
exec sql insert into PERS (PNR, PNAME) values (4711, 'Ernie');
exec sql insert into PERS (PNR, PNAME) values (4712, 'Bert');
printf("ANR ? "); scanf(" %s", X);
exec sql select sum (GEHALT) into :GSum from PERS where ANR = :X;
printf("Gehaltssumme: %d\n", GSum)
exec sql commit work;
exec sql disconnect;
}
```

- Anmerkungen
  - eingebettete SQL-Anweisungen werden durch "EXEC SQL" eingeleitet und durch spezielles Symbol (hier ";") beendet, um Compiler Unterscheidung von anderen Anweisungen zu ermöglichen
  - Verwendung von AP-Variablen in SQL-Anweisungen verlangt Deklaration innerhalb eines "declare section"-Blocks sowie Angabe des Präfix ":" innerhalb von SQL-Anweisungen
  - Werteabbildung mit Typanpassung durch INTO-Klausel bei SELECT
  - Kommunikationsbereich SQLCA (Rückgabe von Statusanzeigern u.ä.)

Verarbeitung von ESQL-Programmen



Cursor-Konzept

- Kernproblem bei SQL-Einbettung in konventionelle Programmiersprachen: Abbildung von Tupelmengen auf die Variablen der Programmiersprache
- Cursor-Konzept zur satzweisen Abarbeitung von DBS-Ergebnismengen
  - Cursor ist ein Iterator, der einer Anfrage (Relation) zugeordnet wird und mit dessen Hilfe die Tupeln des Ergebnismenge einzeln (one tuple at a time) im Programm bereitgestellt werden
  - Trennung von Qualifikation und Bereitstellung/Verarbeitung von Tupeln
- Operationen auf einen Cursor C1
  - DECLARE C1 CURSOR FOR table-exp
  - OPEN C1
  - FETCH C1 INTO VAR1, VAR2, . . ., VARn
  - CLOSE C1
- Anbindung einer SQL-Anweisung an die Wirtssprachen-Umgebung
  - Übergabe der Werte eines Tupels mit Hilfe der INTO-Klausel bei FETCH
    ==> INTO target-commalist (Variablenliste d. Wirtsprogramms)
  - Anpassung der Datentypen (Konversion)
- kein Cursor erforderlich für Select-Anweisungen, die nur einen Ergebnissatz liefern (SELECT INTO)

Cursor-Konzept (2)

- Beispielprogramm in C (vereinfacht)

```
...
   exec sql begin declare section;
     char  X[50];
     char  Y[8];
   exec sql end declare section;
   exec sql declare c1 cursor  for select NAME from PERS where ANR = :Y;
   printf("ANR ? "); scanf(" %s", Y);
   exec  sql  open  C1;
   while  (sqlcode == ok) {
         exec  sql  fetch  C1  into  :X;
         printf("%s\n", X)}
   exec  sql  close  C1;
...
```

- Anmerkungen
  - DECLARE C1 . . . ordnet der Anfrage einen Cursor C1 zu
  - OPEN C1 bindet die Werte der Eingabevariablen
  - Systemvariable SQLCODE zur Übergabe von Fehlermeldungen (Teil von SQLCA)

Cursor-Konzept (3)

```
DECLARE cursor [INSENSITIVE] [SCROLL] CURSOR FOR table-exp
          [ORDER BY order-item-commalist]
          [FOR {READ ONLY | UPDATE [OF column-commalist]}]
```

- Erweiterte Positionierungsmöglichkeiten durch SCROLL
  - Cursor-Definition (Bsp.):

```
EXEC SQL DECLARE  C2  SCROLL  CURSOR FOR SELECT ...
```

- Erweitertes FETCH-Statement:

```
EXEC SQL FETCH [[<fetch orientation>] FROM] <cursor>  INTO <target list>

Fetch orientation:  NEXT, PRIOR, FIRST, LAST,
          ABSOLUTE <expression>, RELATIVE <expression>
```

- Beispiele

Aktualisierung mit Bezugnahme auf eine Position

- Wenn die Tupeln, die ein Cursor verwaltet (active set), eindeutig Tupeln einer Relation entsprechen, können sie über Bezugnahme durch den Cursor geändert werden.

```
positioned-update ::= UPDATE  table  SET  update-assignment-commalist
                      WHERE CURRENT OF  cursor

positioned-delete ::= DELETE  FROM  table WHERE CURRENT OF cursor
```

- Beispiel:

```
while  (sqlcode == ok) {
      exec sql fetch  C1  into  :X;
           /* Berechne das neue Gehalt in Z */
      exec sql update  PERS
                set  GEHALT  =  :Z
                where  current  of  C1;
    }
```

- keine Bezugnahme bei INSERT möglich!

Indikatorkonzept

- Indikatorvariablen zum Erkennen von Nullwerten

```
      EXEC SQL FETCH C INTO  :X  INDICATOR  :X_INDIC
bzw. EXEC SQL FETCH C INTO  :X :X_INDIC, :Y :Y_INDIC;
```

- mögliche Werte einer Indikatorvariable

```
    = 0:  zugehörige Wirtsprogrammvariable hat regulären Wert
    = -1: es liegt ein Nullwert vor
    > 0: zugehörige Wirtsprogrammvariable enthält abgeschnittene Zeichenkette
```

- Beispiel

```
exec sql begin declare section;
    int pnummer, mnummer, mind;
exec sql end declare section;
exec sql select MNR into :mnummer :mind from PERS where PNR = :pnummer;
if (mind == 0) { /* kein Nullwert */
else {  /* Nullwert */ }

exec sql insert into PERS (PNR, MNR)
         values (:pnummer, :mnummer indicator :mind);
```


Verwaltung von Verbindungen

- Zugriff auf DB erfordert i.a. zunächst, eine Verbindung herzustellen, v.a. in Client/Server-Umgebungen
  - Aufbau der Verbindung mit CONNECT, Abbau mit DISCONNECT
  - jeder Verbindung ist eine Session zugeordnet
  - Anwendung kann Verbindungen (Sessions) zu mehreren Datenbanken offenhalten
  - Umschalten der "aktiven" Verbindung durch SET CONNECTION

```
CONNECT TO target [AS connect-name] [USER user-name]
```

```
SET CONNECTION  { connect-name | DEFAULT }
```

```
DISCONNECT  { CURRENT | connect-name | ALL }
```

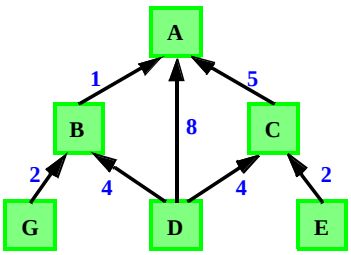

Beispiel: Stücklistenauflösung

- Darstellungsmöglichkeit im RM:

  TEIL (TNR, BEZ, MAT, BESTAND)

  STRUKTUR (OTNR, UTNR, ANZAHL)

Gozinto-Graph



TEIL

TNR	BEZ	MAT	BESTAND
A	Getriebe	-	10
B	Gehäuse	Alu	0
C	Welle	Stahl	100
D	Schraube	Stahl	200
E	Kugellager	Stahl	50
F	Scheibe	Blei	0
G	Schraube	Chrom	100

STRUKTUR

OTNR	UTNR	ANZAHL
A	B	1
A	C	5
A	D	8
B	D	4
B	G	2
C	D	4
C	E	2

- Aufgabe: Ausgabe aller Endprodukte sowie deren Komponenten

Beispiel: Stücklistenauflösung (2)

- max. Schachtelungstiefe sei bekannt (hier: 2)

```
exec sql begin declare section; char T0[10], T1[10], T2[10]; int ANZ;
exec sql end declare section;
exec sql declare C0 cursor for select distinct OTNR from STRUKTUR S1
    where not exists (select * from STRUKTUR S2 where S2.UTNR = S1.OTNR);
exec sql declare C1 cursor for
    select UTNR, ANZAHL from STRUKTUR where OTNR = :T0;
exec sql declare C2 cursor for
    select UTNR, ANZAHL from STRUKTUR where OTNR = :T1;
exec sql open C0;
while (1) {
    exec sql fetch C0 into :T0;
    if (sqlcode == notfound) break;
    printf ("%s\n", T0);
    exec sql open C1;
    while (2) { exec sql fetch C1 into :T1, :ANZ;
                if (sqlcode == notfound) break;
                printf ("  %s: %d\n",T1, ANZ);
                exec sql open C2;
                while (3) {   exec sql fetch C2 INTO :T2, :ANZ;
                              if (sqlcode == notfound) break;
                              printf ("         %s: %d\n",T2, ANZ);    }
                exec sql close C2;  }
    exec sql close C1;  } /* END WHILE */
exec sql close (C0);
```


Dynamisches SQL

- dynamisches SQL: Festlegung von SQL-Anweisungen zur Laufzeit
  -> Query-Optimierung i.a. erst zur Laufzeit möglich

- SQL-Anweisungen werden vom Compiler wie Zeichenketten behandelt
  - Deklaration DECLARE STATEMENT
  - Anweisungen enthalten SQL-Parameter (?) statt Programmvariablen

- 2 Varianten: Prepare-and-Execute bzw. Execute Immediate

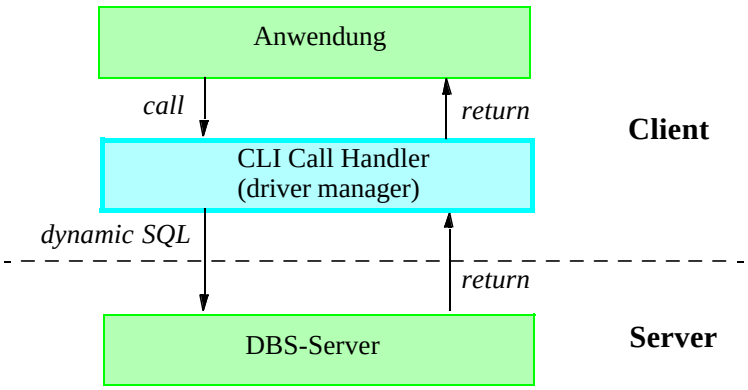
```
exec sql begin declare section;
      char  Anweisung[256], X[6];
exec sql end declare section;
exec sql declare SQLanw statement;
Anweisung = 'DELETE FROM PERS WHERE  ANR = ?'; /*  bzw.  Einlesen  */
exec sql prepare SQLanw from :Anweisung;
exec sql execute SQLanw using 'K51';
scanf(" %s", X);
exec sql execute SQLanw using :X;
```

- bei einmaliger Ausführung EXECUTE IMMEDIATE ausreichend

```
scanf(" %s", Anweisung);
exec sql execute immediate :Anweisung;
```


Call-Level-Interface

- alternative Möglichkeit zum Aufruf von SQL-Befehlen innerhalb von Anwendungsprogrammen
- direkte Aufrufe von Prozeduren/Funktionen einer standardisierten Bibliothek (API)
- Hauptvorteil: keine Präkompilierung von Anwendungen
  - Anwendungen mit SQL-Aufrufen brauchen nicht im Source-Code bereitgestellt zu werden
  - wichtig zur Realisierung von kommerzieller Anwendungs-Software bzw. Tools
- Einsatz v.a. in Client/Server-Umgebungen



Call-Level-Interface (2)

- Unterschiede in der SQL-Programmierung zu eingebettetem SQL
  - CLI impliziert i.a. dynamisches SQL (Optimierung zur Laufzeit)
  - komplexere Programmierung
  - explizite Anweisungen zur Datenabbildung zwischen DBS und Programmvariablen
  - einheitliche Behandlung von mengenwertigen und einfachen Selects (<-> Cursor-Behandlung bei ESQL)
- SQL-Standardisierung des CLI erfolgte 1996 (vorgezogener Teil von SQL99)
  - starke Anlehnung an ODBC
  - Standard-CLI umfaßt über 40 Routinen:
    Verbindungskontrolle, Ressourcen-Allokation, Ausführung von SQL-Befehlen, Zugriff auf Diagnoseinformation, Transaktionsklammerung, Informationsanforderung zur Implementierung
  - JDBC: neuere Variante (s. Kap. 2)
- CLI-Handler verwaltet Verarbeitungskontext für Anwendungen
  - “CLI Descriptor Area”
  - gleichzeitige Bedienung mehrerer Transaktionen/Operationen bzw. Benutzer
- Verwendung von “handles” für Zugriff auf Laufzeitinformationen
  - environment handle
  - connection handle
  - statement handle

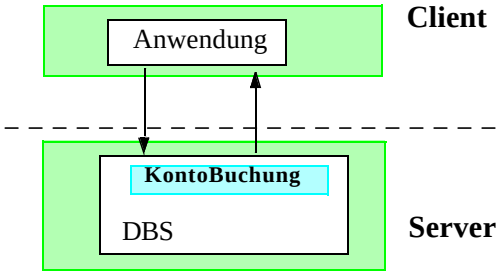
Standard-CLI: Beispiel

```
#include "sqlcli.h"
#include <string.h>
...
{
SQLCHAR* server;
SQLCHAR* uid;
SQLCHAR* pwd;
HENV henv; // environment handle
HDBC hdbc; // connection handle
HSTMT hstmt; // statement handle
SQLINTEGER id;
SQLCHAR name[51];

/* connect to database */
SQLAllocEnv (&henv);
SQLAllocConnect (henv, &hdbc);
if (SQLConnect (hdbc, server, uid,
   pwd, ...)!= SQL_SUCCESS)
   return (print_err (hdbc, ...));

/* create table */
SQLAllocStmt (hdbc, &hstmt);
{ SQLCHAR create[]="CREATE TABLE
   NAMEID (ID integer,
   NAME varchar(50))";
```

```
if (SQLExecDirect (hstmt, create, ...)
   != SQL_SUCCESS)
   return (print_err (hdbc, hstmt));
}
SQLTransact (henv, hdbc, SQL_COMMIT);

/* insert row */
{ SQLCHAR insert[]="INSERT INTO NAMEID
   VALUES (?, ?)";
if (SQLPrepare (hstmt, insert, ...) !=
   SQL_SUCCESS)
   return (print_err (hdbc, hstmt));
SQLBindParam (hstmt, 1, ..., id, ...);
SQLBindParam (hstmt, 2, ..., name,
   ...);
id =500; strcpy (name, "Schmidt");

if (SQLExecute (hstmt) != SQL_SUCCESS)
   return (print_err (hdbc, hstmt));}

SQLTransact (henv, hdbc, SQL_COMMIT);
}
```


Gespeicherte Prozeduren (Stored Procedures)

- Eigenschaften
  - Prozeduren, ähnlich Unterprogrammen einer prozeduralen Programmiersprache, werden durch das DBS gespeichert und verwaltet
  - benutzerdefinierte Prozeduren oder als Systemprozeduren integraler Bestandteil der DB
  - Parameter, Rückgabewerte, Verschachtelungen sind möglich
  - Programmierung der Prozeduren in SQL oder allgemeiner Programmiersprache
- SQL-Erweiterungen erforderlich (u.a. allgemeine Kontrollanweisungen IF, WHILE, etc.)
  - herstellerspezifische Festlegungen seit 1987 (Sybase Transact-SQL)
  - 1996: SQL-Standardisierung zu Persistent Storage Modules (PSM)
- Vorteile:
  - prä-compilierte Ausführungspläne werden gespeichert, sind wiederverwendbar
  - als gemeinsamer Code für verschiedene Anwendungsprogramme nutzbar
  - Anzahl der Zugriffe des Anwendungsprogramms auf die DB werden reduziert
  - Performance-Vorteile v.a. in Client-Server-Umgebungen
  - höherer Grad der Isolation der Anwendung von DB wird erreicht



- Nachteile ?

Persistente SQL-Module (PSM)

- vorab (1996) fertiggestellter Teil des SQL99-Standards
- Module sowie Routinen sind Schema-Objekte (wie Tabellen etc.) und können im Katalog aufgenommen werden (beim DBS/Server)
  - Module enthalten Routinen: Prozeduren und Funktionen
  - Routinen in SQL (SQL routine) oder in externer Programmiersprache (external routine) geschrieben
  - zusätzliche CREATE- und DROP-Anweisungen für Module und Routinen
- Beispiel:

```
CREATE MODULE b1 LANGUAGE SQL

CREATE PROCEDURE KontoBuchung (konto INTEGER, betrag DECIMAL (15,2));
BEGIN        DECLARE C1 CURSOR FOR ...;

             UPDATE account SET balance = balance + betrag
             WHERE account_# = konto;
...
END;
DECLARE EXTERNAL sinus (FLOAT)  RETURNS FLOAT LANGUAGE FORTRAN;
END MODULE;
```


PSM (2)

- **externe Module** in beliebiger Programmiersprache (C, PASCAL, FORTRAN, ...)
  - Nutzung bestehender Bibliotheken, Akzeptanz
  - 2 Sprachen / Typsysteme, Typkonversion, Typüberwachung außerhalb von SQL
- **SQL-Routinen**: in SQL geschriebene Prozeduren/Funktionen
  - Deklarationen lokaler Variablen etc. innerhalb der Routinen
  - Nutzung zusätzlicher Kontrollanweisungen: Zuweisung, Blockbildung, IF, LOOP, etc.
  - Exception Handling (SIGNAL, RESIGNAL)
  - integrierte Programmierumgebung, keine Typkonversionen
- Prozeduren werden über CALL-Anweisung aufgerufen:

```
EXEC SQL CALL KontoBuchung  (:account_#, :balance);
```

PSM (3)

- Beispiel einer SQL-Funktion:

```
CREATE FUNCTION kontostand (konto INTEGER)  RETURNS DECIMAL (15,2);
 BEGIN          DECLARE ks  INTEGER;

                SELECT balance INTO ks
                FROM    account
                WHERE account_#  = konto;

               RETURN ks;
 END;
```

- Aufruf persistenter Funktionen (SQL und externe) in SQL-Anweisungen wie Built-in-Funktionen

```
SELECT  *
FROM account
WHERE
```

- Prozedur- und Funktionsaufrufe können rekursiv sein

Prozedurale Spracherweiterungen: Kontrollanweisungen

Compound Statement	BEGIN ... END;
SQL-Variablendeklaration	DECLARE var type;
If-Anweisung	IF condition THEN ... ELSE ... :
Case-Anweisung	CASE expression WHEN x THEN ... WHEN ... :
Loop-Anweisung	WHILE i < 100 LOOP ... END LOOP;
For-Anweisung	FOR result AS ... DO ... END FOR;
Leave-Anweisung	LEAVE ...;
Prozeduraufruf	CALL procedure_x (1, 2, 3);
Zuweisung	SET x = “abc”;
Return-Anweisung	RETURN x;
Signal/Resignal	SIGNAL division_by_zero;

Beispiel

```
outer: BEGIN
    DECLARE account INTEGER DEFAULT 0;
    DECLARE balance DECIMAL (15,2);
    DECLARE no_money EXCEPTION FOR SQLSTATE VALUE 'xxxxx';
    DECLARE DB_inconsistent EXCEPTION FOR SQLSTATE VALUE 'yyyyy';

    SELECT account_#, balance INTO account, balance FROM accounts ...;
    IF (balance - 10) < 0 THEN SIGNAL no_money;
    BEGIN ATOMIC
      DECLARE cursor1 SCROLL CURSOR ...;
      DECLARE balance DECIMAL (15,2);
      SET balance = outer.balance - 10;
      UPDATE accounts SET balance = balance WHERE account_# = account;
      INSERT INTO account_history VALUES (account, CURRENT_DATE, 'W', balance); .....
    END;
    EXCEPTION
      WHEN no_money THEN
       CASE (SELECT account_type FROM accounts WHERE account_# = account)
            WHEN 'VIP' THEN INSERT INTO send_letter ....
            WHEN 'NON-VIP' THEN INSERT INTO blocked_accounts ...
       ELSE SIGNAL DB_inconsistent;

      WHEN DB_inconsistent THEN
            BEGIN .... END;

      END;
```


Zusammenfassung

- Cursor-Konzept zur satzweisen Verarbeitung von Datenmengen
  - Operationen: DECLARE CURSOR, OPEN, FETCH, CLOSE
  - Erweiterungen: Scroll-Cursor, Sichtbarkeit von Änderungen
- Statisches (eingebettetes) SQL
  - hohe Effizienz
  - gesamte Typprüfung und Konvertierung erfolgen durch Präcompiler
  - relativ einfache Programmierung
  - Aufbau aller SQL-Befehle muß zur Übersetzungszeit festliegen
  - es können nicht dynamisch zur Laufzeit verschiedene Datenbanken angesprochen werden
- Dynamisches SQL: hohe Flexibilität, schwierigere Programmierung, geringere Effizienz
- Call-Level-Interface
  - keine Nutzung eines Präcompilers
  - Einsatz v.a. in Client-Server-Systemen
  - SQL-Standardisierung erfolgte 1996 (stark an ODBC angelegt)
- Stored Procedures: Performance-Gewinn durch reduzierte Häufigkeit von DBS-Aufrufen
  - SQL-Standardisierung: Persistent Storage Modules (PSM)
  - umfassende prozedurale Spracherweiterungen von SQL

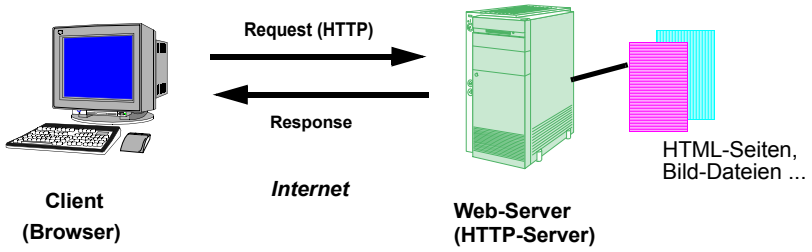
2. Web-Anbindung von Datenbanken

- Einführung: Architekturvarianten für Web-Informationssysteme
- JDBC und SQLJ
- Client-seitige DB-Zugriffe
- Server-seitige DB-Anbindung
  - CGI-Kopplung
  - Servlets
  - Java Server Pages (JSP)
  - PHP-Skripte

Architektur-Varianten für Web-Informationssysteme

- im Web verfügbare Daten liegen noch oft in einfachen Dateien
  - statisch festgelegter Informationsgehalt
  - relativ einfache Datenbereitstellung (HTML)
  - einfache Integration von Multimedia-Objekten (Bild, Video, ...) sowie externen Quellen
  - Aktualitätsprobleme für Daten und Links
  - oft Übertragung wenig relevanter Daten, hohe Datenvolumen . . .

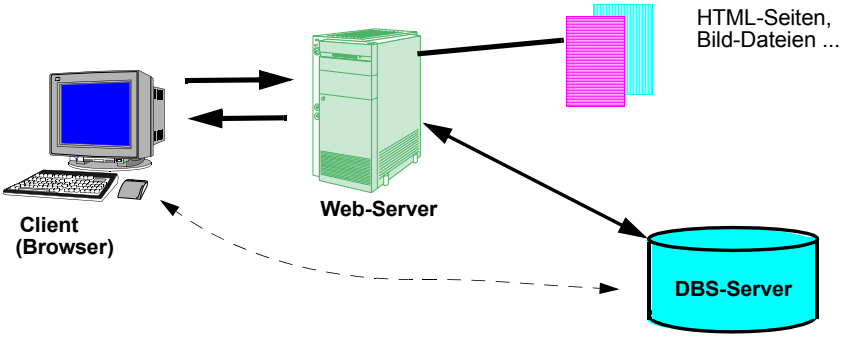
Statische WebIS



- Zustandslosigkeit von HTTP erfordert Zusatzmaßnahmen für Mehrschritt-Interaktionen / Sitzungskontrolle
  - sukzessives Abholen großer Ergebnismengen, Verwaltung von Einkaufskörben, ...
  - Verwaltung von Session-/Benutzer-IDs, Nutzung von Cookies etc.

Architektur-Varianten (2)

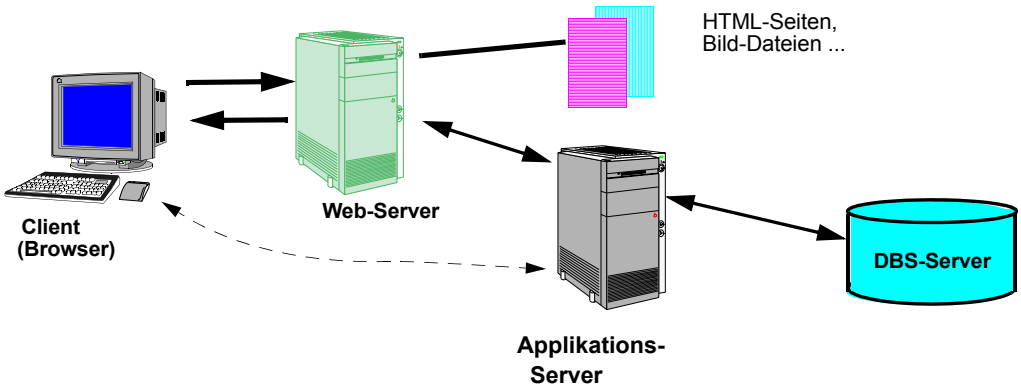
WebIS
mit
DB-Anbindung



- Anbindung von Datenbanksystemen
  - dynamischer Informationsgehalt durch Zugriff auf Datenbanken (existierend oder neu)
  - Bereitstellung aktueller Informationen
  - geringerer Datenumfang
  - bessere Skalierbarkeit
  - Mehrbenutzerfähigkeit (auch bei Änderungen) . . .
- Verwaltung/Speicherung von HTML/XML-Seiten, Dokumenten, Multimedia-Daten etc. durch DBS
  - Content Management / Website Management
  - hohe Flexibilität (erweiterte Suchmöglichkeiten, ...), bessere Konsistenzwahrung für Links ...

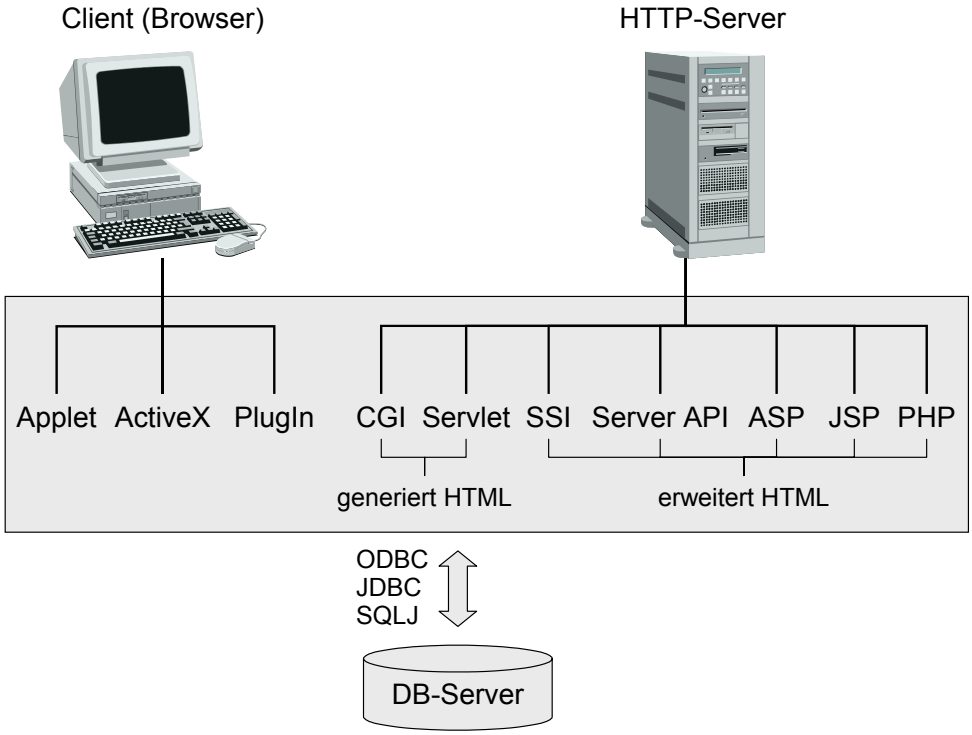
Architektur-Varianten (3)

Applikations-
orientierte
WebIS



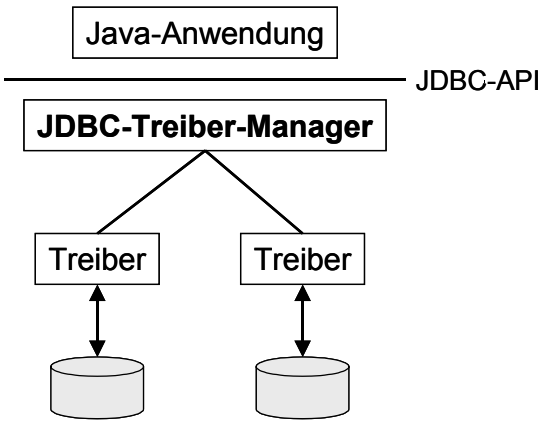
- Vorteile der Applikations-Server-Architektur
  - Unterstützung komplexer (Geschäfts-) Anwendungen
  - Skalierbarkeit (mehrere Applikations-Server)
  - Einbindung mehrerer DBS-Server / Datenquellen
  - Transaktions-Verwaltung
  - ggf. Unterstützung von Lastbalancierung, Caching, etc.

Übersicht Techniken zur Web-Anbindung von Datenbanken



JDBC (Java Database Connectivity)†

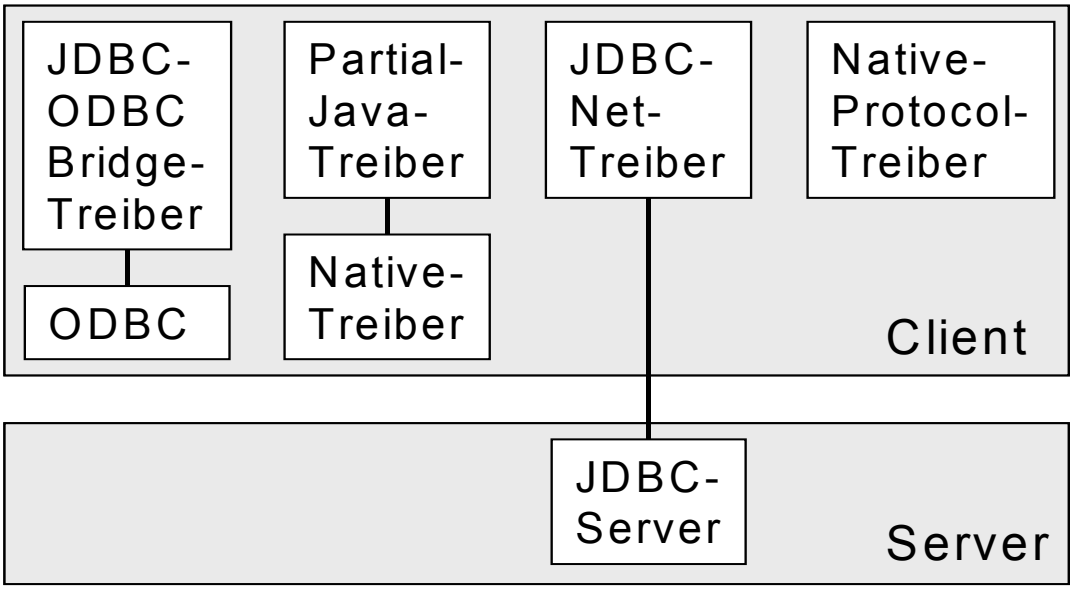
- Standardschnittstelle für den Zugriff auf SQL-Datenbanken unter Java
- basiert auf dem SQL/CLI (call-level-interface)
- Grobarchitektur



- durch Auswahl eines anderen JDBC-Treibers kann ein Java-Programm ohne Neuübersetzung auf ein anderes Datenbanksystem zugreifen

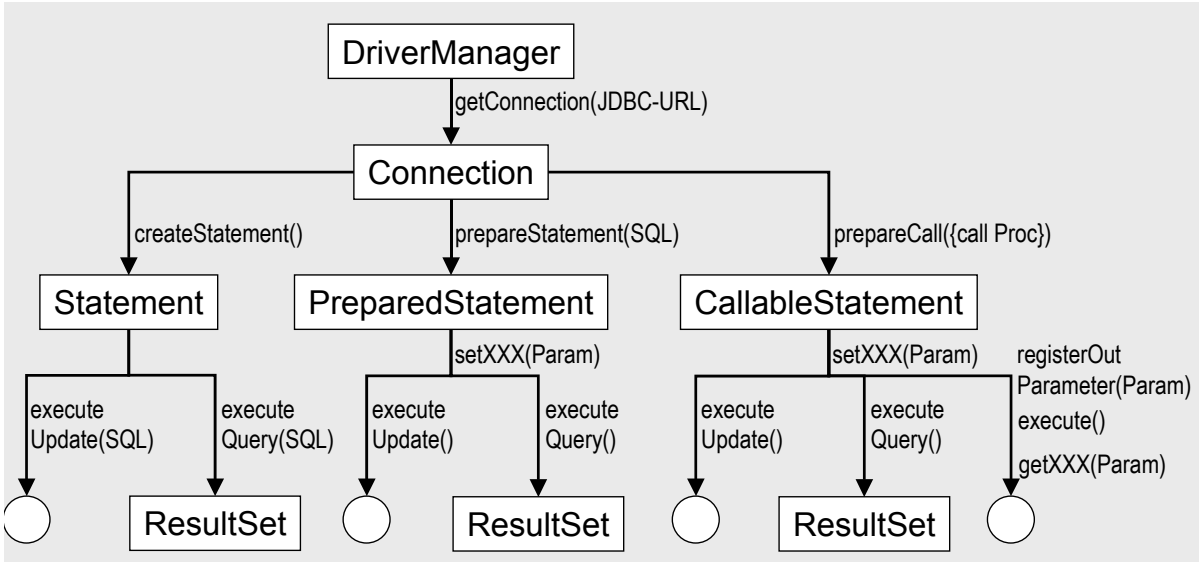
† http://java.sun.com/jdbc/

JDBC: Treiber-Arten



JDBC-Klassen

- streng typisierte objekt-orientierte API
- Aufrufbeziehungen (Ausschnitt)



JDBC: Beispiel

- Beispiel: Füge alle Matrikelnummern aus Tabelle ‘Student’ in eine Tabelle ‘Statistik’ ein.

```
import java.sql.*;
...
public void copyStudents() {

 String drvClass = “COM.ibm.db2.jdbc.net.DB2Driver“;
 try {
  // lade den JDBC-Treiber (zur Laufzeit)
  Class.forName(drvClass);
 } catch (ClassNotFoundException e) {// Fehlerbehandl.}

 try {
  // über die URL wird der passende JDBC-Treiber
  // herausgesucht, der Verbindungsdaten erhält
  String url = “jdbc:db2://host:6789/myDB”
  Connection con = DriverManager.getConnection
                    (url, “login”, “password”);

  String query = “SELECT matrikel FROM student”;
  String insert = “INSERT INTO statistik (matrikel) “ +
                “VALUES ( ? )“;
  // Ausführen von Queries mit Statement-Objekt
  Statement stmt = con.createStatement();
  // für wiederholte Ausführung Prepared-Stmts
  PreparedStatement pStmt =
                con.prepareStatement(insert);
  // führe Query aus
  ResultSet rs = stmt.executeQuery(query);
  // lese die Ergebnisdatensätze aus
  while (rs.next()) {
   String matrikel = rs.getString(1);
   // setze den Parameter und führe Insert-Operation aus
   pStmt.setString (1, matrikel);
   pStmt.executeUpdate();
  }
 } catch (SQLException e) { // Fehlerbehandlung}
}
```


JDBC: Transaktionskontrolle

- **Transaktionskontrolle** durch Methodenaufrufe der Klasse `Connection`
  - `setAutoCommit`: Ein-/Abschalten des Autocommit-Modus (jedes Statement ist eigene Transaktion)
  - `setReadOnly:` Festlegung ob lesende oder ändernde Transaktion
  - `setTransactionIsolation`: Festlegung der Synchronisationsanforderungen (None, Read Uncommitted, Read Committed, Repeatable Read, Serializable)
  - `commit` bzw. `rollback`: erfolgreiches Transaktionsende bzw. Transaktionsabbruch

- Beispiel

```
try {
    con.setAutoCommit (false);
        // einige Änderungsbefehle, z.B. Inserts
    con.commit ();
  } catch (SQLException e) {
        con.rollback ();
  }
```


Erweiterungen in JDBC 2, JDBC 3

- Erweiterungen von JDBC 2
  - erweiterte Funktionalität von ResultSets (Scroll-Unterstützung; Änderbarkeit)
  - Batch Updates
  - Unterstützung von SQL99-Datentypen (BLOB, CLOB, ARRAY, REF, UDTs)
  - Optionales Package (javax.sql) mit Funktionen für Server-seitige Programme
    - JNDI (Java Naming and Directory Interface)-Unterstützung zur Spezifikation von Datenquellen
    - Connection Pooling
    - Verteilte Transaktionen (2-Phasen-Commit-Protokoll)

- Erweiterungen von JDBC 3
  - Package javax.sql ist nicht mehr optional
  - Unterstützung von Savepoints in Transaktionen
  - neue Datentypen BOOLEAN, DATALINK (Referenz auf externe Daten)
  - Zugriff auf Metadaten von UDTs und UDFs

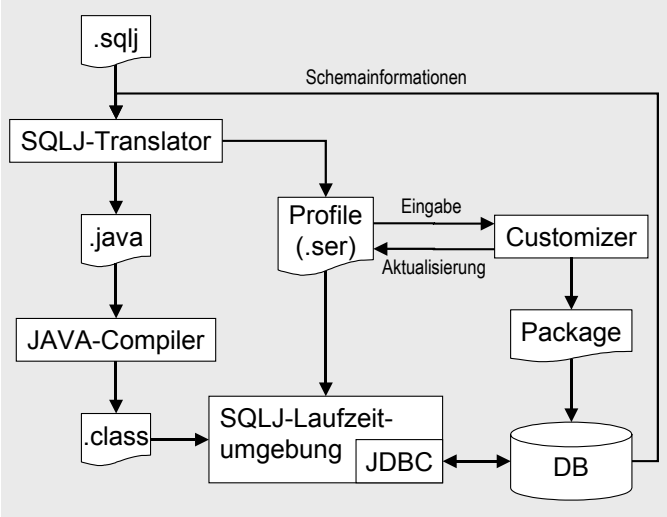
SQLJ‡

- Eingebettetes SQL (Embedded SQL) für Java
  - direkte Einbettung von SQL-Anweisungen in Java-Code
  - statische SQL-Anweisungen können zur Übersetzungszeit überprüft werden (syntaktisch und semantisch)
  - SQLJ-Programme müssen mittels Präprozessor in Java-Quelltext transformiert werden

- Spezifikation besteht aus
  - Embedded SQL für Java (Teil 0), *SQL Object Language Binding (OLB)*
  - Java Stored Procedures (Teil 1)
  - Java-Klassen für UDTs (Teil 2)

- Vorteile
  - Syntax- und Typprüfung zur Übersetzungszeit
  - Vor-Übersetzung (Performance)
  - einfacher/kompakter als JDBC
  - streng typisierte Iteratoren (Cursor-Konzept)

- Nachteile
  - geringere Flexibilität (Operationen müssen zur Übersetzungszeit feststehen)



‡ http://www.sqlj.org

SQLJ (2)

- eingebettete SQL-Anweisungen: **#sql** [[<context>]] { <SQL-Anweisung> }
  - beginnen mit **#sql** und können mehrere Zeilen umfassen
  - können Variablen der Programmiersprache (:x) bzw. Ausdrücke (:y + :z) enthalten
  - können Default-Verbindung oder explizite Verbindung verwenden

- Vergleich SQLJ - JDBC (1-Tupel-Select)

SQLJ

```
#sql [con]{ SELECT name INTO :name
            FROM student WHERE matrikel = :mat};
```

JDBC

```
java.sql.PreparedStatement ps =
      con.prepareStatement („SELECT name "+
                 „FROM student WHERE matrikel = ?");
ps.setString (1, mat);
java.sql.ResultSet rs = ps.executeQuery();
rs.next()
name= rs.getString(1);
rs.close;
```

- Iteratoren zur Realisierung eines Cursor-Konzepts
  - benannte Iteratoren: Zugriff auf Spalten des Ergebnisses über Methode mit dem Spaltennamen
  - Positionsiteratoren: Zugriff über FETCH-Anweisung und Host-Variablen

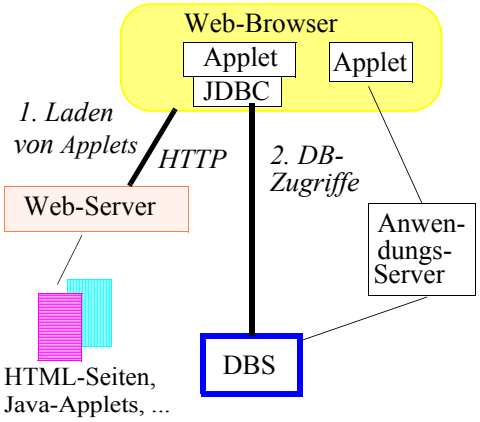
SQLJ (3)

- Beispiel: Füge alle Matrikelnummern aus Tabelle 'Student' in eine Tabelle 'Statistik' ein.

```
import java.sql.*;
import sqlj.runtime.ref.DefaultContext;
...
public void copyStudents() {
 String drvClass = "COM.ibm.db2.jdbc.net.DB2Driver";
 try {
  Class.forName(drvClass);
 } catch (ClassNotFoundException e) { // errorlog }

 try {
  String url = "jdbc:db2://host:6789/myDB"
  Connection con = DriverManager.getConnection
                   (url, "login", "password");
  // erzeuge einen Verbindungskontext
  // (ein Kontext pro Datenbankverbindung)
  DefaultContext ctx = new DefaultContext(con);
  // definiere Kontext als Standard-Kontext
  DefaultContext.setDefaultContext(ctx);
  // deklariere Typ für benannten Iterator
  #sql public iterator MatrikelIter (String matrikel);
  // erzeuge Iterator-Objekt
  MatrikelIter mIter;
  // Zuordnung und Aufruf der SQL-Anfrage
  #sql mIter = { SELECT matrikel FROM student };
  // navigiere über der Ergebnismenge
  while (mIter.next()) {
   // füge aktuelles Ergebnis in Tabelle Statistik ein
   #sql {INSERT INTO statistik (matrikel)
         VALUES ( mIter.matrikel()) };
  }
  mIter.close();
 } catch (SQLException e) { // errorlog }
}
```

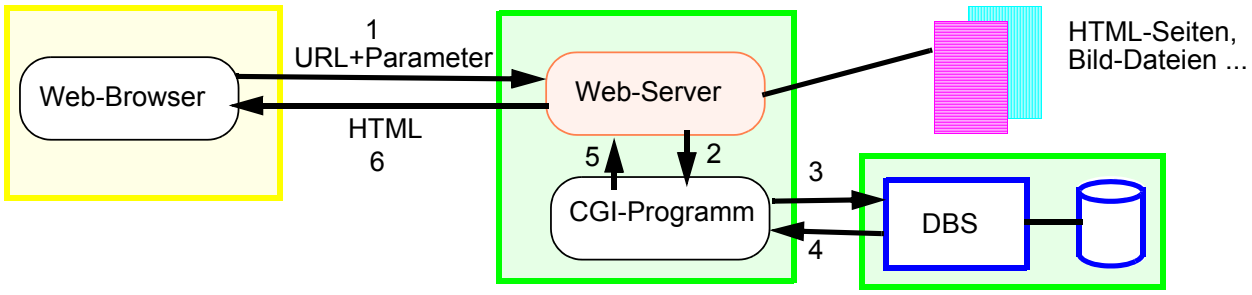

Client-seitige Web-Anbindung von Datenbanken

- Nutzung von Java-Applets
  - dynamisches Laden vom Web-Server (HTTP-Server)
  - Eingabevalidierung und Datenaufbereitung auf Client-Seite
  - Anwendungslogik auf Client-Seite (Applet) oder ausgelagert
  - direkter DB-Zugriff (ohne Web-Server) z.B. über JDBC
  - hohe Portabilität durch Java
  - Kontext für Mehrschritt-Interaktionen im Applet
  - Beschränkungen aufgrund von Sicherheitsproblemen
- Alternative zu Applets: ActiveX Controls (z.B. in Visual Basic)



Server-seitige Web-Anbindung: CGI-Kopplung

- CGI: Common Gateway Interface
  - plattformunabhängige Schnittstelle zwischen Web-Server (HTTP-Server) und externen Anwendungen
  - wird von jedem Web-Server unterstützt



- CGI-Programme (z.B. realisiert in Perl, C, Shell-Skripte)
  - erhalten Benutzereingaben (aus HTML-Formularen) vom Web-Server als Parameter
  - können beliebige Berechnungen vornehmen und auf Datenbanken zugreifen
  - Ergebnisse werden als dynamisch erzeugte HTML-Seiten an Client geschickt

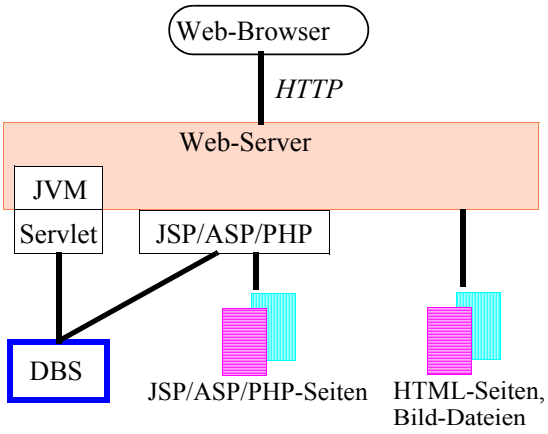
CGI-Kopplung (2)

- Anfangszeit: Selbst-Programmierung von CGI-Programmen
  - pro Aufgabe / Anfrage eigenes Programm
  - aufwendige / umständliche Programmierung
  - kompakter Programmumfang
- Tool-Unterstützung
  - reduzierter Programmieraufwand
  - i.a. 1 generisches CGI-Programm
  - DB-Anweisungen in HTML-Seiten eingebettet oder Festlegung über Makro-Dateien
- mögliche Performance-Probleme
  - Eingabefehler werden erst im CGI-Programm erkannt
  - für jede Interaktion erneutes Starten des CGI-Programms
  - für jede Programmaktivierung erneuter Aufbau der DB-Verbindung

```
#!/bin/perl
use Msql;
# Seitenkopf ausgeben:
print"Content-type: text/html\n\n";
# [...]
# Verbindung mit dem DB-Server herstellen:
$testdb = Msql->connect;
$testdb->selectdb("INFBIBLIOTHEK");
# DB-Anfrage
$q = $testdb->query  ("select Autor, Titel  from ...");
# Resultat ausgeben:
print"<TABLE BORDER=1>\n"; print"<TR>\n
   <TH>Autor<TH>Titel</TR>"; $rows = $q -> numrows;
while ($rows>0) {
     @sqlrow = $q->fetchrow;
     print "<tr><td>",@sqlrow[0],"</td><td>",
                    @sqlrow[1],</td></tr>\n";
      $rows--; }
print"</TABLE>\n";
# Seitenende ausgeben
# [...]
```


Server-seitige Web-Anbindung: weitere Ansätze

- Integration von CGI-Programmen in Web-Server
  - proprietäre API-Lösungen: NSAPI (Netscape), ISAPI (Microsoft)
  - Einsatz spezieller DB-Web-Server (Oracle, ...) mit direktem DB-Zugriff
  - Server Side Includes (SSI): Auswertung spezieller HTML-Erweiterungen
  - kein Starten eigener CGI-Prozesse, DB-Verbindungen können offen bleiben
- Einsatz von Java-Servlets
  - herstellerunabhängige Erweiterung von Web-Servern (Java Servlet-API)
  - erfordert Integration einer Java Virtual Machine (JVM) im Web-Server -> Servlet-Container
- server-seitige Erweiterung von HTML-Seiten um Skript-/Programmlogik
  - Java Server Pages
  - Active Server Pages (Microsoft-Lösung)
  - PHP-Anweisungen
- Integration von Auswertungslogik in DB-Prozeduren (stored procedures)



Servlets

- spezielle Java-Klassen, die in Servlet-Container ausgeführt werden
  - besondere Unterstützung von HTTP-Anfragen
  - Zugriff auf das gesamte Java-API, insbesondere auch auf JDBC-Klassen für DB-Zugriff
  - plattformunabhängig, serverunabhängig

Employee #1

Name: Jaime Husmillo
Address: 2040 Westlake N
City/State/Zip: Seattle, WA 98109

- Servlet-Container für viele Web- und Application-Server verfügbar (Apache, iPlanet, IIS, IBM WebSphere, Tomcat ...)

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class EmpServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {
      response.setContentType ("text/html");
      PrintWriter out = response.getWriter();
      out.println("<HTML><BODY>\n" +
                  "<H1>Employee #1</H1>\n");
      try {// oeffne Datenbankverbindung
        Connection con = DriverManager.getConnection(
                  "jdbc:db2:myDB", "login", "password");
        // erzeuge Statement-Objekt
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM "
                  + „Employees WHERE EmployeeID = 1";);
        if (rs.next()) {
        out.println("<B>Name:</B> "+ rs.getString("Name") + "<BR>"
          out.println("<B>Address:</B><BR>");
          out.println("<B>City/State/ZIP:</B> " + rs.getString("City
                  + " " + rs.getString("State") +
                    " " + rs.getString("ZipCode"));
          }
          rs.close();
          stmt.close();
          con.close();
        }
      catch (SQLException e) { log(e.getMessage());
        }
        out.println ("</BODY></HTML>");
      }
    }
```


Java Server Pages (JSP)

- Entwurf von dynamischen HTML-Seiten mittels HTML-Templates und XML-artiger Tags
- Trennung von Layout und Applikationslogik durch Verwendung von Java-Beans
- Erweiterbar durch benutzerdefinierte Tags (z.B. für DB-Zugriff, Sprachlokalisierung, ...)
- JSP-Prozessor oft als Servlet realisiert
  - JSP-Seite wird durch JSP-Prozessor in ein Servlet übersetzt
  - JSP kann überall eingesetzt werden, wo ein Servlet-Container vorhanden ist

JSP-Seite:

```
<HTML>
 <BODY>
  <jsp:useBean id="EmpData" class="FetchEmpDataBean"
                scope="session">
    <jsp:setProperty name="EmpData"
                     property="empNumber" value="1" />
  </jsp:useBean>
  <H1>Employee #1</H1>
  <B>Name:</B> <%=EmpData.getName()%><BR>
  <B>Address:</B> <%=EmpData.getAddress()%><BR>
  <B>City/State/Zip:</B>
  <%=EmpData.getCity()%>,
  <%=EmpData.getState()%>
  <%=EmpData.getZip()%>
 </BODY>
</HTML>
```

Bean:

```
class FetchEmpDataBean {
  private Stringname, address, city, state, zip;
  private int empNumber = -1;

  public void setEmpNumber(int nr) {
    empNumber = nr;
    try {
      Connection con =DriverManager.getConnection("jdbc:db2:myDB", "login", "pwd");
      Statement stmt = con.createStatement();
      ResultSet rs = stmt.executeQuery(
          "SELECT * FROM Employees WHERE EmployeeID=" + nr);
      if (rs.next()) {
        name = rs.getString("Name"); address = rs.getString("Address");
        city = rs.getString("City"); state = rs.getString("State"); zip = rs.getString("ZipCode"
      }
      rs.close(); stmt.close(); con.close();
    } catch (SQLException e) { //...}
  }
  public String getName() { return name; }
  public String getAddress() { return address; } ...
```

Vergleich Servlets - JSP

- Gemeinsamkeiten
  - plattformunabhängig, serverunabhängig
  - Unterstützung von Session-Konzepten
  - Zugriff auf gesamtes Java-API
  - Datenbankzugriff auf alle Datenbanken, für die ein JDBC-Treiber existiert
- Unterschiede
  - Servlet:
    - Layout (HTML) muß in Applikationslogik erzeugt werden
    - nur für Web-Seiten mit geringem Anteil an HTML-Code oder zur Erzeugung von nicht-HTML-Daten geeignet
  - JSP:
    - direkte Verwendung von HTML-Tags
    - Logik kann in Beans ausgelagert werden
    - getrennte Entwicklung von Web-Design und Datenverarbeitung (z.B. Datenbankschnittstelle) möglich

PHP (PHP: Hypertext Preprocessor)

- Open Source Skriptsprache, angelehnt an C, Java und Perl, zur Einbettung in HTML-Seiten
- besonders effizient bei der Erzeugung und Auswertung von HTML-Formularen
- Funktionsbibliothek durch Vielzahl von Modulen erweiterbar (meist frei erhältlich)
- Funktionen zum Datenbankzugriff sind DBS-spezifisch; DBS-unabhängiger Zugriff über ODBC oder Abstraktionsmodule (z.B. dbx, Pear::DB) möglich

```
<html>
 <body>
  <h1>Employee #1</h1>
<?php
        $con = dbx_connect(DBX_PGSQL, "host", "myDB", "login", "password")
                or die("Verbindungsfehler!</body></html>");
        $result = dbx_query($con,
                                "SELECT * FROM Employees WHERE EmployeeID = 1");
        if ( is_object($result) and ($result->rows > 0) ) {
?>
  <b>Name:</b>          <?php echo $result->data[0]["Name"] ?><br>
  <b>Address:</b>       <?php echo $result->data[0]["Address"] ?><br>
  <b>City/State/ZIP:</b> <?php echo $result->data[0]["City"] . ", " . $result->data[0]["State"] . " " . $result->data[0]["ZipCode"]
                        ?><br>
<?php   } else { echo "Unknown employee!"; } ?>
 </body>
</html>
```

Employee #1

Name: Jaime Husmillo
Address: 2040 Westlake N
City/State/Zip: Seattle, WA 98109

Vergleich JSP-PHP

- Gemeinsamkeiten
  - serverseitige Skriptsprachen zur Einbindung in HTML-Seiten
  - setzen HTTP-Servermodul mit JSP/PHP Laufzeitumgebung voraus
  - Seiten müssen gelesen und interpretiert werden
- JSP
  + plattformunabhängig
  + Programmierung erfolgt in Java (keine neue Programmiersprache)
  + Zugriff auf alle Java-APIs und Java-Klassen
  + unterstützt Trennung von Layout und Programmlogik
  + einheitlicher Zugriff auf praktisch alle relationalen DBS über JDBC
  - großer Ressourcenbedarf für Java-Laufzeitumgebung
- PHP
  + typfreie Variablen und dynamische Arraystrukturen vereinfachen Web-Programmierung
  + Automatismen zur Verarbeitung von Formularfeldern
  + Fehlertolerant insbesondere bei der Arbeit mit Variablen
  + enthält große Anzahl von Modulen (z.B. für Datenkomprimierung, Bezahldienste, XML-Verarbeitung)
  - unterstütze DB-Funktionalität abhängig von Datenbank; DBS-spezifische Funktionen
  - umfangreiche Programmlogik muss als externes Modul (meist in C, C++) realisiert werden

Zusammenfassung

- Notwendigkeit dynamisch erzeugter HTML-Seiten mit Zugriff auf Datenbanken
- breite Verwendung von Java-Technologien
  - JDBC: Standardansatz für DB-Zugriff mit Java
  - SQLJ/SQL OLB: eingebettetes SQL für Java
- Anwendungslogik mit DB-Zugriffen auf Client-Seite (z.B. Java Applets) oder auf Server-Seite
- server-seitige Programmierschnittstellen: CGI, ASP, JSP, PHP ...
- CGI
  - Standardisierte Schnittstelle, die von allen HTTP-Server unterstützt wird
  - pro Interaktion erneutes Starten des CGI-Programms sowie Aufbau der DB-Verbindung notwendig
  - keine Unterstützung zur Trennung von Layout und Programmlogik
- Java Server Pages (JSP), Active Server Pages (ASP), PHP
  - Einbettung von Programmcode in HTML-Seiten
  - Programmlogik kann in externe Komponenten (z.B. Beans, COM) ausgelagert werden
  - JSP: Verwendung von Java (Servlets) mit Datenbankzugriff über JDBC, SQLJ
  - ASP: auf Microsoft-Server-Plattformen beschränkt; mehrere Programmiersprachen; Datenbankzugriff über ADO (Active Data Objects, basiert auf OLE/DB)
- komplexe Anwendungen erfordern Einsatz von Applikations-Servern

3. Neuere DB-Anwendungsgebiete

- Einleitung
- Multimedia-Anwendungen
- Information Retrieval / Digitale Bibliotheken
- Geoinformationssysteme, CAD-Datenbanken
- Bio-Datenbanken
- Wissensbasierte Anwendungen
- Beschränkungen des Relationenmodells
  - Beispiel: Modellierung von 3D-Objekten

DBS-Einsatz in Unternehmen

- Einsatz in “kommerziellen” Anwendungen
  - Produktionsplanung und -steuerung
  - Lagerverwaltung
  - Personalverwaltung ...
- als Kern von Transaktionssystemen (OLTP, Online Transaction Processing)) für die interaktive Auskunftsbearbeitung, Buchung, Datenerfassung
- meist einfache Verhältnisse
  - überschaubare Anzahl von Relationen
  - nur wenige, einfache Datentypen
  - einfache Operationen
  - einfache und kurze Transaktionen
  - hohe Transaktionsraten
- Entscheidungsunterstützung / Decision Support
  - komplexe Auswertungen; OLAP (Online Analytical Processing) und Data Mining
  - oft auf separater und aus mehreren Datenquellen konsolidierter Datenbank (Data Warehouse)

Anspruchsvolle DB-Anwendungen

- zunehmende Bandbreite von Anwendungen mit unterschiedlichsten Anforderungen

- Anwendungsklassen
  - multimediale Anwendungen
  - Web-Informationssysteme, z.B. digitale Bibliotheken
  - Geographische Informationssysteme (GIS)
  - Bioinformatik: Gen-Datenbanken, Protein-Datenbanken, ...
  - Ingenieur-Anwendungen (CAD/CAM), VLSI-Entwurf
  - . . .

- Aufgaben/Eigenschaften
  - große Datenmengen
  - Verwaltung und Handhabung von komplexen Objekten
  - Gewährleistung komplexer Integritätsbedingungen
  - Operationen / Schnittstelle zugeschnitten auf Anwendungsklasse
  - Unterstützung von Zeit/Versionen
  - hohe Leistungsanforderungen . . .

Multimedia-Anwendungen

- Medien: Text, Bilder, Grafik, Ton, Audio, Video, ...

- großer Datenumfang: Einsatz von Kompressionstechniken obligatorisch

Medium	Format	Datenumfang	Transferrate
Text	ASCII	1 MB / 500 Seiten	2 KB / Seite
SW-Bilder	G3/4-FAX	32 MB / 500 Bilder	64 KB / Bild
Farbbilder	GIF, TIFF JPEG	1.6 GB / 500 Bilder 0.2 GB / 500 Bilder	3.2 MB / Bild 0.4 MB / Bild
Sprache	MPEG audio	2.4 MB / 5 Min.	8 KB/sec
CD-Musik	CD	52.8 MB/5 Min.	176 KB/sec
Standard-Video	PAL	6.6 GB/5 Min.	22 MB/sec
Qualitätsvideo	HDTV	33 GB/5 Min.	110 MB/sec

- besondere Schwierigkeiten durch zeitabhängige Medien: Sprache, Audio, Video, Animation
  - zeitliche Synchronisation
  - Koordinierung von Ton und Bild bei Video
  - zeitbehaftete Operationen: Playback, Record, Fast-Forward, ...

- Verwaltung der Multimedia-Daten innerhalb von Dateien oder im DBS

Verwaltung von Multimedia-Objekten innerhalb von DBS?

- Nachteile
  - aufwendiger als Einsatz von Dateien
  - Performance-Nachteile für Speicherung und Lesen einzelner Objekte möglich
  - viele Programme/Tools arbeiten auf Dateibasis
- Vorteile
  - gleichartige Behandlung von einfachen Daten und Multimedia-Objekten
  - Transaktionsunterstützung (Mehrbenutzerbetrieb, automatischen Recovery, Integritätsbedingungen)
  - Vermeidung von Redundanz
  - Geräteunabhängigkeit (CD-ROM, optische Platten etc.)
  - Optimierung der Performance durch System • • •
- meist Einsatz von BLOBs (Binary Large Objects) bzw. "long fields"
  - Speichern von großen, unstrukturierten Bytefolgen (z.B. Rasterbilder)
  - i.a. nur einfache Operationen erforderlich
- Alternative: DBS mit Multimedia-Datentypen und spezifischen Operationen

Information-Retrieval-Systeme (IRS)

- zahlreiche Einsatzmöglichkeiten: Bibliotheken, Literaturrecherche, Informationsdienste (Chemie, Recht, Patent-DB, ...), Web-Dokumente ...
- Aufgaben/Eigenschaften
  - Verwaltung von Dokumenten, Büchern, Abstracts usw.
  - effiziente Suche in großen Datenmengen
  - typischerweise nur Retrieval im Mehrbenutzerbetrieb
  - Anfragesprache für Retrieval: Annäherung an natürliche Sprache wünschenswert
- Dokumente in IRS: unstrukturierte Daten (Texte), keine dem IRS bekannte Dokumentsstruktur
- Indexierung bei unstrukturierten Daten
  - Invertierung des gesamten Textes eines Dokumentes
  - was sind geeignete Suchbegriffe/Schlüsselwörter?

Information-Retrieval-Systeme (2)

- Art der Suche
  - Anfragen relativ unscharf
  - Mehrdeutigkeit: Synonym-, Homonymproblem
  - Ähnlichkeitssuche (nearest neighbor, best match, pattern matching usw.)
  - Ergebnisbewertung: Relevanzproblem (Precision, Recall)

N: Menge aller Dokumente
G: Menge gefundener Dokumente
R: Menge relevanter Dokumente

P =

R =

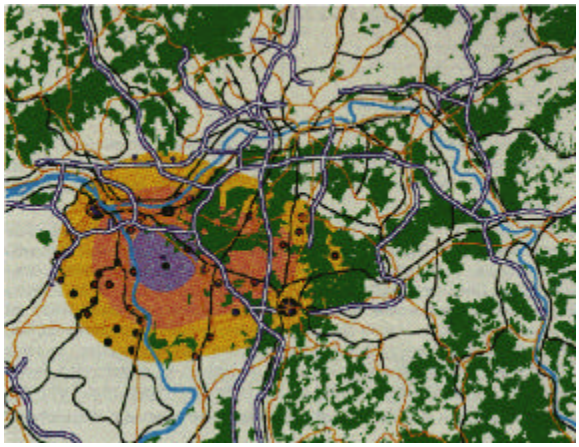
- Verbesserungen
  - Wortstammbildung (Stemming), Synonymsuche, Einsatz eines Thesaurus (komplex organisiertes Wörterbuch)
  - nicht nur boole'sche Anfragen, sondern Gewichtung (Vektorraum-Ansatz), statistische Ansätze ...
  - Ranking entscheidend für große Ergebnislisten

Digitale Bibliotheken

- riesige Dokumenten- und Medienarchive im Intranet bzw. Internet
- sehr große Benutzerzahl
- Kombination von DB- und IR-Funktionalität für Dokumente
  - vage Suchanfragen (natürliche Sprache)
  - Ranking von Ergebnismengen (Relevanzbewertung)
  - Text Mining
- Berücksichtigung der Dokumentenstruktur in Suchanfragen
- inhaltsbasierte Suche auf Multimedia-Objekten (Bilder, Videos)
- Autorisierung, Zahlungsmodelle
- Integration unterschiedlicher Bibliotheken
- effiziente Verwaltung großer Kollektionen von XML-Dokumenten (semi-strukturierte Daten)

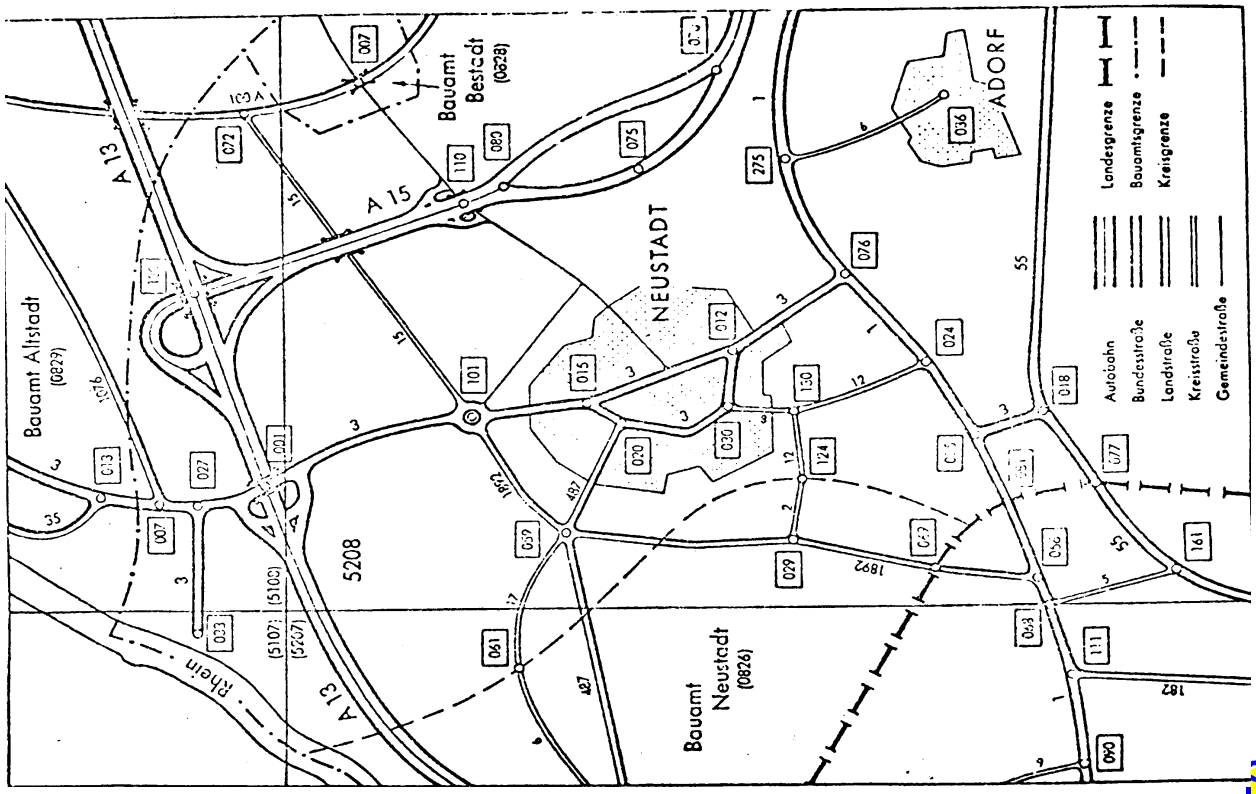
Geo-Informationssysteme

- Karthographie, Katasterwesen, Planungsaufgaben, Umweltinformationssysteme, ...
- räumliche Daten (Geometrie, Topologie)
- Vektordarstellung
  - Grundelemente: Punkt, Linie, Fläche
  - gute Datenstrukturierung, geometrische Berechnungen
- Rasterdarstellung
  - flächenmäßige Objektdarstellung durch Pixelmenge
  - einfache Datenerfassung, jedoch geringe Strukturierung
  - große Datenmengen



- große Datenmengen, aufwendige Berechnungen und Suchvorgänge ("spatial data mining")
- Satellitenbild-Archive für Klimastudien etc. im Petabyte-Bereich (z.B. NASA Earth Observing System liefert 4 TB/Tag, 15 PB bis 2007)

Inhalte einer Straßen-DB



Beispiel: Straßeninformationssystem

- **Straßendatenbank ist Bestandsnachweis für das Straßennetz eines Bundeslandes**
  - Es gibt verschiedene Straßentypen (Autobahnen, Bundesstraßen, Kreisstraßen, Gemeindestraßen etc.)
  - Die Straßen sind in Abschnitte eingeteilt, die durch ihre Endknoten definiert sind.
  - Straßenabschnitte sind jeweils einem Bauamt zugeordnet
  - Die Straßenabschnitte gehen durch Gemeinden. Die Gemeinden gehören zu Kreisen.
  - Ein Straßenabschnitt kann mehrere Äste aufweisen (z.B. Aufteilung in 2 Einbahnstrecken). Eine Straße kann an einem Netzknoten unterbrochen sein und an einem anderen Netzknoten weiterführen. Ein Straßenabschnitt kann auf mehreren Straßen (z.B. Bundesstraße und Kreisstraße) gleichzeitig verlaufen.

Topologische Information

- Zusätzlich sind jedem Straßenabschnitt Daten zugeordnet, welche den geometrischen Verlauf zwischen den begrenzenden Netzknoten festlegen (Trassierungselemente: Kreise, Geraden, Klothoiden).
- Die Bauwerke (Brücken, Durchlässe, Signalanlagen etc.) sind dem geometrischen Verlauf des Straßenabschnitts ebenso zugeordnet wie Fußgängerüberwege, Radwege, Gehsteige, Daten des Fahrbahnaufbaus, Höheninformation, Entwässerungsschächte etc.

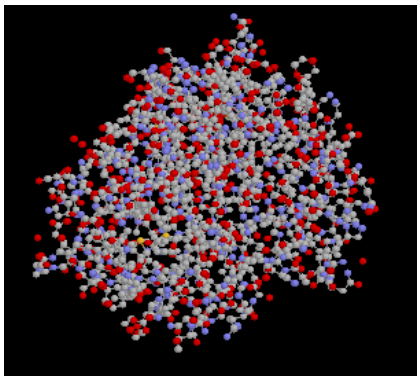
Unfalldaten, Verkehrsmengen, Frostsicherheit etc. sind weitere Attribute zum Straßenabschnitt

Typische Fragen:

- Auswahl aller Kreisstraßen im Kreis ... mit Breite < 5m und NN-Höhe > 500m.
- Zusammenstellung aller Strecken mit Radwegen getrennt für Ortsdurchfahrt und freie Strecke.
- Auswahl aller Bundesstraßenstrecken im Bauamt ... mit Neigungen größer als 7%.
- Berechnung der befestigten Straßenfläche für alle im Jahr 1980 gebauten Bundesstraßenstrecken

Datenbanken in der Bioinformatik

- Wesentliches Ziel der Bioinformatik ist die Analyse und Verarbeitung von Daten aus Molekularbiologie und Genomforschung
- Grundzusammenhänge
  - Menschliches Genom (DNA) besteht aus einer Sequenz von ca. 3 Milliarden Nukleotiden (4 Varianten: A, G, C, T)
  - Proteine setzen sich aus bis zu Tausenden von Aminosäuren (20 Varianten) zusammen und werden durch DNA-Abschnitte (Gene) kodiert
  - menschliche Körper hat über 30.000 Gene
  - jede Zelle hat vollständige DNA, jedoch ist nur Teilmenge der Gene aktiv („exprimiert“), je nach Funktion der Zelle und Umgebungsbedingungen (Gesundheitszustand, Medikamenteneinnahme, etc.)



- komplexe Suchaufgaben
  - Lokalisierung unbekannter Gene, Funktionsbestimmung von Genen
  - genetische Ursachen bestimmter Krankheiten, Therapiemöglichkeiten . . .
- z.Zt. > 100 Datenbanken in Molekularbiologie zu Genen, Proteinen etc. unterschiedlicher Spezies
  - grosse Datenmengen: GenBank enthält 10 Millionen Nukleotid-Sequenzen mit ca. $11 * 10^{12}$ Nukleotid-Vorkommen (Dez. 2000)
  - stark wachsender Umfang
  - Semantik von Sequenzen steckt in Annotationen

Wissensbasierte Anwendungen

- Verwaltung von Fakten (formatierte Daten = extensionale DB) und Regeln (intensionale DB)
- Regeln dienen zur Ableitung von implizit vorhandenen Informationen
- Hauptanforderung: effiziente Regelauswertung (Inferenz), Behandlung von Rekursion

Fakten:

F1: Elternteil(C, A) <-
F2: Elternteil (D, A) <-
F3: Elternteil (D, B) <-
F4: Elternteil (G, B) <-
F5: Elternteil (E, C) <-
F6: Elternteil (F, D) <-
F7: Elternteil (H, E) <-

Regeln:

R1: Vorfahr (x, y) <- Elternteil (x, y)
R2: Vorfahr (x, y) <- Elternteil (x, z), Vorfahr (z, y)

Anfrage: ? Vorfahr (x, A)

- Nutzung für KI-Anwendungen, z.B. in Verbindung mit Expertensystemen
- auch konventionelle Anwendungen profitieren von rekursiven Anfragemöglichkeiten (Berechnung der transitiven Hülle): Stücklistenauflösung, Wegeprobleme etc.
- Deduktive DBS bzw. Wissensbankverwaltungssysteme (Regeln bzw. sonstige Form der Wissensrepräsentation)

Objekt-Darstellung

- Standard-Anwendung:
  pro Objekt gibt es genau eine Satzausprägung, die alle beschreibenden Attribute enthält

Schema

ANGESTELLTER

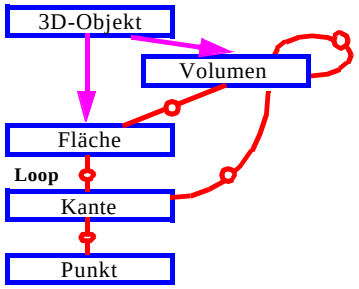
Satztyp (Relation)

Ausprägungen

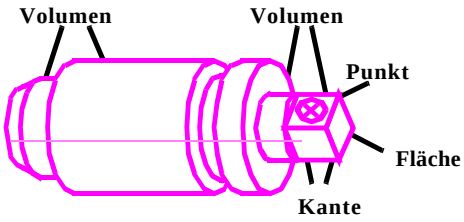
PNR	NAME	TAETIGKEIT	GEHALT	ALTER
496	Peinl	Pfoertner	2100	63
497	Kinzinger	Kopist	2800	25
498	Meyweg	Kalligraph	4500	56

- CAD-Anwendung: das komplexe Objekt „Werkstück“ setzt sich aus einfacheren (komplexen) Objekten verschiedenen Typs zusammen.

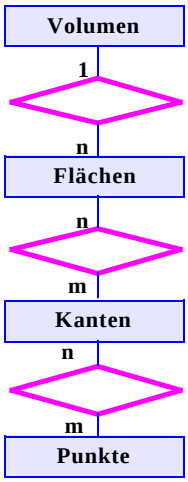
Schema

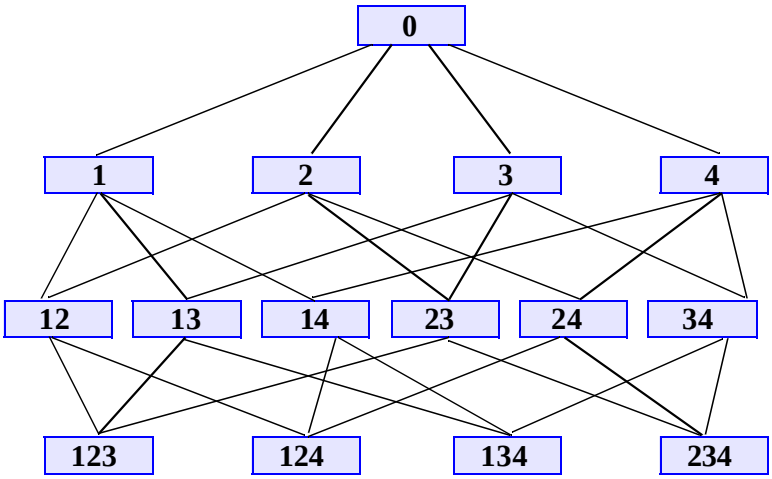


Objektausprägung

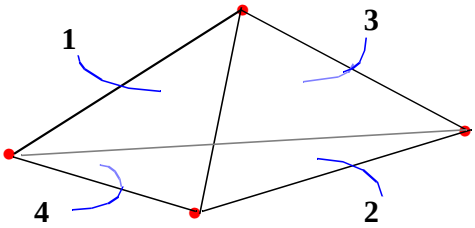


Modellierung von 3D-Objekten im ER-Modell



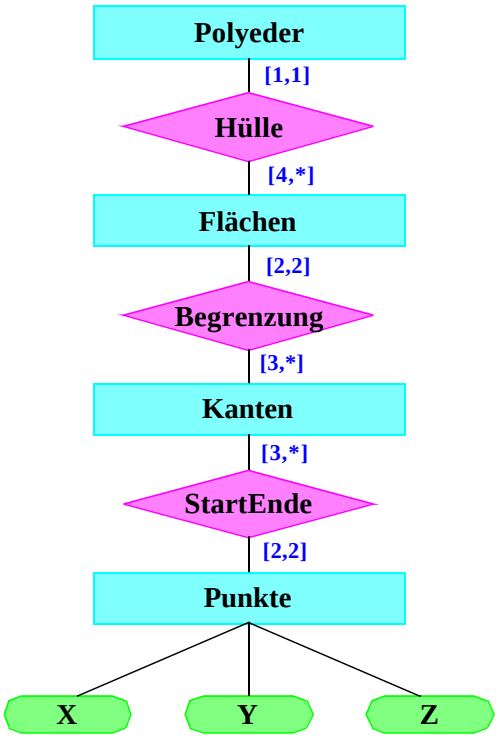


Beispiel: Tetraeder



Beispiel: Modellierung von Polyedern im RM

ERM



Modellierung im Relationenmodell

```
CREATE TABLE Polyeder
    (polyid:     INTEGER,
    anzflächen:INTEGER,
    PRIMARY KEY (polyid));
```

```
CREATE TABLE Fläche
    (fid:          INTEGER,
    anzkanten:INTEGER,
    pref :         INTEGER,
    PRIMARY KEY (fid),
    FOREIGN KEY (pref)
REFERENCES Polyeder);
```

```
CREATE TABLE FK_Rel
    (fid:          INTEGER,
    kid:           INTEGER,
    PRIMARY KEY  (fid, kid),
    FOREIGN KEY (fid)
        REFERENCES Fläche,
    FOREIGN KEY (kid)
        REFERENCES Kante);
```

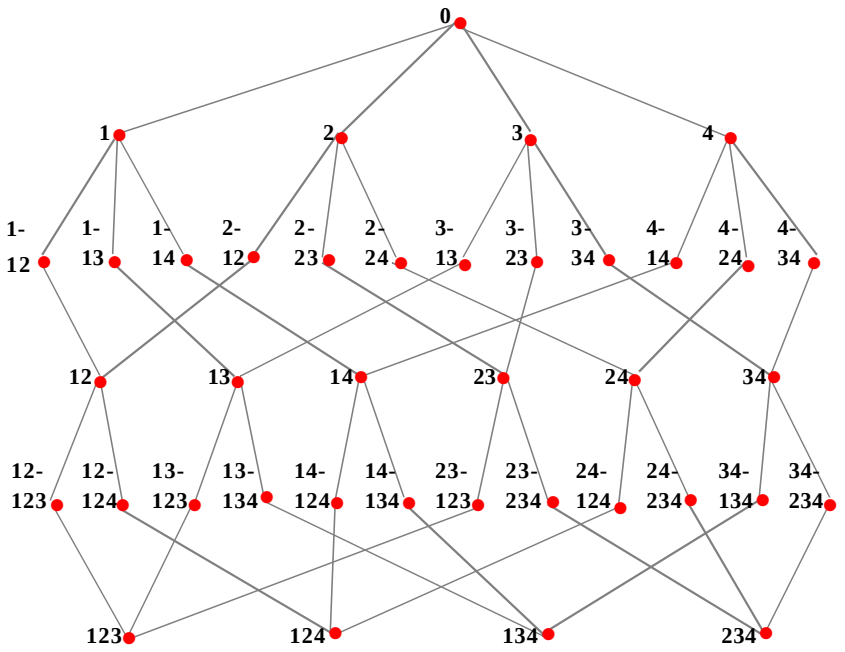
```
CREATE TABLE Kante
    (kid:          INTEGER,
    ktyp:          CHAR(5),
    PRIMARY KEY (kid));
```

```
CREATE TABLE KP_Rel
    (kid:          INTEGER,
    pid:           INTEGER,
    PRIMARY KEY  (kid, pid),
    FOREIGN KEY (kid)
        REFERENCES Kante,
    FOREIGN KEY (pid)
        REFERENCES Punkt);
```

```
CREATE TABLE Punkt
    (pid:          INTEGER,
    x, y, z:       INTEGER,
    PRIMARY        KEY
```

RM - angemessene Modellierung?

Darstellung eines Tetraeder mit vid = 0



Relationen

Polyeder

Fläche

FK-Rel

Kante

KP-Rel

Punkt

Referenzbeispiel in relationaler Darstellung: Anfragen

- Finde alle Punkte, die zu Flächenobjekten mit F.fid < 3 gehören

```
SELECT  F.fid, P.x, P.y, P.z
FROM    Punkt P, KP-Rel S, Kante K, FK-Rel T, Fläche F
WHERE   F.fid < 3
   AND  P.pid = S.pid          /*  Rekonstruktion */
   AND  S.kid  = K.kid         /*  komplexer Objekte*/
   AND  K.kid  = T.kid         /*  zur Laufzeit   */
   AND  T.fid  =  F.fid;
```

- Symmetrischer Zugriff: Finde alle Flächen, die mit Punkt (50,44,75) assoziiert sind

```
SELECT  F.fid
FROM    Punkt P, KP-Rel S, Kante K, FK-Rel T, Fläche F
WHERE   P.x = 50 AND P.y = 44 AND P.z = 75
   AND  P.pid = S.pid
   AND  S.kid  = K.kid
   AND  K.kid  = T.kid
   AND  T.fid  =  F.fid;
```

Beschränkungen des Relationenmodells

- Datenmodellierung
  - nur einfach strukturierte Datenobjekte (satzorientiert, festes Format)
  - nur einfache (Standard-) Datentypen
  - nur einfache Integritätsbedingungen
  - keine Unterstützung der Abstraktionskonzepte (Generalisierung, Aggregation)
  - keine Unterstützung von Versionen und Zeit
  - keine Spezifikation von "Verhalten" (Funktionen)
  - geringe Semantik (beliebige Namensvergabe, Benutzer muß Bedeutung der Daten kennen)
- begrenzte Auswahlmächtigkeit der Anfragesprachen
  - nicht sämtliche Berechnungen möglich
  - keine Unterstützung von Rekursion (Berechnung der transitiven Hülle)
  - Änderungen jeweils auf 1 Relation beschränkt, ...
- umständliche Einbettung in Programmiersprachen (impedance mismatch)
- auf kurze Transaktionen zugeschnitten (ACID)
- oft schlechte Effizienz für anspruchsvolle Anwendungen

Zusammenfassung

- DBS-Technologie
  - strukturierte Verwaltung von persistenten Daten
  - effizienter Datenzugriff
  - flexibler Mehrbenutzerbetrieb
  - Konzepte: **Datenmodell und DB-Sprache**, Transaktion als Kontrollstruktur
- zunehmende Anzahl anspruchsvoller DB-Anwendungen mit besonderen Anforderungen bezüglich Datenstrukturierung und Operationen
- Vielzahl von DBS-Entwicklungen
  - Multimedia-DBS
  - Volltext-DBS
  - XML-DBS
  - Deduktive DBS
  - Geo-DBS
  - CAD-DBS
  - Bio-DBS . . .
- Relationenmodell zur Unterstützung von anspruchsvollerer Anwendungen nur bedingt geeignet

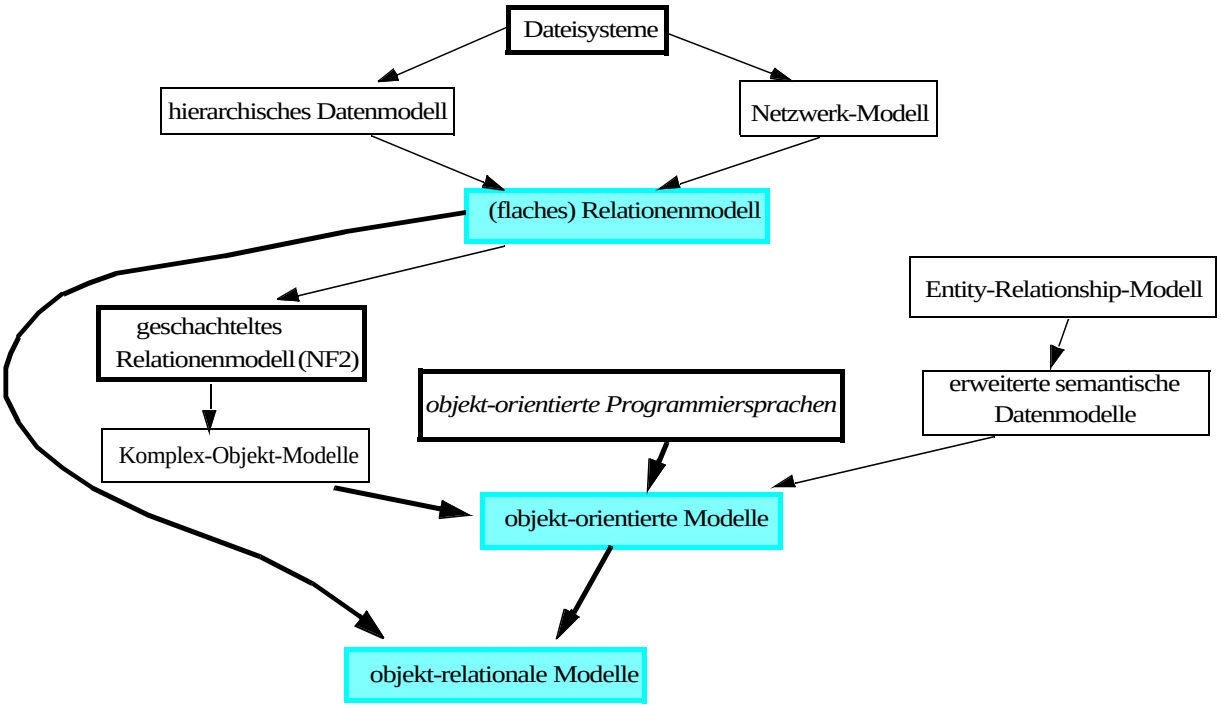
4. Grundkonzepte von objekt-orientierten und objekt-relationalen DBS

- Einführung
  - Anforderungen
  - Klassifikation von Datenmodellen
  - NF2-Ansatz
- OODBS vs. ORDBS
- Konzepte
  - Objektidentität, Komplexe Objekte
  - Typen/Klassen; Kapselung/Abstrakte Datentypen
  - Typhierarchie/Vererbung; Überladen/spätes Binden
  - Operationale Vollständigkeit / Programmiersprachen-Integration
- Weitere Aspekte
  - Verwaltung von Multimedia-Objekten
  - Versionen und Konfigurationen
  - erweiterte Transaktionsmodelle

Forderungen an Datenmodell

- Definition von
  - statischen Eigenschaften (Objekte, Attribute, Datentypen, Beziehungen),
  - dynamischen Eigenschaften (Operationen) sowie
  - statischen und dynamischen Integritätsbedingungen
- Mächtigkeit:
  - direkte Modellierbarkeit von Anwendungsobjekten (=> Unterstützung komplexer Objekte)
  - Unterstützung spezieller semantischer Konstrukte (n:m-Beziehungen, etc.)
- Abstraktionsvermögen (Information hiding)
- Erweiterbarkeit bei Datentypen und Operationen
- Exakte, formale Definition zur Gewährleistung von
  - Konsistenz (Nicht-Widersprüchlichkeit)
  - Vollständigkeit und
  - geringer Redundanz

Entwicklung von Datenmodellen



NF^2-Modell

- **N**on-**F**irst **N**ormal **F**orm ('nested relations', unnormalisierte Relationen)
- wie Relationenmodell, jedoch können Attributwerte auch Relationen (Mengen von Tupeln) sein
- Probleme bei netzwerkartigen Beziehungen
- Polyeder-Beispiel

```
CREATE  TABLE Volumen
        { [     VId          INT,
                Bez          CHAR(20),
                AnzFlaechen  INT,
                Flaechen     { [   FId         INT,
                                   AnzKantenINT,
                                   Kanten { [ KId        INT,
                                              Punkte     { [   PId    INT,
                                                               X      INT,
                                                               Y      INT,
                                                               Z      INT
                                                            ] }
                                            ] }
                                ] }
        ] }
```

{ } *Mengenkonstruktor,* [ ] *Tupelkonstruktur*

NF²-Ausprägungen

Volumen							
VId	Bez	Flaechen					
		FId	Kanten				
			KId	Punkte			
				PId	X	Y	Z
0	Tetraeder	1	12	123	0	0	0
				124	100	0	0
			13	123	0	0	0
				134	50	44	75
			14	124	100	0	0
				134	50	44	75
		2	12	123	0	0	0
				124	100	0	0
			23	123	0	0	0
				234	50	87	0
			24	124	100	0	0
				234	50	87	0
		3	13	123	0	0	0
				134	50	44	75
			23	123	0	0	0
				234	50	87	0
			34	134	50	44	75
				234	50	87	0
		4	14	124	100	0	0
				134	50	44	75
			24	124	100	0	0
				234	50	87	0
			34	134	50	44	75
				234	50	87	0

NF²-Modell (3)

- Erweiterte relationale Algebra
  - Erweiterung von Projektion und Selektion auf geschachtelte Strukturen
  - ***NEST***-Operation: Erzeugen geschachtelter Relationenformate aus flachen Relationen
  - ***UNNEST***-Operation: Normalisierung (“Flachklopfen”) geschachtelter Relationen
  - NEST und UNNEST sind i.a. nicht invers zueinander !

A	B
1	2 3
1	4 5

A	B

A	B

- Erweiterter natürlicher Join

R1

A	B	X	
		C	D
a1	b1	c1	d1
		c2	d2
		c1	d3
a2	b2	c1	d2
		c3	d2

R2

E	B	X	
		C	D
e1	b1	c1	d1
		c1	d3
		c3	d4
e3	b2	c3	d2

R1 ⋈ R2

Bewertung des NF^2-Modells

- Vorteile:
  - einfaches Relationenmodell als Spezialfall enthalten
  - Unterstützung komplex-strukturierter Objekte
  - reduzierte Join-Häufigkeit
  - Clusterung einfach möglich
  - sicheres theoretisches Fundament (NF2-Algebra)

- Nachteile:
  - überlappende/gemeinsame Teilkomponenten führen zu Redundanz
  - unsymmetrischer Zugriff
  - rekursiv definierte Objekte nicht zulässig
  - keine Unterstützung von Generalisierung und Vererbung
  - keine benutzerdefinierten Datentypen und Operationen

Objekt-orientierte DBS (OODBS)

- OODBS-Anforderungen
  - Modellierung der Struktur (komplexe Objekte)
  - Modellierung von Verhalten (ADTs, Methoden)
- Auseinanderklaffen von Datenbanksystem und Programmiersprachen: “Impedance mismatch” (Fehlanpassung)
  - “Struktur” wird durch das DBS verwaltet
  - “Verhalten” wird weitgehend von Anwendungsprogramm (Programmiersprache) abgedeckt
  - unterschiedliche Datentypen
  - Mengen- vs. Satzverarbeitung
  - Notwendigkeit der “Einbettung” von Datenbank- in Programmiersprache
- Ansätze zur objekt-orientierten Datenverwaltung
  - Anreicherung von Programmiersprachen um Persistenz und andere DB-Eigenschaften wie Integrität, Zuverlässigkeit,...) -> *persistente Programmiersprachen*
  - Anreicherung von DBS um objekt-orientierte Konzepte
- OODBS: Beseitigung des “impedance mismatch”/ bessere Effizienz
  - Integration von objektorientierter Programmierung und DB-Sprache
  - verbesserte Daten- sowie verbesserte Verhaltensmodellierung (Programmentwicklung)
  - Unterstützung von Entwurfs- und Wartungseffizienz sowie Ausführungseffizienz

Objekt-relationale DBS

- Merkmale von ORDBS
  - Erweiterung des relationalen Datenmodells um Objekt-Orientierung
  - benutzerdefinierte Datentypen (u.a. Multimedia-Datentypen)
  - komplexe, nicht-atomare Attributtypen (z.B. relationenwertige Attribute)
  - erweiterbares Verhalten über gespeicherte Prozeduren und benutzerdefinierte Funktionen
  - Bewahrung der Grundlagen relationaler DBS, insbesonderer deklarativer Datenzugriff, Sichtkonzept etc.
  - alle Objekte müssen innerhalb von Tabellen verwaltet werden

- Vergleich
  - relationale DBS: einfache Datentypen, Queries, ...
  - OODBS: komplexe Datentypen, gute Programmiersprachen-Integration, hohe Leistung für navigierende Zugriffe
  - ORDBS: komplexe Datentypen, Querying ...
  - erste Standardisierung von ORDBS durch SQL99

Grobvergleich nach Stonebraker

	simple data	complex data
query		
no query		

- kein Systemansatz erfüllt alle Anforderungen gleichermaßen gut
- OODBS basierend auf objekt-orientierten Programmiersprachen bleiben Nischen-Produkte
- Bedeutung von ORDBS nimmt stark zu

Definition eines objekt-orientierten DBS

- ein objekt-orientiertes DBS (OODBS) muß zwei Kriterien erfüllen
  - es muß ein DBS sein
  - es muß ein objekt-orientiertes System sein

- DBS-Aspekte:
  - Persistenz
  - Externspeicherverwaltung
  - Datenunabhängigkeit
  - Transaktionsverwaltung: Synchronisation, Logging, Recovery
  - Ad-Hoc-Anfragesprache

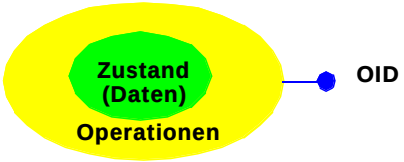
- essentielle OOS-Aspekte:
  - Objektidentität
  - komplexe Objekte
  - Kapselung
  - Typ-/Klassenhierarchie, Vererbung
  - Überladen und spätes Binden
  - operationale Vollständigkeit
  - Erweiterbarkeit

- optionale OOS-Aspekte:
  - Mehrfachvererbung
  - Versionen
  - lang-lebige Transaktionen •••

Objektidentität

- Relationenmodell ist "wertebasiert"
  - Identifikation von Daten durch Schlüsselwerte
  - Benutzer muß häufig künstliche Schlüsselattribute einführen
  - Repräsentation von Beziehungen führt zu erheblicher Redundanz (Fremdschlüssel)
  - schwierige (ineffiziente) Modellierung komplexer Strukturen
  - Änderungsprobleme (referentielle Integrität, Sichtkonzept)

- OODBS: Objekt = (OID, Zustand, Operationen)
  - OID: Identifikator
  - internen Zustand, der mit Hilfe von Attributen beschrieben wird
  - Menge von Operationen (Methoden), die ihre Schnittstelle zur externen Welt definieren

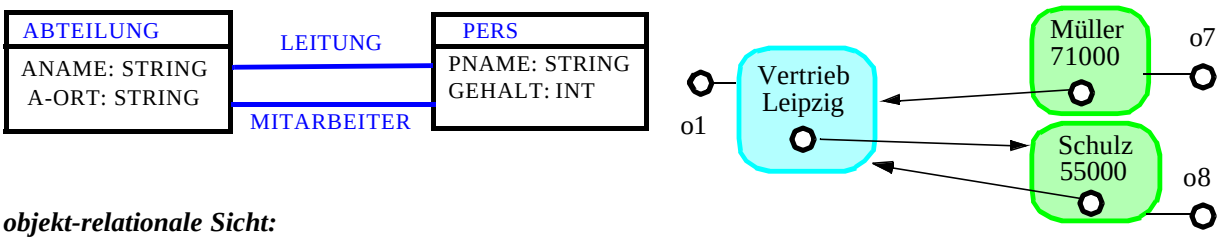


- Objektidentität
  - systemweit eindeutige Objekt-Identifikatoren
  - OID während Objektlebensdauer konstant, üblicherweise systemverwaltet
  - OID tragen keine Semantik (<-> Primärschlüssel im RM)
  - Änderungen beliebiger Art (auch des Primärschlüssels im RM) ergeben *dasselbe* Objekt

- Notwendigkeit künstlicher Primärschlüssel wird vermieden

Objektidentität (2)

- Realisierung von Beziehungen über OIDs (Verwendung von Referenz-Attributen)



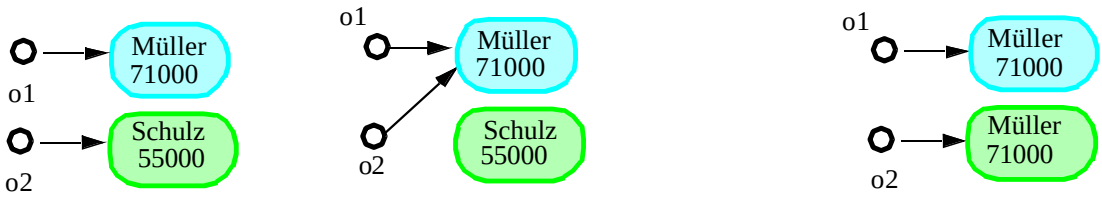
objekt-relationale Sicht:

```
ABT  (OID:  IDENTIFIER, (* implizit *)
      ANAME:  STRING,
      A-ORT: STRING,
      LEITER: REF (PERS) ... (* Referenz-Attribut *)

PERS   (OID: IDENTIFIER (* implizit *),
        PNAME: STRING,
        GEHALT: INT,
        ABTLG: REF (ABT), (* Referenz-Attribut *)
```

- gemeinsame Teilobjekte ohne Redundanz möglich (referential sharing)
- Stabilität der OIDs erleichtert Wartung der referentiellen Integrität
- Objekt-Ids / Referenzen erlauben Bildung komplexer Objekte bestehend aus Teilobjekten
- implizite Dereferenzierung über *Pfadausdrücke* anstatt expliziter Verbund-anweisungen

Objektidentität (3)

- Identität vs. Gleicheit
  - zwei Objekte sind identisch (O1 == O2), wenn sie dieselbe OID haben
  - zwei Objekte sind gleich (O1 = O2), wenn sie den gleichen Zustand besitzen
  - beides sollte ausdrückbar sein
- Zuweisungen: Referenzsemantik vs. Wertesemantik



Referenzsemantik
(ändert Referenz von o2 auf o1)

Wertesemantik
(kopiert gesamten Zustand)

- eindeutige Semantik von Änderungsoperationen im Gegensatz zu Werteän-derungen im RM

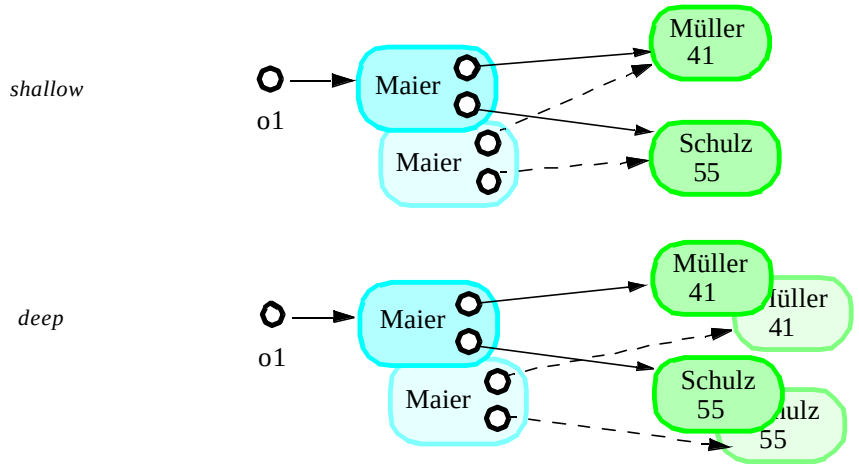
Name	Gehalt
Müller	71K
Schulz	55K

→

Name	Gehalt
Müller	80K

Objektidentität (4)

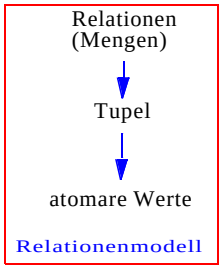
- *Flaches* vs. *tiefes Kopieren* (shallow vs. deep *copy*)
  - Flaches Kopieren: neues Objekt erhält gleichen Zustand wie zu kopierendes Objekt
  - Tiefes Kopieren: referenzierte Objekte (Komponenten) werden auch kopiert (rekursiv über alle Stufen)



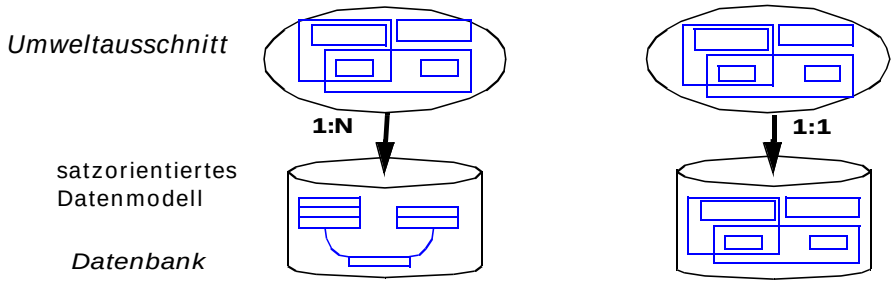
- Unterschiedliche Gleichheitstests erforderlich: Shallow Equal vs. Deep Equal

Komplexe Objekte

- Relationenmodell
  - nur einfache Attribute, keine zusammengesetzte oder mengenwertige Attribute
  - nur zwei Typkonstruktoren: Bildung von Tupeln und Relationen (Mengen)
  - keine rekursive Anwendbarkeit von Tupel- und Mengenkonstruktoren



- OODBS: Objekte können Teilobjekte enthalten (Aggregation)
  - Komposition (Wertesemantik) vs. Referenzierung über OIDs
- Objektattribute können sein
  - Standardtypen (Integer, Char, ...)
  - zusammengesetzte Typen
  - benutzerdefinierte Typen (Zeit, Datum etc.)
  - andere Objekte (eingebettet oder referenziert)



Komplexe Objekte: Typkonstruktoren

- Typkonstruktoren zum Erzeugen strukturierter (zusammengesetzter) Datentypen aus Basistypen
  - TUPLE (RECORD)
  - SET
  - BAG (MULTISET)
  - LIST (SEQUENCE)
  - ARRAY (VECTOR)

Typ	Duplikate	Ordnung	Heterogenität	#Elemente	Elementzugriff über
TUPLE	JA	JA	JA	konstant	Namen
SET	NEIN	NEIN	NEIN	variabel	Iterator
BAG	JA	NEIN	NEIN	variabel	Iterator
LIST	JA	JA	NEIN	variabel	Iterator / Position
ARRAY	JA	JA	NEIN	konstant	Index

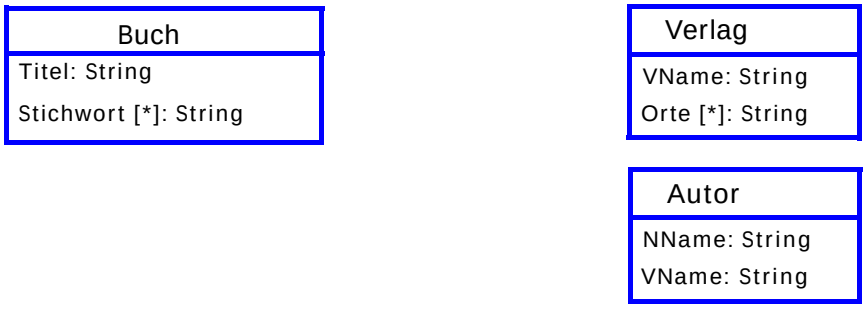
- beliebige (rekursive) Kombination aller Konstruktoren zum Aufbau komplexer Objekte
- SET/BAG/LIST/ARRAY verwalten homogene Kollektionen: Kollektionstypen

Komplexe Objekte: Beispiel

- Beispiel: Objektmenge "Buch“

```
BUCH SET ( TUPLE ( Titel:        String,
                   Autoren:      LIST ( REF (Autor) )
                   Buchverlag:   REF (VERLAG), (* Referenzattribut *)
                   Stichwort:    SET (String) ) )
```

- UML-Darstellung



Komplexe Objekte: Beispiel (2)

- Ausprägungen in einer objekt-relationalen Sicht (Beispiel)

Buch

	Titel	Autoren	Buchverlag	Stichwort
#10	Mehrrechner-DBS	LIST (#21)	#31	SET (DBS, Verteilung, Parallelität)
#11	Datenbanksysteme	LIST (#22, #21)	#32	SET (Implementierung, DBS)

Autor

	NName	VName
#21	Rahm	Erhard
#22	Härder	Theo
#23	Lockemann	Peter

Verlag

	VName	Orte
#31	Oldenbourg	SET (München, Wien)
#32	Springer	SET (Berlin, Heidelberg, New York)

- Generische Operationen auf Tupel-/Kollektionstypen:
  - Komponentenzugriff
  - Methoden wie MEMBER, FIRST, ...

Typen / Klassen

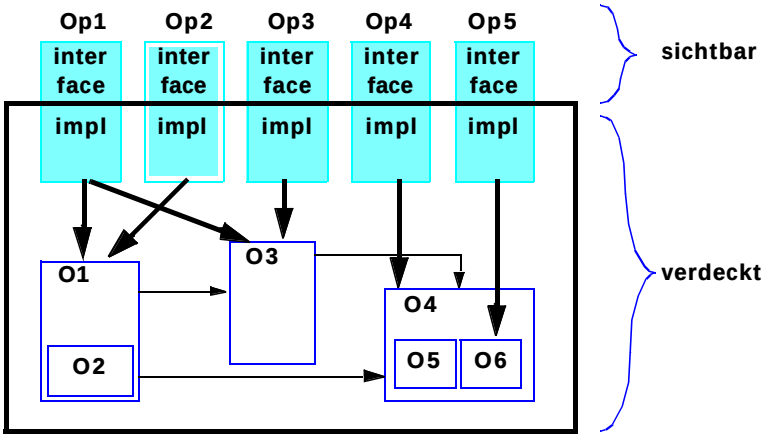
- Typ / Klasse
  - Intensionale Festlegung von Struktur (Attribute und ihre Wertebereiche) und Operationen für Objekte
  - Begriffe „Typ“ und „Klasse“ meist synonym

PERS
PNAME: STRING GEHALT: INT GEBDATUM: DATUM
INT Alter()

- Objekt = Instanziierung eines Typs mit konkreten Wertebelegungen der Attribute
- Vordefinierte vs. benutzerdefinierte Typen (BDT, UDT)
- Arten von Operationen
  - Vordefinierte vs. benutzerdefinierte Operationen / Funktionen (BDF, UDF)
  - Konstruktoren (Erzeugung/Initialisierung neuer Instanzen eines Objekttyps); Destruktoren
  - lesende Operationen (Beobachter); Mutatoren
  - generische Operationen auf Kollektionen (Mengen, Listen, Arrays)
- Verwaltung der Operationen im DBS
  - Reduzierung des "impedance mismatch"
  - verringerte Kommunikationshäufigkeit zwischen Anwendung und DBS

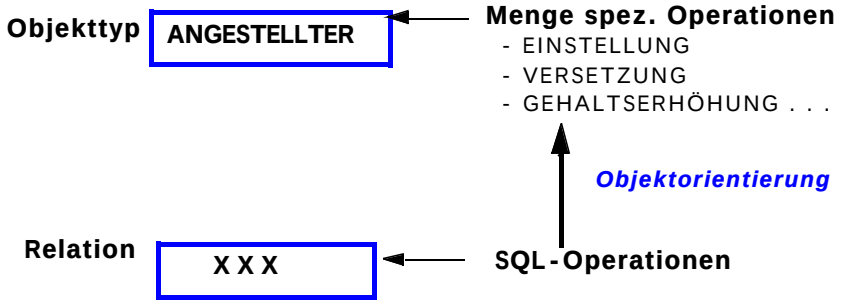
Abstrakte Datentypen / Kapselung

- Objekt = Struktur + Verhalten
- Abstrakte Datentypen (ADTs) verlangen Kapselung: Geheimnisprinzip (Information Hiding)
  - Struktur des Objektes ist verborgen
  - Verhalten des Objektes ist ausschließlich durch seine *Operationen* (Methoden) bestimmt
  - nur Namen und Signatur (Argumenttypen, Ergebnistyp) von Operationen werden bekannt gemacht
  - Implementierung der Operationen bleibt verborgen



Abstrakte Datentypen / Kapselung (2)

- Vorteile der Kapselung
  - größerer Abstraktionsgrad
  - erhöht (logische) Datenunabhängigkeit
  - Datenschutz



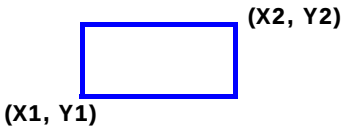
- Aber: strikte Kapselung oft zu restriktiv
  - eingeschränkte Flexibilität
  - mangelnde Eignung für Ad-Hoc-Anfragen
- Koexistenz von attributbezogenen Anfragen und objekttypspezifischen Operationen
- ADTs vs. Benutzerdefinierte Datentypen (BDTs)
- rekursive Anwendbarkeit des BDT-Konzeptes auf Klassen und Attribute

Anwendungsbeispiel: Verwaltung räumlicher Objekte

- Relationenmodell bietet keine Unterstützung
- hohe Komplexität bereits in einfachen Fällen
- Beispiel: Darstellung von Rechtecken in der Ebene

a) Relationenmodell

R-ECK (RNR, X1, Y1, X2, Y2: INTEGER)



b) BDT-Lösung

BDT **BOX** mit Funktionen
INTERSECT, CONTAINS, AREA, usw.

R-ECK (RNR: INTEGER, Beschr: BOX)

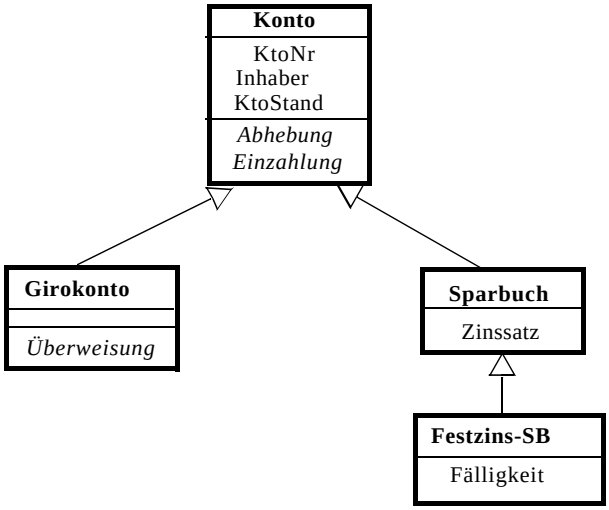
Finde alle Rechtecke, die das Rechteck ((0,0) (1,1)) schneiden!

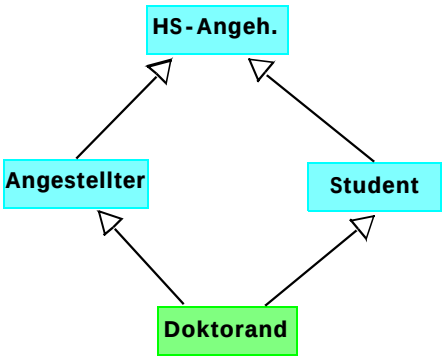
```
SELECT RNR FROM R-ECK
WHERE (X1 > 0 AND X1 < 1 AND Y1 > 0 AND Y1 < 1)
OR (X1 > 0 AND X1 < 1 AND Y2 > 0 AND Y2 < 1)
OR (X2 > 0 AND X2 < 1 AND Y1 > 0 AND Y1 < 1)
OR (X2 > 0 AND X2 < 1 AND Y2 > 0 AND Y2 < 1)
OR . . .
```

```
SELECT RNR FROM R-ECK
WHERE INTERSECT (Beschr, (0, 0, 1, 1))
```


Typ/Klassen-Hierarchie

- Anordnung von Objekttypen in Generalisierungs-/Spezialisierungshierarchie (IS-A-Beziehung)
- Vererbung von Attributen, Methoden, Integritätsbedingungen und Default-Werten
- Arten der Vererbung: einfach (Hierarchie) vs. mehrfach (Typverband)





Typ/Klassen-Hierarchie (2)

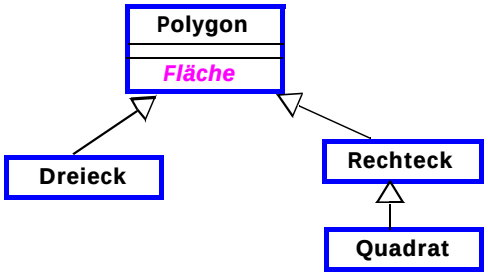
- Prinzip der Substituierbarkeit: Instanz einer Subklasse B kann in jedem Kontext verwendet werden, in dem Instanzen der Superklasse A möglich sind (jedoch nicht umgekehrt)
  - Methoden der Oberklasse sind auch für Objekte einer Unterklasse ausführbar
  - Zuweisungsregel: einer Oberklasse können Objekte von Unterklassen zugewiesen werden

```
define float berechne-zinsen (x: Sparbuch)
a: Sparbuch;  b: Festzins-SB; c: float;
a := b;
c := berechne-zinsen (b);
```

- Substituierbarkeitsprinzip impliziert, dass Klasse heterogene Objekte enthalten kann
- Vorteile des Vererbungsprinzips:
  - Code-Wiederverwendung (reusability)
  - Erweiterbarkeit
  - Repräsentation zusätzlicher Semantik
  - Modellierungsdisziplin (schrittweise Verfeinerung)

Überladen (Overloading)

- Overloading: derselbe Methodenname wird für unterschiedliche Prozeduren verwendet
- Overloading kann auch innerhalb von Typ-Hierarchien angewendet werden
  - Redefinition von Methoden für Subtypen (Overriding)
  - spezialisierte Methode mit gleichem Namen



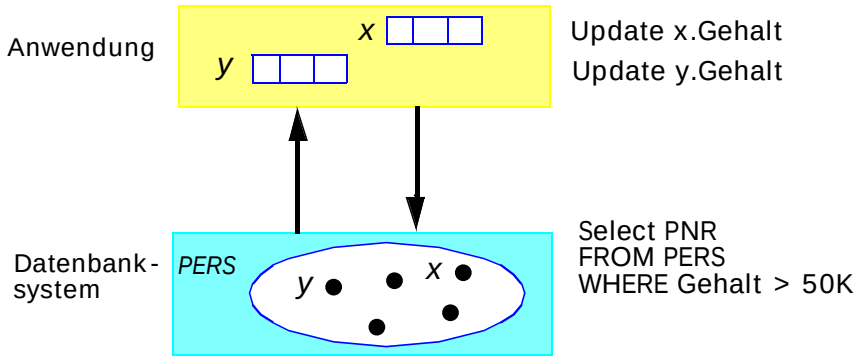
- Verwendung eines einzigen Funktionsnamens erleichtert Realisierung nutzender Programme und verbessert Software-Wiederverwendbarkeit
- Überladen impliziert dynamisches (spätes) Binden zur Laufzeit (late binding)
- Polymorphismus (griech. "viele Formen"): gemeinsame Schnittstelle für unterschiedliche Operationen
  - polymorphe Methoden: Name der Methode bleibt gleich, Wirkung der Methode ist objektabhängig

Operationale Vollständigkeit (computational completeness)

- nicht alle Berechnungen in herkömmlichen DB-Anfragesprachen möglich (DML ist Teilsprache)

  => Einbettung in allgemeine Programmiersprache

- Verwendung zweier Sprachen führt zu *Impedance Mismatch*:
  - Fehlanpassung von Datenbanksprachen (DDL/DML) und herkömmlichen Programmiersprachen
  - unterschiedliche Typ-Systeme (nur begrenzte Typ-Prüfungen möglich, Typkonversionen)
  - deklarative DML vs. prozedurale PS
  - mengen- vs. satzorientierte Verarbeitung => Cursorkonzept
  - umständliche, fehleranfällige Programmierung

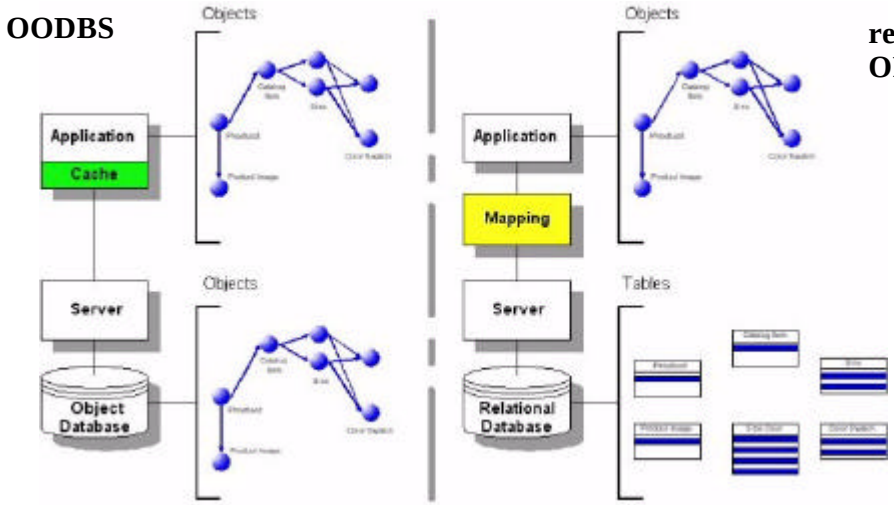


Operationale Vollständigkeit in OODBS

- Alternative: einheitliche DB-Programmiersprache (DBPL) / persistente OO-Programmiersprache
  - operationale Vollständigkeit
  - nur eine Programmierschnittstelle
  - keine Konversionen von Datentypen
  - Objekt-Orientierung, erweiterbares Typsystem
  - gleichartige Behandlung von transienten und persistenten Datenstrukturen
  - Unterstützung von Anfragen, Transaktionen und allgemeinen Berechnungen auf beliebigen Datenstrukturen

- Probleme
  - Einschränkungen bezüglich Query-Mächtigkeit / mengenorientierter Verarbeitung
  - Einschränkungen bezüglich Datenunabhängigkeit

Effiziente Navigation



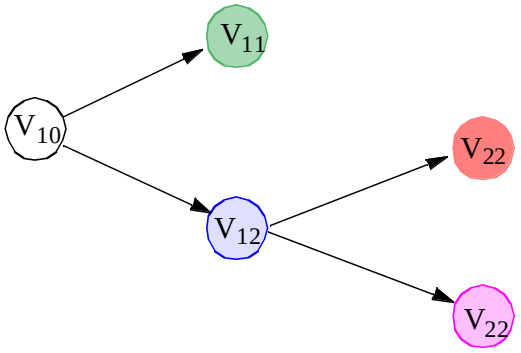
- transparente Abbildung zwischen physischem Speicher und virtuellem Speicher
- *Pointer Swizzling*: automatische Adressumsetzung
  OID <-> Adressen des Hauptspeichers (virtuellen Speichers)
- sehr schnelle Navigation auch in Client/Server-Umgebungen (µs statt ms)
- besonders wichtig für interaktive Designaufgaben
  - CAD-Anwendungen erfordern z.B. 10^5 Objektreferenzen pro sec

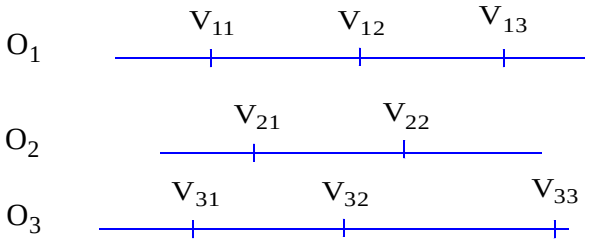
Verwaltung von Multimedia-Objekten

- integrierte Verwaltung von Text-, Bild-, Audio-, Video- ... Daten
- Einsatz von BLOBs (Binary Large Objects) bzw. "long fields"
  - Speichern von großen, unstrukturierten Bytefolgen (z.B. Rasterbilder)
  - i.a. nur einfache Operationen erforderlich
- Einsatz von Multimedia-ADT
  - zugeschnittene Operationen
  - Verwendung von ADT-Funktionen in DB-Anfragen
  - explizite Repräsentation von Objektstruktur (z.B. bei Texten/Dokumenten und Geo-Objekten)
- hohe Leistungsanforderungen an DBMS
  - Erweiterbarkeit des DBS um neue Datentypen und Operationen
  - große Datenmengen, komplexe Operationen
  - Erweiterbarkeit erforderlich bei Query-Optimierung, Indexstrukturen, Pufferverwaltung, Externspeicherverwaltung ...
  - parallele Anfrageverarbeitung auf neuen Datentypen

Unterstützung von Versionen und Alternativen

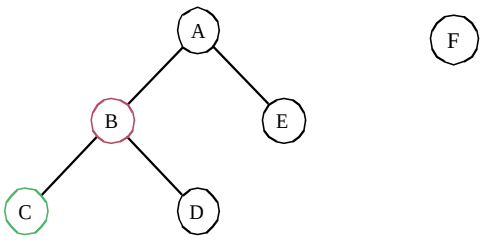
- wichtig v.a. für Entwurfsumgebungen (CAD, CASE)
- Verwaltung von Versionsgraphen
- *Konfiguration*: Menge der Versionen, die ein aktuelles Produkt bilden (-> Konfigurationsverwaltung)





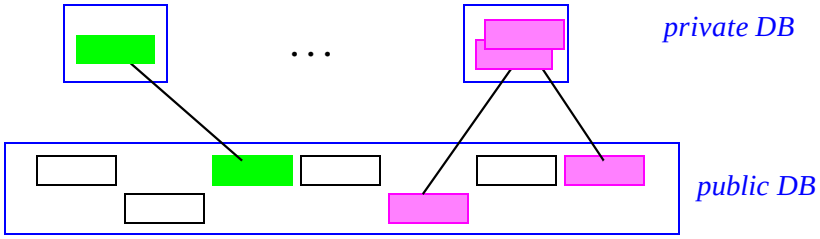
Unterstützung erweiterter Transaktionsmodelle

- Traditionelles Transaktionskonzept: ACID (Atomicity, Consistency, Isolation, Durability)
- Beschränkungen
  - auf kurze Transaktionen zugeschnitten, Probleme mit "lang-lebigen" Aktivitäten
  - Alles-oder-Nichts-Eigenschaft oft inakzeptabel: hoher Arbeitsverlust
  - Isolation: Leistungsprobleme durch "lange" Sperren; fehlende Unterstützung zur Kooperation
  - keine Binnenstruktur ("flache" Transaktionen)
  - keine Unterstützung zur Parallelisierung
  - fehlende Benutzerkontrolle
- größere Flexibilität durch geschachtelte Transaktionen: Hierarchie von (Sub-)Transaktionen
  - ACID-Eigenschaften gelten nur für Wurzel (Top-Level-) Transaktion
  - Sub-Transaktionen: parallele Ausführbarkeit, isolierte Rücksetzbarkeit, Vererbung von Sperren



DB-Verarbeitung in Entwurfsumgebungen

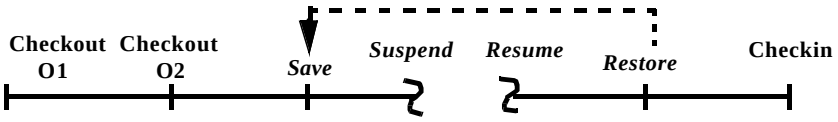
- Merkmale
  - Lange Dauer von Entwurfsvorgängen (Wochen/Monate)
  - Kontrollierte Kooperation zwischen mehreren Entwerfern
  - Unterstützung von Versionen
  - Benutzerkontrolle (nicht-deterministischer Ablauf)
- Checkout/Checkin-Modell
  - 1 public DB, n private DB (Änderungen in privater DB)
  - CHECKOUT: Kopieren des Entwurfsobjektes (public DB → private DB)
  - CHECKIN: (atomares) Einbringen der Änderungen am Ende der Entwurfstransaktion (private → public DB)



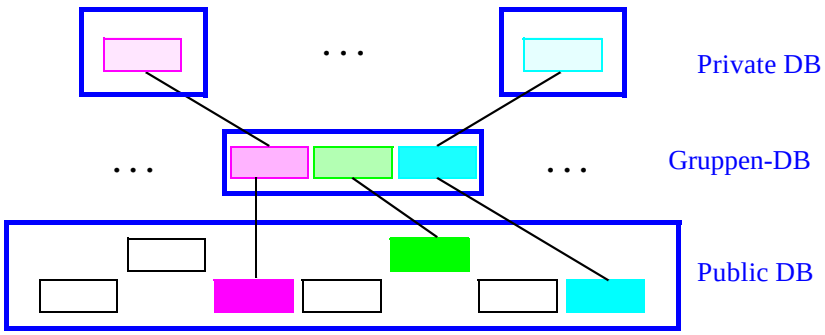
- nach CHECKOUT bleiben Objekte in der public DB gesperrt (permanente Langzeitsperren)
  - keine parallelen Änderungen
  - Lesen der ungeänderten Objekte weiterhin möglich
  - Verarbeitung im Einbenutzerbetrieb

Erweiterungen des Checkout/Checkin-Modells

- Einführung von transaktionsinternen Rücksetzpunkten
  - benutzerdefinierte Savepoints (Operationen SAVE/RESTORE)
  - SUSPEND/RESUME-Operationen zur Unterbrechung/Fortführung der Verarbeitung



- Austausch vorläufiger Entwurfsdaten über Gruppen-DB: 2-stufiges Checkout/Checkin-Konzept



Zusammenfassung

- OODBS / ORDBS unterstützen
  - komplexe Objekte und Objektidentität
  - Erweiterbarkeit bezüglich Datentypen und Verhalten (BDTs, BDFs)
  - Typhierarchien und Vererbung
- Benutzer hat zwischen Objekten und Datenwerten zu unterscheiden
  - Objekte werden durch Werte beschrieben
  - sie sind i.a. explizit zu erzeugen und löschen
- Kapselung vs. Flexibilität (Ad-hoc-Anfragemöglichkeiten)
- Einbettung von DB-Zugriffen (impedence mismatch) vs. Integration / operationale Vollständigkeit
- OODBS / ORDBS eignen sich für anspruchsvolle Anwendungen
- Entwurfsanwendungen (CAD ...):
  - schnelle Objektnavigation
  - lange Verarbeitungsvorgänge -> Checkout/Checkin-Modell
  - Versionen
  - Kooperation

Zusammenfassung (2)

- Es gibt unterschiedliche Datenmodelle
  - OODBS basierend auf objekt-orientierten Programmiersprachen, Bsp.: ODMG-Modell
  - objekt-relationale Ansätze, Bsp.: NF2, SQL99
- Schwächen von OODBS gegenüber ORDBS
  - unzureichende Mehrsprachenfähigkeit
  - unzureichende Kompatibilität zu SQL
  - ein allgemeines Sichtenkonzept ist nicht verfügbar
  - z.T. unzureichende mengenorientierte Anfragemöglichkeiten
  - geringe Marktbedeutung
- Vorteile
  - nur eine (objekt-orientierte) Programmierschnittstelle
  - keine Konversionen von Datentypen
  - operationale Vollständigkeit
  - hohe Leistung für navigierende Zugriffe
  - gute Verträglichkeit mit C++ bzw. Java

5. Objektrelationale DBS / SQL:1999

- Objektrelationale DBS
- Überblick zu SQL:1999
  - Einführung
  - Large Objects (BLOBS, CLOBS, NCLOBS)
  - Neue Typkonstruktoren: ROW, ARRAY, REF
  - Benutzerdefinierte Typen und Funktionen (UDTs, UDFs)
  - DISTINCT-Typen und strukturierte Datentypen
  - Typhierarchien / Tabellenhierarchien
  - Rekursive Anfragen
- ORDBS-Produkte
  - DB2
  - Oracle

Objektrelationale DBS (ORDBS)

- Erweiterung des relationalen Datenmodells und SQL um Objekt-Orientierung
- Erweiterbarkeit
  - benutzerdefinierte Datentypen (u.a. Multimedia-Datentypen)
  - benutzerdefinierte Funktionen
- komplexe, nicht-atomare Attributtypen (z.B. relationenwertige Attribute)
- Bewahrung der Grundlagen relationaler DBS
  - deklarativer Datenzugriff
  - Sichtkonzept etc.
- Standardisierung beginnend mit SQL:1999

SQL-Standardisierung

1986 SQL86
- DDL-Anweisungen für Relationen, Sichten, Zugriffsrechte
- DML-Anweisungen für Standard-Operationen
- keine Integritätszusicherungen

1989 SQL89 (120 Seiten)
- Revision von SQL86
- DEFAULT-Klausel (Wertzuweisung, verschieden von NULL)
- CHECK-Bedingung (rudimentäres Domain-Konzept)
- Basiskonzept der Referentiellen Integrität (Referenzen auf Primärschlüssel und Schlüsselkandidaten)

1992 SQL92 (SQL2)
- Entry Level: ~ SQL89 + gerinfügige Erweiterungen und Korrekturen)
- Intermediate Level: Dynamic SQL, Join-Varianten, CASCADE DELETE, Domains ...
- Full Level (580 Seiten): CASCADE UPDATE, Subquery in CHECK, Assertions, DEFERRED ...

1995/96 Nachträge zu SQL-92: Call-Level-Interface / Persistent Stored Modules (Stored Procedures)

1999 SQL:1999 (SQL3), knapp 2000 Seiten †

2003 SQL4 (SQL:2003)

† A. Eisenberg, J. Melton: SQL:1999, formerly known as SQL3, ACM SIGMOD Record, 1999 (1), March 1999

Aufbau des SQL-Standards

Part 1: SQL/Framework (beschreibt Aufbau des Standards)

Part 2: SQL/Foundation
- erweiterbares Typsystem (benutzerdef. Datentypen und Funktionen, ...), Vererbung, Overloading
- Trigger, Rekursion, asynchrone Ausführung von Operationen, Savepoints, Rollen, . . .

Part 3: SQL/CLI (Call Level Interface)

Part 4: SQL/PSM (Persistent Storage Modules)
- gespeicherte Prozeduren / Funktionen (stored procedures)
- operationale Vollständigkeit durch zusätzliche Kontrollanweisungen; Ausnahmebehandlung

Part 5: SQL/Bindings

- Weitere Teile bereits fertiggestellt bzw. in Vorbereitung:
  - Part 9: SQL/MED (Management of External Data)
  - Part 10: SQL/OLB (Object Language Binding)
  - Part 13: SQL/JRT (SQL Routines and Types using the Java Programing Language)
  - Part 11: SQL/Schemata: Information and Definition Schemas
  - Part 14: SQL/XML: XML-related Specifications
- Separater Standard: SQL/MM (SQL Multimedia and Application Packages)
  - Full Text, Spatial, Still-Image

Typsystem von SQL:1999

- erweiterbares Typkonzept
  - vordefinierte Datentypen
  - konstruierte Typen (Konstruktoren): REF, Tupel-Typen (ROW-Typ), Kollektionstyp ARRAY
  - benutzerdefinierte Datentypen (User-Defined Types, UDT): Distinct Types und Structured Types
- Definition von UDTs unter Verwendung von vordefinierten Typen, konstruierten Typen und vorher definierten UDTs
- UDTs unterstützen
  - Kapselung
  - Vererbung (Subtypen)
  - Overloading
- alle Daten werden weiterhin innerhalb von Tabellen gehalten
  - Definition von Tabellen auf Basis von strukturierten UDTs möglich
  - Bildung von Subtabellen (analog zu UDT-Subtypen)
- neue vordefinierte Datentypen
  - Boolean (Werte: TRUE, FALSE, UNKNOWN)
  - Large Objects (Binärdarstellung bzw. Texte)

Large Objects

- 3 neue Datentypen:
  - BLOB (Binary Large Object)
  - CLOB (Character Large Object): Texte mit 1-Byte Character-Daten
  - NCLOB (National Character Large Objects): ermöglicht 2-Byte Character-Daten für nationale Sonderzeichen (z.B. Unicode)
- Verwaltung großer Objekte im DBS (nicht in separaten Dateien)
  - umgeht große Datentransfers und Pufferung durch Anwendung
  - Zugriff auf Teilbereiche

```
CREATE TABLE Pers   (PNR          INTEGER,
                     Name         VARCHAR (40),
                     Vollzeit     BOOLEAN,
                     Lebenslauf   CLOB (75K),
                     Unterschrift BLOB (1M),
                     Bild         BLOB (12M))
```

- indirekte Verarbeitung großer Objekte über Locator-Konzept (ohne Datentransfer zur Anwendung)

Large Objects (2)

- unterstützte Operationen
  - Suchen und Ersetzen von Werten (bzw. partiellen Werten)
  - LIKE-Prädikate, CONTAINS, POSITION, SIMILAR
  - Konkatenation ||, SUBSTRING, LENGTH, IS [NOT] NULL ...

- einige Operationen sind auf LOBs nicht möglich
  - Schlüsselbedingung
  - Kleiner/Größer-Vergleiche
  - Sortierung (ORDER BY, GROUP BY)

Tupel-Typen (ROW-Typen)

- Tupel-Datentyp (row type)
  - Sequenz von Feldern (fields), bestehend aus Feldname und Datentyp:
    `ROW (<feldname1> <datentyp1>, <feldname2> <datentyp2>, ...)`
  - eingebettet innerhalb von Typ- bzw. Tabellendefinitionen
- Beispiel

```
CREATE TABLE  Pers   (PNR        int,
                      PName      ROW ( VName VARCHAR (20),
                                       NName VARCHAR (20)),
                •••);
ALTER TABLE Pers ADD COLUMN Anschrift  ROW (Strasse   VARCHAR (40),
                                           PLZ       CHAR (5),
                                           Ort       VARCHAR (40) );
```

- Geschachtelte Rows möglich
- Operationen
  - Erzeugung mit Konstruktor ROW: ROW („Peter“, „Meister“)
  - Zugriff auf Tupelfeld mit Punktnotation:
  - Vergleiche

ARRAY-Kollektionstyp

- in SQL:1999 wird nur Kollektionstyp ARRAY unterstützt (kein Set, List ...)
- Spezifikation: `<Elementtyp> ARRAY [<maximale Kardinalität>]`
  - Elementtypen: alle SQL99-Datentypen (z.B. Basisdatentypen, benutzerdefinierte Typen)
  - jedoch keine geschachtelten (mehrdimensionale) Arrays !

```
CREATE TABLE Mitarbeiter
        (   Name        ROW ( VName VARCHAR (20), NName VARCHAR (20)),
            Sprachen    VARCHAR(15) ARRAY[8], ... )
```

- Array-Operationen
  - Typkonstruktor ARRAY, Element-Zugriff
  - Bildung von Sub-Arrays, Konkatenation (||) von Arrays
  - CARDINALITY
  - UNNEST (Entschachtelung)

```
INSERT INTO Mitarbeiter (Name, Sprachen)
VALUES ( ROW („Peter“, „Meister“), ARRAY [„Deutsch“, „Englisch“] )
UPDATE Mitarbeiter SET Sprachen[3]=“Französisch“ WHERE Name.NName=“Meister“
```

Welche Sprachen kennt der Mitarbeiter „Meister“?

Syntax der UDT-Definition (vereinfacht)

CREATE TYPE <UDT name> [[<subtype clause>] [AS <representation>] [<instantiable clause>]
<finality> [<reference type specification>] [<cast option>]
[<method specification list>]

<subtype clause> ::= UNDER <supertype name>
<representation> ::= <predefined type> | [**(** <member> , ... **)**]
<instantiable clause> ::= INSTANTIABLE | NOT INSTANTIABLE
<finality> ::= FINAL | NOT FINAL
<member> ::= <attribute definition>
<method specification> ::= <original method specification> | <overriding method specification>
<original method specification> ::= <partial method specification> <routine characteristics>
<overriding method specification> ::= OVERRIDING <partial method specification>
<partial method specification> ::= [INSTANCE | STATIC | CONSTRUCTOR] METHOD
<routine name> <SQL parameter declaration list> <returns clause>

- Definition von DISTINCT-Typen (<representation> entspricht <predefined type>) und strukturierten Typen
- Aufbau von Typhierarchien (Subtyp-Klausel): nur einfache Vererbung
- Spezifikation von Datenelementen (Attributen) und Methoden

DISTINCT-Typen *(Umbenannte Typen)*

- dienen der Wiederverwendung schon definierter Datentypen
  - einfache UDT, welche auf einem vordefinierten Datentyp basieren
  - mit neuen Typnamen versehen
  - DISTINCT-Typen sind unterscheidbar von dem darunterliegenden (und verdeckten) Basis-Typ

```
CREATE TYPE Dollar AS REAL FINAL;
CREATE TYPE Euro AS REAL FINAL;
CREATE TABLE Dollar_SALES (Custno INTEGER, Total Dollar, ...)
CREATE TABLE Euro_SALES (Custno INTEGER, Total Euro, ...)
SELECT D.Custno
FROM Dollar_SALES D, Euro_SALES E
WHERE D.Custno = E.Custno AND
```

- keine direkte Vergleichbarkeit mit Basisdatentyp (Namensäquivalenz)
- Verwendung von Konversionsfunktionen zur Herstellung der Vergleichbarkeit (CAST)

```
UPDATE Dollar_SALES SET Total = Total * 1.16
```

- keine Vererbung (FINAL)

Benutzerdefinierte strukturierte Typen: Beispiel

```
CREATE TYPE Adresse
        (Strasse        VARCHAR (40),
         PLZ            CHAR (5),
         Ort            VARCHAR (40) ) NOT FINAL;

CREATE TYPE PersonT
        (Name           VARCHAR (40),
        Anschrift       Adresse,
        PNR             int,
        Manager         REF (PersonT),
        Gehalt          REAL,
        Kinder          REF (PersonT) ARRAY [10] )
        INSTANTIABLE
        NOT FINAL
        INSTANCE METHOD  Einkommen () RETURNS REAL);

CREATE TABLE Mitarbeiter OF PersonT (Manager WITH OPTIONS SCOPE Mitarbeiter ... )
```

- Löschen einer Typ-Definition: DROP TYPE Adresse RESTRICT

Typisierte Tabellen

- Tabellen: Einziges Konzept (container), um Daten persistent zu speichern
- SQL99: Attribute können Array-, Tupel-, Objekt- oder Referenz-wertig sein
- Typ einer Tabelle kann durch strukturierten Typ festgelegt sein: typisierte Tabellen (Objekttabellen)
  - Zeilen entsprechen Instanzen (Objekten) des festgelegten Typs
  - OIDs sytemgeneriert, benutzerdefiniert oder aus Attribut(en) abgeleitet

```
CREATE TABLE Tabellenname OF StrukturierterTyp [UNDER Supertabelle] [(

        [ REF IS oid USER GENERATED |
          REF IS oid SYSTEM GENERATED |
          REF IS oid DERIVED (Attributliste) ]

        [Attributoptionsliste]

  ) ]

Attributoption:        Attributname WITH OPTIONS Optionsliste

Option:                SCOPE TypisierteTabelle | DEFAULT Wert |
Integritätsbedingung
```

- Bezugstabelle für REF-Attribute erforderlich (SCOPE-Klausel)

REF-Typen

- dienen zur Realisierung von Beziehungen zwischen Typen bzw. Tupeln (OID-Semantik)

```
<reference type> ::= REF ( <user-defined type> ) [ SCOPE <table name>]
                [REFERENCES ARE [NOT] CHECKED]
                [ON DELETE <delete_action> ]
<delete_action> ::= NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT
```

  - jedes Referenzattribut muss sich auf genau eine Tabelle beziehen (SCOPE-Klausel)
  - nur typisierte Tabellen (aus strukturierten UDT abgeleitet) können referenziert werden
  - referentielle Integrität kann überwacht werden (Default: keine Prüfung)
- Beispiel

```
CREATE TABLE Abteilung OF AbteilungT;

CREATE TABLE Person
        (PNR        INT,
         Name       VARCHAR (40),
         Abt        REF (AbteilungT) SCOPE Abteilung,
         Manager    REF (PersonT) SCOPE Mitarbeiter,
         Anschrift  Adresse, ...
  );
```

REF-Typen (2)

- Verwendung von Referenzen innerhalb von Pfadausdrücken: Dereferenzierung über ->

```
SELECT P.Name
FROM PERS P
WHERE P.Manager->Name = “Schmidt” AND P.Anschrift.Ort = “Leipzig”
```

- Referenzauflösung auch mittels DEREF-Operator (liefert Attributwerte eines referenzierten Objekts):

```
SELECT DEREF (P.Manager)
FROM PERS P
WHERE P.Name= “Meister”
```

- nur Top-Level-Tupel in Tabellen können referenziert werden
  - Referenzwert bleibt während Lebenszeit des referenzierten Tupels unverändert
  - DBS-weit eindeutige Referenzwerte
  - keine Wiederverwendung von Referenzwerten

UDT-Einsatz

- UDTs können überall verwendet werden, wo vordefinierte Datentypen in SQL zum Einsatz kommen
  - als Attributtyp anderer UDTs
  - als Parametertyp von Funktionen und Prozeduren
  - als Typ von SQL-Variablen
  - als Typ von Domains und Spalten von Tabellen (table columns)
  - als Basistyp für typisierte Tabellen (Objekttabellen)

```
CREATE TABLE  Pers    (Stammdaten   PersonT,
                       Lebenslauf   CLOB (75K),
                       Bild         BLOB (12M));

SELECT    Stammdaten.Name
FROM      Pers
WHERE     Stammdaten.Gehalt < 30000 AND POSITION (“Diplom” IN Lebenslauf) > 0
```

Funktionen und Methoden

- Routinen (Funktionen und Prozeduren) als eigenständige Schemaobjekte bereits in SQL/PSM

	SQL-Routinen	Externe Routinen
Aufruf in SQL (SQL-invoked routines)	SQL-Funktionen (inkl. Methoden) und SQL-Prozeduren	externe Funktionen und Prozeduren
Externer Aufruf	nur SQL-Prozeduren (keine Funktionen)	(nicht relevant für SQL)

- Methoden: beziehen sich auf genau einen UDT
- Realisierung aller Routinen und Methoden über prozedurale SQL-Spracherweiterungen oder in externer Programmiersprache (C, Java, ...)
- Namen von SQL-Routinen und Methoden können überladen werden (keine Eindeutigkeit erforderlich)
  - bei SQL-Routinen wird zur Übersetzungszeit anhand der Anzahl und Typen der Parameter bereits die am “besten passende” Routine ausgewählt
  - bei Methoden wird dynamisches Binden zur Laufzeit unterstützt

UDT-Kapselung

- vollständige Kapselung
- Attributzugriff erfolgt ausschließlich über Methoden
  - keine Unterscheidung zwischen Attributzugriff und Methodenaufruf
  - sichtbare UDT-Schnittstelle besteht aus Menge von Methoden
- für jedes Attribut wird implizit eine Methode zum Lesen (Observer) sowie zum Ändern (Mutator) erzeugt
- implizit erzeugte Methoden für UDT Adresse

Observer-Methoden:

```
METHOD   Strasse ()   RETURNS VARCHAR (40);
METHOD   PLZ ()       RETURNS CHAR (5);
METHOD   Ort ()       RETURNS VARCHAR (40);
```

Mutator-Methoden:

```
METHOD   Strasse (VARCHAR (40))   RETURNS Adresse;
METHOD   PLZ (CHAR(5))            RETURNS Adresse;
METHOD   Ort (VARCHAR (40))       RETURNS Adresse;
```

Punkt (dot)-Notation

- Manipulation von UDT-Instanzen durch Aufruf von UDT-Methoden (bzw. Attributzugriff)
- Methodenaufruf kann an jeder Stelle auftreten, wo ein skalarer Wert in SQL möglich ist
  - innerhalb von INSERT, UPDATE, DELETE, SELECT, ...
  - geschachtelte Methodenaufrufe und Pfadausdrücke sind möglich
- für Attributzugriff kann wahlweise Methodenaufruf oder Punkt-Notation (.) verwendet werden
  - Punkt-Notation entspricht "syntaktischem Zucker" für Methodenaufrufe

a.x	ist äquivalent zu	a.x ()
SET a.x = y	ist äquivalent zu	a.x (y)

- Beispiel

```
DECLARE p      PersonT;
 ...
SET x = p.Gehalt;
SET p.Gehalt = 2500.66;
SET y = p.Anschrift.Ort;
SET p.Anschrift.PLZ = "04109";
```


Initialisierung von UDT-Instanzen

- DBS stellt für jeden instantiierbaren UDT Default-Konstruktor zum Erzeugen von UDT-Instanzen bereit

  CONSTRUCTOR METHOD PersonT () RETURNS PersonT
  - Parameterlos, kann nicht überschrieben werden
  - besitzt gleichen Namen wie zugehöriger UDT
  - belegt jedes der UDT-Attribute mit dem in der UDT-Definition vorgesehenen Defaultwert (falls vorhanden)
- Initialisierung kann durch Mutator-Methoden ergänzt werden
- Benutzer kann beliebige Anzahl eigener Konstruktoren definieren, um Initialisierung beim Anlegen des Objektes zu erreichen (über Parameter)

```
CREATE METHOD PersonT(n varchar(40), a Adresse) FOR PersonT RETURNS PersonT
BEGIN
        DECLARE p PersonT;
        SET p = PersonT();
        SET p.Name = n;
        SET p.Anschrift = a;
        RETURN p;
END;

INSERT INTO Pers
VALUES ( PersonT ("Peter Schulz", Adresse ("Seestraße 12", "50321", "Köln")), NULL, NULL)
```

Generalisierung / Spezialisierung

- Spezialisierung in Form von Subtypen und Subtabellen
- in SQL:1999 nur Einfachvererbung
- es muss keine alleinige Wurzel in der Generalisierungshierarchie geben
- Subtyp
  - erbt alle Attribute und Methoden des Supertyps
  - kann eigene zusätzliche Attribute und Methoden besitzen
  - Methoden von Supertypen können überladen werden (Overriding)
- Supertyp muss selbst strukturierter Typ sein

CREATE TYPE PersonT (PNR INT, Name CHAR (20), Grundgehalt REAL, ...) NOT FINAL
CREATE TYPE Techn-AngT **UNDER** PersonT (Techn-Zulage REAL, ...) NOT FINAL
CREATE TYPE Verw-AngT **UNDER** PersonT (Verw-Zulage REAL, ...) NOT FINAL

- Subtabellen (Teilmengenbildung) analog zu Typhierarchien

CREATE TABLE Pers OF PersonT (PRIMARY KEY PNR)
CREATE TABLE Techn-Ang OF Techn_AngT **UNDER** Pers
CREATE TABLE Verw-Ang OF Verw-AngT **UNDER** Pers

Substituierbarkeit

- Instanz eines Subtyps kann in jedem Kontext genutzt werden, wo Instanz eines Supertyps nutzbar ist
  - Eingabeargumente für Funktionen und Prozeduren, deren formale Parameter auf dem Supertyp definiert sind
  - Rückgabe als Funktionsergebnis, für das Supertyp als formaler Typ definiert wurde
  - Zuweisungen zu Variablen oder Attributen des Supertyps
- Verwendung von Supertypen in Tabellen kann somit zu heterogenen Tupelmengen führen

```
CREATE TABLE Pers OF PersonT( ...)
INSERT INTO Pers VALUES (8217, 'Hans', 89500 ...)
INSERT INTO Techn_Ang
            VALUES (PersonT (5581, 'Rita', ...), 4300)
INSERT INTO Verw_Ang
            VALUES (PersonT (3375, 'Anna', ...), 5400)
```

Tabelle: ***Pers***

PNR	Name	Techn-Zulage	Verw-Zulage
8217	Hans ...		
5581	Rita ...	4300	
3375	Anna ...		5400
1463	Elke ...		
...			

- Homogene Ergebnismengen mit ONLY-Prädikat bzw. durch Zugriff auf Subtabellen

```
SELECT *
FROM ONLY Pers
WHERE Grundgehalt > 80000
```

```
SELECT *
FROM Verw_Ang
WHERE Grundgehalt > 80000
```

Dynamisches Binden

- Overloading (Polymorphismus) von Funktionen und Methoden wird unterstützt

```
CREATE TYPE PersonT (PNR INT,  ... ) NOT FINAL
   METHOD Einkommen () RETURNS REAL,    ...
```

```
CREATE TYPE Techn_AngT UNDER PersonT
  (Techn-ZulageREAL,  ...) NOT FINAL
  OVERRIDING METHOD Einkommen ()
    RETURNS REAL,
  ...
```

```
CREATE TYPE Verw_AngT UNDER PersonT
  (Verw-Zulage REAL,  ... ) NOT FINAL
  OVERRIDING METHOD Einkommen ()
    RETURNS REAL,
  ...
```

- Anwendungsbeispiel einer polymorphen Funktion:

```
CREATE TABLE Pers OF PersonT (...)

SELECT P.Einkommen
FROM Pers P
WHERE P.Name ='Müller';
```

- dynamische Funktionsauswahl zur Laufzeit aufgrund spezifischem Typ

Rekursion

- Berechnung rekursiver Anfragen (z.B. transitive Hülle) über rekursiv definierte Sichten
- Grundgerüst

```
WITH RECURSIVE RekursiveTabelle (...) AS
( SELECT ... FROM Tabelle WHERE ...
  UNION
  SELECT ...From Tabelle, RekursiveTabelle     WHERE ...)
SELECT * From RekursiveTabelle
```

- Beispiel

```
CREATE TABLE Eltern (Kind   CHAR (20), Elternteil  CHAR (20));
```

Alle Vorfahren von „John“ ?

```
WITH RECURSIVE Vorfahren (K, V) AS
( SELECT * FROM Eltern
    UNION
  SELECT E.Kind, V.V
 FROM Vorfahren V, Eltern E
 WHERE E.Elternteil = V.K)
SELECT V FROM Vorfahren WHERE K = "John"
```

Rekursion (2)

- Syntax (WITH-Klausel)

  <query expression> ::= [WITH [RECURSIVE] <with list>] <query expression body>

  <with list element> ::= <query name> [**(** <with column list> **)**] AS **(** <query expression> **)** [SEARCH <search order> SET <sequence column>] [CYCLE <cycle column list>]

  <search order> ::= DEPTH FIRST BY <sort specification list> | BREADTH FIRST BY <sort specification list>

- Merkmale
  - verschiedene Suchstrategien (Depth First, Breadth First)
  - Zyklusbehandlung
  - lineare oder allgemeine Rekursion
- Naive Auswertungsstrategie
  - Ergebnistabelle („Vorfahren“) ist monoton steigend (neue transitive Verbindungen kommen hinzu)
  - Sobald sich Ergebnistabelle nicht mehr ändert, d.h. keine neuen transitiven Verbindungen abgeleitet werden können, ist Endergebnis erreicht
  - Problem: Zyklen werden nicht erkannt
- Zyklenbehandlung durch Einschränkung der Rekursionstiefe
  - z.B. durch zusätzliche Attribute, die Schachtelungstiefe codieren (z.B. Attribut in Vorfahren-Tabelle, welches Generationen mitzählt)

SQL/MM (Multimedia and Applications Packages)‡

- Multimedia/Anwendungs-Erweiterungen auf Basis von SQL:1999
  - Full Text (Datentypen und Funktionen zur Volltextsuche)
  - Spatial (räumliche Daten)
  - Still Image (Bilder)
  - Data Mining
- SQL/MM Full Text
  - neue Datentypen `FullText` und `FT_Pattern` (Suchmuster)
  - Suchausdruck: `Contains`
  - Ähnlichkeitssuche: `Rank` (Grad der Überseinstimmung)
  - Kontextsuche: `in same sentence as, in same paragraph as`
  - linguistische Suche: `stemmed form of`

Create Table Person (PNR INT, Name VARCHAR(40), Lebenslauf FULLTEXT, ...)

Select Name From Person P
Where P.Lebenslauf.`Contains` (‘„Java“ `in same paragraph as` „SQL“’) = 1 and
P.Lebenslauf.`Rank` („Datenbank“) > 0.25

‡ J. Melton, A. Eisenberg: *SQL Multimedia and Application Packages (SQL/MM).* ACM Sigmod Record, SIGMOD Record 30(4): 97-102 (2001)

SQL/MM (2)

- SQL/MM Still Image
  - Datentyp `SI_StillImage` für digitale Rasterbilder (interne Speicherung als BLOB)
  - Methoden für Formatkonversionen, Skalierung, Rotation, Ausschnittbildung, Thumbnail-Erzeugung ...
  - Unterstützung der Suche durch Feature-Datentypen:
    `SI_AverageColor` (durchschnittlicher Farbwert),
    `SI_ColorHistogram` (Farbverteilung),
    `SI_PositionalColor` (Farbverteilung bezüglich Bildlage),
    `SI_Texture` (Angaben zu Auflösung, Kontrast etc.)
  - Suchmethode `SI_Score` (liefert Maß der Übereinstimmung)

Select * From Produkte P
Where `SI_ColorHistogram (beispiel).SI_Score (P.Bild) > 0.8`

- SQL/MM Spatial
  - zunächst nur 0-, 1- und 2-dimensionale Objekte (Punkte, Linien, Flächenformen)
  - Bereitstellung standardisierter Typhierarchien wie `ST_Geometry` (Subtypen `ST_Point`, `ST_Curve`, `ST_MultiPolygon`) und `ST_SpatialRefSys` (räumliche Referenzsysteme)
  - Methoden zur Erzeugung von Geo-Objekten und Berechnungen (intersect, overlap, area ...)

DB2 Universal Database (IBM)

- objekt-relationale Features (V8):
  - BLOB, CLOB, DBCLOB (entspricht NCLOB in SQL99, DB = Double Byte) (max. 2GB)
  - User-defined data types (UDT): Distinct types, Structured types
  - geschachtelte Tupel-Definitionen erlaubt
  - Definition von Methoden / UDFs
  - Vererbung (Spezialisierung)
  - Zuweisung an Attribute, Tabellen, Views
- Rekursion
- **nicht unterstützt:** Kollektionstypen (auch keine Arrays)

```
CREATE TYPE angestellter AS
    (name        VARCHAR(32),
     id          INT,
     manager     REF (angestellter),
     gehalt      DECIMAL(10,2))
    REF USING INT
    MODE DB2SQL

CREATE TYPE mgr UNDER angestellter AS
    (bonus       DECIMAL(10,2))
    MODE DB2SQL
```

```
Create Table Eltern (Kind       CHAR (20),
                     Elternteil CHAR (20));

WITH Vorfahren (Generation, K, V) AS
        (   Select 1, Kind, Elternteil
            From Eltern
            Where Kind = "John"
        UNION ALL
            Select V.Generation+1, E.Kind, V.V
            From Vorfahren V, Eltern E
            Where E.Elternteil = V.K and
                  V.Generation < 3)
Select V From Vorfahren
```

DB2 (2)

- Erweiterbarkeit über Extenders
  - benutzerdefinierte Datentypen
  - benutzerdefinierte Routinen
  - benutzerdefinierte Zugriffspfade (Indexstrukturen)
- Verfügbare Extenders
  - DB2 XML Extender
  - DB2 Text Extender
  - DB2 Image Extender
  - DB2 Audio Extender
  - DB2 Video Extender
  - DB2 Video Charger Extender
  - DB2 Spatial Extender
  - DB2 Net Search Extender
  - EcoWin Time Series Solution
  - Fillmore SQL Expander
  - Protegrity Secure Data
  - ...

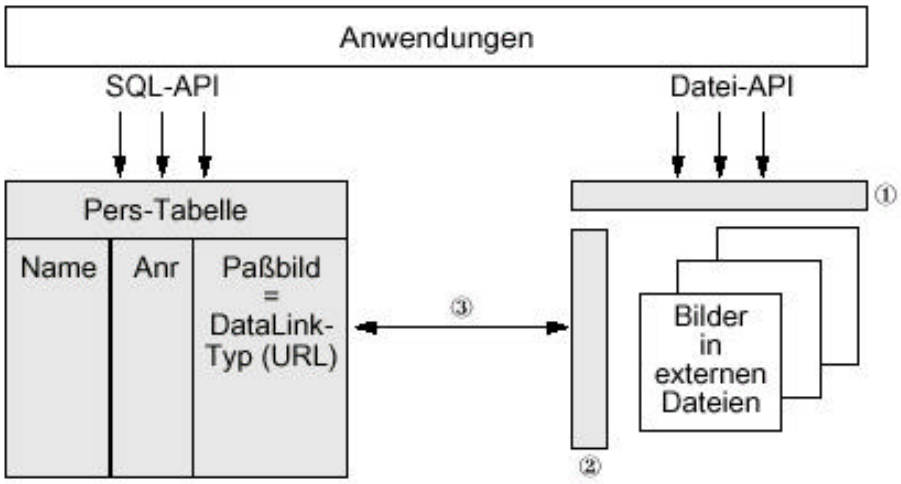
DB2 (3)

- DB2 Image Extender
  - neue Datentypen für Bilder mit spezifischen Attributen (Größe, Format, Breite, Höhe, Farben, ...)
  - Import und Export sowie Formatkonvertierung
  - Anfragen über Bildinhalt (query by image content - QBIC)
- DB2 Text Extender
  - Linguistische Indexierung und Suche für 22 Sprachen
  - Boolesche-, Freitext- und Fuzzy-Suche
  - Ranking der Suchergebnisse
  - Parser und Filter für verschiedene Dokumentformate
- DB2 Spatial Extender
  - Erweiterung von DB2 zu einem geografischen DBMS
  - Modellierung räumlicher Daten (13 UDTs)
  - Index auf räumlichen Daten
  - UDFs (z.B. Abstand)
  - Werkzeuge zur Administrierung, Datenaustausch
  - Browser zur Visualisierung von Anfragen auf räumliche Daten

```
SELECT titel, isbn, kommentar
FROM buecher
WHERE
CONTAINS (titelhandle,
'STEMMED FORM OF GERMAN
                "Nacht"') = 1
AND
CONTAINS (kommentarhandle,
'SYNONYM FORM OF GERMAN
                "spannend"') = 1
```

DB2 (4)

- Verwendung des DATALINK-Datentyps in SQL-Tabellendefinitionen möglich
  - DATALINK-Attribut referenziert Objekt, das außerhalb einer Datenbank gespeichert ist
  - DATALINK-Werte codieren den Namen eines Data Links-Servers, der die Datei enthält, sowie den Dateinamen als Uniform Resource Locator (URL)
  - Behandlung eines DATALINK-Werts, als sei Objekt in Datenbank gespeichert (mit Integritäts- und Zugriffskontrolle)



DB2 (5)

- Beispiel

```
CREATE TABLE Pers (
    Name VARCHAR (30), Anr INTEGER,
    Paßbild DATALINK (200)
                LINKTYPE URL
                FILE LINK CONTROL
                INTEGRITY all
                READ PERMISSION DB
                WRITE PERMISSION blocked
                RECOVERY yes
                ON UNLINK restore);
```

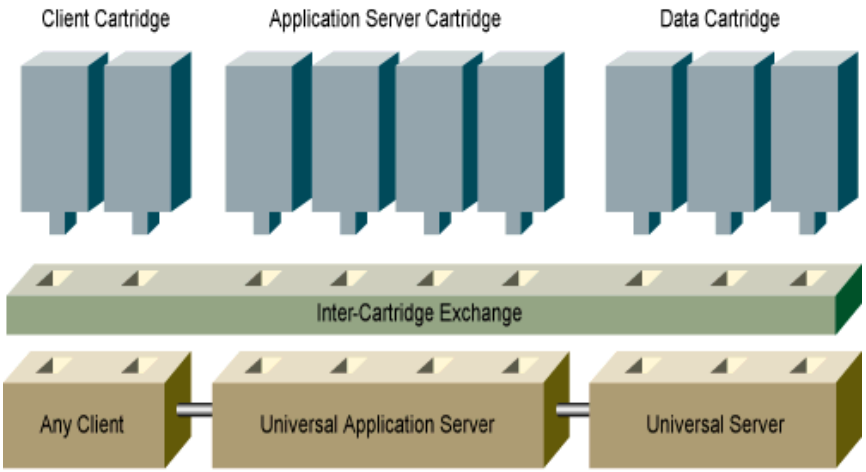
  - File Link Control: Korrektheit der URL wird geprüft
  - Integrity all: Link kann nicht direkt gelöscht werden
  - Read Permission: Dateisystem oder DBS. Autorisierung wird als Token in URL eingebettet
  - Write Permission: bleibt beim Dateisystem oder wird blockiert
  - On Unlink: Datei kann gelöscht oder zur Verwaltung ans Dateisystem zurückgegeben werden

- DATALINK-Standardisierung durch SQL/MED**

** J. Melton et al.: *SQL and Management of External Data.* ACM SIGMOD Record 30(1): 70-77 (2001)

Oracle

- objekt-relationale Features seit Oracle8 (derzeit Version 9i)
- Erweiterbarkeit über Data Cartridges



- Unterstützung benutzerdefinierter Typen, Arrays sowie von (einfach) geschachtelten Tabellen
- Einfache Vererbung von benutzerdefinierten Typen

Oracle (2)

- Neue Basisdatentypen
  - Large Objects: BLOB ("Internes" Binary Large Object), CLOB ("Internes" Character Large Object), BFILE ("Externes" Large Object)
  - Variabler Arraytyp VARRAY
- VARRAY: Geordnete Multimenge mit indiziertem Zugriff
  - Grösse wird bei der Instanzierung festgelegt
  - Alle Elemente besitzen denselben Typ
  - Elementindex als Iterator
  - Achtung: VARRAY-Werte nicht direkt in SQL, sondern nur in PL/SQL nutzbar
  - Syntax:

```
CREATE TYPE Arraytypname AS VARRAY(Grösse) OF Elementtyp

CREATE TYPE TelefonArrayTyp AS VARRAY(30) OF VARCHAR(20);
ALTER TABLE MitarbeiterTupelTabelle ADD Telefone TelefonArrayTyp;
```

Oracle (3)

- Geschachtelte Tabellen
  - Schachtelung auf typisierte Tabellen (“Objekt-Tabellen”) und maximal eine Schachtelungsstufe beschränkt
  - Zugriff über TABLE-Operator
- Beispiel

```
CREATE TYPE PersonT AS OBJECT (
                    PNR             INTEGER,
                    Name            VARCHAR2 (40),
                    Gebdatum        DATE, ...
                    MEMBER FUNCTION Alter RETURN INTEGER ... );
CREATE TYPE PersTabellenTyp AS TABLE OF PersonT;

CREATE TABLE Abt (ANR INTEGER, Angestellte PersTabellenTyp)
NESTED TABLE Angestellte STORE AS AngTabelle;
INSERT INTO Abt VALUES (1, PersTabellenTyp (PersonT (4711, “Müller”, ...),
                    PersonT (4712, “Schulz”, ...), ...)
                ... )

SELECT P.Name
FROM TABLE (SELECT A.Angestellte FROM Abt A WHERE A.ANR=1) P
WHERE P.Alter < 30
```


Zusammenfassung

- SQL:1999-Standardisierung
  - Kompatibilität mit existierenden SQL-Systemen
  - Objektorientierung: Benutzerdefinierte Datentypen und Methoden, Typhierarchien und Vererbung, Objekt-Identität (REF-Typen)
  - erweiterbares Typsystem stellt signifikante Verbesserung der Modellierungsfähigkeiten dar
  - beschränkte Unterstützung von Kollektionstypen (Array)
  - Zahlreiche weitere Fähigkeiten in SQL:1999 (Trigger, Rekursion, • • •)
- SQL/MM: standardisierte Erweiterungen für Fulltext, Still Image, Spatial
- SQL/MED: DBS-Einbindung von externen Daten / Dateien
  - DATALINK-Datentyp ermöglicht referentielle Integrität, Zugriffskontrolle und Logging für Dateien
- Implementierungen objekt-relationaler DBS
  - Beispiele: DB2 Universal Database, Oracle, ...
  - zum Teil noch begrenzte Unterstützung der Objekt-Orientierung (Vererbung, komplexe Objekte)
  - Nutzung nicht-standardisierter Erweiterungen
- Hohe Komplexität für Benutzer und DBS-Implementierung

6. Objektorientierte DBS

- Einführung
- ODMG-Objektmodell
  - Literale vs. Objekte
  - Typen
  - Kollektionen
  - Attribute und Beziehungen (relationships)
- Objektdefinition mit ODL
- Anfragesprache (OQL)
- Beispielimplementierung: FastObjects (Poet)

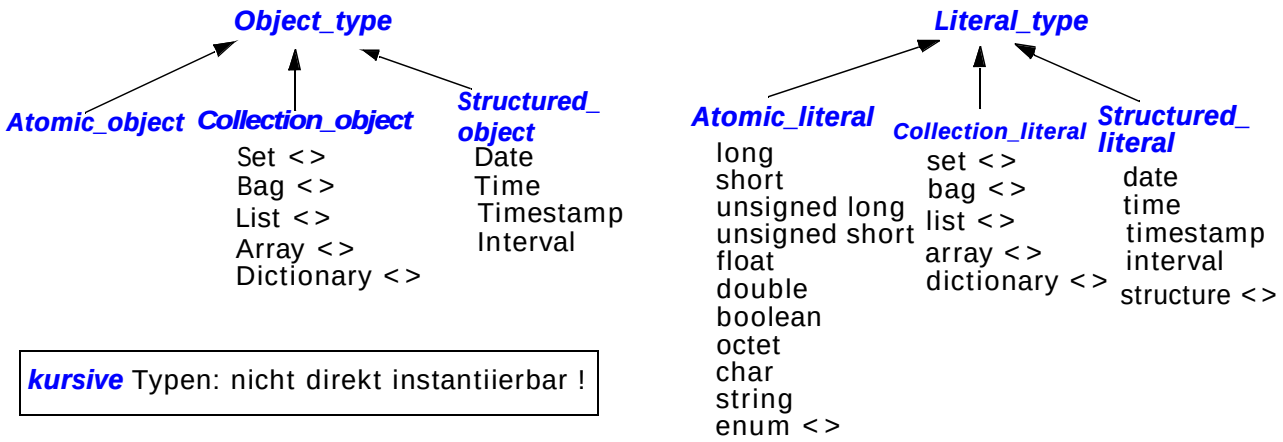
ODMG (Object Data Management Group)*

- gegründet 1991 im Rahmen der OMG (Object Management Group, www.omg.org)
- Ziel: Entwicklung von Standards für objektorientierte DBS
- ODMG-Standardisierung besteht aus
  - Objektmodell (Erweiterung des OMG Kern-Objektmodells)
  - Objekt-Definitionssprache (ODL)
  - Objekt-Anfragesprache (OQL)
  - Sprachbindungen (language bindings) für C++, Smalltalk und Java
- ODMG-Versionen (jeweils in einem Buch beschrieben)
  - 1993 (V1)
  - 1997 (V2)
  - 2000 (V3)
- Cattell, R.: *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann, 2000

* http://www.odmg.org

ODMG-Objektmodell: Überblick

- Unterscheidung zwischen Objekten (haben OID) und Literalen/Werten (haben keinen Objekt-Identifier)
- sowohl Literale als auch Objekte haben einen Typ



kursive Typen: nicht direkt instantiierbar !

- Objekte eines Typs weisen gemeinsame Charakteristika bezüglich Zustand (Attribute, Beziehungen) und Verhalten (Operationen) auf

Typen

- Typen bestehen aus
  - externer Spezifikation: Festlegung der sichtbaren Eigenschaften, Operationen und Ausnahmen (exceptions)
  - einer (oder mehreren) Implementierung(en): sprachabhängige Festlegung der Datenstrukturen und Methoden zur Realisierung der Schnittstelle
- *Typ-Spezifikation* ist implementierungsunabhängig; Beschreibung durch ODL
  - *Interface*-Definitionen: nur Operationen, nicht instanziierbar
  - *Klassen*-Definitionen: Operationen + Zustand (Attribute, Beziehungen), instanziierbar
  - *Literal*-Definitionen: Zustandsbeschreibung eines Literal-Typs

```
interface Auto { int Alter (); ... };
class Person { attribute string Name; ... };
struct Complex { float re; float im; };
```

- *Typ-Implementierung* ist sprachabhängig und im DB-Schema nicht sichtbar
  - Festlegung der Repräsentation (Instanzvariablen) und Methoden
  - Kapselung
  - Mehrsprachenfähigkeit durch Language Bindings

Typen (2)

- Typen können durch Sub-Typen spezialisiert werden (IS-A-Beziehung, Vererbung von Operationen)
  - Definition zusätzlicher Operationen
  - Operationen können überladen werden (Overloading)
- Interfaces
  - Einfach- und Mehrfachvererbung
  - Interfaces können von anderen Interfaces erben/abgeleitet werden, jedoch nicht von Klassen

```
interface PKW : Auto { ... }
```

- Klassen
  - Klassen können von Interfaces abgeleitet werden
  - zusätzliche Zustands-Vererbung (nur Einfachvererbung) für Klassen: EXTENDS-Beziehung

```
class Angestellter EXTENDS Person {
attribute float Gehalt;
attribute date Einstellungstermin; ...
}
```


Objekte

- Objekte eines Typs: gemeinsame Charakteristika bezüglich Zustand und Verhalten
  - Zustand: Werte für bestimmte Eigenschaften (properties), d.h. Attribute und Beziehungen zu anderen Objekten
  - Verhalten: Menge von Operationen
- Objekte
  - explizite Erzeugung neuer Objekte durch new-Konstruktor
  - datenbank-weit eindeutige Objekt-Identifier (→ Referenzierbarkeit)
  - Objekte können optional über Namen angesprochen werden ("Einstiegspunkte" in die Datenbank)
  - Objektlebensdauer: transient oder persistent (einheitliche Verwaltung beider Objektarten ↔ relationale DBS)

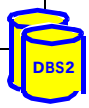
```
interface ObjectFactory {
        Object new ();
}
```

```
interface Object {
        boolean same_as (in Object anObject);   // Identitätstest
        Object copy ();
        void delete (); ...
}
```

- 2 Arten zur Verwaltung persistenter Objekte
  - Persistenz durch Erreichbarkeit: alle von einem persistenten Objekt (Einstiegspunkt/Wurzel-Objekt) erreichbaren Objekte werden dauerhaft in der DB gespeichert
  - explizites Anlegen eines **Extent** zu einem Typ T: umfasst Menge von (persist.) Instanzen von T; für Subtyp T2 von T gilt, dass der Extent von T2 eine Teilmenge des Extents von T ist

Kollektionen

- Kollektionen
  - Gruppierungen homogener Objekte (gleicher Typ)
  - Kollektionstypen sind *generische/parametrisierte Typen* (statische Festlegung der Elementtypen)
  - jede Kollektion ist ein Objekt (eigene OID)
- Kollektionstypen
  - Set <t>: keine Duplikate (Duplikate möglich bei Bag, List, Array)
  - Bag <t>: Duplikate möglich
  - List <t>: Ordnung wird beim Einfügen festgelegt
  - Arrays <t>: 1-dimensional, feste (jedoch änderbare) Länge
  - Dictionary <t,v>: ungeordnete Folge von Schlüssel/Werte-Paaren. Keine Duplikate
- Cursorbasierter (navigierender) Zugriff auf Kollektionen über *Iteratoren*
  - Verwaltung der aktuellen Position
  - Operationen: *reset*, *next_position*, *get_element*
  - es können mehrere Iteratoren pro Kollektion definiert werden
  - *bidirektionale Iteratoren*: geordnete Kollektionen können vorwärts und rückwärts bearbeitet werden
  - *stabile Iteratoren*: Änderungen während der Iterierung bleiben ohne Auswirkungen



Kollektionsschnittstellen

```
interface Collection: Object {
unsigned long    cardinality();
boolean          is_empty();
void             insert_element (in any element);
void             remove_element (in any element);
boolean          contains_element (in any element);
Iterator         create_iterator (in boolean stability);
}
```

```
interface Set: Collection {
Set      union_with (in Set other);
Set      intersection_with (in Set other);
Set      difference_with (in Set other);
boolean  is_subset_of (in Set other);
boolean  is_superset_of (in Set other);
}
```

```
interface Bag: Collection {
Bag      union_with (in Bag other);
Bag      intersection_with (in Bag other);
Bag      difference_with (in Bag other);
}
```

```
interface Dictionary: Collection {
exception  KeyNotFound {any key};
any        lookup (in any key) raises (KeyNotFound);
•••}
```

```
interface Iterator: Object {
exception  NoMoreElements{};
boolean    is_stable();
boolean    at_end();
void       reset();
any        get_element () raises (NoMoreElements);
any        next_position () raises (NoMoreElements);
}
```

```
interface List: Collection {
void     replace_element_at (in unsigned long index,
         in any element);
void     remove_element_at (in unsigned long index);
void     retrieve_element_at (in unsigned long index);
void     insert_element_first (in any obj);
void     insert_element_last (in any obj); • • •
List     concat (in List other);
}
```

```
interface Array: Collection {
void     replace_element_at (in unsigned long index,
         in any element);
void     remove_element_at (in unsigned long index);
void     retrieve_element_at (in unsigned long index);
void     resize (in unsigned long new_size);
}
```


Attribute

- Objekteigenschaften: Attribute und Relationships
- Attribute
  - sind jeweils genau 1 Objekttyp zugeordnet
  - Attributwerte sind Literale oder Objekt-Identifier (Nullwert: nil)
  - Attribute selbst sind keine Objekte (keine OID)

```
class Person {

    attribute date Geburtsdatum;
    attribute enum Geschlecht {m, w};
    attribute Adresse Heimanschrift;
    atrribute Set<string> Hobbies;
    attribute Abt Abteilung;
}
```

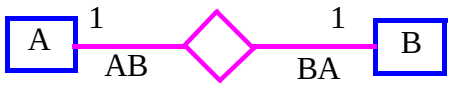

Relationships

- Relationship
  - Beziehung zwischen 2 Objekttypen (nur binäre Beziehungen)
  - 3 Arten: 1:1, 1:n, n:m
  - symmetrische Definition in beiden beteiligten Objekttypen
  - jede Zugriffsrichtung (traversal path) bekommt eigenen Namen
- symmetrische Beziehungen erlauben automatische Wartung der referentiellen Integrität
- Repräsentation von (einfachen) n:m-Beziehungen ohne künstliche Objekttypen
- Pfade auf Objektmengen können geordnet sein (z.B. über List)

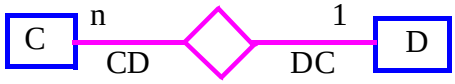
```
class Vorlesung { • • •
relationship set<Student> Hörer inverse Student::hört;
}
class Student { • • •
relationship set<Vorlesung> hört
                inverse Vorlesung::Hörer;
}
```

- Built-In-Operationen auf Traversierungspfaden:
  - *form/drop:* Einrichten/Löschen einer Mitgliedschaft eines einzelnen Objektes in 1:1- bzw. 1:n-Beziehung
  - Kollektionsoperationen für mehrwertige Beziehungen

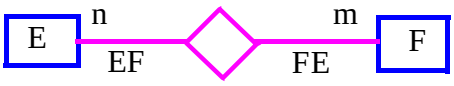
Relationships (2)



class A {
relationship **inverse** }
class B {
relationship **inverse** }



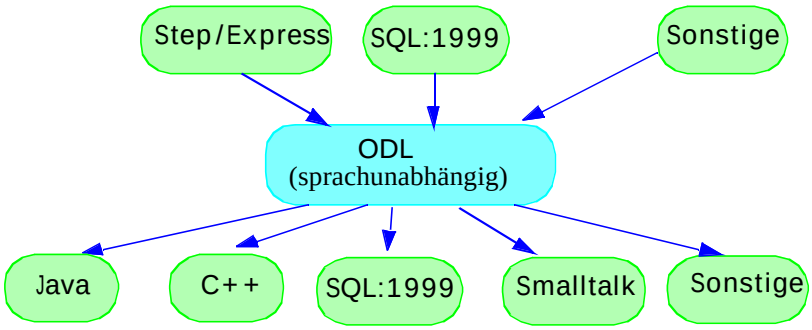
class C {
relationship **inverse** }
class D {
relationship **inverse** }



class E {
relationship **inverse** }
class F {
relationship **inverse** }

ODL (Object Definition Language)

- portable Spezifikation von Objekttypen
  - keine vollständige Programmiersprache
  - Erweiterung der OMG Interface Definition Language (IDL)
- ODL kann auf verschiedene konkrete Sprachen abgebildet werden => unterstützt Zusammenarbeit zwischen heterogenen Systemen



  - Abbildung auf C++, Smalltalk und Java festgelegt
- neben ODL existieren noch zum Datenaustausch
  - Object Interchange Format (OIF)
  - OIFML: XML-basierte Austauschsprache für ODMG-Objekte

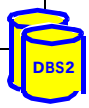
ODL (2)

- Typspezifikation durch Interface- oder Class-Deklaration

```
<interface_dcl>        ::=   <interface_header> {[ <interface_body> ]}
<interface_header>     ::=   interface <identifier> [<inheritance_spec>]
<inheritance_spec>     ::=   : <identifier> [, <inheritance_spec>]

<class_dcl>            ::=   <class_header> { <interface_body> }
<class_header>         ::=   class <identifier> [ extends <class-identifier>]
                                 [ <inheritance_spec> ] [ <type_property_list> ]

<type_property_list> ::=     ( [ <extent_spec> | <key_spec> ] )
<extent_spec>          ::=   extent <string>
<key_spec>             ::=   key[s] <key_list>
<key_list>             ::=   <key> | <key>, <key_list>
<key>                  ::=   <property_name> | ( <property_name_list> )
```

- Beispiel

```
class Professor extends Person
            ( extent Profs   keys PNR, (Name, Gebdat) )
            {
                <interface_body> };
```

ODL (3)

- Instanzen-Properties: Attribute und Relationships

```
<interface_body> ::=   <export> | <export> interface_body>
<export          ::=   <type_dcl> | <const_dcl> | <except_dcl> | <attr_dcl> | <rel_dcl> | <op_dcl>
<type_dcl>       ::=   typedef <type_spec_declarators> | <struct_type> | <union_type> | <enum_type>
<struct_type>::=       struct <identifier> { <member_list> }
<const_dcl>      ::=   const <const_type> <identifier> = <const_exp>
<except_dcl>::=        exception <identifier> { [ <member_list> ] }
<attr_dcl>       ::=   [readonly] attribute <domain_type> <identifier> [ <fixed_array_size> ]
<domain_type> ::= <simple_type_spec> | <struct_type> | <enum_type>
<rel_dcl>        ::=   relationship <target_of_path>  <identifier> inverse  <inverse_path>
<target_of_path>::= <identifier>  | <rel_collection_type> < <identifier> >
<inverse_path>::= <identifier> :: <identifier>
```

- Beispiel

```
class Professor: Person ( extent Profs;   keys PNR, (Name, Gebdat) )
{       struct Kind {string Kname, unsigned short Alter};
        attribute string Name;
        attribute unsigned long PNR [8];
        attribute date Gebdat;
        attribute set<Kind> Kinder;
        relationship ABT Abteilung inverse ABT::Abt_Angehörige
        relationship set<Student> Prüflinge inverse Student::Prüfer;
        ... }
```


ODL (4)

- Definition der Operationen wie in IDL

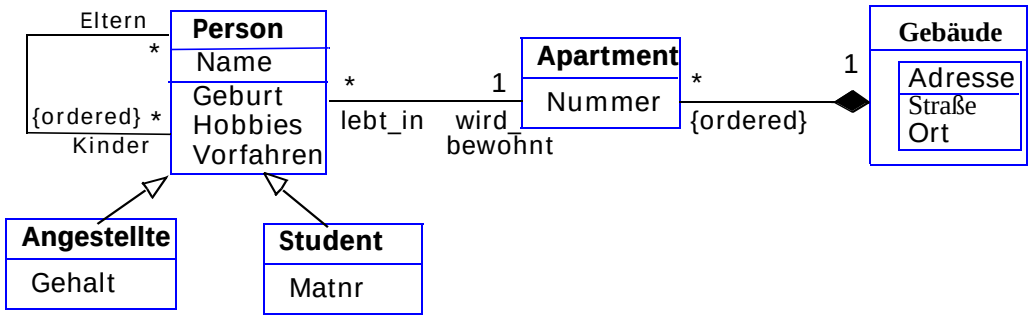
<op_dcl> ::= [<op_attribute>] <op_type_spec> <identifier>
([<param_dcl_list>]**)** [<raises_expr>] [<context_expr>]
<op_attribute>::= **oneway**
<op_type_spec>::=< simple_type_spec> | **void**
<param_dcl_list> ::= <param_dcl> | <param_dcl>, <param_dcl_list>
<param_dcl>::= <param_attribute> <simple_type_spec> <identifier>
<param_attribute>::= **in** | **out** | **inout**
<raises_expr> ::= **raises** (exception_list)

- Beispiele:

void Neuer_Pruefling (in Student s);

unsigned short Alter () **raises** (Geb_unbekannt);

ODL-Beispiel



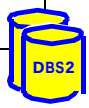

```
class Person (extent PERS) {
   attribute string Name
   relationship  Apartment lebt_in inverse Apartment::wird_bewohnt_von;
   relationship set<Person> Eltern inverse Person::Kinder;
   relationship  List<Person> Kinder inverse Person::Eltern;
   void Geburt ( in Person Kind);
   set<string> Hobbies();
   set< Person> Vorfahren ();};

class Gebäude{
   attribute struct Adresse {string Straße,string Ort};
   relationship  List<Apartment> Apartments inverse  Apartment::Haus;};

class Apartment{
   attribute short Nummer;
   relationship Gebäude Haus inverse Gebäude::Apartment
   relationship set <Person> wird_bewohnt_von inverse Person lebt_in;};

class Angestellte extends Person{
   attribute float Gehalt; };

class Student extends Person{
   attribute string Matnr; };
```

Spracheinbettung C++

- Realisierung der ODL in C++
  - nur ein zusätzliches Sprachkonstrukt für Beziehungen (neues Schlüsselwort **inverse**)
  - Klassenbibliothek mit Templates für Kollektionen etc.
  - Ref-Template d_Ref<T> für alle Klassen T, die persistent sein können

```
class Person {
   public:String Name;
   d_Ref<Apartment> lebt_in
            inverse wird_bewohnt_von;
   d_Set  < d_Ref<Person> > Eltern
            inverse Kinder;
   d_List < d_Ref<Person> > Kinder
            inverse Eltern;
   // Methods:
   Person(); // Konstruktor
   void Geburt ( d_Ref<Person> Kind);
   virtual d_Set<String> Hobbies();
   d_Set< d_Ref<Person> > Vorfahren();
   };
class Angestellte: Person{
   public: float Gehalt;
   };
class Student: Person{
   public: string Matnr; };
```

```
class Adresse{
     String Straße;
     String Ort;
   };
class Gebäude{
     Adresse adresse;
     d_List< < d_Ref<Apartment> > Apartments
            inverse Haus;
     // Method
     d_Ref<Apartment> preiswert ();
     };
class Apartment{
     int Nummer;
     d_Ref<Gebäude> Haus inverse Apartments;
     d_Set <d_Ref<Person>> wird_bewohnt_von
            inverse lebt_in;
     };
d_Set <d_Ref<Person>>  Pers;    // Personen-Extent
d_Set <d_Ref<Apartment>> Apartments;
```

OQL (Object Query Language)

- Deklarative Spezifikation von Ad-Hoc-Anfragen sowie eingebetteter Queries
- Merkmale von OQL
  - an SQL angelehnt (insbesondere seit V2), aber keine vollständige DML bzw. OML
  - keine Änderungsoperationen (Änderungen müssen durch typspezifische Operationen erfolgen)
  - Verarbeitung beliebiger Objekte (nicht nur von Tupeln und Tabellen/Sets)
  - Unterstützung von Pfadausdrücken sowie von Methodenaufrufen innerhalb Queries
  - OQL-Aufrufe aus Anw.programmen möglich (für Programmiersprachen mit ODMG-Binding)
- funktionale Query-Sprache
  - Query = Ausdruck
  - orthogonale Verwendbarkeit von Query-Ausdrücken
- Beispiel:

```
select x.Matnr
from Studenten x
where x.Name="Schmidt"
```

- Anfrageergebnis
  - ist entweder Kollektion von Objekten, Kollektion von Literalen, ein Objekt oder ein Literal
  - kann explizit strukturiert werden

```
select  struct (Prof: x.Name, sgS: (select y  from x.Prueflinge y
                                    where y.Note < 1,5))
from Profs x
```


OQL (2)

- direkte Traversierung entlang von Relationships anstelle von Joins
  - Zugriff über "." oder "→" - Notation: p.lebt-in.Haus.Adresse.Straße
- Zugriff bei mengenwertigen Beziehungen über *select*
- Pfad-Ausdrücke in From- und Where-Klausel möglich
- Weitere Aspekte
  - Mengenausdrücke (anwendbar auf Set- bzw. Bag-Objekte): intersect, union, except
  - Existenzquantifizierung: select x.Name from Profs x where exists y in x.Prueflinge: y.Note = 1,0
  - Allquantifizierung (Ergebnis ist true oder false): for all x in Studenten: x.Matnr > 0
  - Sortierbeispiel: sort x in Personen by x.Alter, x.Name
  - Aggregatfunktionen: count, sum, ...

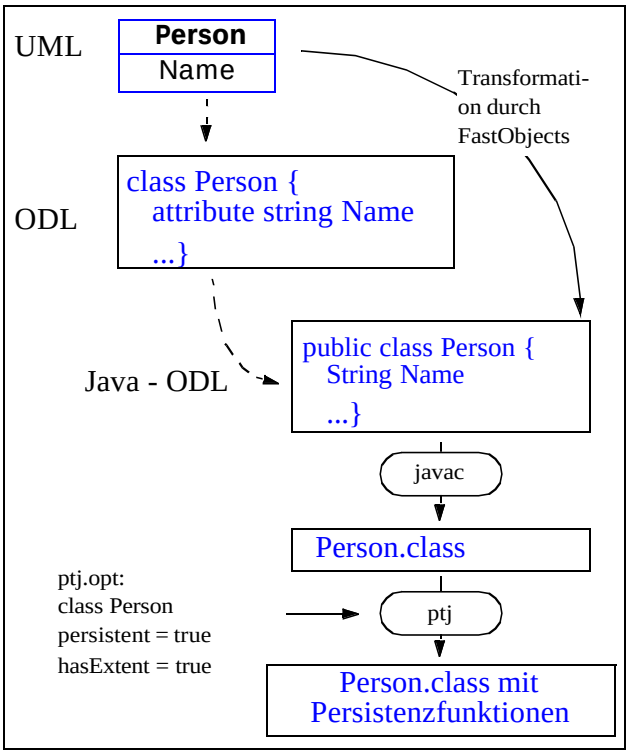
FastObjects*

- OODBS-Produkte der Firma POET (Hamburg, San Moteo)
  - früher: Poet Object Server Suite
  - FastObjects t7: Client/Server-Version
  - FastObjects e7: 1-Schichten-Architektur ohne Mehrbenutzerbetrieb (enge Kopplung DBS/Anwendung)
  - FastObjects j7 (bisheriger Name: Navajo): Minimalversion mit Ausrichtung auf mobile Systeme
  - Lauffähig auf Windows, Linux und Solaris (letzteres nur für t7)
- Schnittstellen für C++ und Java
- Geschachtelte Transaktionen
- Teilweise Unterstützung von OQL
- Java-Werkzeuge
  - Postprozessor: Erweitert Java-Klassen mit Persistenzfunktionen
  - Dienstprogramme, z.B. für Aktualisierung von Datenbanken nach Schemaänderungen
- Plug-Ins für "Third Party Products"
  - UML-Modellierung: Rational Rose
  - Java-Entwicklung: Forte (Sun), JBuilder (Borland)

* http://www.fastobjects.com

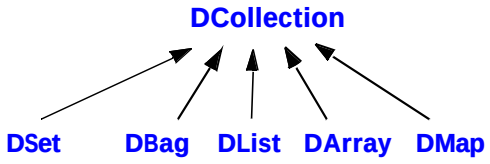
FastObjects (2): Schemadefinition

- Unterstützung der Abbildung UML → Java
- Abbildungspfad UML → ODL → Java wird von FastObjects nicht unterstützt
- Java-Klassenschemata werden mit ODL-orientierten Konstrukten ergänzt (z.B. zur Abbildung von Objektbeziehungen)
- Durch Java-Compiler erzeugte .class-Dateien werden unter Verwendung einer Optionsdatei durch den Postprozessor (ptj) persistenzfähig gemacht



FastObjects (3)

- Persistenzorientierte Erweiterungen gegenüber Java
  - "Persistenz durch Erreichbarkeit": Ausgewählte Objekte werden explizit als persistente "Einstiegs"-Objekte definiert; von dort aus Zugriff auf andere Objekte per Navigation
  - persistente Objekte in Extensionen (Kollektionen)
  - zusätzliche Templates für Kollektionen:



DMap: Repräsentiert Abbildungen, die mit Hash-Tabellen vergleichbar sind

- Verwendung von Kollektionsobjekten in Klassen
  - „Originäres" Java

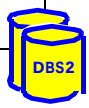
```
public class Person   {
        ...
        private Person[] kinder;
```

  - Java - ODL in FastObject

```
public class Person   {
        ...
        private DArray kinder;
```

FastObjects: OQL-Queries

- Keine vollständige Implementierung von OQL
- Unterstützt werden im Rahmen der „select-from-where“-Struktur
  - Extraktion einzelner Attribute oder ganzer Objekte
  - Pfadausdrücke
  - Aggregatsfunktion *count*
  - Existenzquantifizierung (exists ...)
  - Allquantifizierung (for all ...) (nur auf Extensionsebene, nicht in where-Klausel)
  - Sortierung
- Nicht unterstützt werden
  - Auslesen von Attributkombinationen (struct)
  - Gruppierung
  - Methodenaufrufe in Anfragen
  - Aggregatsfunktionen außer count (z.B. sum)



FastObjects: OQL (2)

- Anfragearten
  - interaktives OQL im FastObjects Developer (unter Windows)
  - OQL-Einbettung in Java (oder C++)
- OQL-Einbettung in Java

```
import com.poet.odmg;
import com.poet.odmg.util;
import org.odmg;

// Annahmen:
// - String-Parameter "name" ist bereits gesetzt
// - Klasse "Person" verfügt über überschriebene
//    Präsentationsroutine "toString()"
...
// Aufbau der Anfrage: Liefere alle Personen die Kind mit
// Namen "name" haben

String qstr = "select p.Name from p in PersonExtent " +
                "where exists k in p.kinder: "
                "k.Name = $1"

OQLQuery query = new OQLQuery(qstr);

// Übergabe des "name“-Wertes an OQLQuery-Objekt
query.bind(name);
```

```
Object res = null;
try {
        res = query.execute(); }
catch (POETRuntimeException pre) {
        System.out.println(pre); }

if (res != null) {
    if (res   instanceof  CollectionOfObject) { // falls Kollektion
        Iterator it = ((CollectionOfObject) res).iterator();
        int i = 0;
      while (it.hasNext())
            System.out.println(it.next().toString()); // Objektausgab
      }
    else // falls keine Kollektion
        System.out.println(res.toString());
  }
...
```


Zusammenfassung

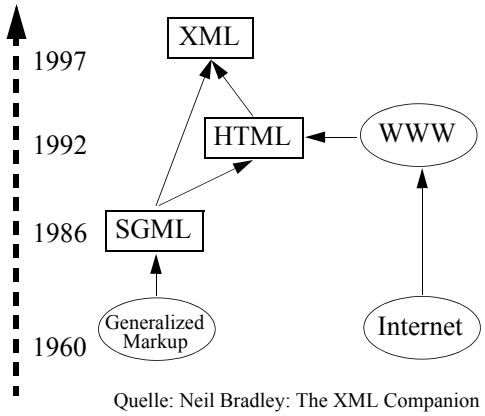
- gute Zusammenarbeit mit objektorientierten Programmiersprachen
- ODMG-Objektmodell
  - Erweiterung des OMG-Kernmodells
  - Objekte und Literale
  - Operationen und Properties: Attribute und Relationships
  - symmetrische binäre Beziehungen
  - Kollektionstypen mit Iteratoren und Strukturen
- standardisierte Schemabeschreibung über ODL
- Anfragesprache OQL
  - deskriptive Teilsprache, an SQL angelehnt
  - hohe Orthogonalität
  - Queries können Pfadausdrücke sowie Methodenaufrufe enthalten
  - Defizite: keine Änderungen, Integritätsbedingungen, Sichtkonzept, Autorisierung
  - keine Kompatibilität zu relationalen Datenbanken / SQL
- geringe Marktbedeutung objektorientierter DBS

7. XML-Datenbanken

- XML
- XML Schema
- XPath
- XQuery
- Speicherung von XML-Dokumenten
  - Klassifikation von XML-Dokumenten
  - Speicherungsverfahren
  - XML Mapping in ORDBS
- XML-Unterstützung in kommerziellen DBS
  - DB2 XML Extender
  - XIS
- Literatur
  - Klettke, M.; Meyer, H.: XML & Datenbanken. dpunkt.verlag, 2003
  - Kazakos, W., et al.: Datenbanken und XML. Springer, 2002
  - Rahm, E.; Vossen, G. (Hrsg.): Web & Datenbanken. dpunkt.verlag, 2003

XML- eXtensible Markup Language

- Metasprache: dient zur einheitlichen Definition von Datenformaten
  - Markup-Sprache: Dokumentabschnitte werden durch Marker ausgezeichnet
- hervorgegangen aus SGML und HTML
  - SGML: Standard Generalized Markup Language (1986)
    - Metasprache zur Beschreibung von Dokumentformaten (Markup-Sprachen), welche standardisiert bearbeitet werden können
    - Zielgruppe: Regierungsorganisationen, Firmen und Verlage mit großen Dokumentkollektionen
    - Problem: sehr hohe Komplexität
  - HTML: HyperText Markup Language (1994 V2.0)
    - Markup-Sprache mit festgelegter Menge an Auszeichnungselementen für Struktur und Darstelung von Dokumenten
    - ist eine SGML-Anwendung, d.h. HTML-Syntax ist in SGML beschrieben
    - Problem: nicht erweiterbar
  - XML (1998 als Empfehlung vom W3C - World Wide Web Consortium - veröffentlicht)
    - Teilmenge von SGML unter Verwendung der HTML-Konventionen
    - Ziele: einfach anwendbar, SGML-Kompatibilität, klare und lesbare Dokumentformate, u.a.



Quelle: Neil Bradley: The XML Companion

XML - Beispiel

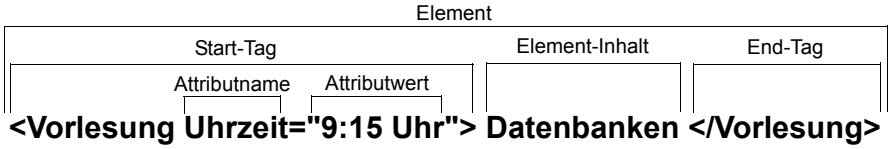
- Elemente bilden logische Struktur
  - XML-Dokumente haben Baumstruktur, deren Knoten die Elemente sind (Schachtelung von Elementen; keine Überlappung der Elemente)
  - Schema ist implizit in den XML-Daten enthalten, kann jedoch auch explizit durch DTD (document type definition) vorgegeben werden
    - wohlgeformt: XML-Daten sind syntaktisch korrekt
    - gültig: XML-Daten sind wohlgeformt und entsprechen einer DTD

```
<?XML version="1.0"?>
<!DOCTYPE Vorlesungsverzeichnis SYSTEM
  "http://dbs.uni-leipzig.de/dtd/VLVerzeichnis.dtd">
<VLVerzeichnis>
    <Vorlesung Uhrzeit="9:15 Uhr">
        <Thema>DBS2</Thema>
        <Dozent>
            <Name>Prof. Rahm</Name>
            <Einrichtung>Uni Leipzig</Einrichtung>
        </Dozent>
    </Vorlesung>
</VLVerzeichnis>
```

VLVerzeichnis.dtd:

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT VLVerzeichnis (Vorlesung)* >
<!ELEMENT Vorlesung (Thema, Dozent) >
<!ATTLIST Vorlesung
   Uhrzeit   CDATA    #REQUIRED >
<!ELEMENT Thema (#PCDATA) >
<!ELEMENT Dozent (Name, Einrichtung?)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Einrichtung (#PCDATA)>
```


XML: Elemente und Attribute



- Groß-/Kleinschreibung ist relevant (gilt für alle Bezeichner in XML)

- Elemente
  - Start- und End-Tag müssen vorhanden sein Ausnahme: leeres Element (Bsp. <Leer />)
  - feste Reihenfolge von Geschwisterelementen
  - Element-Inhalt besteht entweder
    - nur aus weiteren Elementen (element content)
    - aus Zeichendaten optional vermischt mit Elementen (mixed content)

- Attribute
  - Attributwerte können nicht strukturiert werden
  - Attributreihenfolge beliebig

- weitere Bestandteile von XML (hier nicht behandelt):
  - Entities (Ersetzungsmechanismus zur physischen Strukturierung, Zeichen(ketten)ersetzung)
  - Processing Instructions (Anweisungen an das verarbeitende Programm - XML Processor)
  - Kommentare (eingeschlossen durch '<!--' und '-->')

DTD - Document Type Definition

- beschreibt eine Dokumentstruktur und legt damit einen Dokumenttyp fest
- im Dokument erfolgt Verweis auf DTD in einer Dokumenttyp-Deklaration
  - interne DTD `<!DOCTYPE Vorlesungsverzeichnis [ <!ELEMENT VLVerzeichnis (Vorlesung)* > ... ]>`
  - externe DTD `<!DOCTYPE Vorlesungsverzeichnis SYSTEM "http://dbs.uni-leipzig.de/dtd/VLVerzeichnis.dtd" >`
- Definition von Elementen in einer DTD
  - Sequenz: (A,B) - vorgegebene Reihenfolge `<!ELEMENT Vorlesung (Thema, Dozent) >`
  - Alternative: (A|B) - entweder A oder B (XOR) `<!ELEMENT Adresse (PLZ, Ort, (Str, Nr) | Postfach) >`
  - Wiederholung:
    - A? - 0..1 Mal `<!ELEMENT Dozent (Name, Einrichtung?)>`
    - A+ - 1..n Mal `<!ELEMENT Name (Vorname+, Nachname)>`
    - A* - 0..n Mal `<!ELEMENT VLVerzeichnis (Vorlesung)* >`
  - Mixed Content: (#PCDATA | A | B)* - A, B und Text treten in beliebiger Reihenfolge und Anzahl auf `<!ELEMENT Text (#PCDATA | Link)* >`
  - Leeres Element (kein Element-Inhalt) `<!ELEMENT br EMPTY >`

DTD (2)

- Definition von Attributen in einer DTD

  `<!ATTLIST Elementname (Attributname Typ Auftreten)*>`

  - Attribute gehören zu einem Element
  - haben einen Namen
  - besitzen einen Typ
    - CDATA, ID, IDREF/IDREFS, ENTITY/ENTITYS, NMTOKEN/NMTOKENS
    - Aufzählung möglicher Werte (wert1|wert2|...)
  - besitzen eine Angabe bzgl. des Auftretens des Attributs im Dokument
    - #REQUIRED - das Attribut muss angegeben werden
    - #IMPLIED - es gibt keinen Defaultwert
    - (#FIXED) DEFAULTWERT - der Defaultwert wird angenommen, wenn das Attribut nicht angegeben wird; ist #FIXED spezifiziert, kann das Attribut nur den Defaultwert als Wert besitzen
  - Beispiele:
    - `<!ATTLIST Entfernung Einheit CDATA #FIXED "km">`
    - `<!ATTLIST Karosse Farbe ("rot" | "gelb" | "blau") "blau">`
    - `<!ATTLIST Artikel id ID #REQUIRED>`
    - `<!ATTLIST Rechnungsposten Artikel IDREF #REQUIRED>`
  - ID- und IDREF-Attribute ermöglichen Querverweise innerhalb eines Dokumentes (Graphstruktur)

Einsatz von Elementen und Attributen

- Datenmodellierung oftmals sowohl durch ein Element als auch durch ein Attribut möglich
- Einsatzkriterien unter Verwendung einer DTD (nach Klettke, Meyer: XML & Datenbanken)

	Element	Attribut
Identifikation	–	ID/IDREF/IDREFS
Quantoren	1/?/*/+	REQUIRED/IMPLIED
Alternativen	+	–
Defaultwerte	–	+
Aufzählungstypen	–	+
Inhalte	komplex	nur atomar
Ordnungserhaltung	ja	nein

- bei Einsatz von XML Schema anstelle einer DTD können Elemente alle Eigenschaften von Attributen erhalten; trotzdem sollten Attribute bevorzugt werden wenn:
  - Aufzählungstypen mit atomaren Werten und Defaultwert zu modellieren sind
  - es sich um 'Zusatzinformation' handelt (Bsp. Währung, Einheit)
  - das Dokument effizient verarbeitet (geparsed) werden soll

XML Schemabeschreibungssprachen

- Schemainformationen über Struktur der XML-Daten sowie der enthaltenen Datentypen wichtig für effiziente Speicherung in vielen XML-Datenbanken
  - Anlegen der benötigten Datenstrukturen
  - Typvalidierung und typspezifische Operationen
- Defizite der DTD:
  - nur Strukturbeschreibung (kaum Typisierung von Elementinhalt und Attributwerten möglich)
  - zusätzliche Syntax (kein XML-Format) -> extra Parser notwendig
  - keine Unterstützung von Namespaces (Namensräume für Element- und Attributnamen)
- Entwicklung alternativer Schemabeschreibungssprachen
  - RELAX NG (http://www.oasis-open.org/committees/relax-ng/)
    - Vereinigung der Sprachen RELAX und TREX unter Leitung des OASIS Standardisierungsgremiums
    - einfach, grammatikbasiert, verwendet XML-Syntax, unterstützt Namensräume, behandelt Attribute und Elemente nahezu gleichwertig, kann externe Datentypbeschreibungssprache verwenden (z.B. XML Schema)
  - Schematron (http://www.ascc.net/xml/resource/schematron/schematron.html)
    - entwickelt von Rick Jelliffe, Academia Sinica Computing Centre, Taiwan; ISO Standardisierung läuft
    - regelbasiert (nicht grammatikbasiert wie die anderen Ansätze), testet Konformität mittels Tree Patterns (definiert über XPath)

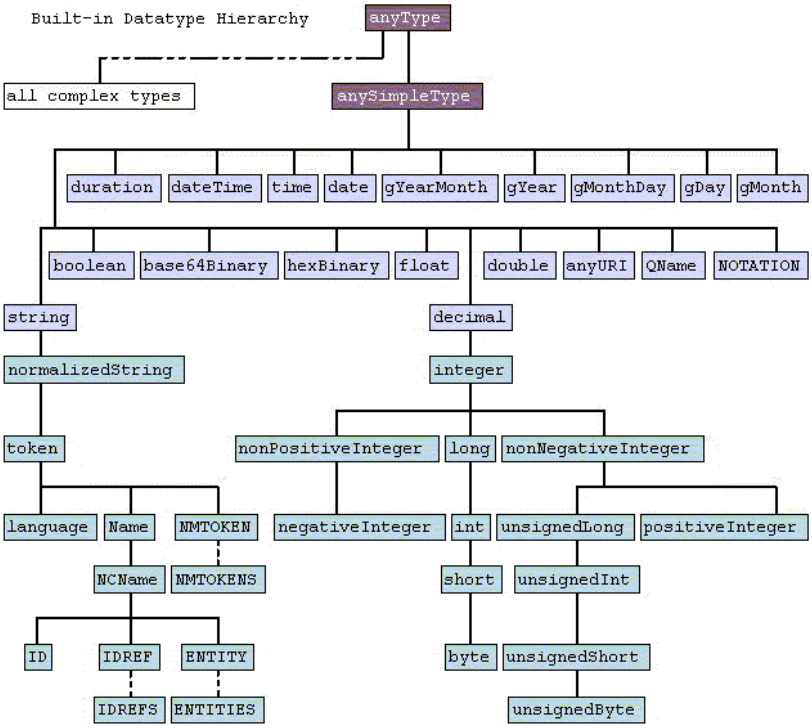
XML Schema*

- XML Schemabeschreibungssprache, entwickelt durch das W3C
  - Teil 1: Strukturbeschreibung
    - ersetzt Strukturbeschreibung der DTD
    - Unterstützung von Namespaces
  - Teil 2: Datentypen
    - Definition Basistypsystem
    - Erweiterbares Typsystem
- Eigenschaften
  - verwendet XML-Syntax
  - vielfältige vordefinierte Datentypen
  - Ableitung neuer Datentypen durch Restriktionen oder Erweiterungen
  - vielfältige Beschreibungsmöglichkeiten
  - erweiterbar
  - erlaubt Definition von Integritätsbedingungen (unique, key)
  - keine Unterstützung von Entitäten (diese können jedoch teilweise nachgebildet werden)

* http://www.w3c.org/XML/Schema

XML Schema: Typen

- Vielzahl vordefinierter einfacher Standardtypen



XML Schema: Typen (2)

- benutzerdefinierte einfache Datentypen ableitbar durch Angabe von einschränkenden Eigenschaften (facets)

```
<xs:simpleType name="LoginName">
 <xs:restriction base="xs:string">
  <xs:length value="8" />
 </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="OktalDigit">
 <xs:restriction base="xs:integer">
  <xs:minInclusive value="0" />
  <xs:maxInclusive value="7" />
 </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="BibId">
 <xs:restriction base="xs:string">
  <xs:pattern value="[A-Z][1-9][0-9]*" />
 </xs:restriction>
</xs:simpleType>
```


XML Schema: Typen (3)

- komplexe Datentypen dienen der Beschreibung von Elementinhalten

```
<xs:complexType name="DozentTyp">
 <xs:complexContent>
  <xs:restriction base="xs:anyType">
   <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="Einrichtung" type="xs:string"
                minOccurs="0"/>
   </xs:sequence>
  </xs:restriction>
 </xs:complexContent>
</xs:complexType>
```


XML Schema: Deklarationen

- Elementdeklaration
  - es gibt verschiedene Möglichkeiten der Elementdeklaration
    - direkte Definition des Elementinhalts

```
<xs:element name="Einrichtung" type="xs:string" minOccurs="0" />

<xs:element name="Dozent">
  <xs:complexType> ... </xs:complexType>
</xs:element>
```

    - Verweis auf komplexen Datentyp

```
<xs:element name="Dozent" type="DozentTyp"/>
```

    - Verweis auf Elementdeklaration

```
<xs:element ref="Dozent"/>
```

- Attributdeklaration

```
<xs:attribute name="Einheit" type="xs:string" fixed="km"/>
```

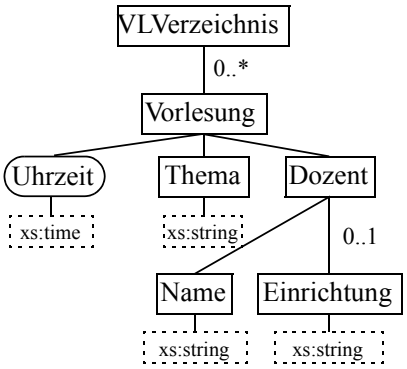
```
<xs:attribute name="Farbe" default="blau">
  <xs:simpleType>
    <xs:restriction base="xs:NMTOKEN">
      <xs:enumeration value="rot" />
      <xs:enumeration value="gelb" />
      <xs:enumeration value="blau" />
    </xs:restriction>
  </simpleType>
</xs:attribute>
```

```
<xs:attribute name="id" type="xs:ID" use="required"/>
```

```
<xs:attribute name="Artikel" type="xs:IDREF"
                use:="required"/>
```

XML Schema: Beispiel

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="VLVerzeichnis">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Vorlesung" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Thema" type="xs:string"/>
              <xs:element name="Dozent" type="xs:string">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Name" type="xs:string"/>
                    <xs:element name="Einrichtung" type="xs:string" minOccurs="0"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="Uhrzeit" type="xs:time" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```



XML Schema: Integritätsbedingungen

- pro Integritätsbedingung wird eine Elementmenge, auf die die Bedingung anzuwenden ist, per XPath-Ausdruck selektiert und in einem field-Ausdruck der Elementwert bzw. das Attribut, bzgl. dessen getestet werden soll, angegeben
- im Gegensatz zu ID-Attributen müssen die Werte nur innerhalb der selektierten Mengen eindeutig sein
  - Eindeutigkeitsbedingung: unique

```
<xs:unique name="UniqueBed">
  <xs:selector xpath="/VLVerzeichnis/Vorlesung" />
  <xs:field xpath="Thema/text()" />
</xs:unique>
```

  - Schlüsselbedingung: key/keyref

```
<xs:key name="ArtikelKey">
  <xs:selector xpath="/Produkte/Artikel" />
  <xs:field xpath="@artikelID" />
</xs:key>

<xs:keyref name="ArtikelKeyRef" refer="ArtikelKey">
  <xs:selector xpath="/Verzeichnis/Artikel" />
  <xs:field xpath="@artikelID" />
</xs:keyref>
```


XML Schema: Annotationen

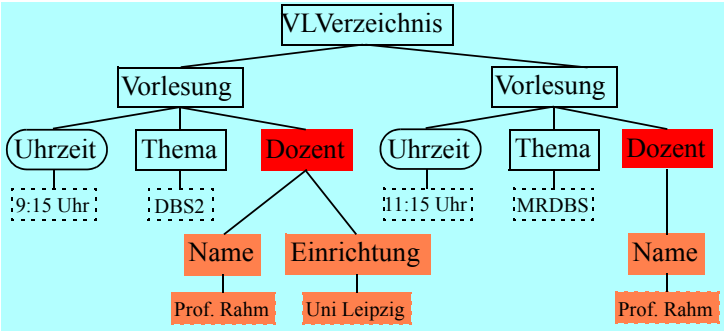
- XML Schemas können zur Information von Anwendern und Anwendungen am Anfang der meisten Schemakonstrukte annotiert werden
- z.B. verwendet Tamino Annotationen für Indexierungsinformationen

```
<xs:element name="Debug">
  <xs:annotation>
    <xs:documentation>
      Dieses Element nur in der Entwicklungsphase verwenden.
    </xs:documetation>
    <xs:appinfo>
      <!-- Anweisung an das verarbeitende Programm-->
    </xs:appinfo>
  </xs:annotation>
  <xs:complexType>...</xs:complexType>
</xs:element>
```


XPath 1.0*

- Sprache zur Selektion von XML-Teildokumenten (Dokumentfragmente, Elemente, Attribute, Kommentare, Text, ...)
- ist Bestandteil von XQuery, XSLT und XPointer
- Selektion erfolgt durch schrittweise Navigation im Dokumentbaum ausgehend von Kontextknoten (Pfad im Dokument, ähnlich Navigation im Dateisystem)

  Bsp.: */VLVerzeichnis/Vorlesung/Dozent*

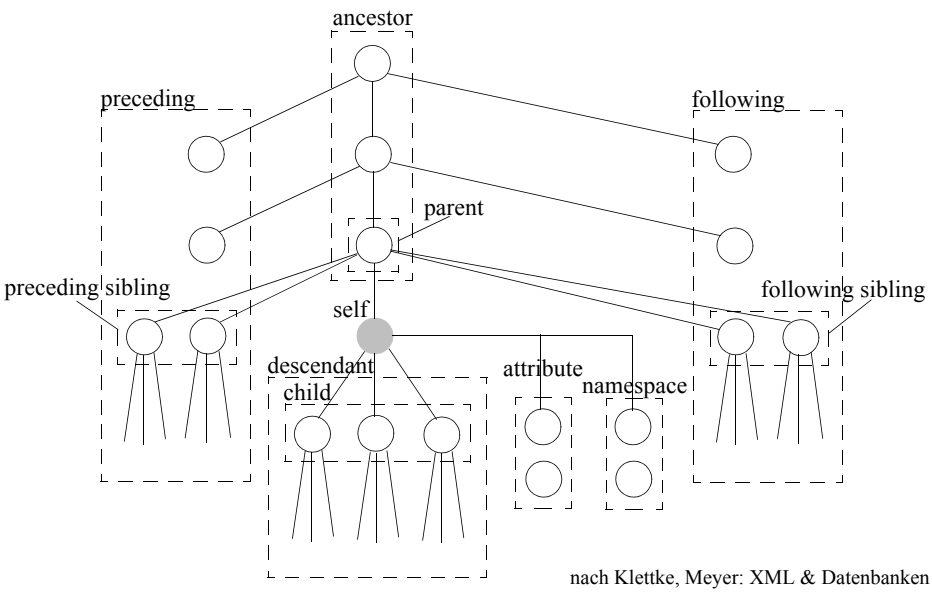


- Abarbeitung von links nach rechts
- jeder Schritt liefert Knotenmenge oder Werte
- absolute und relative Pfadausdrücke (absolut: /schritt1/schritt2; relativ: schritt1/schritt2)

* http://www.w3c.org/TR/xpath

XPath: Schritte

- Syntax eines Schrittes: **Achse::Knotentest[Prädikat]**
  - Achse: Richtung vom aktuellen Kontextknoten aus, in der die Knoten selektiert werden sollen (13 Achsen: child, descendant, parent, ancestor, following, preceding, attribute, ...)



nach Klettke, Meyer: XML & Datenbanken

XPath: Schritte (2)

- Knotentest: selektiert Knoten aus der durch Achse vorgegebenen Menge (Selektion eines Knotens, wenn Test „wahr“ liefert)

Knotentest	selektierte Menge (aus Knotenmenge der Achse)
Name	Elemente bzw. Attribute bzw. Namensraum mit diesem Namen
node()	alle Knoten
*	alle Elementknoten
text()	alle Textknoten
comment()	alle Kommentarknoten
processing-instruction()	alle Processing instructions

- Prädikat: Filterausdruck, der Knoten aus Menge nach dem Knotentest selektiert (Selektion eines Knotens, wenn Prädikat „wahr“ liefert)
  - kann XPath-Ausdrücke enthalten (testet Existenz bestimmter Elemente/Attribute/Attributwerte)
    Bsp.: */child::VLVerzeichnis/child::Vorlesung/child::Dozent[child::Einrichtung]/child::Name*
    alternativ: */VLVerzeichnis/Vorlesung/Dozent[Einrichtung]/Name*
  - bei Angabe von Zahlen werden Knoten der entsprechenden Kontextpositionen selektiert
    Bsp.: */child::Produktliste/child::Produkt[position()=1]*
    alternativ: */Produktliste/Produkt[1]*
  - kann logische Operatoren (and, or), Vergleichsoperatoren (’<’, ’<=’, ’!=’, ...), Vereinigung von Knotenmengen (’|’) enthalten

XPath: Abgekürzte Syntax / Funktionen

- XPath-Ausdrücke können durch abgekürzte Syntax vereinfacht werden

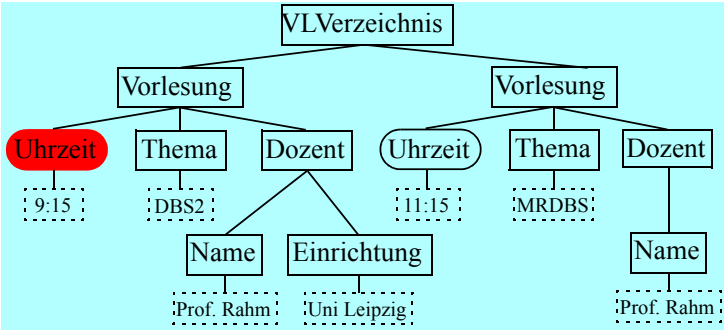
Langform	Abkürzung	Bemerkung
child::Knotentest	Knotentest	ohne Achsenangabe wird child-Achse verwendet
self::node()	.	aktueller Knoten
parent::node()	..	Elternknoten
descendant-or-self::node()	//	alle Nachkommen des aktuellen Knotens
attribute::Name	@Name	
[position()=X]	[X]	Prädikat zur Selektion von Knoten an Knotenposition X

- folgende Funktionen sind in XPath verfügbar (Auswahl):
  - für Knotenmenge: last(), position(), count(Knotenmenge), id(Name)
  - für Strings: concat(String1, String2, ...), starts-with(String, Pattern), contains(String, Pattern), substring(String, Start[, Länge]), string-length()
  - für Zahlen: sum(Knotenmenge), round(Zahl)
  - Boolesche Funktionen: not(Boolean), true(), false()

XPath: Beispiele

- *//Thema[text() ="DBS2"]/../@Uhrzeit*

  Finde die Uhrzeit zur Vorlesung "DBS2".



- Liste das Thema aller Vorlesungen, die "9:15" beginnen und deren Dozent zur "Uni Leipzig" gehört

- Liste alle Vorlesungen, deren Dozent "Prof. Rahm" oder "Dr. Sosna" ist

XPath: Beispiele (2)

- Wieviele Vorlesungen sind im Verzeichnis enthalten?

- Liste alle Einrichtungen, die den Text "Leipzig" enthalten

- Welche Mitarbeiter sind nach 1975 geboren?

```
<!ELEMENT Mitarbeiterverzeichnis (Mitarbeiter* )>
<!ELEMENT Mitarbeiter (Name, Lebenslauf, TelNr?)>
<!ATTLIST Mitarbeiter GeburtsJahr CDATA #REQUIRED >
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Lebenslauf (#PCDATA)>
<!ELEMENT TelNr (#PCDATA)>
```

- Liste Name und Telefonnummer aller Mitarbeiter

XPath 2.0*

- XPath 2.0 wird vom W3C im Rahmen der XQuery-Entwicklung standardisiert
- Änderungen zu XPath 1.0:
  - Verwendet Typsystem von XML Schema (XPath 1.0 nur Knotenmenge, boolesche, numerische und Zeichenkettenwerte)
  - basiert auf Sequenzen (XPath 1.0 basiert auf Mengen); Unterschied bei Duplikaten
  - unterstützt Referenzen
  - kann mit Dokumentkollektionen arbeiten
  - kennt Variable
  - Funktionen bzw. Variable können Pfadausdrücken vorangestellt werden
  - Unterscheidung Wertegleichheit und Knotenidentität
  - wesentlich erweiterte Funktionsbibliothek
  - Bereichsausdrücke in Prädikaten

* http://www.w3.org/TR/xpath20/

XQuery*

- neue Anfragesprache für XML notwendig
  - bisherige Anfragesprachen wie SQL oder OQL sind für XML unzureichend
    - kaum Unterstützung für hierarchische und sequenzbasierte Pfadnavigation
    - keine Unterstützung von Wildcards in Pfaden
    - unzureichende Anfragemöglichkeit zu Metadaten
    - keine Unterstützung zur Neustrukturierung der Ergebnismenge
  - XPath ist als Anfragesprache unzureichend
    - keine Joins, keine Restrukturierungsmöglichkeiten, keine Quantifizierer
    - geringe Funktionsbibliothek (es fehlen z.B. Aggregationsfunktionen, Sortierfunktionen)
- XQuery: W3C-Standardisierungsprozeß für einheitliche XML-Anfragesprache
  - abgeleitet von vorangegangenen proprietären XML-Anfragesprachen (Quilt, XPath, XQL, XML-QL, ...) sowie SQL und OQL
  - funktionale Anfragesprache
  - komplexe Pfadausdrücke (basierend auf XPath 2.0), Funktionen, konditionale und quantifizierte Ausdrücke, Ausdrücke zum Testen/Modifizieren von Datentypen, Elementkonstruktoren
  - FLWOR-Syntax (For ... Let ... Where ... Order By ... Return)

* http://www.w3c.org/XML/Query

XQuery: FLWOR

- generelle Vorgehensweise
  - Dokumentzugriff mit document(url)
  - Knotennavigation mit XPath 2.0
  - Variablenbindung im Kontext existierender Bindungen
  - Operationen auf den Knotenmengen
  - Erzeugung neuer Knoten
- FLWOR-Ausdrücke

```
FLWOR-expr      ::= (FOR-expr | LET-expr)+
                    WHERE-expr? ORDERBY-expr? RETURN-expr
FOR-expr        ::= for $var in expr (, $var in expr)*
LET-expr        ::= let $var := expr (, $var := expr)*
WHERE-expr      ::= where expr
ORDERBY-expr    ::= (order by | stable order by) OrderSpecList
  OrderSpecList ::= OrderSpec (, OrderSpec)*
  OrderSpec     ::= expr OrderModifier
  OrderModifier ::= (ascending | descending)? (empty greatest | empty least)? (collation StringLiteral)?
RETURN-expr     ::= return expr
```


XQuery: FLWOR (2)

- LET-Ausdruck
  - bindet Menge von Werten an Variable
  - Menge wird geschlossen an Variable gebunden

```
let $d := //Dozent/Name
return $d
```

- FOR-Ausdruck
  - für jedes Element der Ergebnismenge erfolgt Bindung an Variable (Iteration über die Menge)
  - Beispiel liefert gleiches Ergebnis wie bei LET, jedoch wird es anders abgearbeitet:
    - $d wird jeweils an Elemente der Sequenz von Dozentennamen gebunden (für jeden Namen genau einmal)
    - RETURN wird für jede Bindung einmal ausgeführt
    - Zwischenergebnisse werden zu Gesamtergebnis zusammengefaßt

```
for $d := //Dozent/Name
return $d
```


XQuery: Beispiele

- folgendes Schema liegt den XQuery-Beispielen zugrunde (aus XQuery Use cases)

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```


XQuery: Beispiele (2)

- Liste alle Buchtitel, in denen "Rahm" Erstautor ist (Dokument sei "bibl.xml")

- Liste alle Titel von Büchern, die bei Morgan Kaufmann nach 1998 publiziert wurden, in alphabetischer Reihenfolge

- Gebe alle Herausgeber und den Durchschnittspreis ihrer Bücher aus.

```
for $p in distinct-values(document("bib.xml")//publisher)
let $a := avg(document("bib.xml")//book[publisher = $p]/price)
return <publisher>
            <name> {$p/text()} </name>
            <avgprice> {$a} </avgprice>
  </publisher>
```

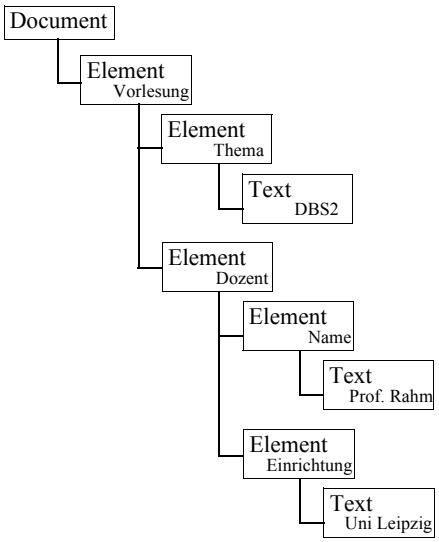

XQuery: Beispiele (3)

- Liste für jeden Autor seinen Namen und die Titel all seiner Bücher, gruppiert in einem Ergebnis-Element

```
<Liste>
  { for $a in distinct-values(document("http://dbs.uni-leipzig.de/bib.xml")//author)
    return <Ergebnis>
              { $a }
               { for $b in document("http://dbs.uni-leipzig.de/bib.xml")/bib/book
                where some $ba in $b/author satisfies deep-equal($ba,$a)
              return $b/title
              }
               </Ergebnis>
             }
</Liste>
```


XML-Prozessoren

- verarbeiten XML-Dokumente und stellen deren Inhalt einer Anwendung zur Verfügung
  - Auflösen von Entitäten, Validierung (bzgl. DTD oder XML Schema)
- standardisierte Schnittstellen:
  - SAX (Simple API for XML): ereignisorientierte, sequentielle Abarbeitung; zustandslos
  - DOM (Document Object Model): Empfehlung des W3C
- DOM (Document Object Model)
  - beschreibt API zum Zugriff auf XML-Dokumente und zur Datenmanipulation
  - zugrundeliegendes Datenmodell: Baum von Knoten (Elemente, Attribute, Textinhalt usw. sind spezielle Knoten, abgeleitet von Klasse Node)
  - Klasse Node enthält Methoden zur Identifikation des Knotentypes / zur Navigation durch die Dokumentstruktur und zur Manipulation der Dokumentstruktur
  - kontextabhängige Zugriffe
  - DOM-Struktur meist im Hauptspeicher, daher problematisch für große XML-Dokumente



Klassifikation von XML-Dokumenten

- breites Anwendungsgebiet für XML
  - große Varianz in Dokumenteigenschaften (strukturiert, semi-strukturiert, Textdokumente)
  - Auswirkung auf Verarbeitung der Daten
- Klassifikation von XML-Dokumenten (semi-strukturiert)

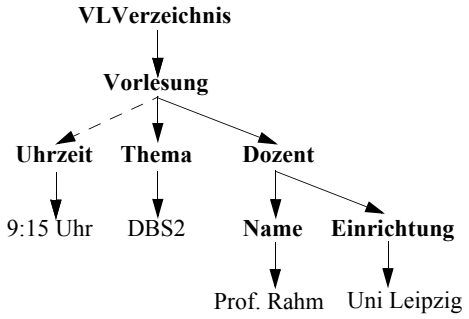
	datenorientiert	dokumentorientiert	gemischt strukturiert
Struktur	feingranular strukturiert	grobgranular und unstrukturiert	enthält sowohl datenorientierte als auch dokumentorientierte Eigenschaften
Schema	explizit vorgegeben	meist kein explizites Schema	
Typ-Information	getypt	ungetypt	
Elementordnung	nicht signifikant	signifikant	
Elementinhalt	element content	mixed content	
Operationen	Zugriff/Änderung einzelner Dokumentbereiche	Zugriff auf komplette Dokumente (originalgetreue Wiederherstellung notwendig)	
Beispiele	Bestellungen, Rechnungen, XML Darstellung relationaler Daten	wissenschaftliche Artikel, Geschäftsberichte	Online-Buchladen (datenorientiert: Titel, Verlag, usw; dokumentorientiert: Inhaltsangabe, Rezensionen), Produktkataloge

Speicherungsverfahren

	Speicherung als Ganzes	Dekomposition		Hybride Speicherung
		generisch	schemabasiert	
Speicherung	komplett (nicht zerlegt) in Datei oder DBS (Attributtyp: CLOB, BLOB, XMLType)	zerlegt bzgl. eines generischen Modells (z.B. Graphmodell, DOM)	anwendungsbezogene strukturierte Speicherung in DB (automatisches/manuelles Mapping)	meist anwendungsbezogene Kombination verschiedener Speicherungsmethoden
Zugriff	– Indizes – Parsen des Dokumentes	– Indizes – Pfadnavigation	auf spezifische DB-Struktur	
Vorteile	– originalgetreue Dokumentrekonstruktion – sehr schneller Zugriff auf komplettes Dokument – sehr schnelles Einbringen der Daten	– nur ein Parsingdurchlauf beim Laden – einfache Änderung von Dokumentteilen – schemaunabhängig	– performanter Zugriff/Änderung einzelner Daten	– Anpassung auf unterschiedliche Eigenschaften innerhalb eines Dokumentes
Nachteile	– hoher Zeitaufwand zum Parsen – hoher Änderungsaufwand	– hoher Aufwand zur Rekonstruktion von Dokumentfragmenten – oft keine originalgetreue Rekonstruktion	– pro Schema muss Mapping erstellt werden	– höherer Verwaltungsaufwand
geeignet für XML Dok.	dokumentenorientierte		datenorientierte	
	gemischt strukturierte			

Speicherung in ORDBS

- Speicherung als Ganzes
  - in BLOB/CLOB bzw. UDT
  - Anlegen von Indizes für ausgewähte Elemente/Attribute
  - Zugriff über UDFs
- generische Dekomposition
  - Abbildung Hierarchie (XML-Instanz) bzw. Graph (XML Schema) auf Relationen mittels Mapping
  - modellbasiertes Mapping: Überführung in generisches Datenmodell, welches auf ein Relationenschema abgebildet wird



Elemente

docID	elementname	ID	vorgänger	pos	wert
1	VLVerzeichnis	1	null	1	null
1	Vorlesung	2	1	1	null
1	Thema	3	2	1	DBS2
1	Dozent	4	2	2	null
1	Name	5	4	1	Prof. Rahm
1	Einrichtung	6	4	2	Uni Leipzig

Attribute

docID	attributname	elementID	wert
1	Uhrzeit	2	9:15 Uhr

Speicherung in ORDBS (2)

- schemabasierte Dekomposition
  - Relationenschema wird in Abhängigkeit vom XML-Schema erzeugt
  - automatisches Mapping
    - nach Bourret* (komplexe Elemente erhalten eigene Relation)

VLVerzeichnis

VLVerzeichnisID
1

Vorlesung

VorlesungID	Vorlesung-Pos	VLVerzeichnisID	Uhrzeit	Uhrzeit-Pos	Thema	Thema-Pos
1	1	1	9:15 Uhr	1	DBS2	2

Dozent

DozentID	DozentPos	VorlesungID	Name	NamePos	Einrichtung	EinrichtungPos
1	3	1	Prof. Rahm	1	Uni Leipzig	2

    - mit ROW-Typ, ohne Elementposition

VLVerzeichnis

VLVerzeichnisID
1

Vorlesung

VorlesungID	Uhrzeit	Thema	Dozent	
			Name	Einrichtung
1	9:15 Uhr	DBS2	Prof. Rahm	Uni Leipzig

* http://www.rpbourret.com/xmldbms/

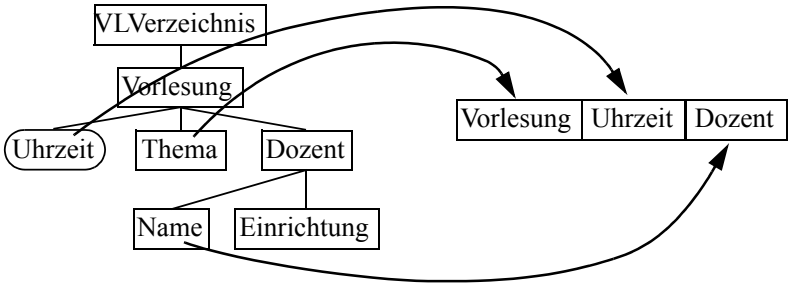
Speicherung in ORDBS (3)

- manuelles Mapping
  - Mapping zwischen XML-Dokument und Datenbank wird manuell festgelegt
  - Mapping muss nicht vollständig sein

XML-Daten

```
<Vorlesung Uhrzeit="9:15 Uhr">
  <Thema>DBS2</Thema>
  ...
</Vorlesung>
```

Mapping-Vorschrift



Datenbank

VLVerzeichnis

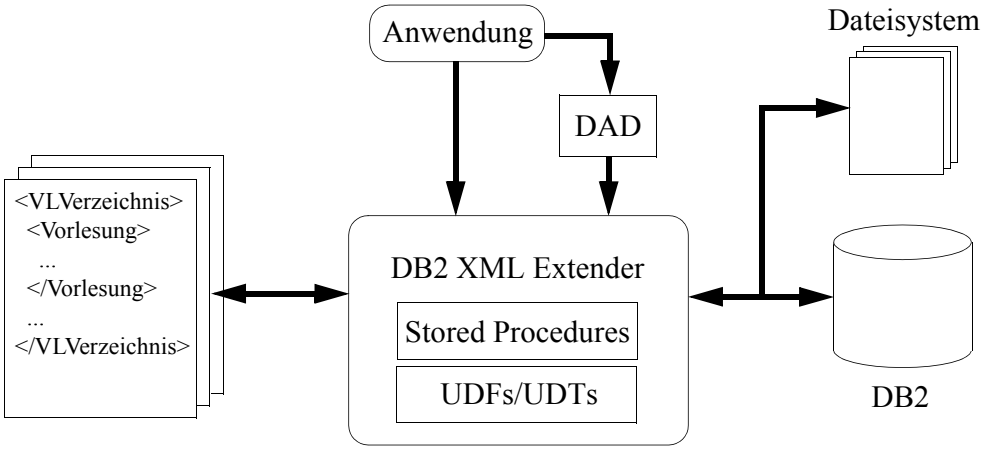
Vorlesung	Uhrzeit	Dozent
DBS2	9:15 Uhr	Prof. Rahm

XML-Unterstützung in kommerziellen DBS

- native XML-DBS
  - Datenzugriff erfolgt vorwiegend über XML-orientierte Schnittstellen (z.B. XPath, XSLT, DOM, XQuery)
  - DBS ist in erster Linie zur Speicherung und Manipulation von XML-Daten bestimmt
  - für alle XML-Dokumentarten geeignet (besonders für dokumentorientierte und gemischt strukturierte)
  - Beispiele: Tamino, eXtensible Information Server (XIS), Infonyte DB ...
- DBS mit XML-Erweiterung
  - DBS mit XML-unabhängigen Schnittstellen und Datenmodell (z.B. (objekt)relational, objektorientiert)
  - Datenzugriff vorrangig über Anfragesprache des DBS
  - enthält Erweiterungen zur Transformation zwischen XML-Dokumenten und internen Datenstrukturen (z.B. relationale DBS mit UDTs und UDFs zum Speichern und Abfragen von XML-Dokumenten)
  - vorrangig für datenorientierte XML-Dokumente optimiert
  - Bsp.: Oracle 9i, IBM DB2, Microsoft SQL-Server

IBM DB2 XML Extender

- XML-Erweiterung für DB2

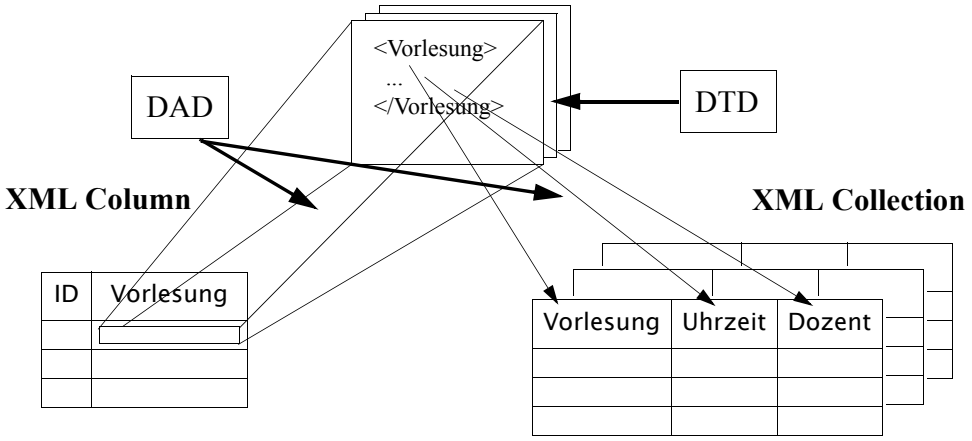


nach Rahm, Vossen (Hrsg.): Web & Datenbanken

- Mapping wird durch Anwender/Anwendung mittels einer *Data Access Definition* (DAD) festgelegt (für Import und Export)
- Verwaltung der XML-Daten im DBS oder im Dateisystem (*DataLink*)

DB2 XML Extender (2)

- XML Speicherungsformen
  - XML Column
    - Speicherung der XML-Dokumente als Ganzes
    - definiert Datentypen XMLCLOB, XMLVARCHAR, XMLFILE
  - XML Collection
    - anwendungsbasierte zerlegte relationale Speicherung



nach Rahm, Vossen (Hrsg.): Web & Datenbanken

DB2 XML Extender (3)

- XML Column
  - erlaubt eingeschränkte XPath-Anfragen über UDFs
  - Indexierung einzelner Elementinhalte über Anlegen von Zusatztabellen (Side Tables); definiert in DAD
  - Tabelle muss Attribut mit XML-UDT besitzen und aktiviert werden

```
CREATE TABLE vorlesungen(
      id INT NOT NULL PRIMARY KEY,
       vorlesung db2xml.XMLVARCHAR)
```

```
<DAD>
 <dtdid>Vorlesung.dtd</dtdid>
 <validation>YES</validation>
 <Xcolumn>
  <table name="dozent_sidetab">
   <column name="dozent"
            type="varchar(40)"
            path="/Vorlesung/Dozent/Name"
           multi_occurrence="YES"/>
  </table>
 </Xcolumn>
</DAD>
```

  - Tabelle muss aktiviert werden

```
dxxadm ENABLE_COLUMN vl_db vorlesungen vorlesung "vorlesung.dad" -r id
```

  - Einbringen von Dokumenten mittels UDF:

```
INSERT INTO vorlesungen(id, vorlesung) VALUES ('1', db2xml.XMLVarcharFromFile('vlverzeichnis.xml'
```

  - Auslesen über Extraktions-UDFs:

```
SELECT db2xml.extractVarchar(vorlesung, '/Vorlesung/Dozent/Name') FROM vorlesungen
```

    oder Side Tables:

```
SELECT dozent FROM dozent_sidetab
```

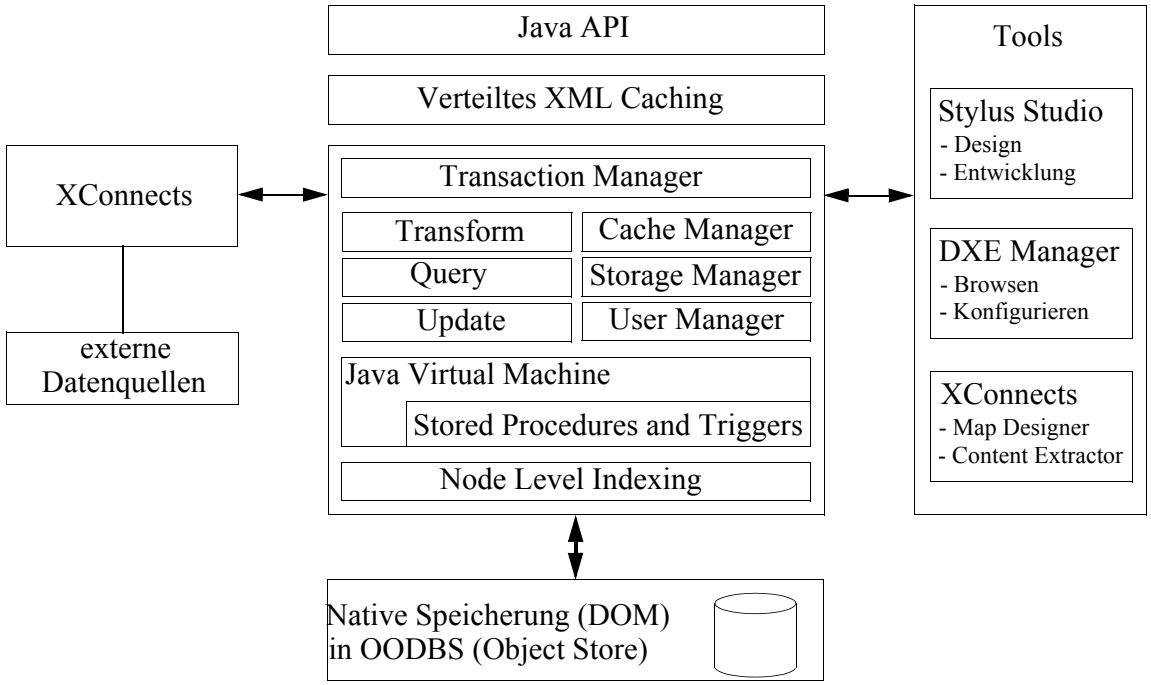

DB2 XML Extender (4)

- XML Collection
  - Speichert Elementinhalte und Attribute als Datenbankattribute
  - direkte Abbildung von Relationen auf XML-Strukturen
  - Stored Procedures für
    - Zerlegung von XML-Dokumenten in Relationen (dxxShredXML)
    - Generierung von XML-Dokumenten aus Relationen (dxxGenXML)
  - DAD enthält Abbildungsvorschrift
    - Komposition von XML-Dokumenten aus Relationen mit SQL_stmt
    - Komposition und Dekomposition mittels RDB_node

```
<?xml version="1.0" ?>
<!DOCTYPE DAD SYSTEM "/db2/dxx/dtd/dad.dtd">
<DAD>
 <validation>NO</validation>
 <Xcollection>
  <SQL_stmt>
   select vorlesung, dozent from vorlesungen
  </SQL_stmt>
  <root_node>
   <element_node name="Vorlesung">
    <attribute_node name="name">
     <column name="vorlesung"/>
    </attribute_node>
    <attribute_node name="dozent">
     <column name="dozent"/>
    </attribute_node>...
   </element_node>
  </root_node>
 </Xcollection>
</DAD>
```


eXtensible Information Server (XIS)

- Hersteller: eXcelon; jetzt: Progress Software



XIS (2)

- Speicherung ist schemaunabhängig
- Anfragesprachen
  - XPath 1.0
  - XQuery (noch nicht vollständig)
  - XSLT zur Transformation von Anfrageergebnissen
- Änderungen über spezielle Update-Sprache (an SQL und XPath angelehnt)

```
<update select="/Vorlesung/Dozent">
  <element location="insert">
    <Einrichtung>Uni Leipzig</Einrichtung>
  </element>
</update>
```

- direkter Zugriff über DOM
- unterstützt Struktur-, Text- und Werteindizes

Zusammenfassung

- XML: flexibles Format für strukturierte und semistrukturierte Daten
- XML Schema ermöglicht Typisierung von Elementinhalten und Attributwerten
- gängige XML-Prozessoren/APIs: SAX, DOM
- XML-Anfragesprachen: XPath, XQuery
- breites Anwendungsspektrum bedingt unterschiedliche Eigenschaften von XML-Dokumenten (datenorientiert, dokumentorientiert, gemischt strukturiert)
- XML stellt erhöhte Anforderungen an Datenbanksysteme
  - unterschiedliche Architekturen von XML-Datenbanken mit Fokussierung auf bestimmte Dokumenttypen (Speicherung als Ganzes, Dekomposition, hybride Speicherung)
  - relationale Datenbanksystemen benötigen Mapping der XML-Struktur in Tabellen des Datenbanksystems (unterschiedliche proprietäre Lösungen der Datenbankhersteller)
  - XML-Unterstützung in relationalen DBS dient derzeit hauptsächlich dem Austausch strukturierter Daten in XML (datenorientiert)

Vorschau SS2003

- Grundstudium: Datenbanksysteme 1 (2+1, Prof. Rahm)
- Hauptstudium
  - Implementierung von DBS 2 (Prof. Rahm)
  - Geoinformationssysteme 1 (Dr. Sosna)
  - Workflow-Management und E-Services (Dr. Müller)
- SHK für DBS1 gesucht !
  bitte bei Dr. R. Müller melden; *mueller@informatik.uni-leipzig.de*